FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	DeepRCar: An Autonomous Car Model
Student:	Bc. David Ungurean
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

Search for RC car models suitable for autonomous driving implementation. Focus on models which allow the control software to be run onboard. The autonomous control will be based primarily on visual inputs from an onboard camera. Other sensors such as ultrasonic distance sensors can be also used. Implement the control algorithms using deep neural networks which use the visual information as input and produce control signals for steering and speed control. Design and implement several experiments to test various neural network architectures and image preprocessing methods.

References

- J. Koutnik, J. Schmidhuber, F. Gomez, Online evolution of deep convolutional network for vision-based RL, SAB 2014: http://people.idsia.ch/ koutnik/papers/koutnik2014sab.pdf
- Zheng Wang, Self Driving RC Car: https://zhengludwig.wordpress.com/projects/self-driving-rc-car/

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

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

Prague February 16, 2018

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

DeepRCar: An Autonomous Car Model

Bc. David Ungurean

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

May 9, 2018

Acknowledgements

First of all, I would like to thank my supervisor Dr. Buk for giving me the chance to explore such an interesting field of machine learning and for always pointing me in the right direction, whenever I was starting to get lost. I am also very grateful to Dr. Doina Caragea, who oversaw my writing at Kansas State University and provided valuable insight and resources which made this work possible.

Additionally, I thank Krištof Pučejdl who helped me to design the top plastic plate for the RC car. I also offer my heartfelt thanks to Matěj Hlaváček for his friendship and great advice.

Finally, I want to thank my friends and family for their love and support.

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 particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on May 9, 2018 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Ungurean, David. *DeepRCar: An Autonomous Car Model.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

Tato práce se zabývá stavbou modelu autonomního vozu na dálkové ovládání a jeho kontrolou pomocí hlubokých neuronových sítí. Vozidlo je schopno zatáčet samo pouze na základě vizuálního vstupu z přední kamery. Text práce popisuje jeho kompletní vývoj od výběru hardwarových komponent, návrhu kontrolního systému, až po selekci a učení konvoluční neuronové sítě, která ovládá nastavení kol. Model se naučil rozeznat okraje jízdního pruhu, přestože měl během učení přístup pouze ke dvojicím vstupní obrázek a příslušný úhel kol pro danou situaci. Finální systém operuje při 20 snímcích za sekundu na jednodeskovém počítači Raspberry Pi 3.

Klíčová slova neuronové sítě, deep learning, konvoluce, autonomní vozidlo, Raspberry Pi, učení s učitelem, strojové vidění

Abstract

I present DeepRCar, a simplified self-driving radio controlled car platform that is controlled by deep neural networks. This car takes images from a front facing camera as its only input and produces steering commands as output. This thesis describes the entire process of its creation from hardware requirements, through the design of the control system, up to the selection and training of a convolutional neural network that manages its driving decisions. The network was trained in an end-to-end manner and learned to recognize useful road features, such as lane markings, when only camera images and corresponding steering angles were presented during training. The final system is capable of running at 20 frames per second on a Raspberry Pi 3.

Keywords neural network, deep learning, convolution, self-driving car, Raspberry Pi, end-to-end, supervised learning, computer vision

Contents

List of Figures

List of Tables

Introduction

Motivation

Convolutional Neural Networks (CNNs) and other deep architectures have achieved tremendous results in the field of computer vision. In most cases, they surpassed previous hand-crafted feature extraction based systems and set up a new state-of-the-art for tasks such as image classification [7], [8], [9], image captioning [10], [11], object detection [12] or semantic segmentation [13] [14]. These networks learn automatically from training data and are able to capture complex relationships that are otherwise very difficult for humans to describe by a set of hand-written rules. With the exponentially growing amount of available data and the increasing processing power of modern computers, we are able to train these networks to yield better and better results. Their impact can be seen almost anywhere. Whether it is the improved quality of automatic machine translation [15], human competitive results at malignancy detection in radiology imagery [16], [17], or just the possibility to scan for pictures containing our face on social media. Another field that could largely benefit from deep neural networks is the automotive industry with self-driving cars.

Once developed, autonomous vehicles will revolutionize transportation. Self-driving cars will save millions of lives in situations where it is impossible for a person to prevent a car accident. The reaction times and alertness of a machine are much better. In addition, long distance cameras and ultrasonic sensors further equip these cars with super-human abilities. For such reasons, we see many corporations and researchers trying to develop technologies that would allow for a fully autonomous driving experience. Companies like *Google* or *Udacity* also try to educate the public on this topic. They provide free courses and open-source libraries as `tensorflow` that enable practically anyone to build machine learning models and participate in solving of the puzzle.

