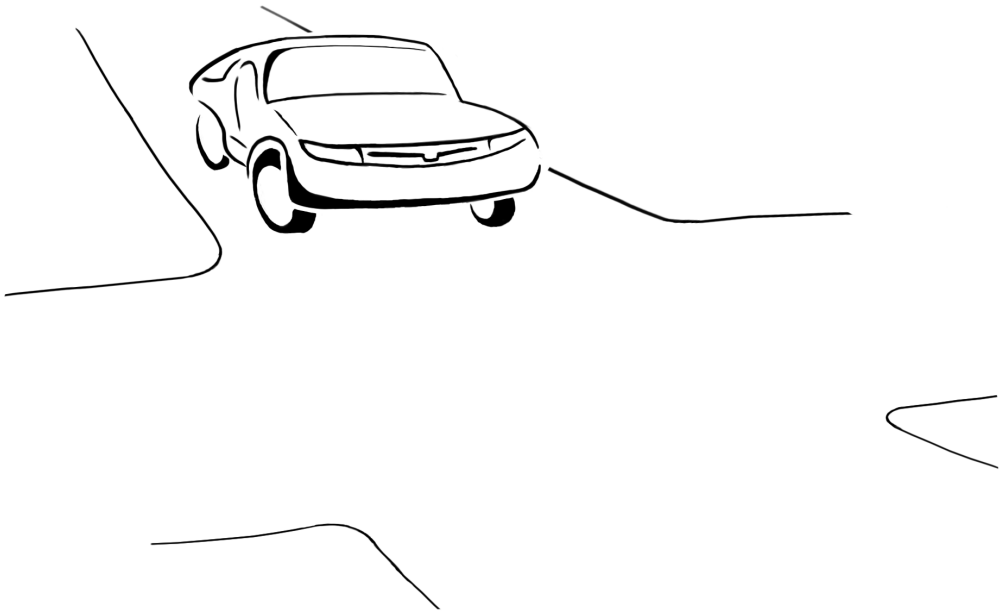
CHALMERS



Autonomous Driving in Crossings using Reinforcement Learning

Master's Thesis in Computer Science - Algorithms, Languages and Logic

ROBIN GRÖNBERG & ANTON JANSSON

Department of Electrical Engineering
Chalmers University of Technology
Gothenburg, Sweden 2017

Report No. EX096/2017

Master's Thesis 2017

Autonomous Driving in Crossings using Reinforcement Learning

Investigation of Action Value Based Reinforcement Learning for Autonomous Vehicle Decision Control in Partially Observable Environments

ROBIN GRÖNBERG
ANTON JANSSON

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering
Chalmers University of Technology
Gothenburg, Sweden 2017

Autonomous Driving in Crossings using Reinforcement Learning
Investigation of Action Value Based Reinforcement Learning for Autonomous Vehicle Decision Control in Partially Observable Environments

ROBIN GRÖNBERG
ANTON JANSSON

Technical report no. EX096/2017
Supervisor: Tommy Tram, Zenuity
Examiner: Jonas Sjöberg, Department of Electrical Engineering
Master's Thesis 2017
Department of Electrical Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Sweden
Telephone +46 31 772 1000

Cover: Stylized illustration of a car at a road crossing
Typeset in LaTeX
Gothenburg, Sweden 2017

Autonomous Driving in Crossings using Reinforcement Learning
Investigation of Action Value Based Reinforcement Learning for Autonomous Vehicle Decision Control in Partially Observable Environments

ROBIN GRÖNBERG
ANTON JANSSON
Department of Electrical Engineering
Chalmers University of Technology

Abstract

Machine learning techniques such as artificial neural networks have recently shown very promising results for decision control tasks when combined with reinforcement learning. This thesis presents an applicable approach using longitudinal acceleration control for autonomous vehicles driving through crossings in a simulated traffic environment. An Acceleration Regulator, which controls the autonomous vehicle, is trained using reinforcement learning and attempts to take advantage of gaps between cars, while following a pre-planned path along its lane. The results show that policies can be trained to successfully drive comfortably through a crossing, avoiding collision with other cars and being too passive. Comfortability is achieved by constraining jerk. The Acceleration Regulator generalize over different types of traffic crossings and driver behaviors. In 95% of all attempts, the learned policy is able to handle traffic situations with a varied number of cars having the same behavior on four types of crossings, involving several lanes and turns. When other cars vary between four different behaviors, driving in only one fixed crossing, the policy is successful in 97.2% of all attempts. Deep Q-learning is used to find a policy for the Acceleration Regulator, that can solve this task. This learning algorithm does not infer information over time. To enable this, a Deep Recurrent Q-Network is tested and compared to the Deep Q-learning approach. Results show that a Deep Recurrent Q-Network succeeds in three out of four attempts where a Deep Q-Network fails.

Keywords: autonomous driving, reinforcement learning, Deep Q-Network, neural network, machine learning

Acknowledgements

Both of us are grateful for all the support and help received during our work with this thesis. A special thanks goes to Zenuity for recognizing our skills and to give us the opportunity to work on this project. We thank Mohammad Ali for continuously approaching us in the afternoons, discussing how everything is coming together and providing potential ideas how to proceed with the project. We also thank Samuel Scheidegger for providing hardware used for training policies and running simulations. We specially thank our supervisor at Zenuity, Tommy Tram, for helping us all the way with ideas, thesis feedback and providing presentation sessions to ensure involvement of employees at Zenuity making our work relevant. We thank our superviser, Prof. Jonas Sjöberg, for providing helpful feedback on our thesis. We also thank our work colleagues for being part of interesting discussions and good company. At last, we thank our family and friends for their support.

Robin Grönberg and Anton Jansson, Gothenburg, May 23, 2017

Contents

List of Figures

Glossary

Acceleration Regulator The function constructed in this thesis, controlling the acceleration of the ego car. iii, 2–4, 6, 7, 30, 41, 48

action Actions are executed by agents, and can influence the environment. 1, 5, 7, 18–22, 30, 32, 33, 35–38, 41–43, 48, 50

action space The number of possible actions a system can have. 6, 18, 30, 48

adaptive cruise control A function extending the capabilities of cruise control by also adjusting the velocity to the leading car. xiv, 5

agent An agent assumes the role of a driver, by controlling the vehicle's acceleration directly. For non-autonomous cars, a human takes the role of the agent. 2–7, 18, 19, 21, 22, 24, 28, 29, 34–38, 42–44, 47–49

cruise control A function used in vehicles to automatically adapt the velocity of ego car to keep a certain speed. xii, xiv, 31

ego agent The agent that is learning a policy. The ego agent drives the ego car. 2–7, 29, 33, 35, 41, 44, 47, 48, 50

ego car The car that the learning agent drives. 2–6, 24, 33–35, 42, 47, 48, 50

ego lane The lane that the ego car is currently using. 3, 31, 33–35

episode A traffic scenario simulation ending when the agent either has passed the crossing, collides with another car or is too passive. 4, 6, 21, 24, 42, 43

episodic reward The sum of all rewards given during an episode. 41, 42

experience memory Saved data from previous simulations that an agent can be trained on. 21, 22

follow vehicle When an agent adjusts the velocity such that it will end up behind a target vehicle. 32, 33, 36, 37

give way When an agent adjusts the velocity such that other cars with conflicting planned trajectories have priority, and proceed before the ego car. 5, 28, 32, 33, 48

leadning car The car in front of the ego car. 5, 35

policy A set of rules an agent is using when making decisions. These rules could be clearly defined, or completely obscure and hard to break down. 3, 7, 8, 18–21, 29, 41–43, 47, 48

reward A feedback signal to the agent, which describe how well the agent is performing a task. 3, 7, 18–21, 30, 36, 41, 43, 48, 49

state An agent's representation of the environment, based on its observations. 16–22, 33, 41, 45, 47, 49

state space The number of possible states a system can have. 3, 6, 18

take way When an agent adjusts the velocity of its car such that it has priority over other cars with crossing planned trajectories. 5, 28, 32, 33, 43, 48

target car A car that is not the ego car, and is somehow of interest. 2–7, 24, 28, 29, 32–35, 37, 41–44, 47, 50

target lane The lane that a target car is currently using. 34, 35

Acronyms

ACC Adaptive Cruise Control. 5, 31, 32, *Glossary:* adaptive cruise control

CC Cruise Control. 31, 32, *Glossary:* cruise control

DQN Deep Q-Network. 5, 8, 18, 20–22, 30, 33, 36, 38, 40–42, 44, 45, 50, 56

DRQN Deep Recurrent Q-Network. 5, 7, 18, 22, 30, 36–39, 42–46, 49, 50, 56

LSTM Long Short-Term Memory. 8, 16, 17, 22, 23, 37, 39, 44–46

MDP Markov Decision Process. 17, 18, 20, 22, 30, 33, 37, 44

POMDP Partially Observable Markov Decision Process. 17, 18, 22, 30, 44

RNN Recurrent Neural Network. 8, 14–16

SGD Stochastic Gradient Descent. 12

STG Short Term Goal. 3–7, 28, 31–33, 35, 36, 48, 50

1

Introduction

This chapter will give a brief background to autonomous driving, why it is a hard problem to solve and why reinforcement learning could potentially be used to solve parts where previous state-of-the-art methods fail. The problem of autonomous driving in crossings is given together with an overview of the implemented approach. The approach is founded upon assumptions and limitations defined in the scope section. Thereafter, the contributions from this thesis are presented and the last section gives an outline of the chapters in this thesis.

1.1 Background

It is estimated that 10% of traffic deaths in the US were caused by lack of drivers' attention [1]. Today, there exist safety systems that can step in when the driver lacks the performance needed for a given situation, and assist in avoiding casualties [2]. In the future, fully autonomous vehicles could further improve safety and also efficiency. An experiment shows that a traffic crossing with only autonomous vehicles have the potential to improve traffic flow, by making decisions and communicating faster than humans while ignoring lanes and traffic signals without compromising safety [3]. In order to achieve this in today's traffic, an autonomous vehicle must be able to interpret the intentions of both autonomous and human drivers.

The idea of autonomous vehicles has been around since at least 1939 [4], indicating that it is a hard problem to solve. A possible explanation could be that it is difficult to program a rule-based algorithm that can determine the correct action in all given situations. This is shown in DARPA's Urban Challenge, held in 2007, in which autonomous vehicles attempted to drive in an isolated urban traffic environment. During the challenge, almost half of all vehicles were removed from the race because of incorrect decision-making [5]. When solving problems where rule-based algorithms struggle, machine learning techniques have in some cases shown great potential compared to the

previous state-of-the-art methods [6, 7, 8, 9]. One such machine learning technique is reinforcement learning, where an agent learns what to do by trial and error. Typically, machine learning algorithms require large sets of training data. Reinforcement learning can utilize both existing training data and experiences generated from the agent's actions, which makes it flexible and cost efficient. Reinforcement learning algorithms using artificial neural networks have shown promising results [7, 8, 10, 11, 12]. Two such reinforcement learning algorithms are Deep Q-Network and Deep Recurrent Q-Network, which are used in this thesis to implement decision-making for autonomous vehicles.

1.2 Autonomous Driving in Crossings

In this thesis, a function controlling the longitudinal acceleration of an autonomous car is constructed. This function is referred to as Acceleration Regulator. The Acceleration Regulator is used to drive the autonomous car through a crossing by finding gaps between surrounding cars, referred to as target cars. The controlled car is referred to as the ego car, and an ego agent drives the ego car by using the Acceleration Regulator. The ego car follows a pre-planned path along its own lane, approaching the crossing, and the ego agent makes decisions on how to drive through the crossing without colliding. The ego agent does not consider any traffic rules when making decisions, but instead it observes the movements of the target cars. The target cars are also driven by agents, which can have different behaviors, as seen in Figure 1.1. Some of these target car agents slow down before the crossing, providing more time for the ego agent to pass, while others do not. The ego agent can differentiate between target car agents to make different decisions by observing their behavior over time. This enables the ego agent to collaborate with the target cars. The ego agent can collaborate with four different agents, described in Section 3.5 and is able to drive in four different crossings, each modelled by a traffic scenario, presented in Figure 1.2, without explicit knowledge of which of the crossing it drives in.

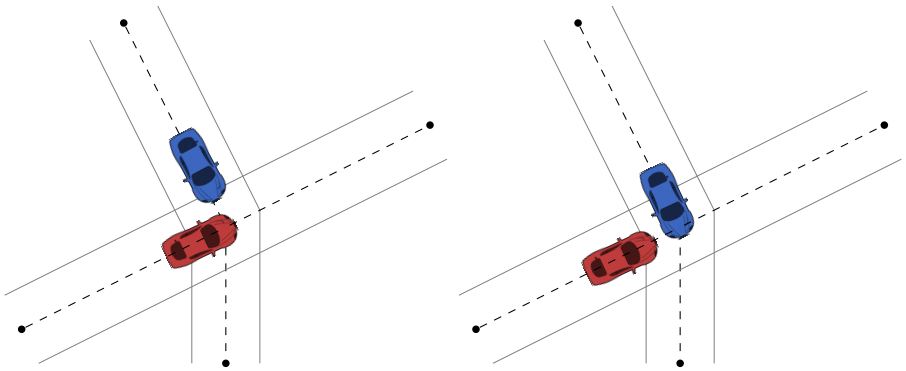


Figure 1.1: The figure illustrates different driver behaviors of a target car's agent. The ego car is red, and the target car is blue. Left: The target car's agent is passive, and gives way to the ego car. Therefore, the ego agent decides to drive. Right: The target car is aggressive and takes way. Therefore, the ego agent decides to stop.

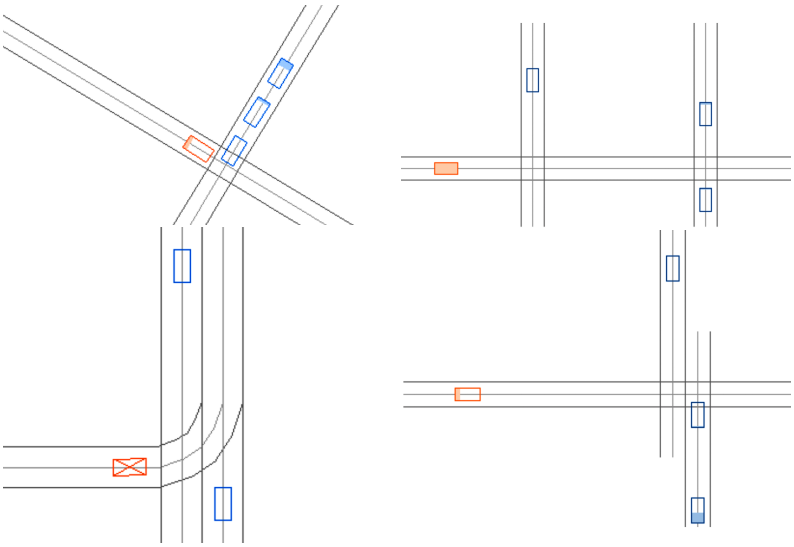


Figure 1.2: Four traffic scenarios that the ego agent is able to drive in. The ego car is represented by a red box while target cars are represented with blue boxes. Top left: a simple crossing. Top and bottom right: two crossings with two crossing lanes, which forces the ego agent to plan its trajectory to make sure it can stop for the second crossing lane if needed. Bottom left: a scenario where the ego lane turns left in the crossing and merges onto a target lane. The turn implies that a lower speed must be held by the ego car. Since the ego lane is merging onto another lane, the ego car must adjust to the traffic flow in that lane. It must also drive across the first lane partly against the traffic direction.

1.3 Approach

The Acceleration Regulator is divided into two separate parts: a high-level controller and a low-level controller, as seen in Figure 1.3. The high-level controller makes high-level decisions such as ”Take Way” or ”Give Way”, referred to as Short Term Goals (STGs). The low-level controller regulates the acceleration of the ego car based on the selected STG. A policy is a function that defines how an agent drives. The ego agents's policy is defined by the Acceleration Regulator.

The high-level controller is constructed using reinforcement learning, by iteratively training the ego agent to drive comfortably through crossings using a traffic simulator. During training, the ego agent's policy is improved by receiving a feedback signal from a reward function. The defined reward function gives positive feedback to the ego agent when it has successfully driven through the crossing and negative when the ego car collides or is too passive. The ego agent explores the traffic environment by using ϵ-greedy, which sometimes chooses a random STG, to search the available state space. Reinforcement learning is described in more detail in Section 2.2. To maintain comfortability, low jerk is desired [13], which is achieved by limiting jerk in the traffic simulator and reducing the reward when jerk is high, as seen in Section 3.4 and 4.5.

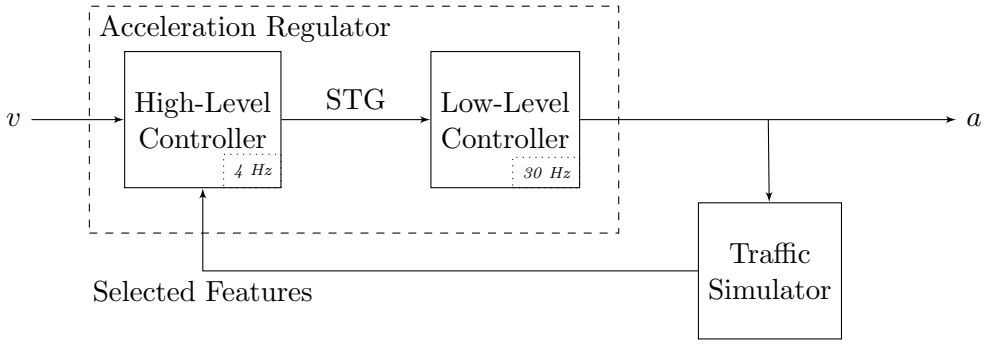


Figure 1.3: The Acceleration Regulator interacts with the traffic simulator by controlling the acceleration a of the ego car. The high-level controller chooses an STG, 4 times every second, based on the selected features that describes the traffic environment. The selected features contains speeds and distances to reference points in the crossing for the ego car and all target cars, described in Section 1.4.4. The low-level controller computes the acceleration based on the selected STG and is executed 30 times per second. The reference signal v determines the ego car's desired velocity and is set to its maximum speed.

1.4 Implementation Choices

This section presents more specifically what a traffic scenario is. Thereafter, why the traffic is a multi-agent environment, and what algorithm is chosen because of that, is described. The different STGs are presented and explained. The last section presents how the features that the agent uses to make decisions are defined using a coordinate system implemented by the traffic simulator.

1.4.1 Traffic Scenarios

There are four traffic scenarios, shown in Figure 1.2, which are used to construct the training data. Each traffic scenario is defined by configurations of lanes and target cars, and is simulated over the course of one episode. The training data contains the results from many simulated episodes. In a traffic scenario, the ego car, and up to four target cars, are placed onto lanes with initial values for position and speed. To achieve variation in the training data, the initial values are sampled at random within a specified interval defined by the traffic scenario, and are re-sampled each time a new episode starts. The ego agent is able to drive in the four types of crossings, after many episodes of training.

