FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	3D simulator for vision-based training of autonomous robots
Student:	Bc. Daniel Laube
Supervisor:	Ing. Zdeněk Buk, Ph.D.
Study Programme:	Informatics
Study Branch:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	Until the end of winter semester 2019/20

Instructions

Using the Unity game engine implement a 3D simulator for vision-based training of autonomous robots controllers. Primary focus on self-driving cars. Input to the controller will be a visual information from the camera. Implement a training algorithms and neural network-based controller. Create a communication protocol for connection between the simulator and the training algorithm. Test the system using experimental scenarios.

References

Will be provided by the supervisor.

Ing. Karel Klouda, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 19, 2018

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

3D simulator for vision-based training of autonomous robots

Bc. Daniel Laube

Department of Applied Mathematics
Supervisor: Ing. Zdeněk Buk, Ph.D.

May 7, 2018

Acknowledgements

I would like to thank Ing. Zdeněk Buk, Ph.D., my tutor, for all his advice, patience, and time he spent helping me finish this thesis. I would also like to thank Amy Safarik, M.A., B.Ed., B.A., an English teacher in Canada, for her prompt and helpful corrections of grammar, making the text much more readable. Many thanks belongs to my family and friends who supported me during my studies.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on May 7, 2018 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Laube, Daniel. *3D simulator for vision-based training of autonomous robots.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

Tato práce se zabývá návrhem a implementací prostředí vhodného pro učení neuronové sítě k ovládání robotů podobných autům. Neuronová síť se k prostředí může připojit pomocí protokolu TCP a tím pádem nelimituje implementaci neuronové sítě.

Druhou částí této práce je skript, který učí neuronovou síť zaparkovat auto na parkovací místo. Parkování probíhá na základě obrazu z kamery na autě. Tato část má sloužit jako důkaz použitelnosti implementovaného prostředí k učení neuronové sítě, jak ovládat autu podobného robota. Finální neuronová síť je rozdělena na dvě části, kde první část lokalizuje parkovací místo na obrazu z kamery. Výstup této sítě je pak zpracován plně rekurentní neuronovou sítí sloužící jako kontroler dávající povely autu.

Prostředí je implementováno v herním enginu Unity 3D a skript v Wolfram Mathematica.

Klíčová slova Autonomní auto, neuronová síť, konvoluce, rekurentní neuronová síť, Unity 3D, Wolfram Mathematica, TCP

Abstract

This work focuses on the design and implementation of an environment suitable for training a neural neural network how to control a robot similar to a car. A neural network can be connected to the environment via TCP protocol and, thus, does not limit the implementation of neural network.

The second part of this work is a script that teaches the neural network how to park a car using data from the car's cameras. This part serves as a evidence of the usability of the created environment for the purpose of teaching the neural network how to control a car-like robot. The final neural network consists of two parts, where the convolutional network localizes the parking spot from an image from a camera and output coordinates are fed to the recurrent neural network giving commands to the car.

The environment is implemented in the game engine Unity 3D and script in Wolfram Mathematica.

Keywords Autonomous car, neural network, convolution, recurrent neural network, Unity 3D, Wolfram Mathematica, TCP

Contents

List of Figures

List of Tables

Introduction

Self-driving cars

The first experiments with driverless cars were conducted in the 1920's, which can be read about in an article in the Milwaukee Sentinel from 8th December 1926[2]. The car was controlled remotely by a person in a second car.

The first experiments with a truly autonomous car or so called road following robot where conducted much later in 1980's by Carnegie-Mellon University[3] and Bundeswehr University Munich[4]. Autonomous cars have developed significantly since these experiments. Road following robots developed to cars that have a human driver present only as a safety precaution for unpredictable situation; autonomous cars are still in development phase and are not publicly available for purchase. A set of cameras and distance sensors are used for navigation, and data from sensors are input for a neural network which outputs commands for the car. Companies like Google or Uber are now conducting test in public traffic. In March 2018 in Arizona, the first casualty was lost as a result of such test, when Uber self driving car hit a pedestrian[5]. Self driving software is still not capable of full autonomy, and it will probably take years before fully autonomous cars will be available for the public. However, a couple autonomous features are available for the public, like autonomous parking or speed adjusting to the next car[1].

Ideal result

The ideal result of this work consists of two parts. The first part is a physical based simulator that simulates the actions of a robot, primarily car-like robot. This robot will have an optional number of wheels, where each wheel can be controlled separately. Each wheel can also have different attributes, like torque force, steering range, break force and size. The robot should be able to perceive its environment with cameras and distance sensors. These sensors will have a delay that can be set up by the user and will scan the surroundings

with the period set up by the user as well. This environment should not be limited to only one robot at a time, so the movement of multiple robots can be simulated. The environment should be able to simulate different kinds of terrains containing obstacles, a target and other robots. Communication with the environment should not limit which kind of programming language is used for the neural network.

The second part is a neural network proving valid functionality of the environment. The proof in the ideal case would be a neural network capable of parking a car-like robot into a parking space. Commands provided by the neural network should be based only on the images from the robot's camera and the distances provided by the distance sensors. The movement of the robot should resemble a car being parked by an actual human driver.

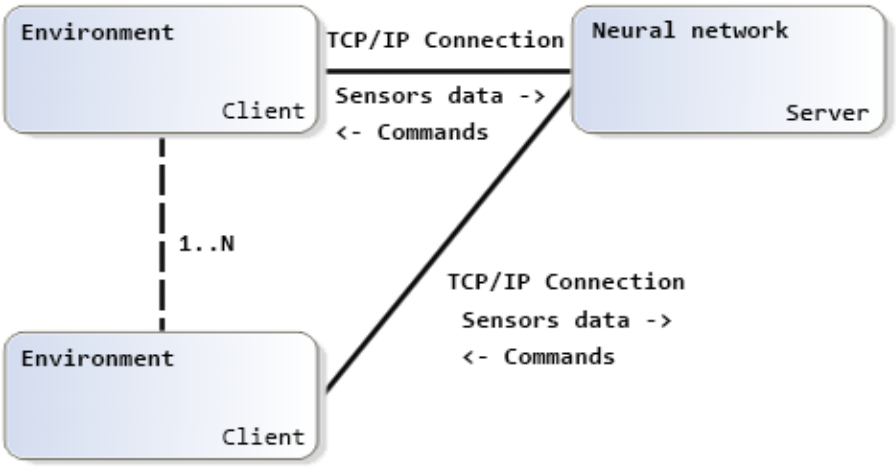

Figure 0.1: Ideal architecture of solution

Assignment analysis

1. Using the Unity game engine implement a 3D simulator for vision-based training of autonomous robots controllers. Primary focus on self-driving cars.
    - Unity is a component based game engine with support for all types of games. Among these games are racing simulators behaving according to physics. This is going to be very helpful for creating simulator with conditions similar to the real world.
2. Input to the controller will be a visual information from the camera. Implement a training algorithms and neural network-based controller.

- Training algorithms and controller are going to be implemented using some verified library or solution capable of running and teaching the neural network.

3. Create a communication protocol for connection between the simulator and the training algorithm.

   - Communication protocol should cover all necessary types of messages needed for training the neural network such as sending steering commands, requesting data from sensors and cameras or setting up the scene. This protocol should be as simple as possible so it does not slow down anything.

4. Test the system using experimental scenarios.

   - Scenarios are going to be focused on parking a car in a parking spot. Scenarios are going to be simple scene with a car and parking spot in different locations with different rotation. The neural network will navigate the car to the spot and should align the car in it.

Chapter 1

Research

This chapter is about exploring existing solutions of simulators for training neural networks and about tools and methods which can be used for implementation simulator.

- The first subchapter is about existing simulators.
- The next subchapter explores possibilities for the implementation of a simulator, focusing on game engines.
- The third subchapter looks at usable methods for creating a neural network.
- The fourth subchapter briefly explains how certain types of networks work.
- The last subchapter is about possible solutions for communication between simulator and neural network.

1.1 Existing simulators

1.1.1 Udacity

Udacity[6] created a simulator for a self-driving car using Unity and is now open source. The simulator is inspired by Nvidia experiments with self-driving cars. Nvidia conducted an experiment in which they attached three cameras to a real car; one camera pointed left, one right and one forward. They recorded 72 hours of driving for training data, collecting images from cameras and information about thrust, direction and brakes. This data was used for training the convolutional neural network. The Udacity simulator does the same thing as Nvidia did in their experiment, except that testing data is generated by driving in their simulator. Compared to the Nvidia experiment, only a few minutes of driving is required for training the network.

Figure 1.1: Screenshot of simulator made by Udacity

1.1.2 Carla

An interesting and advanced project is called Carla[7] (Car Learning to Act). Carla is not based on any game engine and was built from scratch. It focuses on simulations of urban locations with advanced graphics containing assets of many types of cars, buildings, people and even simulates weather.

Figure 1.2: Screenshot of Carla simulator

We did not want to use Udacity's simulator, because it is built for a different type of training than we would like to experiment with. Also Carla did not fit our needs since it is not free for possible commercial use. Both these environments limit the usable networks to certain types of frameworks

which are compatible with their platform; we want to create an environment independent of the neural network implementation as much as possible.

1.2 Methods for visual simulator

1.2.1 Game engines

Game engines, which usually have solution for 3D graphics, physics, GUI and much more, are a good option for building a visual simulator. For the purpose of this thesis, we considered three game engines:

1. Unity 3D[8]
    - User friendly engine, with a lot of implemented components. Free for commercial use up to $100.000 annual gross revenue. Large active community. Uses C# and Javascript for scripting. Supports around 30 platforms.
2. Unreal Engine 4[9]
    - Free for non-commercial use. Active community, creating tutorials and other useful content. User friendly. Uses C++. Supports Windows, iOS, Android, Playstation 4, Xbox One and more.
3. CryEngine 3[10]
    - Quite complex and difficult to learn the engine, with a smaller community compared to Unity 3D and Unreal Engine 4. Supports Windows, Playstation 4, XBox One and Oculus Rift.

1.3 Methods for neural networks

In this subchapter selected methods for creating the neural network are assessed. These methods include frameworks, libraries and whole solutions.

1.3.1 Frameworks

There are many good frameworks suitable for this kind of task, including the following three:

1. Tensorflow[11]
    - Open source software library for high performance numerical computation. Python, C++, Java and GO API. C++, Java and GO API are not covered by Tensorflow API stability promises.
2. Keras[12]

- High level neural networks API written in Python and capable of running on top of Tensorflow, CNTK or Theano.

3. Caffe2[13]

- Modular deep learning framework. C++ and Python API.

1.3.2 Whole solutions

1. Wolfram Mathematica

- Computational tool based on symbolic Wolfram language. Implements algebraic manipulation, visualisation, image processing, networking, neural networks and much more.

2. Matlab

- Wolfram Mathematica competitor. Similar functions like Wolfram Mathematica.

1.4 Types of neural networks

1.4.1 Feed forward network

A basic feed forward neural network consists of one or more layers of perceptrons. Each perceptron has one or more inputs and single output, which can be sent to multiple perceptrons in the next layer as their input. This structure is illustrated in the image 1.3.