Goal of the Project

The ultimate goal of the thesis is to build a low cost prototype of an autonomous RC car through end-to-end machine learning, primarily using deep neural networks (details in Chapter 4).

This car should be able to drive itself on a flat surface that will mimic a simplified road. The main input of the car will be real time video from a camera, which will be mounted on the top. The system should then output corresponding steering commands and control the car accordingly. Since the camera will be the only input for the controller, the goal of the thesis will be to teach the car how to steer. Avoiding obstacles is a different problem which is also possible to solve, but combining it with steering and creating a single controller to handle both situations is beyond the scope of this thesis. However, I will use ultrasonic sensors to detect obstacles on the road and stop the car accordingly. This will work as a separate module that will not interfere with the process of steering which will be managed exclusively by the neural network.

The network will be trained on a separate machine and then transferred to an on-board computer, that will control the car. The vehicle will then be fully independent of other machines.

Another goal of the thesis is to make this system affordable and easy to build. I will use a mainstream single board computer and an inexpensive car chassis. The software will be written with standard machine learning libraries and easy to extend and reproduce.

Document Structure

This thesis starts with a short summary of the background associated with self-driving cars in Chapter 1. That is followed by a description of how to assemble an RC car that is able to drive autonomously. Chapter 5 focuses on the design, creation and training of a controller which operates such car. A more thorough examination of the controller's performance is described in Chapter 6.

Chapter 1

Background and Related Work

1.1 Levels of Autonomous Driving

The National Highway and Traffic Safety Administration (NHTSA) defines five levels of vehicle autonomy [18].

- *no automation (level 0)* — The human driver does all the driving.
- *driver assistance (level 1)* — The car assists the driver with a single vehicle control such as steering or braking, but not both at the same time.

  Example: *lane keeping, cruise control or assisted breaking.*
- *partial automation (level 2)* — The execution of both steering and accelerating/decelerating is performed by the car. Human driver is still expected to perform all remaining tasks associated with driving and monitoring of the driving environment.

  Example: *Tesla Autopilot [1]*
- *conditional automation (level 3)* — The driving system takes complete control over the entire driving task under special circumstances. Human driver is expected to intervene when required. When the circumstances aren't met, human driver does all the driving.

  Example: *Waymo (Google) self-driving car [19]*
- *high automation (level 4)* — The driving system takes complete control over the entire driving task under special circumstances. The human driver does not need to pay attention or intervene in those situations. If a change of circumstances arises, the car needs to be able to stop and park safely in case the driver does not retake control.

  Example: *Waymo announced in 2017 that they are testing level 4 driving* [20].

- *full automation (level 5)* — The driving system takes complete control over the entire driving task under **all** circumstances. The human driver does not need to be inside the car.

  Example: *JohnyCab from Total Recall*

This thesis aims to create a system with a level of automation between 1 and 2. Steering will be fully autonomous, decelerating will only work when an obstacle is met.

1.2 Approaches to Steering in Autonomous Driving

There are currently three main ways of how to deal with steering in a self-driving vehicle.

- Non-AI Approach (manual engineering)
- AI Approach
- Combination of AI and Non-AI

1.2.1 Non-AI Approach

The non-AI approach uses control theory to calculate a steering angle to keep the vehicle on the desired trajectory, which is usually detected through computer vision algorithms. One of the most popular methods in control theory is PID (Proportional Integral Derivative) controller [21]. The controller works in a loop, which continuously calculates an error value `e(t)` as a difference between the vehicle's feedback and the next command signal. Afterwards, a correction is calculated and applied.

The correction value `u(t)` consists of three parts (proportional, integral, derivative) and can be computed from the error `e(t)` as shown in Figure 1.1.

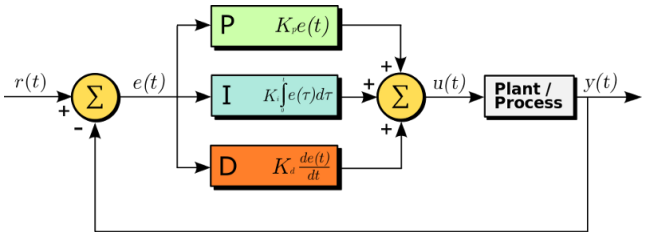


Figure 1.1: Calculation of correction value in a PID loop

The whole mathematical formula is then following:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)\,\mathrm{d}\tau + K_d \frac{\mathrm{d}e(t)}{\mathrm{d}t} \tag{1.1}$$

Each term has its coefficient (K_p, K_i, K_d) which needs to be tuned. I will not go into detail on options for parameter tuning as the approach for autonomous driving that will be used in this project is the AI approach. For more material on PID controllers and their use in autonomous vehicles, please refer to [22], [23], [24].