1.4.2 Multi-Agent Traffic Environment

Traffic is a multi-agent environment, meaning that not only the ego agent's decisions influence the environment, but also decisions taken by target cars. Therefore, the Markov property assumption described in Section 2.2.1 does not hold for traffic environments [14]. To solve this under the Markov property assumption, the target cars' behaviors must be part of the underlying system state, which makes the environment partially observable. Therefore, two reinforcement learning algorithms, described in Chapter 2.2, are

implemented and compared: Deep Q-Network (DQN), and Deep Recurrent Q-Network (DRQN), which is an extension to DQN that supports partially observable environments. Motivation for the choice of DQN and DRQN is described in Section 4.1. Results show that by using a DRQN, the ego agent can better respond to different behaviors of target cars, compared to a DQN with one observation. The DRQN succeeds in three out of four attempts where the DQN fails, as shown in Section 5.5.1. However, when using several observations in a DQN, the performance is similar between the two.

1.4.3 Short Term Goals as Actions

The high-level controller chooses between six discrete actions, defined by STGs. The STGs are high-level choices passed to the low-level controller, which computes an acceleration based on regulators typically used in autonomous driving today. The low-level controller makes sure that each action slows down in curved lanes, and adjusts to leadning cars using Adaptive Cruise Control (ACC). The actions allow the agent to give way or take way, which are two options that human drivers typically face when approaching a crossing. The actions also allow the agent to follow behind a specific target car, enabling proper timing when passing crossings and merging lanes. Further details about how STGs are used as actions is presented in Section 4.3, while details about how the low-level controller works is described in Section 4.2.2.

1.4.4 Selection of Features Used for Decision Making

The selected features consists of 8 *target features* for each target car and 7 *ego features*, only related the ego car. These are high-level features that are computed from a one-dimensional coordinate system representing the lanes along their longitudinal axes. This coordinate system makes abstractions, hiding some irrelevant dissimilarities between different lanes and types of crossings, which can make generalization among them easier. For example, merging lanes and roundabouts can be modelled the same way as crossings. The coordinate system is described in Section 3.2.

The selected features are computed using three reference points on the ego car's and each target car's crossing lanes, shown in Figure 1.4. The features include the ego and target car's speeds and relative positions to the three reference points. The list of all features is presented in Section 4.4.

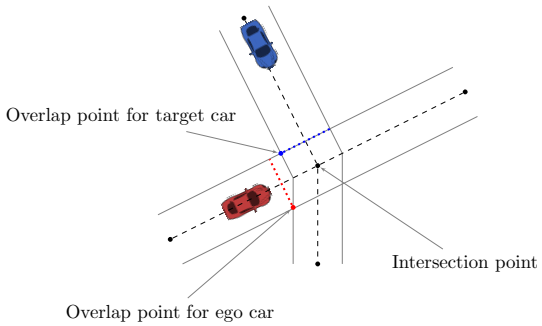


Figure 1.4: An illustration of the intersection point and overlap points for the ego car and target car. The ego car is illustrated by a red car, while target car is blue.

The focus in this thesis is not to evaluate the importance of the selected features, which is why the selected high-level features were chosen as they provide enough information for the ego agent to drive in the four defined traffic scenarios.

1.5 Scope

The traffic simulator, described in Chapter 3, models vehicles and no other road users. It is assumed that these vehicles use a set of ideal sensors that capture the information needed to compute the features described in Section 4.4. Even if ideal sensors are used, the environment could totally obscure other vehicles, for instance by large buildings close to the crossing. Such traffic scenarios are not considered.

Only the longitudinal position along the road will be considered. In practice, lateral position could be considered equally important. Each car is always positioned in the center of its lane and is angled in the lane's direction. By disregarding lateral positions, the problem is simplified, which reduces the state- and action spaces.

The ego agent is not required to handle emergency situations or other unpredictable occurrences. However, the ego agent should not be careless while driving. A recent study shows that reinforcement learning algorithms generally have problems avoiding casualties [15]. In order to guarantee total safety, reinforcement learning is insufficient. Requirements for guaranteed safety are out of scope for this research, and are assumed to be handled on another level with methods that can guarantee safety.

The number of target cars is assumed to be constant throughout a whole episode. This means that no cars enters or leaves the traffic environment during an episode. The agent also assumes that there are no more than 4 target cars in the traffic environment.

1.6 Contributions

A method for constructing the Acceleration Regulator that plan the ego car's longitudinal acceleration in crossings is implemented. This solution uses a high-level controller, which utilize Deep Q-Learning and Deep Recurrent Q-Learning. The high-level controller uses features such as speeds and distances to the intersection for each vehicle to choose an STGs. The low-level controller uses the chosen STG to compute the desired acceleration. This thesis presents the following contributions:

- The ego agent can drive in all four scenarios in Figure 1.2, without including the type of crossing the ego car drives in as part of the high-level features described in Section 4.4.
- The ego agent can make decisions based on target cars' behaviors rather than following traffic rules, as seen in Section 5.5.
- The agents choose between different STGs using a high-level controller, rather than controlling the acceleration directly. Details about these STGs are found in Section 4.2.2.

- The learning process converge faster by introducing an artificial neural network structure where some network parameters are shared between target cars. More details about the neural network structures are presented in Section 4.6.
- A stochastic sequence length used when training the DRQN is compared to a fixed sequence length. However, results indicate similar performance between the two. Stochastic sequence length is described in Section 4.6.4.

The results for the Acceleration Regulator are evaluated by measuring the success rate, which is how often the ego agent is able to drive to the other side of the crossing without colliding or being too cautious. For DRQN, the success rate is 95% when driving in different crossings, and 97.2% when target cars have different behaviors.

1.7 Thesis Outline

- **Chapter 2. Machine Learning Background** explains machine learning background needed to understand theory and concepts used throughout this thesis. In particular, artificial neural networks and reinforcement learning algorithms are described.
- **Chapter 3. Traffic Simulator for Crossings** presents the traffic simulator, the lane coordinate system, the car mechanics and the different agents.
- **Chapter 4. Acceleration Regulator** explains how the Acceleration Regulator is implemented and discusses the choice of reinforcement learning algorithm. It presents the selected features for desicion making, the actions chosen by the high-level controller, how the low-level controller is implemented for all STGs, and the reward function. It also proposes different neural network structures that are evaluated in the Result chapter.
- **Chapter 5. Result** presents graphs for trained policies for different simulation setups, including the four crossings in Figure 1.2 and comparison between shared and unshared weights.
- **Chapter 6. Discussion** discusses the results and implications of made choices. Additions to the proposed method is briefed about, as well as other possible approaches to solve the problem.
- **Chapter 7. Conclusion** concludes the result and the contribution made in this thesis.

2

Machine Learning Background

In the first section of this chapter, the basics of artificial neural networks will be explained, as well as how to train them. This section covers three particular structures of artificial neural networks: feed forward neural networks, Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks. The neural networks are used together with reinforcement learning algorithms, which are also presented in this chapter. The basics of reinforcement learning will be presented, together with an explanation of Q-learning. The DQN algorithm will be explained, as well as potential inconsistency issues, and how to reduce potential fluctuations in the policy due to these issues. How to support reinforcement learning for partially observable environments is discussed at the end of this chapter.

2.1 Artificial Neural Networks

Biological neural networks enables translation of raw information from signals, such as human senses, into performing complex tasks [16]. These networks consists of neurons connected to each other. As early as 1943, a model representing a biological neuron was proposed, referred to as an *artificial neuron* [17]. The following sections describe the basics of artificial neurons and how to connect them together in layers to form a *feed forward network*, which is an efficient model to solve statistical pattern recognition tasks. Furthermore, gradient descent and backpropagation are introduced, which are used to train feed forward networks. How initial weights are properly selected, for stable training sessions, is presented. Two alternative neural network structures are presented, recurrent networks and LSTMs, which extend feed forward networks with support for a sequence of inputs. At last, overfitting is explained along with how to reduce its effect with dropout.

2.1.1 Artificial Neuron

An artificial neuron is defined to return a value ζ for a given input vector $\boldsymbol{\xi} = (\xi_1, ..., \xi_p)^T$ of size p. The return value is computed as a weighted sum using a weight vector $\boldsymbol{w} = (w_1, ..., w_p)^T$ and a bias value b. The weights define the connection strength between an input signal ξ and the artificial neuron. The bias defines how strong output signal ζ the artificial neuron will transmit to other artificial neurons. The output signal is defined by the following function [18]:

$$\zeta = \sum_{i=1}^{p} w_i \xi_i + b \tag{2.1}$$

The equation can be presented as a graph, as seen in Figure 2.1.

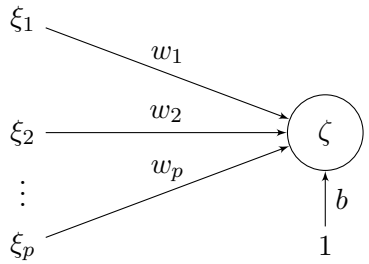


Figure 2.1: The image describes the artificial neuron and how the inputs $\boldsymbol{\xi}$ are used to compute the output ζ. Neurons are represented by circles. The value at the start of each arrow is multiplied with the value presented beside the arrow, and then summed together at the end, together with other arrows pointing towards the same neuron. The arrows represent connections between the input $\boldsymbol{\xi}$ and output ζ with its connection strength presented by its weight $\boldsymbol{w}$. The bias connection is sometimes omitted when presenting artificial neural networks using this notation.

2.1.2 Feed Forward Networks

A feed forward network consists of several layers of neurons, where each layer's neurons are connected to every neuron in the next layer. Such connection between layers are referred to as *fully connected* layers. This network of neurons can be used for function approximation. The goal with a neural network is to find parameters θ such that the network with output $\boldsymbol{o} = f(\boldsymbol{\xi} \,|\, \theta)$ approximates $\boldsymbol{o} \approx \boldsymbol{\zeta}$ as close as possible, where $\boldsymbol{\zeta}$ is the target value to approximate for input $\boldsymbol{\xi}$. A network's parameters θ include all weights and biases for the neurons in the network [18].

Hidden layers are the layers between the input layer $\boldsymbol{\xi}$ and the output layer $\boldsymbol{o}$. They are required to approximate non-linear functions. Each layer becomes a function for which its result is passed as input into the next layer. The first hidden layer consists of neurons taking $\boldsymbol{\xi}$ as input. A network with two hidden layers can be represented by three chained functions, one for each hidden layer $f^{(1)}, f^{(2)}$ and one for the output layer $f^{(o)}$ [18]:

$$\boldsymbol{o} = f(\boldsymbol{\xi} \mid \theta) = f^{(o)}\left(f^{(2)}\left(f^{(1)}(\boldsymbol{\xi} \mid \theta) \mid \theta\right) \mid \theta\right)$$

When a network has several layers, an activation function g is used to compute the hidden values between the layers. An activation function must be monotonic and differentiable. When using activation functions and at least one hidden layer, a neural network can approximate any non-linear function [19]. The output of a neuron in the first hidden layer $\boldsymbol{h}^{(1)} = (h_1^{(1)}, \ldots, h_n^{(1)})^T$ is computed as in Equation 2.2. The superscript of $\boldsymbol{h}^{(l)}$ denotes the layer.

$$h_j^{(1)} = g\left(\sum_{i=1}^{p} w_{ji}^{(1)} \xi_i + b_j^{(1)}\right) \tag{2.2}$$

The weight values $w_{ji}^{(1)}$ becomes a matrix $\boldsymbol{W}^{(1)}$ weighting each component of $\boldsymbol{\xi}$ to each component of $\boldsymbol{h}^{(1)}$, and bias values $b_j^{(1)}$ becomes the vector $\boldsymbol{b}^{(1)}$ representing the threshold for each neuron in the hidden layer $\boldsymbol{h}^{(1)}$:

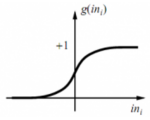
$$\boldsymbol{W^{(1)}} = \begin{bmatrix} w_{11}^{(1)} & w_{12}^{(1)} & \ldots & w_{1p}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} & \ldots & w_{2p}^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1}^{(1)} & w_{n2}^{(1)} & \ldots & w_{np}^{(1)} \end{bmatrix} \qquad \boldsymbol{b}^{(1)} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \\ \vdots \\ b_n^{(1)} \end{bmatrix}$$

Using matrix notation, Equation 2.2 is equivalent to:

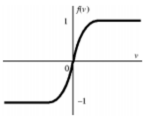
$$\boldsymbol{h}^{(1)} = g\left(\boldsymbol{W}^{(1)}\boldsymbol{\xi} + \boldsymbol{b}^{(1)}\right)$$

Two examples of activation functions are the sigmoid function that transforms x into a value between 0 and 1, or tanh returning a value between -1 and 1.

sigmoid: $g(x) = \frac{1}{1+e^{-x}}$: $\mathbb{R} \to (0,1)$



tanh: $g(x) = \tanh(x)$: $\mathbb{R} \to (-1,1)$



The output vector $\boldsymbol{o} = (o_1, ..., o_n)^T$ is computed using the values from the last hidden layer $\boldsymbol{h}^{(d-1)} = (h_1^{(d-1)}, ..., h_q^{(d-1)})^T$, where d is the depth of the neural network:

$$o_k = g\left(\sum_{j=1}^{q} w_{kj}^{(d)} h_j^{(d-1)} + b_k^{(d)}\right)$$
$$\boldsymbol{o} = g\left(\boldsymbol{W}^{(d)}\boldsymbol{h}^{(d-1)} + \boldsymbol{b}^{(d)}\right)$$

A neural network with two hidden layers can be expressed as in Equation 2.3 and 2.4. The same neural network is also presented in Figure 2.2 as a graph.

$$o_k = g^{(o)}\left(\sum_{j'=1}^{j} w_{kj'}^{(o)} g^{(2)}\left(\sum_{i'=1}^{i} w_{j'i'}^{(2)} g^{(1)}\left(\sum_{p'=1}^{p} w_{i'p'}^{(1)}\xi_{p'} + b_{i'}^{(1)}\right) + b_{j'}^{(2)}\right) + b_k^{(o)}\right) \tag{2.3}$$
$$\boldsymbol{o} = g^{(o)}\left(\boldsymbol{W}^{(o)} g^{(2)}\left(\boldsymbol{W}^{(2)} g^{(1)}\left(\boldsymbol{W}^{(1)}\boldsymbol{\xi} + \boldsymbol{b}^{(1)}\right) + \boldsymbol{b}^{(2)}\right) + \boldsymbol{b}^{(o)}\right) \tag{2.4}$$

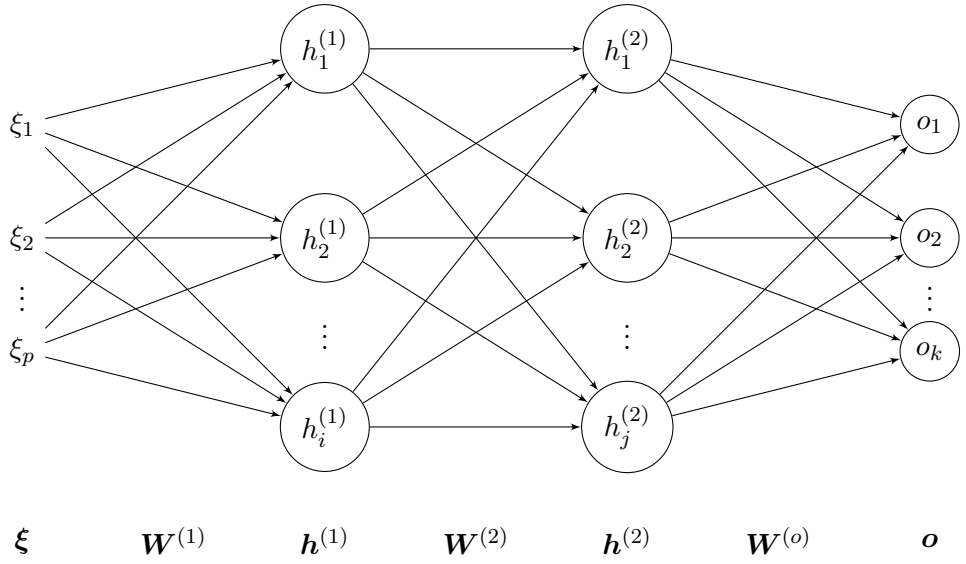


Figure 2.2: A neural network representing a function $f(\boldsymbol{\xi}|\theta) = \boldsymbol{o}$, with two hidden layers; $\boldsymbol{h}^{(1)}$, with i hidden neurons, and $\boldsymbol{h}^{(2)}$, with j hidden neurons, where $f : \mathbb{R}^p \to \mathbb{R}^k$ and $\theta = \{(\boldsymbol{W}^{(1)}, \boldsymbol{b}^{(1)}), (\boldsymbol{W}^{(2)}, \boldsymbol{b}^{(2)}), (\boldsymbol{W}^{(o)}, \boldsymbol{b}^{(o)})\}$. Biases are omitted in this figure.

2.1.3 Optmizing Neural Networks

The neural network parameters θ are learned by minimizing a loss function $L(\theta)$. The loss function give a value for how large the difference is between the network's approximation $\boldsymbol{o}^{(\mu)}$ and the target value $\boldsymbol{\zeta}^{(\mu)}$ it approximates, where μ identifies a training instance. Sum of squared errors, one of many loss functions, is used. With N number of instances to approximate, where the superscript denotes the instance, the loss function is defined by [18]:

$$L(\theta) = \frac{1}{2N} \sum_{\mu=1}^{N} ||\boldsymbol{o}^{(\mu)} - \boldsymbol{\zeta}^{(\mu)}||^2 \tag{2.5}$$

$$\text{where } \boldsymbol{o}^{(\mu)} = f(\boldsymbol{\xi}^{(\mu)}|\theta)$$

2.1.4 Gradient Descent

The network parameters θ that minimize $L(\theta)$ are found by computing the gradients of L with respect to θ, $\nabla_\theta L(\theta)$, for all instances in the training set. The weights and biases are iteratively updated in the direction of the negative gradient, with a small learning rate $0 < \eta \ll 1$ [20]:

$$\theta_t = \theta_{t-1} - \eta \nabla_{\theta_{t-1}} L(\theta_{t-1}) \tag{2.6}$$

where t is the training iteration. Batch gradient descent, as the method is called, is proven to converge to the global minimum only on convex problems, which neural networks typically are not [18]. Calculating the gradients using batch gradient descent can also be slow for large datasets. Many training instances can be similar, which makes some gradient computations in every update redundant. Instead, the gradients can be calculated for a sampled set of training instances, using Stochastic Gradient Descent (SGD). Let the sample size of training instances be $B \in [1,N]$. The update formula for SGD is then given by [18]:

$$\theta_t = \theta_{t-1} - \eta \nabla_{\theta_{t-1}} L\left(\theta_{t-1} \mid \left\{(\boldsymbol{\xi}^{(1)}, \boldsymbol{\zeta}^{(1)}), ..., (\boldsymbol{\xi}^{(B)}, \boldsymbol{\zeta}^{(B)})\right\}\right)$$

SGD updates have a higher variance than the batch gradient descent update. This means that $L(\theta)$ can fluctuate, which increases the chance of not getting stuck in a poor local minimum. However, the global minimum will also be more difficult to converge to. By decreasing the learning rate over time the variance can be reduced [18].