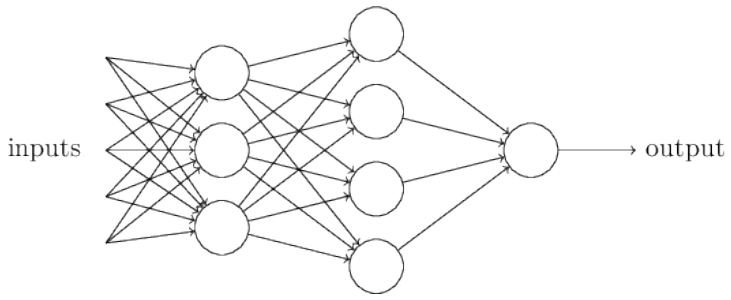

Figure 1.3: Illustration of structure of simple feed forward network[14]

Perceptrons usually have one more input called a bias, which has a value independent of input data and stays the same once the network has been trained and is specific for each perceptron.

The evaluation of the input data is done by layers of the network, where each neuron calculates its output from the input data, bias and weights. This is usually done using these formulas:

- $y(s) = \frac{1}{(1+e^{-s})}$
- $s = \sum_{i=0}^{n}(w_i * x_i) + b * w_b$
  - $y(s)$ – output value (logistic function) of perceptron
  - w_i – weight of input with index i
  - x_i – value of input with index i
  - b – value of bias of this neuron
  - w_b – weight of bias
  - n – number of inputs

From the formula it is clear that the output value of the neuron is from interval (0,1). Computers have limited precision which can cause the network to be unable to find the best values of weights and biases, because the logistic function $y(s)$ grows too fast.

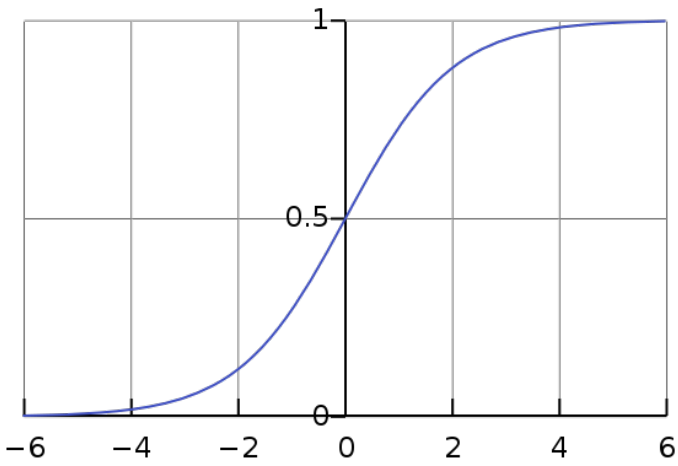

Figure 1.4: Graph of logistic function[15]

To overcome this issue, constant k is added, changing the function to this form.

$$y(s) = \frac{1}{(1 + e^{-s*k})}$$

The value of k is set experimentally. The other option is to use library specialized in unlimited precision for the price of higher computational requirements.

For training this type of network a backpropagation algorithm is used. The basic version of this algorithm takes input data, allows the network to

compute its output for this data, compares the output of the network to the correct output, and changes weights in the network so the network for this input gives the right output. Usually we want the network to give the correct output for more than one input, and these changes can be done based on summed errors from more inputs.

1.4.2 Convolutional networks

Convolutional neural networks[16] got their name from the convolutional operator, which is their basic operation to process data. The convolutional operator is also widely used in image processing for many purposes including blurring and sharpening an image, extracting edges, and many others. The main purpose of the convolutional operator in neural networks is to extract features from the input data. Whole convolutional networks usually consist of multiple layers of different kinds. This can be illustrated by the image 1.5.

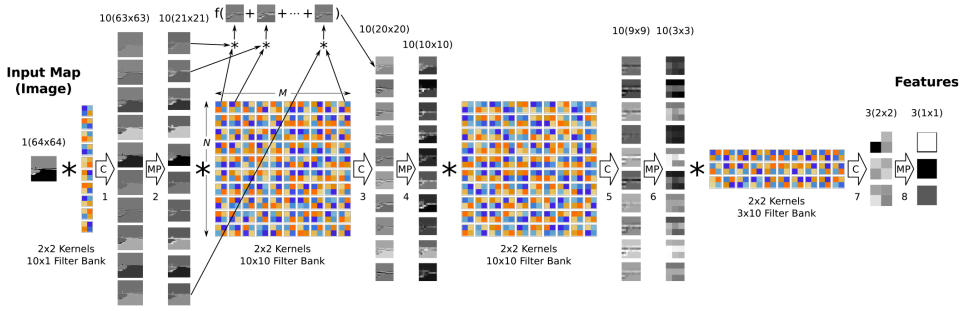

Figure 1.5: Example of a Convolutional Neural Network[17]

The convolutional operator takes a predefined matrix of values called the kernel. The kernel is then slide over the whole image with predefined step size called stride. In each step it takes the input data and the kernel and does the predefined operation with this data to get a new value. Table 1.1 shows example of a 2D convolution with an image size of 5 by 5 pixels and kernel size of 3 by 3 pixels. In each step is each field in kernel multiplied with one field from an image. The result of the step is a sum of these products.

The convolved feature image can then be fed to another layer or taken as final output of this network. Usually one convolutional layer uses more kernels and has multiple convolved feature images as output. This is because each kernel can extract different features. The number of used kernels is called depth. As the example 1.1 shows, the convolved feature image is smaller than the image fed to the convolutional operator. Sometimes it is useful to prevent this by using a technique called zero padding, where the convolved image is padded by zeros.

Table 1.1: Illustration of how convolutional operator works

<table>
<tr><td colspan="2">Image matrix</td><td colspan="2">Kernel</td></tr>
<tr><td colspan="2">

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

</td><td colspan="2">

1	0	1
0	1	0
1	0	1

</td></tr>
<tr><td colspan="2">Step 1</td><td colspan="2">Step 2</td></tr>
<tr><td colspan="2">

$1_{\times 1}$	$1_{\times 0}$	$1_{\times 1}$	0	0
$0_{\times 0}$	$1_{\times 1}$	$1_{\times 0}$	1	0
$0_{\times 1}$	$0_{\times 0}$	$1_{\times 1}$	1	1
0	0	1	1	0
0	1	1	0	0

4		

</td><td colspan="2">

1	$1_{\times 1}$	$1_{\times 0}$	$0_{\times 1}$	0
0	$1_{\times 0}$	$1_{\times 1}$	$1_{\times 0}$	0
0	$0_{\times 1}$	$1_{\times 0}$	$1_{\times 1}$	1
0	0	1	1	0
0	1	1	0	0

4	3	

</td></tr>
<tr><td colspan="2">Step 9</td><td colspan="2">Convolved feature image</td></tr>
<tr><td colspan="2">

1	1	1	0	0
0	1	1	1	0
0	0	$1_{\times 1}$	$1_{\times 0}$	$1_{\times 1}$
0	0	$1_{\times 0}$	$1_{\times 1}$	$0_{\times 0}$
0	1	$1_{\times 1}$	$0_{\times 0}$	$0_{\times 1}$

4	3	4
2	4	3
2	3	4

</td><td colspan="2">

4	3	4
2	4	3
2	3	4

</td></tr>
</table>

The convolutional layer is not the only layer type used in convolutional networks. As essential layers have proven ReLU (Rectified Linear Unit) and pooling layers. ReLU layers are used to introduce non-linearity. ReLU layer does elementwise operation, where each pixels value is modified by following the formula.

$$output = Max(0, input)$$

Pooling layers purpose is to reduce dimensionality of feature maps, which basically works similar to the convolutional layer. It slides a window of predefined size over the entire image with a predefined step. The step is usually the same size as a sliding window. In each step the specified operation to the data in the window is applied. These operations include taking the largest or smallest value as an output, or sum of these values. It can be illustrated by image 1.8.

The final neural network consist of combinations of these layers with different parameters. The last layer is usually layer of perceptrons with softmax

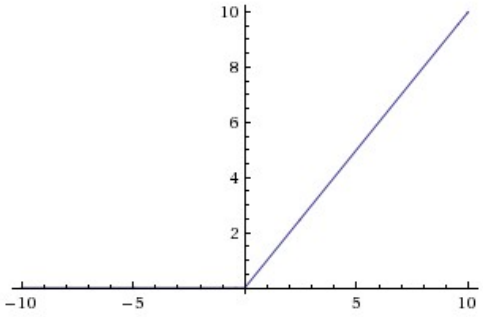

Figure 1.6: Graph of rectifier function

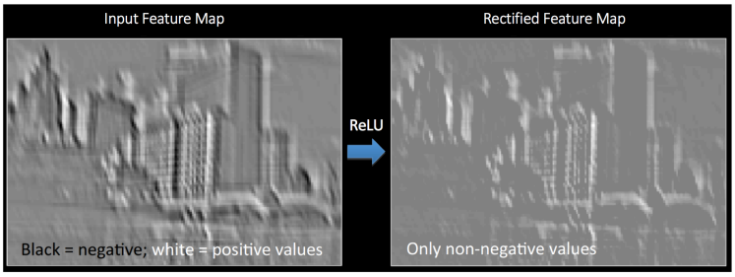

Figure 1.7: Example of image before and after ReLU layer[16]

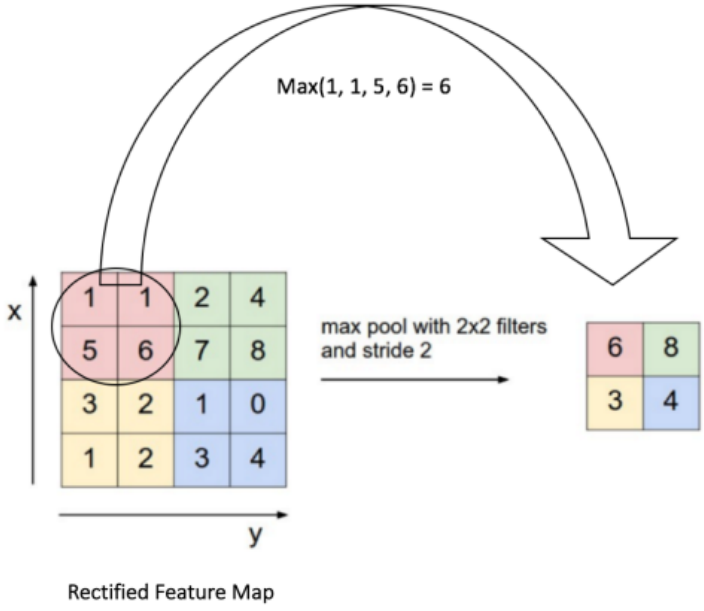

Figure 1.8: How max pooling works[16]

activation giving the final output of the network.

This type of network has proven to be very powerful for all sorts of purposes from locating objects, to recognizing persons or objects, to replacing objects on images and so on.

1.4.3 Recurrent networks

The main difference between feedforward and recurrent networks[18] is the way the outputs of neurons are handled. In feedforward networks, the output of each neuron is fed to the next layer. In the case of recurrent layers, the output of neurons is also fed in the next steps to one or more previous layers or the same layer depending on the network structure. This allows the network to have memory, and the output of the network is dependent not only on actual input data, but on all previous processed data. The image 1.9 illustrates how the structure of a recurrent network can look.

