Autonomous vehicle control via deep reinforcement learning

Master's thesis in Systems, Control and Mechatronics

Simon Kardell
Mattias Kuosku

Department of Electrical Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2017

Master's thesis 2017:093

Autonomous vehicle control via deep reinforcement learning

Simon Kardell
Mattias Kuosku

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering
Division of Signal Processing and Biomedical Engineering
Chalmers University of Technology
Gothenburg, Sweden 2017

Autonomous vehicle control via deep reinforcement learning
Simon Kardell
Mattias Kuosku

Supervisor: Jonas Karlsson, Berge
Examiner: Lennart Svensson, Electrical Engineering

Master's Thesis EX093/2017
Department of Electrical Engineering
Division of Signal Processing and Biomedical Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover:
Image of the synthetic environment used in this master thesis.

Autonomous driving using deep reinforcement learning

Simon Kardell
Mattias Kuosku

Department of Electrical Engineering
Chalmers University of Technology

Abstract

The automotive industry as well as academia are currently conducting a lot of research related to autonomous driving. Autonomous driving is an interesting topic that holds the potential to benefit society in many ways, such as reduce the number of fatalities and reducing the environmental footprint of modern day traffic.
In this thesis, we investigate Machine Learning algorithms that can automatically learn to control a vehicle based on its own experience of driving. More specifically we employ two Reinforcement Learning (RL) algorithms called Deterministic Policy Gradient (DDPG), and Actor-Critic with Experience Replay (ACER). The algorithms were trained and evaluated in a synthetic environment. The input to both models are images captured by a front-facing camera and internal states of the ego-vehicle, i.e., velocity, acceleration, and jerk. The results presented in this thesis show that current RL-methods are capable of controlling the vehicle steering, using only images to provide information regarding the position of the ego-vehicle. The results also indicate that a driving policy obtained via RL is more robust towards tricky driving scenarios than policies obtained via supervised learning techniques. However, evaluation of data captured from the real world domain is still needed, to verify the usability of models trained on synthetic data.

Keywords: ACER, DDPG, Reinforcement Learning, Deep Learning, Neural Networks, Machine Learning, Autonomous Driving, Unreal 4 engine

Acknowledgements

We would first of all thank **Peter Karlsson** and **Berge**, for making this thesis possible. We would also like to thank the employees, for producing a cheerful atmosphere. A special thanks to our supervisor **Jonas Karlsson** and the Intelligent Systems team for their support and guidance throughout this thesis and the Visualization team for assistance regarding the Unreal 4 game engine. We would also like to thank **Camilla Kusibojovska** for sharing her office-space. Lastly, we would also like to thank our examiner at Chalmers, **Lennart Svensson**, for helpful insights during our meetings.

Simon Kardell and Mattias Kuosku
Gothenburg, October 2017

Contents

1 Introduction

To drive a car on the road is a simple task for a human being. In artificial intelligence, the goal is to solve complex tasks consisting of high dimensional data as input. Unfortunately, it has proven hard to construct sophisticated agents that are capable of driving a car with human-like performance.

1.1 Background

Autonomous driving has been a large research area for the last couple of years, and it will most likely continue to be so for the foreseeable future. The development of Advanced Driving Assistance Systems (ADAS) has been an ongoing process since the digitization of the car. From the year 2000, the development has accelerated and today there exist systems for cruise control, automated parking, blind spot indicator, collision avoidance systems, driver monitoring system, intersection assistance, traffic sign recognition and much more.

The information these applications utilize to conduct its decisions normally consists of data provided by sensors such as radar, lidar, and camera. These types of sensors often generate data in large quantities. A popular approach to solve problems with high dimensional data as input is a field in Machine Learning (ML) called Deep Learning (DL). Deep Learning methods are capable of extracting relevant features from high-dimensional data, by passing the high-dimensional data through a function-approximator that utilizes a multi-layer computational graph called a Neural Network. During the last few years, the adaptation of neural network has increased tremendously, mostly because of recent development in two fields. Firstly there has been a release of large labeled datasets such as the Large Scale Visual Recognition Challenge [1], that can be used for training and verification of classifiers. Secondly, there has been large progress in the development of the computational capacity of graphical processing units (GPU), which enables faster training of Neural Networks.

In recent years deep learning methods have been utilized as function approximators to Reinforcement Learning (RL) models, yielding deep Reinforcement Learning. Unlike general DL-methods where the model is trained on a labeled dataset, reinforcement learning models are trained by interacting with an environment. The goal of the RL-models is to find an optimal behavior, by exploring the environment in an iterative manner. Primarily these deep reinforcement learning models have been

deployed in games, such as old Atari games [2] and the ancient game of Go [3]. There have also been some breakthroughs for continuous control Reinforcement Learning methods [4][5][6]. These algorithms have primarily been evaluated with good results on the MujoCo physics engine [7], which acts as a benchmark for reinforcement learning algorithms.

One of the drawbacks of deep learning is that these networks need large amounts of data. Many methods for data augmentations exist, but even with these methods, it can be hard to generate sufficiently large datasets. Due to this, the use of synthetic data for training machine learning algorithms have been an active research field [8][9][10]. For the case of autonomous driving, acquiring training data is difficult since the dataset will very likely be biased towards containing samples from ideal driving conditions. A remedy to this problem is to train the models on synthetic data since non-ideal driving scenarios can safely be simulated within a synthetic environment. Training on synthetic data is particularly useful for Reinforcement Learning (RL) since these algorithms require the agent to interact with the environment in an iterative learning-by-doing scheme. Since the RL-models learn by exploring the environment and different actions, training using a real vehicle to conduct the exploration is currently highly improbable, since as accidents will likely occur in this setting.

1.2 Related Work

The topic of this thesis is to investigate recent methods in the field of Artificial Intelligence (AI), for autonomous driving. There are numerous implementations in both simulation environments and the physical world. In 2005 the DAVE project developed an end to end solution to control a RC-car to navigate in outdoor environments [11]. They trained a shallow network on data collected by human interaction and achieved results where the agent was capable of driving roughly 20 meters without bumping into obstacles. The DAVE project has expanded in recent years and upgraded their hardware for their robots. The new hardware has been used in research of learning long-range vision for autonomous driving [12]. A group of developers at NVIDIA were inspired by the work in DAVE and developed DAVE-2 [13]. The project used modern hardware and created an autonomous agent that is capable of driving a full-sized car in urban environments. What is most impressive is that the Neural Network in DAVE-2 only takes inputs from camera images captured by the camera of the car and the current steering wheel angle.

Similar to DAVE-2 Brody Huval et al. [14], have applied Neural Networks to images captured in front of a car. The project's focus was to implement a real-time system evaluating Neural Networks performance in highway environment detecting lanes and cars. Similar to DAVE and DAVE-2 they used supervised learning where they trained on data recorded from human drivers. The training was conducted during a 14 day period where they registered data for a few hours per day. It is evident that data acquisition is a tedious process where many hours of human resources are needed. When developing in the framework of Machine Learning, the bene-

fits from simulation environments are tremendous since the data can be generated instantly, instead of recording data in real life. Recently the project $(CAD)^2RL$ showed promising results, where they trained collision avoidance using a discrete RL-algorithm called Q-learning to a quadrotor. Remarkably they only used synthetic data for training and deployed the agent in the real world without any fine tuning on the parameters of the network with successful results [15].

One of the most popular simulation platforms for racing is Torcs [16], which has been used to deploy numerous autonomous agents. Examples of algorithms utilized in the environment are Monte Carlo tree search [17], evolutionary algorithms [18] and Q-learning [19]. The Monte Carlo project used a Tree Search algorithm and forward motion model to explore randomly in the action space to maximize an objective function designed for racing. To use their forward motion model, they transformed the sensor inputs to Euclidean space. Similar to the Monte Carlo project Loiacono (et al.) [19], handcrafted the feature-vectors from the state-space and controlled the car with high-level navigation inputs. Nevertheless, they train a neural network to outperform the currently best AI-methods for overtaking in Torcs. An evolutionary algorithm [18] has expanded the state-space with images and utilized the Fourier transform to refine the images and train a Neural Network to drive autonomously. Apart from Torcs, the CARMA project was inspired by DeepMind's [2] progress with the Deep Q Network (DQN) algorithm in the ATARI environment. They scaled up the project in the Vdrift [20] environment and discretized the action space to fit the DQN algorithm. Using a handcrafted and simplistic reward function together with both sensory inputs and images they could obtain results where they outperformed the handcrafted controller in three criteria, namely, average reward, average speed, and top speed.

1.3 Purpose

The purpose of this project is to investigate Reinforcement Learning (RL) methods for autonomous vehicle control. More specifically, the thesis will investigate two Reinforcement models called Deterministic Policy Gradient (DDPG), and Actor-Critic with Experience Replay (ACER) using only image data and internal states of the vehicle as input. The two RL-models will also be compared with a baseline obtained via a Deep Learning technique called supervised learning or imitation learning. Investigations of the different model's robustness towards non-ideal traffic scenarios will also be conducted. Training, simulation, and evaluation will be performed in a synthetic environment that replicates the real world.

1.4 Scope and Boundaries

The models evaluated in this thesis will use inputs to the network that are similar to those utilized in the DAVE-2 project [13]. The inputs to the models will consist of images captured from a camera directed in the forward facing direction of the ego-vehicle, and internal states of the vehicle, i.e., velocity, acceleration, and jerk. The agents and algorithms will be trained and evaluated in the Unreal 4 game engine [21] in a synthetic world of the AstaZero proving grounds rural road [22]. The simulation environment will also be greatly simplified since it only contains the ego-vehicle that operates in the environment. Since the environment only will contain the ego-vehicle, complex decisions regarding interactions with other road-users will be outside the scope of this thesis.
The control signals will also be limited compared to the ones utilized during regular driving and what is required for a fully autonomous vehicle. The control signals used within this thesis will relate to acceleration and steering, signals such as indicating and gear-selection will be excluded from the action-space.

1.5 Thesis outline

The remainder of this report is divided into six chapters, Theory, Method, Results, Discussion, Conclusion and Future Work. The Theory chapter is subsequently divided into two subcategories, Neural Networks, and Reinforcement Learning. The theoretical chapter presents the reader to the theoretical aspects of the models and algorithms used in the project. Furthermore, the Method starts with a small introduction to the simulation environment used and moves onto introducing the implementation of the Imitation Learning process and the RL-algorithms. In the Results, plots from the training are presented together with histograms of the performance of the final driving policies, as well as some test of the general robustness for the obtained driving-policies. The results are then interpreted in the Discussion, and finally, the thesis is summarized with Conclusion and Future Work.

2
Theory

This master thesis mainly concerns two areas of science, namely Neural Networks and Reinforcement Learning. Neural Networks are the backbone of the model and acts as function approximator to the Reinforcement Learning algorithm. Reinforcement Learning is a family of algorithms that iteratively tries to improve on its current behavior, given some objective that characterizes the performance.

The following sections explain the essential concepts and frameworks. Firstly by introducing the basic concept of designing and training Neural Networks, followed by an introduction to Reinforcement Learning and the algorithms used within this thesis.

2.1 Artificial Neural Networks

Neural Networks for artificial intelligence are mathematical models inspired by natural structures in the human brain and applied in modern computers. Typical applications are to use the models as complex function approximators. In this chapter brief explanations and motivations for these building-blocks in Neural Networks are summarized. The mathematical models, algorithms, and concepts presented in this section are relatively short. For a more thorough explanation the reader is encouraged to take part of the theoretical material in [23].

2.1.1 Feed Forward Neural Networks

A Neural Network is simply a computational graph, with the objective to approximate some function $f^*(x)$. The Neural Network models the function through $f(x, \theta)$ by adjusting its parameters θ. The input x, flows through layers of artificial neurons, where an artificial neuron i, in layer j is defined as

$$a_j^i = \phi\left(\sum_{k=0}^{N} w_{jk}^i x_{jk}^i + b_j^i\right), \quad w_j^i, b_j^i \in \theta. \tag{2.1}$$

In the equation above ϕ is a nonlinear transformation, N is the number of input nodes, b is a bias and $\mathbf{w}$ and $\mathbf{x}$ correspond to the weights and inputs respectively. The forward pass for one node is illustrated in Figure 2.1 and Equation (2.1) can be written in condensed form as

$$[a_j^1, a_j^2, ...a_j^n] = \mathbf{a}_j = \phi\left(\mathbf{w}_j^T \mathbf{x}_j + \mathbf{b}_j\right), \tag{2.2}$$

where n is the number of output nodes. In a feed forward neural network all the hidden activations, a_j, from one layer are passed through the network as input to the next layer.

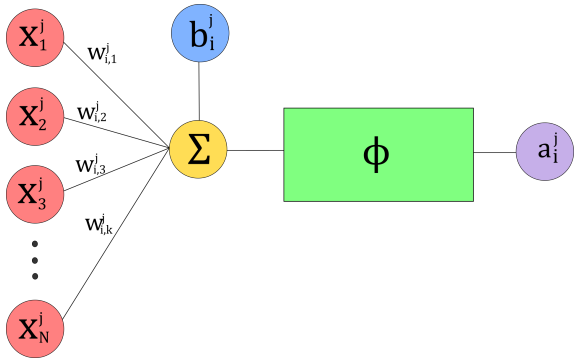


Figure 2.1: A forward pass in layer j to node i where x is the input vector and w is weight vector. The input values are multiplied with the corresponding weights and summarized with the bias term. The summations are then passed through the non-linear function ϕ to produce the output a_i^j.

2.1.2 Activation Functions

For a feed-forward neural network to represent nonlinear mappings, non-linearities called activation functions has to be inserted at different layers in the model. There are many types of activation functions available, and many new ones are still being developed. The functions presented here only cover the ones necessary for the reader to understand the remaining context of the thesis.

Sigmoid Function

The sigmoid function was mostly used in neural networks prior to the introduction of more modern activation functions such as the Rectified Linear Unit (ReLU). The upside of the sigmoid function is that the output is bounded between 0 and 1, making it suitable for modeling probabilities, e.g., logistic regression. The downside of the logistic function is the large saturated regions, making it unsuitable for gradient-based learning [24].
The sigmoid function can be described by the following equation

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{2.3}$$

Tanh Function

The hyperbolic tangent or tanh function was like the sigmoidal function mostly used prior to the introduction of the ReLU, but due to the large saturated regions, it has become less popular as an activation for the hidden layers. The tanh function can be described as

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \tag{2.4}$$

Rectified Linear Unit

The ReLU is currently most commonly used activation function, due to that it does not suffer from those large saturated regions as seen in the sigmoidal- and tanh-function, making it more suitable for gradient-based learning. The ReLU is defined by the following equation

$$\mathrm{ReLU}(x) = \max\{0, x\}. \tag{2.5}$$

As can be seen from the equation it does not suffer from the large saturated regions since it is active in half its domain.

Exponential Linear Unit

The Exponential Linear Unit (ELU) is an extension of the ReLU, but unlike the ReLU it does not saturate for all negative values. Since the ELU has negative values it pushes the mean activation towards zero, effectively reducing the need for normalization of the input [25]. The ELU activation function can be described via the following equation

$$\mathrm{ELU}(x) = \begin{cases} x, & if\ x > 0 \\ \alpha\left(e^x - 1\right), & if\ x \leq 0 \end{cases}, \tag{2.6}$$

where the hyperparameter α governs the negative saturation limit of the ELU activation function. The most common value used for α is 1 and is used in this thesis.

2.1.3 Backpropagation

Backpropagation is an algorithm for calculating the gradient given a loss $J(\theta)$. Using forward propagation an output is generated by the neural network. The output is then used for obtaining the scalar loss $J(\theta)$ which then propagates through the network backward retracting the gradient with respect to the parameters in the model $\nabla J(\theta)$.