2.1.5 Backpropagation

Backpropagation is an algorithm for computing derivatives for $L(\theta)$ with respect to θ. With these derivatives, an optimization method such as gradient descent is used to update the network's weights and biases to minimize $L(\theta)$. The gradients are partial derivatives for $L(\theta)$ with respect to each weight and bias parameter. Equation 2.6 from gradient descent can be applied for a single weight parameter $w_{ji}^{(l)}$, which creates the update rule for that weight [21]:

$$w_{ji,t}^{(l)} = w_{ji,t-1}^{(l)} - \eta \frac{\partial L}{\partial w_{ji,t-1}^{(l)}}$$

The partial derivative of $w_{ji}^{(l)}$ in layer l is defined using the chain rule as:

$$\frac{\partial L}{\partial w_{ji}^{(l)}} = \frac{\partial L}{\partial h_j^{(l)}} \frac{\partial h_j^{(l)}}{\partial w_{ji}^{(l)}}$$

Deriving the full gradient for all weights and biases in all layers can simply be done using the chain rule. Therefore, the loss function and network function to approximate must be differentiable. Backpropagation assumes that the loss function can be computed for each training instance separately and averaged over the mini-batch. Backpropagation is presented in algorithm 1 [22, pp. 241-245].

Data: A minibatch $\hat{B} = \{(\boldsymbol{\xi}^{(1)}, \boldsymbol{\zeta}^{(1)}), \ldots, (\boldsymbol{\xi}^{(B)}, \boldsymbol{\zeta}^{(B)})\}$ and network parameters $\theta_{t-1} = \{(\boldsymbol{W}_{t-1}^{(1)}, \boldsymbol{b}_{t-1}^{(1)}), \ldots, (\boldsymbol{W}_{t-1}^{(m)}, \boldsymbol{b}_{t-1}^{(m)})\}$
Result: Updated network parameters θ_t
Compute output $\boldsymbol{o}^{(\mu)}$ from $\boldsymbol{\xi}^{(\mu)}$ for all instances $\mu \in [1,B]$
for *layer l in all m layers* **do**
  Compute $\frac{\partial L}{\partial w_{ji,t-1}^{(l)}}$ and $\frac{\partial L}{\partial b_{j,t-1}^{(l)}}$ using target values $\boldsymbol{\zeta}^{(\mu)}$ and output values $\boldsymbol{o}^{(\mu)}$
end
for *layer l in all m layers* **do**
  Update all weights and biases in layer l, element-wise:
  $$w_{ji,t}^{(l)} = w_{ji,t-1}^{(l)} - \eta \frac{\partial L}{\partial w_{ji,t-1}^{(l)}}$$
  $$b_{j,t}^{(l)} = b_{j,t-1}^{(l)} - \eta \frac{\partial L}{\partial b_{j,t-1}^{(l)}}$$
end
$\theta_t = \{(\boldsymbol{W}_t^{(1)}, \boldsymbol{b}_t^{(1)}), \ldots, (\boldsymbol{W}_t^{(m)}, \boldsymbol{b}_t^{(m)})\}$

Algorithm 1: Backpropagation

2.1.6 Weight Initialization

Since the optimization problem is non-linear with respect to the network parameters θ, there can be several local minima. The initial values for θ can determine what local minimum the algorithm finds and how quickly it converges [18]. By trying different initial parameter values for θ, the chance of finding a good local minimum increases. Also, too large parameter values can cause the optimization problem to be numerically ill-conditioned. This can be solved by for instance using regularization techniques [23], or by generating an initial value from a uniform distribution $U[-a, a]$ where the weights are scaled by each neuron's number of input connections n_{in}, called *fan-in size* [24]:

$$w_{ji} \sim U\left[-\frac{1}{\sqrt{n_{\text{in}}}}, \frac{1}{\sqrt{n_{\text{in}}}}\right]$$

2.1.7 Recurrent Networks

RNNs are used when the output is computed from a sequence of inputs. In contrast to a feed forward network, neuron connections in an RNN can form loops that allow the network to persist information between inputs in the input sequence. In other words, the output is not only determined by the fed input, but also from previous inputs [25].

An RNN with input sequence $(\boldsymbol{\xi}_1, ..., \boldsymbol{\xi}_t)$ have an output sequence $(\boldsymbol{o}_1, ..., \boldsymbol{o}_t)$. Let $\boldsymbol{h}_t$ be a layer of neurons that is connected to a previous layer in the network, creating the feedback loop. In a simple example of a recurrent network, the previously computed values $\boldsymbol{h}_{t-1}$ can be combined with the current input $\boldsymbol{\xi}_t$ and used to compute $\boldsymbol{o}_t$. Let $\boldsymbol{z}_t$ be the concatenation of $\boldsymbol{\xi}_t$ and $\boldsymbol{h}_{t-1}$ of size $j = ||\boldsymbol{\xi}_t|| + ||\boldsymbol{h}_{t-1}||$, $i = ||\boldsymbol{h}_t||$, and $g^{(h)}, g^{(o)}$ arbitrary activation functions. Let $\boldsymbol{W}^{(h)}$ and $\boldsymbol{b}^{(h)}$ be the weights and biases between $\boldsymbol{z}_t$ and $\boldsymbol{h}_t$ for any t. Likewise, let $\boldsymbol{W}^{(o)}$ and $\boldsymbol{b}^{(o)}$ be the weights and biases between $\boldsymbol{h}_t$ and the output $\boldsymbol{o}_t$. Then the network will be defined as [26]:

$$\boldsymbol{h}_t = g^{(h)}\left(\boldsymbol{W}^{(h)}\boldsymbol{z}_t + \boldsymbol{b}^{(h)}\right)$$
$$\boldsymbol{o}_t = g^{(o)}\left(\boldsymbol{W}^{(o)}\boldsymbol{h}_t + \boldsymbol{b}^{(o)}\right)$$

The feedback layer $\boldsymbol{h}_t$ can be connected to any previous layer. The recurrent network example is presented as a graph in Figure 2.3.

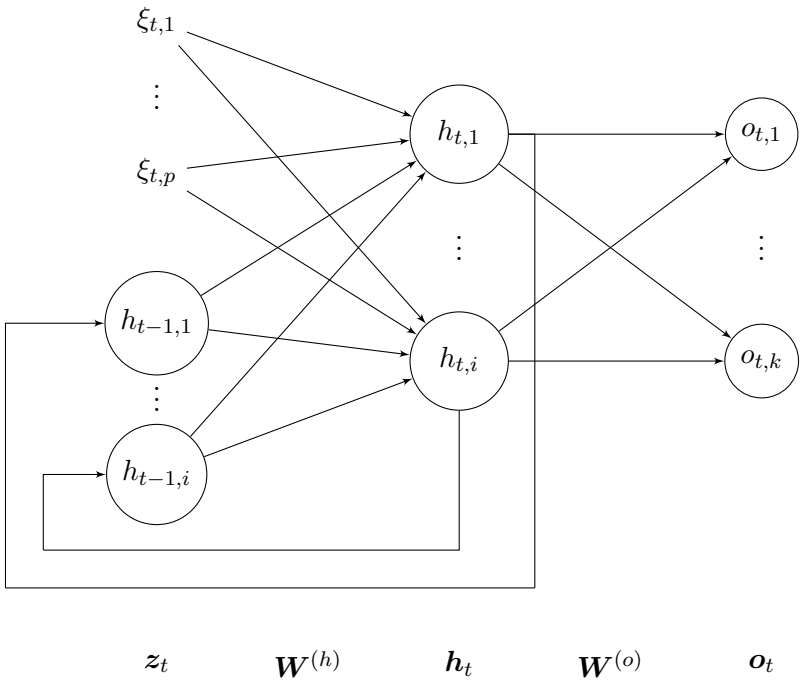


Figure 2.3: A RNN with input $\boldsymbol{\xi}_t = (\xi_{t,1}, \ldots, \xi_{t,p})^T$, one hidden layer $\boldsymbol{h}_t = (h_{t,1}, \ldots, h_{t,i})^T$ and the output layer $\boldsymbol{o}_t = (o_{t,1}, \ldots, o_{t,k})^T$ where every neuron in the hidden layer is recurrent.

When running backpropagation on a recurrent network, the network needs to be unrolled a fixed number of steps in order to calculate the derivatives. Unrolling can be seen as copying the part of the network that exists within the loop, and placing the copies in a sequence, each connected to the next one. The last unrolled step, $\boldsymbol{h}_1 = (h_{1,1} \dots h_{1,i})^T$, will contain some inputs with no values. To solve this, these values can be initialized with a default value of 0. An unrolled RNN over three time steps is presented in Figure 2.4. Calculating derivatives for backpropagation is then no different than for a feed forward network [25].

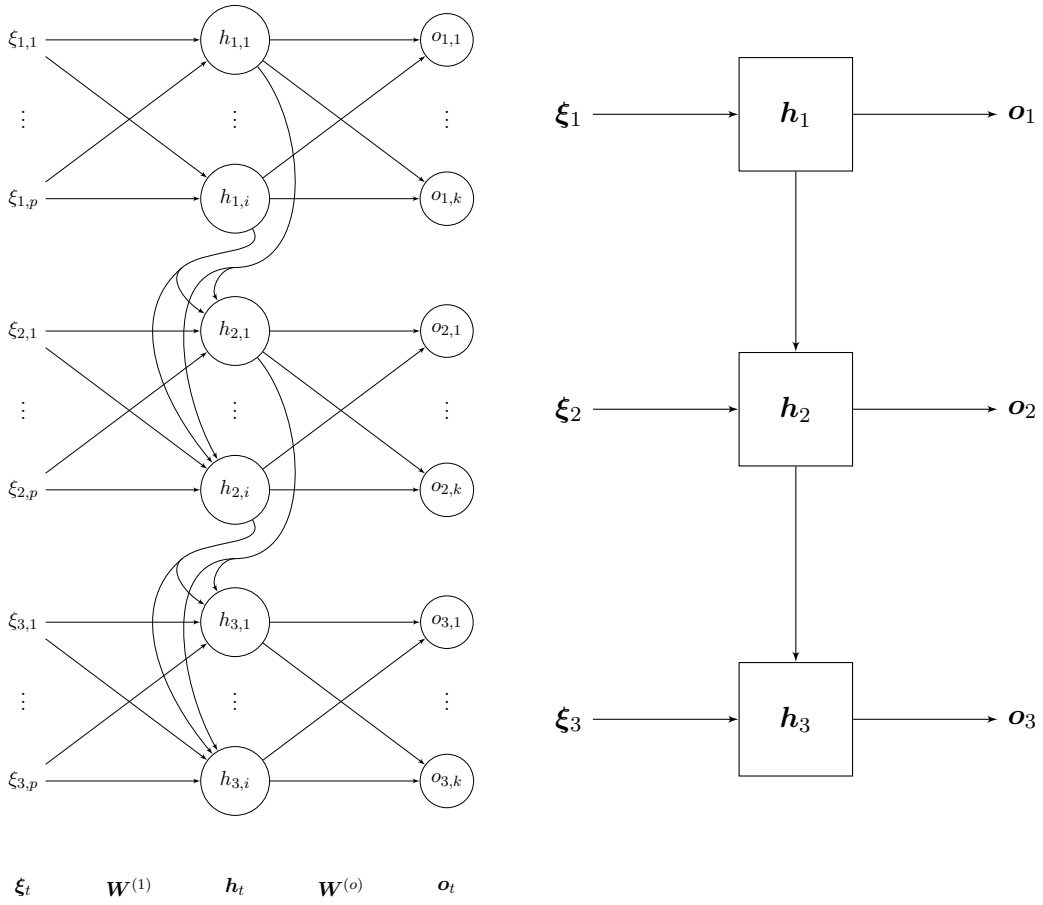


Figure 2.4: To the left; an unfolded RNN, where every neuron in the hidden layer is recurrent. The network is unfolded three time steps, with input $\boldsymbol{\xi}_t = (\xi_{t,1}, \dots, \xi_{t,p})$, one hidden layer $\boldsymbol{h}_t = (h_{t,1}, \dots, h_{t,i})$ and the output layer $\boldsymbol{o}_t = (o_{t,1}, \dots, o_{t,k})$, where t is the time step. Note that the same network weights are used each time step. The same neural network is presented on the right, where each box represents a neural network layer, and each arrow represents input and output data used by the layer.

2.1.8 Long Short-Term Memory

An LSTM is a recurrent network constructed for tasks with long-term dependencies. A regular RNN struggle to remember longer sequences due to vanishing gradients, which means that the gradients decrease exponentially for each unrolled time step. This is not the case for LSTM networks, due to their internal structure. LSTM networks attempt to mimic a Turing complete machine, by storing memory in cells, and modify this memory by using insert and forget gates. During training, the network learns how to control these gates. As a result of this, both newly seen observations and observations seen a long time ago can be stored and used by the network [27].

A hidden layer in an LSTM consists of four fully connected internal layers that together creates an output $\boldsymbol{h}_t$ and an internal cell state $\boldsymbol{C}_t$, which are both remembered between observations. The cell state is modified by inserting data to it or by removing remembered data. Two of the internal layers are the forget gate layer $\boldsymbol{f}_f$ and the insert gate layer $\boldsymbol{f}_i$. They both return a vector containing values between 0 and 1, where each value describes how important that element is in $\boldsymbol{C}_t$. For the forget layer, a 1 will keep the remembered data and a 0 will forget it. The layers are using a sigmoid activation function $\sigma(x)$ and will later be used for an element-wise multiplication (denoted by $\odot$) with the cell state [27]:

$$\begin{aligned}\boldsymbol{f}_f &= \sigma(\boldsymbol{W}_f \boldsymbol{z}_t + \boldsymbol{b}_f) \\ \boldsymbol{f}_i &= \sigma(\boldsymbol{W}_i \boldsymbol{z}_t + \boldsymbol{b}_i)\end{aligned}$$

A candidate for the new cell state, $\tilde{\boldsymbol{C}}_t$, is computed from $\boldsymbol{z}_t = (\boldsymbol{\xi}_t \,||\, \boldsymbol{h}_{t-1})$, where $||$ denotes a concatenation. The cell state $\boldsymbol{C}_t$ is updated by passing the old cell state $\boldsymbol{C}_{t-1}$ through the forget gate and the candidate cell state $\tilde{\boldsymbol{C}}$ through the insert gate [27]:

$$\begin{aligned}\tilde{\boldsymbol{C}}_t &= \tanh(\boldsymbol{W}_C \boldsymbol{z}_t + \boldsymbol{b}_C) \\ \boldsymbol{C}_t &= \boldsymbol{f}_f \odot \boldsymbol{C}_{t-1} + \boldsymbol{f}_i \odot \tilde{\boldsymbol{C}}_t\end{aligned}$$

The output of the cell, $\boldsymbol{h}_t$, contains selected values from the cell state. Another sigmoid layer, $\boldsymbol{f}_o$, is used to filter what parts of $\boldsymbol{C}_t$ to include in the output [27]:

$$\begin{aligned}\boldsymbol{f}_o &= \sigma(\boldsymbol{W}_o \boldsymbol{z}_t + \boldsymbol{b}_o) \\ \boldsymbol{h}_t &= \boldsymbol{f}_o \odot \tanh(\boldsymbol{C}_t)\end{aligned}$$

A memory for $\boldsymbol{C}_t$ and $\boldsymbol{h}_t$ stores these values for the next observation. A network can contain multiple LSTM cell layers. In that case, each cell has its own memory and computes $\boldsymbol{h}_t$ based on its own $\boldsymbol{C}_{t-1}$ and $\boldsymbol{h}_{t-1}$. The parameters for an LSTM cell are $\theta = \{\boldsymbol{W}_f, \boldsymbol{W}_i, \boldsymbol{W}_C, \boldsymbol{W}_o, \boldsymbol{b}_f, \boldsymbol{b}_i, \boldsymbol{b}_C, \boldsymbol{b}_o\}$, which are trained using backpropagation by unrolling each layer over time, similar to a normal RNN [27]. A graphical representation for an LSTM is presented in Figure 2.5.

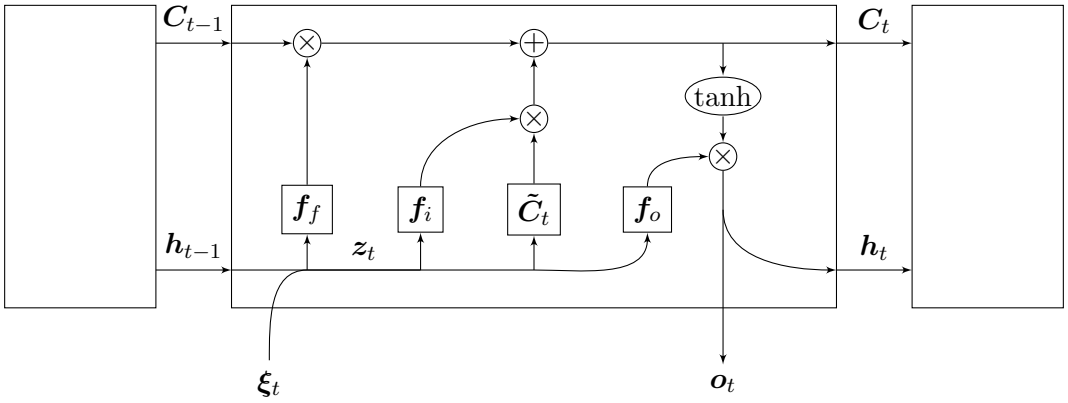


Figure 2.5: The internal structure of an LSTM cell. The picture shows how to take the input $\boldsymbol{\xi}_t$, the previously hidden state $\boldsymbol{h}_{t-1}$ and the previous cell state $\boldsymbol{C}_{t-1}$, and compute the output $\boldsymbol{o}_t$, the new hidden state $\boldsymbol{h}_t$, and the new cell state $\boldsymbol{C}_t$. Two merging arrows denotes a concatenation. Two splitting arrows propagate the same data to every output. Circles denotes a simple mathematical operation. A rectangle represents a fully connected neural network layer.

2.1.9 Dropout to Prevent Overfitting

Overfit neural networks have bad generalization performance [28]. Dropout is used to help reduce overfitting. The idea with dropout is to temporarily remove random hidden neurons with their connections from the network, before each training iteration. This is done by, independently, for each neuron, setting its value to 0 with a probability p [29]. Let $\boldsymbol{m}^{(l)}$ be a mask where each element is 1 with probability p and 0 otherwise. The function for a hidden layer defined in section 2.1.2 is updated to the following equation:

$$\boldsymbol{h}^{(l)} = \boldsymbol{m}^{(l)} \odot g\left(\boldsymbol{W}^{(l)}\boldsymbol{\xi} + \boldsymbol{b}^{(l)}\right)$$

The backpropagation algorithm is updated accordingly. A network used in a training iteration will contain a subset of the neurons from the full neural network. There are an exponential number of these subsets, and backpropagation with dropout can be seen as an algorithm training all those sub-networks. The final network will become an average of all sub-networks. During validation, all neurons are present but their weights are multiplied by p. This makes the expected output of the network during validation the same as during training [29].