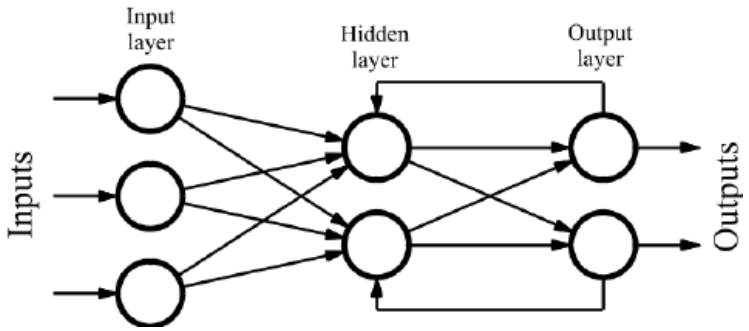

Figure 1.9: Example of recurrent network[19]

For tasks where events processed by the network in previous steps are important, it is necessary to have memory. This includes driving a car, generating text or music, and so on. For more complex tasks many different types of neurons that can replace the perceptron neuron have been developed. One of the widely used and well performing types is LSTM (Long-Short Term Memory) with a much more complex structure.

The fact that the output of the recurrent layer is dependent on the previous data makes their training complicated, and backpropagation will not work in its basic form. In order to teach a recurrent neural network, a modified version of this algorithms is used and is called backpropagation through time. This algorithm requires set of training data and correct outputs, which in some cases is not possible to obtain. One of these cases is learning how to control a robot. There are useful evolutionary algorithms for these cases.

1.4.4 Evolutionary algorithms

Evolutionary algorithms in general have a population of candidate solutions encoded in the genome. The population of candidates have their fitness function evaluated in each iteration. The fitness function gives the candidate a score, based on how well it performed. For the next iteration a new population of candidates is created based on the previous population. There are many heuristics how to select from the previous population and how to modify these candidates for further processing. These heuristics include selection operators, crossover operators and mutation operators.

The selection operator selects a subset of the tested population for crossover operator. The selection can be done in many ways, such as using a roulette selection or tournament selection.

- Roulette selection selects candidates with better fitness with higher probability.
- Tournament selection selects two random candidates and for the new population the candidate with better fitness is selected.

It is important to note that always selecting only the best performing candidates can lead to being stuck in local optimum.

The crossover operator then takes two candidates from the set selected by the selection operator and creates two new candidates by swapping parts of their genes. These two new candidates then became part of the new population. Crossover can be done on predefined or random points in the genome with certain probability.

The mutation operator then with certain probability modifies randomly selected genes of the new population to randomized values.

The fitness of the newly created population is evaluated again, and all the steps above are repeated. This is just a basic description of evolutionary algorithms, which can be extended with other heuristics, like island model (two separated populations, which crossover only at predefined points) or elitism (always adding the best performing candidate to the new population).

1.5 Methods for communication

There are two basic options to connect the game engine with the neural network: either use API or the neural network library in the game engine, or connect the engine to the neural network by a network socket.

1.5.1 Networking

Like with the game engines and neural network frameworks, there are many possibilities, but we will consider only the most widely used TCP/IP protocol.

Nevertheless this communication might seem similar to multiplayer games and point us to UDP protocol; however, we can not use UDP protocol because we need to be sure that all messages arrive intact. Some noise can be simulated later for the purpose of training neural network.

1.5.2 API

As a good example of API serves .NET/Link allowing calling Wolfram Mathematica functions from C# code. This .NET/Link could be used for combination of Unity 3D and Wolfram Mathematica, since Unity 3D uses C# for scripting.

Chapter 2

Analysis

This chapter assesses all the options mentioned in the previous chapter including methods for simulation, methods for creating neural network, and methods for connecting the simulator to a neural network.

2.1 Methods for simulation

The method for the simulation was specified in the assignment, based on experience with this platform.

2.2 Methods for neural network

The method for creating and training a neural network was selected based on our experience, as well as the simplicity of prototyping neural networks. The chosen platform is Wolfram Mathematica 11.2.

2.3 Methods for communication

2.3.1 Networking

Networking has one primary advantage compared to API its independence on implementation of other communicating side. A possible disadvantage is a lower communication speed, which could be crucial for vision based training, where image resolution might be affected as a result of maintaining a real time processing.

2.3.2 API

API, on the other hand should be able to maintain the maximum possible communication speed between the simulator and neural network, since the environment would focused only on one type of platform.

Chapter 3

Design

This chapter describes the design portion of the work, including the requirements of the final simulator, communication protocol, and the neural network.

1. The first subchapter lists functional, non functional requirements and use cases. The structure of the whole work is also described in this chapter.

2. The next subchapter describes protocol designed for communication between the environment and the neural network.

3. The last subchapter is dedicated to the neural network itself.

3.1 Requirements

3.1.1 Functional requirements

- Create an environment suitable for training the neural network to control an autonomous robot, primarily focusing on car-like robots.

- Create testing scenarios using environment interface - placement of robots, obstacles, terrain.

- Test functionality of the environment by using experimental scenarios for training the neural network.

- The robots distance sensors and cameras have to have an optional delay of output.

- Multiple robots can be controlled in the scene.

- Each wheel of the robot can receive instructions for steering - direction and torque.

- Request data from all sensors and cameras from the robot.
- Set IP and port on which will environment/neural network wait for connection.

3.1.2 Non-functional requirements

- 3D visualisation of the learning environment.
- Environment is implemented in Unity 5.
- The neural network is implemented in Wolfram Mathematica.
- Communication protocol uses TCP/IP sockets.
- The environment is able to run 30+fps with one robot with one camera.
- The neural network is able to receive 10+ fps from robots camera.

3.1.3 Use cases

- Train the neural network how to control a robot by using the created environment.

3.1.4 Server-client

We have decided to use TCP/IP for communication between Unity and Mathematica. Therefore it is necessary to decide which part will pose as the server and which as the client. The first idea was to make Mathematica the server and Unity the client for the possibility of letting Mathematica operate with multiple simulations at the same time and speeding up the training process.

This model was first tested with a simplified Python server without any neural network running. Python server accepted a message, waited for some time to simulate neural network processing accepted data and then it sent response. This model was working quite well, and Python server was able to communicate with around 6 instances of Unity at the same time, where each socket had its own thread and all this was happening with stable 30 fps on all Unity instances. The resolution in this experiment was set to 400 by 300 pixels.

Unfortunately it was not possible to implement this model with Mathematica. The reason for this were possibly unresolved bugs in Mathematicas functions serving to create and use a TCP server socket or documentation not explaining these functions well enough to use them properly. The reason for this is probably that these functions are still marked as experimental and most of them were added in a version used for this work that is Wolfram Mathematica 11.2. These possible bugs included:

- While all packets were received, the function did not return received data and blocked further communication.
- While reading from socket packet by packet, random errors occurred throwing exceptions. If this exception was processed, it was possible to continue reading from the socket and read the whole message without any missing bits. Yet Mathematica stopped the evaluation when this exception occurred.

These errors occurred while using following Mathematica functions and objects:

- `SocketReadMessage`
- `ReadByteArray`
- `ReadString`
- `SocketListener`

We tried to solve these errors and experimented with different combinations of Unity, Python and Mathematica. For both architectures, we managed to fix communication between Mathematica and Python and Unity and Python. However, for some unknown reason, when Python server was replaced by Mathematica, all problems returned.

All these errors led to change of the architecture, and Mathematica become the client and Unity the server. Unity environment maintained both options to pose as the client and as the server for the possibility of using of this environment with a different neural network or when the bugs are resolved in future versions.

After these problems were solved another arose; the speed of communication between Unity and Mathematica. During testing Unity to Python communication, sending Full HD images was done in a matter of hundredths of a second; when it came to Unity to Mathematica, the time of the transmission rose to a few seconds. For this reason the resolution was lowered to 160 by 100 pixels, and average time of transmission decreased to 0.05 seconds.

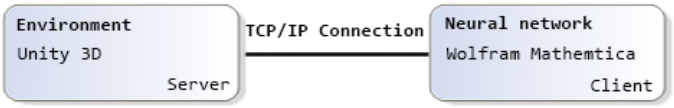

Figure 3.1: Graph of final architecture

3.2 Communication protocol

Communication protocol was designed to cover all necessary messages needed for training a neural network controlling a car-like robot. But since we are aware that it might not cover all future purposes of this environment, the ease of extendability of this protocol was taken in account. Basic form of each message is following:

`<messageLength><messageType><messageData>'\0'`

Messages that require more than confirmation needed have the same <messageType> for response to those messages to keep communication clearer.

Here follows a list of all implemented messages. If the response message is not specified, it is expected to receive the standard confirmation or the standard error message.

- Standard confirmation response message
  - `<messageLength>ok'\0'`
- Standard error message
  - `<messageLength>ko'\0'`
- Get image from camera
  - Request image from camera(s).
    - `<messageLength>cam<#cameras><id><id>...<id>'\0'`
  - Send image from camera(s).
    - `<messageLength>cam<#cameras><width><height><pixels>...<width><height><pixels>'\0'`
  - Order of images in response is expected to be the same as in request.
- Get distance from sensors
  - Request distance from sensor(s).
    - `<messageLength>dst<#sensors><id><id>...<id>'\0'`
  - Send distance from sensor(s).
    - `<messageLength>dst<distance><distance>...<distance>'\0'`
  - Order of distances in response is expected to be the same as in request.
- Set steering for wheel(s)
  - Set torque for chosen wheels.

        * `<messageLength>stw<carID><wheelId><torque>...<wheelId><torque>'\0'`

    - Set direction for chosen wheels.
        * `<messageLength>sdw<carID><wheelId><direction>...<wheelId><direction>'\0'`

- Prepare playground messages
    - Set size of flat playground.
        * `<messageLength>sps<carID><width><height>'\0'`
    - Set target position and spawn it, if it was not previously spawned.
        * `<messageLength>tgp<carID><x><y><z>'\0'`
    - Set target rotation and spawn it, if it was not previously spawned.
        * `<messageLength>tgr<carID><x><y><z>'\0'`
    - Set car starting position and spawn it, if it was not previously spawned.
        * `<messageLength>crp<carID><x><y><z>'\0'`
    - Set car starting rotation and spawn it, if it was not previously spawned.
        * `<messageLength>crr<carID><x><y><z>'\0'`

- Choose car type
    - Changes the car type of car with selected carID.
    - If this message is not sent during communication, environment uses default car type.
        * `<messageLength>car<carID><type>'\0'`

- Start session
    - First message sent by environment to neural network.
        * `<messageLength>str'\0'`

- End session
  - Last message sent by neural network to environment, which ends simulation on Unity side and makes environment go to main menu.
    - `<messageLength>str'\0'`

- Clear playground
  - Deletes all objects from scene loaded by LevelBuilder class.
  - Does not end simulation.
    - `<messageLength>clr'\0'`

- Capture whole state
  - Returns data from all cameras and images in order by their ids from selected robot.
  - Request:
    - `<messageLength>cpt<carID>'\0'`
  - Response:
    - `<messageLength>cpt<carID><#sensors>`
      `<distance>...<distance><#cameras>`
      `<width><heigth><pixels>...<pixels>'\0'`

- Get absolute position/rotation of car/target
  - Request of position and rotation of car.
    - `<messageLength>cpr<carID>'\0'`
  - Response:
    - `<messageLength>cpr<carID>`
      `<xCoord><yCoord><zCoord><xRot><yRot><zRot>'\0'`
  - Request of position and rotation of target
    - `<messageLength>tpr<carID>'\0'`
  - Response:
    - `<messageLength>tpr<carID>`
      `<xCoord><yCoord><zCoord><xRot><yRot><zRot>'\0'`

- Load predefined scenario by id
  - `<messageLength>scn<scenarioID>'\0'`
- Place predefined obstacle by id
  - `<messageLength>obs<carID><obstacleID>`
    `<xCoord><yCoord><zCoord><zCoord>`
    `<xRot><yRot><zRot><xSize><ySize><zSize>|`
- Continue simulation without communication for a specified number of stepes, before resuming communication
  - `<messageLength>stp<#steps>'\0'`

Each part of message has expected data type.