The backbone of backpropagation is the chain rule of calculus. The chain rule forms derivative by multiplying known derivatives from known functions. Given function $y = g(x)$ and $z = f(y) = f(g(x))$ the chain rule yields the following

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}. \tag{2.7}$$

Generalizing the case to multidimensional scalars, where $f(\mathbf{x})$, $g(\mathbf{y})$, $\mathbf{x} \in \mathbb{R}^m$, $\mathbf{y} \in \mathbb{R}^n$, g maps from $\mathbb{R}^n$ to $\mathbb{R}^m$ and f maps from $\mathbb{R}^m$ to $\mathbb{R}$

$$\frac{dz}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}. \tag{2.8}$$

In vector notation the equation can be reformulated as

$$\nabla_{\mathbf{x}} z = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)^T \nabla_{\mathbf{y}} z \tag{2.9}$$

where $\left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)^T$ is the Jacobian of $g(\mathbf{x})$. The chain rule is a single step of the backpropagation algorithm. If the Neural Network is constructed as a feed-forward network, the algorithm can calculate the gradients for each layer in the graph based on the gradients from the nodes earlier in the graph. Naturally, the algorithm needs to use the calculated gradients several times since the gradients from one layer are applied to each node in the layer above it. An implementation of the backpropagation algorithm then has to decide whether it needs to recalculate the gradients or store them in memory. For complicated graphs, there can be an exponential number of recalculations of the gradients which would be infeasible. The problem is solved by trading computational time for memory, using dynamic programming.

2.1.4 Training and Optimization Algorithms

To learn or train a model is known as optimization. In its most naive implementation, the parameters are updated in the direction of the gradient of one sample with a small step-size repeatedly until convergence. The method is called gradient descent and is a slow method due to high variance and bias in the gradients. A faster algorithm is the Mini-batch Stochastic Gradient Descent (SGD) that utilizes the average gradient over a batch of randomly selected samples. Even if the SGD is faster than the standard gradient descent, it can be considered slow and inaccurate in some cases. The algorithm tends to be slow, particularly when facing narrow cliffs in the parameter space. The gradients on each side of the cliff are pointing nearly perpendicular to the optimal and causing the state to traverse back and forth of the cliff more than downwards, see Figure 2.2. The algorithm also struggles with state spaces that have local optimums where it gets stuck.

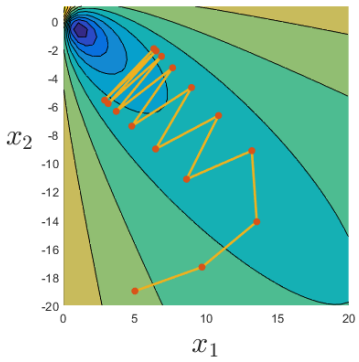


Figure 2.2: Standard gradient ascent searching for optimal solution through a narrow cliff. The figure illustrate the struggle of the algorithm facing narrow cliffs where it traverse back and forth instead of downwards.

Introducing the concept of momentum into the algorithm can increase the convergence speed facing narrow cliffs and helps the algorithm to overcome shallow local optima. The idea of momentum has its heritage in classical mechanics of motion. The introduction of momentum introduces the variable v that governs the movement

of the particle/state in the algorithm. The variable v accumulates exponentially decaying gradient of the past. Thereby the particle/state will move in the direction of aligned gradients in a sequence. The update step of the algorithm is formally written as

$$\begin{aligned} \mathbf{v} &\leftarrow \alpha\mathbf{v} - \epsilon\nabla_\theta \left(\frac{1}{m} \sum_{i=1}^{m} L\left(f(\mathbf{x^{(i)}}|\theta), \mathbf{y^{(i)}} \right) \right), \\ \theta &\leftarrow \theta + \mathbf{v} \end{aligned} \tag{2.10}$$

where the hyperparameter α determines how fast the contribution of past gradients will vanish, L is the loss function and ϵ is the learning rate

However, one of the most used optimization algorithms for neural networks is the Adam optimization algorithm. Adam is short for *Adaptive momentum*, which is one of the key features of the optimization algorithm. To prevent that the algorithm gets stuck in a local optimum the algorithm adapts its learning rate ϵ by introducing first and second order momentum to the gradients. Therefore the momentum enables the algorithm to push through local optima and find better ones. As discussed previously, the moment also allows faster learning because it adapts the learning rate with momentum. Addition to that the Adam algorithm is also applying a bias correction term. The biggest advantage of the Adam optimization algorithm is that it is insensitive to the choice of hyperparameters. The pseudocode for the algorithm is presented in Appendix A.

2.1.5 Convolutional Neural Networks

The most common operation between layers in a Neural Network is the matrix multiplication. Networks that have at least one layer that applies the mathematical operation of *convolution* is called a *convolutional neural network*. Convolution operates on a time series of data and has linear properties which makes it applicable for neural networks analyzing data-structures such as images, sensor readings and much more. There are several implementations of convolution in Neural Network because of digitization and dimension of layers, but they all derive from the mathematical expression of convolution

$$s(t) = \int x(a)w(t-a)da \quad \text{or} \quad s(t) = (x * w)(t) \tag{2.11}$$

where the input data is from $x(t)$ and the kernel $w(t)$ is used to map $x(t)$ to the feature-map $s(t)$. Due to the fact that computers can not handle real values the formulation is generalizing into the 1 dimensional discretized form bounded by the data

$$s(i) = \sum_m x(m)w(i-m). \tag{2.12}$$

Assuming that the kernel and the input data is stored in computer memory, it is applied such that the functions are 0 outside of the valid scope. There are more sophisticated methods for assigning values out of bounds that cope with several

issues regarding convolution. A portion of these methods will be explained further down. Convolution has the property of being commutative, meaning we can stride backward through the data instead of the kernel, i.e.

$$s(i) = \sum_m x(i-m)w(m). \tag{2.13}$$

The formulation above is more straightforward to implement since there are fewer variations of valid values in the data than in the kernel. To implement the modified convolution in Equation (2.13) it is more convenient to implement the cross-correlation which is the same as the convolution but without flipped kernel

$$s(i) = \sum_m x(i+m)w(m). \tag{2.14}$$

The flipped kernel is only useful for the commutative property, and that is not important for neural networks. Finally, the 1-dimensional case can be extended for several dimensions by adding more variables and summations. Given an image, the equation would be formulated as

$$s(i,j) = \sum_m \sum_n x(i+m, j+n)w(m,n). \tag{2.15}$$

A visualization of the convolutional process is illustrated in Figure 2.3. Finally, in

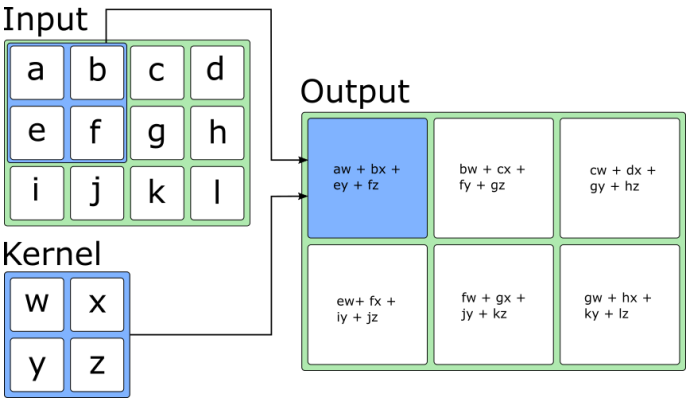


Figure 2.3: Example of convolution with valid padding, input size of 3x4, kernel size 2x2 and a stride of 1.

Neural Networks step size of convolution is called striding, a thorough explanation of the concept is presented for the reader in the succeeding chapter.

Striding

One of the strengths about convolutional layers is that it can reduce the size of parameters. In Section 2.1.5 the formulation of the convolution strides through the image, step by step. However, when designing a convolutional neural network the length of the stride can be chosen freely. In Figure 2.3 the stride is 1 and in Figure 2.4 the stride is 2. Notice that the output images are smaller in the later example because the stride is larger and the steps between each analysis are larger. Applying

strides of larger magnitude enables the model parameters to shrink in size which lowers the computational and statistical burden of the next layer. Mathematically the equation (2.15) can be expanded with the striding parameter

$$s(i,j) = \sum_m \sum_n x(i \cdot s_t + m, j \cdot s_t + n) w(m,n) \tag{2.16}$$

where s_t is the striding parameter. The operation is commonly called down-sampling.

Padding

As mentioned in Section 2.1.5 values outside of the data-scope are filled up with zeros to cope with cases when the kernel is investigating elements outside of the data. How to assign values to these elements is called padding since we are recreating data to extend the size of our output. Examples of padding are zero padding, valid padding, and mirror padding. As mentioned in Chapter 2.1.5 zero-padding extend the input with zeros where the kernel is sliding over the edge of the border. Valid-padding is even simpler and does not pad at all. If a layer is using valid padding the kernel will only stride over valid elements in the data, for illustrations see Figure 2.3 and Figure 2.4. Lastly, mirror padding is when the data in the image-border are mirrored to the padding and creates a pattern that is a copy of the data closest to the border.

Pooling

In one sentence, pooling layer gathers information from small areas in input data and condense that information into a more dense representation. As in convolution the pooling filter is convoluted over the data and summarizes the data in summarized statistics. Common statistics to use is max-pooling, L^2-norm, average and the weighted average based on distance from the center pixel. An example of max-pooling is illustrated in Figure 2.4. By analyzing a small area, pooling makes the neural network robust against small translations of features. Therefore features do not need to be located at their exact position to be detected by the Neural Network.

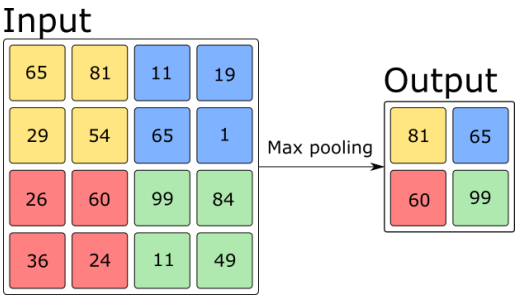


Figure 2.4: Example of max-pooling with valid padding, input size of 4x4, pooling size 2x2 and stride=2.

2.1.6 Recurrent Neural Networks

One shortcoming of regular feed forward Neural Networks is that they can only take inputs of a fixed size. The fixed input size makes problems with non-Markovian

samples hard to model with a regular feed forward neural network since the necessary information is spread out over a sequence of samples.
Recurrent Neural Networks (RNN) are a family of Neural Networks that deals with this issue by inserting a feedback loop in the network, effectively allowing information to persist over a sequence of samples. See Figure 2.5 for an illustration of an RNN unrolled over a sequence of samples.

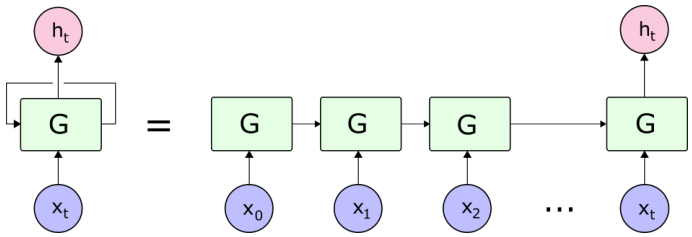


Figure 2.5: Example of unrolled recurrent neural network.[1]

Long Short Term Memory Cell

The Long Short Term Memory cell (LSTM), is a type of RNN that does not suffer from some of the common problems of recurrent neural networks such as vanishing gradients and exploding gradients [26]. The LSTM consists of two feedback connections, the hidden state, h, and the cell state, C, and four layers that are commonly called gates in the context of RNN. The cell state can be viewed as the memory, and the hidden state corresponds to the output of the LSTM at the last time instance.

The first layers that interacts with the input to the LSTM is the forget gate, f_t. The forget gate combines the information from the hidden state and the input, x_t, to decide how which parts of the current cell-state should be kept for the next time instance, i.e.

$$f_t = \sigma\left(\mathbf{w}_f \left[h_{t-1},\ x_t\right]^T + \mathbf{b}_f\right). \tag{2.17}$$

The next interaction with the input x_t is via the *input gate*. This gate consists of two streams, first sigmoidal layers, i_t, that decides how much each element should contribute to the new cell state via the following equation

$$i_t = \sigma\left(\mathbf{w}_i \left[h_{t-1},\ x_t\right]^T + \mathbf{b}_i\right). \tag{2.18}$$

The next step of the *input gate* is to propose new candidate values for the cell state, $\tilde{C}_t$ via a tanh layer, i.e.

$$\tilde{C}_t = \tanh\left(\mathbf{w}_c \left[h_{t-1},\ x_t\right]^T + \mathbf{b}_c\right). \tag{2.19}$$

The cell state is then updated using the forget gate and the input gate, where $\tilde{C}_t$ is scaled by i_t element-wise. The cell state update equation can be described as

[1]The unrolled figure is strongly influenced by Christopher Olah illustrations, http://colah.github.io/posts/2015-08-Understanding-LSTMs/

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \tag{2.20}$$

where, $\odot$ denotes the Hadamard product or element-wise multiplication.
The last part of the LSTM-cell is the *output gate*, which determines which parts of our current cell state we want to output. The output of the LSTM-cell will be a filtered version of our current cell state, filtered with a sigmoid layer o_t, i.e.

$$o_t = \sigma\left(\mathbf{w}_o \left[h_{t-1},\ x_t\right]^T + \mathbf{b}_o\right). \tag{2.21}$$

For the final output the cell state, C_t, is mapped through a tanh function and multiplied with the o_t element-wise, i.e.

$$h_t = o_t \odot \tanh\left(C_t\right).$$

An illustration of a LSTM-cell can be viewed in Figure 2.6

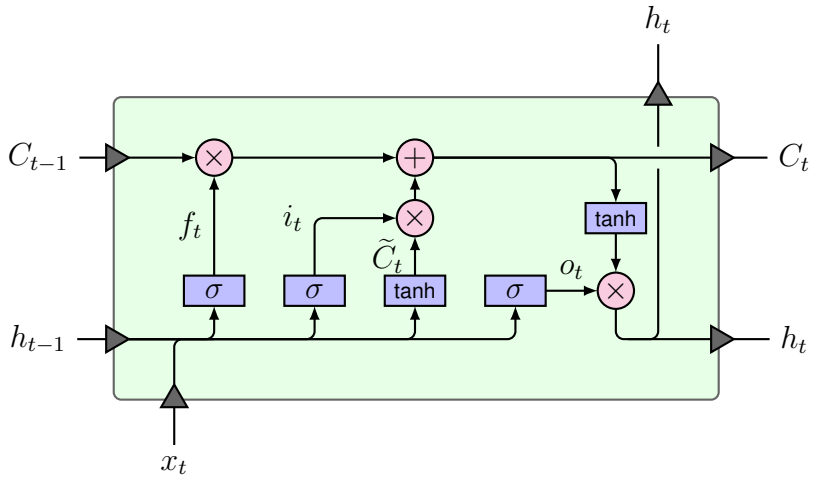


Figure 2.6: An illustration of a LSTM-cell. All mathematical operations in the figure are performed element-wise.[2]

2.2 Reinforcement Learning

Premises for RL are that there exists an environment and a controllable Actor. The Actor is capable of changing the state of the environment with actions, e.g., move left or right. In a broad sense, the environment is rewarding the Actor based on its current behavior and in reinforcement learning the objective is to find the behavior/policy that maximizes the cumulative reward. This section will introcude the reader to the basic theory in RL [27] and recent algorithms that are relevant for the thesis [5][28][6].

[2]The illustration of the LSTM cell is influenced by the bloggpost http://colah.github.io/posts/2015-08-Understanding-LSTMs/ written by Christopher Olah.

2.2.1 Markov Decision Process

The environment for which an RL agent operates can formally be described as a Markov Decision Process (MDP), where the environment is fully observable. More formally this is known as the Markov property which states that the future is independent of the past given the present, i.e.

$$P(s_{t+1}|s_t) = P(s_{t+1}|s_1, s_2, ..., s_t). \tag{2.22}$$

An MDP is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$, of a set of states, actions, transitional probabilities, rewards and a discount factor to keep the expectations finite in the case of an MDP without terminal states.

- $\mathcal{S}$ is a finite set of states
- $\mathcal{A}$ is a finite set of actions
- $\mathcal{P}$ is the state transition probability matrix, $\mathcal{P}^a_{ss'} = P(S_{t+1} = s'|S_t = s, A_t = a)$
- γ is a discount factor $\in [0, 1]$

Figure 2.7 illustrates a simple MDP with four different states.

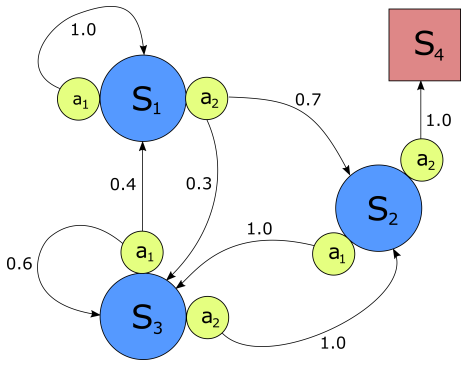


Figure 2.7: Illustration of a MDP. The terminal state is s_4 and the reward is -1 for all actions and the probability for a transition is labeled near the transition arrows.