1.2.2 AI Approach

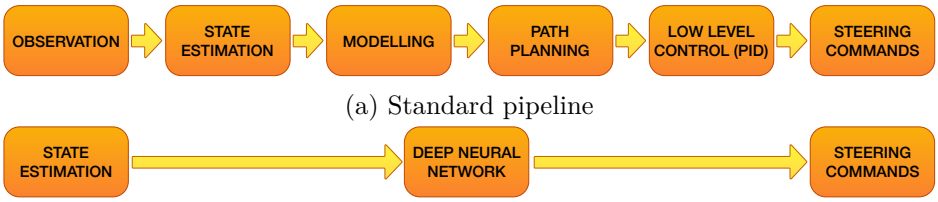
The AI approach, in comparison to the previous, does not calculate the precise steering angle using mathematical equations, but instead relies on an intelligent agent which chooses the best course of action.

Such agent can be trained with deep learning on large datasets of driving data with the goal of recognizing road features and predicting the direction in which the vehicle should travel in order to follow the road.

With the rise of deep learning and recent accomplishments of companies like Google, Tesla or Uber in the field of machine learning, new possibilities arise. The combination of artificial neural networks being a universal approximator [25], breakthroughs in research on CNNs and the improvements in GPU performance [26] makes creation of an NN-based controller, which can drive an RC car, seem like a reachable goal.

1.3 End-To-End Deep Learning

As I will discuss in Section 1.4, the standard approach in solving the task of autonomous driving is to decompose the problem into several steps such as pedestrian detection, lane marking detection, path/motion planning and motor control. In recent years several attempts [27], [28], [29], [30] were made to simplify this pipeline with end-to-end deep learning. This approach merges all the aforementioned processing steps together, leading to a smaller, more elegant system. Figure 1.2 shows the difference between the two pipelines.



(a) Standard pipeline

(b) End-to-end learning pipeline

Figure 1.2: Differences between modularized and end-to-end approach

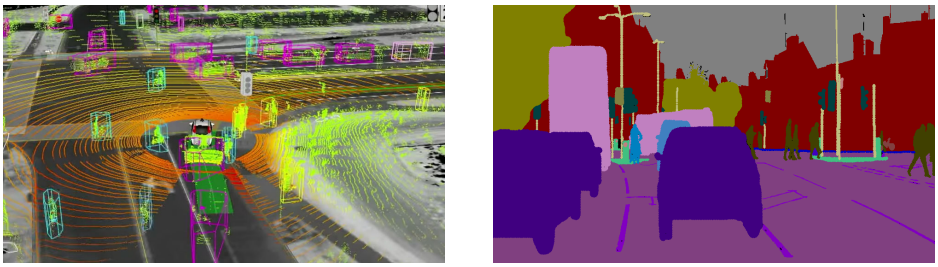
The first big breakthrough in the use of end-to-end deep learning for self-driving cars is dated to 1989 and the accomplishments of Pomerleau [27], who built the Autonomous Land Vehicle in a Neural Network (ALVINN). He used a simple feed forward neural network with a single hidden layer of 29 units.

This research shows that it is possible to use end-to-end neural nets to drive a car. In 2004, the Defense Advanced Research Projects Agency (DARPA) came up with a project known as DARPA Autonomous Vehicle (DAVE) [28], which is a 1/10th scale autonomous RC car prototype able to drive in terrain and avoid obstacles. DAVE served as groundwork for the most recent achievement of NVIDIA in the field of self-driving cars – the DAVE-2. In the paper *End-To-End Learning for Self-Driving Cars* [29], NVIDIA published a state-of-the-art network architecture that benefits from the modern convolutions and processing power of present-day GPUs. Their car prototype was able to drive on highways and in simple traffic on local flat roads.

1.4 Real World Examples of Self-Driving Cars

Modern self-driving car companies do not use end-to-end learning as it is infeasible to collect enough training data, to cover all possible scenarios in a real world driving experience. As [31] showed in 2016, in a regime where extremely high accuracy is necessary, the amount of data required to train an end-to-end system grows exponentially compared to a modular system. In a modular approach, the system is broken down into sub-modules with different responsibilities as pedestrian detection or path planning. Each module is then trained either using machine learning or manual engineering, depending on empirical success.

Another important difference between real world self-driving cars and the RC car model used in this thesis is the form and size of input. Companies like Waymo or Tesla combine data from multiple sensors and cameras together to create a visual model of the car's surroundings (see Figure 1.3 for better visualization).