- All IDs (car, camera, sensor, obstacle, wheels, car type), playground size and pixels from camera are expected to be in 8 bit integer.
- All coordinates, distances from sensors are expected to be 32 bit floats (in terms of Mathematica reals).
- All message length and camera resolutions are expected to be 32 bit integers.
- Message type is expected to be 2 or 3 ASCII characters.

3.3 Neural network

This subchapter describes the types of networks used. The first subchapter is about convolutional networks, and the second is about recurrent networks. The last two chapters are about the final shape of the image processing network and controller network.

3.3.1 Image processing network

The image processing network has the very important task of locating the target and returning its distance and rotation from the car's perspective. We decided to use a 2D convolution network.

For this purpose we used a network that has proven its qualities already. We used YOLO[20] (You Only Look Once) network, which is network for real time object detection. The graph shows comparison of YOLO with other detectors. Comparison was done using COCO[21] dataset.

Since the network was originally trained to locate different objects than parking spots, we had to train the network from the beginning. For this

purpose we created a set of 10000 images taken from cars' camera. Images were saved in custom format.

- `<xDistance><yDistance><angle><imageData>`
  - xDistance, yDistance and angle are saved as a 4 byte float/real
  - Angle is in radians.
  - imageData are saved as a sequence of blocks `<r><g><b>` where each pixel is 1 byte long integer.

8000 images were used as a training set and the remaining 2000 as a validation set. Images had to be resized to 160 by 100 pixels before feeding them to the neural network. The output of the network are four numbers - xDistance, yDistance, $\sin(angle)$ and $\cos(angle)$. Each training iteration took around 20 minutes on Nvidia GeForce GTX 960M. This time could be lower, but all images did not fit in the memory, so they had to be loaded from hard drive many times. After a couple tens of iterations the network was achieving acceptable performance and was ready to use. The achieved average distance precision was around 1/12 of a parking spot length, and the average rotation error was under 5 degrees. This was achieved using default settings or Mathematicas function `NetTrain`.

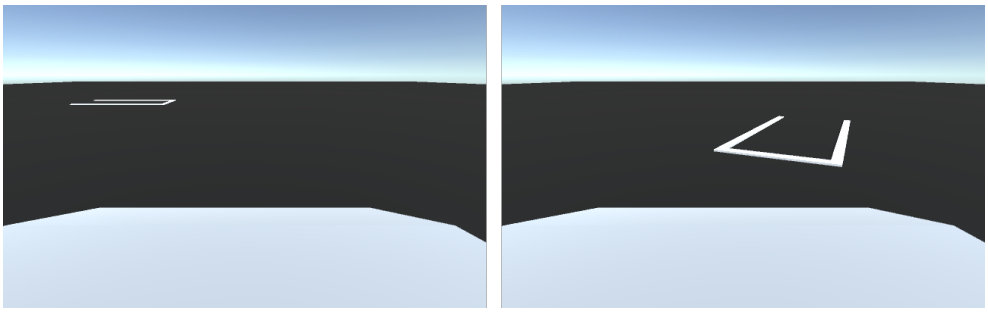

Figure 3.2: Images from cars camera

Chapter 4

Implementation

4.1 Implementation details

4.1.1 Unity section

Two scenes, one for the main menu and the second for simulation, were implemented in Unity 3D. The main menu serves for setting up IP, port, the role of the environment (client/server); this information is passed to the simulation scene in instance of class EnvironmentInterface. For the simulation the following classes were implemented:

- Initializer
- EnvironmentInterface
- LevelBuilder
- Scenario
- CarInterface
- Steering
- WheelController
- Sensor
- CameraInterface

The image 4.1 shows the directions in which messages are processed and commands executed. Mentioned classes are described in the next few paragraphs.

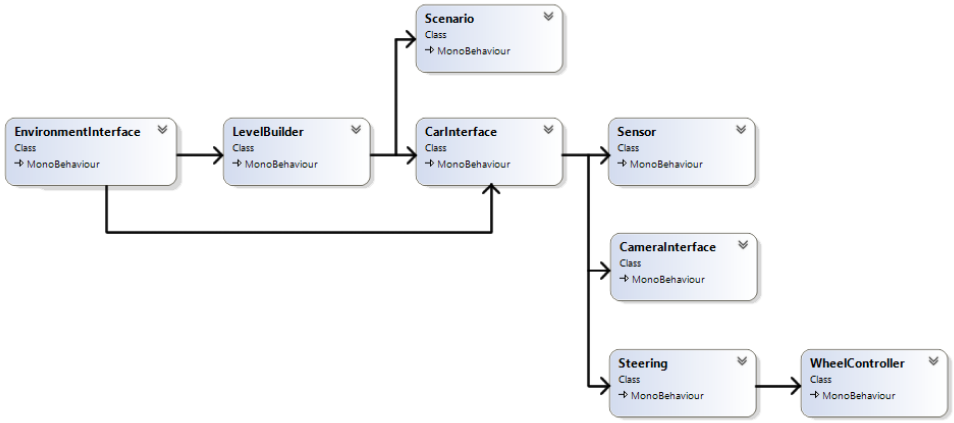

Figure 4.1: Graph illustrating how message is spread in simulator

4.1.1.1 Initializer

The initializer class should be the only class with an implemented Start method among objects saved in scene. This is because Unity does not the guarantee the order by which `Start` method will be called on all objects. Therefore this class should create all other objects needed at the beginning of the simulation, or call their `Init` functions. This way of using uninitialized objects should be prevented. In the current state it initializes only an instance of EnvironmentInterface class, but previous experience with Unity says that this is a good practice.

4.1.1.2 EnvironmentInterface

This is the most important class in the project, which serves as an interface between Environment and neural network. This class creates and manages TCP socket.

Init function finds instance of `LevelBuilder` class; if this object is not present, an exception is thrown. Then follows call of the function `makeConnection`. The function `makeConnection` creates an appropriate type of TCP socket. When the connection is made `SayHelloToServer` function is called and the communication begins.

Coroutine function `receiveMessage` is responsible for accepting the incoming message. This function assembles the whole message from incoming packets and then passes the whole message to `processMessage`.

Function `processMessage` takes the message type (fourth to sixth byte) and calls the appropriate function to process this message. If the message type is not recognized, a standard error message is sent to the neural network. In most cases, these functions just parse incoming data and pass it

to `LevelBuilder` or `CarInterface`. Here is a list of functions called from `processMessage` and explanation of how they work:

- `sendImageFromCamera`
  - Is a coroutine function. First it parses a message, then prepares array for a new message with the exact size. Counts maximum delay on all requested cameras. Sends requests for image from all requested cameras using function of `CarInterface` `getCurrentImageFromCamera`, which takes three parameters
    - `int cameraID`
    - `byte[] message` - array to which the image will be stored
    - `int index` - index at which should the image start
  - After this request, function waits for the maximum delay and calls `sendMessge` function.
- `sendDistanceFromSensor`
  - Is a coroutine function. First it parses a message, then prepares array for a new message with the exact size. Counts maximum delay on all requested sensors. Sends requests for distances from all requested sensors using function of `CarInterface` `getDistanceFromSensor`, which takes three parameters
    - `int sensorID`
    - `byte[] message` - array to which the distance will be stored
    - `int index` - index at which should the image start
  - After this request, function waits for the maximum delay and calls `sendMessge` function
- `captureWholePlaygroundState`
  - Is a coroutine function. First it parses a message, then prepares array for a new message with the exact size. Counts maximum delay on all requested sensors and cameras. Sends requests for distances and images from all requested sensors and cameras using functions of `CarInterface` `getDistanceFromSensor` and `getCurrentImageFromCamera`. After this request, function waits for the maximum delay and calls `sendMessge` function
- `sendPositionAndRotation`
  - Based on first parameter `char type` this function creates message containing position and rotation of car `'c'` or target `'t'`.
- `waitUpdates`

  - Continues simulation without communication for a specified number of steppes, before resuming communication. One step is equal to one `FixedUpdate` in Unity; in this project set to 0.01 seconds.

- `endSession`
  - Closes TCP socket and loads main menu.

Functions mentioned in this list only parse message and call `LevelBuilder` function with received parameters.

- `setPlaygroundSize`
- `setTargetPosition`
- `setTargetRotation`
- `setCarPosition`
- `setCarRotation`
- `clearPlayground`
- `setSteeringTorque`
- `setSteeringDirection`
- `setCarType`
- `loadScenario`

4.1.1.3 LevelBuilder

`LevelBuilder` is class responsible for instantiating and destroying terrain, target, cars and obstacles. `LevelBuilder` also keeps references to all these objects. All functions that modify the car, target or obstacles always make sure that the modified object is spawned; if not they spawn it first with default parameters and modified only a given parameter. For example if a car is not spawned and the function `setCarPosition` is called, it will be spawned with default rotation and default car type on location given to function `setCarPosition`.

Whenever function `loadScenario` is called, it destroys all objects using function `clearPlayground` and then spawns new ones based on the information contained in instance of class `Scenario` with given ID.

