CHALMERS
UNIVERSITY OF TECHNOLOGY

Neural Networks for Collision Avoidance

Preliminary Investigations of Training Neural Networks Using Deep Q-learning and Genetic Algorithms for Active Safety Functions

Master's thesis in Complex Adaptive Systems and Applied Physics

Jonathan Leiditz Thorsson

Olof Steinert

Department of Signals and Systems
Chalmers University of Technology
Gothenburg, Sweden 2016

Master's thesis 2016:XX

Neural Networks for Collision Avoidance

Preliminary Investigations of Training Neural Networks Using Deep Q-learning and Genetic Algorithms for Active Safety Functions

JONATHAN LEIDITZ THORSSON
OLOF STEINERT

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Signals and Systems
Signal Processing and Biomedical Engineering
Signal Processing
Chalmers University of Technology
Gothenburg, Sweden 2016

Neural Networks for Collision Avoidance
Preliminary Investigations of Training Neural Networks Using Deep Q-learning and Genetic Algorithms for Active Safety Functions
JONATHAN LEIDITZ THORSSON
OLOF STEINERT

Supervisor: Rickard Nilsson, Volvo Car Corporation
Examiner: Karl Granström, Department of Signals and Systems
Master's Thesis 2016:XX
Department of Signals and Systems
Signal Processing and Biomedical Engineering
Signal Processing
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Printed by [Name of printing company]
Gothenburg, Sweden 2016

Neural Networks for Collision Avoidance
Preliminary Investigations of Training Neural Networks Using Deep Q-learning and Genetic Algorithms for Active Safety Functions
JONATHAN LEIDITZ THORSSON
OLOF STEINERT
Department of Signals and Systems
Chalmers University of Technology

Abstract

Artificial neural networks, and deep neural networks in particular, have recently achieved very promising results in a wide range of applications. The Volvo Car Corporation is interested in knowing how well neural network based methods can used to design a good collision avoidance system. Therefore, during this thesis, preliminary investigations of active safety functions based on artificial neural networks were conducted. The investigations were limited to rear-end collisions.

A simulation environment developed by Volvo Car Corporation was used to construct traffic scenarios and simulate corresponding sensory data of the car. The input to the neural networks consisted of feature vector observations obtained from the simulation environment.

Two different approaches were examined during this thesis. The first approach was based on deep Q-learning and inspired by the algorithm developed by Google DeepMind, which was able to achieve superhuman performance on a diverse range of Atari 2600 games. Our findings indicate that it is difficult to find a good way to promote desired braking policies and that there are concerns related to the stability of the algorithm. Although the trained network can handle a few simple scenarios, it generalises poorly on test sets. Likely, larger networks and longer training times would lead to better generalisation but the associated computational cost has limited these investigations.

In the second method a genetic algorithm was used to train neural networks. The genetic algorithm simplified the problem of finding a good measure of performance. This is probably the main reason as to why this method outperformed the deep Q-learning approach.

Keywords: artificial neural networks, collision avoidance, deep Q-learning, genetic algorithm, rear-end collision

Acknowledgements

First of all we would like to thank our supervisor Rickard Nilsson for valuable discussions, bringing in interesting ideas to the project and supporting us in the process. We would also like to thank our examiner Karl Granström for helping out with all the formal work as well as discussing ideas related to the project. Overall the Signal Processing group at the department of Signals and Systems has been very supportive during the entire thesis. During the course of the thesis we have had several interesting and worthwhile discussions with Mattias Wahde at the Adaptive Systems research group at Chalmers University of Technology. Lastly, we would like to thank friends and family for their support.

Jonathan Leiditz Thorsson, Olof Steinert, Gothenburg, July 2016

Contents

List of Figures **xiii**

List of Tables **xv**

1 Introduction **1**
1.1 Background . . . 1
1.2 Purpose . . . 1
1.3 Objective . . . 1
1.4 Scope . . . 2

2 Theory **3**
2.1 Neural networks . . . 3
2.1.1 Artificial Neurons . . . 3
2.1.2 Feedforward Neural Networks . . . 4
2.1.3 Activation Functions . . . 5
2.1.4 Weight initialisation . . . 6
2.1.5 Optimising Neural Networks . . . 6
2.1.5.1 Training Using Gradient Descent . . . 6
2.1.5.2 Error Backpropagation . . . 6
2.1.5.3 Problems With Training Deep Networks . . . 8
2.1.5.4 Improved Training Methods . . . 8
2.1.5.4.1 The Momentum Algorithm . . . 8
2.1.5.4.2 Nesterov Momentum . . . 9
2.1.5.4.3 Adaptive Learning Rates . . . 9
2.1.5.4.4 RMSProp . . . 9
2.1.5.4.5 Adam . . . 9
2.2 Reinforcement Learning . . . 10
2.2.1 Markov Decision Process . . . 10
2.2.2 Q-learning . . . 11
2.2.2.1 Action-value Function . . . 11
2.2.2.2 The Q-learning Algorithm . . . 11
2.2.3 Exploration . . . 11
2.3 Deep Q-Learning . . . 11
2.3.1 Stability Issues . . . 12
2.3.2 The Deep Q-learning Algorithm . . . 12
2.3.3 Improvements to the Deep Q-learning Algorithm . . . 13

2.3.3.1 Double Q-learning . . . 13
2.3.3.2 Prioritised Experience Replay . . . 13
2.3.4 Curriculum Learning . . . 14
2.4 Genetic Algorithm . . . 14
2.4.1 Fitness . . . 15
2.4.2 Selection . . . 15
2.4.2.1 Roulette-wheel Selection . . . 15
2.4.2.2 Tournament Selection . . . 15
2.4.3 Crossover . . . 15
2.4.4 Mutation . . . 16
2.4.5 Elitism . . . 16
2.4.6 Standard Genetic Algorithm . . . 16

3 Methods **19**
3.1 Implementation Strategy . . . 19
3.2 Simulation Model . . . 20
3.3 Implementation Details . . . 22
3.4 Simple Problem: Grid World . . . 23
3.5 Training a DQN Agent for Collision Avoidance . . . 23
3.5.1 Parameter Sweeps . . . 25
3.5.2 Training on Multiple Rear-end Scenarios . . . 25
3.5.3 Curriculum Learning . . . 26
3.5.4 Reward Function . . . 27
3.5.5 Prioritised Experience Replay . . . 27
3.6 Training a Neural Network With a Genetic Algorithm for Collision Avoidance . . . 27
3.7 Measuring Performance . . . 29
3.8 Benchmarking . . . 30
3.9 Noisy Data . . . 30

4 Results **31**
4.1 Verifying the Implementation on Grid World . . . 31
4.2 Training a DQN Agent for Collision Avoidance . . . 33
4.2.1 Parameter Sweeps . . . 33
4.2.1.1 Stability of the Algorithm . . . 34
4.2.1.2 Network size . . . 36
4.2.1.3 Optimisation Algorithm . . . 38
4.2.1.4 Reward Function . . . 39
4.2.1.5 Parameters with Indistinct Impact . . . 40
4.2.2 Improvements of the Algorithm . . . 41
4.2.3 Adding Training Data for Improved Generalisation . . . 42
4.2.3.1 Curriculum Learning . . . 43
4.2.4 Generalisation . . . 43
4.3 Genetic Algorithm . . . 44
4.4 Benchmark . . . 46
4.5 Noisy Data . . . 48

5 Discussion **51**
5.1 Future Work 54

6 Conclusion **55**

List of Figures

2.1 An FFNN with inputs $x_1, \ldots, x_D$, hidden nodes $z_1, \ldots, z_M$ and outputs $y_1, \ldots, y_K$. The blue nodes represent biases. 4
2.2 An illustration of single-point crossover. The two chromosomes coloured white and gray respectively are crossed at the chosen point represented by the blue line to form two new chromosomes. 16

3.1 The figure shows a snapshot of a simulated traffic scenario seen from a bird's view. The red vehicle (host) is driving behind the green vehicle (target) on a straight road visualised as a blue line. 21
3.2 A flowchart describing the control loop. A traffic scenario was defined and fed to the simulation model. Information describing the state of the host vehicle and its environment could be obtained at every 0.02 s from the simulation model. This information was used as input to a neural network. Based on the input the neural network gave as output a decision of whether or not emergency braking was needed. In the case of emergency braking the host vehicle trajectory was recalculated. The main script was used to monitor the process. . 22
3.3 An illustration of the game Grid World. The player is represented by the stick figure and the goal is to find the shortest path to the diamond without falling into the fire pit. The black box represents a wall. There is also a wall around the entire grid. 23
3.4 The figure shows the genetic algorithm optimisation loop. 29

4.1 Optimal policies for the game Grid World given two different reward functions. 32
4.2 Score per episode and average predicted Q-values during 3000 training episodes. 32
4.3 An example of how the Q-network failed to converge when forcing the velocity of the host vehicle to zero. 33
4.4 Figure illustrating the stability of the algorithm. 35
4.5 Four figures showing the braking policy during training. Initially the braking policy is random, but towards the end it has stabilised. . . . 35
4.6 Four figures showing average predicted Q-values during training. It can be seen that a large network is required for convergence. 36
4.7 Four figures showing the score per episode. A larger network shows better increase in score during training. 37

4.8 Four figures showing average predicted Q-values during training with different optimisation algorithm. The choice of optimisation algorithm has a large impact on the results. 38
4.9 Four figures showing the moving average score using different optimisation algorithms. Adam has the steadiest increase in score per episode and training using stochastic gradient descent is actually worse than the initialisation. 39
4.10 Four figures showing what happens to the moving average of score per episode when using different combinations of prioritised experience replay and double DQN. 41
4.11 Four figures showing the learnt policies when using different combinations of prioritised experience replay and double DQN. 42
4.12 Comparison of the moving average of score per episode with and without curriculum learning. 43
4.13 A histogram showing the relative impact velocity. The DQN agent shifts the bars to the left meaning that the impact velocity on average was decreased. 44
4.14 A histogram of the minimum distance to the target vehicle after an intervention. 44
4.15 The training and validation fitness as functions of generation. The x-axis measures the number of training epochs and the y-axis the fitness. 45
4.16 A histogram of the minimum relative distance to the target during braking. Collisions are left out. 46
4.17 A histogram showing the reduction of relative impact velocity. The relative impact velocity was reduced in 98 % of the collisions. On average the relative impact velocity was reduced by 58 %. 46
4.18 A histogram of the minimum distance to the target vehicle after an intervention. 47
4.19 A histogram of the time difference for the first brake request. 47
4.20 A histogram showing the minimum distance to the target vehicle after an intervention. The blue distribution was obtained with perturbed input data, and the green distribution without any misreadings. The network was trained with deep Q-learning. 48
4.21 A histogram showing the minimum distance to the target vehicle after an intervention. The blue distribution was obtained with perturbed input data, and the green distribution without any misreadings. The network was trained with a GA. 49

List of Tables

3.1 A list of all inputs to the neural network. 24
3.2 A table of all parameters used for the network and the DQN. 26

4.1 Default parameter settings during the parameter sweeps on a training set of one single scenario. 34
4.2 The different rewards functions used in the performance comparison between 14 different reward functions. 40

1
Introduction

1.1 Background

The Volvo Car Corporation's current collision avoidance system is based on the accurate modelling of vehicle dynamics identify and act in dangerous situations. The Volvo Car Corporation is interested to know how well a neural network based system can realise a good collision avoidance system. The current collision avoidance system is active in some intersection scenarios such as left turn across path but can also prevent rear-end collisions [1]. However, the system is not designed to handle all possible situations. This is one of the reasons why the Volvo Car Corporation would like to investigate how well a neural network based approach performs as a collision avoidance system. Given a sufficient amount of training data deep learning techniques have achieved impressive results in a wide range of applications. Can a collision avoidance system based on an artificial neural network (ANN) learn to detect and assess threats?

1.2 Purpose

The purpose of the project was to investigate if deep learning for collision avoidance functionality is a viable path forward. The intended solution approach was heavily influenced by the result of Google DeepMind. Google DeepMind was able to achieve superhuman performance on a diverse range of Atari 2600 games using a combination of deep learning and reinforcement learning. In addition, the Japanese company Preferred Networks' has managed to simulate self driving cars using a similar technique [2].

1.3 Objective

The objective was to create and train ANNs for preliminary investigations of active safety functions. In order to accomplish this a simulation environment had to be created where ANNs could be trained using data similar to what is available in the real cars. One of the most important aspects to investigate was the networks' ability to generalise. Only a limited set of traffic scenarios could be used for training and it was therefore important to examine how well the networks generalise to traffic scenarios not seen during training.

Exactly how neural networks should be employed in a collision avoidance system is far from obvious. For this reason the first objective was to come up with a suitable strategy for this. There are several different types of neural network and investigate all of them would have been too time-consuming. Therefore a suitable choice had to be based on a literature review. However, the size and layout of the network had to be found from experimental investigations.

Google DeepMind published an article in February 2015 where an agent learnt control policies in Atari 2600 games using a technique called deep Q-learning [3]. Some adaptation to the original algorithm have since then been suggested [4], [5]. The objective was to implement the deep Q-learning algorithm and the suggested adaptations to see if the technique is suitable for learning braking policies.

Genetic algorithms can be used to train neural networks. Such an approach has previously been successfully used to represent driving strategies [6]. The objective was to see if this approach is more promising than the deep Q-learning technique.

Since it is important that a collision avoidance system can handle real noisy input data an additional objective was to investigate the performance on perturbed data.

1.4 Scope

During this study autonomous emergency braking systems have been developed and studied. Such systems should help to avoid collisions by automatically applying the brakes in emergency situations. Even more important is that these systems do not apply the brakes in situations that do not require interventions.

It should be pointed out that the intention never was to develop any control mechanisms to accelerate the vehicle after an intervention or at any time to perform assisted steering of the vehicle. In addition the system should not control the level of braking. It should only decide when the brakes should be applied and not how they are applied.

Ideally a collision avoidance system can detect and avoid all types of imminent collisions. There are, however, numerous ways a collision can occur and to reduce the complexity of the problem only one common type of collisions called rear-end collision was considered during this project. A rear-end collision is a type of collision wherein a vehicle (the host) crashes into the vehicle (the target) in front of it. It was assumed that the traffic scenarios only included two vehicles.

2
Theory

This chapter seeks to provide a theoretical background to important concepts of the project. Artificial neural networks (ANNs) are central for understanding and will therefore be briefly presented. The review focuses on one type of ANNs, namely the feed forward neural network (FFNN).

In addition, reinforcement learning is introduced and a technique combining ANNs and reinforcement learning is described.

Lastly, the components of a genetic algorithm (GA) are described. It is explained how FFNNs can be trained with a GA. There will also be some discussions of problems associated with the different techniques.

2.1 Neural networks

ANNs are inspired by structures and properties of biological neural networks. Biological neural networks contains a large number of interconnected elements that operate together to perform complex tasks [7]. However, an ANN does not seek to be an exact copy of a biological brain. In the context of pattern recognition, a realistic biological model would impose completely unnecessary constraints [8].

There are several types of ANNs being used in different applications. The FFNN is the most common type of ANN. An FFNN only contains forward-pointing connections. Other types, such as a recurrent neural network (RNN) have connections pointing both forward and backward (feedback) to allow for a dynamic time dependent behaviour. Other more complex architectures, e.g. convolutional neural networks (CNNs) have been very successful in image recognition applications [9]. It is worth noticing that an ANN is not an optimisation method but a structure to which an optimisation algorithm can be applied effectively. The goal of training an ANN is to find an optimal set of weights to reproduce a given set of data. Several training methods are available and a few of these will be discussed later in this chapter.

2.1.1 Artificial Neurons

An ANN is composed of computational units called neurons. Given a set of D inputs $x_1, x_2, \ldots, x_D$ we construct an activation as a weighted sum of the inputs as follows

$$a = \sum_{j=1}^{D} w_i x_i + b \tag{2.1}$$

where w_i are the network weights, x_i the inputs and b the bias term. The output of the neuron is then obtained by applying a differentiable, non-linear function h to the activation [7]

$$z = h(a) \tag{2.2}$$

A non-linear activation function is required to obtain a non-linear representation of the input data.

2.1.2 Feedforward Neural Networks

An FFNN is a layered network where neurons in one layer only connect to neurons in the next layer. An FFNN contains one input layer and one output layer and between these two layers there may be one or several hidden layers. An FFNN with one hidden layer is illustrated in Fig. 2.1.

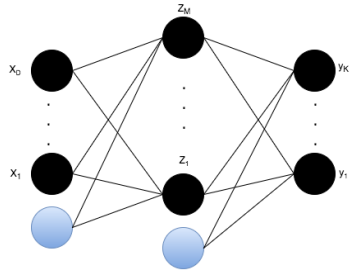


Figure 2.1: An FFNN with inputs $x_1, \ldots, x_D$, hidden nodes $z_1, \ldots, z_M$ and outputs $y_1, \ldots, y_K$. The blue nodes represent biases.

To calculate the output of an FFNN with one hidden layer one would first create M hidden unit activations from the D inputs $x_1, x_2, \ldots, x_D$

$$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \tag{2.3}$$

where $w_{ji}^{(1)}$ are the weights and $w_{j0}^{(1)}$ weights associated with the bias neurons. The superscript (1) indicates that the parameters belong to the first layer of the network. The activations are then transformed by applying a non-linear activation function h.

$$z_j = h(a_j) \tag{2.4}$$

Similarly, the output unit activations are calculated as

$$a_k = \sum_{j=1}^{M} w_{kj}^{(2)} z_j + w_{k0}^{(2)} \tag{2.5}$$

Finally, the output activations a_k are transformed using a suitable activation function to obtain the outputs y_k

$$y_k = h(a_k) \tag{2.6}$$

In the case of several hidden layers, the idea is the same. Activations are calculated for the first layer and these are then passed through an activation function. The resulting output is then input to the next layer. This process is repeated until reaching the final output layer. In this way, information is forward propagated through the network. Two layers are called fully connected if all nodes in the first layer are connected to all nodes in the next layer.