Sensor fusion in Waymo SDC [19] Image segmentation with SegNet [13]

Figure 1.3: Technologies used to model surroundings of modern self-driving cars

Expensive technologies such as LIDAR [32], radar and ultrasonic sensors are necessary to create a realistic, 360 degree model of the environment. For example *Tesla model S* uses a combination of wide, narrow, normal front

cameras, side, rearward looking side cameras and a single back camera. In total there are twelve ultrasonic sensors and on top of all that, there is a front facing RADAR used primarily to detect relative speed of objects in front of the car.

Figure 1.4: Sensors and cameras on Tesla Model S [1]

Such cars also use variations of Bayesian Simultaneous Localization and Mapping (SLAM) [33] algorithms which blend together the data from all sensors.

1.5 Chosen Approach

The goal of this thesis is to create an autonomous RC car which will drive on artificial flat roads without traffic. Such task is much simpler than a real world scenario, so it is not necessary to complicate the design with a modular architecture. Instead, it is a good opportunity to show the strength and clarity of an end-to-end approach applied to challenging tasks such as autonomous driving.

Chapter 2

Car Design

2.1 Requirements on the System

In order to ensure successful training, we need the system to have the properties listed below.

- *It needs to capture images from camera and send them in real time to an onboard computer.*

  The car will have a camera mounted on top which will be connected to the onboard computer

- *The hardware needs to be able to receive and execute orders from the agent.*

  The onboard computer, which will run the agent, will be connected to a servo and, with the use of libraries, control its steering angle.

  This requirement restrains us the most when choosing which RC vehicle to buy. Some cars simply won't allow the user to precisely control the steering angle. Fortunately, this can be solved by using a custom servo and a servo controlling unit.

- *The onboard computer needs to either receive orders from a distant machine or be able to store large neural network models and classify images in real time.*

  I chose the second approach of storing the model directly on the onboard computer which will also perform the decision-making process. This way the car will be truly autonomous and will not have to rely on other machines. However, the model will be trained on a separate machine with greater computational capabilities.

2.2 Final Car Setup

Usually the car is attached to a remote control which sends signals to steer the wheels depending on which buttons the user presses. In our case, we want to control the car purely with a computer program. In order to do that, we will need to buy several extra parts, rewire the car's servo and connect it to the onboard computer. We will also need a camera to collect images and a motor controller. The final list of all required parts can be seen in Table 2.1.

2.2.1 Car Chassis

The RC car is one of the most vital components of the whole system. There are several things that we need to consider when choosing which one to buy. The car's ESC (electronic speed controller) needs to be separate from its signal receiver. We need to be able to connect the steering servo and the ESC to our motor controller in order to fully control both. The ESC should also allow for fine-tuning of the steering angle. Some low-end cars only grant discrete angle steps (left, right and center). It is still possible to make such car drive, but having the full control over steering is preferred. The last thing to consider is the size, or more precisely the scale of the model. Scales can range from 1/32 to 1/8 and beyond. Here, the only restriction is that the car needs to be big enough to carry all parts and batteries.

Final choice: Buggy Radio Car 1/16 2.4Ghz Exceed RC Blaze

I chose an RC car from *Exceed* in 1/16 scale, because its size is sufficient to fit and carry all mandatory components and it is still compact enough that I will not have to create an enormous driving environment for it to train. It is affordable, has good servo and ESC setup and drives very well.

Figure 2.1: Buggy Radio Car 1/16 2.4Ghz Exceed RC Blaze

2.2.2 Servos, Motors and ESC

I did not have to opt for a custom ESC because the *Exceed* buggy has a fairly good ESC, which will allow me to finely modify its speed and angle of its wheels. Please note that a custom ESC may be required if you want to recreate this project with a cheaper RC car. For more info on how to choose a viable ESC, refer to [34].

2.2.3 Camera

The resolution of the camera is not very important as we will downsample the input to reduce training time. Wider angle camera will be beneficial, as it can capture more information about the car's surroundings.

Final choice: SainSmart Wide Angle Fish-Eye Camera Lenses

The SainSmart Wide Angle Camera is a good trade-off between price and build quality. It is sufficient for this task and readily available in most online stores.

Figure 2.2: SainSmart Wide Angle Fish-Eye Camera Lenses

2.2.4 Onboard Computer

Depending on the scale of your project, you can choose from a wide variety of onboard computers. One can go with *NVIDIA DRIVE PX Pegasus* for larger projects. *Arduino, Raspbery Pi* and *Orange Pi* stand on the other end of the spectrum as lower cost single-board computers.

Final choice: Raspberry Pi 3 model B

This project aims to be an affordable do-it-yourself version of a self-driving-car. I chose Raspberry Pi 3, because I am familiar with the device and because it also has a built-in Wi-Fi module that will ease both manipulation and debugging. It is also very well adopted by users and there are numerous

materials covering its use publicly available on the worldwide web. Another important factor that played in Pi's favor is the availability of *tensorflow for Raspberry Pi* and its relatively good processing power.