2.2 Reinforcement Learning

This section introduces the Markov Decision Process (MDP) model followed by the fundamental framework of reinforcement learning, as well as explanations to exploration and exploitation. Thereafter, Deep Q-Learning algorithm and its stability issues are presented, together with how to solve them. Partially Observable Markov Decision

Process (POMDP), an extension to MDP, is introduced. At last, DRQN is presented, which extends DQN for partially observable environments by using a POMDP.

2.2.1 Markov Decision Process

Reinforcement learning algorithms described in this chapter require the problem to be modeled as a MDP.

Definition 1. An MDP for some environment $\mathcal{E}$ is described by a 5-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$, where $\mathcal{S}$ is the finite state space, $\mathcal{A}$ is the finite action space, $\mathcal{P}$ is the transition probability distribution for traversing from state $s_t \in \mathcal{S}$ to another state $s_{t+1} \in \mathcal{S}$ given an action $a_t \in \mathcal{A}$. The reward $r_t = R(s_t, a_t) \in \mathcal{R}$ is the real-valued immediate rewards, given that action a_t was taken at state s_t. The last parameter $\gamma \in (0, 1]$, which is the discount factor, is used to weight short term and long term rewards.

The MDP model assumes that a state s_{t+1} only depends on the previous state s_t and action a_t. This is known as the Markov property [30].

2.2.2 Reinforcement Learning Framework

Reinforcement learning is an optimization problem where an agent learns a policy by trial and error. Using the MDP model, an agent receives an observation o_t at time t from an environment $\mathcal{E}$. These observations can be fully observable or partially observable of the full environment. The environment could be deterministic or stochastic. From a set of observations, a state s_t is constructed, e.g $s_t = f(o_0, o_1, ..., o_t)$ for some function f. The agent chooses an action a_t, given the state. The action impacts the environment to some degree, which is then used to obtain a new observation o_{t+1} and construct another state s_{t+1}. Some states are terminating states, meaning they have no outgoing actions, and no new states can be constructed [31]. In order to capture both stochastic and deterministic environments, a transition function; $T_{\mathcal{E}}(s_t,a_t,s_{t+1}) \sim \mathcal{P}$ is used to describe the probability to transition from state s_t to s_{t+1} given that action a_t was taken.

To determine whether an action is good or bad, the agent receives a reward $r_t = R(s_t,a_t)$ after choosing an action a_t in a state s_t. The reward function returns either a positive value to reward the agent when choosing good actions, or a negative value to punish it. The agent tries to maximize the reward it gets. The function $R(s_t,a_t)$ must be defined in the model before the learning starts. The definition of the reward function $R(s_t,a_t)$ influences the behavior of the learned policy, as it defines the goal of the task.

A policy π defines the behavior of an agent, by specifying what actions the agent takes. The goal of reinforcement learning is to converge towards the optimal policy π^*. A policy function can be modeled in two ways:

- A deterministic policy $\pi : \mathcal{S} \rightarrow \mathcal{A}$, is a function describing which action the agent will take, given a state.
- A stochastic policy $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0,1]$, is a function describing the probability for taking an action given a state.

In order to learn the optimal policy, two functions; $V_\pi : \mathcal{S} \rightarrow \mathcal{R}$ and $Q_\pi : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{R}$ are used. The value function; $V_\pi(s_t)$, describes the expected future reward given a state s_t using policy π. The action value function; $Q_\pi(s_t,a_t)$, describes the expected future reward for taking an action a_t in a state s_t using the same policy. Comparing $Q_\pi(s_t,a_t)$ to the reward function $R(s_t,a_t)$, the functions are similar, except the Q_π-function also takes expected future rewards into consideration.

The optimal policy π^* takes actions according to an optimal $Q_\pi(s_t,a_t)$ function, denoted $Q_{\pi^*}(s_t,a_t)$. The policy π is updated by iteratively reevaluating all Q_π-values according to the following Bellman equation, resulting in a new policy π':

$$Q_{\pi'}(s_t,a_t) = r_t + \gamma \sum_{s_{t+1} \in S} T_{\mathcal{E}}(s_t,a_t,s_{t+1}) V_\pi(s_{t+1}) \tag{2.7}$$

where $V_\pi(s_t)$ is the value function for the old policy π, defined as:

$$V_\pi(s_t) = \max_{a_t} Q_\pi(s_t,a_t)$$

For a policy π, actions are taken according to the highest Q_π-value [32]:

$$\pi(s_t) = \arg\max_{a_t} Q_\pi(s_t,a_t)$$

When the Q_π-function is iteratively updated, $Q_\pi \rightarrow Q_{\pi*}$ as the number of updates goes to infinity [32].

2.2.3 Exploration and Exploitation

The agent takes actions by exploiting the knowledge it has previously gained, by choosing actions according to its policy. By only exploiting its policy, the agent explores a limited set of states in the environment. There can be unexplored states that give a higher reward unknown to the current policy. By using ϵ-greedy, the agent chooses a random action with probability ϵ and otherwise follows the policy, thus utilizing exploration. The balance between exploration and exploitation can be adjusted by setting ϵ. In the beginning of a training session, ϵ is close to 1, and then decreased over time [31].

2.2.4 Deep Q-Learning

This section presents the Deep Q-Learning algorithm. Two solutions handling stability issues with Deep Q-Learning, experience replay and fixed target network, are also explained.

2.2.4.1 Deep Q-Learning Algorithm

The Deep Q-Learning algorithm is a reinforcement learning algorithm that uses a neural network as a function approximator for the Q_π-function [10]. This neural network is called DQN. The Q_π-function approximated by the DQN is denoted as $Q(s_t,a_t|\theta^\pi)$, where θ^π are the neural network parameters for policy π. When using a DQN, the reward function is optimally distributed around $[-1, 1]$. If the defined reward values are too large, the Q_π values can become large and cause the gradients to grow [33].

Deep Q-Learning is a model-free reinforcement learning algorithm, which means that the transition function $T_\mathcal{E}(s_t,a_t,s_{t+1})$ for the MDP is unknown. By conducting experiments and receiving experiences e from the environment $\mathcal{E}$, the transition function $T_\mathcal{E}(s_t,a_t,s_{t+1})$ is embedded inside the Q_π values. Actions are taken according to the ϵ-greedy strategy, and experiences from taken actions are recorded [10]. An experience e is defined as a tuple of the current state s_t, taken action a_t, observed reward r_t and the new state s_{t+1}; $e = (s_t,a_t,r_t,s_{t+1})$. These experiences are stored in an experience memory E and are used for training the neural network. The neural network trains its weights and biases θ^{π_i}, where π_i is the policy for training iteration i. In order to train the DQN, Q_i^{target} is computed from an experience, which is the real immediate reward and predicted future rewards for iteration i, derived from Equation 2.7:

$$Q_i^{target} = \begin{cases} r_t & \text{if } s_t \text{ is terminating the episode} \\ r_t + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}|\theta^-) & \text{otherwise} \end{cases}$$

A separate set of network parameters θ^-, which are computed from $\theta^{\pi_{i-1}}$, is used for computing Q_i^{target}. This is called a fixed target network, and is introduced to make Q_i^{target} independent from θ^{π_i} to make DQN updates more stable [11]. How to set the target network parameters θ^- is found in Section 2.2.4.3.

The loss function $L(\theta^{\pi_i})$, for a DQN with network parameters θ^{π_i}, measures the difference between the real reward and the DQN's predicted reward, which is implemented by computing the sum of squared errors between $Q(s_t,a_t|\theta^{\pi_i})$ and Q_i^{target} [10]:

$$L(\theta^{\pi_i}) = \frac{1}{2N} \sum_{(s_t,a_t)\sim\rho_\pi} \left(Q(s_t, a_t|\theta^{\pi_i}) - Q_i^{target} \right)^2 \tag{2.8}$$

where ρ_π is a probability distribution over states s_t and actions a_t for policy π, which is used to sample N state and action pairs from the experience memory E. Using the loss function, the following gradient is computed, which is the gradient used when performing backpropagation on the DQN [10]:

$$\nabla_{\theta^{\pi_i}} L(\theta^{\pi_i}) = \frac{1}{N} \sum_{(s_t,a_t)\sim\rho_\pi; s_{t+1}\sim\mathcal{E}} \left(Q(s_t, a_t|\theta^{\pi_i}) - Q_i^{target} \right) \nabla_{\theta^{\pi_i}} Q(s_t, a_t|\theta^{\pi_i}) \tag{2.9}$$

The Deep Q-Learning algorithm, using a fixed target Q-network, is presented below [10].

Data: An environment $\mathcal{E}$, Reward function R and actions $\mathcal{A}$
Result: Trained neural network parameters θ^{π}
Initialize experience memory E with capacity N
Initialize random weights $\theta^{\pi} = \theta^{-}$ for the neural networks
for *training episode 1 to M* **do**
  Initialize environment and receive initial state s_1
  for *t=1 to T* **do**
    Select random action a_t with prob ϵ otherwise set $a_t := \arg\max_a Q(s_t,a|\theta^{\pi})$
    Execute action a_t, receive reward r_t and observe the new state s_{t+1}
    Store experience $e_t := (s_t,a_t,r_t,s_{t+1})$ in E
    Sample B experiences E' from E
    for $e_j = (s_j, a_j, r_j, s_{j+1})$ *in* E' **do**
      $$Q^{target} := \begin{cases} r_j \text{ if } s_j \text{ is a terminating state} \\ r_j + \gamma \max_{a_{j+1}} Q(s_{j+1},a_{j+1}|\theta^{-}) \text{ otherwise} \end{cases}$$
      Learn from experience e_j by minimizing L from the gradient in Equation 2.9 using gradient descent, and thus compute new network weights θ^{π}
      Update target network parameters θ^{-} by using the main network parameters θ^{π} with Equation 2.10 or 2.11
    end
  end
end

Algorithm 2: Deep-Q Learning

2.2.4.2 Experience Replay

A problem with Deep Q-Learning is that with a small change in the action-value function, the policy can drastically change, due to the ϵ-greedy policy rule. The distribution of future training samples will be greatly influenced by the updated policy. If the network only trains on recent experiences, a biased distribution of samples is used. Such behavior can cause unwanted feedback loops which can lead to divergence of the trained network [34].

In order to make DQN more robust, the agent can sample experiences E' from the experience memory E. The sampled experiences are fed to the gradient decent algorithm as a mini-batch. Thus, the agent learns on an average of the experiences in E and is likely to train on the same experiences multiple times, which can speed up the network's convergence and reduce oscillations [10, 35].

2.2.4.3 Fixed Target Q-Network

The target network, with weights and biases θ_i^{-}, for training iteration i, has the same structure as the main Q network, with weights and biases θ^{π_i}. The target network is

updated at a slower pace than the main network, which reduce oscillations [11]. Two different update strategies for the target network's parameters can be used:

$$1. \quad \text{With interval } C\text{: } \theta_i^- = \theta^{\pi_i} \tag{2.10}$$

$$2. \quad \text{Every network update: } \theta_i^- = \tau \cdot \theta_{i-1}^- + (1-\tau) \cdot \theta^{\pi_i} \tag{2.11}$$

where i is the update iteration. For the interval update (1), C determines how frequently the target network is updated. A smooth update (2) can also be used, where τ represents how much to interpolate the target network for the last update with the current main network [11, 36].

2.2.5 Partial Observability

DQN relies on the MDP model described in Definition 1. When dealing with partially observable environments, a model can be constructed by using a partially observable MDP model. The problem is still assumed to be an MDP, except that parts of the state are hidden from the agent [14].

Definition 2. A POMDP for some environment $\mathcal{E}$ is described by a 6-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \Omega, O)$, where $\mathcal{S}$, $\mathcal{A}$, $\mathcal{P}$ and $\mathcal{R}$ represents an MDP. An observation $o_t \in \Omega$ is observed, where Ω is the finite observation space. Finally, $O(s_t)$ is the probability distribution for receiving observation o_t given an underlying hidden state s_t: $o_t \sim O(s_t)$.

2.2.6 Deep Recurrent Q-Network

A DRQN is a DQN with an LSTM cell added after the last hidden layer, before the output layer, enabling support for POMDP environments [12]. The LSTM layer adds a recurrence over time, such that the network's output for observation o_t depends on a history of previously seen observations. The LSTM's internal cell state keeps information about this observation history. Thus, only the current observation o_t needs to be fed to the network to make a prediction. By using a history of observations, DRQN can better estimate the underlying system state of the POMDP. The internal cell state is reset to zeros at the start of a new episode.

A DRQN is trained by feeding a sequence of observations $o_{t-l+1}, \ldots, o_t$ to the unrolled neural network, where l is the sequence length, and o_{t+1} as the next observation. The LSTM's internal cell state is not stored in the experience memory, as it is dependent of the network parameters θ. An experience is redefined to contain sequences of observations, actions and rewards; $e = (o_{t-l+1} \ldots o_t,\ a_{t-l+1} \ldots a_t,\ r_{t-l+1} \ldots r_t, o_{t+1})$ [12]. Before each training update, the LSTM's internal state is zeroed. This method preserve random sampling from the experience memory, described in Section 2.2.4.2, which makes the recurrent update stable.

This method is extended by using the first h observations of an experience to build the LSTM layer's cell state. Only the rest of the observations, $o_{t-l+h+1}, \ldots, o_t$, are trained on [37]. Figure 2.6 visualizes how these observations are used.

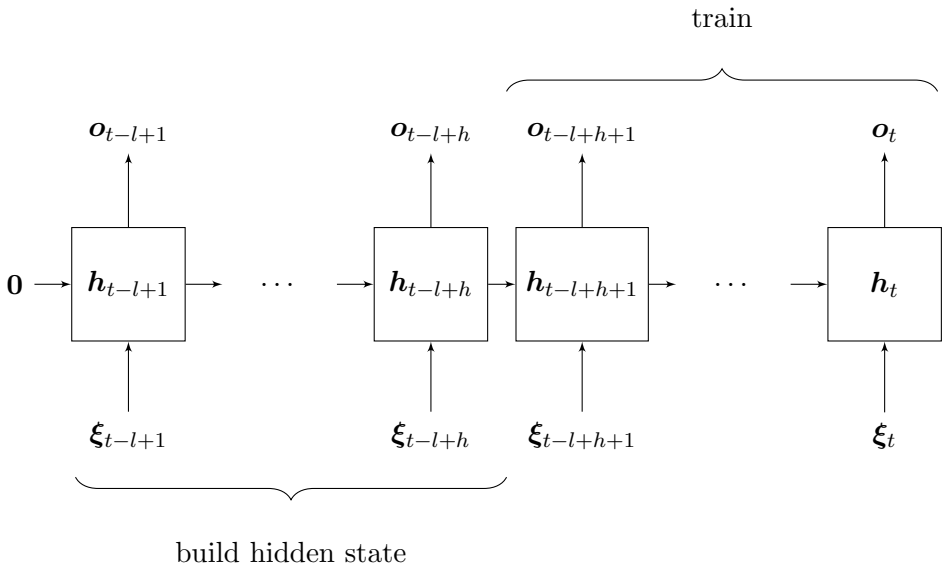


Figure 2.6: Use h number of observations in a sequence to build the LSTM's cell state. The rest $l - h$ number of observations are trained on. The observations are fed as inputs $\boldsymbol{\xi}_t$. The initial cell state of $\boldsymbol{h}_{t-l+1}$ is zero.

3

Traffic Simulator for Crossings

This chapter describes the fundamental mechanics of the simulator. This includes an episode's terminating conditions. The coordinate system for lanes is defined together with how lanes relate to each other in the coordinate system. Furthermore, the representation of curved lanes is explained. Thereafter, the parameters that the car model use, as well as its mechanics are presented. At last, the agents that drives target cars are presented.

3.1 Episode Timeframe

Each time the traffic simulator starts a new episode, it initializes a traffic scenario with cars and lanes, as described in Section 1.4.1. The episode ends when one of the following terminating conditions are fulfilled:

- The ego car has successfully arrived at its target destination along the lane, specified in the traffic scenario.
- The ego car has collided with another car. A vehicle is shaped like a rectangle, and overlapping of two such rectangles results in a collision between the two vehicles.
- A timeout has occurred; $t > t_m$ where t is the current time from the start of the current episode, and t_m is the maximum amount of time allowed for that episode, specified by the scenario.

3.2 Coordinate System for Lanes

A coordinate system [1] models crossings and merging lanes, such as the scenarios described in section 1.3. A lane L is defined as a sequence of coordinates $L := (\psi_1 =$

[1] The coordinate system is proposed by Mohammad Ali at Zenuity.

$(x_1,y_1), \boldsymbol{\psi}_2 = (x_2,y_2), \dots)$ and a lane width $w^{(L)}$, constructing a sequence of straight *lane segments* that cars can drive on. A lane segment is denoted as $(\boldsymbol{\psi}_i,\boldsymbol{\psi}_{i+1}); \boldsymbol{\psi}_i \in L \wedge \boldsymbol{\psi}_{i+1} \in L$. An illustration of a lane and its lane segments is presented in Figure 3.1. An *intersection point* between two lanes L and L' is denoted by $\boldsymbol{\psi}_\times$ and exists if the two lanes have a common point:

$$\boldsymbol{\psi}_\times \in L \wedge \boldsymbol{\psi}_\times \in L'$$

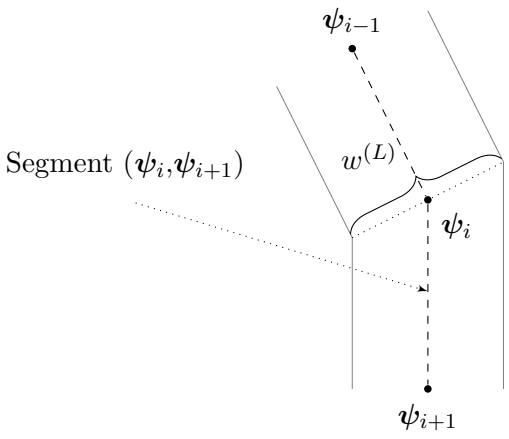


Figure 3.1: An illustration of three consecutive points $\boldsymbol{\psi}_{i-1}, \boldsymbol{\psi}_1$ and $\boldsymbol{\psi}_{i+1}$ for a lane L. The lane has two segments with one of the segments $(\boldsymbol{\psi}_i,\boldsymbol{\psi}_{i+1})$ explicitly shown at the end of the dotted arrow. The lane width $w^{(L)}$ is also shown in the figure.