2.1.3 Activation Functions

A common type of non-linear activation function is the sigmoid function

$$h(x) = \sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.7}$$

with an output range of [0,1]
The sigmoid function is related to the hyperbolic tangent function as follows

$$\sigma(2x) = \frac{1 + \tanh(x)}{2} \tag{2.8}$$

and is another commonly used activation function. An important difference between the two functions is that $\tanh(x)$ returns a value in the range [-1,1].

In modern neural networks, the recommendation is to use the Rectified Linear Unit (ReLU) defined by the following activation function [10]:

$$g(z) = \max\{0,z\} \tag{2.9}$$

The linear rectifier is suitable for most FFNNs and has recently replaced the sigmoid and hyperbolic tangent function in many applications. A major benefit is the reduced likelihood of vanishing gradients. The derivatives of the sigmoid and hyperbolic tangent function quickly vanish away from the origin. Vanishing gradients are problematic when deep ANNs are trained with gradient-based learning methods such as backpropagation. Vanishing gradients is not a problem when ANNs are optimised with a genetic algorithm. However, there are additional benefits of ReLU. ReLU achieves sparser representation compared to the sigmoid and hyperbolic tangent function.

One potential problem with the linear rectifier is the zero gradient for $x < 0$. ReLUs can irreversibly die if the weights are updated in such a way that the neuron never activates. The zero gradient means that gradient-based update methods will never change the weights from the 'dead' state. Leaky ReLU seeks to solve the problem with zero activation. The Leaky rectifier has a small negative slope for $x < 0$ and is defined as

$$g(z) = \max\{\alpha x, x\} \tag{2.10}$$

The parameter α is often set to a small value close to zero, e.g. 0.01. Empirical studies have shown that leaky ReLU improves performance in some cases [11]. ReLU is non differentiable at $z = 0$. However, in practice this is not a problem, since the function is differentiable in points arbitrarily close to zero.

2.1.4 Weight initialisation

Glorot and Bengio have proposed a method for initialising weights in an ANN [12]. The formula is based on the assumption that there are no non-linearities between layers, which is an invalid assumption. However, this initialisation works well in many applications. He et al. refined the method to include ReLu activation functions [13]. He et al. suggested that weights were drawn from a normal distribution with zero mean and standard deviation $\sigma = \sqrt{\frac{2}{n_l}}$ where n_l is the number of nodes in a given layer. This method was used to initialise the weights of the networks. The biases were initialised to 0.

2.1.5 Optimising Neural Networks

There are several ways to optimise an ANN, i.e. find the best set of weights for a given task. Backpropagation is the most common method of training an ANN. However, there are several other optimisation methods. Evolutionary algorithms, such as a Genetic Algorithm (GA) and Particle Swarm Optimisation (PSO) can outperform gradient-based updates in some cases [7].

2.1.5.1 Training Using Gradient Descent

The simplest approach to use gradient information to update the weights in the network is to choose the weight updates as a small step in the negative gradient direction.

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla_{\mathbf{w}} E(\mathbf{w}^{(\tau)}) \tag{2.11}$$

The parameter $\eta > 0$ is called the learning rate and the function $E(\mathbf{w}^{(\tau)})$ is known as the error function. In many applications the error function is referred to as a loss function or cost function. After each update the gradient is re-evaluated and the process is repeated to optimise the weights. The approach of moving the weights in the direction of the negative gradient is called gradient descent.

2.1.5.2 Error Backpropagation

One technique for evaluating the gradient of the error function $E(\mathbf{w})$ is called error backpropagation. Suppose the error function comprise a sum of terms containing the error from each data point in the training set

$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w}) \tag{2.12}$$

We would like to calculate the derivative of the error with respect to the network weights which can be calculated using the chain rule

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}} \tag{2.13}$$

Now the following notation is introduced

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j} \tag{2.14}$$

If an additional parameter $z_0 \equiv 1$ is defined, the bias parameters can be absorbed into the weights

$$a_j = \sum_{i=1}^{D} w_{ji} z_i + w_{j0} = \sum_{i=0}^{D} w_{ji} z_i \tag{2.15}$$

Using this equation we obtain

$$\frac{\partial a_j}{\partial w_{ji}} = z_i \tag{2.16}$$

Inserting the equations (2.14) and (2.16) into Eq. (2.13) the expression for the derivatives becomes

$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i \tag{2.17}$$

For the output units we have

$$\delta_k = y_k - t_k \tag{2.18}$$

where y_k is the output from the network and t_k are the target values. For the hidden units we again use the chain rule

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k} \frac{\partial a_k}{\partial a_j} \tag{2.19}$$

By using the definition in Eq. (2.14) once again together with Eq. (2.15) and (2.4) we obtain the backpropagation formula

$$\delta_j = \sum_k \frac{\partial E_n}{\partial a_k} \frac{\partial a_k}{\partial a_j} = h'(a_j) \sum_k w_{kj} \delta_k \tag{2.20}$$

This equation tells us that the δ of a particular hidden unit is obtained by propagating the error backwards from units of a layer further on in the network. The backpropagation algorithm can be summarised as follows:

Algorithm 1. *Backpropagation*

1. *Use an input vector and forward propagate through the network by calculating the activations and applying the activation functions as in Eq.* (2.15) *and* (2.4).
2. *Calculate the* δ_k *using Eq.* (2.14)
3. *Backpropagate the errors to obtain the* δ_j *for all hidden units in the network using Eq.* (2.20)
4. *Calculate the derivatives using Eq.* (2.17) *and update the weights.*

2.1.5.3 Problems With Training Deep Networks

Optimisation in general is a difficult task. By designing objective functions and constraints such that the problem is convex any local minimum will also be a global minimum [14]. However, if the problem is non-convex local minimum is no longer guaranteed to be the global minimum. ANNs are non-convex functions and can have a large number of local minima.

Problems arise if the local minima have high cost compared to the global minimum. If such local minima are common, gradient-based algorithms might perform poorly [10]. However, research indicates that for sufficiently large networks, most local minima are associated with a low cost value [15]; [16].

Another problem with training deep networks is the problem of unstable gradients. As the error is backpropagated to earlier layers the gradient might vanish or explode. A small gradient does not necessarily mean that we are already near a minimum. This is unlikely given a random initialisation of the weights. The formula in Eq. (2.20) gives intuition to why the gradient can vanish in earlier layers. If the derivative of the non-linear function is small ($h'(a_j) \ll 1$), δ_j will become very small in earlier layers provided that the weights w_{kj} are not large. However, if instead, the weights w_{kj} are large, δ_j could become very large in earlier layers resulting in an exploding gradient [17].

The problem of the weights becoming too large during training can be addressed by adding a regulariser to the cost function, thus punishing large values of the weights [8]. By using ReLu or leaky ReLu the derivatives will always be equal to 1 at points on the positive axis. This is helpful when dealing with vanishing gradients [10].

2.1.5.4 Improved Training Methods

As of now, no optimisation algorithm consistently outperforms all other algorithms. A study was conducted in 2013 to test optimisation algorithms on a wide variety of learning tasks. Optimisation algorithms with adaptive learning rates required less parameter tuning and algorithms such as RMSprop and AdaDelta were rather robust for all tasks, even though the performance varied [18].

2.1.5.4.1 The Momentum Algorithm One method to accelerate learning is to use the momentum algorithm [19]. An exponentially decaying moving average of past gradients is accumulated and the algorithm continues to move in this direction. The momentum algorithm introduces a velocity parameter $\mathbf{v}$ which describes in which direction and at which speed the parameters moves through the parameter space. The momentum algorithm updates the weights as follows

$$\mathbf{v} \leftarrow \alpha\mathbf{v} - \epsilon\nabla_{\mathbf{w}}\left(\frac{1}{N}\sum_{n=1}^{N} E_n(\mathbf{w})\right)$$
$$\mathbf{w} \leftarrow \mathbf{w} + \mathbf{v}$$

If α is increased in comparison to ϵ previous gradients have more effect on the current direction [10].

2.1.5.4.2 Nesterov Momentum Nesterov momentum is a modification of the momentum algorithm. Nesterov momentum uses similar parameters as momentum but evaluates the gradient after the velocity is applied [20]. In Nesterov momentum the weights are updated as follows

$$\mathbf{v} \leftarrow \alpha\mathbf{v} - \epsilon\nabla_{\mathbf{w}}\left(\frac{1}{N}\sum_{n=1}^{N} E_n(\mathbf{w} + \alpha\mathbf{v})\right)$$
$$\mathbf{w} \leftarrow \mathbf{w} + \mathbf{v}$$

The default momentum rate in Lasagne is set to 0.9 for both ordinary momentum and Nesterov momentum.

2.1.5.4.3 Adaptive Learning Rates The learning rate used for training ANNs can be difficult to set as it strongly affects performance. By using momentum this is no longer as much of an issue, but instead another hyperparameter is introduced. Another way is to use a learning rate for each parameter and adapt these learning rates during training.

2.1.5.4.4 RMSProp One popular optimisation algorithm using adaptive learning rates is called RMSProp. It is designed to work well even for non-convex problems and is one of the most common optimisation algorithms used in the context of deep learning [10]. The full algorithm is summarised as follows:

Algorithm 2. *RMSprop*

1. *Required parameters: global learning rate* ϵ*, small constant* $\delta \approx 10^{-7}$ *for numerical stability*
2. *Initialise gradient accumulation variable* $\mathbf{r} = \mathbf{0}$
3. ***while*** *stopping criterion not met* ***do****:*
   (a) *Sample a minibatch* $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\}$ *from the training set with corresponding targets* $\mathbf{y}^{(i)}, i = 1, \ldots m$
   (b) *Compute gradient:* $\mathbf{g} \leftarrow \frac{1}{m}\nabla_{\mathbf{w}}\left(\sum_{n=1}^{m} E_n(\mathbf{w})\right)$
   (c) *Accumulate squared gradient:* $\mathbf{r} \leftarrow \mathbf{r} + \mathbf{g} \odot \mathbf{g}$
   (d) *Compute update:* $\Delta\mathbf{w} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\mathbf{r}}} \odot \mathbf{g}$ *(Element-wise division and square root)*
   (e) $\mathbf{w} \leftarrow \mathbf{w} + \Delta\mathbf{w}$

2.1.5.4.5 Adam Another recently proposed algorithm for stochastic optimisation is Adam. This method is designed to combine the advantages of RMSProp and AdaGrad [21]. RMSProp works well for on-line learning and non-stationary settings. AdaGrad is designed to work well with sparse and noisy gradients.

Algorithm 3. *Adam*

1. *Required parameters: Step size* α*, Exponential decay rates for the moment estimates* $\beta_1, \beta_2 \in [0,1)$ *and small constant* $\epsilon \approx 10^{-8}$ *for numerical stability*
2. *Initialise weights* $\mathbf{w}$*, 1st moment vector* $\mathbf{m}$*, 2nd moment vector* $\mathbf{v}$ *and time step* t
3. ***while*** *stopping criterion not met* ***do****:*
   (a) *Perform time step* $t \leftarrow t + 1$
   (b) *Sample a minibatch* $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\}$ *from the training set with corresponding targets* $\mathbf{y}^{(i)}, i = 1, \ldots m$
   (c) *Compute gradient:* $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\mathbf{w}} \left(\sum_{n=1}^{m} E_n(\mathbf{w}) \right)$
   (d) *Update biased first moment estimate:* $\mathbf{m} \leftarrow \beta_1 \cdot \mathbf{m} + (1 - \beta_1) \cdot \mathbf{g}$
   (e) *Update biased second raw moment estimate:* $\mathbf{v} \leftarrow \beta_2 \cdot \mathbf{v} + (1 - \beta_2) \cdot \mathbf{g} \odot \mathbf{g}$ *(element-wise square)*
   (f) *Compute bias-corrected first moment estimate:* $\hat{\mathbf{m}} \leftarrow \frac{\mathbf{m}}{1-\beta_1^t}$
   (g) *Compute bias-corrected second raw moment estimate:* $\hat{\mathbf{v}} \leftarrow \frac{\mathbf{v}}{1-\beta_2^t}$
   (h) *Update parameters:* $\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\hat{\mathbf{m}}}{\sqrt{\hat{\mathbf{v}}} + \epsilon}$

The exponential decay rates are often set to $\beta_1 = 0.9$ and $\beta_2 = 0.999$ and a good default value for the step size is $\alpha = 0.001$

2.2 Reinforcement Learning

Reinforcement learning is one field of machine learning dealing where artificial agents learn to take actions that maximise some cumulative reward.

2.2.1 Markov Decision Process

To provide a mathematical foundation for modelling decision making, we can make use of Markov Decision Processes (MDPs).

Definition 1. *An MDP is a 5-tuple* (S, A, R, P, γ) *where* S *is a set of states,* A *is the set of actions,* $R = R_a(s,a)$ *is the reward function,* $P = P_a(s'|s,a)$ *is the state transition function from state* s *to* s' *and* γ *is the discount factor. The discount factor is the factor by which a future reward must be multiplied to obtain the present value of the reward.*

MDPs satisfies the Markov property [22], i.e. given an action a and a state s the state transition function $P_a(s'|s,a)$ is conditionally independent of all previous actions and states. In other words the probability that the process moves into a state s' depends only on the current state s and not on the prior history.

A policy π specifies the action the agent will choose in a given state. The main problem of MDPs is to find a policy $\pi = P(a|s)$ maximising future rewards given the current state s and action a.

2.2.2 Q-learning

Q-learning is a model-free method for reinforcement learning and can be used to learn optimal policies in MDPs.

2.2.2.1 Action-value Function

Reinforcement learning can be used to solve MDPs without explicitly knowing the state transition function $P_a(s,s')$. In this case it is useful to introduce an action-value function $Q(s,a)$ which takes states and actions (s,a) and gives the value of this state-action pair.

The action-value function is defined as

$$Q(s,a) = \mathbb{E}\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... | s_t = s, a_t = a, \pi\right] \tag{2.21}$$

where r_t, a_t and s_t denote the reward, action and state at time t respectively. γ is the discount factor.

2.2.2.2 The Q-learning Algorithm

The Q-learning algorithm is an algorithm that iteratively updates an approximate action-value function $Q(s,a)$ to approach the optimal function $Q^*(s,a)$. In Q-learning the Q-function is updated using the following update rule [23]:

$$Q_{t+1}(s,a) \leftarrow Q_t(s,a) + \alpha_t \left(\underbrace{r_{t+1} + \gamma Q(s_{t+1}, a_{t+1})}_{\text{learnt value}} - \underbrace{Q(s_t,a_t)}_{\text{old value}} \right) \tag{2.22}$$

The difference between the learnt value and the old value is known as the TD-error

$$\delta = r_{t+1} + \gamma \hat{Q}(s_{t+1}, a_{t+1}) - Q(s_t,a_t) \tag{2.23}$$

2.2.3 Exploration

Q-learning is an off-policy learning method meaning that it learns the value of the optimal policy independently of executed actions. The state and action spaces is explored to to build a better estimate of the optimal Q-function. Exploring the search space using an ϵ-greedy strategy is often very effective [24]. The ϵ-greedy strategy is a way of selecting actions. A random action is executed with probability ϵ and the action corresponding to the highest Q-value is executed with probability $1 - \epsilon$.

2.3 Deep Q-Learning

In deep Q-learning the concepts of ANNs and Q-learning are combined by using an ANN to approximate the optimal action-value function

$$Q^*(s,a) = \max_{\pi} \mathbb{E}\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... | s_t = s, a_t = a, \pi\right]$$

The optimal action-value function maximises the sum of future rewards discounted by a factor γ given a state $s_t = s$, an action $a_t = a$ and a policy π.

When using gradient-based methods for optimising the ANN a loss function has to be defined. As in Q-learning, the TD-error is used to define a loss function which can be done in several ways. One possibility is to define the loss as

$$E_j(\mathbf{w}) = |\delta_j|^2 = |r_{j+1} + \gamma \hat{Q}(s_{j+1}, a_{j+1}) - Q(s_j,a_j)|^2 \tag{2.24}$$

To improve stability one can also use a Huber loss function [25]

$$E_j(\mathbf{w}) = \min(|\delta_j|^2, |\delta_j|) \tag{2.25}$$

A deep neural network approximating an action-value function is called a deep Q-network (DQN).

2.3.1 Stability Issues

There are known stability issues when using a non-linear function approximator to approximate the action-value function in a reinforcement learning context. The main causes are the correlation between observations, the correlation between Q-values and their target values $r + \gamma \max_a Q(s,a)$ and the fact that a small update to the action-value function $Q(s,a)$ can cause a large change in the policy. To alleviate these issues, two concepts are introduced.

The first concept is experience replay which randomises over the data by sampling a mini-batch of transitions $(s_j, a_j r_j, s_{j+1})$ from a memory. Experience replay removes the correlation between consecutive samples and smooths changes in the data distribution[3]. Experiences $e_t = (s_t, a_t, r_t, s_{t+1})$ are stored in a memory at each time step t. During training a mini-batch of experiences is sampled from the experience replay memory and this mini-batch is then used to update the Q-function using a suitable optimisation method. In the simplest case samples are drawn uniformly from the experience replay memory. However, other strategies are possible. Prioritised experience replay is a strategy where important transitions are sampled more frequently. This technique will be discussed in detail later in this chapter.

The second concept is to use a target network $\hat{Q}(s,a)$ to calculate the target values. This network is only updated every C time steps, thus removing the correlations between the Q-values and the target values.

2.3.2 The Deep Q-learning Algorithm

Deep Q-learning has several advantages over standard online Q-learning. Learning from consecutive transitions is inefficient due to the correlation between the transitions. Second, when using experience replay memory each experience is potentially used in multiple updates of the weights, thus improving data efficiency.