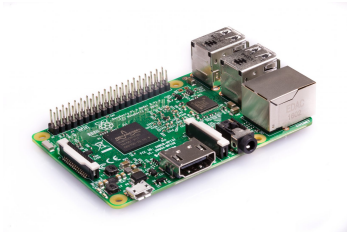


Figure 2.3: Raspberry Pi 3 model B

2.2.5 Motor and Servo Controller

To control the motor and servos, we need to send it specific PWM signals. I decided to get a special add-on board for the Pi which will make it easier. *Adafruit Industries, LLC* manufactures numerous Raspberry Pi compatible motor controllers. Another option could be *Emlid NAVIO2* [35]. It has a lot of useful sensors like accelerometers, gyroscopes and magnetometers. Unfortunately, the price of $164 makes it too expensive for this type of project.

Final choice: Adafruit PCA9685 16 Channel 12 Bit PWM Servo Driver

I was inspired by [36] and chose the *Adafruit PCA9685*, which can control up to 4 servos with full PWM speed control [37]. It is relatively small, so it will not get in the way. Lastly, it is affordable, relatively easy to set-up and comes with a publicly available software library.

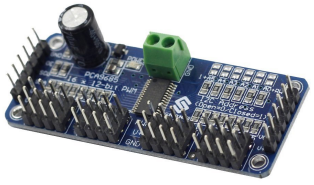


Figure 2.4: Adafruit PCA9685 16 Channel 12 Bit PWM Servo Driver

2.2.6 Cables, Battery, Screws

The ESC is powered from the RC car's battery. All other components, mainly the Pi, need to be connected to an external source of power. This can be either a set of AA batteries or any 5V/2A output powerbank. The size and capacity of the powerbank depends on the size of the car and how long does the user want to drive it. Together with the powerbank, we will need a set of jumper wires to connect the PCA9685 to the Raspberry Pi.

Figure 2.5: The powerbank (Anker Astro E1) and jumper wires used in this project

2.2.7 Sensors

Sensors are optional. I decided to use HC-SR04 ultrasonic sensors to detect obstacles in front of the car. Another useful options might be accelerometers or gyroscopes.

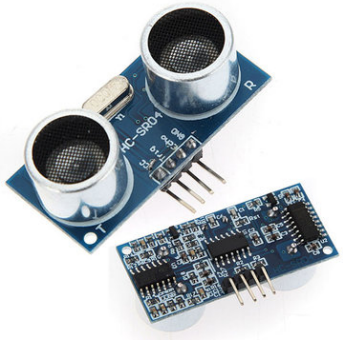


Figure 2.6: HC-SR04 Ultrasonic sensor

2.2.8 Top Plastic Plate

Fitting everything on top of the car can be tricky. To avoid using duct tape, I designed a plastic plate that can be placed on top of the car instead of the original cover. This plate has several supports, where it is possible to attach both the Raspberry Pi and the motor controller. It also has a small hole in the center through which one can reach to the power switch and lead necessary cables from.

Together with the plate, I had also 3d printed a camera mount. This mount can be adjusted to different heights and attached under various angles. You can see the design in Figure 2.7. All OpenSCAD source files and STLs can be found on the enclosed CD.

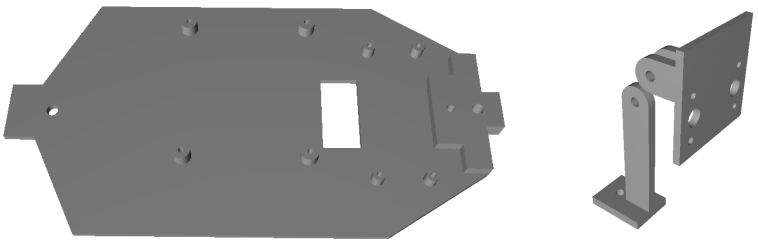


Figure 2.7: Top plate with attachments (left) and the camera mount (right)

2.2.9 Parts Summary

The list of all parts can be seen in Table 2.1. Every part is available for purchase on Amazon, from where the prices were obtained.

Part	Count	Cost
Buggy Radio Car 1/16 2.4Ghz Exceed RC Blaze	1x	$74.95
Raspberry Pi 3 model B	1x	$36.39
Adafruit PCA9685 12 Bit PWM Servo Driver	1x	$20.76
SainSmart Wide Angle Fish-Eye Camera Lenses	1x	$26.99
HC-SR04 Ultrasonic sensor	2x	$3.49
Anker Astro E1 (power bank)	1x	$17.99
Total		$180.57

Table 2.1: List of used parts and their prices

By relying heavily on camera as the main input, using fewer sensors and having a smaller scale chassis, I was able to cut down the costs to $180. The price is much lower compared to other prototype vehicles which can cost up to several thousand dollars [28], [32].