2.2.2 Partially Observable Markov Decision Process

In a Partially Observable MDP (POMDP), the agents state does no longer equal the state of the environment, i.e., the Markov property does not hold for the state representation. Consider an agent that navigates through a maze, where the state is defined as a single first-person view frame. Clearly, the information provided from that single frame is not sufficient to make an optimal decision, hence the violation of the Markov property. In a POMDP the agent must construct its own state representation to make optimal decisions, e.g., append the entire history to the state.

2.2.3 Policy

The policy, π is what characterizes the agents behavior in the MDP, more formally the policy is a distribution of actions given states,

$$\pi(a|s) = P(A_t = a|S_t = s). \tag{2.23}$$

2.2.4 Discounted reward and objective function

For each action an Actor makes in the environment, it will earn a reward r_t. The cumulative reward can be written as a sum. Adding the discount factor makes the future reward less important and makes the sum finite in, e.g., the continuous problem without terminal states. The discounted reward can be defined as

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}. \tag{2.24}$$

The goal of RL is to maximize the discounted expected return. Following a policy π the objective function can then be formalized as

$$J = \mathbb{E}_{r_i, s_i \sim E, a_i \sim \pi} [G_1] \tag{2.25}$$

where state s_i is sampled from the environment E and actions are sampled from the policy π. In RL the objective function can also be formulated as the value or action-value function. These functions are described in the two following sections.

2.2.5 State Value Function

The state value function is defined as the expected sum of future rewards following policy, π, i.e.

$$V^{\pi}(s) = \mathbb{E}_{\pi} [G_t | S_t = s] . \tag{2.26}$$

The value function can also be expressed recursively with the Bellman expectation equation as

$$V^{\pi}(s) = \mathbb{E}_{\pi} [r_t + \gamma V^{\pi}(s_{t+1}) | S_t = s] . \tag{2.27}$$

2.2.6 Action Value Function

The action value function can be decomposed similarly as for the state value function, starting with the return

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi} [G_t | S_t = s, A_t = a] . \tag{2.28}$$

to the recursive form obtained with the Bellman expectation equation

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi} [r_t + \gamma Q^{\pi}(S_{t+1}, A_{t+1}) | S_t = s, A_t = a] . \tag{2.29}$$

2.2.7 Policy Gradient

Policy gradient methods perform gradient ascent on the policy objective function J, with respect to the parameters θ of policy π. The policy gradient can be defined as

$$\nabla J(\theta) = \mathbb{E}_{\pi_\theta} \left[\nabla_\theta \log \pi_\theta(s, a) Q^{\pi\theta}(s, a) \right] . \tag{2.30}$$

Monte-Carlo policy gradient method uses G_t as an unbiased sample of $Q^{\pi\theta}(s,a)$. Even though this gradient is unbiased, it still has large variance, which yields noisy gradient estimates.
A common way to reduce the variance is to use estimates of $Q^{\pi\theta}(s,a)$ and $V^{\pi\theta}(s)$ to calculate the Advantage function, i.e.

$$A^{\pi\theta}(s,a) = Q^{\pi\theta}(s,a) - V^{\pi\theta}(s). \tag{2.31}$$

A lower variance representation of the policy gradient can then be stated as

$$\nabla J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a) A^{\pi\theta}(s,a)\right]. \tag{2.32}$$

Though this representation has lower variance, it does come with the cost of an additional bias due to imperfections in the estimation of $Q^{\pi\theta}(s,a)$ and $V^{\pi\theta}(s)$.
Intuitively the choice of $A^{\pi\theta}(s,a)$ can be justified due to that it increases the probability of actions that are better than average and decrease the probability of actions that are worse than average [29].

2.2.8 On-policy vs off-policy

RL algorithms that train its agents purely on experience retrieved from its current policy is called on-policy algorithms. On-policy algorithms are sample inefficient in the sense that if you train your agent, you will also change the agent's policy and behavior. This means that past experience is no longer valid to be utilized for training. Off-policy algorithms are therefore preferable since they are capable of using the experience that is obtained with other policies.

2.2.9 Replay buffer

Simulation time is computationally heavy, a common concept to lower the simulation-time in RL is to learn from experience what the agent has been exposed to previously. To remember how the environment reacts to the agent's actions the tuple $\langle s_t, a_t, r_t, s_{t+1}, t_t \rangle$ is stored in a data structure that can be accessed for a low computational cost. For some algorithms, it is important to have chained data that is decoupled as time sequences. The replay buffer is therefore extended to store sequences of tuples $\langle s_{t+n}, a_{t+n}, r_{t+n}, t_t \rangle$ where n is the position of the tuple in the time sequence.

2.2.10 Exploration vs. Exploitation

One fundamental decision in RL is the exploration vs. exploitation dilemma. While exploitation means that the agent should conduct the best decision given the current information available, it does not mean that the particular action is anywhere near optimal. For the agent to make optimal or near optimal decisions, it has to explore the action space to gather more information, to optimize the current policy. A policy that has a high exploitation rate will have problems to converge towards an optimal or near-optimal behavior. For a policy that has an exploration strategy that explores too much, it will be hard to see any improvements, due to that the agent's behavior

in large part appear random. A sensible guideline is to have an exploration strategy that has a large initial exploration rate that decays over time to see improvements in the policy and to visit parts of the state space that can be hard to detect with to much exploration noise. A common strategy to use for continuous actions spaces is to add zero-mean Gaussian noise for exploration.

2.2.11 Actor-Critic

The Actor-Critic architecture uses two structures to optimize the expected return. The Actor and Critic operate together and is trained separately for their purpose in the algorithms. The Actor defines the current policy and therefore is intended to generate actions according to the current policy. The Critic's task is to estimate the value function to the problem. Learning is commonly on-policy, and the Critic must learn what the expected action value that is conducted from the current policy defined by the Actor. The Critic can then criticize action taken by the policy as a Temporal Difference-Error (TD-Error). TD-Error is the temporal difference between the value function estimates of the two different states. The evaluation is described mathematically as

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t). \tag{2.33}$$

The TD-error is then used for optimizing the parameters in the Actor and Critic model. If $\delta_t > 0$ the outcome of the current action, a_t, is better than expected and it would therefore be desirable to increase the probability of $\pi(a_t|s_t)$. The general setup of the algorithm is illustrated in Figure 2.8.

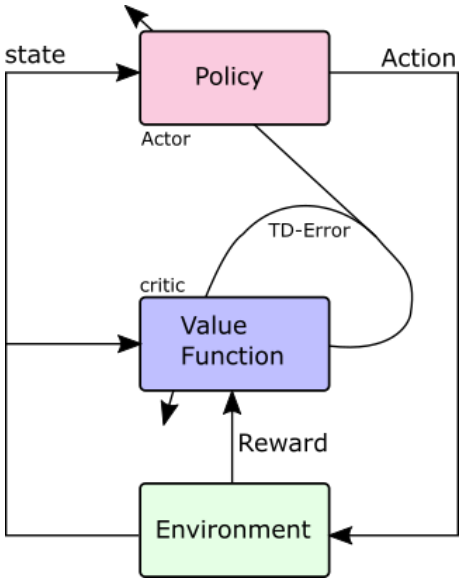


Figure 2.8: Basic visualization of the Actor-Critic algorithm and setup.

2.2.12 Deterministic Policy Gradient

There is a lot of variants of Deterministic Policy Gradient(DPG). In this thesis, the off-policy deterministic Actor-Critic will be considered, where ρ is the discounted

state distribution and β represents a distinct policy to the current policy π. For more details see the paper [30]. Furthermore, the performance objective can be stated from the value function or the action value function

$$\begin{aligned} J_\beta(\mu_\theta) &= \int_{\mathcal{S}} \rho^\beta(s) V^\mu(s) ds \\ &= \int_{\mathcal{S}} \rho^\beta(s) Q^\mu_\theta(s)) ds. \end{aligned} \tag{2.34}$$

The gradients for the parameters of the Actors model can then be approximated with

$$\begin{aligned} \nabla_\theta J_\beta(\mu) &\approx \int_{\mathcal{S}} \rho^\beta(s) \nabla_\theta \mu_\theta(a|s) Q^\mu(s,a) ds \\ &= \mathbb{E}_{s\sim\rho^\beta}\left[\nabla_\theta \mu_\theta(s) \nabla_a Q^\mu(s,a)|_{a=\mu_\theta(s)}\right]. \end{aligned} \tag{2.35}$$

The true action value function is replaced with a general function approximator $Q^w \approx Q^\mu$ and is trained to minimize the true action-value function. Furthermore, the basic steps of the algorithm are

$$\begin{aligned} \text{TD-Error: } \delta_t &= r_t + \gamma Q^w(s_{t+1}, \mu_\theta(s_{t+1})) - Q^w(s_t, a^t) \\ \text{Update Critic weight: } w_{t+1} &= w_t + \alpha_w \delta_t \nabla_w Q^w(s_t, a_t) \\ \text{Update Actor weight: } \theta_{t+1} &= \theta_t + \alpha_\theta \nabla_\theta \mu_\theta(s_t) \nabla_a Q^w(s_t, a^t)|_{a=\mu_\theta(s)}. \end{aligned} \tag{2.36}$$

2.2.13 Deep Deterministic Policy Gradient

Deep Deterministic Policy Gradient (DDPG)[5] apply the DPG-algorithm using Neural Networks as general function approximators. Applying Neural Networks in RL with continuous actions spaces lead to three major issues: correlated data, instability, and insufficient exploration. This chapter will describe three methods that solve these issues.

Training Neural Networks require that the training data is independently and identically distributed [23], which is not the case when generating samples sequentially in the environment. The DDPG algorithm uses the replay buffer to store previous experience. The replay buffer can be utilized when a sufficient amount of data is collected. The goal is to withstand the curse of correlated data. The loss function for the Actor-Critic can then be formulated as the squared loss from samples

$$L(\theta^Q) = \mathbb{E}_{s_t\sim\rho^\beta, a_t\sim\beta, r_t\sim E}\left[(Q(s_t, a_t|\theta^Q) - y_t)^2\right] \tag{2.37}$$

where

$$y_t = r(s_t, a_t) + \gamma Q(s_{t+1}, \mu(s_{t+1})|\theta^Q). \tag{2.38}$$

The Neural Network for the Critic is prone to diverge since y_t is calculated with the same network as we are optimizing. The solution for this problem is to make copies of the networks and then update them with soft updates. It has been experienced that it is most efficient to make copies of both the Actor and Critic to enable stability. The copies are denoted as

$$\begin{aligned} &Q'(s, a|\theta^{Q'}) \\ &\mu'(s|\theta^{\mu'}) \end{aligned} \tag{2.39}$$

and the soft updates can mathematically be formulated as

$$\begin{aligned} \theta^{Q'} &\leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'} \\ \theta^{\mu'} &\leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'} \end{aligned} \tag{2.40}$$

where $\tau \ll 1$. Exploration of continuous action spaces is hard due to the infinite amount of permutations that exist. In off-policy algorithms, the exploration can be constructed independently from the learning algorithm. The simplest way of constructing an exploratory Actor is to add an exploration noise to the Actors action.

$$u_{exp}(s_t) = \mu(s_t|\theta_t^{\mu}) + \mathcal{N} \tag{2.41}$$

where $\mathcal{N}$ can be chosen to fit with the environment. Pseudo code for the algorithm can be found in Appendix B.

2.2.14 Actor-Critic with Experience Replay

The Actor-Critic with experience replay (ACER) [6], builds on the ideas presented in Asynchronous Advantage Actor-Critic (A3C) [31].

Briefly summarized the A3C utilizes shared parameters between the Actor and the Critic in the lower layers of the Neural Network and uses on-policy updates with an advantage estimate to update the network. A3C is however like all other on-policy algorithms sample inefficient by design. To make the collection of samples more efficient, A3C utilizes multithreading and assigns a number of different workers to explore the environment, effectively leading to less correlated data in the updates. Another benefit of having a set of workers is that many different exploration strategies can be deployed simultaneously.

Unlike A3C, ACER utilizes off-policy samples via an experience replay buffer, making it much more sample efficient than its on-policy counterpart.

One of the main challenges when working with off-policy data is to get an accurate estimate of the action value function for the current policy, π. To compensate for trajectories generated from behavior policies, ACER uses the Retrace algorithm to estimate $Q^{\pi}(x_t, a_t)$ [32]. Given a trajectory generated under a behavior policy μ, Retrace can be expressed recursively via

$$Q^{ret}(x_t, a_t) = r_t + \gamma \min\left\{c, \frac{\pi(a_t|x_t)}{\mu(a_t|x_t)}\right\}\left[Q^{ret}(x_{t+1}, a_{t+1}) - Q(x_{t+1}, a_{t+1})\right] + \gamma V(x_{t+1}) \tag{2.42}$$

where $Q(x_t, a_t)$ and $V(x_t)$ are the current estimates of $Q^{\pi}(x_t, a_t)$ and $V^{\pi}(x_t)$ respectively. As can be seen from the Equation (2.42), the Retrace estimator utilizes the return for near on-policy behavior and otherwise bootstrap to the current state value estimate. The parameter c, ensures that the variance of the estimator does not become too large by preventing importance sampling ratios larger than c.

To separate the weights for the policy and the value estimates, the network is divided into two computational streams with the earlier layers shared between the two streams. To estimate $Q^{\pi}(x_t, a_t)$, ACER utilizes the dueling network architecture [33], but adapts it to a stochastic version for continuous action spaces, i.e.

$$\tilde{Q}(x_t, a_t) = V(x_t) + A(x_t, a_t) - \frac{1}{N}\sum_{i=1}^{N} A(x_t, a_i'), \quad a_i' \sim \pi(\cdot|x_t) \tag{2.43}$$

where the subtraction of the sample average is to compensate for any potential bias in the advantage estimate. The estimate of $Q^{\pi}(x_t, a_t)$ are updated using the mean squared error loss, i.e.

$$L_Q(\mathbf{x}, \mathbf{a}) = \frac{1}{N}\sum_{i=i}^{N}(Q^{ret}(x_i, a_i) - \tilde{Q}(x_i, a_i))^2 \tag{2.44}$$

and to update the estimate of $V^{\pi}(x_t, a_t)$, the gradients of the following loss are used with respect to the current estimate of V^{π}

$$L_V(\mathbf{x}, \mathbf{a}) = \frac{1}{N}\sum_{i=i}^{N} \min\left\{1, \frac{\pi(a_t|x_t)}{\mu(a_t|x_t)}\right\}(Q^{ret}(x_i, a_i) - \tilde{Q}(x_i, a_i))^2. \tag{2.45}$$

To calculate the policy gradient from off-policy trajectories, ACER utilizes truncated importance ratios with a correction term to compensate for the bias introduced by the truncation. The policy gradient for ACER can be expressed as

$$\begin{aligned}\hat{g}_t^{acer} =& \min\left\{c, \frac{\pi(a_t|x_t)}{\mu(a_t|x_t)}\right\}\nabla_{\phi(x_t)}\log\pi\left(a_t|(x_t)\right)\left(Q^{ret}\left(x_t, a_t\right) - V\left(x_t\right)\right) \\ &+ \frac{1}{N}\sum_{i=1}^{N}\left[\frac{\rho_t(a_i') - c}{\rho_t(a_i')}\right]_+ \left(\tilde{Q}(x_t, a_i') - V(x_t)\right)\nabla_{\phi(x_t)}\log\pi(a_i'|x_t).\end{aligned} \tag{2.46}$$

where $\rho_t(a') = \frac{\pi(a'|x_t)}{\mu(a'|x_t)}$ and $[x]_+ = \max\{0, x\}$.