The deep Q-learning algorithm is summarised as follows:

Algorithm 4. *Deep Q-learning with Experience Replay*

1. *Initialise action-value function with weights* $\mathbf{w}$*, target action-value function with weights* $\mathbf{w}^- = \mathbf{w}$ *and experience replay memory* D *with capacity* N
   for *episode* $= 1, \ldots, M$ ***do****:*
      for $t = 1, \ldots, T$ ***do****:*
      i. *With probability* ϵ *select random action* a_t*, otherwise select* $a_t = \arg\max_a Q(s_t, a; \mathbf{w})$
      ii. *Execute action* a_t *and observe reward* r_t *and state* s_{t+1}*.*
      iii. *Store transition* (s_t, a_t, r_t, s_{t+1}) *in* D
      iv. *Sample mini-batch of transitions* (s_j, a_j, r_j, s_{j+1}) *from* D
      v. *Set* $y_j = \begin{cases} r_j \text{ if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(s_{j+1}, a'; \mathbf{w}^-) \text{ otherwise} \end{cases}$
      vi. *Perform a gradient step on* $E(\mathbf{w})$ *with respect to the parameters* $\mathbf{w}$
      vii. *Every* C *steps, set* $\hat{Q} = Q$

2.3.3 Improvements to the Deep Q-learning Algorithm

Since the deep Q-learning algorithm first was introduced, several improvements have been suggested to further improve stability and speed of convergence. Here, two of these improvements called double Q-learning and prioritised experience replay will be discussed.

2.3.3.1 Double Q-learning

Under some circumstances the Q-values will be overestimated and this can harm performance in some cases. Double Q-learning (DDQN) can be used to give more accurate action-value estimates. DDQN has shown to be much more stable [5]. A second network is introduced to decouple action selection and action evaluation when calculating the target values. By using a target network $\hat{Q}$, a natural candidate already exists for the second network.

In DDQN the updates are the same as in DQN but the target updates

$$y_j^{\text{DQN}} = \begin{cases} r_j \text{ if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(s_{j+1}, a'; \mathbf{w}^-) \text{ otherwise} \end{cases} \tag{2.26}$$

in DQN are replaced by

$$y_j^{\text{DDQN}} = \begin{cases} r_j \text{ if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(s_{j+1}, \arg\max_a Q(s_{j+1,a;\mathbf{w}}); \mathbf{w}^-) \text{ otherwise} \end{cases} \tag{2.27}$$

2.3.3.2 Prioritised Experience Replay

When sampling experiences uniformly from the memory, transitions will be replayed at the same frequency with which they were originally experienced, regardless of

their importance. By using prioritised experience replay, important transitions are replayed more frequently, which makes learning more efficient. In prioritised experience replay samples with large TD-errors are replayed more often.

The probability of sampling transition j is defined as

$$P(j) = \frac{p_j^\alpha}{\sum_k p_k^\alpha} \tag{2.28}$$

where p_j is the priority of transition j. The exponent $\alpha \in [0,1]$ describes how much prioritisation is used. On way to define the priority of transition j is to use proportional prioritisation where $p_j = |\delta_j| + \epsilon$ where ϵ is a small constant that prevents that some transitions are never revisited once their error is zero [4].

2.3.4 Curriculum Learning

One possible way to deal with global optimisation of non-convex functions is to use so called continuation methods. A particular form of a continuation method that can be used in the context of ANN optimisation is called curriculum learning. By gradually increasing the difficulty of learning and not presenting the training examples randomly the speed of convergence increases in some cases [26].

Curriculum learning is a training strategy where training examples are not randomly presented but introduced in a well-thought-out order. More complex concepts are gradually introduced. Such a training strategy can improve generalisation [26]. When using curriculum learning it is easier to see if the ANN is learning something useful without having to complete the entire training process. The network should perform reasonably well on simple situations early on in training.

2.4 Genetic Algorithm

Another way to optimise an ANN is to use a Genetic Algorithm (GA) approach. In a GA a population of N individuals, each representing the weights in an ANN is initialised and encoded into a string of digits called chromosomes. The digits building up the chromosome are referred to as genes. Inspired by evolution, the fittest individuals survive until the next generation. A fitness measure has to be defined in order to decide which individuals are the fittest. A GA forms new individuals in two ways. One way is called mutation where the networks weights in a chromosome are modified with a certain probability. The other way to form new individuals is referred to as crossover. Crossover combines two chromosomes to form two new chromosomes, which in turn represents two new networks.

There are some important differences between how the ANN is updated with a GA compared to in deep Q-learning. A GA evaluates the performance on a set of training data by defining a fitness function. This fitness measure should not be confused with the reward function used in deep Q-learning. In the deep Q-learning algorithm the weights are updated at each time step with backpropagation. In a GA new individuals are formed based on the performance over the whole training set. A GA does not involve any gradient information for updating and can be efficient at avoiding local optima [7].

2.4.1 Fitness

In a GA every individual has an associated fitness value, F_i, representing how well it performs on a given task. There are many ways of defining a fitness function and finding a good fitness function for a given problem might require some trial and error.

2.4.2 Selection

The process of choosing chromosomes used to form new individuals is called selection. The two most common selection methods are roulette-wheel selection and tournament selection.

2.4.2.1 Roulette-wheel Selection

Roulette-wheel selection selects individuals in proportion to fitness. By defining cumulative relative fitness values ϕ_j as

$$\phi_j = \frac{\sum_{i=1}^{j} \phi_i}{\sum_{i=1}^{N} \phi_i} \tag{2.29}$$

and by generating a random number $r \in [0,1)$ the individual with the smallest j such that $\phi_j > r$ is selected. The procedure can be visualised as a roulette-wheel, where each individual takes up a slice proportional to its fitness [7].

A potential problem with roulette-wheel selection is that an individual with above-average fitness in the randomly generated population might dominate the population after a few generations. One solution is to use fitness ranking where the fitness values are ranked so that the individual with the highest fitness is given fitness N, the second best individual is given fitness $N-1$ etc. In general the reassignment of fitness values can be written as

$$F_i^{\text{rank}} = F_{\max} - (F_{\max} - F_{\min})\frac{R(i)-1}{N-1} \tag{2.30}$$

where $R(i)$ is the ranking of individual i [7].

2.4.2.2 Tournament Selection

In tournament selection, j individuals are drawn randomly from the population. The individual with the highest fitness value is chosen with probability p_{tour} which typically takes values between 0.7 and 0.8. If the best individual is not chosen, the best of the remaining $j-1$ individuals is chosen with probability p_{tour}. The procedure is repeated until one of the j individuals is chosen [7].

2.4.3 Crossover

Crossover is an efficient way to gather partial solutions from different parts of the parameter space and form new individuals that might perform better. Two individuals

are selected for crossover by using one of the selection methods (roulette-wheel or tournament selection). The most common type of crossover is single-point crossover. In single-point crossover one of the $m-1$ available points in a chromosome of length m is randomly chosen. A new individual is created by crossing the chromosomes at the chosen point with probability p_{cross} [7]. For an illustration of crossover, see Fig. 2.2

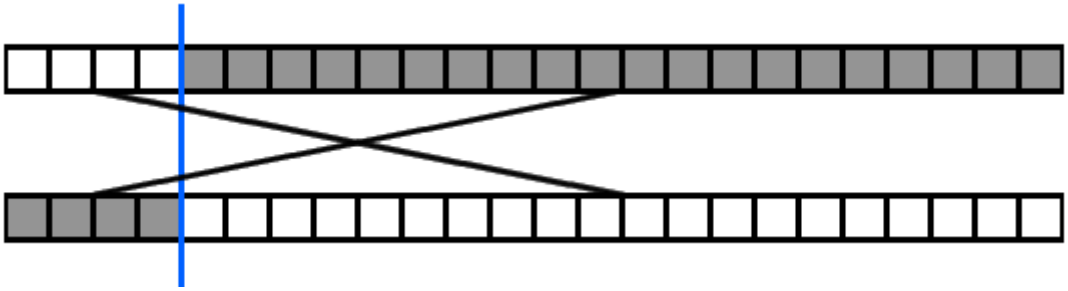


Figure 2.2: An illustration of single-point crossover. The two chromosomes coloured white and gray respectively are crossed at the chosen point represented by the blue line to form two new chromosomes.

2.4.4 Mutation

Another way new genetic material is introduced is by mutation. Each gene is mutated with probability p_{mut} which is normally set to c/m where the constant c is of order 1 and m is the length of the chromosome. If the chromosomes contain real-valued elements (e.g. when using ANNs) it is common to use a method called real-numbered creep mutation when generating new values for the genes. A new value is drawn from a distribution centred around its previous value, and a creep rate describes the range of the new value [7].

2.4.5 Elitism

Even though the best individual in a given generation is very likely to be selected for reproduction, there is no guarantee that it will be selected. To make sure that the best individual survives until the next generation the best individual is copied over to the next generation. This procedure is known as elitism.

2.4.6 Standard Genetic Algorithm

Combining these concepts we can summarise a standard GA with real-value encoding in the following way:

Algorithm 5. *Genetic Algorithm with Real-number Encoding*

1. *Initialise the population by randomly generating N strings of real numbers (chromosomes)* $c_i, i = 1,...,N$ *of length* m *containing variables* $x_{ij}, j = 1,...,n$.
2. ***while*** *stopping criterion not met* ***do****:*
   (a) *Evaluate the individuals:*
      i. *Evaluate the objective function* f *using the variable values and assign a fitness value* $F_i = f(x_i)\,\forall i$.
      ii. *Set* $i_{best} \leftarrow 1, F_{best} \leftarrow F_1$ *and loop through all individuals; if* $F_i > F_{best} : F_{best} \leftarrow F_i, i_{best} \leftarrow i$
   (b) *Form the next generation*
      i. *Make an exact copy of the best chromosome* $c_{i,best}$
      ii. *Select two individuals* i_1 *and* i_2 *from the evaluated population, using a suitable selection operator.*
      iii. *Generate two offspring chromosomes by crossing the two chromosomes* c_{i_1} *and* c_{i_2} *of the two parents with probability* p_{cross}. *With probability* $1 - p_{cross}$, *copy the parent chromosomes without modification.*
      iv. *Mutate the two offspring chromosomes.*
      v. *repeat (ii-iv) until* $N - 1$ *additional individuals have been generated. Then replace the* N *old individuals by the* N *newly generated individuals.*

3
Methods

The fundamental idea of the project was to use neural networks to construct a collision avoidance system. Although neural network have achieved impressive results in a wide range of applications it is not entirely obvious how to best incorporate the technique in a collision avoidance system. There are several possible design strategies. Therefore, during the first stage of the project, a literature review was conducted in order to get familiar with important concepts and previous work in the field. The study was mainly focused on a technique combining neural networks and reinforcement learning since such an approach has been shown to be successful when learning control policies directly from sensory input. This technique is called deep Q-learning.

A simulation environment developed by the Volvo Car Corporation was used to construct traffic scenarios and simulate corresponding sensory data from a car. Feature vector observations obtained from the simulation environment were used as input to the neural networks during training. To be able to train the networks on a sufficiently large number of traffic situations a parametric model of traffic scenarios was used to automatically create and generate training data.

Before any attempt to train a full scale collision avoidance system based on deep Q-learning some simple sample problems were solved to improve the understanding of fundamental concepts. This way a correct implementation of the algorithm could be ensured and the complexity of the problem gradually increased.

During the last phase of the project an alternative approach was investigated. Neural networks were used to represent braking strategies and the network weights were updated with a genetic algorithm.

3.1 Implementation Strategy

There are several possible ways neural networks can be employed in a collision avoidance system. The problem of learning braking policies could be treated as a classification problem. A neural network could then be used to identify to which of the two classes *brake* and *no brake* an observation belongs. One obvious weakness with such an approach is that it requires knowledge of how to define the two classes. Another possible approach is to use neural networks to solve the non-linear problem of predicting vehicle paths. Such predictions would of course be a helpful contribution, however, they would not be sufficient to determine whether an intervention is necessary, which was the purpose of the project. Deep Q-learning has been shown to be successful when learning control policies directly from sensory input. Deep Q-

learning is an agent-based algorithm where the agent takes actions and influences its environment. The technique is in some sense a general framework for goal directed behaviour and may therefore be suitable when building an agent that can solve the problem of finding adequate braking policy.

An important factor when choosing implementation strategy was extensibility. When moving towards fully autonomous driving deep Q-learning is promising as it can be extended to more actions than braking. Reinforcement learning can be seen as a general framework concerning how an agent is interacting in an environment. A Japanese research group have managed to simulate self driving cars using a deep Q-learning approach.

One potential difficulty with deep Q-learning is that the agent gets a reward at each time step even though feedback about an action may only be received many time-steps later. This requires the agent to be able to plan many time steps into the future. Google DeepMind struggled with games demanding temporally extended planning strategies [3]. A genetic algorithm is a natural choice of optimisation method when a useful set of input-output pairs is not available and is an effective method in large and complex search spaces. The performance of a network optimised with a genetic algorithm is measured by a single scalar value given at the end of the evaluations. This means that difficulties linked to rewarding the agent at each time step can be escaped. The two neural network based approaches were at the end benchmarked against a simulation model developed by Volvo.

3.2 Simulation Model

The Volvo Car Corporation has developed a simulation environment where traffic scenarios can be generated and real time sensory data can be simulated conveniently. It was convenient to use this model during training since arbitrary traffic scenarios could be created by simply specifying vehicle trajectories, velocities and accelerations. A further motivation for using the already existing model was that it simplified benchmarking against the current collision avoidance system. To train a neural network a large set of training data is often necessary and in the case of deep Q-learning the agent must be able to explore and influence its environment. A feedback loop made the simulation environment appropriate for training of a DQN agent. Velocities and accelerations were recalculated when the brakes were applied and this made it possible to see the consequences of any chosen actions. To simulate a brake delay the brake request was ramped up to the maximum deceleration over several time-steps.

The simulation environment is implemented as a Simulink model. Since neural networks require lots of training data the simulation environment was converted to C code for improved speed. The code generation made it possible to make use of existing deep learning libraries to improve speed and reduce the amount of code needed. A shared library was built from the generated C code.

In the simulation environment it was possible to create complex traffic scenarios of arbitrary type and with several vehicles and pedestrians. However, only one type of collisions called rear-end collision was of interest for the purpose of this project. A rear-end collision is a traffic accident wherein a vehicle crashes into

the rear of another vehicle in front. When creating such traffic scenarios only two vehicles were assumed to be immediately involved. The controlled car (the host) was always assumed to be driving behind the other car (the target). No pedestrians were included in the traffic scenarios. To further simplify the problem only straight roads were considered. In Fig. 3.1 a snapshot of a simulated traffic scenario seen from a bird's view is shown. The red rectangle is the host vehicle driving behind the target, the green rectangle, on a straight road.

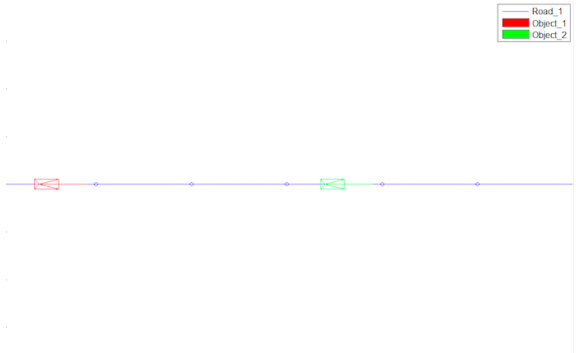


Figure 3.1: The figure shows a snapshot of a simulated traffic scenario seen from a bird's view. The red vehicle (host) is driving behind the green vehicle (target) on a straight road visualised as a blue line.

The current collision avoidance system is given information about surrounding objects as feature vectors. A large number of objects can be detected simultaneously and observations are updated at 50 Hz. The dimension of the information available is in other words dependent on the number of detected objects. In the simulation model the host vehicle is given information concerning surrounding objects the same way as in real life. This was important for the validity of the simulations.

A parametric model of rear-end collision scenarios was developed to enable automatic generation of scenarios. This way an unlimited number of training examples could be generated. The total duration of each scenarios was set to 15 s. The longitudinal distance travelled by the vehicles is described by the following equation

$$l(t) = l_0 + v(t)t + \frac{1}{2}a(t)t^2 \tag{3.1}$$

In the equation l denotes the distance travelled, l_0 an offset along the specified trajectory, v the longitudinal velocity and a the longitudinal acceleration. The acceleration was changed at three discrete times during each scenarios. The time of these events was randomly chosen and unique for each vehicle. The accelerations measured in $\mathrm{m/s^2}$ were drawn from a normal distribution with zero mean and unit variance clipped at $\pm 6\,\mathrm{m/s^2}$. The maximum allowed velocity was 50 m/s and the initial velocity of each vehicle was randomly chosen within the allowed range. As in simulations of the current system there was a brake delay of 0.21 s.

To achieve good generalisation the set of training data had to include scenarios both requiring and not requiring interventions.

3.3 Implementation Details

Feature vectors describing surrounding objects obtained from the simulations were used as input to the neural networks. A potential problem with such an approach is that the size of the network input then depends on the number of detected objects. A possible solution to this problem is to use some kind of embedding method. However, there was only one other vehicle in the rear-end scenarios studied, for that reason, embedding was not necessary in this project.

A schematic flowchart showing the control loop is shown in Fig. 3.2. The neural network was used to determine if emergency braking was needed based on the input obtained from the simulation model. In the case of emergency braking velocities and accelerations of the host vehicle were recalculated and updated in the simulations.