2.3 Assembling

This section provides the reader with a brief step-by-step tutorial on how to connect all parts together.

- Gather all parts from Section 2.2.9
- Print out the top plate and the camera mount
- Drill through the holes in top plate to fit M2.5 screws
- Assemble the camera mount
- Screw the camera mount to the top plate
- Attach the Raspberry Pi and motor controller to the top plate
- Disconnect the servo and ESC cables from the car
- Connect everything together according to the schema in Figure 2.8

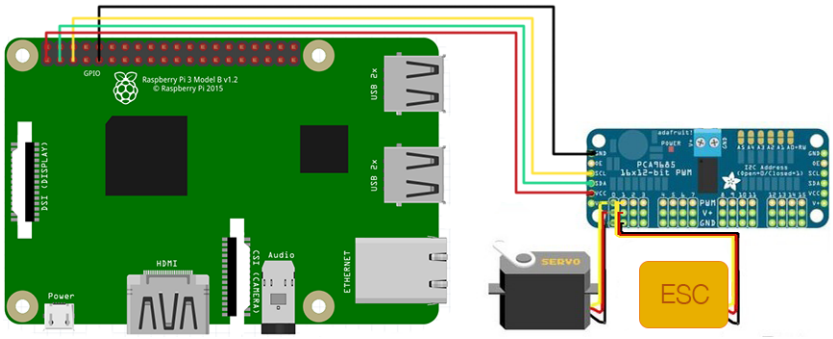


Figure 2.8: Electronic schema of the system's architecture

- Attach the camera to the camera mount and connect it to the Pi
- Attach the powerbank to the back of the car and lead a cable from it to the Pi.
- Put the top plate with all parts on top of the RC car. It should fit tightly.
- Optionally, connect the HC-SR04 ultrasonic sensor.
- Turn on the power switch through a small hole in the 3D printed top plate

If everything is in place, the ESC should beep and lights on the Raspberry Pi and the motor controller will turn on. The finished and ready to drive vehicle can be seen in Figure 2.9.

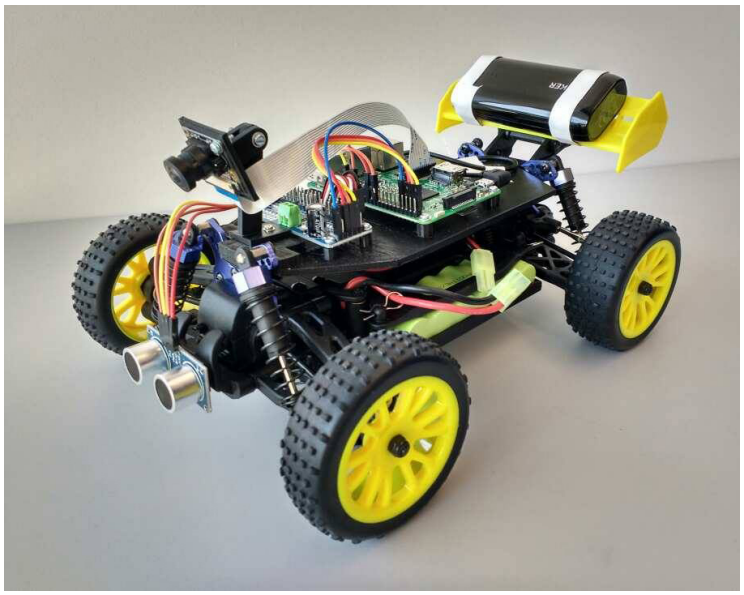


Figure 2.9: Assembled RC car

Chapter 3

Controlling the Car

This chapter describes the architecture of the software that was used to control the autonomous vehicle. I briefly mention which technologies I used to create every module and how the main run loop works.

3.1 System Architecture

The main application is run on the Raspberry Pi. Together with the main program, I have also implemented a web interface through which users can interact with the system. The vehicle itself can be either driven manually from a web browser, or automatically using a pretrained ML model. Deployment diagram depicting the situation can be seen in Figure 3.1. Section 3.6 then describes the web interface in more detail.