Since the variance of policy updates often is larger than desirable, the stability of policy gradient can be compromised, leading to so-called policy collapses. To get a more stable behavior the per-step changes of the current policy has to be limited, to prevent too large steps on the policy causing a collapse of the current policies performance. Trust Region Policy Optimization (TRPO) provides a solution to this problem [4], however, despite the effectiveness of TRPO it requires multiple computations of Fisher-vector products for each update, which does not scale well to larger models [6]. To prevent too large steps on the current policy ACER uses a different version of TRPO that scales better to larger domains. The TRPO scheme used in ACER keeps a running average over past policies by softly updating the parameters θ_a for the average model towards the policy parameters, i.e.

$$\theta_a \leftarrow \tau\theta + (1-\tau)\theta_a \tag{2.47}$$

The ACER TRPO-scheme restricts too large steps from the average policy by introducing a linearized Kulback Liebler divergence constraint, i.e.

$$\begin{aligned} &\underset{z}{\text{minimize}} \ \frac{1}{2}||g_t^{acer} - z||_2^2 \\ &\text{subject to } k < \delta \end{aligned}$$

where δ is a hyperparameter that affects the allowed deviation from the average policy, i.e., the trust region, k corresponds to

$$k = \nabla_{\phi_\mu(x_t)} \left(\int_{-\infty}^{\infty} \pi_{\theta_a}(a|x_t) \log \frac{\pi_{\theta_a}(a|x_t)}{\pi_\theta(a|x_t)} da \right). \tag{2.48}$$

By inserting the constraint, it becomes a quadratic programming problem for which the solution is:

$$z^* = g_t^{acer} - \max\left\{0, \frac{k^T g_t^{acer} - \delta}{||k||_2^2}\right\} k.$$

Since the resulting gradient is only w.r.t. the output of the policy stream, i.e., the mean μ of the policy distribution, the gradients w.r.t. θ remains to be calculated. The remaining gradients can be calculated by using the properties of the chain rule as

$$\frac{\partial \mu_\theta(x_t)}{\partial \theta} z^*.$$

Since ACER works off-policy, it also uses an experience replay buffer, see Section 2.2.9. Stored in the replay buffer is a sequence of tuples, with the first state, action, next state, reward, terminal flag and the moments of the distribution, i.e. $\langle s_t, a_t, r_t, s_{t+1}, t_t, \mu_t, \sigma_t \rangle$.

3
Methods

In this chapter, the simulation environment, and the simulation setup is described, followed by how the simulation environment was used to train an imitation learning agent using supervised learning. After that, a detailed explanation of the reward function employed in the environment. Lastly, the exploration strategy and the implementation details for the different models are presented.

3.1 Training Environment

The Training Environment is developed in the game engine Unreal 4. The goal is to control a basic implementation of the standard car model provided from the starter content in Unreal. However, the standard car did not provide measurements for acceleration, jerk, heading relative to road curvature, lane position or frontal images. Those features were therefore implemented into the car model. The states that will be used as input to the RL models are summarized in Table 3.1.

Table 3.1: Parameter fetched from the simulation environment.

	short explanation	used as input
p_x	global x position	
p_y	global y position	
p_z	global z position	
v_x	velocity x direction	x
v_y	velocity y direction	x
v_z	velocity z direction	x
a_x	acceleration x direction	x
a_y	acceleration y direction	x
a_z	acceleration z direction	x
j_x	jerk x direction	x
j_y	jerk y direction	x
j_z	jerk z direction	x
α	lane position	
φ	relative heading angle	

The measurement of the lane position is the distance to the center of the lane from the center of the car and images are captured approximately in the windscreen of the car and has a resolution of 80x240. Example images captured by the windscreen camera are presented in Figure 3.1.

Figure 3.1: Examples of images captured by the camera located in the frontal windscreen.

The lane position α is measured by calculating the horizontal distance to the road surface edge and then approximating the center line as half of that distance. Because the measurement is towards the edge of the road instead of the edge of the lane, the lane center is approximated as 60% of the distance from the center lane to the edge of the road. α is then calculated as the distance from the center of the car to the center of the approximated lane-center. The normalized distance α is depicted in Figure 3.2. Unfortunately, since the distance between the road edge and lane marking is not constant the α value will deviate from the desired value since the estimated lane center is imperfect.

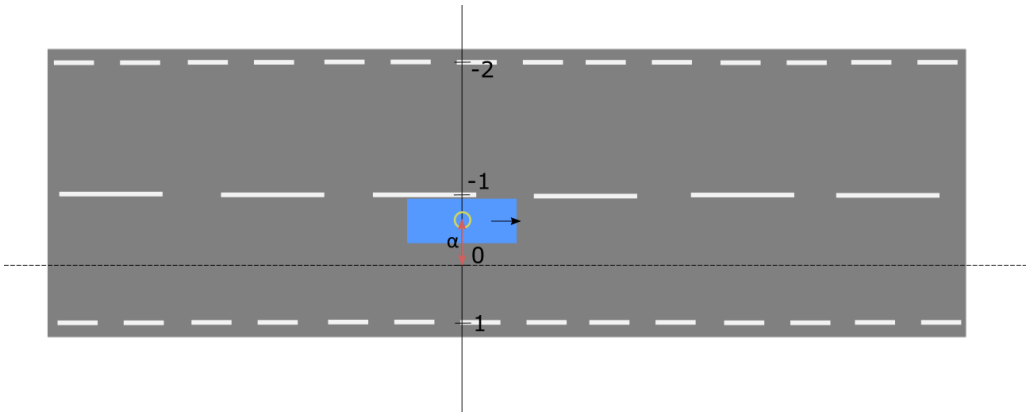


Figure 3.2: Image demonstrating the α value in the reward function. If the car is centered in the correct lane the value is 0. The value α is normalized to indicate the outer boundaries of the correct lane with 1 and -1. If the car is out of the road to the left side the value will be less then -2.

Furthermore, the surrounding visual is a model over the AztaZero proving ground for autonomous vehicle testing. In this thesis, only the rural track will be considered. The track is considered smooth with no sharp corners or steep hills but inherits access-points from smaller roads in the form of T-junctions and slip-roads. The shape of the rural road is illustrated in the Figure 3.3 without the T-crosses and slip-roads marked.

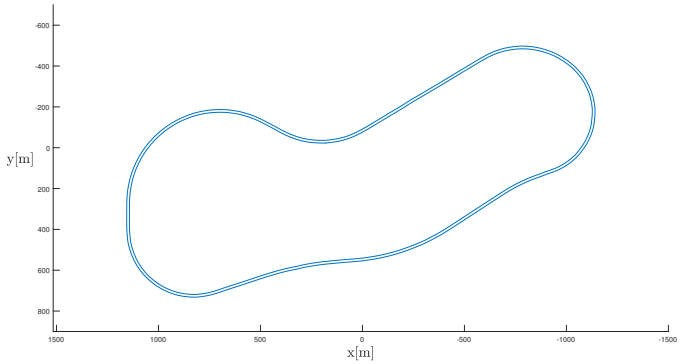


Figure 3.3: Illustration of the rural road in simulation environment of AztaZero proving ground without T-junctions and slip-roads marked.

3.1.1 Simulation Setup

The interface towards the training environment was designed to handle two cases of control, one case for control of the steering angles, the brake and acceleration and one for just the steering angles. The reference velocity was kept at a fixed value of 50 km/h during all simulations. In the case where the level of control includes both acceleration and braking, the throttle and braking were modeled with a single tanh function, where values larger than zero are viewed as a signal for the throttle, and conversely, values smaller than zero are viewed as a control signal for the braking. For the case where the agents level of control only includes the steering angle, the throttle and brake are modeled with a simple PID-controller.

The control flow of the simulation environment is constructed such that the environment executes the action decided by the agent for a fixed number of frames and then pause the simulation while it waits for a new action from the agent. After the agent has performed one step in the environment, the environment returns a new observation, a reward and a boolean variable that states if the current state of the environment corresponds to a terminal state or not. The environment only sets the boolean corresponding to the terminal state to true if the agent is located outside of the road or if it is traveling in the opposite direction. The nominal number of frames per second in the simulation environment is 60 frames, however, this might vary over time if the rendering of the current scene requires a significant computational load. For all the simulations, the number of frames that the environment executes the action was set to 4, yielding a frequency of roughly 15 Hz for decisions.

During the training, the initial position for the agents is sampled uniformly from a set of 66 locations spread out equidistant and facing in both directions of the road. All training was performed at the end of each episode to make the different policies for the episodes more comparable. Since all the training was conducted between different episodes, the episodic length had to be capped to allow training to be performed on policies that otherwise would not reach a terminal state. Hence a maximum episodic length was introduced, and the maximum episodic length used for all simulation was set to 1000 steps.

For the training, each episodic reward was logged and compared to the 10 best episodic rewards. If the current episodic reward is within the best 10 results, an inference run was performed where the weights of the network with the current best inference reward are stored. For the inference runs, the episodic length was extended to 10 000 steps and start location was kept fixed to yield more comparable results. For clarity, the training inference scheme is presented in Algorithm 1.

Algorithm 1 Inference training scheme

```
Initialize weights
i = 0
while i < max episodes do
    Run 1 episode
    if episode reward > min(top ten episode rewards) then
        Run 1 inference episode
        if inference reward > best inference reward then
            Store weights for the model
    Train model
    i++
```

3.2 Imitation Learning

In this section, the motivation, method and model design of the agent obtained via imitation learning will be described in detail. Firstly a short motivation of the agent obtained from imitation learning is summarized, followed by the collection of training data. Lastly, a description of the different parts of the imitation learning network is presented.

3.2.1 Motivation

The purpose of training an agent via imitation learning for this thesis is twofold. Firstly the agent can be used to collect data of good driving behavior and store it in the experience replay buffer. The benefit of doing this is that there will always be access to reliable data when optimizing the policy, which at least intuitively should speed up the learning process. The second benefit is that many of the evaluated algorithms include recurrent LSTM units, which will lead to large demands on memory if the convolutional layers have to be unfolded in time during optimization. Therefore the convolutional layers obtained from the imitation learning process will be stored and used to prepossess the images for the other algorithms.

3.2.2 Data Acquisition

The data collection for the supervised training was collected by sending continuous control signals to the simulation environment with a Logitech gaming pad operated by a human. Logging of the data aimed to be versatile and uniform for the situations where the agent is plausible to be in. To be able to store data where the agent is recovering from bad states a pause button was implemented into the system. The pause button decides whether the system should store data or not and therefore could operate the car to be in dangerous states, start logging and then operate the car to a satisfied state. A session of 8 hours was performed and collected around 100.000 samples.

3.2.3 Models

In this section, the model used for the imitation learning will be described. The architecture will be presented in two sections, one for the convolutional layers and one for the fully connected layers. The separation of these sections is because the convolutional layers will be reused in preceding models.

The Convolutional Layers

In order to decode images captured by the camera mounted on the windscreen, a set of five convolutional layers was chosen. The input to the first convolutional layer is an image captured of the current view for the camera. The image is cropped to 80 pixels in height and 240 in width with all three color channels, see Figure 3.1 for a set of input images. To reduce the spatial dimensions of the data, the first four convolutional layers are followed by a max pooling operation, with a stride of two in both directions and a pooling region of 2×2. The last layer is followed by a max pooling operation with a pooling region of 2×1 and a stride of two in the height dimension and one in the width dimension. This was done deliberately since intuitively it seems more probable that this part of the feature maps contains more information regarding the lane position. For more details regarding the design of the convolutional layers, see Figure 3.4.

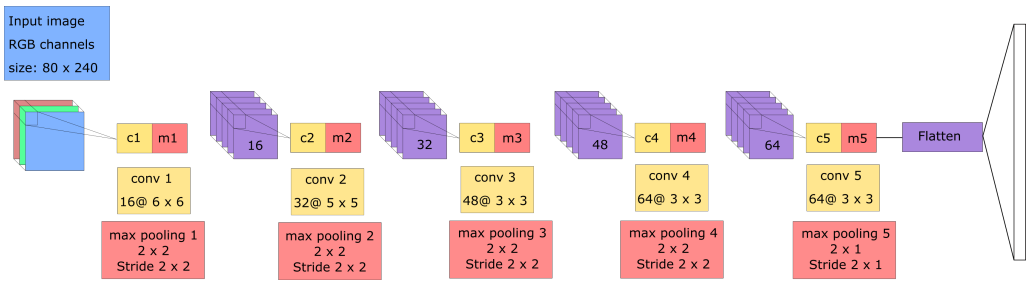


Figure 3.4: Schematic overview of the convolutional network. All activation functions are ELU in the convolutional layers. The input image is a normalized and represented as 3 layers each representing the corresponding RGB value for that pixel. The output is a 2880 units feature vector.

The Fully Connected Layers

The depth of the fully connected layers for the imitation learning agent was kept relatively shallow. This was done deliberately to minimize the risk of vanishing gradients and therefore increase the quality of the training of the convolutional layers. The hidden activation's used ELUs as activation function, and the activation of the last layer was just a simple identity function. The architecture of the fully connected layers is visualized in Figure 3.5.

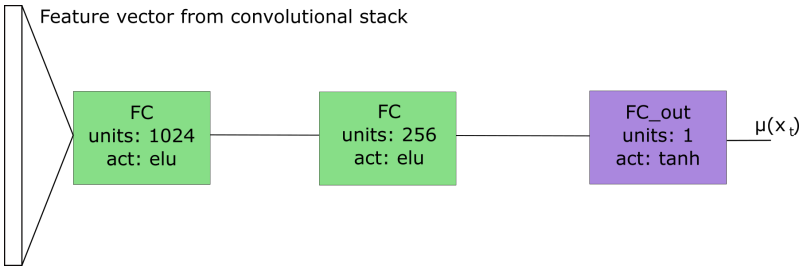


Figure 3.5: A schematic overview for the fully connected network for the supervised agent. The input is the feature vector from the convolutional stack and the output is the steering angle.

3.2.4 Training

Before training the agent obtained via Imitation Learning, the dataset was split into two different subsets, one for training and one for validation. The size of the validation-set was chosen to be 4000 samples, and the remaining 996000 was used for training. There was no regularization used during the training, except early-stopping. Briefly, early stopping can be described as a technique to avoid overfitting on the training data by storing the network that has the best result on the validation-set, and stop the training when there is no improvement on the validation-set over a fixed amount of steps.
The loss-function used during training was the mean squared error (MSE), i.e.

$$L_{\text{MSE}} = \frac{1}{N}\sum_{i=1}^{N}(y_i - f(x_i, \theta))^2 \tag{3.1}$$

where $f(x_i, \theta)$ is the output of the Neural Network given the image x_i parametarized by θ and y_i is the label corresponding to image x_i.
The optimization was conducted with the Adam-optimizer, with the learning rate of 10^{-4} and a batch size of 256.

3.3 Reward Function

The reward function implicitly determines the optimization objective, and it is a vital part since it is via the reward function the optimal behavior is described. Designing a reward function that describes optimal driving behavior is notoriously

hard [34], since driving a car consists of many different types of scenarios that can be hard to derive to an explicit equation. However since the scope of this thesis is limited to have the ego-vehicle as the only road user and the environment consist of a road that goes in a loop, many of the challenges of designing a reward function for driving can be ignored.

To guide the agent towards the optimal behavior, the designed reward function was chosen to be dense. The designed reward function consists of two parts, one that assigns a reward based on the progress the agent is currently making and one that is more focused on driver comfort metrics and therefore assigns a cost for large accelerations and jerks. The reward for the agent's progress was designed to be

$$R_{prog} = \begin{cases} \cos(\varphi)\frac{v}{v_{ref}} - \left|\alpha\frac{v}{v_{ref}}\right|, & \text{if } v \leq v_{ref} \\ \cos(\varphi)\left(\frac{2v}{v_{ref}} - \left(\frac{v}{v_{ref}}\right)^2\right) - \left|\alpha\left(\frac{2v}{v_{ref}} - \left(\frac{v}{v_{ref}}\right)^2\right)\right|, & \text{if } v > v_{ref} \end{cases} \tag{3.2}$$

where v_{ref} is the reference speed, ϕ is the heading angle relative to the road, and α is a normalized distance to the center of the right lane, see Figure 3.2. As can be seen in the equation above, the reward promotes driving with small heading angles as well as proper lane position. The reward is designed such that given both a negative and positive offset with equal amplitude and the same steering policy, the negative offset will yield a larger reward.