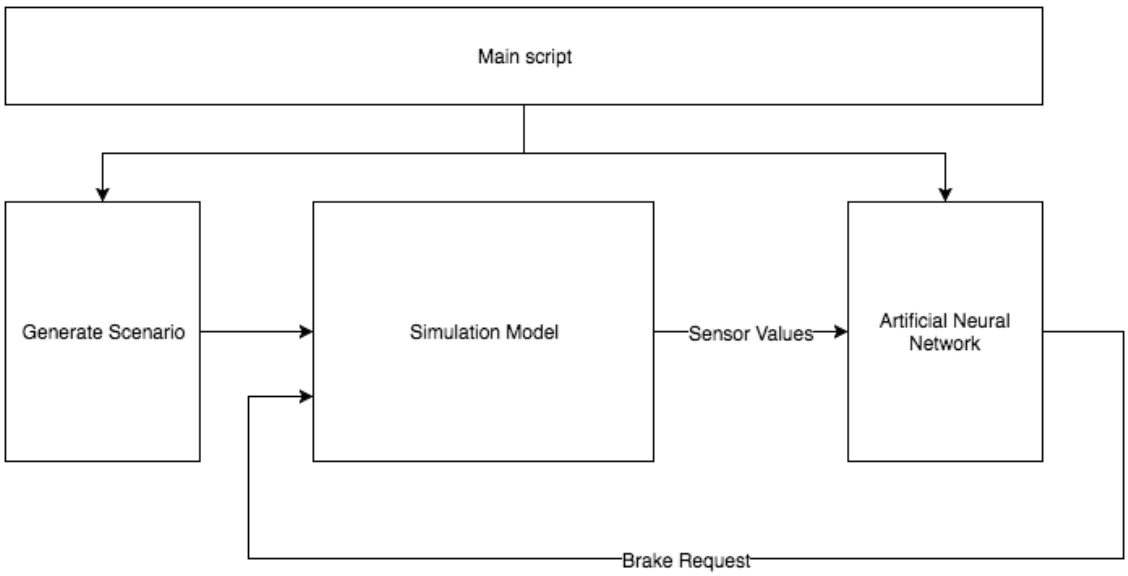


Figure 3.2: A flowchart describing the control loop. A traffic scenario was defined and fed to the simulation model. Information describing the state of the host vehicle and its environment could be obtained at every 0.02 s from the simulation model. This information was used as input to a neural network. Based on the input the neural network gave as output a decision of whether or not emergency braking was needed. In the case of emergency braking the host vehicle trajectory was recalculated. The main script was used to monitor the process.

Theano and Lasagne are two Python libraries that can be used to efficiently build and optimise neural networks. To permit the use of these libraries the main script was implemented in Python. Theano uses symbolic expressions and allows for efficient evaluation and optimisation of mathematical expressions. Lasagne is built on top of Theano and aimed specifically at building and training neural networks in Theano. Lasagne facilitates testing of different optimisation algorithms, regularisers, and set of hyperparameters. In addition, Lasagne enabled training of networks on GPUs. By using ctypes the simulation environment, as a shared library, was loaded into the Python process. Generation of traffic scenarios, retrieval of sensory data, stepping the simulation environment and accessing variables from the simulation

environment could then be done from python.

3.4 Simple Problem: Grid World

To make sure that the algorithm was working as expected the Q-network was tested on a simple game called Grid World. Grid World is a simple game in which a player moves around on a 3×4 grid of squares. The player can move up, down, left and right. There are 3 objects placed on the grid: a pit, a goal, and a wall. The target of the game is to is to find the shortest path to the goal. The game is lost if the player falls into the pit and the wall works as a roadblock. For an overview of the game, see Fig. 3.3

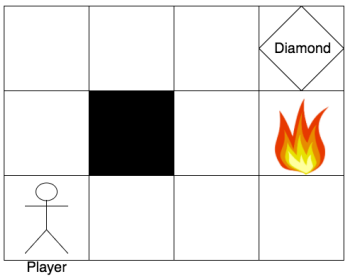


Figure 3.3: An illustration of the game Grid World. The player is represented by the stick figure and the goal is to find the shortest path to the diamond without falling into the fire pit. The black box represents a wall. There is also a wall around the entire grid.

Reinforcement learning is about getting an agent to take actions so as to maximise some cumulative reward. Therefore, to solve the Grid World problem with deep Q-learning, a reward function was introduced. Naturally, the optimal policy that maximises the cumulative reward is dependent on the way the agent is rewarded. A large reward should be received when the player reaches the goal. Likewise a large negative reward should be associated with falling into the pit. To promote the fastest path the player should receive a small negative living reward each step.

To verify that new implementations were working correctly, the Grid World problem was returned to several times. It is worth noticing that the fact that the algorithm can solve the Grid World problem does not guarantee that it will find an optimal braking strategy. However, the algorithm should definitely be efficient enough to solve such a simple problem to have a chance at solving the more complex problem of finding optimal braking strategies.

3.5 Training a DQN Agent for Collision Avoidance

The deep Q-learning algorithm 4, includes a large set of parameters. These parameters need to be finely tuned for good performance. It is not obvious that suitable parameters for one problem also are suitable for another.

1	Host velocity
2	Host acceleration
3	Host yaw rate
4	Longitudinal position of target
5	Lateral position of target
6	Target velocity
7	Target acceleration
8	Target heading
9	Relative Velocity

Table 3.1: A list of all inputs to the neural network.

As a first step, to reduce the complexity of the problem and to verify the set up, the DQN agent was trained to handle one single scenario. By reducing the size of the training set different parameter sets could be examined more conveniently due to the shorter training times. The strategy was to first find a suitable set of parameters solving the simplified version of the problem and then slowly relax the simplifications made by gradually increase the size of the training set.

The input of the Q-network consisted of a set of values that described the state of the vehicle and its environment and it gave as output a decision of whether or not emergency braking was needed. The input to the network was obtained from the simulation model and included the following quantities:

The yaw rate and target heading angle is not necessary on a straight road, but in general the network must be able to handle scenarios where these inputs are zero. The Q-values corresponding to each action was obtained as output from the network.

During training the state space was explored using an ϵ-greedy policy. The exploration rate, ϵ, was linearly annealed during training to a final value ϵ_{final} after reaching this value the exploration rate was fixed.

Transitions (s_t, a_t, r_t, s_{t+1}) were stored in the experience replay memory. Due to the finite memory size older transition were overwritten by more recent ones. In the deep Q-learning algorithm the weights of the Q-network are updated every time step during training. Every C updates the Q-network is cloned to obtain a target network $\hat{Q}$. This network is then used for approximate the targets y_j. In each time step, a mini-batch of transitions (s_j, a_j, r_j, s_{j+1}) were sampled from the experience replay memory and used to update the weights. Two different techniques were used to sample from the experience replay memory: (1) sampling uniformly at random from the memory and (2) sampling important transitions more frequently by letting the TD-error measure the importance of each transition.

There are some difficulties associated with training a DQN agent having only two allowed actions: *brake/no brake*. The difficulties are mostly due to that emergency braking is very decisive and the action should be very rare. For this reason it is tricky to know how to best handle a brake request in the control loop. If the host velocity is forced to zero at an intervention the output of the Q-network will be overruled in the time steps leading to the stationary state. Emergency braking is in

such a case a terminal action and the time between s_t and s_{t+1} will in most cases be significantly longer than one time step. It is known that problems requiring much future planning are difficult for DQN agents [3]. For this reason treating emergency braking as a terminal action seemed unfitting. Another approach is to let the agent let go of the brake when it does no longer see the situation as a threat. This is potentially a more difficult problem to solve but on the other hand the agent can be given a reward in each time step.

Simulations were stopped prematurely if any of the following events occurred: (1) speed of host vehicle below 0.01 m/s, (2) collision and (3) target vehicle finished predefined path without collision. The state in the final time step in each episode is referred to as a terminal state where the agent can no longer accumulate further reward. The agent was trained for a given number of episodes where each episode corresponded to a traffic scenario from the beginning until the end.

3.5.1 Parameter Sweeps

A common way of performing hyperparameter optimisation is by conducting a grid search. In practice a grid search means an extensive search through a specified subset of the parameter space. For grid search to be meaningful a pre-defined performance metric is necessary. In deep Q-learning the only good performance metric is obtained by periodically evaluate the learnt policy. This is cumbersome and computationally expensive.

Due to the high computational cost associated with performing an extensive search through the parameter space parameters were instead selected by performing an informal search on a simplified version of the problem. In addition some attempts to identify which parameters that seemed to have largest impact on performance were made. Table 3.2 contains a list of the parameters included in the deep Q-learning algorithm.

For an informal parameter sweep to be purposeful the algorithm must be stable. To investigate the stability of the algorithm agents were retrained several times with constant parameter settings and the results of the different runs were compared. Once the algorithm was stable enough, a set of reward functions were tested to see how the reward function affected the learning performance. Regularisation methods, such as dropout, were tested as well to see if it was possible to make the network more robust.

3.5.2 Training on Multiple Rear-end Scenarios

It is important to remember that it is not guaranteed that a parameter set that works well for training on a single scenario will work well for a larger training set. As a consequence some additional tuning of sensitive parameters were done when increasing the complexity of the problem by extending the training set. For example it is probable that a more complex problem will require a larger network. For this reason a few different network layouts were tested. The size of the networks was increased until the Q-function converged. When training a Q-network too small to accurately represent the Q-function it is impossible to learn a good braking policy.

Parameter	Description
Network depth	The number of hidden layers in the neural network
Network width	The number of nodes in each layer
Optimisation algorithm parameters	Different optimisation algorithms requires different parameters, such as learning rate, momentum rates, decay rates etc.
Number of training episodes	The number of episodes seen by the algorithm during training
Agent history length	The number of most recent observations given as input to the Q-network.
Initial exploration	Initial exploration rate ϵ in the ϵ-greedy strategy
Final exploration	Final exploration rate ϵ in the ϵ-greedy strategy
Final exploration episode	The number of episodes during which the exploration rate is linearly decreased to its final value.
Experience replay memory size	Network updates are sampled from this number of most recent states.
Batch size	The number of training cases used for computing the network updates in each step.
Target network update frequency	At every C steps the target network is updated and set equal to the neural network. This corresponds to the target network update frequency.
Discount factor	The discount factor γ representing the importance of present rewards compared to future rewards

Table 3.2: A table of all parameters used for the network and the DQN.

A large network requires more computation time to optimise the large amount of weights in the network.

Neural networks in active safety functions must generalise well to different kinds of traffic situations. One step in this direction is to train the network on multiple rear-end scenarios to see if the generalisation improves. To begin with, a network was trained on five scenarios. The number of scenarios experienced by the agent was then increased to 50 and further to 250.

3.5.3 Curriculum Learning

There are several approaches to achieve curriculum learning. To gradually increase the complexity of the problem new training examples were sequentially added the training set. In the beginning the agent was only trained on a single scenario, but at regular intervals a new scenario was added. During every training episode one

of these scenarios was randomly chosen. To make sure that the agent was trained on the latest scenario added to the training batch, this was given a 25 % chance of being sampled. This was repeated until another scenario was added to the training batch and this procedure continued until the test batch contained all scenarios.

To test the effects of curriculum learning, the agent was allowed to train on a new scenario during each training episode. In contrast to curriculum learning this does not allow a scenario to be reused in another episode. Because these randomly generated scenarios were never revisited, there is a risk of forgetting already learnt behaviour, also known as catastrophic forgetting.

3.5.4 Reward Function

There is no universally good way of constructing a reward function. A common guideline is to keep the function as simple as possible to limit the number of parameters. Even more importantly, the reward function must contain enough guidance to promote a desired behaviour. Multiple reward functions were tested systematically.

The reward functions penalised collisions. The penalty was either constant or a function of the relative impact velocity. A small living reward was used to encourage the agent to cover as long distance as possible. This reward was either set to a constant value or proportional to the velocity of the host vehicle. To prevent the agent from braking too early there was a penalty associated with applying the brakes. The negative contribution was either based on the time or the distance to the target vehicle. Finally, a reward was given for having a minimum distance to the target vehicle within a certain range. The ranges tested were [0.5, 1.0]m and [0.0, 1.0]m.

3.5.5 Prioritised Experience Replay

When using prioritised experience replay in practice, the priorities of every experience in the experience replay memory have to be stored in an efficient way. The probability of sampling an experience is normalised by the sum over all priorities, which can be seen in Eq. (2.28). Storing all priorities and taking the sum over all elements requires $\mathcal{O}(N)$ operations. If instead, all priorities are stored as leaves in a sum tree, the summation only requires $\mathcal{O}(\log N)$ operations. A sum tree is a tree structure where the parent is the sum of the two children. By letting the leaves of the tree contain all priorities, the root then stores the sum of all priorities.

3.6 Training a Neural Network With a Genetic Algorithm for Collision Avoidance

In addition to deep Q-learning another method to train neural networks to represent braking strategies was investigated. The braking strategies were again represented by FFNNs, however, a GA was used to update the weights of the networks. The reason for choosing a genetic algorithm was because suitable input-output pairs were not available or at least hard to define since the result of actions may be evident

first several time steps later. The networks took feature vector observations as input and gave as output a decision of whether or not emergency braking is needed. The vehicle was decelerated to 0 m/s at an intervention. The same input information as for the DQN agent was used to describe the state of the vehicle and its environment, see table 3.1.

It is not entirely obvious how to construct a fitness measure that reflects a good braking strategy. The system should only intervene when the driver has no other options than to brake to avoid a collision. There is an obvious trade-off between avoiding collisions and avoiding false positives. An important aspect of emergency braking in rear-end scenarios is that it often prolongs the distance travelled by the vehicle since a collision would prevent the vehicle from going any further. Therefore the distance travelled in a traffic scenario was used as starting point for the fitness measure. However, there are cases where the distance travelled is not maximised by an intervention. One such case is when the target vehicle is stationary. Therefore additional information had to be incorporated in the fitness measure. The trade off between a collisions and a too early intervention must be reflected by the fitness measure. It would be too blunt to let all collisions and false positives give a constant negative contribution to the fitness measure since that would give very little guidance on how to adjust the strategy. The optimal fitness measure is not easy to find and therefore a few different measures were tested and evaluated.

Another important aspect of the fitness measure is how to calculate the complete fitness measure for a given network over a set of different traffic scenarios. There are several possible strategies to do this. One way could be to calculate it as the average fitness over all traffic scenarios in the set. Another approach could be to set it equal to the fitness of the worst evaluation. A linear combination of the average and the worst is also possible. The most suitable choice is not known a priori and therefore different ways to calculate the complete fitness was investigate. The fitness measures of each scenario were normalised against a never-brake-strategy.

In Fig. 3.4 the optimisation loop is illustrated. The first step was to initialise a population of chromosomes. In this case the chromosomes encoded the weights of the networks, using real-number encoding. The weights were sampled from a normal distribution with a standard deviation of $\sigma = \sqrt{(2/\text{fan}_{\text{in}})}$ where fan_{in} is the number of input to a hidden unit. The leaky ReLu was used as non-linear activation function.

The networks were trained using holdout validation meaning that both training and validation sets were used in the evaluation step. The training and validation sets consisted of sets of rear-end scenarios both requiring and not requiring interventions. Feedback concerning the performance of a network on the training set was given to the optimisation algorithm by assigning a fitness value to each network based on the overall performance on the training set. The validation set was used to find the network with the highest ability to generalise, but no feedback about the performance on this set was given to the optimisation loop. The validation set was used to determine when overfitting occurred. The network with the highest validation fitness has the highest ability to generalise. By storing this network during training the best network could be retrieved.

A new population was generated after evaluation. Each generation of N indi-

viduals was replaced by a new generation of the same size. New individuals were generated using crossover and mutations. Three different techniques to select individuals for crossover was tested: (1) Roulette-wheel selection, (2) Ranked roulette-wheel selection and (3) tournament selection. The crossover probability was kept at a rather low value since crossover is known to often have negative effect on neural networks [7]. Two different types of mutations were used: ordinary mutations and creep mutations. Lastly, the best individual was copied to the next generation. Several new generations generated were and evaluated before the optimisation loop was stopped.

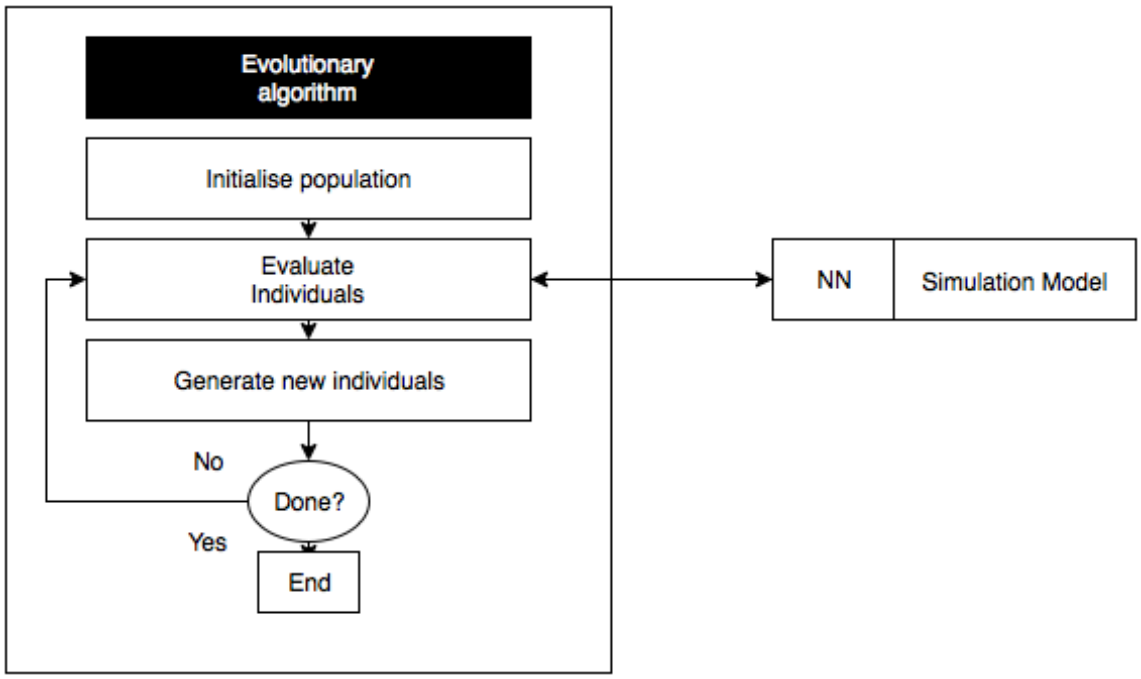


Figure 3.4: The figure shows the genetic algorithm optimisation loop.