The longitudinal position along a lane is described by $\phi_i^{(L)}$ where i identifies a point $\boldsymbol{\psi}_i$ along the lane L. The longitudinal position up to a certain point along the lane is defined recursively as follows:

$$\begin{aligned}\phi_1^{(L)} &= 0 \\ \phi_i^{(L)} &= ||\boldsymbol{\psi}_i - \boldsymbol{\psi}_{i-1}|| + \phi_{i-1}^{(L)}\end{aligned}$$

The lane length is $\phi_l^{(L)}$ where l identifies the last point $\boldsymbol{\psi}_l$ on the lane. For any point $\boldsymbol{\psi}$, its longitudinal position along the lane L is described by $\phi^{(L)}$, and can be any number in the interval $[0,\phi_l^{(L)}]$. All lanes have a driving direction which increase longitudinal positions as the car moves forward along the lane. Cars can only traverse forward on lanes. A car's longitudinal position can be described from lanes other than its current lane. Let L_c denote the current lane for a car c, and let L_o denote another lane. Then, the car has a longitudinal position along lane L_o if and only if there exists an intersection point $\boldsymbol{\psi}_\times$ between the two lanes, and this point is ahead of the car. Formally:

$$\boldsymbol{\psi}_\times \in L_c \wedge \boldsymbol{\psi}_\times \in L_o \wedge \phi_\times^{(L_c)} \geq p_t^{(c,L_c)}$$

where $p_t^{(c,L_c)}$ denotes the longitudinal position of the car c along its lane L_c at time step t, and $\phi_\times^{(L_c)}$ is the longitudinal position of the intersection point between the lanes L_c and L_o along the lane L_c. When a longitudinal position of car c on another lane L_o is calculated, the first intersection point is always used. If there are multiple intersection

points, the two lanes merge. The first intersection point is the one with the lowest longitudinal position $\phi_{\times}^{(L)}$. Figure 3.2 (left) presents two intersecting lanes with multiple intersection points, and shows the first and the last intersection points. The car c's position along another lane L_o, $p_t^{(c,L_o)}$, is calculated such that the longitudinal distance to the first intersecting point $\boldsymbol{\psi}_{\times}$ is the same on both lanes L_c and L_o, by using Equation 3.1. Figure 3.2 (right) displays an example of the longitudinal position of the same car on two different intersecting lanes.

$$p_t^{(c,L_o)} = \phi_j^{(L_o)} - \phi_i^{(L_c)} + p_t^{(c,L_c)} \; ; \quad \boldsymbol{\psi}_i \in L_c, \boldsymbol{\psi}_j \in L_o, \boldsymbol{\psi}_i = \boldsymbol{\psi}_j \tag{3.1}$$

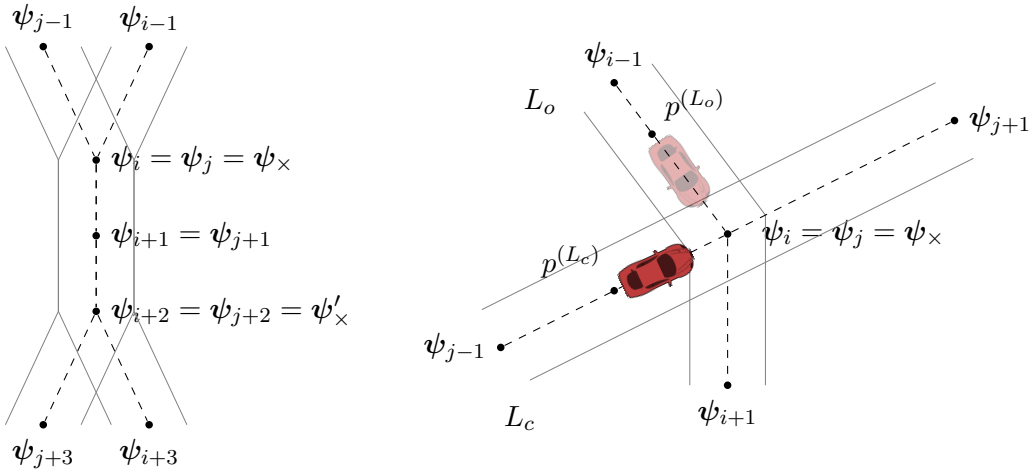


Figure 3.2: Left figure: Two lanes merge and then diverge again. There exists three intersection points between the two lanes, where the first intersection point is denoted $\boldsymbol{\psi}_{\times}$ and the last intersection point is denoted $\boldsymbol{\psi}'_{\times}$. Right figure: Two lanes L_c and L_o intersect at $\boldsymbol{\psi}_{\times}$. A car c in front of the intersection has a position $p^{(L_c)}$ on lane L_c and $p^{(L_o)}$ on lane L_o

In order to determine if any point $\boldsymbol{\psi}$ is on a specific lane, $\boldsymbol{\psi}$ is projected onto every segment of the lane. If the distance between the original point $\boldsymbol{\psi}$ and the projected point is less than half the lane width $w^{(L)}$, the point is considered to be on the lane. An *overlap point* $\boldsymbol{\psi}_{\circ}^{(L)}$ is defined between two lanes L and L' if and only if there exists at least one point $\boldsymbol{\psi}$ that is on both lanes. The overlap point for lane L is the projected point from the set of all points on both lanes with the lowest longitudinal position. Let $\phi_{\circ}^{(L)}$ denote the longitudinal position of such a projected point, then

$$\phi_{\circ}^{(L)} = \min \phi^{(L)}$$

where $\phi^{(L)}$ comes from the set of all projected longitudinal positions, which are projected from every point $\boldsymbol{\psi}$ that are on both the lanes L and L'. Figure 3.3 presents two lanes, with the set of all points that are on both lanes (potential overlap points, marked in grey) as well as the overlap point $\boldsymbol{\psi}_{\circ}^{(L)}$.

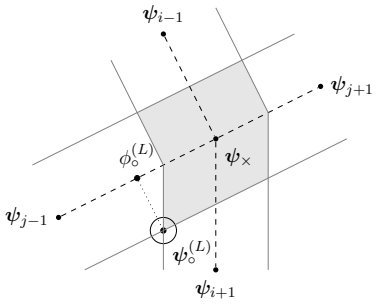


Figure 3.3: Two overlapping lanes L and L'. The grey area is the set of all points which are considered on both lanes. The overlapping point $\boldsymbol{\psi}_\circ^{(L)}$ for lane L is presented at the center of a circle. The longitudinal distance of the overlap point along the lane L; $\phi_\circ^{(L)}$ is also presented. Note that the overlap point for lane L'; $\boldsymbol{\psi}_\circ^{(L')}$ might not always be the same point as $\boldsymbol{\psi}_\circ^{(L)}$.

3.3 Approximating Curved Lanes

Curved lanes are modeled by multiple segments. The curvature c_i is defined for each coordinate $\boldsymbol{\psi}_i \in L$. The curvature is approximated by constructing a fictive radius r_i of the curve, given the angle β_i between the two segments around $\boldsymbol{\psi}_i$ and the length of the shortest segment. This is shown in Figure 3.4 and in the following equation:

$$r_i = \frac{\min(||\boldsymbol{\psi}_i - \boldsymbol{\psi}_{i-1}||, ||\boldsymbol{\psi}_{i+1} - \boldsymbol{\psi}_i||)}{2\arctan(\frac{\beta_i}{2})}$$

$$c_i = \frac{1}{r_i}$$

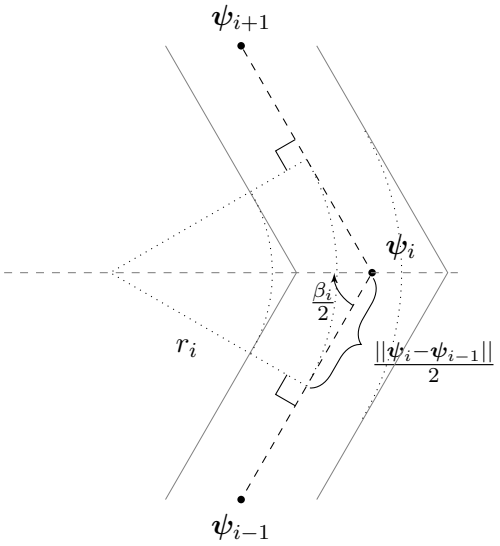


Figure 3.4: Curvature approximation between two straight lane segments $(\boldsymbol{\psi}_{i-1}, \boldsymbol{\psi}_i)$ and $(\boldsymbol{\psi}_i, \boldsymbol{\psi}_{i+1})$. Assuming $(\boldsymbol{\psi}_{i-1}, \boldsymbol{\psi}_i)$ is the shortest of the two, the radius is computed using $\arctan(\frac{\beta_i}{2})$ and $\frac{||\boldsymbol{\psi}_i - \boldsymbol{\psi}_{i-1}||}{2}$. If $(\boldsymbol{\psi}_i, \boldsymbol{\psi}_{i+1})$ is the shortest, $\frac{||\boldsymbol{\psi}_{i+1} - \boldsymbol{\psi}_i||}{2}$ is used instead.

3.4 Car Model

A vehicle c is described by these parameters at time t:

$p_t^{(c)}$: The longitudinal position of the vehicle along its lane, positioned at the rear axle.

$v_t^{(c)}$: The speed along the direction of the vehicle.

$a_t^{(c)}$: The acceleration along the direction of the vehicle.

$\boldsymbol{s}^{(c)} = \begin{bmatrix} s_1^{(c)} \\ s_2^{(c)} \end{bmatrix}$: The width and length of the vehicle.

A vehicle's position, speed, and acceleration are updated according to the following equations:

$$
\begin{aligned}
a_{t+1}^{(c)} &= a_t^{(c)} + j_{t+1}^{(c)} \Delta t \\
v_{t+1}^{(c)} &= v_t^{(c)} + a_{t+1}^{(c)} \Delta t - \frac{1}{2} j_{t+1}^{(c)} \Delta t^2 \\
p_{t+1}^{(c)} &= p_t^{(c)} + v_{t+1}^{(c)} \Delta t - \frac{1}{2} a_{t+1}^{(c)} \Delta t^2
\end{aligned}
$$

where $j_t^{(c)}$ is the *jerk* along the lane, defined as $j_t^{(c)} := \max(\min((\hat{a}_t^{(c)} - a_t^{(c)})/\Delta t), j_{\max}, -j_{\max})$, with $j_{\max}$ as *max jerk*, $\hat{a}_t^{(c)}$ is the desired acceleration, used to control the vehicle, and Δt denotes the time between two simulation updates, thus approximating continuous time with discrete updates. Max jerk $j_{\max}$ is used to limit the acceleration control signal, thus making sure agents are limited by artificial inertia. Whenever a car reach the end of its lane, it is instantly put at the start of the lane, to keep the same number of cars throughout the episode.

3.5 Car Agents with Different Behaviours

The following agents are defined as drivers for target cars. Each agent have manually defined STGs which are passed to the low-level controller. How to compute the desired acceleration from an STG is defined is described in Section 4.2.2:

- **Take way agent**. An agent always driving according to the take way STG.
- **Give way late agent**. The agent initially has the same behavior as the take way agent, but very late decides to stop at the next crossing using the give way STG.
- **Cautious agent**. The agent initially has the same behavior as the take way agent, but slows down before the crossing. In contrast to the give way late agent, the agent does not stop, and switches to the take way STG when it is close to the crossing. A parameter determining how much the agent slows down when approaching a crossing is configurable, and is referred to as *cautiousness*.

- **Trained agent**. An agent using a policy from a previously trained ego agent. The policy of this agent depends on how it was trained. The trained agent differs from an ego agent because it cannot further update its policy, and is used to drive target cars.

4

Acceleration Regulator

This chapter discusses the how the Acceleration Regulator works and motivates the algorithm choices. Both the high- and low-level controller presented in Section 1.3 are described. Furthermore, all the features that the high-level controller base its decisions on are presented. The reward function used to train the high-level controller is presented. At last, the different DQN structures the high-level controller is trained and tested with in the Result chapter are presented.

4.1 System Design Choices

As mentioned in Section 1.3, the Acceleration Regulator is divided into a high- and low-level controller. The high-level controller chooses between discrete actions, which the low-level controller use to compute the final acceleration. This architecture enables the high-level controller to use a reinforcement learning algorithm with discrete outputs. A policy-based or an actor-critic method could be used without the need for a low-level controller, as it supports continuous output [38, 39]. However, an approach with discrete outputs reduces the action space, by constraining the possible output acceleration distribution. Also, a model-free reinforcement learning algorithm is necessary to avoid creating a manual model for the MDP of the traffic environment. DQN and DRQN, presented in Sections 2.2.4.1 and 2.2.6, are both model-free and use discrete actions. DQN, however, relies on the Markov property assumption, which [14] argue does not hold for traffic environments. DRQN solves this problem by using a POMDP environment instead of an MDP [12]. Both DQN and DRQN are chosen as implementation for the high-level controller, and compared to see if POMDP is necessary.

4.2 Car Control using the Low-Level Controller

The low-level controller computes a vehicle c's desired acceleration $\hat{a}_t^{(c)}$ at time t, for an STG, using regulators. These regulators and how the low-level controller computes the desired acceleration from STGs, are defined in the following sub-sections.

4.2.1 Regulators

There are three regulators that can be used by the low-level controller, where each regulator returns an acceleration:

- *Cruise Control (CC).* Accelerates or decelerates the car to a desired speed v_{desired} and then keeps that speed. The regulator function takes a relative speed $v_{\text{rel},t}$ as argument and returns an acceleration:

$$CC(v_{\text{rel},t})$$
$$\text{where } v_{\text{rel},t} = v_{\text{desired}} - v_t^{(c)}$$

- *Adaptive Cruise Control (ACC).* Used for keeping a desired distance to a selected target, for instance a leading car or a crossing. The car will accelerate or decelerate to reach a relative distance and relative speed of 0 to the target. The regulator takes a relative distance $d_{\text{range},t}$ and relative speed $v_{\text{rel},t}$:

$$ACC(d_{\text{range},t}, v_{\text{rel},t})$$

- *Curve Speed Adaptation.* Limits the speed in a curve. Given curvatures c_i (described in Section 3.3) for all future sample points $\boldsymbol{\psi}_i$ on the ego lane L_c, the maximum speed for every point is calculated. The maximum longitudinal acceleration for the regulator is given by limiting the vehicle's speed in each sample point, where a_{lat} is the maximum allowed lateral acceleration:

$$CSA(p_t^{(c)}, v_t^{(c)}) = \min_i ACC\left(\phi_i^{(L_c)} - p_t^{(c)}, \sqrt{\frac{a_{\text{lat}}}{c_i}} - v_t^{(c)}\right)$$
$$\text{where } \boldsymbol{\psi}_i \in L_c \wedge \phi_i^{(L_c)} > p_t^{(c)}$$

A common notation is used for constraining the vehicle acceleration according to the three given regulators. If a car c has a leading car l, the acceleration is limited by an ACC regulator with the leading car l as target. It also takes the curvature of the lane into account, in order to make sure that the car does not experience too much lateral acceleration. The constrained acceleration $\bar{a}_t^{(c)}$ in time step t, is defined as

$$\bar{a}_t^{(c)} = \begin{cases} \min\left(\tilde{a}, ACC(p_t^{(l)} - p_t^{(c)} - s_2^{(c)} - d_{\text{offset}}, v_t^{(l)} - v_t^{(c)})\right) & \text{if leading car exists} \\ \tilde{a} & \text{otherwise} \end{cases} \tag{4.1}$$

$$\tilde{a} = \min\left(a_{\max}, CC(v_{\max} - v_t^{(c)}), CSA(p_t^{(c)}, v_t^{(c)})\right)$$

where $a_{\max}$ denotes the maximum amount of acceleration, $v_{\max}$ is the maximum allowed speed, d_{offset} is the desired distance to the leading car, and $\bar{a}_t^{(c)}$ denotes the constrained acceleration for car c.

4.2.2 Low-Level Controller Implementation using Short Term Goals

As seen in Section 1.4.3, the low-level controller use regulators, defined in Section 4.2.1, to adjust the acceleration for a vehicle c according to an STG. It outputs a desired acceleration $\hat{a}_t^{(c)}$, which the car model in Section 3.4 use to update the vehicle's acceleration $a^{(c)}$. The low-level controller is defined as follows for each STG:

- *Take way.* Aims to keep a specified maximum speed $v_{\max}$, using a CC regulator. The desired acceleration $\hat{a}_t^{(c)}$ is defined by the constrained acceleration $\bar{a}_t^{(c)}$, defined in Equation 4.1 for a car c:

  $$\hat{a}_t^{(c)} = \bar{a}_t^{(c)}$$

- *Give way.* Stops the car at the next intersection, located at $\boldsymbol{\psi}_i$ on the car's lane L_c. The desired acceleration $\hat{a}_t^{(c)}$ is limited by both the constrained acceleration $\bar{a}_t^{(c)}$ and an ACC regulator targeted on a stopping point in front of the intersection. Define the distance to the stopping point as:

  $$d_i^{(c)} = \phi_i^{(L_c)} - p_t^{(c)} - s_2^{(c)} - d_{\text{margin}}$$

  where d_{margin} is the distance from the intersection i to the desired stopping point. The desired acceleration $\hat{a}_t^{(c)}$ is then given by:

  $$\hat{a}_t^{(c)} = \min\left(\bar{a}_t^{(c)}, ACC(d_i^{(c)}, -v_t^{(c)})\right)$$

- *Follow vehicle.* Using an ACC regulator targeted on a target car o, to keep the same speed $v_t^{(o)}$ and a desired distance d_{offset} to that target car. The desired acceleration $\hat{a}_t^{(c)}$ is given by:

  $$\hat{a}_t^{(c)} = \min\left(\bar{a}_t^{(c)}, ACC(p_t^{(o)} - p_t^{(c)} - s_2^{(c)} - d_{\text{offset}}, v_t^{(o)} - v_t^{(c)})\right)$$

4.3 Actions Chosen by the High-Level Controller

The high-level controller chooses one of 6 actions $\{\alpha_1, \ldots, \alpha_6\} \in \mathcal{A}$. As described in Section 1.4.3, an action is defined to be an STG, used by the low-level controller to compute the desired acceleration. The 6 actions are:

- α_1: Take way STG.
- α_2: Give way STG.
- α_3: Follow vehicle STG, 1.
- α_4: Follow vehicle STG, 2.
- α_5: Follow vehicle STG, 3.
- α_6: Follow vehicle STG, 4.

The four follow vehicle STG actions, $\alpha_3, \ldots, \alpha_6$; one for each target car, are used to improve the ego car's timing in a crossing or a lane merge. The intended use of the follow vehicle action is to follow the traffic flow in the target car's lane. Each follow vehicle action is only valid if its target car is visible to the ego agent. The low-level controller's output is undefined for invalid follow vehicle actions. If an invalid follow vehicle action is chosen, the acceleration from the take way action is used instead.

4.4 Selected Features

The features used for the high-level controller's decision making include features concerning the ego car e and up to 4 target cars in the environment. These features represent the MDP state $s_t = (\xi_1, \ldots, \xi_p) \in \mathcal{S}$ which Deep Q-Learning base its decisions on, described in Section 2.2.1. As mentioned in Section 1.4.4, there are 32 *target features*, 8 for each target car, and 7 *ego features*. Motivations for the features are presented after the features are defined.

4.4.1 Target features

Target features include all features related to a specific target car c and is repeated for each car visible to the ego car. Because the features are used as input to a DQN, the number of features must be fixed. The features were therefore decided to support up to 4 target cars, as mentioned in Section 1.5, but the approach can support more by simply expanding the number of target cars. A target car is visible to the ego car if the two cars have a common point $\boldsymbol{\psi}$ on their respective lanes, ahead of the ego car: $\phi^{(L_e)} > p_t^{(e)}$, where $\phi^{(L_e)}$ is the longitudinal position of the common point $\boldsymbol{\psi}$ along the ego lane L_e. Thus, target cars driving on already passed lanes are not visible. This means that a traffic scenario can have less than 4 visible target cars, which can leave empty target features. In such cases, the empty target features are set to -1. All target features are scaled to be a value between or close to $[-1,1]$ by using a car's sight range $p_{\max}$, maximum speed $v_{\max}$ and maximum acceleration $a_{\max}$.