Studies show that humans are tolerant to acceleration and jerks up to a certain level [35], at which the acceleration and jerks become much more intolerable. For this reason, a cost function suggested in [36] that considers this, was used together with the corresponding hyperparameters. The cost function used can be defined as

$$C(x) = \begin{cases} \left(\frac{x}{g}\right)^2, & \text{if } x \leq g \\ \left(\frac{5}{6} + \frac{1}{6}\left(\frac{x}{g}\right)^2\right)^6, & \text{if } x > g. \end{cases} \tag{3.3}$$

The cost function was used both for the acceleration and the jerk. The threshold parameter g, for which after the cost rapidly increases was set to 2 for the acceleration and 1.5 for the jerk, based on the information provided in [35].
Apart from that, the cost function is based purely on acceleration and jerks, a cost was also assigned to the angular velocity of the steering angle, i.e. $\dot{\varphi}$. Combining the reward for the progress with the different cost functions yields the following reward function

$$R(a_t, x_t) = R_{prog} - (\beta_{\dot{\varphi}}\dot{\varphi} + \beta_a C(a) + \beta_j C(j)) \tag{3.4}$$

where β_i, $i \in [\dot{\varphi}, a, j]$ is a hyperparameter that decides much a influence the different costs will have on the total reward. The hyperparameters used for the different costs chosen to $\beta_a = 0.67$, $\beta_j = 1.33$ and $\beta_{\dot{\varphi}} = 0.25$.

3.4 Sequential Updates

Because the ACER-model utilizes sequential updates, a sequence of n observations $x_0, x_1, ..., x_n$ were sampled from experience replay for each update. Even though the LSTM-cell exploits the sequential structure of the data, naively calculating the gradients for the entire sequence, might cause unwanted imperfections in the gradient updates, due to that the first gradient estimates will be based on a non-existing history. To mitigate this problem a number of skip steps, h, were used that allows the internal states of the LSTM to build up useful information of the history. As a result, the gradients for the sequence $x_0, x_1, ..., x_h$, are not backpropagated through the network to avoid any imperfections that a non-existing history of the LSTM-cell might inflict.

3.5 Exploration Strategy

To explore the environment, all the evaluated models used a Gaussian distribution as exploration noise and for the case where multiple actions were used the covariance was modeled with a diagonal covariance matrix. The exploration strategy used consists of decaying the covariance over time until a threshold value is reached, where the covariance is kept constant to still allow some exploration. The decay rate of the exploration noise is based on the number of steps the agent did take in the environment for the current episode. By relating the amount of exploration based on how much progress the agent is making in the environment rather than just the number of episodes conducted. The agent will be more likely to carry out lots of exploration while it is still learning the basics of the assigned task. The larger decay rate yields smaller exploration noise when the general concept of task is understood by the agent, but a smaller amplitude exploration noise is still necessary to make corrections to better optimize the policy. The decay rate is controlled by the parameter η. The exploration strategy is described in pseudocode format in Algorithm 2.

Algorithm 2 Exploration Strategy

```
# Initialize σ_exp, σ_end, Δ_σ
while i < max episodes do
    Reset environment
    j=0
    while j < max steps ∧ ¬terminal do
        take 1 step in environment
        j++
    # decay exploration
    σ_exp = max{σ_exp − jη, σ_end}
    i++
```

3.6 Reinforcement Learning Models

In this section, the necessary implementation details of the algorithms will be described. Firstly the DDPG algorithms model and parameters will be described followed by the complicated model of ACER along with its parameters. For all test and models, the Adam optimizer was used with learning rate 10^{-4}, first order momentum 0.9 and second-order momentum 0.999.

3.6.1 DDPG-Agent

As explained in Section 3.1.1 the simulations are carried out in two cases, with one and two actions respectively. Furthermore, the Actor and Critic are implemented with three fully connected layers. The proposed action from the Actor is merged with the output from the first layer in the Critic. The number of units used in each layer and visual representation of the Actor and Critic network is illustrated in Figure 3.6.

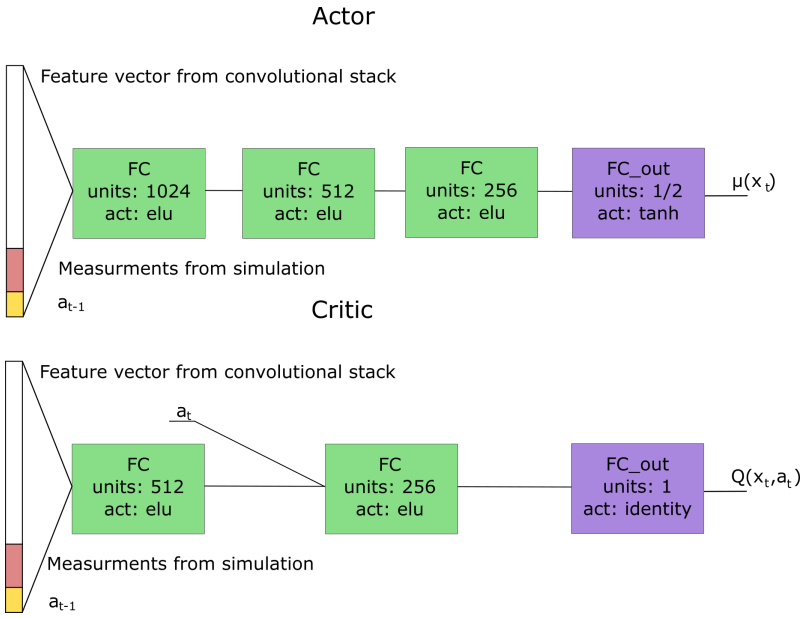


Figure 3.6: Schematic overview of the neural networks for the Actor and Critic in DDPG algorithm. The Actor acts upon the feature-vector from the pre-trained convolutional stack of the Imitation Learning agent, previous actions and the internal states obtained from the simulation environment. The output is modeled with one or two units depending on whether the model controls both steering and acceleration or just steering angles. The Critic uses both the feature-vector and the action from the Actor to estimate a Q-value. The action is concatenated with the output from the first layer to avoid that the actions drown in the input feature-vector.

To avoid catastrophic forgetting and be able to learn from previous policies experience a replay buffer is implemented. The buffer is designed to hold 50.000 transitions tuples. Before training, tuples were sampled from the buffer to form a batch of 256 transitions.
The exploration strategy follows the scheme presented in Section 3.5. The exploration noise for the steering angles starts with a standard deviation of 0.25 and decay for each episode until it stops at 0.025. Similarly, the exploration for the acceleration is performed, the exploration starts at 0.2 and stops at 0.01. The soft update parameter for the target network is set to 0.001. All parameters can be found in condensed form in Table 3.2.

Table 3.2: Hyperparameters used for the DDPG-algorithm for both one and two actions.

	1-dim	2-dim
τ	0.001	0.001
σ_{steer}^{init}	0.25	0.25
σ_{steer}^{end}	0.025	0.025
σ_{acc}^{init}	NA	0.2
σ_{acc}^{end}	NA	0.01
η_{steer}	3.75×10^{-7}	1.27×10^{-6}
η_{acc}	NA	2.53×10^{-7}
N_{batch}	256	256

3.6.2 ACER-Agent

To make the model capable of handling POMDPs, the original ACER architecture was extended to include a LSTM-cell. By adding the LSTM-cell, the model is capable of exploiting the sequential structure of the data, with the drawback that it will be harder to train.

The first layer after all the convolutions are performed is the LSTM-cell. The LSTM-cell receives an observation filtered through the convolutional layers trained using the supervised learning techniques. Worth noting is that the parameters for the convolutional layers are not a part of the parameters, θ, of the model. After the LSTM-cell the network is divided into two computational streams, one for the Critic and one for the Actor, hence the parameters in the LSTM-cell are shared between the Actor and the Critic. The computational stream for the Actor follows a conventional feed-forward neural network structure. However the computational stream for the Critic is divided into two different branches, one for the state value function $V^{\pi}(x_i)$ and one for the advantage function $A^{\pi}(x_i, a_i)$. The advantage function is calculated via the stochastic dueling network technique as

$$\tilde{A}(x_t, a_t) = \mathbf{w}_2^T \mathrm{ELU}\left(\mathbf{w}_1^T \left[h_{in}, a_t\right]^T\right) - \frac{1}{N}\sum_{i=1}^{N} \mathbf{w}_2^T \mathrm{ELU}\left(\mathbf{w}_1^T \left[h_{in}, \tilde{a}_i\right]^T\right), \ \tilde{a}_i \sim \pi\left(\cdot|x_t\right) \tag{3.5}$$

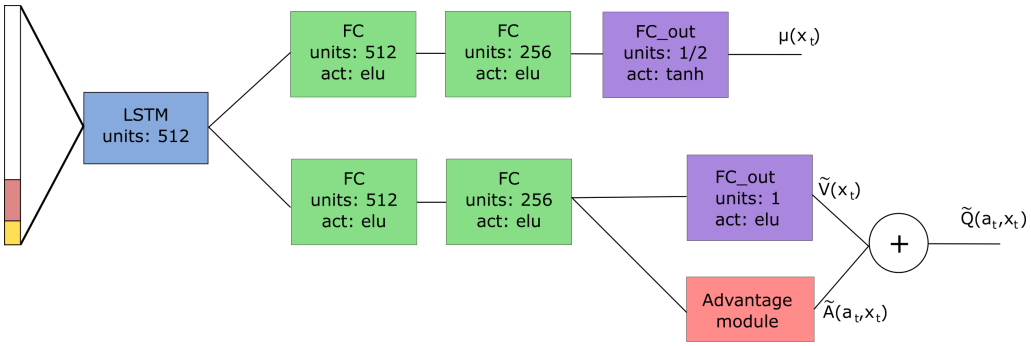


Figure 3.7: A schematic overview of the ACER agents network. The input vector from the image, previous action and measurements from the simulation is feed to an LSTM-cell with 512 units. The stream of data is then split into two branches of fully connected layers. The top stream proposes an action and lower stream estimates the action-value function through the advantage module. Depending on whether the acceleration is controlled through the model the action output units are either 1 or 2. Where the later option both steering angles and acceleration are controlled simultaneously. Equation (3.5) describes the mathematical operations in the advantage module.

where h_{in} are the hidden activations from the previous layers,
$\mathbf{w}_1$ is a $[128 \times (\dim(a_t) + \dim(h_{in}))]$ weight matrix and $\mathbf{w}_2$ is a $[1 \times 128]$ weight matrix.
The current estimate of $Q^{\pi}(x_t, a_t)$ is simply calculated by adding the two computational streams for $\tilde{A}(x_t, a_t)$ and $V(x_t)$, i.e

$$\tilde{Q}(x_t, a_t) = \tilde{A}(x_t, a_t) + V(x_t). \tag{3.6}$$

The policy, π, is modeled as a Gaussian distribution with a diagonal covariance matrix. The covariance matrix was not modeled by the network. Instead, a constant covariance matrix was used with the diagonal elements at a value of 0.3^2, as Z. Wang et.al used in the paper [6]. For the agent to explore the environment and try out different actions, the covariance matrix used for the sampling of actions followed the exploration scheme presented in Section 3.5 and thereby deviated from the distribution used for the gradients.

During the simulation episodes from the imitation, learning agent was inserted every 50th episode to provide high-quality data for the policy updates. However, the data obtained from the imitation learning agent was strictly used for the policy updates and not for the value-function updates For the case where only the steering-wheel angle was considered, the hyperparameters in Table 3.3 was used.
There were no changes made to the setup for the simulation for the control of both steering and acceleration except for the rather apparent extension of the action-space. To speed up the training of the model, the bias of the output node was initialized to the mean value for the acceleration obtained from the PID-controller.

The trust-region constraint δ, was chosen to be 0.01, to prioritize small steps on the policy instead of large noisy steps to minimize the risk of policy collapses. The value, τ, that controls the rate of the average policy updates was chosen analogously with the one presented in the ACER paper [6], to 0.005. The sequence length used for each update was chosen to be 80 transitions with a fairly large number of 20 skip steps to allow the internal states of the LSTM-cell to settle. The batch size was chosen to be 30, yielding a number of 1800 transitions for each update, which has to be considered as a very large batch, but empirically this showed to stabilize the training. The experience replay stored last 100 episodes in memory and each sequence was sampled uniformly from that set for each update. The initial standard deviation for the exploration noise regarding the steering angle was set to 0.25 and 0.2 and decayed to values of 0.025 and 0.01 respectively. The rate of the decay of the exploration noise is presented in Table 3.3, along with all the other hyperparameters for the simulation.

Table 3.3: Hyperparameters used for the ACER-algorithm for both one and two actions. In the table N_{batch} denotes the number of sequences used for each batch, N_{adv} denotes the number of samples used for Equation (3.5), l denotes the length of the sequence and h denotes the skip count.

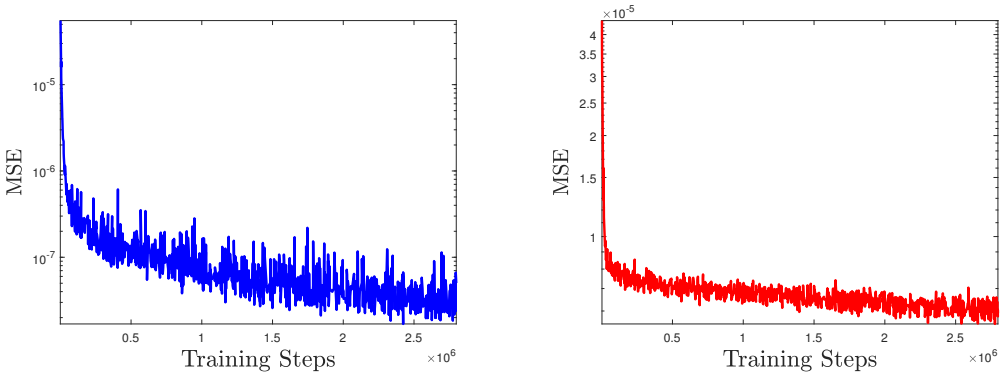
	1-dim	2-dim
τ	0.005	0.005
δ	0.01	0.01
c	5	5
l	80	80
h	20	20
σ_{steer}^{init}	0.25	0.25
σ_{steer}^{end}	0.025	0.025
σ_{acc}^{init}	NA	0.2
σ_{acc}^{end}	NA	0.01
η_{steer}	3.75×10^{-7}	1.27×10^{-6}
η_{acc}	NA	2.53×10^{-7}
N_{batch}	30	30
N_{adv}	100	100

4
Results

4.1 Training Results

4.1.1 Imitation Learning

The pre-training of the convolutional layers and consequently the training of the Imitation Learning agent was conducted analogously with the methodology described in Section 3.2.4. The training and validation losses are illustrated in Figure 4.1. As can be seen from the figure, the trends for both the training- and validation loss, are continuously decaying throughout the training procedure. Due to this artifact and time constraints, the early stopping never occurred during the training process. Due to the time constraints, the parameters of the model with the best results on the validation set after roughly 1.26×10^6 training steps were used. Even though the smaller validation losses were obtained after that point, the parameters of the convolutional layers, remained fixed to make the different results comparable.



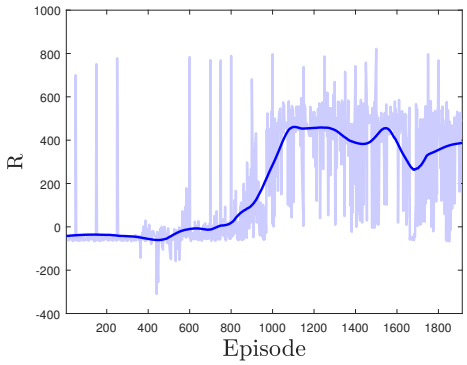
(a) Mean squared error loss on the training-set during the training.

(b) Mean squared error loss on the validation-set during the training.

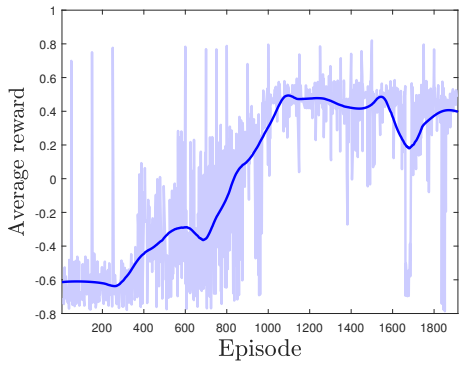
Figure 4.1: Mean squared error loss on both the training- and validation-set during the training.

4.1.2 DDPG

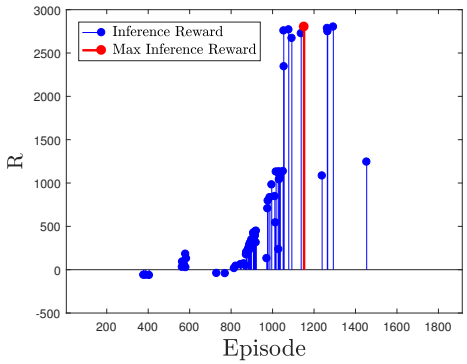
The training of the DDPG-agent was conducted analogously with the methodology presented in Section 3.6.1. Since the DDPG-algorithm works off-policy, the Imitation Learning agent was used to provide high-quality data for the replay buffer, every 50th episode. Figure 4.2 summarizes the training results obtained for the DDPG-agent. As can be seen from the Figure 4.2, both the average reward and the accumulative reward remains relatively unchanged until approximately 400 episodes after which the trend is increasing until roughly 1000 episodes. After approximately 1250 episodes the exploration noise has reached its minimum value, as can be seen from the figure, this low level of explorations also yields a slightly decaying trend on the model's performance.