3.7 Measuring Performance

To track the progress of the DQN agents some metrics measuring the learning were necessary. Mainly three indices of learning were studied. The evolution of the agent's average score per episode and average predicted Q-values were used. The score per episode is the sum of all rewards during an episode and the predicted Q-values are the highest Q-value obtained as output from the network in each time step. The backpropagation algorithm minimises the training loss by adjusting the weights of the neural network. The training loss is thus expected to be reduced during training. However, the training loss is not sufficient to illustrating the performance of the DQN agent. A low training loss does not necessary correspond to a good policy. Therefore, the agent's average score per episode and average predicted Q-values were monitored. Although the scores should increase over time the growth is not expected to be perfectly steady. The reason is the strong influence of even small changes of the weights on the overall distribution of states visited [3]. The average

predicted Q-values generally improves more smoothly over time and is therefore also an interesting measure to study [3]. The predicted Q-values are estimates of the maximum discounted reward the agent can obtain by following its current policy.

When using a genetic algorithm the performance is instead measure by a single scalar value: the fitness. When using holdout validation the performance on the training set and the validation set had to be measured during training. The performance on the training set was important since it was used in the selection step and the score on the validation set was used to find the network with the highest ability to generalise.

After training more immediate measures of performance is available. It is possible to test the agent on a set of traffic scenarios to directly measure the performance. The same test set was used to evaluate the performance of all different networks. The test set was uniquely used for testing the agents and it consisted of 1000 randomly generated rear-end scenarios both requiring and not requiring interventions. During testing the total driving time, the number of collisions and the minimum distance to vehicle in front during braking were logged. A minimum relative distance to the target vehicle larger than 1.0 m during braking was considered to be a false positive.

3.8 Benchmarking

Two different methods to train a neural network to learn successful braking policies were investigated during this project, one based on DQN and the other on a genetic algorithm. The performance of both methods was compared on a large test set consisting of 1000 randomly generated rear-end scenarios both requiring and not requiring interventions. In addition to the neural network based systems developed, the Volvo Car Corporation has developed a collision avoidance system based on the accurate modelling of vehicle dynamics. To know how well the neural network based methods can compete with such a model-based system the system developed by the Volvo Car Corporation was tested on the same test set.

The metrics used to benchmark the systems were (1) the total number of collisions, (2) the relative impact velocity and (3) the minimum distance to vehicle in front during braking. An intervention is considered correct if it avoids a collision and the minimum distance to the target vehicle is less than one meter.

3.9 Noisy Data

For good generalisation it was important to train the network on a large and diverse set of traffic situations. Since it is important that a collision avoidance system can handle real, noisy sensor data Gaussian noise was added to the senor readings during testing to see how the network could handle inexact information. From the result of this conclusion about the robustness of the method could be drawn.

4
Results

This section covers the findings of the project. Firstly, the game Grid World is discussed. Next, the results of the single-scenario training are presented. Most of the parameter sweeps were conducted on a training set of one single scenario and results covering the impact of different parameters will therefore be presented. Thereafter, results of training on larger sets of traffic scenarios are presented. The ability to generalise is evaluated and the effect of a curriculum learning strategy illustrated. Moreover, the performance of the neural network trained with a genetic algorithm is evaluated. Lastly, the two neural network based active safety functions are compared to a model based collision avoidance system developed by the Volvo Car Corporation.

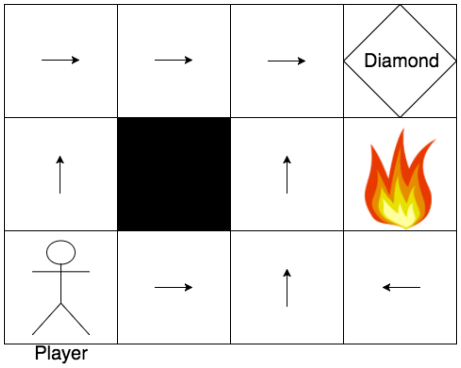
4.1 Verifying the Implementation on Grid World

The optimal policy in Grid World is dependent on the reward function. To illustrate the importance of the reward function two different reward functions were tested and the learnt policies were compared. In Eq. (4.1) and Eq. (4.2) the two reward functions are described.

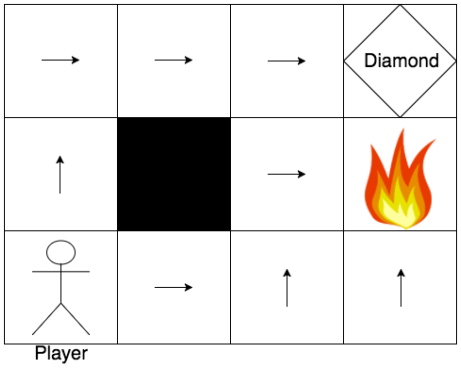
$$R_1 = \begin{cases} 1 \text{ if goal is reached} \\ -1 \text{ if player falls into the pit} \\ -0.1 \text{ otherwise} \end{cases} \tag{4.1}$$

$$R_2 = \begin{cases} 1 \text{ if goal is reached} \\ -1 \text{ if player falls into the pit} \\ -3 \text{ otherwise} \end{cases} \tag{4.2}$$

The first reward function, described in (4.1), should encourage the player to take the shortest path to the goal. The second reward function, described in (4.2), on the other hand should result in an undesirable behaviour close to the fire pit. Due to the size of the punishment received at each step the optimal action when being closer to the pit than the goal is to go straight to the pit. This shows that two similar reward functions can lead to different behaviour. In both cases the agent trained with the deep Q-learning algorithm learnt the optimal policy.

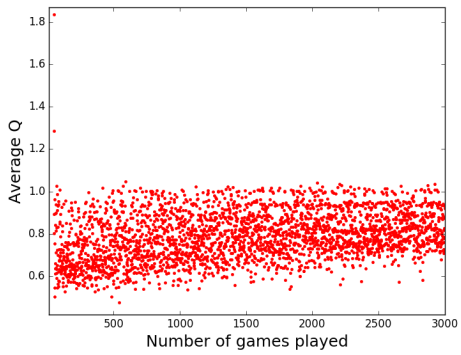


(a) The policy learnt with a living reward of -0.1. The player takes the shortest path to the goal.

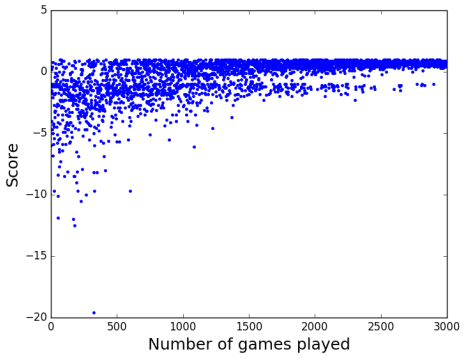


(b) The policy learnt with a living reward of -3. When the player is close to the pit it expects a higher total reward from jumping into the pit compared to staying alive and reaching the goal.

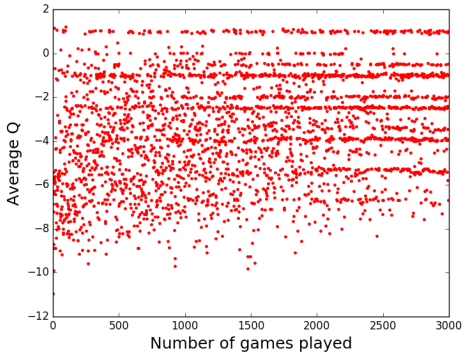
Figure 4.1: Optimal policies for the game Grid World given two different reward functions.



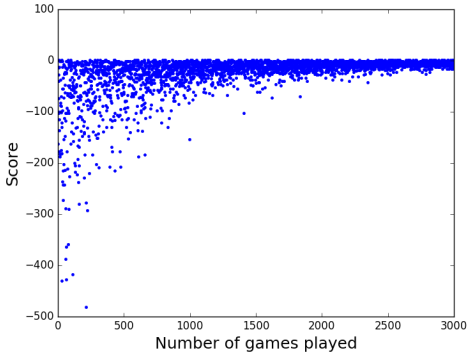
(a) Living reward = -0.1



(b) Living reward = -0.1



(c) Living reward = -3



(d) Living reward = -3

Figure 4.2: Score per episode and average predicted Q-values during 3000 training episodes.

It is also interesting to study convergence of the network. The score per episode as well as the average predicted Q-value are shown in Fig. 4.2. The score per episode clearly increases and stabilises during training and the Q-values also seem to converge. Depending on the starting point of the player the agent can expect a different total sum of rewards, which explains why the Q-values vary and discrete lines are formed.

4.2 Training a DQN Agent for Collision Avoidance

Firstly, it is important to clarify that the velocity of the host vehicle was not forced to zero after an intervention during training since it was found that it was very hard for the agent to learn such temporally extended planning strategies. Fig. 4.3 shows an example of how the Q-network failed to converge when forcing the velocity of the host vehicle to zero.

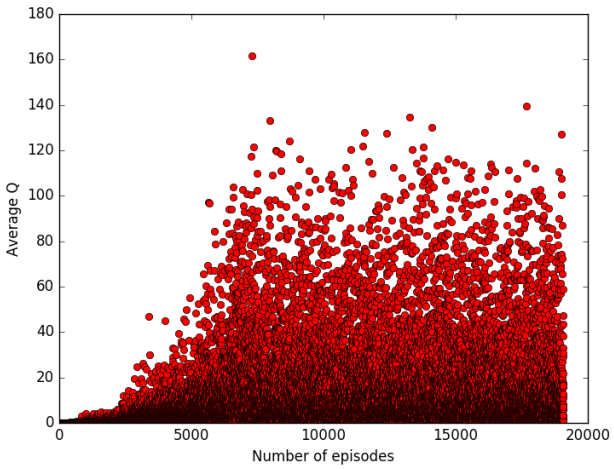


Figure 4.3: An example of how the Q-network failed to converge when forcing the velocity of the host vehicle to zero.

4.2.1 Parameter Sweeps

To methodically examine the impact of key parameters on the performance of the algorithm, it is advisable to vary one parameter at a time. Some initial tuning was necessary to find windows of parameter ranges where learning is successful. A stable algorithm, even though not finely tuned, can give hints about the impact and importance of different parameters. The default parameter values were the following:

The reward function used for parameter tuning was a constant reward for collision, velocity dependent living reward, a time-to-target dependent punishment for braking and a constant sweet spot reward in the range (0.0,1.0) m from the target vehicle. Both double DQN and prioritised experience replay were investigated during the parameter sweeps.

Parameter	Description
Network depth	6 hidden layers
Network width	10 input, 2 output nodes and 6 hidden layers with 100, 50, 50, 30, 20, 10 nodes respectively.
Optimisation algorithm	Optimiser 'Adam' with the learning rate 10^{-6}, and exponential decay rates $\beta_1 = 0.9, \beta_2 = 0.999$.
Number of training episodes	2000
Agent history length	1
Initial exploration	0.5
Final exploration	0.01
Final exploration episode	1800
Batch size	32
Target network update frequency	100
Discount factor	0.99

Table 4.1: Default parameter settings during the parameter sweeps on a training set of one single scenario.

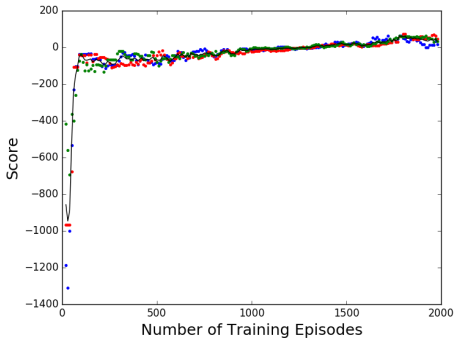
4.2.1.1 Stability of the Algorithm

To draw conclusions from a parameter sweep the algorithm needs to be stable. To examine the stability the score per episode and resulting policy were studied during three runs with constant parameter settings. Fig. 4.4a shows the score per episode during three runs and Fig. 4.4b shows how the policy varies as a function of the longitudinal distance to the target vehicle when remaining parameters were kept constant. The graphs are plotted for $v_{\text{host}} = 20\,\text{m/s}, a_{\text{host}} = 1\,\text{m/s}^2, v_{\text{target}} = 10\,\text{m/s}$ and $a_{\text{target}} = -1\,\text{m/s}^2$.

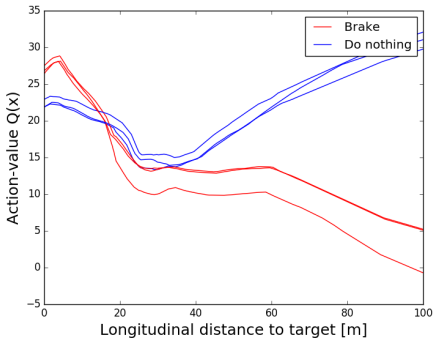
From the figure it is clear that the learnt policy is very similar in all three runs. The agent brakes at distances larger than approximately 20 m from the target. The growth of the score per episode is also fairly similar between the runs. Some noise is present as expected but the convergence is clear.

To investigate how a policy is taking shape during training policies at intermediate steps of training were examined. Fig. 4.5 shows how the policy varies as a function of the longitudinal distance to the target vehicle when remaining parameters were kept constant at $v_{\text{host}} = 20\,\text{m/s}, a_{\text{host}} = 1\,\text{m/s}, v_{\text{target}} = 10\,\text{m/s}$ and $a_{\text{target}} = -1\,\text{m/s}$ respectively.

When training starts the agent do not know anything about the environment and the policy should be considered as completely random. After 1000 training episodes the agent brakes when the target vehicle is at a distance of 13.7 m. After 1800 episodes this distance has increased to 16.9 m and in the end after 2000 episodes the distance is 16.3 m. The policy seems to take shape steadily. The fluctuations are much smaller towards the end of the training, which indicates convergence.

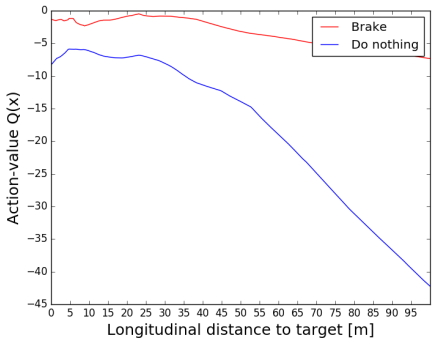


(a) Three runs with the same parameter settings to show how the score per episode increases during training and how it varies between runs. The colours red, blue and green corresponds to different runs.

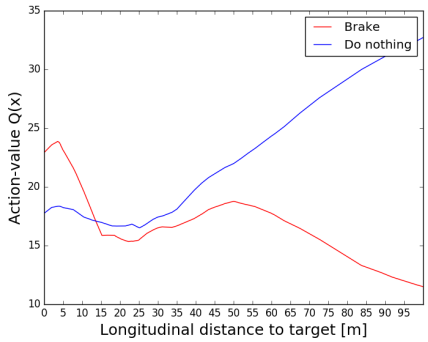


(b) How the policy varies between three runs with the same parameter settings. The agent learns a consistent policy where the brakes are applied when coming closer than a critical distance from the target vehicle.

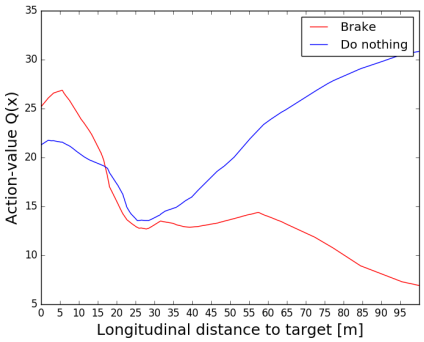
Figure 4.4: Figure illustrating the stability of the algorithm.



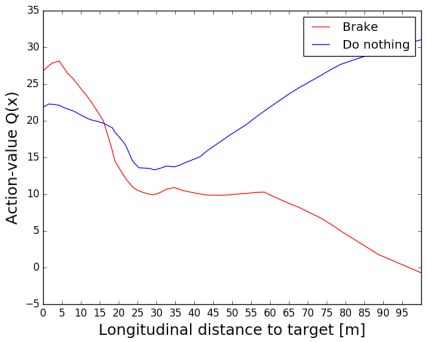
(a) Initial braking policy (random weights)



(b) Braking policy after 1000 episodes.



(c) Braking policy after 1800 episodes

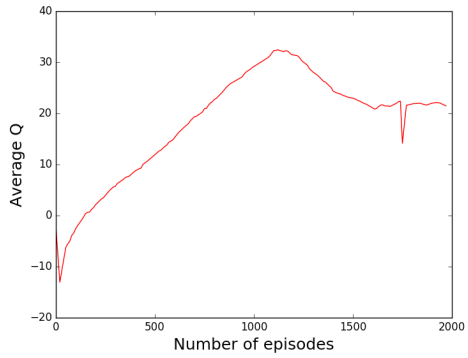


(d) Braking policy after training is complete.