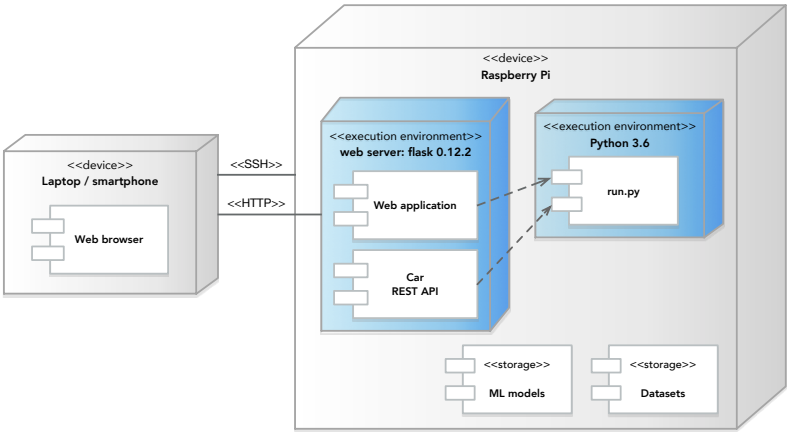


Figure 3.1: Deployment diagram

3.2 Used Technologies

The entire software stack is written in `Python 3.6` except for the view layer of the web server, which had to be written using `HTML` and `Javascript`. I used standard libraries for working with threads and sockets. For higher level constructs, I tried to use widely available 3rd party libraries.

Namely, I used `Flask` and `CherryPy` for the backend part of the web. Frontend was programmed using `Bootstrap`.

The low-level layer that controls hardware is built on libraries as `PiCamera` and `Adafruit_PCA9685`. The latter controls the servos and the first works as a wrapper for the camera.

I used standard machine learning libraries to train the network. Those were `numPy`, `sklearn`, `pandas`, `Tensorflow` and `Keras`.

3.3 Car's Main Loop

The car's main loop runs indefinitely with a defined frequency. At first, the car finds out whether it can continue driving or if there is an obstacle in front of it. Then it gets the latest speed and steering angle, which are provided either by user from the web interface or from a pretrained model (based on the most recent camera image). The speed and angle are then saved and fed into the actuator that executes appropriate steering commands. The process is depicted on the following diagram.

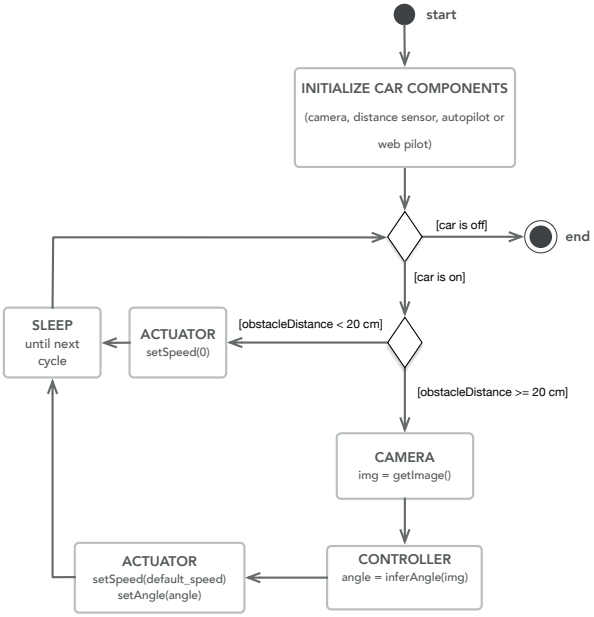


Figure 3.2: Car's life cycle

It is worth to mention that the camera unit, the web interface and the distance sensor all run separately on their own threads to not stall the execution loop. On the other hand, the most computationally demanding part of this loop is the inferring process of the neural network. A more thorough performance breakdown can be found in Section 6.4, where I focus on the suitability of Raspberry Pi for this task and the measured performance of the system.

A simplified version of the main loop code can be seen in Listing 3.1.

```
while self.is_on:
  if distance_sensor.should_stop():
    self.throttle.drive(0)
    continue

  self.update_inputs()         # get latest angle/speed from controller
  self.propagate_changes()     # reflect the changes in actuator

  sleep(time_till_next_cycle) # sleep until next cycle
```

Listing 3.1: Simplified version of the main loop

3.4 Controlling the Servo and the ESC

To control steering and throttle of the car I had to use *PCA9685* HAT, as the Pi is not very good at generating PWM signals. The `Adafruit_PCA9685` library allows the user to connect to channels 0-16 on the board and send pulses with defined frequency. As depicted in Figure 2.8, channel 0 serves for steering and channel 1 for throttle.