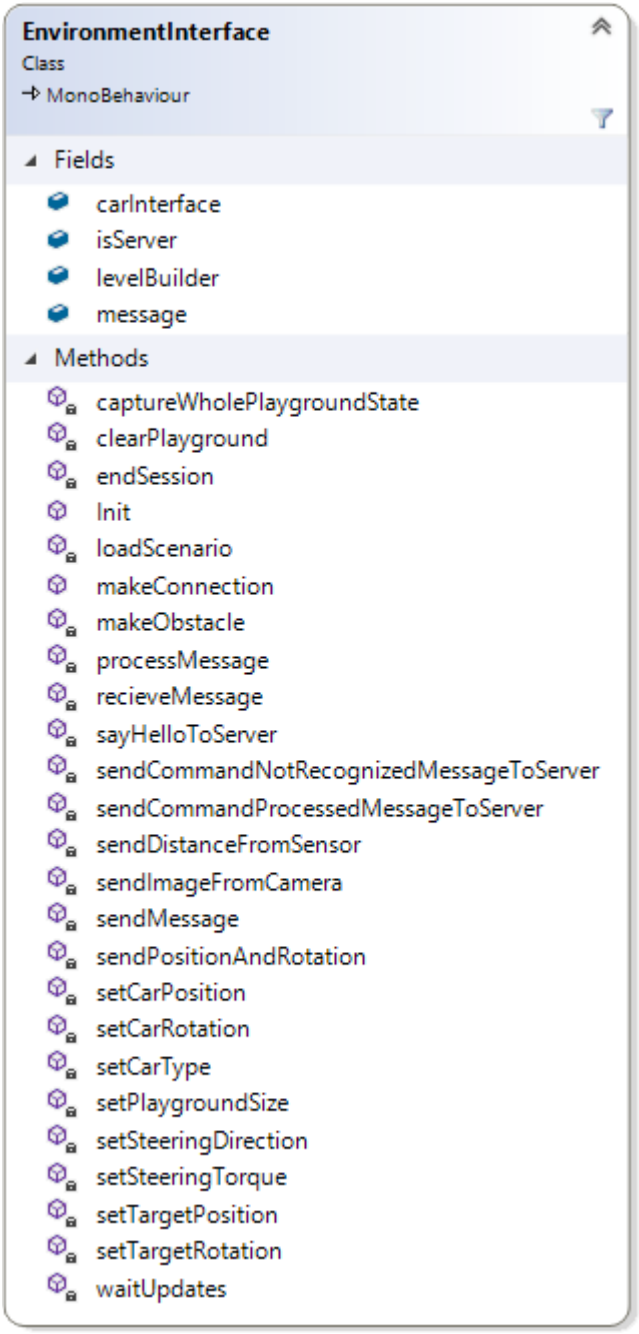

Figure 4.2: EnvironmentInterface - important variables and functions.

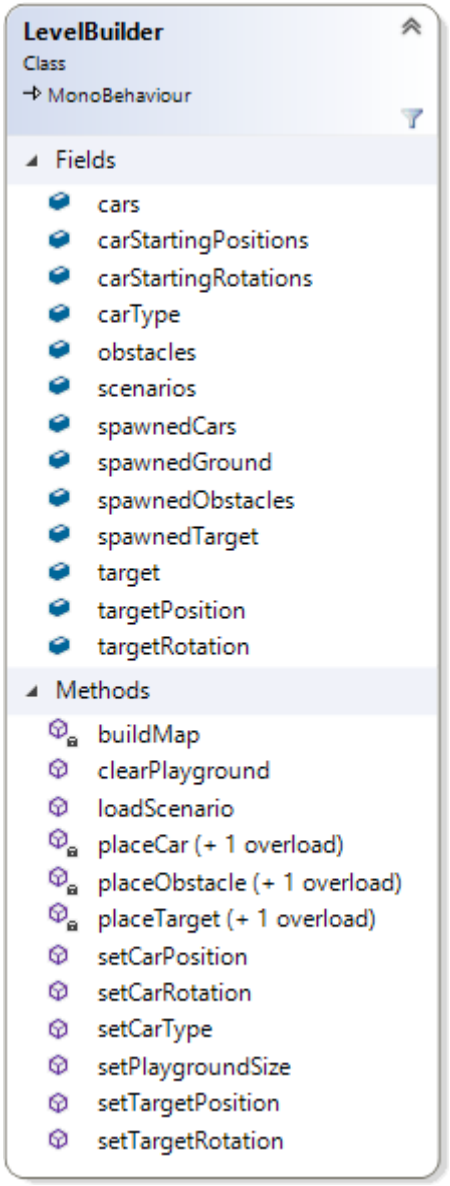

Figure 4.3: LevelBuilder - important variables and functions.

4.1.1.4 Scenario

Serves for storing information about a predefined scenario. Each predefined scenario is a saved instance of this class as a Unity asset. `Scenario` holds information about position, rotation and type of cars, obstacles and targets, as well as terrain size.

4.1.1.5 CarInterface

The main purpose of this class is to serve as an interface between a car's sensors, cameras and wheels and `EnvironmentInterface`. Also calls `Init` function on all car components.

The basic structure of a car robot should model the graph 4.4.

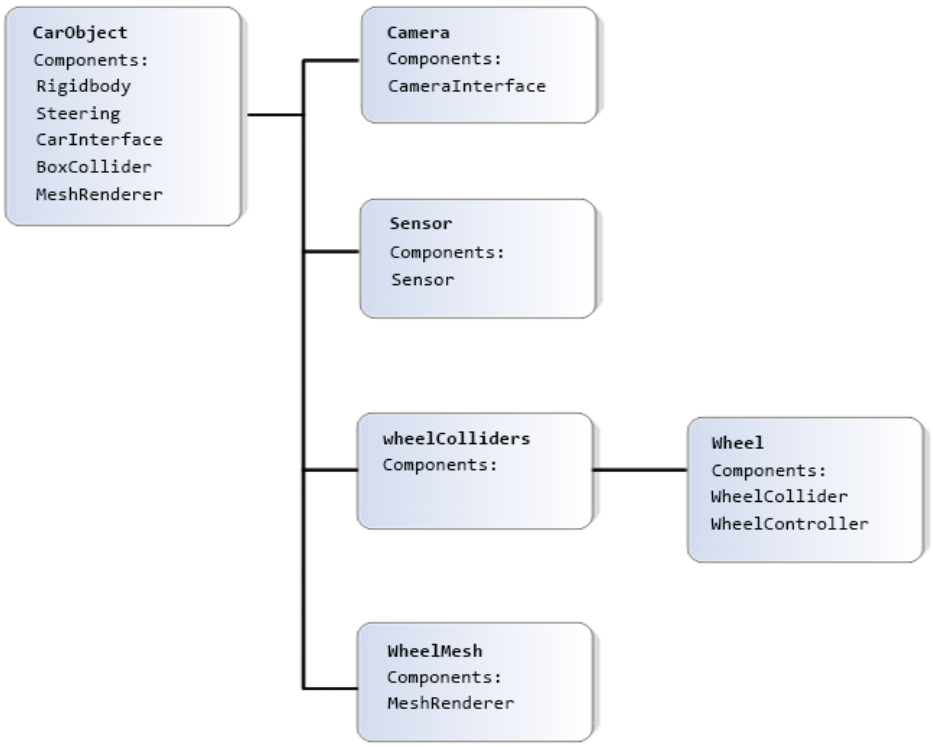

Figure 4.4: Graph of structure of car robot

4.1.1.6 Steering

Serves as an interface between the car and its wheels. Holds references to all wheels and gives them commands. These commands are then evaluated by

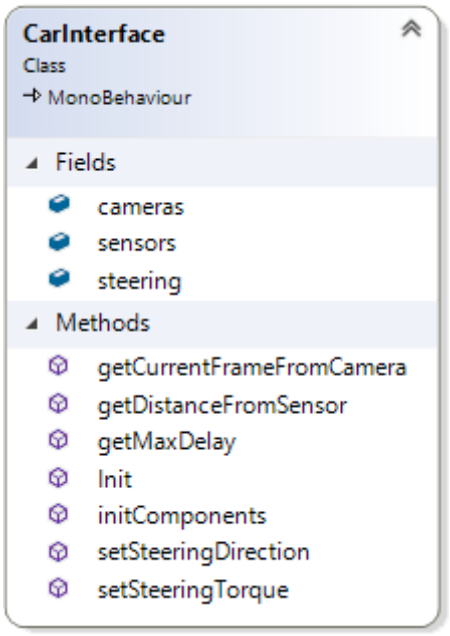

Figure 4.5: CarInterface - important variables and functions.

`WheelController` and corrected (if the value of direction or torque are out of range).

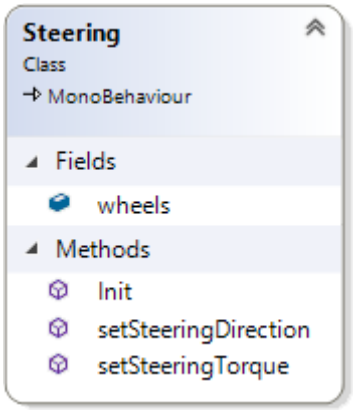

Figure 4.6: Steering - important variables and functions.

4.1.1.7 WheelController

This class makes the car actually move by setting parameters to instances of Unity class `WheelCollider`. Parameters of this class set the wheels abilities.

Abilities include the maximum torque and steering direction or the ability to break.

- `bool isStatic`
  - Wheel can not change direction.
- `bool noTorque`
  - Wheel can not apply acceleration.
- `bool noBreak`
  - Wheel can not apply break force.
- `float steeringForce`, `accelerationForce` and `breakForce`
  - Properties defining how the wheel will behave
- `float steering`
  - Final `steeringAngle` of `WheelCollider` is defined as `steering*steeringForce`
  - Parameter `steering` should be from interval $< -steeringRange; steeringRange >$, if not `WheelCollider` is going to set it to the closest number from this range.
- `float acceleration`
  - Parameter set up by the neural network. Final `motorTorque` of `WheelCollider` is defined as `acceleration*accelerationForce`.
  - Parameter `acceleration` should be from interval $< -1, 1 >$, if not `WheelCollider` is going to set it to the closest number from this range.
- `float breakForce`
  - Break force is applied if two conditions are met. First wheel has to have `noBreak` parameter set to `false`. Second absolute value of `acceleration` is lower than `breakThreshold`.

4.1.1.8 Sensor

With set period stores measured value to buffer. Buffer is an array using modulo indexing. When the `getDistance` function is called from `CarInterface`, it starts coroutine `saveData`, which waits for number of `FixedUpdates` specified in `delay` parameter. After waiting, data from buffer at position actual index minus `sensorDelay` are stored in passed array.

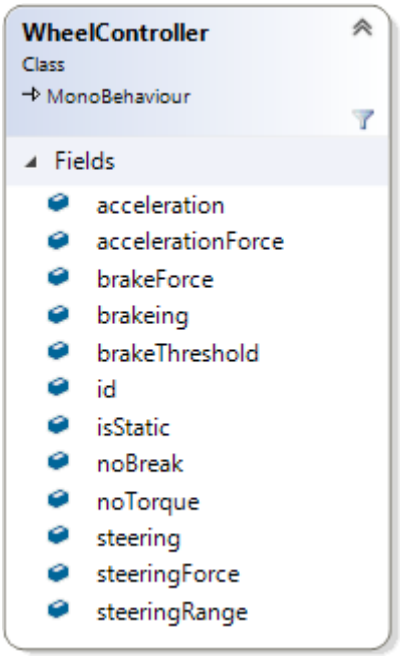

Figure 4.7: WheelController - important variables and functions.

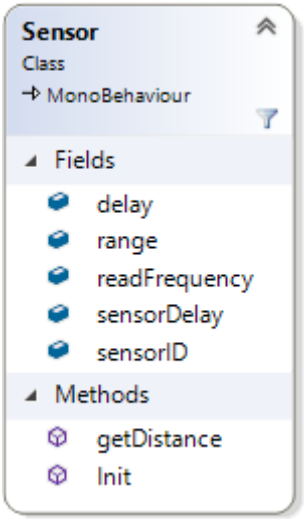

Figure 4.8: Sensor - important variables and functions.