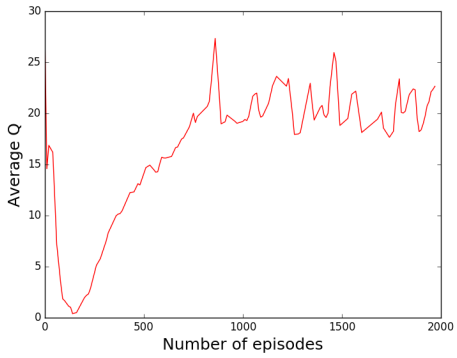
Figure 4.5: Four figures showing the braking policy during training. Initially the braking policy is random, but towards the end it has stabilised.

4.2.1.2 Network size

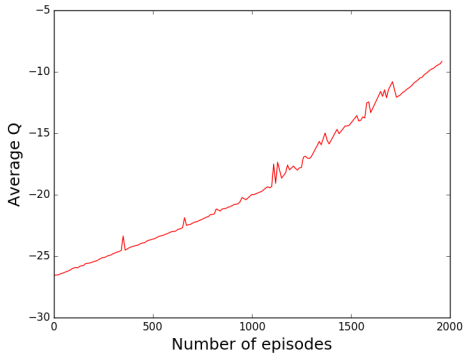
Four networks of different sizes were trained and evaluated to get an indication of the required sizes of the network. The findings indicate that a small network is not able to accurately represent the Q-function. Ideally the Q-function should converge and correspond to the expected total reward, when taking into account the discount factor. The Q-function did not converge when a too small network was used. Two small networks, one with only one hidden layer containing ten nodes and one with two hidden layers with 20 nodes in each layer were tested. A much wider network with four hidden layers containing 500, 250, 250 and 150 nodes was also studied. Lastly, a relatively deep representation with six hidden layers with 100, 50, 50, 30, 20 and 10 nodes respectively was investigated. Fig. 4.6 shows the average predicted Q-values of each of the four networks and Fig. 4.7 shows the corresponding score per episode.



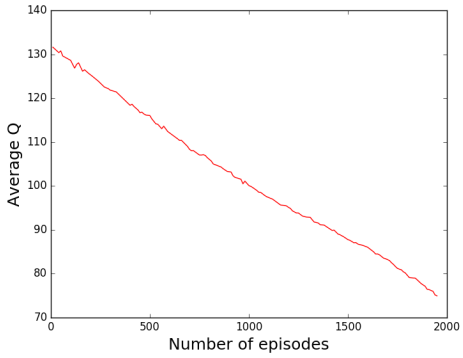
(a) Average predicted Q-values for the deepest network.



(b) Average predicted Q-values for the widest network.



(c) Average predicted Q-values for the smallest network.

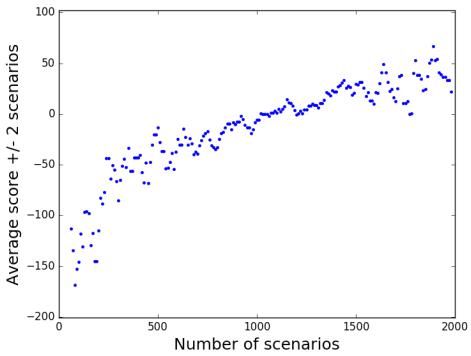


(d) Average predicted Q-values for the second smallest network.

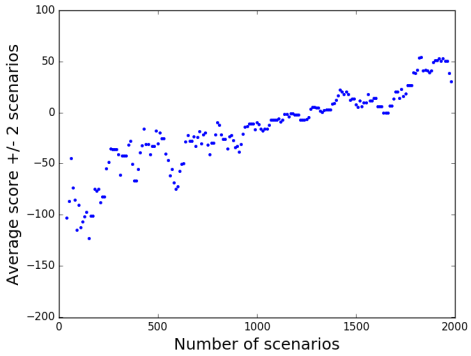
Figure 4.6: Four figures showing average predicted Q-values during training. It can be seen that a large network is required for convergence.

The smallest network predicts a negative total reward even though the score itself is positive in the end. The second smallest network predicts a positive total reward but it does not converge. The two large networks converge to roughly the same value, where the widest network has larger variations in the average predicted

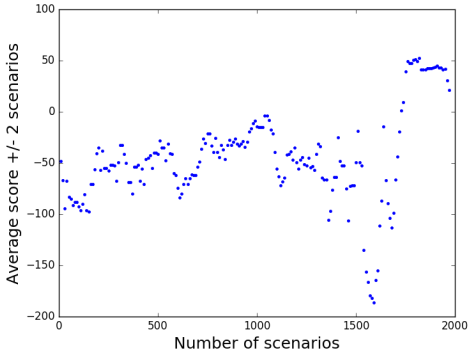
Q-value. However, by looking at the score per episode, the largest network is slightly more stable towards the end of training, but to see if this really is the case a lot of runs with the same settings have to be made.



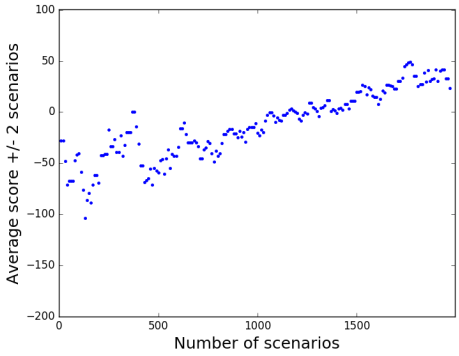
(a) Score per episode for the reference network for 2000 training episodes.



(b) Score per episode for the largest network for 2000 training episodes.



(c) Score per episode for the smallest network for 2000 training episodes.



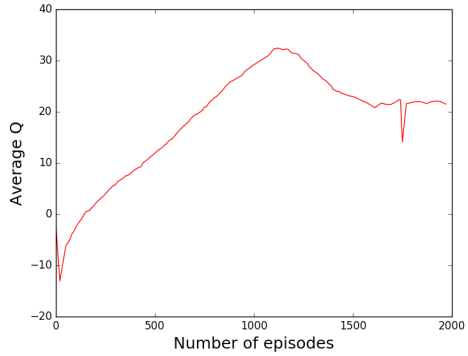
(d) Score per episode for the second smallest network for 2000 training episodes.

Figure 4.7: Four figures showing the score per episode. A larger network shows better increase in score during training.

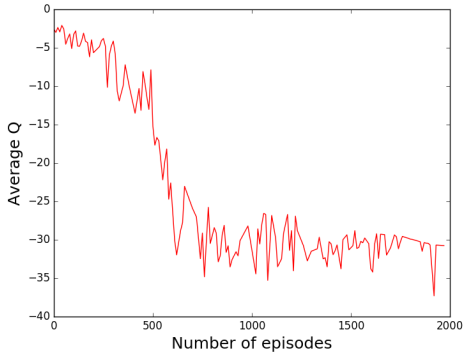
When training on a single scenario a larger network performed better than a smaller network. Training on a larger set of scenarios is potentially much more challenging and will probably require large networks. For this reason only the two larger networks, the widest and the deepest, were trained on 250 traffic scenarios for 5000 episodes. The network needs to be large enough, but a larger network requires longer training times. Worth noticing is that the wider network contains 230 952 nodes whereas the other network only contains 10 982 nodes. On average, the wider network takes approximately 1.6 times longer to train per episode than the narrower but deeper network. Due to computational cost we could not train the wider network until convergence and it is possible that the end result would have been better given sufficient training.

4.2.1.3 Optimisation Algorithm

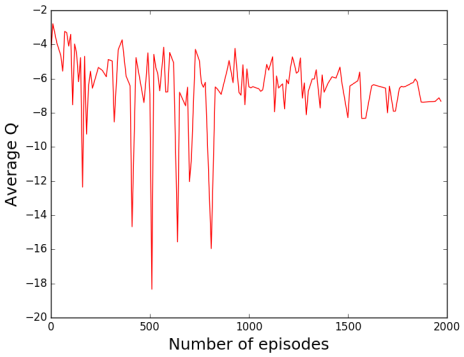
Several optimisation algorithms can be used to optimise a neural network and it is not known a priori which algorithm that works best for a specific problem. Four popular algorithms: (1) stochastic gradient descent, (2) stochastic gradient descent with Nesterov momentum, (3) RMSProp and (4) Adam were therefore tested. For the investigation to be extensive the hyperparameters of the optimisation algorithms should be tuned before any comparison. However, since such an investigation would be very time consuming and the aim was only to find a promising candidate default hyperparameter settings as described in section 2.1.5 were used. The learning rate was set to 10^{-6}. To get an indication of which of the algorithms that works best, the evolution of the score per episode and average predicted Q-values were studied. The average predicted Q-values are shown in Fig. 4.8 and the score per episode for each optimiser is shown in Fig. 4.9.



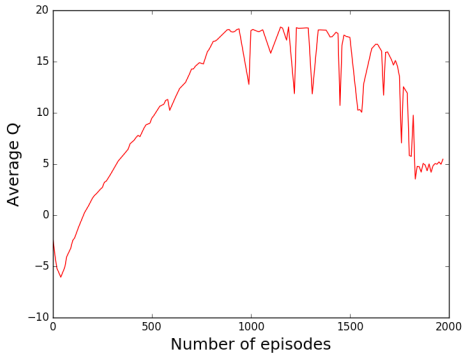
(a) Average predicted Q-values using the optimiser Adam.



(b) Average predicted Q-values using stochastic gradient descent.



(c) Average predicted Q-values using stochastic gradient descent with Nesterov momentum.

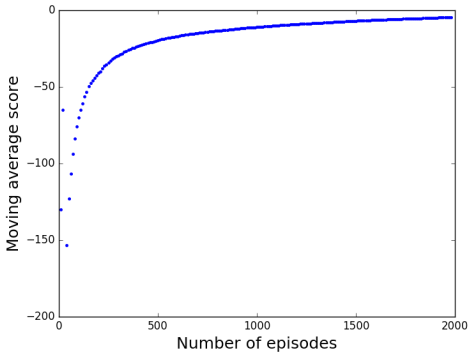


(d) Average predicted Q-values using RMSProp.

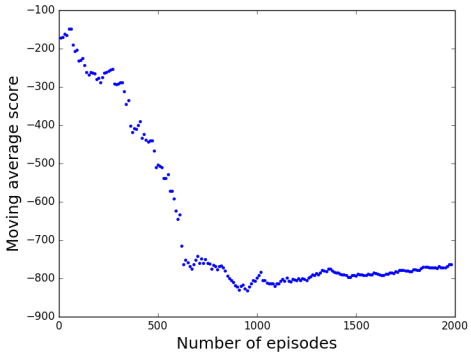
Figure 4.8: Four figures showing average predicted Q-values during training with different optimisation algorithm. The choice of optimisation algorithm has a large impact on the results.

Based on the findings Adam seems to be the most promising candidate. When using Adam the score per episode is increased steadily and the Q-values converge

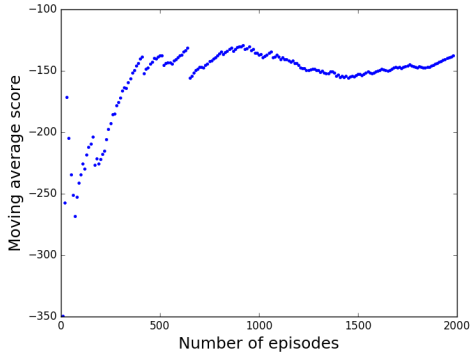
nicely. Stochastic gradient descent fails to improve performance and when adding Nesterov momentum the learning is still rather irregular. Adam is more stable and slightly faster than RMSProp.



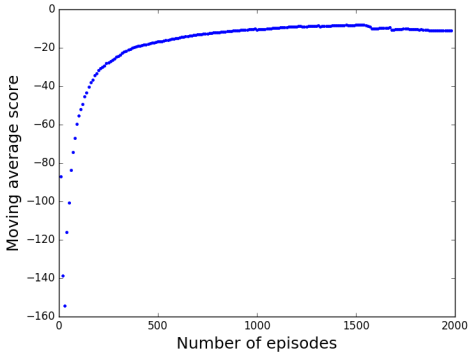
(a) Moving average score for 2000 training episodes using Adam as optimisation algorithm.



(b) Moving average score for 2000 training episodes using stochastic gradient descent.



(c) Moving average score for 2000 training episodes using stochastic gradient descent with Nesterov momentum.



(d) Moving average score for 2000 training episodes using RMSProp.

Figure 4.9: Four figures showing the moving average score using different optimisation algorithms. Adam has the steadiest increase in score per episode and training using stochastic gradient descent is actually worse than the initialisation.

4.2.1.4 Reward Function

As has been described it is difficult to find reliable indicators of the agent's progress. Score per episode and average predicted Q-values can be used to track the progress, however, since the reward function directly affects these two metrics they cannot be used when comparing the quality of different policies. When experimenting with various reward function the quality of a policy is best summarised by letting the agent control the vehicle and study how collisions are avoided. Score per episode and average predicted Q-values are, however, still useful to analyse the learning process.

14 different reward functions were investigated. The training set consisted of one single scenario. Only three out of 14 reward functions lead to a policy where the agent learnt to avoid the collision in training scenario in a satisfying way. Table 4.2 shows the 14 different reward functions used. Reward function number 6, 10 and 14 lead to the agent braking such that the minimum distance was less than one meter. Four other reward functions lead to the agent braking, two of which were too late (number 3 and 9) and two of which were too early (number 11 and 12). Using any of the other seven reward functions, the agent crashed into the vehicle in front without applying the brakes.

	Collision	Living reward	Braking	Sweet spot
1	-100 - $10v_{\text{rel}}$	v_{host} + 1	-living reward	not used
2	-1 - v_{rel}	v_{host}	-t_{target}	not used
3	-1 - v_{rel}	v_{host}	-D_{target}	not used
4	-v_{rel}	0	-t_{target}	10 $\forall D_{\text{target}} \in (0.5,1.0)$ m
5	-v_{rel}	0	-t_{target}	10 $\forall D_{\text{target}} \in (0.0,1.0)$ m
6	-v_{rel}	0	-D_{target}	10 $\forall D_{\text{target}} \in (0.5,1.0)$ m
7	-1	v_{host}	1/2 living reward	10 $\forall D_{\text{target}} \in (0.5,1.0)$ m
8	10 - v_{rel}	0	-$10t_{\text{target}}$	10 $\forall D_{\text{target}} \in (0.0,1.0)$ m
9	-100 - $10v_{\text{rel}}$	1	-$10t_{\text{target}}$	-1 $\forall D_{\text{target}} \in (0.0,0.5)$ m
10	1 - v_{rel}	0	-t_{target}	10 $\forall D_{\text{target}} \in (0.5,1.0)$ m
11	-1	v_{host}	-t_{target}	10 $\forall D_{\text{target}} \in (0.5,1.0)$ m
12	-1 - v_{rel}	0	-$0.1t_{\text{target}}$	10 $\forall D_{\text{target}} \in (0.0,1.0)$ m
13	-1 - v_{rel}	0.1	-t_{target}	10 $\forall D_{\text{target}} \in (0.0,1.0)$ m
14	-1	0	-t_{target}	10 $\forall D_{\text{target}} \in (1.0,0.5)$ m

Table 4.2: The different rewards functions used in the performance comparison between 14 different reward functions.

4.2.1.5 Parameters with Indistinct Impact

Altering remaining parameters only had a small impact on the results. One parameter was altered at the time and reward function number 14 in table 4.2 was used. The discount factor γ gave similar results for $\gamma = 0.95$ as for $\gamma = 0.99$. If it was more dramatically decreased to $\gamma = 0.5$ the magnitude of the Q-values decrease as future reward have less significance. However, the quality of the policy was neither higher nor lower.

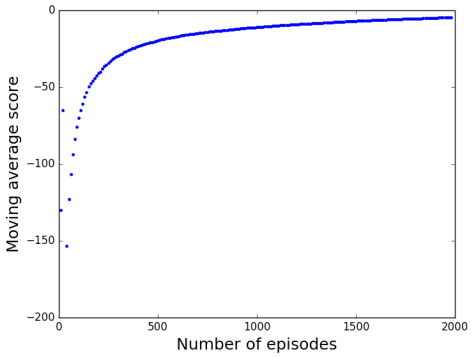
In addition, the target network update frequency and batch size did not seem to impact the results significantly. Tests were performed on a target network update frequency of 10, 100 and 1000, and the batch size was set to 16 and 64.

The two different loss functions, squared loss and huber loss, did not seem to have any distinct influence on the resulting policy.

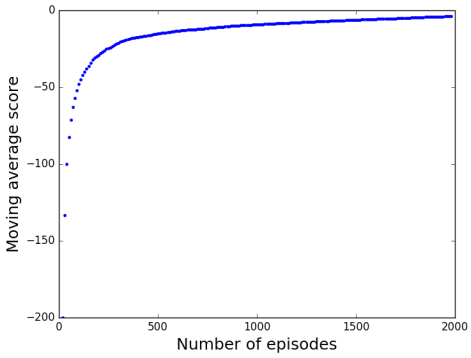
Several different initial and final exploration rates were tested where the initial exploration rate was set to a value between 0.2 and 1 and the final exploration rate was between 0.1 and 0.01. Learning was successful in all cases and no single setting was found to be significantly better than the other.

4.2.2 Improvements of the Algorithm

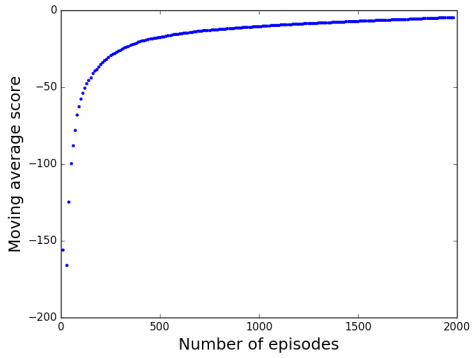
To see if prioritised experience replay and double DQN improve performance the moving average of score per episode and learnt policies of modified algorithms were studied. Fig. 4.10 shows the moving average of score per episode when the agent was trained with and without prioritised experience replay and double DQN. Judging from these graphs there is not much of a difference between when adding prioritised experience replay and double DQN. However, without prioritised experience replay the braking policy lacks consistent distance dependency, see Fig. 4.11. Close to the target vehicle the host ceases to brake. This is undesirable and suggests that prioritised experience replay improves the quality of learnt braking policies.