Let $\boldsymbol{\psi}_\times \in L_e$ be the intersection point between the ego lane L_e and the target car's lane L_c, with the longitudinal position $\phi_\times^{(L_e)}$ along the ego lane. Let $\boldsymbol{\psi}_\circ^{(L_e)}$ and $\boldsymbol{\psi}_\circ^{(L_c)}$ be the overlap points for L_e and L_c respectively, between the two lanes, with a longitudinal position of $\phi_\circ^{(L_e)}$ and $\phi_\circ^{(L_c)}$ respectively. The target features are listed below:

- ξ_1: **Target car's position relative to ego car**, using the target car's position along the ego lane:
$$\frac{p_t^{(c,L_e)} - p_t^{(e)}}{p_{\max}}$$
- ξ_2: **Target car's speed**, $\frac{v_t^{(c)}}{v_{\max}}$
- ξ_3: **Ego car's distance to the first intersection point $\boldsymbol{\psi}_\times$**. If $L_c = L_e$, the distance is set to be 0, otherwise:
$$\frac{\phi_\times^{(L_e)} - p_t^{(e)}}{p_{\max}}$$
- ξ_4: **Ego car's speed**, $\frac{v_t^{(e)}}{v_{\max}}$
- ξ_5: **Ego car's acceleration**, $\frac{a_t^{(e)}}{a_{\max}}$
- ξ_6: **Remaining joined distance of ego lane after $\boldsymbol{\psi}_\times$**. The distance from the first intersection point $\phi_\times^{(L_e)}$ to the last intersection point $\phi_\times^{\prime(L_e)}$ for lanes L_e and L_c. For a crossing, this value is 0. The distance is calculated from $\boldsymbol{\psi}_\times$, or from the ego car, if it has passed $\boldsymbol{\psi}_\times$,
$$\min\left(\frac{\phi_\times^{\prime(L_e)} - \max(\phi_\times^{(L_e)}, p_t^{(e)})}{p_{\max}}, 1\right)$$
- ξ_7: **Target car's distance to overlap with ego lane**. The distance between the target car and the overlapping point between both lanes along the target car's lane L_c, defined as:
$$\frac{\phi_\circ^{(L_c)} - p_t^{(c)}}{p_{\max}}$$
- ξ_8: **Ego car's distance to overlap with target lane**. The distance between the ego car and the overlapping point between both lanes along the ego car's lane L_e, defined as:
$$\frac{\phi_\circ^{(L_e)} - p_t^{(e)}}{p_{\max}}$$

The first four features $\xi_1, \ldots, \xi_4$, which include speeds and positions relative to the intersection point for the ego car and the target car, are included to let the agent infer when the cars will arrive at the intersection, which is essential to the problem. The ego car's acceleration ξ_5 is needed to determine how each action will affect the acceleration, which is essential to minimize jerk. The remaining joined distance ξ_6 is used to differentiate whether the target- and ego lane merge or not. The last two features ξ_7 and ξ_8 are used

to determine the distance that the ego car travels on the target lane, in combination with ξ_3, and vice versa for the target car in combination with ξ_1. Figure 4.1 shows an example of where this is important, since the distance in the turning scenario is longer than in the straight crossing (as given by $\xi_7 + \xi_1 - \xi_3$).

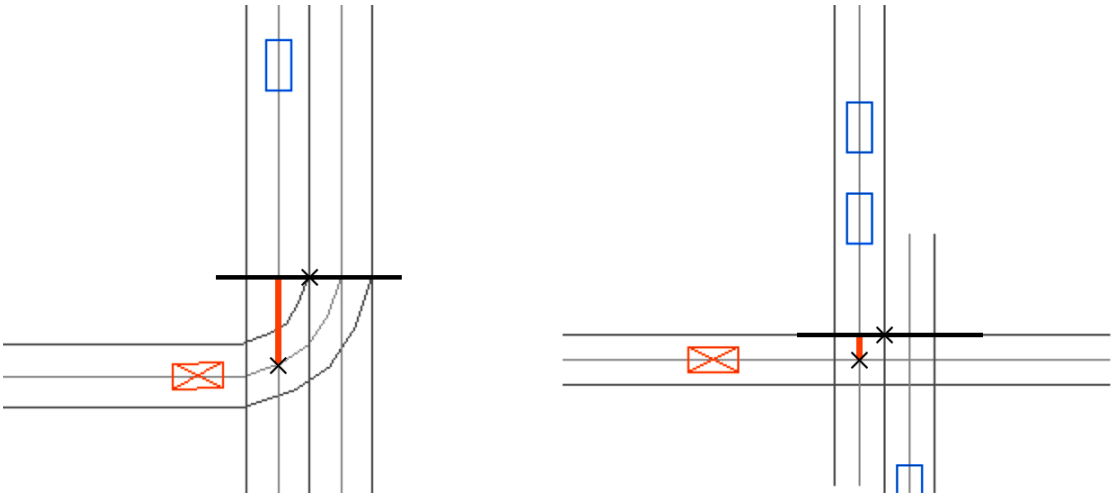


Figure 4.1: In a turning scenario (left), the longitudinal distance between the intersection point $\boldsymbol{\psi}_\times$ and the target car's first overlapping point $\boldsymbol{\psi}_\circ^{(L_c)}$ is larger than in a straight crossing (right).

4.4.2 Ego features

The ego features include inputs that are not related to any target cars. These features tell the ego agent what happens when a specific action is chosen. The ego features are scaled the same way as target features.

- ξ_{33}: **Distance to next intersection.** The distance between the ego car and the intersection point for the closest intersecting lane. Let $L_1, L_2, \ldots$ be all lanes that intersect with ego lane L_e. Then, the distance is defined as:

$$\min_i \left(\frac{\phi_\times^{(L_e)} - p_t^{(e)}}{p_{\max}} \right) \text{ where } \phi_\times^{(L_e)} > p_t^{(e)} \wedge \boldsymbol{\psi}_\times \text{ is intersection point for } (L_e, L_i)$$

- $\xi_{34:39}$: **Acceleration for given action**. The acceleration that an action will return if it is chosen. There is one feature for each action in $\mathcal{A}$, scaled using $a_{\max}$.

If there are multiple lanes intersecting with the ego lane, give way STG will stop the ego car in front of the closest intersecting lane. Feature ξ_{33} provides the ego agent with the distance to that intersection. Features $\xi_{34:39}$ give information about the acceleration for each action, used for the agent to sense acceleration restrictions that originate from curved lanes or leadning cars. It also helps to minimize jerk.

4.5 Reward Function

The reward function is defined as follows:

$$r_t = \hat{r}_t + \begin{cases} 1 - \frac{t}{t_m} & \text{on success,} \\ -2 & \text{on collision} \\ -0.1 & \text{on timeout, i.e. } t \geq t_m \\ -\left(\frac{j_t^{(e)}}{j_{\max}}\right)^2 \frac{\Delta t}{t_m} & \text{on non-terminating updates} \end{cases}$$

$$\text{where } \hat{r}_t = \begin{cases} -1 & \text{if chosen } a_t \text{ is not valid} \\ 0 & \text{otherwise} \end{cases}$$

The actions $\alpha_3, \ldots, \alpha_6$ (follow vehicle) described in Section 4.3 should only be selected when they are valid. To learn when these actions are valid, the agent gets punished on invalid choices using $\hat{r}_t$. The reward function also punishes the agent when the jerk is large. Accelerations returned by the low-level controller for different STGs can vary, which increases jerk and can make the ride uncomfortable.

4.6 DQN Structures for the High-Level Controller

This section presents and motivates the neural network structures that are used by the high-level controller. First, a simple feed-forward neural network structure is presented, followed by a network using shared weights. Two network structures handling multiple observations are presented: a DRQN and a DQN with stacked observations. For these two networks, a sequence length l determines the number of observations used for training. The section also explains how stochastic sequence length can be used for DRQN.

4.6.1 Fully Connected Deep Q-Network

The simplest DQN that is used consists of three fully connected hidden layers followed by the output layer. The selected features from Section 4.4 are fed as input $\boldsymbol{\xi}$ and the network outputs Q-values for all actions $\alpha_1, \alpha_2, \ldots$ as a vector $\boldsymbol{o}$. This network structure is used to compare to other more complicated network structures. It is visualized in Figure 4.2.

4.6.2 Shared Weights Between Cars

To make the learning process faster and to improve performance, the network structure is modified to share some network weights. This means that the same weight parameter are used for multiple neurons in the network. The output of the network should be

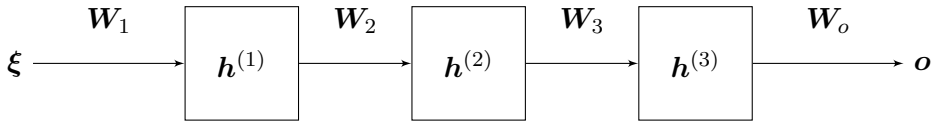


Figure 4.2: A simple feed forward neural network with two hidden layers. A box represents a layer in the network. The network consists of hidden layers $\boldsymbol{h}^{(1)}$, $\boldsymbol{h}^{(2)}$ and $\boldsymbol{h}^{(3)}$ denoted on each the box. Arrows represents the feed forward data flow direction using the network weights $\boldsymbol{W}_1, \boldsymbol{W}_2$, $\boldsymbol{W}_3$ and $\boldsymbol{W}_o$ which are denoted next to each arrow.

independent of the order of the target cars in the input $\boldsymbol{\xi}$. In other words, whether a car is fed into $\xi_1, \ldots, \xi_8$ or into $\xi_{17}, \ldots, \xi_{24}$, the network should optimally result in the same decision, only based on the features' values. However, since the follow vehicle actions $\alpha_3, \ldots, \alpha_6$ depend on the order of the target cars in s_t, the weights for the last layer cannot be shared.

The network is structured such that target features of one car, for instance $\xi_1, \ldots, \xi_8$, are used as input to a sub-network with two layers. Each target car has a copy of this sub-network, resulting in them sharing weights ($\boldsymbol{W}_1$ and $\boldsymbol{W}_2$). The output of each sub-network is fed as input into a third hidden layer. The different sub-networks' outputs are multiplied with different weights $\boldsymbol{W}_{31}, \ldots, \boldsymbol{W}_{34}$ in order to distinguish different cars for different follow car actions. The ego features are also fed into layer 3 with its own weights $\boldsymbol{W}_2^{(ego)}$. The network structure is visualized in Figure 4.3. The neurons in layer $\boldsymbol{h}^{(3)}$ combine the inputs by adding them together as in Equation 4.2.

$$\boldsymbol{h}^{(3)} = g\left(\boldsymbol{W}_2^{(ego)}\,\boldsymbol{h}^{(ego)} + \sum_{c=1}^{4} \boldsymbol{W}_{3c}\,\boldsymbol{h}^{(2,c)}\right) \tag{4.2}$$

where $\boldsymbol{h}^{(2,c)}$ is the sub-network output for car c

The initial values of $\boldsymbol{W}_{31}, \ldots, \boldsymbol{W}_{34}, \boldsymbol{W}_2^{(ego)}$ must be smaller than their individual fan-in size based values, described in Section 2.1.6. The sum in Equation 4.2 can be seen as taking the output of each sub-network and concatenate them into a larger vector, together with the ego features. The input for layer $\boldsymbol{h}^{(3)}$ will be the concatenated vector with a size of $4 \cdot ||\boldsymbol{h}^{(2,c)}|| + ||\boldsymbol{h}^{(ego)}||$. This is the fan-in size n_{in} to use for weight initialization for weights $\boldsymbol{W}_{31}, \ldots, \boldsymbol{W}_{34}$ and $\boldsymbol{W}_2^{(ego)}$. The sub-network's weights will be updated using an average of the back-propagated gradients for all target cars' inputs.

4.6.3 Deep Recurrent Q-Network

As described in Section 4.1, the MDP model can cause issues for an autonomous driving agent. A DRQN is used to solve this, as mentioned in Section 1.4.2. As before, the network contains separate weights for target cars and ego features. The only difference from the network in Figure 4.3 is that an LSTM cell is added as the last hidden layer. The new network structure is shown in Figure 4.4.

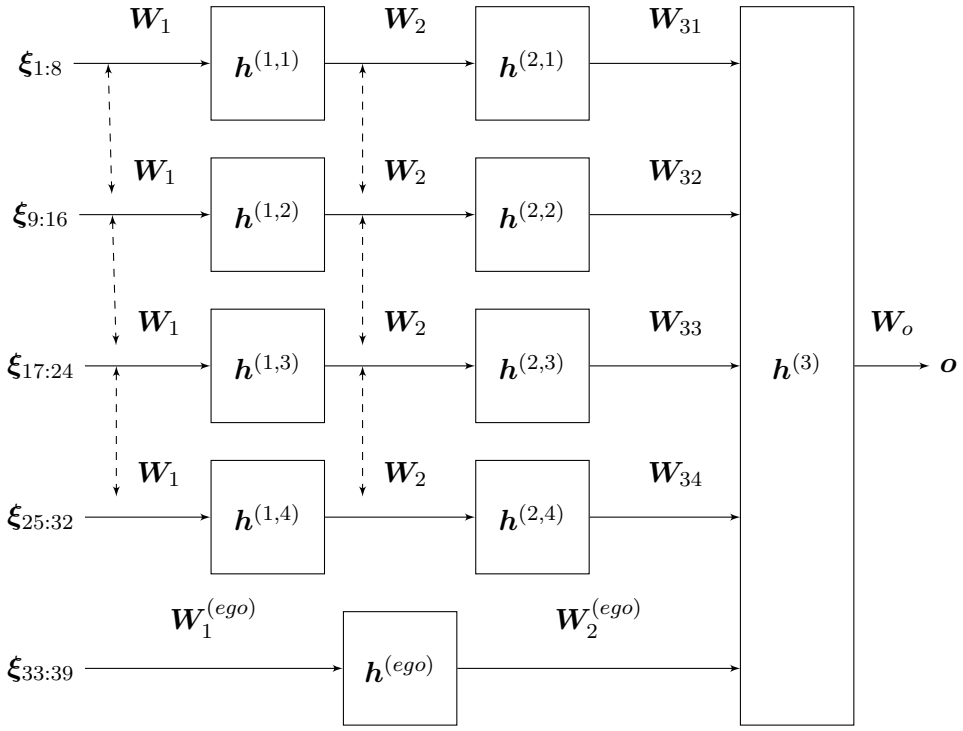


Figure 4.3: The network structure for the neural network with shared weights. A box represents a layer in the network, with the hidden layer $\boldsymbol{h}^{(\cdot)}$ denoted on each box. Arrows represents the feed forward data flow direction using the network weights $\boldsymbol{W}$ which are denoted next to each arrow. The weights $\boldsymbol{W}_1$ and $\boldsymbol{W}_2$ are shared for all the different cars. The dashed arrows represent that these weights are shared and identical.

4.6.4 Stochastic Sequence Length

As described in Section 2.2.6, a DRQN utilizes a sequence length l that determines the number of time steps to use when training the network. Stochastic sequence length uses a random sequence length l for each training update. With a DRQN, decisions are based on the full history of observations used for training. When exploring, this action could be correct for similar observations with different histories. Using stochastic sequence length, the agent could better utilize exploration, by sometimes training on shorter histories, while still being able to learn actions dependent on histories with longer sequences.

4.6.5 DQN with Multiple Observations

A DQN without a recurrent layer is able to take decisions based on a sequence of observations $o_{t-l+1}, \ldots, o_t$. This is done by stacking the observations in the network's input state, such that $s_t = (o_{t-l+1}, \ldots, o_t) = (\boldsymbol{\xi}_{t-l+1}, \ldots, \boldsymbol{\xi}_t)$. In contrast to a recurrent network, the sequence length l must be constant. In both training and prediction, l observations must be fed to the network, since the network has no internal memory.

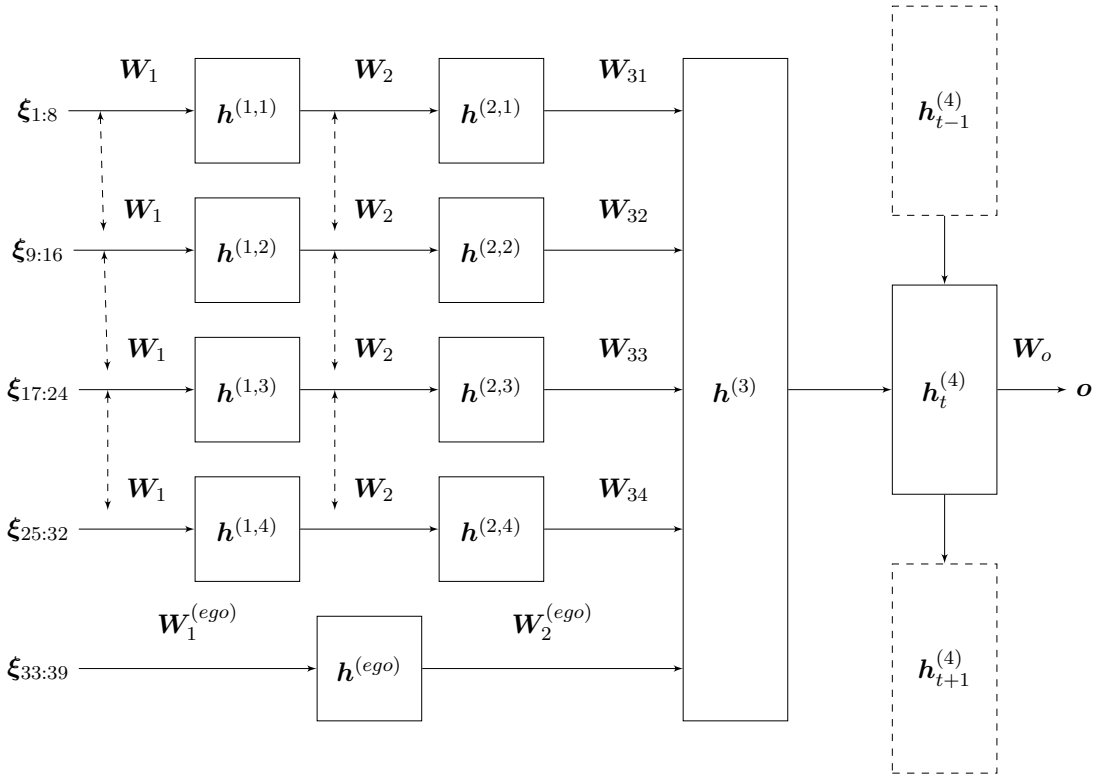


Figure 4.4: The network structure for the neural network with shared weights, and a recurrent LSTM layer before the output layer. The dashed boxes are LSTM layers for the previous and next timestep.