4.1.1.9 CameraInterface

With set period stores image from camera to buffer. Buffer is an array using modulo indexing. When the `getCurrentFrame` function is called from `CarInterface`, it starts coroutine `saveData`, which waits for number of `FixedUpdates` specified in `delay` parameter. After waiting, data from buffer at position actual index minus `sensorDelay` are stored in passed array on specified index together with resolution.

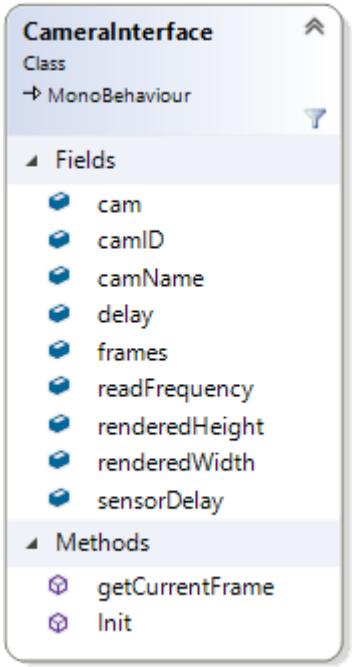

Figure 4.9: CameraInterface - important variables and functions.

4.1.2 Mathematica section

Mathematica was used to create the neural network for experiments and interface between this neural network and simulator. The results are two notebooks, one for training the neural network how to locate a parking spot and the second for training the controller, which takes coordinates from the first network and gives commands to the car.

4.1.2.1 Localization network

The notebook containing code for training the localization network is called localization.nb. It contains functions to load images in batches and to train and test the network. The script takes the file with names of images meant for

training. These images should be generated by the environment, or by other means which will keep the expected format.

- `<xDistance><yDistance><angle><imageData>`
  - `xDistance`, `yDistance` and `angle` are saved as 4 byte float/real
  - `angle` is in radians
  - `imageData` are saved as a sequence of blocks `<r><g><b>` where each pixel is 1 byte integer.

  Images are loaded in batches with size determined by user. For training purposes there are two main functions.

- `trainNet[batchSize_, batchCount_, network_, imagesFilesCount_, namesFile_]`
  - The purpose of this function is to train a network capable of giving location and rotation of parking spot.
  - `batchSize`
    - How many images should be loaded in one batch.
    - Reason for this parameter is that Mathematica function `NetTrain` tends to fail if too large batch of learning data is given to it.
  - `batchCount`
    - How many batches should be loaded at the same time.
    - This parameters purpose is to have a possibility to control RAM consumption.
  - `network`
    - `network` should be instance of `NetChain`.
  - `imagesFilesCount`
    - How many images should be used to train the network in total.
  - `namesFile`
    - Parameter with absolute path to the file containing names of images used for training the neural network. Images should be in the same folder as the file containing their names

- `testNet[batchOffset_, batchSize_, batchCount_, network_, imagesFilesCount_, namesFile_]`
  - The purpose of this function is to test trained network on different set of images from training images.

- Parameters of this function have the same meaning as in trainNet function
- `batchOffset`
  - Is the index of the first image that should be used. If the file for testing images is different than the file with learning images, it should be equal to one.
- `namesFile`
  - Parameter with absolute path to the file containing names of images used for testing the neural network. Images should be in the same folder as the file containing their names.

The trained network is then exported to .wl file, which can be loaded from any other Mathematicas notebook.

Notebook contains the definition of YOLO network, which can be modified for different resolutions, different outputs for future experiments.

```
YOLO = NetInitialize@
NetChain[{ElementwiseLayer[2.*# - 1. &],
  ConvolutionLayer[16, 3, "PaddingSize" -> 1],
  leayReLU[0.1],
  PoolingLayer[2, "Stride" -> 2],
  ConvolutionLayer[32, 3, "PaddingSize" -> 1],
  leayReLU[0.1],
  PoolingLayer[2, "Stride" -> 2],
  ConvolutionLayer[64, 3, "PaddingSize" -> 1],
  leayReLU[0.1],
  PoolingLayer[2, "Stride" -> 2],
  ConvolutionLayer[128, 3, "PaddingSize" -> 1],
  leayReLU[0.1],
  PoolingLayer[2, "Stride" -> 2],
  ConvolutionLayer[256, 3, "PaddingSize" -> 1],
  leayReLU[0.1],
  PoolingLayer[2, "Stride" -> 2],
  ConvolutionLayer[512, 3, "PaddingSize" -> 1],
  leayReLU[0.1],
  PoolingLayer[2, "Stride" -> 2],
  ConvolutionLayer[1024, 3, "PaddingSize" -> 1],
  leayReLU[0.1],
  ConvolutionLayer[1024, 3, "PaddingSize" -> 1],
  leayReLU[0.1],
  ConvolutionLayer[1024, 3, "PaddingSize" -> 1],
  leayReLU[0.1],
  FlattenLayer[], LinearLayer[256], LinearLayer[128],
```

```
 leayReLU[0.1], LinearLayer[4]},
"Input" -> NetEncoder[{"Image", {160, 100}}]];
```

4.1.3 Controller network

For the purpose of training a controller network notebook client.nb was created, which uses 4 other .wl files to keep the code better structured.

- messageProcessing.wl
  - Imports messageParsing.wl and messageCreating.wl.
  - Contains only one function `processMessage[msg_]`, which based on the message type decides how the message should be processed.
- messageParsing.wl
  - Contains functions for extracting data from incoming messages and giving this data format that is better for further processing by Mathematica.
- messageCreating.wl
  - Contains functions that generate outgoing messages to simulator from Mathematicas objects and data types.
- fitness.wl
  - Contains functions with three purposes.
    - To evaluate the fitness of candidate solution.
      - By default top 10 candidates are saved to variable `top`
    - To save the statistics of the fitness progress.
      - Fitness of all tested candidates is saved in a `List` called `fitnessG` where each element of the array contains `List` of fitness values of one population. A graph of best fitness over the populations can be generated using the following command:

        ```
        DiscretePlot[Min[fitnessG[x]],
        {x, 1, Length[fitnessG]}];
        ```

    - To save best performing candidate solutions from the whole learning process, which can then be used as initial population for further learning.

Notebook client.nb uses functions from files above and contains couple functions itself. These functions serve several purposes.

- Loading localization network.
- Running networks without training.
- Creating TCP connection to simulator.
- Creating and training the recurrent neural network.
  - `startLearningSession[imgProcessNetwork_, controllerNetwork_, roundsPerScenario_, scenariosIDs_, initPoints_]`
    - This functions purpose is to evolve weights for controller network.
    - `imgProcessNetwork`
      - Instance of `NetChain`.
    - `controllerNetwork`
      - Strucutre of a recurrent layer is stored in a 2 dimensional `List`.
      - `{{neurons_layer1,inputs_layer1,},...,{neurons_layern,inputs_layern}}`
    - `roundsPerScenario`
      - How many seconds should the network have for a single tested scenario.
      - It can be easily switched to the number of steering commands given to a car.
    - `scenariosIDs`
      - Which scenarios should be used for training the recurrent network.
    - `initPoints`
      - Set of initial weights used for `NMinimize`.
      - If it is equal to an empty `List`, random candidates are used for the first round.
  - `startSession[imgProcessNetwork_, controllerNetwork_, roundsPerScenario_, scenariosIDs_, initPoints_]`
    - This function serves for demonstration of controller.
    - Parameters have almost identical meaning as in case of `startLearningSession`.

Except for the functions, some important parameters are defined in this notebook such as `ip`, `port`, `popSize` (size of evolved population) and `iterations` (number of iterations of evolution).

Training was realized by using Mathematicas function `NMinimize`, which takes a function and tries to find its global minimum. In this case the function was fitness function of candidate solutions. Differential evolution was set as a method used to minimize fitness function.

4.1.3.1 Differential evolution

Differential evolution[22] is one of many versions of evolutionary algorithms. It has specific forms of selection, crossover and mutation operator.

- Selection operator chooses one candidate solution from unmarked candidates. Chosen candidate is marked.
- Three more candidates are selected.
- Differential vector is gained by subtracting first candidate from the second candidate from the second step.
- Differential vector is multiplied by mutation constant and this vector is added to the third candidate from the second step. This vector is called noise vector.
- New candidate solution is gained by going through all genes in genome, where in each step a random number is generated. If this number is lower than crossover constant, a gene from noise vector is used for the new candidate. If this number is bigger than the crossover constant, a gene from a marked candidate is used.
- Fitness function is evaluated for the newly created candidate. Fitness is compared to the fitness of marked candidate. For the next population a candidate with better fitness is used.
- All these steps are applied until all candidate solutions from the previous iteration are marked.
- After applying these operators, fitness is evaluated and process starts again.

Chapter 5

Experiments

5.1 Results of experiments

All experiments were conducted using 13 predefined scenarios with a single four wheel car robot. The target parking spot was always visible from the car's only camera and placed at different distances with different rotations. Examples of scenarios are visualized in the schemas 5.1 and 5.2.

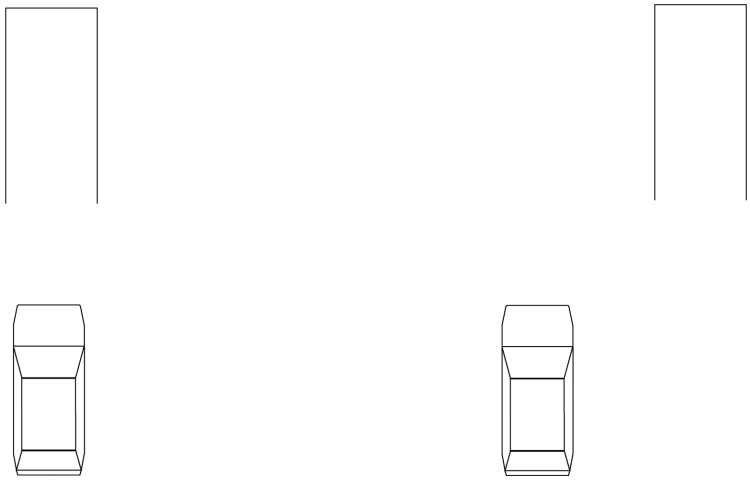

Figure 5.1: Two example scenarios

Experiments were very time consuming and running a simulation of 100 iterations with populaiton of 10, where each candidate had 10 seconds per scenario took around 20 hours.