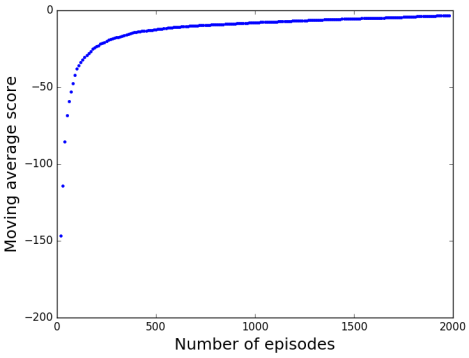
(a) Moving average of score per episode with both double DQN and prioritised experience replay.



(b) Moving average of score per episode with double DQN but no prioritised experience replay.



(c) Moving average of score per episode with prioritised experience replay but no double DQN.

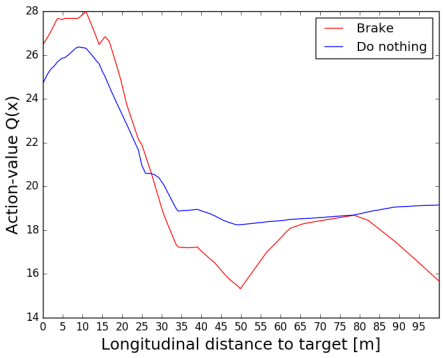


(d) Moving average of score per episode without double DQN and without prioritised experience replay.

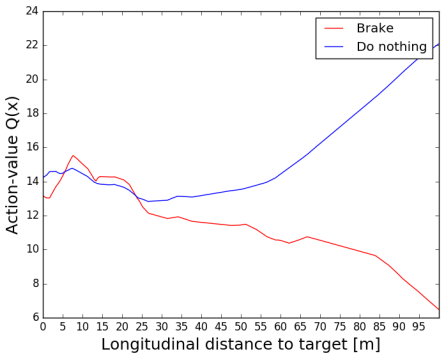
Figure 4.10: Four figures showing what happens to the moving average of score per episode when using different combinations of prioritised experience replay and double DQN.

Double DQN also seems to improve the performance. Without double DQN the brakes are generally applied at a later stage and this leads to a higher relative speed at impact.

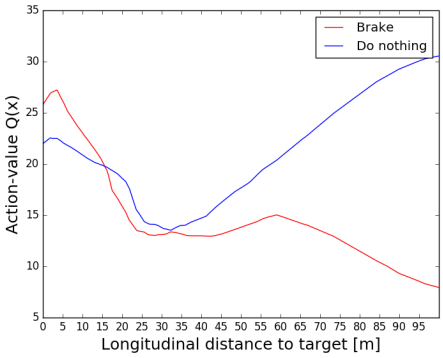
Prioritised experience replay is slower than the original version of experience replay, but the additional computation time can be reduced if a sum tree is used for storing the priorities and the experience memory size is kept at a reasonable size.



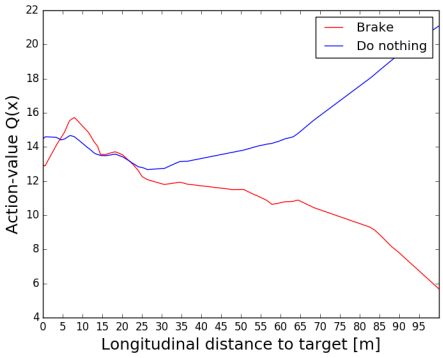
(a) Braking policy with both double DQN and prioritised experience replay.



(b) Braking policy with double DQN but no prioritised experience replay.



(c) Braking policy with prioritised experience replay but no double DQN.



(d) Braking policy without double DQN and without prioritised experience replay.

Figure 4.11: Four figures showing the learnt policies when using different combinations of prioritised experience replay and double DQN.

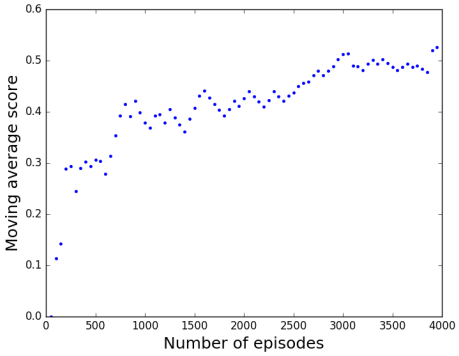
4.2.3 Adding Training Data for Improved Generalisation

For a neural network to be useful as a collision avoidance system it has to generalise well to a large set of traffic situations. Understandably, good generalisation was never obtained when the training set only consisted of one single traffic scenario. The best network was tested on a reference test set consisting of 1000 scenarios. The best network was trained for 2000 episodes using reward function number 14 in table 4.2. Out of these 1000 scenarios, 516 were scenarios where emergency braking was necessary to avoid a collision. The DQN agent managed to avoid 344 collisions, however, it also applied the brakes in 423 scenarios when it never was necessary. Moreover, most of the successful interventions came much too early leading to a large minimum distance to the target vehicle during braking. More training data

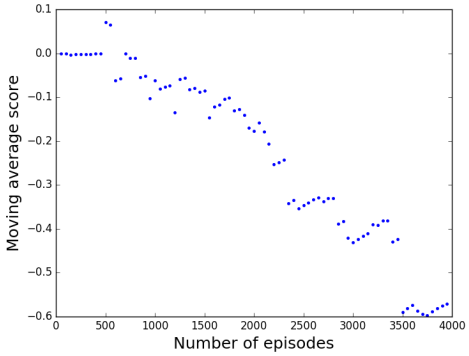
was undoubtedly needed. A curriculum learning approach was used to introduce more traffic scenarios to the training set.

4.2.3.1 Curriculum Learning

Curriculum learning can improve learning by letting the agent learn to handle simple scenarios first before introducing more complex problems. To see the effects of curriculum learning a network with four hidden layers with 500, 250, 250 and 150 nodes respectively was trained for 4 000 episodes using two different approaches. In the case of curriculum learning, 50 training scenarios were sequentially added to the training data. In the other case a new scenario was generated at each episode. The results suggest that curriculum learning could improve learning. The moving average of score per episode for both approaches is shown in Fig. 4.12a and Fig. 4.12b. Without curriculum learning the agent is not improving its score per episode at all during training. Curriculum learning is a much more promising approach as the agent is learning a policy that is improving the score per episode.



(a) Moving average of score per episode when using a curriculum learning approach.



(b) Moving average of score per episode without curriculum learning.

Figure 4.12: Comparison of the moving average of score per episode with and without curriculum learning.

4.2.4 Generalisation

Training a neural network with six hidden layers containing (100, 50, 50, 30, 20 and 10) nodes on 250 scenarios sequentially added for 5000 episodes the DQN agent manages to decrease the relative impact velocity on the test set, see Fig. 4.13. Out of 516 collisions scenarios it collides in 251 of these. However, the goal is to end up as close to the target vehicle as possible. As seen in Fig. 4.14 the host vehicle often brakes too early. Ideally we would like to end up within one meter of the target vehicle but the minimum distance is often much larger.

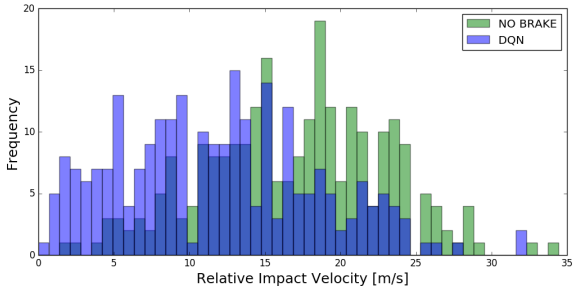


Figure 4.13: A histogram showing the relative impact velocity. The DQN agent shifts the bars to the left meaning that the impact velocity on average was decreased.

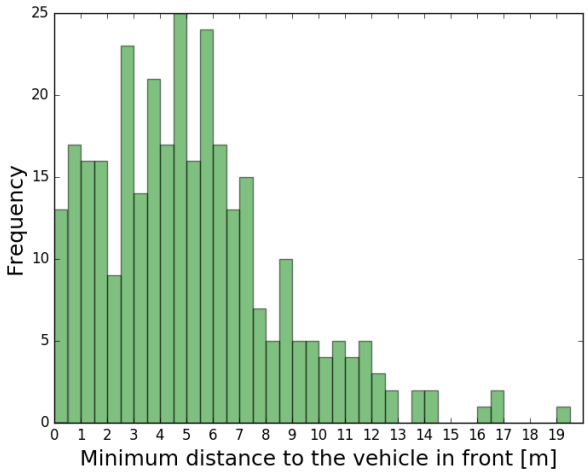


Figure 4.14: A histogram of the minimum distance to the target vehicle after an intervention.

4.3 Genetic Algorithm

The fitness measure is central when evaluating the success of the of the GA. Based on the performance on the test set it was found that a suitable way of calculating the total fitness was as a linear combination of the average $\bar{F}$ fitness over the set of scenarios and the worst evaluation $\min F_i$. The worst evaluation was included to promote consistency. The measure used was

$$F_{tot} = 0.8\bar{F} + 0.2 \min F_i \tag{4.3}$$

The fitness measure resulting in the best performance was based on the same idea as the reward functions in the deep Q-learning algorithm. The distance travelled served as starting point to encourage the networks to not intervene too early. A minimum distance to the target vehicle during braking larger than 1 m was penalised: the further a way the larger penalty. Collisions were penalised proportional to the relative impact velocity.

Roulette wheel selection with fitness ranking was found to be the most suitable selection method. Roulette wheel selection with and without ranking and tournament selection with selection probability of 0.8 and tournament sizes of 3 and 5 were compared by measuring the maximum validation fitness over 5 training runs.

In Fig. 4.15 the training and validation fitness as functions of generation are shown. The training fitness is monotonically increasing since the best individual at all times is copied to the next generation. The validation fitness has its peak at generation number 37. This network has the best ability to generalise. From the figure it is not entirely clear that the maximum attainable validation fitness has been reached. This can never be guaranteed, however, the population of the 31st generation was retrained several times to investigate if even higher fitness values could be achieved. No improvement was observed in three additional training runs. The same training set was used as when training the best performing DQN agent.

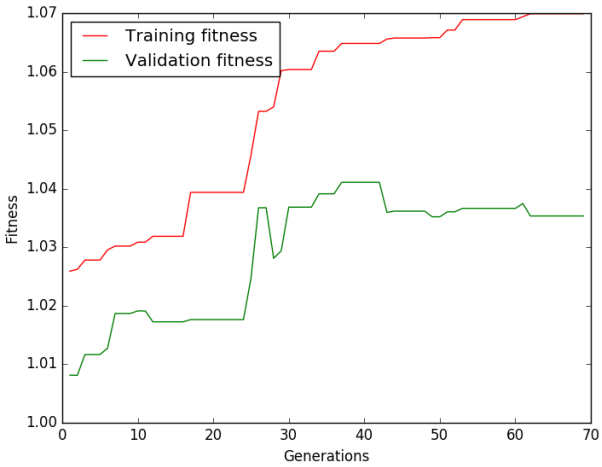


Figure 4.15: The training and validation fitness as functions of generation. The x-axis measures the number of training epochs and the y-axis the fitness.

After training, a test set of 1000 traffic scenarios was used to evaluate the performance of the ANN. Three different metrics were used to measure the performance: (1) the total number of collisions, (2) the minimum distance to vehicle in front during braking and (3) the relative impact velocity.

The result of the best performing network should be compared to the result of doing nothing. Without emergency braking the number of collisions would have been 516. When the network controlled emergency braking 65 % of the collisions were avoided. It is, however, not only the number of avoided collisions that is of interest when evaluating the performance of the collision avoidance system. Equally important to study is at what times the brakes are applied. Studying the minimum distance to vehicle in front during braking does this. Fig. 4.16 shows a histogram of the minimum relative distance to the target during braking. Collisions are left out. From the histogram it can be seen that the maximum minimum relative distance to the target is between 8 and 8.5 m and that most interventions results in a minimum relative distance to target between 0.5 and 2.5 m.

Even though only 65 % of the total number of collisions were avoided the relative

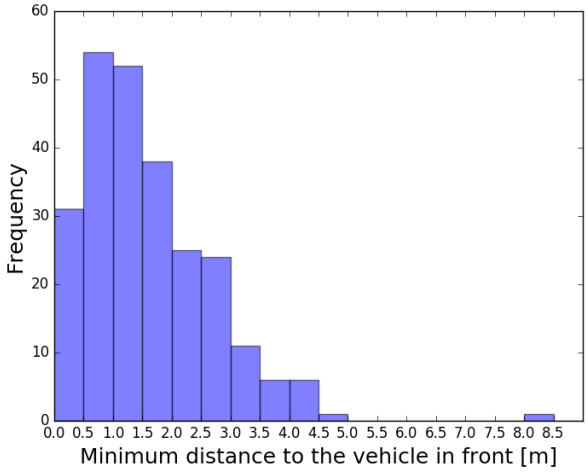


Figure 4.16: A histogram of the minimum relative distance to the target during braking. Collisions are left out.

impact velocity was reduced in 98 % of the cases. In other words the brakes were applied in 98 % of the scenarios requiring interventions. There were two interventions in scenarios not requiring emergency braking, but where the target vehicle was very close to the host. Fig. 4.17 shows how the relative impact velocity is decreased by the emergency braking system. On average the relative impact velocity was reduced by 58 %.

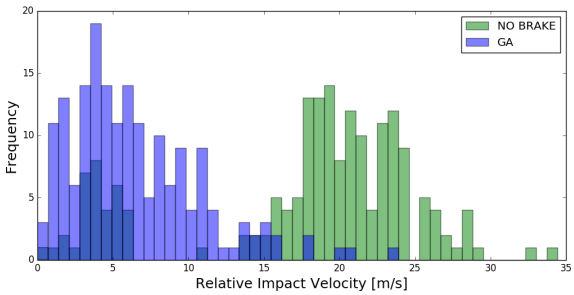


Figure 4.17: A histogram showing the reduction of relative impact velocity. The relative impact velocity was reduced in 98 % of the collisions. On average the relative impact velocity was reduced by 58 %.

4.4 Benchmark

Two different methods to train a neural network to represent emergency braking policies were investigated during this project. To gain understanding of how well the neural network based methods can compete with a model based system developed by the Volvo Car Corporation such a system was also tested on the same test set. The system developed by the Volvo Car Corporation avoided 49 % of the collisions in the test set. It is worth mentioning that some scenarios could be impossible in

the sense that even when applying the brakes immediately it would be impossible to avoid the collision. There are however no collisions during the first second in any scenario used in the test set. Fig. 4.18 shows the minimum relative distance to the target vehicle during braking. Collisions are left out. From the histogram it can be seen that the maximum minimum relative distance to the target is between 2.5 and 3 m and that most interventions results in a minimum relative distance to target between 0.0 and 0.5 m. More than 90 % of the successful interventions had a minimum relative distance to the target vehicle between 0.0 and 0.5 m.

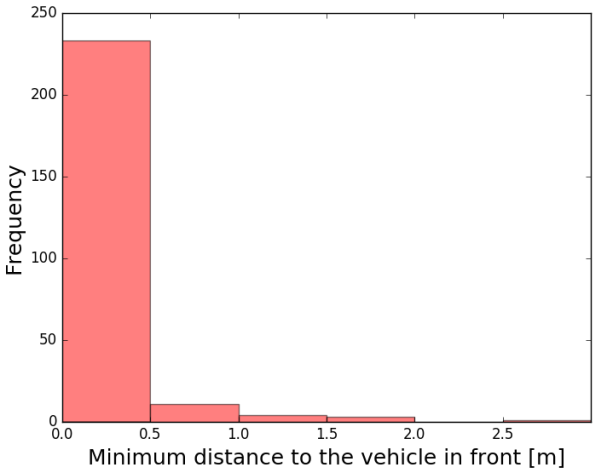


Figure 4.18: A histogram of the minimum distance to the target vehicle after an intervention.

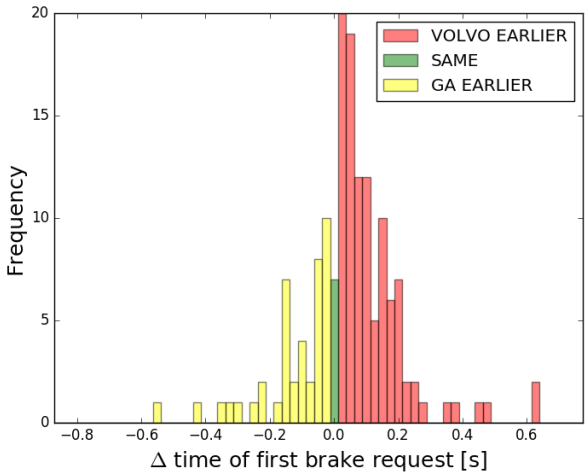


Figure 4.19: A histogram of the time difference for the first brake request.

Another slightly fairer measure of the correctness of an intervention is the time of the first brake request. This since the system developed by the Volvo Car Corporation can alter the braking power and does not force the speed to zero when emergency braking. Fig. 4.19 shows a histogram of the time difference for the first

brake request in successful interventions. For clarity, one scenario is left out in the histogram since the neural network intervened much earlier in this scenario. The time difference of 7 s suggest that the systems braked for two different threats. This intervention is therefore seen as a false intervention, however, included when calculating the average time difference. The neural network trained with a GA intervenes slightly later on average, but the difference is only 0.014 s.