The network structure is similar to the recurrent network in Figure 4.4. Layers $\boldsymbol{h}^{(1,c)}$ to $\boldsymbol{h}^{(3)}$ computes an output for a single observation, resulting in a vector of size $||\boldsymbol{h}^{(3)}||$. This calculation is performed l times, once for each fed observation. The results of those l calculations are stacked and fed into a new hidden layer $\boldsymbol{h}^{(4)}$, replacing the recurrent layer in DRQN. The layer combines the observations of a full sequence, with $l||\boldsymbol{h}^{(3)}||$ number of input connections. Thus, the parameter size of this network grows with the number of observations. The network is presented in Figure 4.5.

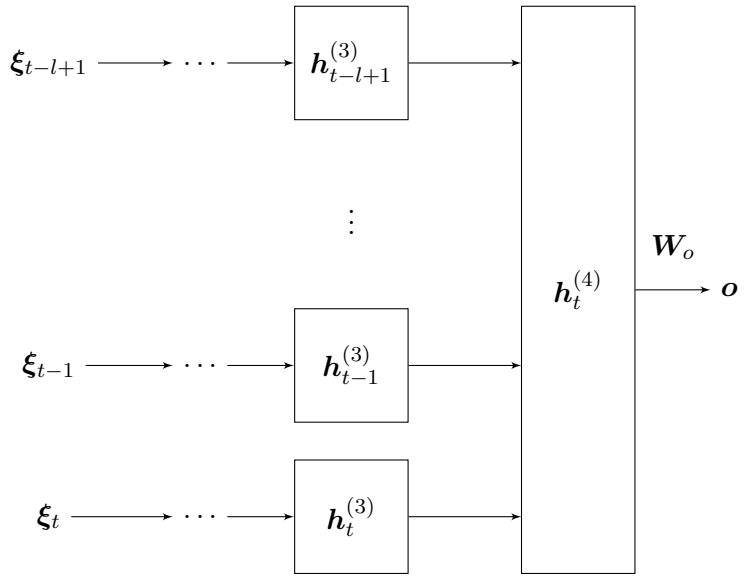


Figure 4.5: DQN stacked network structure, where l observations are used as input. Each observation is fed into a network with the same structure as the shared weights network (Figure 4.3), but $\boldsymbol{h}_t^{(4)}$ is added before the output layer, which merges all l observations. The horizontal dots is a shorthand for the entire shared weights structure.

5

Result

This chapter presents the performance of the Acceleration Regulator when it is used to comfortably drive a car, across multiple traffic crossings and with target cars with different drivers' behaviors, as described in Section 1.2 and 1.3. In Section 1.6, success rate was mentioned to be used for evaluation of the Acceleration Regulator. The first section in this chapter describes all the metrics used for evaluating the performance of the Acceleration Regulator. The next section presents the result for a simple crossing with one car. Thereafter, shared weights between cars is compared to a standard fully connected feed forward network. The following section shows how well the agent is able to generalize between different scenarios. The last section presents how well the ego agent can recognize other agents' behaviors, by using several observations, and thus improve the policy. The hyper-parameters used for each scenario, as well as more details about target cars, which are not explicitly stated next to the results, can be found in Appendix A and Appendix B.

5.1 Evaluation Metrics

In traditional machine learning, the training progress can be evaluated by using the loss function. With a DQN, the policy can improve even if the training loss is increasing [10]. For instance, predicting actions for unexplored states will typically increase loss, even if progression is made. Therefore, the training loss is not necessarily an indicator of the learning progress and several other measurements are necessary.

Three metrics were selected for measurement of a training session: success rate, average episodic reward and collision to fail ratio. With these metrics, the distribution between the three terminating states can be analyzed. The success graph presents the success to fail ratio averaged over the last 100 episodes. Both collisions and timeouts are considered as failures. Average episodic reward is the summed reward over a whole episode, then averaged over 100 episodes. The collision to fail ratio displays the ratio

between the number of collisions and unsuccessful episodes for the last 100 episodes. From the success rate and collision to fail ratio, a final collision rate is computed, which is the amount of episodes resulting in a collision, averaged over 100 episodes. In each graph, there is a faded colored line that show actual sampled values, and a thick line that acts as a trend line. The measurements from the training episodes are not completely representative for the performance of a policy, as a result of the random actions taken in the training algorithm. To evaluate a policy at any given time step, the agent is simulated on evaluation episodes, which mean that the network is not learning during evaluation, and the probability for random actions is 0. The graphs presented in this chapter are only using evaluation episodes. Every 300 episode, the trained network is evaluated over 300 evaluation episodes.

5.2 Single Car in Simple Crossing

In this scenario, the ego car and one target car are driving on two separate lanes in the simple crossing scenario presented in the upper left of Figure 1.2. The target car is assigned a *take way* agent. Both DQN and DRQN are tested and results are presented in figures 5.1 respectively. Since the scenario has only one predictable agent, the difference between a DQN and a DRQN network is negligible. The trend lines of both networks reach a success rate of about 99.5%, thus resulting in a collision rate at around 0.1%, after 10^4 training episodes.

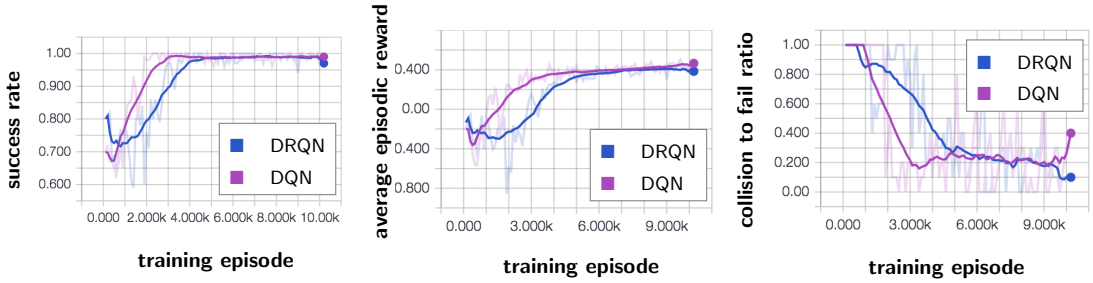


Figure 5.1: Graphs presenting the performances of a DQN and a DRQN agent learned to drive through the simple crossing scenario, with only one target car, over a total of 10^4 training episodes, using a single observation to make decisions. From left to right; the success rate, total episodic reward, and collision to fail ratio. Comparing the faded line, with raw sampled data, to the trend line shows that the variance is notably higher for collision to fail ratio compared to success rate and average episodic reward.

5.3 Shared Weights Between Cars

When introducing multiple cars in the scenario, the success rate converged considerably slower. Using the network structure that share weights between cars, as described in Section 4.6.2, significantly improved how fast the network converged compared to the fully connected DQN (Section 4.6.1). The test was performed using the simple crossing

scenario with a varied number of target cars, all using a take way agent. See Table B.1 in appendix for the complete list of scenarios. The success rate for the trend line converges to 96% after $4 \cdot 10^4$ episodes for shared weights, which is better than 82% after $4 \cdot 10^4$ episodes for the fully connected neural network structure. The evaluation graphs for both networks are shown in Figure 5.2. Note that the rewards are similar between both networks, which means that there is room for improving the reward function.

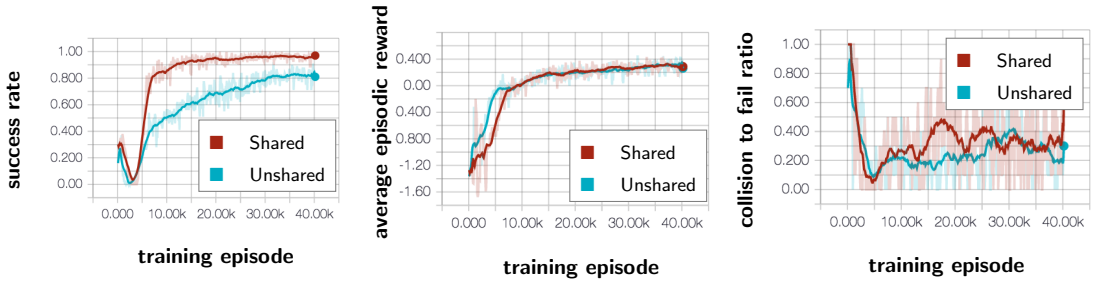


Figure 5.2: The figure shows that the success rate for a network with shared weights (brown line) converge faster than the fully connected network structure which do not share weights (turquoise line).

5.4 Generalize a Policy Across Different Scenarios

An agent is tested to learn on all crossing scenarios described in Section 1.4.1, to evaluate whether the same agent can generalize its policy across multiple scenarios. The complete list of scenarios is given in Table B.2 in the appendix. A network starting to learn from random weights did not perform well on these scenarios. Therefore, a network is first trained on only the simple crossing scenario with a varied number of cars for $8 \cdot 10^4$ episodes. Once the network solves a simple crossing, the learning process is then restarted, with the same network, but trained on all crossing scenarios, and with an empty experience memory. The probability for taking random actions starts at 30% instead of 100% since the network is not initialized with random weights. The network converge to a success rate of about 95% after training on all crossing scenarios, for $5 \cdot 10^4$ episodes. The corresponding graphs for the result is presented if Figure 5.3.

Figure 5.3: Graphs presenting the performance of a DRQN agent learned to drive through the all scenarios with 1 to 4 target cars.

5.5 Recognizing Behavior

The following sections describe how the different network structures performed when learning different target car behaviors, while driving in the simple crossing from Figure 1.2. Specifically, the results for DQNs with one or multiple observations, and DRQNs with different sequence lengths and observations used only for preparing the LSTM cell's state are shown. Table B.3 in the appendix lists the scenarios used for all tests in the following sections.

5.5.1 Network with Recurrent Layer

There was a significant performance improvement in using a DRQN instead of a DQN with a single observation. In other words, the environment for these scenarios is better modeled as a POMDP instead of an MDP, as described in Section 1.4.2, and the ego agent needs multiple observations in order to draw enough conclusions about other cars' behaviors. The trend lines for DRQN converges towards a success rate of around 97.2% and a 0.85% collision rate, compared to a success rate of 87.5% and a collision rate of 1.75% for DQN. The graphs for these two networks are shown in Figure 5.4.

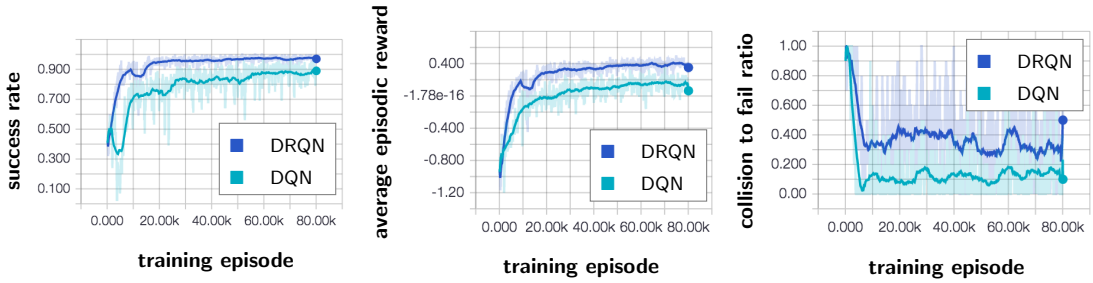


Figure 5.4: Graphs presenting the performance of a DRQN (dark blue) compared to a DQN with a single observation (turquoise), running on scenarios with cars that have different behaviors. When compared a DRQN succeeds in 3 out of 4 attempts, where a DQN fails.

5.5.2 DRQN or DQN with Multiple Observations

According to [12], a DQN with a stacked history of observations can perform as well as a DRQN, with an LSTM layer, in the aspect of Q-value accuracy [12]. The results support this statement. The DRQN was compared to the network described in Section 4.6.5, that uses a DQN with 4 stacked observations. Even though the policy's performances are equal, the computational complexity of a DQN with stacked observations is higher, which makes a DRQN preferable. The lower computational complexity in LSTM is a result of its internal memory, which lead to the network requiring only a single observation to compute an output [12, 27]. The agents' performances for the two networks are compared in Figure 5.5.

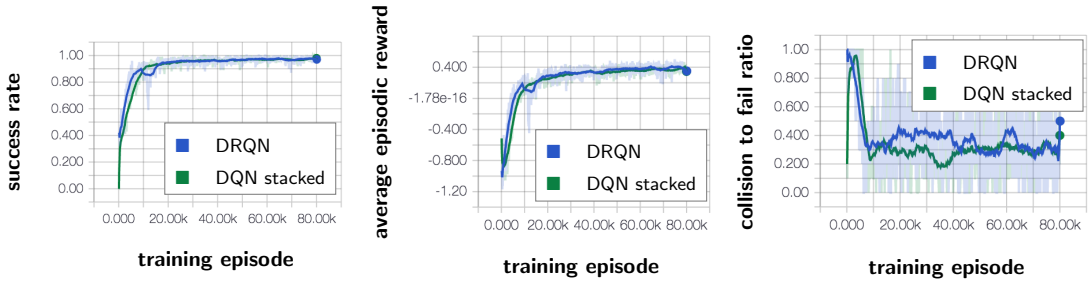


Figure 5.5: The performances of a DRQN (blue) and a DQN with 4 stacked observations (green) are very similar, as suggested by Hausknecht and Stone's theory.

5.5.3 Fixed or Stochastic Sequence Length

The DRQN network was run with different combinations of sequence lengths and number of preparing observations (as described in Section 2.2.6). Figure 5.6 shows the difference between using a fixed sequence length $l = 4$, trained on the last single observation, compared to a stochastic sequence length between 1 and 4, also trained on the last observation. The results show that the trend lines for the success rate between these two setups are negligible. From the trend lines, both networks converge to a success rate of 97.2%, resulting in a collision rate of about 0.85% for stochastic sequence length and 0.95% for fixed sequence length.

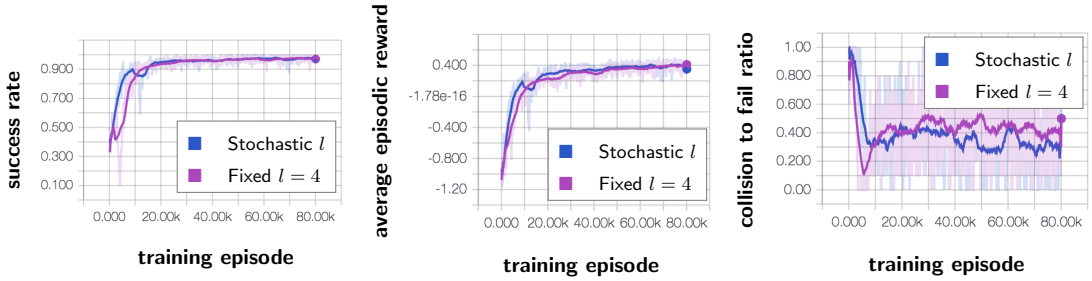


Figure 5.6: This is the result between DRQN using stochastic sequence length, randomized between 1 and 4 at each training step, compared to a fixed sequence length of 4. In both cases, only one observation was trained on.

The performance of having observations to build the LSTM's cell state before training was also measured. Given a sequence length of 4, where the algorithm uses observations (o_1, o_2, o_3, o_4), the DRQN in the last paragraph trains only on o_4, and o_1, o_2, o_3 are used to prepare the cell state. Another network was trained using a sequence length of 4, training on all 4 observations instead of only the last one. This makes the initial LSTM state zero in every training update. As can be seen in Figure 5.7, this network performed worse compared to the DRQN training on only the last observation.

The results show that the best performing network for the scenarios with multiple behaviors is a DRQN training only on the last observation. The learning algorithm used a stochastic sequence length and got a success rate of 97.2% and collision rate of 0.85%. Even if DQN with stacked observations achieved about the same performance, the DRQN

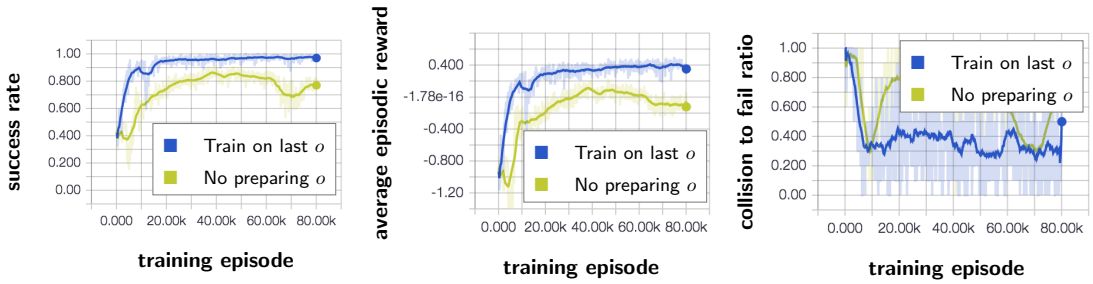


Figure 5.7: Compares a DRQN training on only the last observation (blue) to a DRQN training on all observations (light green), i.e. without any observations preparing the LSTM state.

approach shows other benefits, such as network complexity scaling independently of the number of observations.

6

Discussion

The results show a success rate of around 97.2% for recognizing behaviors, and 95% when driving in all four crossings. However, the ego car still collides too often. The ego car is limited to drive comfortably, meaning that in some cases, the ego agent is not allowed to break hard enough. In a complete system, a collision avoidance procedure, which does not have comfortability constraints, would need to take over the control to ensure a safe ride. However, the probability of this to happen must be much lower, which indicates the need for further improvements to be made before deployment. Even if the collision avoidance systems can take care of emergency situations, the policy learned by reinforcement learning should not have knowledge of these. The policy's goal should be to plan a safe and comfortable trajectory, without having the option to rely on emergency systems.

The choice of scenarios used for training the agent greatly influence the learned policy. For example, during testing, the ego agent sometimes did not learn to accelerate after it had stopped before a crossing. A possible explanation could be that the agent typically timed other cars very well, and thus rarely ended up in a situation where it had to fully stop before the crossing. It would not learn how to continue from that state, simply because it was rare for the agent to visit that state. In order to test if this was the case, a new configuration to the same scenario was added, where the ego car started in this state; standing still at the crossing. This increased the distribution of such states, and the agent successfully learned how to continue from a state where the ego car is standing still in front of the crossing.

Since the environment in this project is built for reinforcement learning in mind, it might be biased in its construction, and this could be a problem. Whenever a target car reaches the end of its lane, it is immediately put at the start of its lane. This car is considered a new car, since the old one left the observable area for the ego car, and a new car entered the observable area at the start of the lane. This might be one such biased artifact, and could cause unwanted artifacts to the learning algorithm, as the

policy might exploit that new cars always appear when old ones disappear.

Running reinforcement learning for autonomous driving in a simulator is very fast and cost effective, compared to real traffic. However, in real traffic, where sensors are not ideal, cars might pop up, or disappear anywhere, while our trained agent learns that new vehicles only pop up at the start of the lane. This could possibly have been avoided by, for instance, stochastically spawning cars or adding noise to the input in the traffic simulator. Continuing, actual human behavior probably differ from the programmed behaviors added to the simulator. It might even be so, that the wanted behavior of those agents could be the policy that the ego agent is trying to learn. The difference between the simulator and real world traffic should therefore be minimized, for deployment in real traffic.