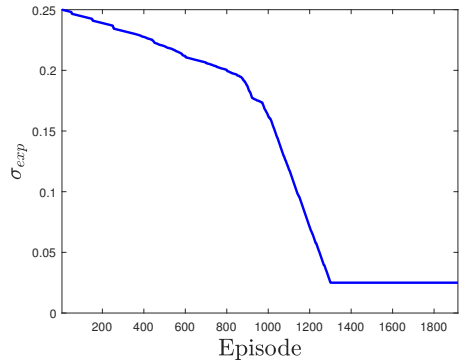
(a) Accumulative reward, R, per episode.



(b) Average reward per step for each episode.



(c) Accumulative reward, R, for each inference run.



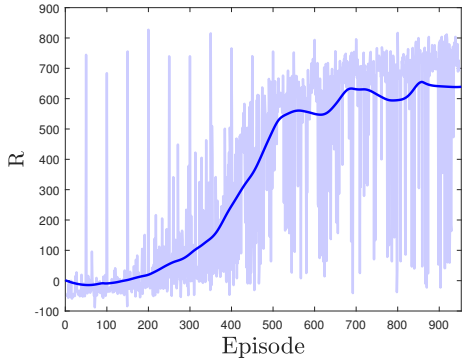
(d) Standard deviation of exploration noise for each episode.

Figure 4.2: Result and logs for the training of the DDPG-algorithm where only the steering angle was considered for control. The dark-blue line in the uppermost images is a smoothened version of the true data.

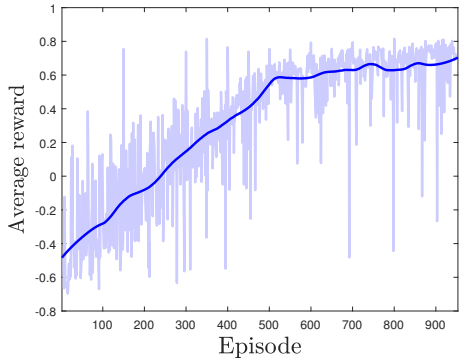
4.1.3 ACER

The design and training of the ACER-agent were performed as described in Section 3.6.2. The agent was trained for two levels of control, one for control only of the steering and one for control of both the acceleration and steering. Firstly the training results for control of only the steering will be presented followed by the training results where the action-space is extended to include the acceleration.

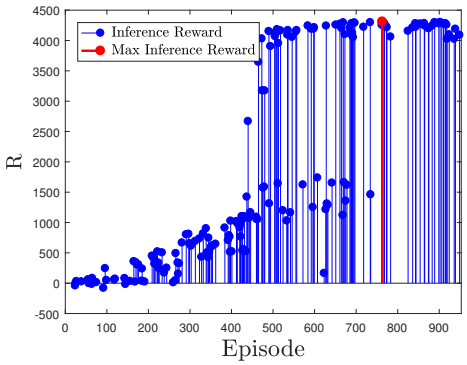
The training of the agent for control of the steering angle is summarized in Figure 4.3. Similarly, as for the training of the DDPG-agent, the Imitation Learning agent was deployed at every 50th episode, to fill up the replay buffer with high-quality data. As can be seen from the figure, the per episode performance of the agent starts to increase per episode almost immediately. The average reward per step seems to almost increase linearly until roughly the point of 500 episodes, after which the inference reward suggests that the increasing average reward is due to the lower rate of exploration noise, as seen in Figure 4.3d.



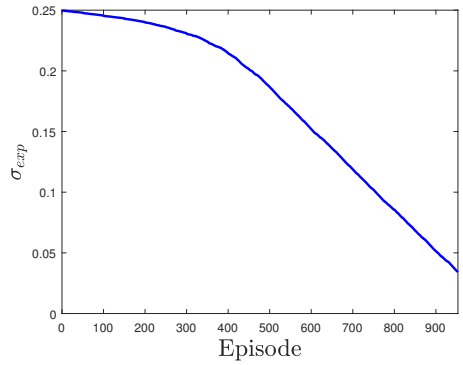
(a) Accumulative reward, R, per episode.



(b) Average reward per step for each episode.



(c) Accumulative reward, R, for each inference run.



(d) Standard deviation of exploration noise for each episode.

Figure 4.3: Result for the training of the ACER-algorithm where only the steering angle was considered for control.

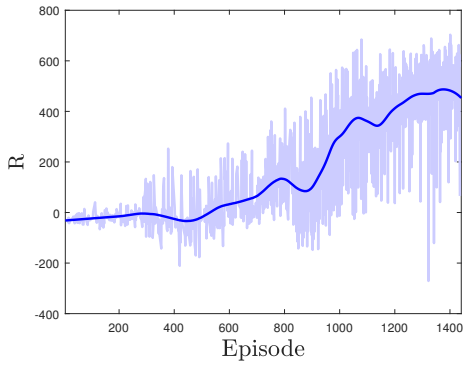
For the training of the agent that controls both the steering and acceleration of the vehicle, the policy stream was initialized with the parameters from the policy with the best inference performance, see Figure 4.3c. As can be seen from Figure 4.4c, it performs relatively well on the inference test directly at the start, due to that the part of the policy that controls the steering is initialized with the best policy from the previous training. From Figure 4.4a and 4.4b, it can be seen that both the accumulative episodic reward and the average reward is increasing at a satisfactory rate. However, due to the good initial steering policy, the agent utilizes the full episodic length which yields a rapid decay of the exploration noise. After roughly 300 episodes the exploration noise was already at a minimum amount, but the agent still struggled with controlling the acceleration. Due to this fact, the exploration strategy was changed to include a fixed standard deviation for the noise of both the acceleration and steering after 300 episodes, see Figure 4.4d. After the exploration strategy was changed, the reward and average reward increased a little while before saturating at around 500 episodes. Figure 4.4a and 4.4b, can, however, be hard to interpret since a larger level of exploration noise effectively will lower both the average reward per step and the accumulative episodic reward. The inference reward, on the other hand, yields a more fair comparison between the performance of the agent under the two exploration strategies, since the agents actions during inference correspond to the modes of the distributions. From Figure 4.3c it can be seen that the accumulative reward obtained during inference increases under the new exploration strategy and reaches its maximum value after roughly a total of 1300 episodes.

4.2 Model Comparison

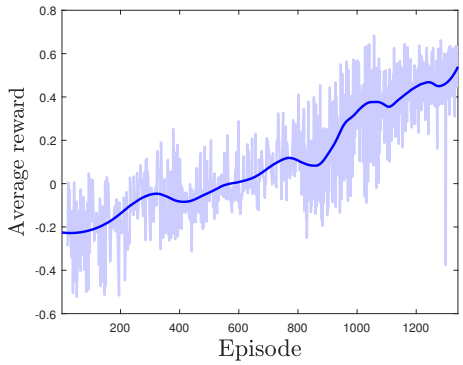
4.2.1 Driver Metrics

To compare the performance of the different agents, the normalized lane position α and heading angle φ, were logged for two validation laps around the track. During the validation laps, the control signals used for the different agents corresponded to the modes of the distributions. Histograms of the normalized lane position, α and heading angle φ for all the validation laps are summarized in Figure 4.5 and Figure 4.6. As can be seen from Figure 4.5b, the histogram for the ACER-agent that controls both the steering and the acceleration, has a considerable fewer number of samples than the other models. This is due to that the agent failed to complete the two laps as it drove outside of the road during the first lap.

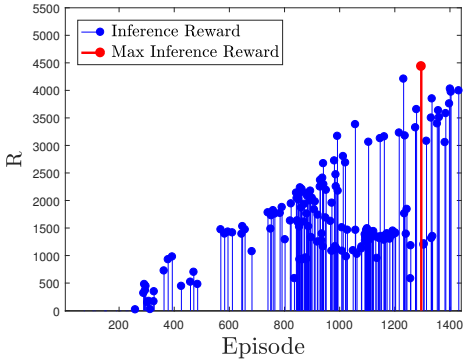
Figure 4.6, summarizes the histograms of the heading angle, φ, obtained from the validation laps for all the agents. As for the case of the histogram of the normalized lane position, α, the histogram for the ACER-agent for control of both the acceleration and steering has considerably fewer samples due to that it failed to complete the two laps.



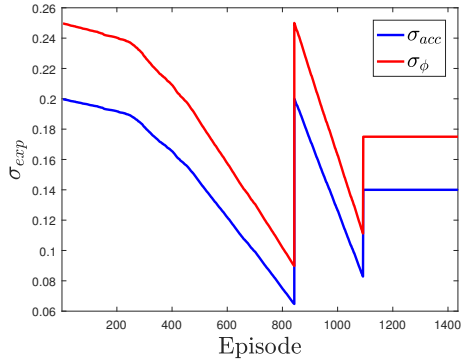
(a) Accumulative reward, R, per episode.



(b) Average reward per step for each episode.



(c) Accumulative reward, R, for each inference run.

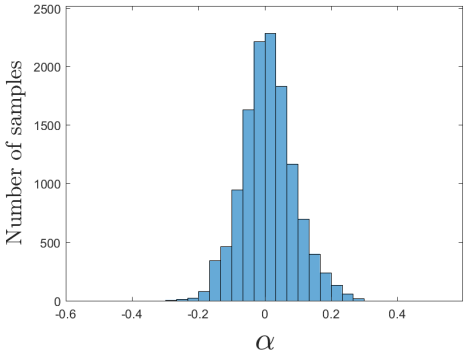


(d) Standard deviation of exploration noise for each episode.

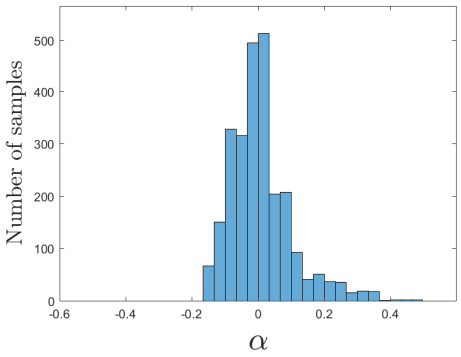
Figure 4.4: Result for the training of the ACER-algorithm where both the steering angle and acceleration were considered for control.

4.2.2 Comfort Metrics

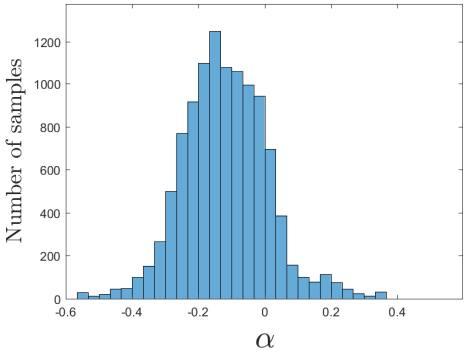
To further analyze the performance of the agents from a human passenger perspective the comfort-metrics acceleration and jerk were logged during the two validation laps of the agents. In Figure 4.7 and Figure 4.8 the red bins indicate levels of acceleration and jerks where humans experience discomfort. As mentioned in Chapter 4.2.1 the histograms for the ACER agent with two actions contains fewer samples because the agent drove off the road.



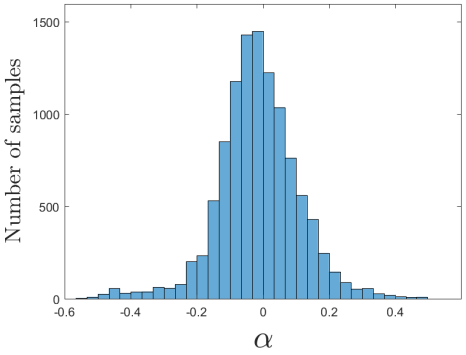
(a) Histogram of normalized lane position, α, for ACER-agent with 1 action.



(b) Histogram of normalized lane position, α, for ACER-agent with 2 action.

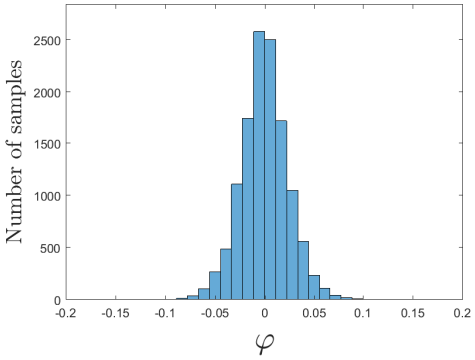


(c) Histogram of normalized lane position, α, for the Imitation Learning agent.

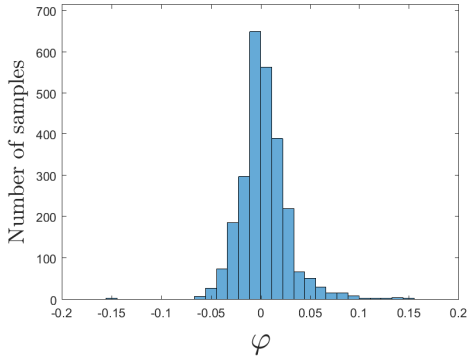


(d) Histogram of normalized lane position, α, for the DDPG-agent with 1 action.

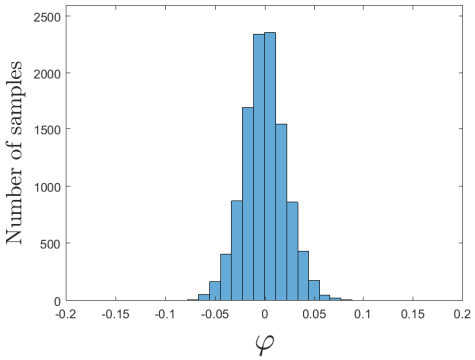
Figure 4.5: Histograms of the normalized lane position α, for all the successful implementations. The data was logged for two laps around the test-track.



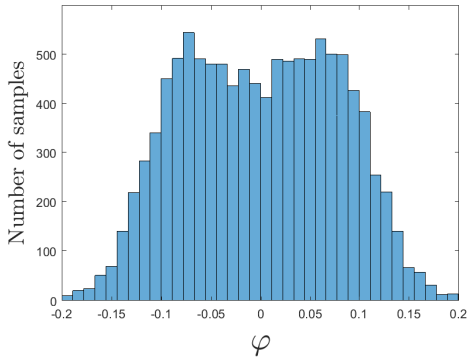
(a) Histogram of the relative heading angle, φ, for ACER-agent with 1 action.



(b) Histogram of the relative heading angle, φ, for ACER-agent with 2 action.

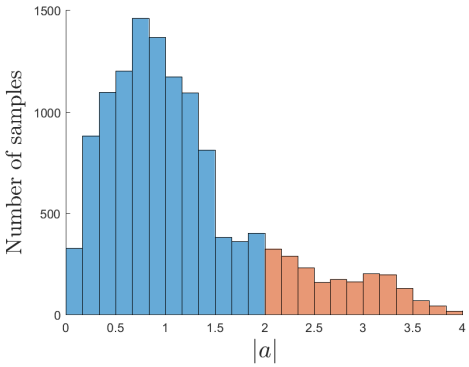


(c) Histogram of the relative heading angle, φ, for the Imitation Learning agent.

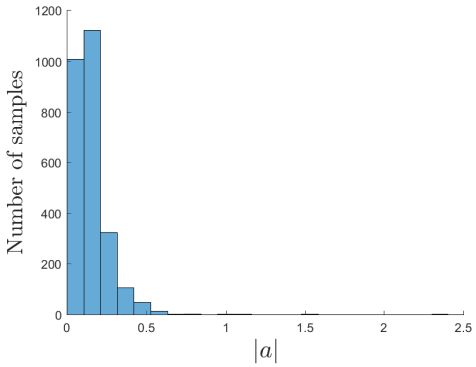


(d) Histogram of the relative heading angle, φ, for the DDPG-agent with 1 action.