4.5 Noisy Data

When adding noise to the input it is likely that the results on the test set decline. To investigate the behaviour on perturbed input data Gaussian noise with mean 0 and standard deviation 2.5 % of the correct value was added to the measures of the target position, velocity and acceleration.

In the case of deep Q-learning the agent collides in 257 scenarios out of 516 when noise is added compared to 251 collisions without. The distribution of minimum distance to the target vehicle is shown in Fig. 4.20. We see that the distribution remains similar but is not identical to the noise-free case. This is expected because the input is what determines the action that the agent takes in each time step.

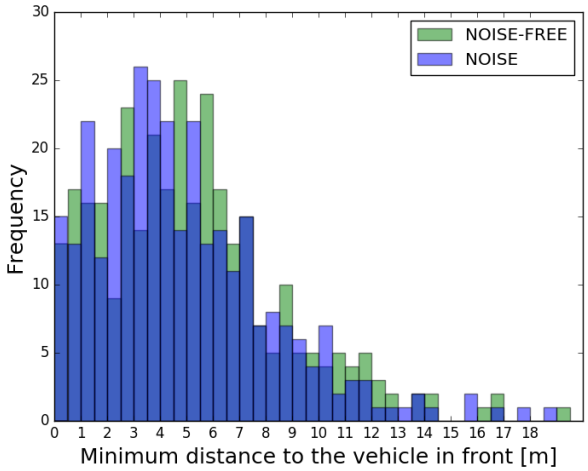


Figure 4.20: A histogram showing the minimum distance to the target vehicle after an intervention. The blue distribution was obtained with perturbed input data, and the green distribution without any misreadings. The network was trained with deep Q-learning.

The trend is the same when using perturbed input data to the network trained with a genetic algorithm. In Fig. 4.21 it can be seen how the distribution of the minimum distance to the target vehicle during braking remains very similar. The distribution is slightly broader as expected. The number of collisions is decreased slightly.

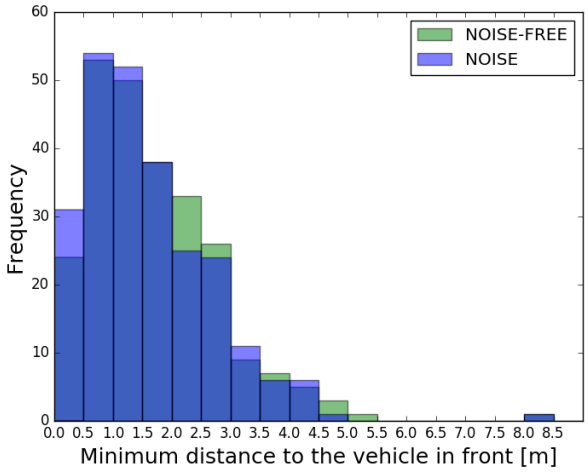


Figure 4.21: A histogram showing the minimum distance to the target vehicle after an intervention. The blue distribution was obtained with perturbed input data, and the green distribution without any misreadings. The network was trained with a GA.

5
Discussion

After systematically investigated the impact of a number of hyperparameters on the performance of the deep Q-learning algorithm it is clear that the algorithm is very sensitive to the choice of hyperparameters. It is undoubtedly difficult to find settings for which the algorithm reliably learns high-quality policies. Outside narrow windows of parameter ranges learning is unsuccessful even on greatly simplified problems.

Two proposed adaptation to the original deep Q-learning algorithm were tested; prioritised experience replay and double DQN. The results indicate that these suggested improvements can improve the quality of the learnt policies and stability of the algorithm. However, the adaptions are not any guarantee for stability or high quality policies. The idea behind using prioritised experience replay was that the DQN agent can learn more effectively from some transitions than from others. Braking and collisions are very important events, but also very rare in any realistic simulation of traffic. Prioritised experience replay is a strategy where important transitions are sampled more frequently. This way learning is more efficient, even if the adaptation increases computational time.

One problem with deep Q-learning is that it is difficult to find reliable indicators of progress in learning. In supervised learning the progress can be measured by studying the value of a loss function. However, in deep Q-learning and reinforcement learning techniques in general the value of the loss function does not immediately reveal the progress rate. The Q-function is a moving target, which means that a reduction of the loss function does not necessarily imply progress in learning. Changes in the magnitude of the predicted Q-function will also affect the values of the loss function. For this reason other measures are needed to track progress, such as score per episode, average predicted Q-value and evaluation of learnt policies. This is unfortunately a noisy, slow and computationally costly process that complicates parameter tuning.

It was sometimes difficult to determine the impact of some parameters. For example the impact on the performance of the discount factor was not clear. Only a few different values could be tested and the noise nature of the performance metrics made it difficult to determine if there was any immediate effect of altering the value of the parameter. However, it has been suggested that learning only is successful for very narrow windows of discount factor [27]. It is therefore possible that the parameter was varied too bluntly to see any real impact.

Adam was the most successful optimiser for the problem. Adam is designed to work well for online learning models and for problems with sparse or noisy gradients. DQN is an online learning model and rare and important events, such as collisions

and braking, may have caused sparse gradients. This might explain why Adam worked well for this particular problem. It is important to remember that the hyperparameters of the optimisation algorithms were not tuned before the different algorithms were compared. Google DeepMind successfully used RMSProp for their application [3]. It may therefore seem likely that RMSProp is the most suitable optimiser. However, it has recently been showed that the deep Q-learning is very sensitive to the hyperparamter selection of the RMSProp algorithm [27]. This might explain why RMSProp was less stable and efficient than Adam.

The deep Q-learning algorithm proved to be more sensitive to the choice of reward function than expected. Needless to say is that the reward function should affect the learnt policies. The reward function informs the agent about what is good and bad. However, our findings suggest that the choice of reward function also can dramatically affect the learning performance. Two reward functions that are essentially rewarding and penalising same things can lead to vastly different policies which is demonstrated in table 4.2. In addition, A. Yu, R. Palefsky-Smith, and R. Bedi have shown that two reward functions that seem very similar still can have a dramatic effect on learning [25]. Based on these findings it is likely that finding a reward function from which an DQN agent can reliably learn good braking policy is very difficult. By extension, this means that it might be hard to extend the algorithm to applications outside video games with well-defined rewards.

Training an artificial neural network (ANN) with a GA has shown to be more stable than deep Q-learning. An advantage with the GA approach is the lower number of hyperparameters. There are also more recommendations found in the literature about suitable parameter ranges. Moreover, when optimising an ANN with a GA the performance of the network is given a single scalar value at the end of the evaluations. This is an important and helpful difference to the deep Q-learning approach since suitable rewards at each time step is difficult to determine when the result of actions may be evident first many time steps later.

It is important that the ANN is large enough. We have seen that a very small network fail to approximate the total future reward. However, there are problems related to training a large network. Optimising the ANN is computationally costly and the response time increases. Given a constant number of training episodes a larger network is not necessarily better. A large network has a double whammy effect on training time. More parameters to tune means that updating the weights takes more time and more training episodes are likely required to optimise the network.

A much smaller network could be used when trained with a GA. This is likely due to the fact that approximating the Q-function in deep Q-learning is more complex than simply estimating the most beneficial action as done by the network trained with the GA. When optimising large ANNs, gradient-based methods are more common. If the ambition is to extend the system to more traffic scenarios or even extend the number of allowed actions a genetic algorithm could eventually be too inefficient to optimise the required network.

Due to computational cost much less training data was used than Google DeepMind had for the Atari games. As a comparison a 15 second scenario with a time step size of 0.025 s corresponds to 600 observations. Training for 10000 episodes then gives 6 million observations. In practice the scenarios are often shorter than

15 seconds since they are terminated when there is a collision or one of the vehicles comes to a halt. Google DeepMind trained their agents for 50 million frames which is in the order of magnitude ten times longer than our agent. Training the six-layer network used for these investigations for 10000 episodes takes over 31 hours with the hardware at our disposal. A curriculum learning training strategy improved generalisation. New traffic scenarios were introduced sequentially and not randomly presented. One difficulty with this approach was that it was hard to determine how many traffic scenarios the training set should consist of and how often new traffic scenarios should be presented to the agent. If the same traffic scenario is repeated too many times it may lead to overfitting.

A fundamental difference to the original algorithm used by Google DeepMind was the input data to the network. Google DeepMind used a convolutional neural network (CNN) with raw pixel information as input. In our application we used low-dimensional feature vector observations as input to a feed forward neural network (FFNN). Vehicle sensors are often replaced which would mean that a new network would have to be trained for every distinct set of sensors. In that sense feature vectors are safer to use. On the other hand CNNs have revolutionised deep learning in recent years and proved to be a very powerful tool in a wide range of applications. It is therefore possible to think that a CNN would better approximate the action-value function in the deep Q-learning algorithm.

When comparing the results of the ANN based systems to a model based system developed by the Volvo Car Corporation we saw that the model based system was much more consistent. An overwhelming majority of the successful interventions had a minimum distance to the target vehicle of less than 1 m. Although the network trained with a GA avoided more collisions, it had a less consistent minimum distance to the target vehicle after an intervention. When comparing the two systems it is important to remember that the system developed by the Volvo Car Corporation can alter the braking power and does not force the speed to zero when emergency braking. For this reason the time of the first brake request in successful interventions was also investigated to have a somewhat fairer measure of the correctness of the interventions. In this comparison the two systems performed evenly well.

There is one parameter in the fitness measure that determines the trade-off between avoiding collisions and braking as late as possible. This makes it possible to easily adjust the fitness measure to desired behaviour, but defining desired behaviour is challenging. It is difficult to precisely specify what the desired behaviour is. To what extent is it worse to brake too early than colliding?

None of the networks seem to be very sensitive to noisy input data. The performance differs only slightly compared to the result of the noise-free data. However, noisy data might have more significance in other types of traffic scenarios. Having a noisy measurement of heading angle could lead to more false positives, but since we did not study these kinds of scenarios it is difficult to say.

Lastly, it has been our experience that learning results tend to be reasonably consistent for a given set of hyper-parameters. One challenge was striking a balance between confirming experiment reproducibility and trying new experiments, given limited time and computational resources.

5.1 Future Work

This thesis comprises preliminary investigations of training ANNs for active safety functions. Two methods were investigated where the genetic algorithm (GA) approach gave the most promising results. The lack of input output pairs and problems related to defining suitable rewards for intermediate states made a GA a natural choice of optimisation method. However, very little time was spent on the GA compared to the deep Q-learning approach. It would therefore have be interesting to see how well a GA approach can perform if more finely tuned. For example such an approach could be extended to include optimisation of both the structure and the parameters of the ANN.

Considering the approach there is some evidence that a larger training data set may lead to better generalisation. To train the algorithm on more training data it would be necessary to optimise the code to run on a GPU or at least make use of parallel programming. With more efficient training it would also be easier to perform a more extensive reward function sweep.

One obvious limitation imposed in this thesis was that only one other vehicle was present in each scenario. In real-world applications often several objects are present at the same time and the number of objects can vary. For a network to handle a varying number of detected objects, it might be possible to use an embedding technique similar to what is used in language processing applications [28]. Another even simpler solution, however less appealing due to its limitations, would be to let the network choose an action for all detected objects one by one to determine if any object requires an intervention. Such an approach could be insufficient in some traffic scenarios where the state of several detected objects must be known simultaneously in order to intervene correctly. This would not be a problem when working with raw sensor data.

In the study only rear-end collisions on straight road were considered. A collision avoidance system should ideally handle all types of traffic scenarios. The investigated methods are, however, equally applicable on other types of scenarios. For this reason it would be interesting to see if several different scenarios could be handled by the same network. When increasing the complexity one potential problem with a genetic algorithm is the way it assumes scenarios to be limited in time. It might be hard to divide real world applications in sequences with defined beginnings and ends. A DQN does not have this limitation due to the discount factor, which limits how much future rewards are taken into account.

It would be interesting to see how well an ANN could perform if it was trained to copy the simulation model by using the simulation model as ground truth. With this method the ANN would never be able to outperform the simulation model. However, this network could perhaps be used when initialising the weights in deep Q-learning or in the GA. Then the initial policy would already be very close to optimal.

6
Conclusion

To summarise this thesis we have investigated two methods to incorporate neural networks in a collision avoidance system. The first approach was based on deep Q-learning, a method developed by Google DeepMind to learn control policies in an on-line setting. The main problems with this approach was the difficulty of finding a stable parameter setting from which the agent could learn a good braking policy. Several factors such as the choice of optimisation algorithm, size of the neural network and the choice reward function all have a strong impact on performance. In particular the reward functions are hard to define since the agent can learn very different policies from seemingly similar reward functions. Although the generalisation on previously unseen scenarios is rather poor the agent can learn decent braking policy on a few simple scenarios. However, on a test set of unseen scenarios it is either too reluctant to brake, resulting in too many collisions or braking too frequently leading to a great amount of false positives.

The genetic algorithm is much more successful. It learns a consistent high quality policy. In addition the size of the artificial neural network required is much smaller and the training time much shorter. The main advantage of the genetic algorithm is that only the performance on complete traffic scenarios is important. It does not have to predict how good a given action is at each time-step, just choose which action is best. Although the GA is able to learn a high-quality policy it not as consistent as the system developed by the Volvo Car Corporation.

We have seen that a neural network can represent a braking policy given low-dimensional feature observations as inputs. In order for the system to be useful in a real world application is has to be more consistent. The consistency is dependent on the trade-off between avoiding collisions and ending up as close to the target vehicle as possible. To achieve a perfect trade-off is difficult in general.

Bibliography

[1] Volvo Car Corporation, "All-new volvo xc90: Two 'world firsts' in one of the safest cars in the world." `https://www.media.volvocars.com/us/en-us/media/pressreleases/148123/all-new-volvo-xc90-two-world-firsts-in-one-of-the-safest-cars-in-the-world`. Accessed: 12-02-2016.

[2] Preferred Networks, "Robot control with distributed deep reinforcement learning." `https://www.youtube.com/watch?v=-YMfJLFynmA`. Accessed: 12-02-2016.

[3] 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.

[4] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[5] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," *arXiv preprint arXiv:1509.06461*, 2015.

[6] P. Lingman and M. Wahde, "Transport and maintenance effective retardation control using neural networks with genetic algorithms," *Vehicle System Dynamics*, vol. 42, no. 1-2, pp. 89–107, 2004.

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

[8] C. M. Bishop, "Pattern recognition," *Machine Learning*, 2006.

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

[10] Y. B. Ian Goodfellow and A. Courville, "Deep learning." Book in preparation for MIT Press, 2016.

[11] B. Xu, N. Wang, T. Chen, and M. Li, "Empirical evaluation of rectified activations in convolutional network," *arXiv preprint arXiv:1505.00853*, 2015.

[12] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *International conference on artificial intelligence and statistics*, pp. 249–256, 2010.

[13] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1026–1034, 2015.

[14] S. Boyd and L. Vandenberghe, *Convex optimization*. Cambridge university press, 2004.

[15] A. M. Saxe, J. L. McClelland, and S. Ganguli, "Exact solutions to the nonlinear dynamics of learning in deep linear neural networks," *CoRR*, vol. abs/1312.6120, 2013.

[16] A. Choromanska, M. Henaff, M. Mathieu, G. B. Arous, and Y. LeCun, "The loss surfaces of multilayer networks," *arXiv preprint arXiv:1412.0233*, 2014.

[17] M. A. Nielsen, "Neural networks and deep learning." `http://neuralnetworksanddeeplearning.com`. Accessed: 12-02-2016.

[18] T. Schaul, I. Antonoglou, and D. Silver, "Unit tests for stochastic optimization," *arXiv preprint arXiv:1312.6055*, 2013.

[19] B. T. Polyak, "Some methods of speeding up the convergence of iteration methods," *USSR Computational Mathematics and Mathematical Physics*, vol. 4, no. 5, pp. 1–17, 1964.

[20] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *Proceedings of the 30th international conference on machine learning (ICML-13)*, pp. 1139–1147, 2013.

[21] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[22] 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*, pp. 1204–1212, 2009.

[23] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[24] J. Vermorel and M. Mohri, "Multi-armed bandit algorithms and empirical evaluation," in *Machine learning: ECML 2005*, pp. 437–448, Springer, 2005.

[25] A. Yu, R. Palefsky-Smith, and R. Bedi, "Deep reinforcement learning for simulated autonomous vehicle control,"

[26] Y. Bengio, J. Louradour, R. Collobert, and J. Weston, "Curriculum learning," in *Proceedings of the 26th annual international conference on machine learning*, pp. 41–48, ACM, 2009.

[27] N. Sprague, "Parameter selection for the deep q-learning algorithm," in *Proceedings of the Multidisciplinary Conference on Reinforcement Learning and Decision Making (RLDM)*, 2015.

[28] Y. Goldberg, "A primer on neural network models for natural language processing," *CoRR*, vol. abs/1510.00726, 2015.

[29] B.-Q. Huang, G.-Y. Cao, and M. Guo, "Reinforcement learning neural network to the problem of autonomous mobile robot obstacle avoidance," in *Machine Learning and Cybernetics, 2005. Proceedings of 2005 International Conference on*, vol. 1, pp. 85–89, IEEE, 2005.

[30] Y. Bengio, "Practical recommendations for gradient-based training of deep architectures," in *Neural Networks: Tricks of the Trade*, pp. 437–478, Springer, 2012.

[31] B.-Q. Huang, G.-Y. Cao, and M. Guo, "Reinforcement learning neural network to the problem of autonomous mobile robot obstacle avoidance," in *Machine Learning and Cybernetics, 2005. Proceedings of 2005 International Conference on*, vol. 1, pp. 85–89, IEEE, 2005.

[32] R. Eldan and O. Shamir, "The power of depth for feedforward neural networks," *arXiv preprint arXiv:1512.03965*, 2015.