Reducing the action space for the high-level controller by choosing discrete actions could be an advantage, as described in Chapter 4. Though, actions that can be performed by the ego agent are limited by how the actions are implemented. To include more freedom, more actions with different strengths must be added. Also, the ego agent often learned to quickly switch between the two actions *take way* and *give way*, for keeping a constant acceleration. This introduces jerk. A policy gradient or actor-critic method could instead control the acceleration directly, and therefore have more control over the vehicle.

Initially, the Acceleration Regulator consisted of a high-level only choosing between two discrete actions, which either increased or decreased the ego car's acceleration linearly, instead of using the implemented STGs. This led to high jerk. A zero-jerk action was therefore introduced, used to maintain the current acceleration. By including this action, however, the agent did not learn how to drive. The reason for this is unclear, and further testing is needed to make a conclusion.

The definition of the reward function plays a large role in what agents attempt to learn. However, the policy is typically not improved by tuning the reward function, as it only tells what goal the agent should achieve. For example; the reward function used in this thesis project attempts to solve the solution as fast as possible. An alternative solution is to punish inactivity, by punishing low velocities. This reward function teaches the agent that it is good to never stop. It will be good at timing the gaps between cars, but when there is no gap, it would rather collide than stop in front of the crossing. If the agent does not learn, or learns to solve the wrong problem, as in the example, then adjusting the reward function is correct.

When learning on multiple traffic crossings (Section 5.4), a network with random initial weights did not learn how to drive. However, when the agent was initiated with an already learned, simpler policy, the agent could improve to actually learn the more difficult problem. To not start from zero, but to take advantage of an already learned policy, is a good idea. This leads to another technique called imitation learning [8], where a policy is initiated by copying parts of another agent's behavior before starting the reinforcement learning loop.

6.1 Future Work

To be able to apply this solution in traffic, the agent must be able to control the lateral position. There might be situations where the agent need to position the car at any side of the lane in order to drive around other vehicles or to better take advantage of gaps. Also, the proposed solution could be extended to work on other areas than crossings. One such area could be to learn how to use both longitudinal and lateral control in order to change from one lane to another.

To make reinforcement learning safer, Shalev-Shwartz et al. propose a method of decreasing the (negative) reward r for an accident if the probability p of that accident happening is low, such that $r < -\frac{1}{p}$ [8]. The expected reward can otherwise become insignificant on very unlikely accidents. The solution, however, introduces high variance. Another method suggested by Lipton et al., called intrinsic fear, is to let a second neural network learn how dangerous a state $s \in \mathcal{S}$ is. That network can be used to avoid ending up in dangerous situations by making the agent fear them. The learning algorithm then does not only maximize reward, it also minimizes the danger level of a state [40].

In this thesis, features used in the state were manually selected. The learning algorithm used, DRQN, can also make use of lower level of observation data, such as raw sampled data points of the road. A solution using such features could eliminate manual feature selection. A technique applicable when using lower level features, presented in [37], is to force the agent to learn specific selected high level features, needed to take correct actions during training, such as known traffic rules. The agent is guided to learn and identify this information, which can improve performance [37].

7

Conclusion

In this thesis, Deep Q-Learning has been implemented in the domain of autonomous vehicle control. The goal of the ego agent is to drive through different crossings, by adjusting longitudinal acceleration using STGs. High-level features computed from a coordinate system are used as observations in order to achieve this. These features were divided into target car features, fed once for each target car, and ego features, dependent only on the ego car. Convergence of the neural network was shown to be improved by sharing weights between the first layers to which the target car features are fed, compared to a fully connected neural network structure.

The results also show that trained policies can generalize over different types of traffic crossings and driver behaviors. The same learned policy is able to respond to target cars' actions and handle traffic scenarios with a varied number of cars and different types of crossings, without knowing traffic rules or the type of crossing it drives in. Multiple observations are needed in order to recognize cars' behaviors, and can be utilized by for instance a DRQN. A DQN with stacked observations is considered to have comparable performance, but it requires a fixed sequence length and more parameters than the DRQN. A stochastic sequence length for DRQN was compared to a fixed sequence length. However, results indicate similar performance between the two.

The results are still limited to the tested traffic scenarios and driver behaviors, and expanding the domain beyond a simulator is a natural next step. The selected features are also limited to crossings. Utilization of lower level data could extend the solution beyond crossings without further handcrafting of additional features.

Bibliography

[1] NHTSA, "Distracted driving 2013," *NHTSA's National Center for Statistics and Analysis*, 2015.

[2] M. Distner, M. Bengtsson, T. Broberg, and L. Jakobsson, "City safety—a system addressing rear-end collisions at low speeds," in *Proc. 21st International Technical Conference on the Enhanced Safety of Vehicles*, no. 09-0371, 2009.

[3] W. Kabbaj, "What a driverless world could look like," *TED@UPS Atlanta*, 2016.

[4] Google, "Google self-driving car project, on the road," https://www.google.com/selfdrivingcar/where/, 2016, accessed: 2016-11-25.

[5] DARPA, "Darpa urban challenge," http://archive.darpa.mil/grandchallenge/, 2007, accessed: 2017-06-26.

[6] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[7] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[8] S. Shalev-Shwartz, S. Shammah, and A. Shashua, "Safe, multi-agent, reinforcement learning for autonomous driving," *CoRR*, vol. abs/1610.03295, 2016.

[9] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang *et al.*, "End to end learning for self-driving cars," *arXiv preprint arXiv:1604.07316*, 2016.

[10] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[11] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *CoRR*, vol. abs/1509.02971, 2015. [Online]. Available: http://arxiv.org/abs/1509.02971

[12] M. Hausknecht and P. Stone, "Deep recurrent q-learning for partially observable mdps," *arXiv preprint arXiv:1507.06527*, 2015.

[13] I. D. Jacobson, L. G. Richards, and A. R. Kuhlthau, "Models of human comfort in vehicle environments," *HUMAN FACTORS IN TRANSPORT RESEARCH EDITED BY DJ OBORNE, JA LEVIS*, vol. 2, 1980.

[14] S. Shalev-Shwartz, N. Ben-Zrihem, A. Cohen, and A. Shashua, "Long-term planning by short-term prediction," *arXiv preprint arXiv:1602.01580*, 2016.

[15] Z. C. Lipton, J. Gao, L. Li, J. Chen, and L. Deng, "Combating reinforcement learning's sisyphean curse with intrinsic fear," *arXiv preprint arXiv:1611.01211*, 2017.

[16] M. Wahde, *Biologically inspired optimization methods: an introduction*. WIT press, 2008.

[17] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bulletin of Mathematical Biology*, vol. 5, pp. 115–133, 1943, classics of Theoretical Biology.

[18] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[19] K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural networks*, vol. 4, no. 2, pp. 251–257, 1991.

[20] S. Ruder, "An overview of gradient descent optimization algorithms," *CoRR*, vol. abs/1609.04747, 2016.

[21] M. A. Nielsen, "Neural networks and deep learning," *Determination Press*, 2015.

[22] C. M. Bishop, "Pattern recognition," *Machine Learning*, vol. 128, pp. 1–58, 2006.

[23] J. Sjöberg and L. Ljung, "Overtraining, regularization and searching for a minimum, with application to neural networks," *International Journal of Control*, vol. 62, no. 6, pp. 1391–1407, 1995.

[24] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks." in *Aistats*, vol. 9, 2010, pp. 249–256.

[25] P. J. Werbos, "Backpropagation through time: what it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

[26] R. J. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," *Neural computation*, vol. 1, no. 2, pp. 270–280, 1989.

[27] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[28] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *arXiv preprint arXiv:1207.0580*, 2012.

[29] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting." *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[30] S. Bhatnagar, D. Precup, D. Silver, R. S. Sutton, H. R. Maei, and C. Szepesvári, "Convergent temporal-difference learning with arbitrary smooth function approximation," in *Advances in Neural Information Processing Systems*, 2009, pp. 1204–1212.

[31] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of Artificial Intelligence Research*, vol. 4, pp. 237–285, 1996.

[32] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press Cambridge, 1998, vol. 1, no. 1.

[33] H. V. Hasselt, A. Guez, M. Hessel, and D. Silver, "Learning functions across many orders of magnitudes," *CoRR*, vol. abs/1602.07714, 2016.

[34] J. N. Tsitsiklis, B. Van Roy *et al.*, "An analysis of temporal-difference learning with function approximation," *IEEE transactions on automatic control*, vol. 42, no. 5, pp. 674–690, 1997.

[35] L.-J. Lin, "Self-improving reactive agents based on reinforcement learning, planning and teaching," *Machine learning*, vol. 8, no. 3-4, pp. 293–321, 1992.

[36] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[37] G. Lample and D. S. Chaplot, "Playing FPS games with deep reinforcement learning," *CoRR*, vol. abs/1609.05521, 2016.

[38] R. S. Sutton, D. A. McAllester, S. P. Singh, Y. Mansour *et al.*, "Policy gradient methods for reinforcement learning with function approximation." in *NIPS*, vol. 99, 1999, pp. 1057–1063.

[39] V. R. Konda and J. N. Tsitsiklis, "Actor-critic algorithms." in *NIPS*, vol. 13, 1999, pp. 1008–1014.

[40] Z. C. Lipton, A. Kumar, J. Gao, L. Li, and L. Deng, "Combating deep reinforcement learning's sisyphean curse with reinforcement learning," *arXiv preprint arXiv:1611.01211*, 2016.

Appendices

A

Hyper parameters

This appendix states the hyper parameters used during training. The parameters presented in table A.1 are the default parameters, used for all training sessions presented in the Result chapter, unless where other values are specified. Figure A.1 presents the number of neurons existing in each layer of the DRQN and DQN network. Figure A.2 shows the number of neurons used in the tested fully connected network that does not contain shared weights. When the experience memory is full, a random experience is replaced with the new experience.

Name	Notation	Value
Exploration	ϵ	$\max\left(0.1, e^{\frac{\ln(0.5)i}{2000}}\right)$; where i is the episode counter
Discount Factor	γ	0.99
Car Size	$\boldsymbol{s}^{(c)}$	$[2,4]^T$
Max Lateral Acceleration	a_{lat}	1.5
Distance to Intersection	d_{margin}	1
Distance to Front Car	d_{offset}	6
Sight Range	$p_{\max}$	50
Max Speed	$v_{\max}$	30
Max Acceleration	$a_{\max}$	5
Max Jerk	$j_{\max}$	3
Learning Rate	η	10^{-3}
Batch Size	B	64
Update Interval	Δt	$\frac{1000}{30}$
Experience Memory Size	N	10^6
Dropout Probability	p	0.75
Sequence Length	l	$l \sim U[1,4]; l \in \mathbb{N}$; Randomized each update.
Build LSTM state	h	$l-1$
Target Network Parameter	τ	0.99

Table A.1: Default values used for simulation in the scenarios presented in the Result chapter.

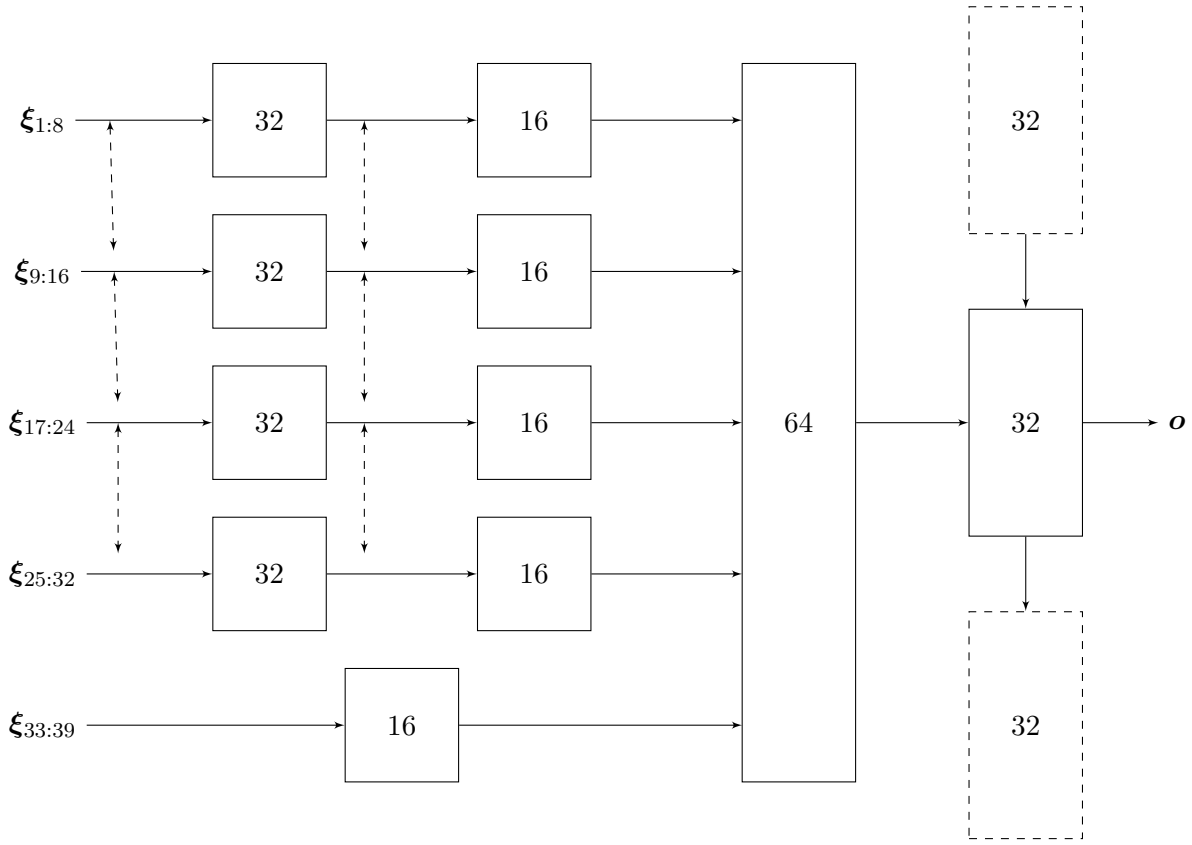


Figure A.1: The number of neurons used at each layer of the network. The last layer is the LSTM layer with 32 neurons. The same neuron count was used in all tests presented in the Result chapter, except the fully connected network, which used an entierly different structure, presented in Figure A.2.

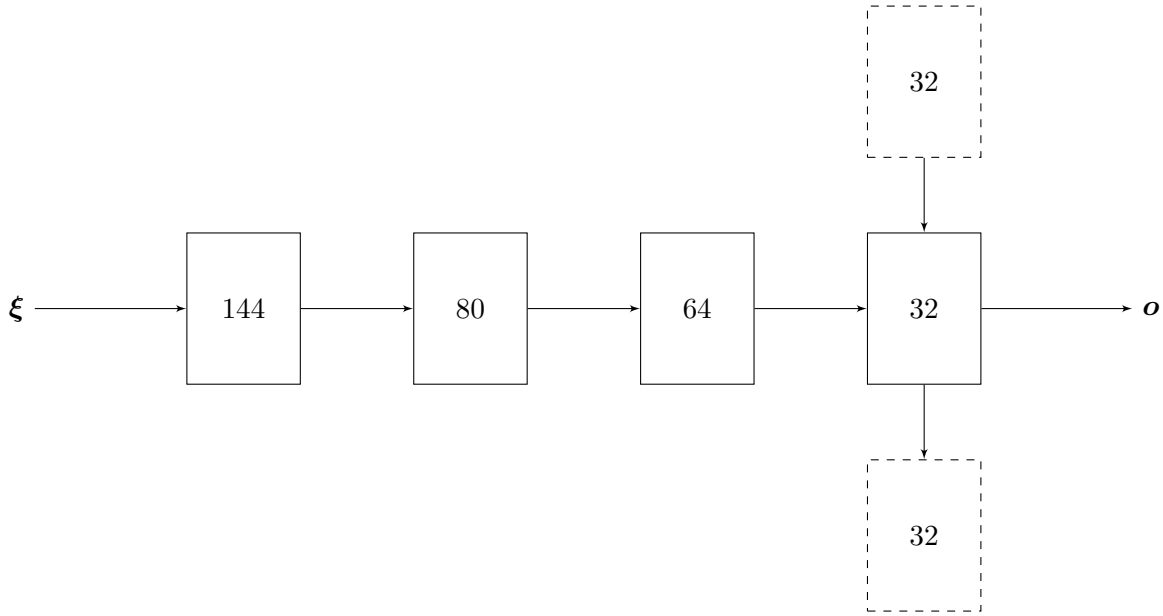


Figure A.2: The number of neurons used at each layer in the fully connected network. The last layer with 32 neurons is an LSTM layer.

B

Scenarios

This appendix chapter describes the scenarios that the agent is trained on. Each scenario is presented in Figure 1.2.

All scenarios used for testing generalization over multiple crossings are listed in table B.2. The target cars in these scenarios are assigned a take way agent. A subset of these scenarios, are used when comparing the performance of the shared network structure to the fully connected network structure. These scenarios is presented in table B.1. All scenarios used for testing generalization over multiple behaviors are listed in table B.3.

Crossing	**Cars**	**Description**
Simple Crossing	1	
Simple Crossing	1	Ego car starts still at crossing.
Simple Crossing	2	
Simple Crossing	2	Ego car must stop for crossing cars.
Simple Crossing	3	
Simple Crossing	3	Ego car must stop for crossing cars.
Simple Crossing	4	
Simple Crossing	4	Ego car must stop for crossing cars.
Simple Crossing	4	Ego car starts still at crossing.

Table B.1: Scenarios used when comparing shared network structure to a fully connected network structure.

Crossing	**Cars**	**Description**
Simple Crossing	1	
Simple Crossing	1	Ego car starts still at crossing.
Simple Crossing	2	
Simple Crossing	2	Ego car must stop for crossing cars.
Simple Crossing	3	
Simple Crossing	3	Ego car must stop for crossing cars.
Simple Crossing	4	
Simple Crossing	4	Ego car must stop for crossing cars.
Simple Crossing	4	Ego car starts still at crossing.
Two Separated Crossings	4	Two cars on each crossing lane.
Turn	1	One car on crossing lane.
Turn	2	One car on each crossing/merging lane.
Turn	2	Two cars on merging lane.
Turn	3	One car on crossing lane, two on merging.
Two-Lane Crossing	4	Two cars on each crossing lane.

Table B.2: Scenarios used for testing multiple types of crossings.

Crossing	**Cars**	**Target Car Behaviours / Description**
Simple Crossing	1	Take way.
Simple Crossing	1	Give way.
Simple Crossing	1	Trained to give way.
Simple Crossing	1	Take way. Ego car starts still at crossing.
Simple Crossing	2	Take way.
Simple Crossing	2	Take way. Ego car must stop for crossing cars.
Simple Crossing	2	Cautious agents.
Simple Crossing	2	Give way.
Simple Crossing	3	Take way.
Simple Crossing	3	Take way. Ego car must stop for crossing cars.
Simple Crossing	4	Take way.
Simple Crossing	4	Take way. Ego car must stop for crossing cars.
Simple Crossing	4	Take way. Ego car starts still at crossing.
Simple Crossing	4	One cautious agent, three take way.

Table B.3: Scenarios used for testing multiple behaviors. To make the problem fair, the driving behaviors and properties such as speeds and positions of target cars are similar on average at the start of each episode.