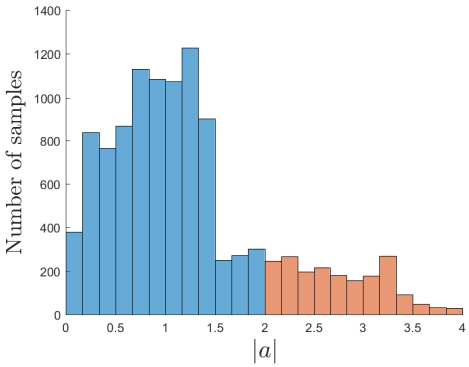
Figure 4.6: Histograms of the relative heading angle φ, for all the successful implementations. The data was logged for two laps around the test-track. The unit for the x-axis of all the plots are radians.



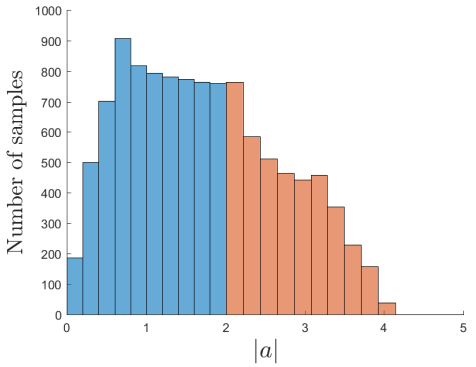
(a) Histograms of accelerations $[m/s^2]$ for the ACER-agent with 1 action.



(b) Histograms of accelerations $[m/s^2]$ for the ACER-agent with 2 action.

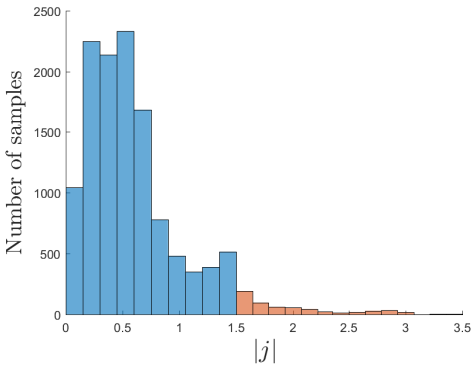


(c) Histograms of accelerations $[m/s^2]$ for the Imitation Learning agent.

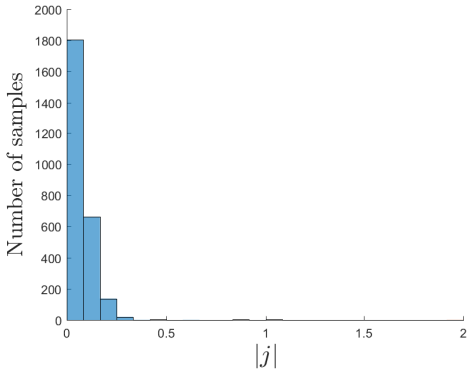


(d) Histograms of accelerations $[m/s^2]$ for the DDPG-agent with 1 action.

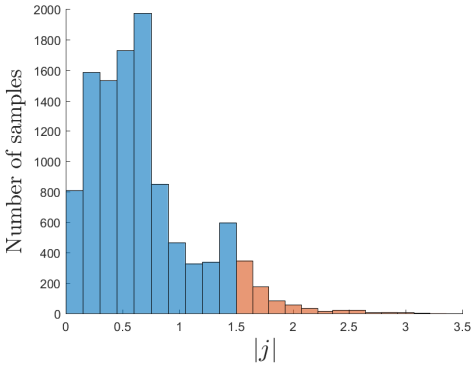
Figure 4.7: Histograms of accelerations $[m/s^2]$, for all the successful implementations. The data was logged for two laps around the test-track. Any red bins suggests that a human would be introduced to a large level of discomfort under those conditions.



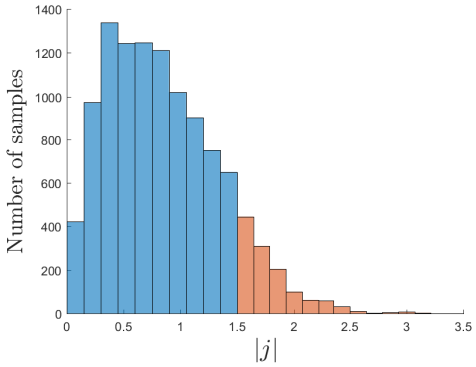
(a) Histograms of jerks $[m/s^3]$ for the ACER-agent with 1 action.



(b) Histograms of jerks $[m/s^3]$ for the ACER-agent with 2 actions.



(c) Histograms of jerks $[m/s^3]$ for the Imitation Learning agent.



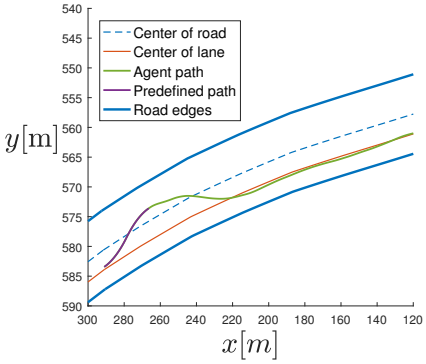
(d) Histograms of jerks $[m/s^3]$ for the DDPG-agent with 1 action.

Figure 4.8: Histograms of jerks $[m/s^3]$, for all the successful implementations. The data was logged for two laps around the test-track. Any red bins suggests that a human would be introduced to a large level of discomfort under those conditions.

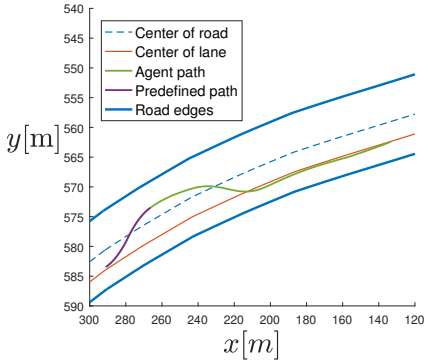
4.2.3 Robustness

To evaluate the model's performance in non-ideal scenarios, the models were evaluated on two different test-sequences. For the first test, the agent is exposed to an error sequence that places the vehicle in the wrong lane of the road. For the second test, the agent is exposed to an error sequence that forces the vehicle to make a sharp left turn. The agent retains control of the vehicle at the point where the vehicle crosses the line that separates the left lane from the right. The evaluated models on the test-sequences were the two agents obtained via the ACER-algorithm, the DDPG-agent for control of the vehicles steering and the agent obtained via Imitation Learning.

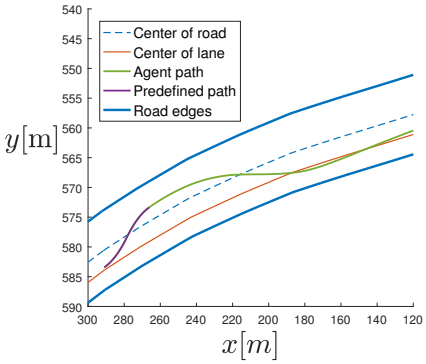
The resulting trajectories for all the models exposed to the first test-sequence are summarized in Figure 4.9. As can be seen from the figure all the models are capable of recovering from the unwanted state where the agent is placed in the wrong lane. The difference between the models are mainly the trajectories generated by the different agents. As can be seen from the Figure 4.9, the DDPG- and ACER-agent for control of the steering angle, recover from the unwanted state faster than the other agents.
For the second test, all the agents did not perform as well as for the first one. As seen in Figure 4.10, only the DDPG- and ACER-agent for steering-control managed to recover from the unwanted state caused by the error-sequence. The seen in the figure, both the ACER-agent for control of both the steering and acceleration as well as the Imitation Learning agent barely applies any compensation on the steering, and therefore fails to stay on the road.



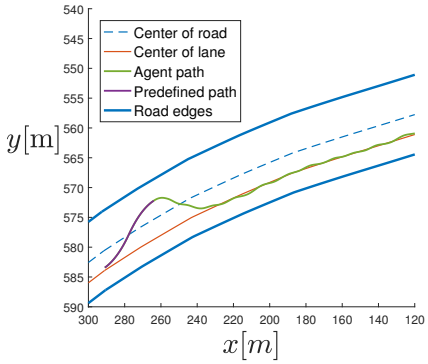
(a) Robustness test for ACER with 1 action.



(b) Robustness test for ACER with 2 action.

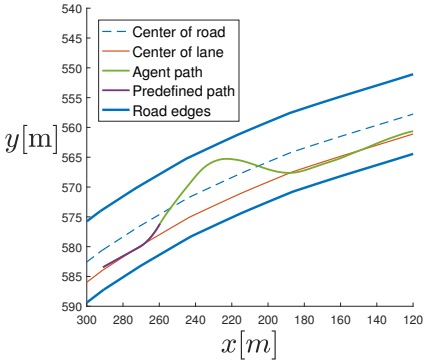


(c) Robustness test for the Imitation Learning agent.

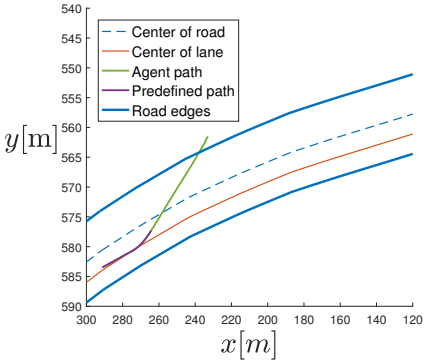


(d) Robustness test for the DDPG algorithm with 1 action.

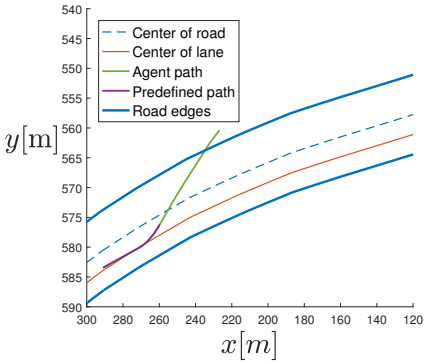
Figure 4.9: Robustness test for the different agents. Each agent is introduced to a error sequence that places the vehicle in the wrong driving lane.



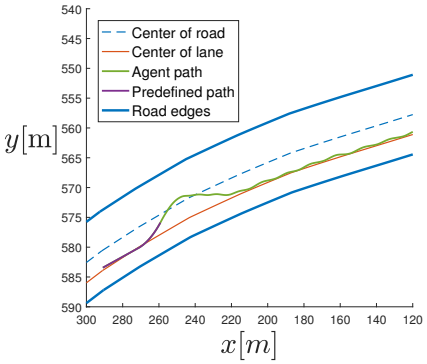
(a) Robustness test for ACER with 1 action.



(b) Robustness test for ACER with 2 action.



(c) Robustness test for the Imitation Learning agent.



(d) Robustness test for the DDPG algorithm with 1 action.

Figure 4.10: Robustness test for the different agents. Each agent is introduced to a error sequence that places the vehicle in the wrong driving lane.

5
Discussion

In this chapter, the results presented in Chapter 4 will be discussed, as well as the overall performance of the models. The Chapter is divided into three sections, Supervised Learning, Training Reinforcement Learning Models and Driving Behaviour and Robustness. In the first section, a discussion is conducted regarding the training results of the convolutional network and the Imitation Learning agent. In the next section, the training and learning from the Reinforcement Learning agent is discussed, as well as a discussion regarding why some agents failed to learn the task. Lastly the general driving behavior, comfort metrics are discussed in aspects concerning the implementation errors.

5.1 Supervised Learning

As discussed in Section 4.1.1, the early-stopping strategy used for the supervised training never materialized, since the validation error continued to decrease. The trend of the validation error is most likely an artifact of large correlation of the samples. Although the data was divided into two subsets of individual samples, there is still a correlation between the two sets, due to the fact that the data was collected on the same track. A better strategy would probably have been to assign the samples into different subsets based on where the data was collected on the track, effectively leading to that the validation-set only containing parts of the track that are unseen by the model during training.

Apart from the early-stopping strategy, there were no regularization techniques applied to the model during training. The absence of regularization is most likely nothing that affected the performance of the models, on the data used in this thesis. However, implementing regularization techniques such as dropout during training could help the models to generalize on unseen data.

The Imitation Learning agent obtained via the supervised learning showed promising results on the test conducted within this thesis. The Imitation Learning agent was never a part of the initial goals for the thesis, but more of an artifact of the pre-training of the convolutional layers. Because of this, the model-design of the Imitation Learning agent is in some sense naive, but based on the performance of the obtained Imitation Learning agent it would be interesting to see how an implementation with recurrent units and better regularization would perform.

As mentioned in Section 4.1.1, the supervised training was performed past the point where the parameters of the model were stored. This was done since that there was simply not enough time to wait any longer, combined with the assumptions about the correlation of the data as mentioned previously in this section.

5.2 Training of Reinforcement Learning Models

Generally, the training of RL models is quite a lengthy process. Because of this a lot of the time spent on this thesis has been dedicated to training the models evaluated in this thesis. Since both the DDPG- and ACER-algorithm contains a number of hyperparameters, there simply has not been enough time to perform any hyperparameter tuning. To which extent the models could be improved with tuned hyperparameters remains unknown, but the DDPG-algorithm is however known to be sensitive to hyperparameter settings[37].

The DDPG-agent for control of one action initially showed a promising learning-curve, but both the accumulative reward and the average reward did unfortunately saturate after roughly 1100 episodes. The resulting agent did seem to understand the basic concept of the task, since it managed to stay in the correct lane. From Figure 4.5d it can also be seen that the agent kept a relatively good lane-position throughout the two inference laps. However from Figure 4.6, it can be seen that it failed to maintain a small value of the heading angle, which suggests an oscillating behavior, which was also verified empirically by observing the agents behavior visually. The oscillating behavior could probably be reduced and thereby increasing the agents performance, by applying a carefully selected filter on the output. However, any further investigations on this subject were never conducted. Since the policy gradient for the DDPG-algorithm is based on estimates of the action value function, a biased action value function will also yield an imperfect policy gradient, which we believe is a probable explanation of the oscillating behavior. A better approach might be to combine gradients from on-policy returns with the deterministic policy gradients, which is utilized with the Q-Prop[37] and Interpolated Policy Gradient algorithm[38].

The DDPG-algorithm did however fail to produce an agent that was capable of obtaining an accumulative reward larger than zero, for the case when the agent controlled both the steering and acceleration. Attempts to train the agent were made with and without data from the Imitation Learning agent included in the replay-buffer. The agent showed promise during the training since it seemed to realize that driving off the road will yield a negative reward, but it failed to find a good combination of steering and acceleration to compensate for that behavior. Maybe another exploration policy could have solved these problems. A popular approach to model the exploration-noise for the DDPG-algorithm is to model the noise as an Ornstein–Uhlenbeck process, which was suggested in the Continuous control with deep reinforcement paper learning[5].

The agent that showed the most promising results among the different agents ob-

tained via Reinforcement Learning methods was the ACER-agent with control of the steering. As can be seen from Figure 4.3, the learning curve concerning the accumulative reward and the average reward shows a positive trend more or less throughout the whole simulation. From Figure 4.5a it can be seen that the agent manages to maintain a relatively good lane position and Figure 4.6a shows that it simultaneously manages to keep a small heading angle.

For the ACER-agent that controls both the acceleration and steering the increased complexity of including the acceleration as a control-signal increased the training time significantly. Due to the growing complexity and the number of required training-steps required, the initial exploration strategy was not well suited to the problem. From Figure 4.4d, it can be seen that the standard deviation of the exploration noise decayed too quickly and the simulation was therefore restarted and restored with the weights of the agent with the best performance on the inference tests at that point. What is seen in the figure is the episodic trace obtained from the agent with the best inference results. The simulation was continued past the first point of interruption, where the results indicated a decaying level of performance. The final agent did not reach the same degree of performance as for the case when only one action was considered. Even though the agent obtained the highest inference reward during training, it failed to replicate the result when the performance metrics presented in Section 4.2.1 and 4.2.2 were evaluated since it crashed during the data acquisition. The main reason for the higher inference reward is most likely due to that control-system used on the acceleration for the other agents was not properly tuned and therefore applied very large control-signal and thus also a significant penalty, due to large acceleration and jerks, as seen in Equation (3.4). Due to the large control-signals obtained from the control system, data from the Imitation Learning agent was not included in the replay buffer.