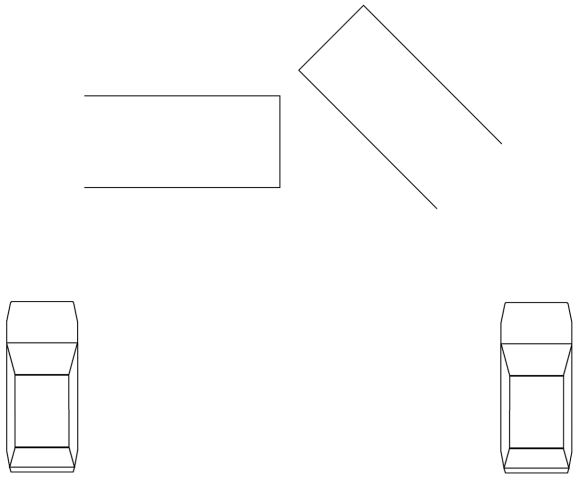

Figure 5.2: Two example scenarios

5.1.1 Experiment 1

The first experiment served as a test of the environment and training script to search for possible bugs preventing the training of the car. The car in this experiment had 5 seconds to park in the parking spot. Fitness was defined only as a distance from the parking spot at the end of simulation. The car started from 10 to 20 units away from the parking spot. After 100 iterations of differential evolution with population of 10, the car was able to get as close as 2.5 units from the parking spot. As a controller was used fully recurrent layer with 8 neurons connected to fully recurrent layer with 2 neurons.

5.1.2 Experiment 2

After reviewing the car's behavior from the first experiment, it was clear that the car was approaching the parking spot just fast enough to be close enough to the parking spot at the end of the simulation and would continue driving after reaching the parking spot. Because of this, we decided to prolong the time of simulation to 10 seconds. Fitness function was also changed to consider the car's alignment in the parking spot.

- $F(x) = d + (d + 1) * (\alpha/180)$
  - $F(x)$ – fitness function
  - x – candidate solution
  - d – distance from parking spot
  - α – rotation difference between a car and a parking spot

After 100 iterations with a population of 10, the car was starting to align with the parking spot, but still was not stopping. Another problem emerged; the car was changing directions between two consecutive commands very often and was shaking. As a controller was used fully recurrent layer with 8 neurons connected to fully recurrent layer with 2 neurons.

Table 5.1: Results of best performing candidate solution in experiment 2

Fitness ($F(x)$)	Distance (d)	Rotation difference (α)
7.402	6.303	42.913

5.1.3 Experiment 3

The car in the previous experiment was still not stopping in the parking spot, only driving slower. We also noticed that car was changing directions a lot which does not resemble a human driver at all. These observations led to the additional change of the fitness function.

- $F(x) = d + (1 + d) * (/180) + (1 + d) * t + (ch/c)$
  - $F(x)$ – fitness function
  - x – candidate solution
  - d – distance from parking spot
  - α – rotation difference between car and parking spot
  - t – torque of car at the end of simulation
  - ch – number of changes of directions between two consecutive commands
  - c – total number of commands sent to car

After another 100 iterations, the car was not behaving better compared to the previous experiment using the same controller network structure.

Table 5.2: Results of best performing candidate solution in experiment 3

Fitness ($F(x)$)	Distance (d)	Rotation difference (α)	Direction change ratio (ch/c)	Final torque (t)
5.062	4.067	39.574	0.036	0.519

5.1.4 Experiment 4

In this experiment, we decided to use L2 norm for the fitness function instead of L1 norm used so far. Changing the fitness function to the following form.

- $F(x) = \sqrt{d^2 + ((1+d) * (/180))^2 + ((1+d) * t)^2 + (ch/c)^2}$
  - $F(x)$ – fitness function
  - x – candidate solution
  - d – distance from parking spot
  - α – rotation difference between car and parking spot
  - t – torque of car at the end of simulation
  - ch – number of changes of directions between two consecutive commands
  - c – total number of commands sent to car

After around 300 iterations with a population of 10, the shaking improved significantly, even if the direction change ratio stayed almost the same, but the distance from the parking spot and the alignment of the car was still quite similar.

Table 5.3: Results of best performing candidate solution in experiment 4

Fitness ($F(x)$)	Distance (d)	Rotation difference (α)	Direction change ratio (ch/c)	Final torque (t)
4.753	3.338	36.875	0.035	0.892

5.1.5 Experiment 5

Observations from previous experiments led to the change of the structure of the controller network to one fully recurrent layer of 4 neurons connected to fully recurrent layer of 4 neurons connected to fully recurrent layer of 2 neurons. Results of this experiment after 100 iterations with population of 10 were similar to Experiment 4 5.1.4 after the same amount of iterations.

Table 5.4: Results of best performing candidate solution in experiment 5

Fitness ($F(x)$)	Distance (d)	Rotation difference (α)	Direction change ratio (ch/c)	Final torque (t)
5.244	4.690	31.773	0.035	0.302

5.1.6 Experiment 6

The last experiment was conducted using the same fitness from Experiments 4 5.1.4 and 5 5.1.5, but changing the size of population to 50. The number of iterations was lowered to 50, since 100 iterations with population of this size would took around 200 hours to compute.

Table 5.5: Results of best performing candidate solution in experiment 5

Fitness ($F(x)$)	Distance (d)	Rotation difference (α)	Direction change ratio (ch/c)	Final torque (t)
4.305	3.722	47.530	0.036	0.238

Conclusion

Assignment assessment

Pros

We managed to create a working environment for training neural networks how to control car-like robots with most of the features we wanted. The environment is able to communicate with any neural network over TCP socket, which makes it possible to create a neural network using almost any platform or programming language. Limits on the resolution of a camera are quite high, since experiments with python proved sending images in Full HD as possible while maintaining stable 30fps with one camera. Full HD is more than enough for the intended purposes of this environment.

The designed communication protocol covers all needs for learning controller to control a car-like robot and is easily extendable. Protocol covers the exchange of data from sensors, images from cameras, sending commands to the robot and setting up a custom training ground.

Cons

We did not have enough time to do enough experiments with a controller network to achieve a controller able to resemble a human driver. Its accuracy was not as good as we hoped for, and often a change of direction was not evaded as well. I believe that more iterations or a more complex controller network would help improve controllers capabilities.

Wolfram mathematica TCP socket did not allow us to run simulations faster than in real time by which times of experiments grew quite high and limited number of iterations.

Assignment completion

- Using the Unity game engine implement a 3D simulator for vision-based training of autonomous robots controllers. Primary focus on self-driving

cars.

  - Chosen parts of implementation are described in chapter Implementation - Unity section 4.1.1. Full implementation is on the attached disk. Even though the implementation still has a lot of space for extensions and improvements, its usability was proven and this part of assignment can be considered completed.

- Input to the controller will be visual information from the camera. Implement a training algorithms and a neural network-based controller.
  - The final environment fully supports the transmission of images of various resolutions with user defined delay, simulating delay on real circuits.
  - Implementation of training algorithms is described in chapter Implementation - Mathematica section 4.1.2 and chapter Design - Neural network 3.3.
  - We managed to train the controller aiming to the target parking spot; nevertheless, the controller has a lot of room for improvement, and this part of assignment can be considered completed.

- Create a communication protocol for connection between the simulator and the training algorithm.
  - Communication protocol is described in chapter Design - Communication protocol 3.2. Designed protocol covered all the needs for training our controller and by that this part of assignment can be considered completed.

- Test the system using experimental scenarios.
  - We created 13 different scenarios for experiments and conducted 6 different experiments with controller networks. More details about experiments are in chapter Experiments 5. This part can be considered as completed as well.

Possible improvements

The training of the controller could be possibly improved by adding obstacles to the scene and giveing their location to the controller as input. This could give controller some sense of space and allow more precise parking, when the parking spot is not visible. This would require creating a new localization network capable of locating these obstacles.

Training could be also improved if we would manage to make Mathematicas TCP socket transmit data faster and would allow faster simulations. Alternatively, we could use a different platform for prototyping neural networks like Keras or TensorFlow which support Python.

Possible extensions

Possible extensions include adding more scenarios, different types of robots, and better graphical visualisation. Implementations of message parsers for different programming languages and platforms. Extensions also include implementation of learning scripts using threading and by that allowing testing multiple candidates at the same time.

Bibliography

[1] History of autonomous cars: *Wikipedia: The Free Encylopedia* [online]. Wikimedia Foundation, 2003. Page last edited 4. 4. 2018 v 04:11. [seen 2018-04-30]. Available at: `https://en.wikipedia.org/wiki/History_of_autonomous_cars`.

[2] Milwaukee Sentinel: *Phantom auto* [online]. Milwaukee Sentinel, 8. 12. 1926. [seen 2018-04-30]. Available at: `https://news.google.com/newspapers?id=unBQAAAAIBAJ&sjid=QQ8EAAAAIBAJ&pg=7304,3766749`.

[3] Takeo Kanade : *Autonomous land vehicle project at CMU* [online]. Carnegie-Mellon University, 1900 . Published 1986. [seen 2018-04-30]. Available at: `https://dl.acm.org/citation.cfm?id=325197`.

[4] Prof. Jürgen Schmidhuber: *Prof. Schmidhuber's highlights of robot car history* [online]. Bundeswehr University Munich, 1973. Published 2009. [seen 2018-04-30]. Available at: `http://people.idsia.ch/~juergen/robotcars.html`.

[5] The Guardian: *Self-driving Uber kills Arizona woman in first fatal crash involving pedestrian* [online]. The Guardian, 1821. Page last edited 22. 3. 2018. [seen 2018-04-30]. Available at: `https://www.theguardian.com/technology/2018/mar/19/uber-self-driving-car-kills-woman-arizona-tempe`.

[6] Udacity: *Self-Driving Car* [online]. Udacity, 2011. [seen 2018-04-30]. Available at: `https://eu.udacity.com/course/self-driving-car-engineer-nanodegree--nd013`.

[7] Carla: *CARLA Open-source simulator for autonomous driving research.* [online]. CARLA Team. [seen 2018-04-30]. Available at: `http://carla.org/`.

[8] Unity: *Unity 3D - FAQ* [online]. Unity Technologies SF, 2004. [seen 2018-04-30]. Available at: https://unity3d.com/unity/faq.

[9] Unreal Engine: *Unreal Engine - FAQ* [online]. Epic Games, 1991. [seen 2018-04-30]. Available at: https://www.unrealengine.com/en-US/faq.

[10] CryEngine: *CryEngine - FAQ* [online]. Crytek GmbH, 1999. [seen 2018-04-30]. Available at: https://www.cryengine.com/faq.

[11] TensorFlow: *TensorFlow Version Compatibility* [online]. Google, 1998. [seen 2018-04-30]. Available at: https://www.tensorflow.org/programmers_guide/version_compat.

[12] François Chollet: *Keras documentation* [online]. François Chollet, 2015. [seen 2018-04-30]. Available at: https://keras.io/.

[13] Caffe2 *Caffe2* [online]. University of California, Berkeley, 1868. [seen 2018-04-30]. Available at: https://caffe2.ai/.

[14] Michael Nielsen: *Using neural nets to recognize handwritten digits* [online]. [vid. 2016-05-13]. Available at: http://neuralnetworksanddeeplearning.com/chap1.html.

[15] Logistic function: *Wikipedia: The Free Encylopedia* [online]. Wikimedia Foundation, 2003. Page last edited 15. 4. 2018 v 08:23. [seen 2018-04-30]. Available at: https://en.wikipedia.org/wiki/Logistic_function.

[16] ujjwalkarn: *An Intuitive Explanation of Convolutional Neural Networks* [online]. [vid. 2016-05-13]. Available at: https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/.