5.3 Driving Behaviour and Robustness

Due to an error in the implementation of the acceleration and jerk in the simulation environment, the samples were not divided with the sampling period. The data used for Figure 4.7 and Figure 4.8, were manipulated to compensate for that error. Based on the results in Figure 4.7 and Figure 4.8, there seems to be a significant deviation between the agents using a PID-controller for the acceleration and the ACER-agent that handles this control-signal internally. That should however not be interpreted as an indication that an RL algorithm is more suitable for controlling the acceleration since it is most likely an artifact of a poorly tuned controller. What can be seen from the figure, however is that the ACER-agent and the Imitation Learning agent produced similar results when it comes to driver comfort.

Even though the scale of the investigations regarding robustness is not thorough enough to draw any major conclusions, the tests performed within this thesis shows some promising results for RL for control of autonomous vehicles. As can be seen from Figure 4.9, all the agents managed to recover from an error-sequence that

placed the vehicle in the incorrect lane. The robustness towards the incorrect lane-position might not be that surprising since all the agents have been exposed to that type of data during the training. This does however show that the agents are not strictly performing the task of driving within the current lane, but also realizes that it should stay on the right side of the road. For the second test where a large error signal was applied to the steering, the two Reinforcement Learning agents that only controls the steering still manages to recover from the unwanted state introduced by the error signal. The ACER-agent for control of both the acceleration and steering as well as the Imitation Learning agent, both failed to recover from the unwanted state introduced. For the Imitation Learning agent, it is very likely that the bad performance is due to that type of data never being included in the training-set. When the data was logged for the training- and validation-set, the acquisition of data with "good behavior" in bad states was a priority. However since this sequence starts with an observation directly followed by a sequence of bad behavior, it is very likely that this type of data was not included in the training-set only because logging of bad behavior was avoided during the data acquisition. Regarding the successful cases, we did not find any objective way to compare the performance between the different results since it is very difficult to quantify. However, intuitively it feels like the DDPG-agent performed the best, since it managed to recover from the unwanted states in the shortest amount of time during both tests, but doing so it also caused large jerks and accelerations.

As can be seen from Figure 4.5, it can be seen that there is a large bias on the lane position for the Imitation Learning agent. This did not agree with the empirical results obtained by monitoring the validation-laps[1]. The biased value for the normalized heading angle, could be an artifact of our own biased driving, but most likely it is due to a bias of the nominal lane-center in the environment.

[1]A short video of the ACER and Supervised agent driving around the track and making the robustness tests, https://www.youtube.com/watch?v=pqGSublT02w.

6
Conclusion

Within this thesis, it is shown that the current Deep Reinforcement Learning algorithms are capable of producing a policy for steering a vehicle in a simulated environment. The models were trained using image data concatenated with internal states of the vehicle, i.e., current velocity, acceleration, and jerk. All of the models were also capable of recovering from an error sequence that placed the vehicle in the wrong driving lane. There are also some indications that the models are more robust than the policy obtained through supervised learning, against large offsets on the heading angle.

7
Future Work

Primarily the agents developed in this thesis would need a proper evaluation on real-world data to evaluate the significance of training on synthetic data. In the current state, the simulation environment is not mature for the available datasets. Several issues need to be handled, to lower the discrepancy between the simulation environment and the available datasets. Most of these issues can, however, be solved by extending the simulation environment with, e.g., other vehicles and city environments. One possible issue that will most likely remain is that the synthetic data obtained from the environment is not very diverse regarding e.g. roads and scenery.

If more features were to be included in the environment, e.g., other road users and city environments, the reward function would most likely become more complex. Due to the complexity of a reward function that extends beyond the limited scenarios used within this thesis, a thorough evaluation of the reward function of driving is needed. Since a reward function that promotes an optimal driving policy might be hard to derive analytically, a promising method to this problem is Inverse Reinforcement Learning. Unlike RL, the goal of Inverse Reinforcement Learning is to estimate a reward function based on some expert behavior.

One of the drawbacks of Neural Networks is the "black-box" behavior of the models. For safety critical implementations such as autonomous driving, this might pose one of the biggest obstacles to overcome to make them a viable solution to autonomous driving. Because of this future research on the verification of Neural Networks in safety critical systems, would be interesting. As a starting point, an in-depth analysis of how the models utilize the available features to make decisions. The two most obvious investigations are an analysis of how much the LSTM-cell exploits the sequential structure of the data and an analysis of which parts of the images yield large activations in the convolutional layers.

Research and development in Neural Networks and RL are very much an active area. Because of the substantial activity within these fields, the performance of the algorithms being developed is still rapidly increasing. Because of the fast increasing performance of the newly proposed models, the investigations performed within this thesis could be revisited as the algorithms become more mature.

Bibliography

[1] IMAGENET, "Large scale visual recognition challenge," http://www.image-net.org/, accessed Mars 21, 2017.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," CoRR, vol. abs/1312.5602, 2013. [Online]. Available: http://arxiv.org/abs/1312.5602

[3] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. P. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," Nature, vol. 529, no. 7587, pp. 484–489, 2016. [Online]. Available: http://dx.doi.org/10.1038/nature16961

[4] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, "Trust region policy optimization," CoRR, vol. abs/1502.05477, 2015. [Online]. Available: http://arxiv.org/abs/1502.05477

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

[6] Z. Wang, V. Bapst, N. Heess, V. Mnih, R. Munos, K. Kavukcuoglu, and N. de Freitas, "Sample efficient actor-critic with experience replay," CoRR, vol. abs/1611.01224, 2016. [Online]. Available: http://arxiv.org/abs/1611.01224

[7] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems, Oct 2012, pp. 5026–5033.

[8] M. Alzantot, S. Chakraborty, and M. B. Srivastava, "Sensegen: A deep learning architecture for synthetic sensor data generation," CoRR, vol. abs/1701.08886, 2017. [Online]. Available: http://arxiv.org/abs/1701.08886

[9] M. Johnson-Roberson, C. Barto, R. Mehta, S. N. Sridhar, and R. Vasudevan, "Driving in the matrix: Can virtual worlds replace human-generated

annotations for real world tasks?" CoRR, vol. abs/1610.01983, 2016. [Online]. Available: http://arxiv.org/abs/1610.01983

[10] X. Zhang, Y. Fu, A. Zang, L. Sigal, and G. Agam, "Learning classifiers from synthetic data using a multichannel autoencoder," CoRR, vol. abs/1503.03163, 2015. [Online]. Available: http://arxiv.org/abs/1503.03163

[11] I. Net-Scale Technologies, "Autonomous off-road vehicle control using end-to-end learning," http://net-scale.com/doc/net-scale-dave-report.pdf, accessed Mars 21, 2017.

[12] R. Hadsell, P. Sermanet, J. Ben, A. Erkan, M. Scoffier, K. Kavukcuoglu, U. Muller, and Y. LeCun, "Learning long-range vision for autonomous off-road driving," Journal of Field Robotics, vol. 26, no. 2, pp. 120–144, 2009.

[13] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, "End to end learning for self-driving cars," CoRR, vol. abs/1604.07316, 2016. [Online]. Available: http://arxiv.org/abs/1604.07316

[14] B. Huval, T. Wang, S. Tandon, J. Kiske, W. Song, J. Pazhayampallil, M. Andriluka, P. Rajpurkar, T. Migimatsu, R. Cheng-Yue, F. Mujica, A. Coates, and A. Y. Ng, "An empirical evaluation of deep learning on highway driving," CoRR, vol. abs/1504.01716, 2015. [Online]. Available: http://arxiv.org/abs/1504.01716

[15] F. Sadeghi and S. Levine, "(cad)^{2}rl: Real single-image flight without a single real image," CoRR, vol. abs/1611.04201, 2016. [Online]. Available: http://arxiv.org/abs/1611.04201

[16] D. Loiacono, L. Cardamone, and P. L. Lanzi, "Simulated car racing championship: Competition software manual," CoRR, vol. abs/1304.1672, 2013. [Online]. Available: http://arxiv.org/abs/1304.1672

[17] J. Fischer, N. Falsted, M. Vielwerth, J. Togelius, and S. Risi, Monte-Carlo Tree Search for Simulated Car Racing. United States: Association for Computing Machinery, 2015.

[18] J. Koutník, G. Cuccu, J. Schmidhuber, and F. Gomez, "Evolving large-scale neural networks for vision-based reinforcement learning," in Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation, ser. GECCO '13. ACM, 2013, pp. 1061–1068. [Online]. Available: https://pdfs.semanticscholar.org/fe4d/de1ea2fb09e65b990ce0c661ef151301dae4.pdf

[19] D. Loiacono, A. Prete, P. L. Lanzi, and L. Cardamone, "Learning to overtake in torcs using simple reinforcement learning," in IEEE Congress on Evolutionary Computation, 2010.

[20] A. N. Matt Vitelli, "Carma: A deep reinforcement learning approach to autonomous driving," 2016, https://web.stanford.edu/~anayebi/projects/CS_239_Final_Project_Writeup.pdf.

[21] UNREAL, "unreal engine-4," https://www.unrealengine.com/what-is-unreal-engine-4, accessed Mars 21, 2017.

[22] "Astazero proving ground," http://www.astazero.com/the-test-site/test-environments/rural-road/, accessed Mars 21, 2017.

[23] I. Goodfellow, Y. Bengio, and A. Courville, Deep Learning. MIT Press, 2016, http://www.deeplearningbook.org, note = accessed Juli 18, 2017.

[24] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in AISTATS, 2010.

[25] D. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (elus)," CoRR, vol. abs/1511.07289, 2015. [Online]. Available: http://arxiv.org/abs/1511.07289

[26] R. Pascanu, T. Mikolov, and Y. Bengio, "Understanding the exploding gradient problem," CoRR, vol. abs/1211.5063, 2012. [Online]. Available: http://arxiv.org/abs/1211.5063

[27] R. S. Sutton and A. G. Barto, Reinforcement Learning An Introduction. MIT Press, 1998, https://mitpress.mit.edu/books/reinforcement-learning.

[28] N. Heess, J. J. Hunt, T. P. Lillicrap, and D. Silver, "Memory-based control with recurrent neural networks," CoRR, vol. abs/1512.04455, 2015. [Online]. Available: http://arxiv.org/abs/1512.04455

[29] J. Schulman, P. Moritz, S. Levine, M. I. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," CoRR, vol. abs/1506.02438, 2015. [Online]. Available: http://arxiv.org/abs/1506.02438

[30] N. H. T. D. D. W. David Silver, Guy Lever and M. Reidmiller, "Deterministic policy gradient algorithms," in ICML, 2014.

[31] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," CoRR, vol. abs/1602.01783, 2016. [Online]. Available: http://arxiv.org/abs/1602.01783

[32] R. Munos, T. Stepleton, A. Harutyunyan, and M. G. Bellemare, "Safe and efficient off-policy reinforcement learning," CoRR, vol. abs/1606.02647, 2016. [Online]. Available: http://arxiv.org/abs/1606.02647

[33] Z. Wang, N. de Freitas, and M. Lanctot, "Dueling network architectures for deep reinforcement learning," CoRR, vol. abs/1511.06581, 2015. [Online]. Available: http://arxiv.org/abs/1511.06581

[34] P. Abbeel and A. Y. Ng, "Apprenticeship learning via inverse reinforcement learning," in Proceedings of the twenty-first international conference on Machine learning. ACM, 2004, p. 1.

[35] Vgu, "grundvärden," vv publikation 2004:80, 2004.

[36] J. Sorstedt, L. Svensson, F. Sandblom, and L. Hammarstrand, "A new vehicle motion model for improved predictions and situation assessment," IEEE Transactions on Intelligent Transportation Systems, vol. 12, no. 4, pp. 1209–1219, 2011.

[37] S. Gu, T. P. Lillicrap, Z. Ghahramani, R. E. Turner, and S. Levine, "Q-prop: Sample-efficient policy gradient with an off-policy critic," CoRR, vol. abs/1611.02247, 2016. [Online]. Available: http://arxiv.org/abs/1611.02247

[38] S. Gu, T. P. Lillicrap, Z. Ghahramani, R. E. Turner, B. Schölkopf, and S. Levine, "Interpolated policy gradient: Merging on-policy and off-policy gradient estimation for deep reinforcement learning," CoRR, vol. abs/1706.00387, 2017. [Online]. Available: http://arxiv.org/abs/1706.00387

A

Pseudocode Adam-optimizer

Algorithm 3 Adam

Require: Step size ϵ (suggested default: 0.001)
Require: Exponentially decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults 0.9 and 0.999 respectively)
Require: Small constant δ used for numerical stabilization. (Suggested default: 10^{-8})
Require: Initial parameters θ
Initialize 1st and 2nd moment variables $\mathbf{s} = \mathbf{0}$, $\mathbf{r} = \mathbf{0}$
Initialize time step t=0
while Stop criterion not met **do**
  Sample a minibatch of m examples from the training set $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\}$ with corresponding targets $\mathbf{y^{(i)}}$.
  Compute gradient: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_\theta \sum_i L(f(\mathbf{x}^{(i)}|\theta), \mathbf{y^{(i)}})$
  $t \leftarrow t + 1$
  Update biased first moment estimate: $s \leftarrow \rho_1 s + (1 - \rho_1)\mathbf{g}$
  Update biased second moment estimate $\mathbf{r} \leftarrow \rho_2 \mathbf{r} + (1 - \rho_2)\mathbf{g} \odot \mathbf{g}$
  Correct bias in first moment: $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1-\rho_1^t}$
  Correct bias in second moment: $\hat{\mathbf{r}} \leftarrow \frac{\mathbf{r}}{1-\rho_2^t}$
  Compute update $\Delta\theta = -\epsilon \frac{\hat{\mathbf{s}}}{\sqrt{\mathbf{r}}+\delta}$ (Operation applied element-wise)
  Apply update: $\theta \leftarrow \theta + \Delta\theta$

B

Pseudocode Deep Deterministic Policy Gradient

Algorithm 4 Deep Deterministic Policy Gradient

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s, \theta^\mu)$ with weights θ^Q and θ^μ
Initialize target network Q' and μ with weights $\theta^{Q'} \leftarrow \theta^Q,\ \theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode=1,M **do**
  Initiate random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t=1,T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, a_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_i, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|s = s_i, a = \mu(s_i)\nabla_{\theta^\mu}\mu(s|\theta^\mu)|s_i$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

C

Pseudocode Sample Efficient Actor-Critic with Experience Replay

Algorithm 5 Sample Efficient Actor-Critic with Experience Replay

sample $\{a_0, x_0, r_0, \mu(\cdot|x_0), ..., a_k, x_k, r_k, \mu(\cdot|x_k)\}$ from replay

for $i \in \{0, ..., k\}$ **do**

$Q^{ret} \leftarrow \begin{cases} 0, \text{ for terminal } x_k \\ V(x_k), \text{otherwise} \end{cases}$

for $i \in \{k-1, ..., 0\}$ **do**

$Q^{ret} \leftarrow r_i + \gamma Q^{ret}$

Accumulate gradients w.r.t the policy

$\{a'_0, a'_1, ..., a'_N\} \sim \pi(\cdot|x_i)$

$g_t^{\text{acer}} = \bar{\rho}_t \nabla_{\phi(x_t)} \log f\left(a_t|\phi\left(x_t\right)\right)\left(Q^{ret}\left(x_t, a_t\right) - V\left(x_t\right)\right)$

$+\frac{1}{N}\sum_{i=1}^{N}\left[\frac{p_t(a'_i)-c}{p_t(a'_i)}\right]_+\left(\tilde{Q}(x_t, a'_i) - V(x_t)\right)\nabla_{\phi(x_t)} \log f(a'_i|\phi(x_t))$

$k \leftarrow \nabla_{\phi_\theta(x_t)} D_{kl}\left[f(\cdot|\phi_{\theta_a}(x_t))||f(\cdot|\phi_\theta(x_t))\right]$

$d\theta \leftarrow d\theta + \frac{\partial \phi_\theta(x_k)}{\partial \theta} + \left(g_t^{acer} - \max\left\{0, \frac{k^T g_t^{acer} - \delta}{||k||_2^2}\right\} k\right)$

Accumulate gradients w.r.t. the value functions

$d\theta \leftarrow d\theta + \left(Q^{ret} - Q(x_t, a_t)\right)\nabla Q(x_t, a_t)$

$d\theta \leftarrow d\theta + \min\left\{1, \rho_i\right\}\left(Q^{ret} - Q(x_t, a_t)\right)\nabla V(x_t)$

Update retrace target

$Q^{ret} \leftarrow c\left(Q^{ret} - Q(x_i, a_i)\right) + V(x_i)$

Update average network

$\theta_a \leftarrow \alpha\theta_a + (1-\alpha)\theta$