[17] Jan Koutník, Jürgen Schmidhuber, Faustino Gomez: *Evolving Deep Unsupervised Convolutional Networks for Vision-Based Reinforcement Learning* [online]. Published 2014. [seen 2018-04-30]. Available at: http://people.idsia.ch/~koutnik/papers/koutnik2014gecco.pdf.

[18] Eclipse Deeplearning4j: *A Beginner's Guide to Recurrent Networks and LSTMs* [online]. [vid. 2016-05-13]. Available at: https://deeplearning4j.org/lstm.html.

[19] Ramon Quiza, J. Paulo Davim *Computational modeling of machining systems* [online]. Published 2009. [seen 2018-04-30]. Available at: https://www.researchgate.net/figure/Graph-of-a-recurrent-neural-network_fig3_234055140.

[20] You only look once: *YOLO: Real-Time Object Detection* [online]. Joseph Chet Redmon. [seen 2018-04-30]. Available at: https://pjreddie.com/darknet/yolo/.

[21] Common objects in context: *Common objects in context* [online]. cocodataset.org. [seen 2018-04-30]. Available at: `http://cocodataset.org/#home`.

[22] Bc. Michal Karger *Algoritmus Diferenciální Evoluce s prvky deterministického chaosu (ChaosDE) v prostředí Mathematica* [online]. Published 2011. [seen 2018-04-30]. Available at: `http://digilib.k.utb.cz/bitstream/handle/10563/16562/karger_2011_dp.pdf?sequence=1&isAllowed=y`.

Appendix A

Acronyms

LSTM Long-Short Term Memory

fps frames per second

YOLO You Only Look Once

COCO Common Objects in Context

TCP Transmission Control Protocol

API Application Programming Interface

UDP User Datagram Protocol

ReLU Rectified Linear Unit

Appendix B

Users manual

B.1 Simulator

The environment has very limited settings once it is compiled, only IP, port and type of a TCP socket can be changed. All other modifications have to be done in Unity editor. These changes include for example:

- Adding new predefined types of cars, obstacles and scenarios
- Changing of
  - properties of wheels
  - resolution of sent images
  - sensors delay
- Adding new types of messages

How to make these changes should be clear from the chapter Implementation4. User should start the simulation by pressing Start button in the main menu. The button should be pressed before/after a neural network is running and posing as a client/server.

B.2 Neural network

The neural network is run from the client.nb file. To run the network all cells have to be evaluated before evaluating the last cell containing call of function `StartSession` or `StartLearningSession`. If the environment is running and IP and port are set up correctly, Mathematica should be able to connect to this environment and start the simulation. Default IP is 127.0.0.1 (localhost) and port 5005, both these values can be of course changed according to the needs of user. More details on how the Mathematicas section works is in chapter Implementation4.

Appendix C

Images

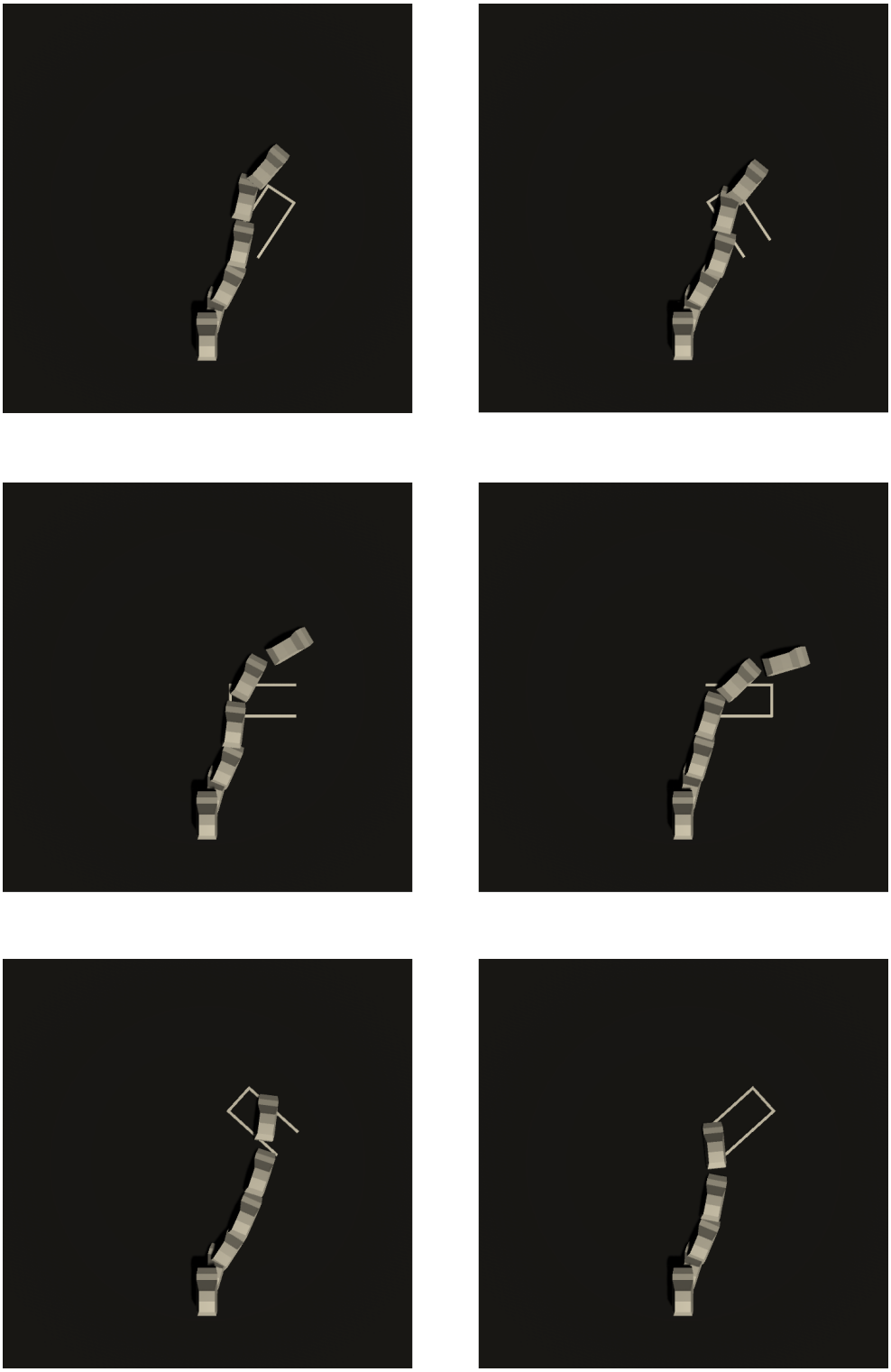

Figure C.1: Scenarios 2 to 7 with trajectory of best performing candidate from Experiment 6 5.1.6

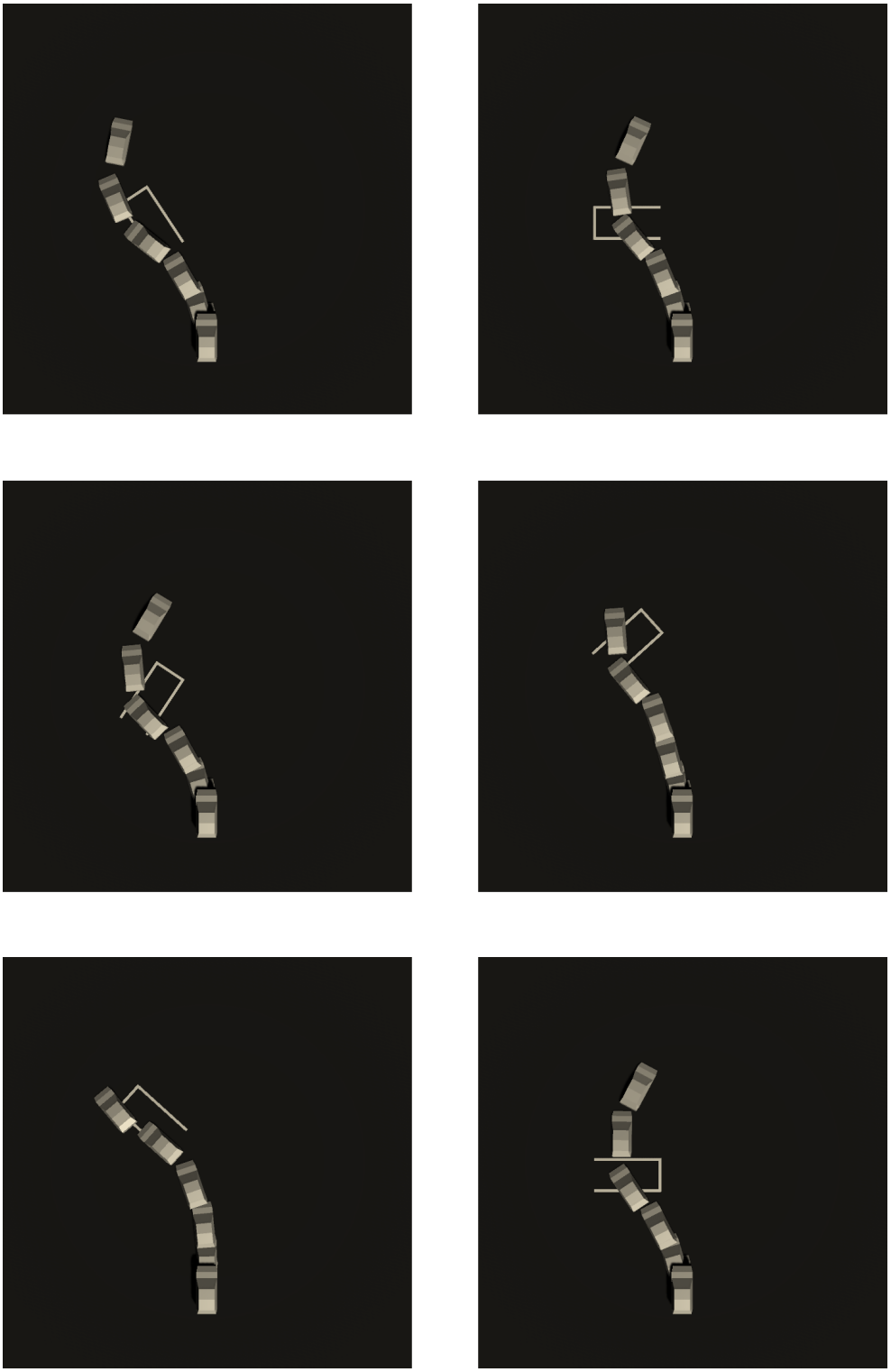

Figure C.2: Scenarios 8 to 13 with trajectory of best performing candidate from Experiment 6 5.1.6

Appendix D

CD content

- `readme.txt` — Description of CD content
- `exe` — Folder with executable implementation
- `src`
  - `impl` — Source code of implementation
    - `neural network` — Mathematica scripts
    - `simulator` — Unity project
  - `thesis` — Source code of thesis in LaTeX
- `text` — Thesis
  - `DP_Laube_Daniel_2018.pdf` — Thesis in PDF