It is essential to find which pulse range the servo and ESC respond to. That can be tested with the script `calibrate.py` which can be found on the enclosed CD. A short example of the script can be seen in Listing 3.2.

```
import Adafruit_PCA9685
# Initialise the PCA9685 using the standard address (0x40).
pwm = Adafruit_PCA9685.PCA9685()
pwm.set_pwm_freq(60)   # Set frequency to 60hz, good for servos.
# Send pulse to channel (0 for steering, 1 for throttle)
pwm.set_pwm(channel, 0, pulse)  # pulse range from 0-1000, usually ~350
```

Listing 3.2: Example of servo control

Once the correct pulse range is found and saved in file `settings.py`, we can use classes `Steering` and `Throttle` from file `car_parts.py` to fully control the car.

3.5 Ultrasonic Sensor

The file `ultrasonic.py` contains an implementation of class DistanceSensor. To get the measurements from the ultrasonic sensor, I used a 3rd party library named `Bluetin_Echo`. This framework provides class `Echo`, which is instantiated using two GPIO pins (*trigger* and *echo*) that the sensor was attached to. The sensor then runs on a separate thread, where it continuously gets the latest distance from obstacles in front of the car. This way, it does not block the main thread. The car instance then asks the DistanceSensor asynchronously in the main loop, whether it should stop or continue driving. See file `car.py` for further implementation details.

3.6 Web Interface

The web interface was built in Python using model-view-controller architecture. I used `flask` for the backend part of the website. For the graphical user interface I used `Bootstrap` [38]. It ensures compatibility between mainstream browsers and makes the website responsive, meaning that the layout will adapt to fit on smaller screens.

Users can interact with almost every part of the car through the web interface. It is possible to steer the car either using buttons and sliders on the web page or through a remote *PlayStation 3* controller. Users can also turn on/off dataset recording, modify speed and see the camera input in real time. Once the car is started, the interface can be found at `ip_address:5000`.

In case it is not possible to interact with the server in graphical interface, the server also has `REST API` with endpoints `/drive`, `/recording` and `/video_feed`.

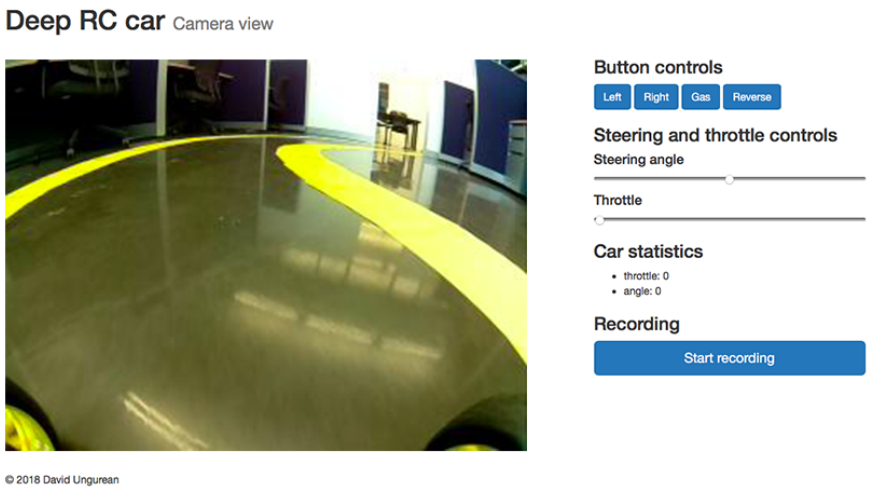


Figure 3.3: Web interface

3.7 Command Line Interface

The car can be run in two modes. Either in autopilot mode, where the steering is managed by the controller (more in Chapter 5), or in manual mode, where the user handles steering via web interface.

`run.py --drive` ***`model_type weights_path`*** will start the car in an autopilot with *model_type* (*mlp, nvidia, custom*) and pretrained model weights saved at path *weights_path*.

`run.py --record` ***`path_to_store`*** will start a dataset recording procedure, which will snap pictures and corresponding values of *speed* and *steering angle* at a frequency described in file `config.py`. The output will be saved at *path_to_store*.

3.8 Dataset Recording

Dataset recording can be initiated either from the web interface or via command line using the command `run.py --record path`. Manual pilot is set by default when the car is run through this command.

The process is governed by a single object of class `Recorder`. It creates a folder hierarchy which reflects the current date and the number of records. A single dataset entry is saved on every tick of the car's main loop. The frequency can be changed in file `config.py`. Since this is a supervised learning problem, we need to have an input object and a corresponding output value. In this case our input is an image and the output is a json file containing the *speed* and *steering angle* at the *time* the picture was taken. These files are later used to train the machine learning model.

Chapter 4

Principles of Deep Learning

This chapter aims to introduce the reader to the concept of Deep Learning and the benefits of Convolutional Neural Networks (CNN)[1]. First, the basic structure of an artificial neuron is described. Next Section focuses on Feed Forward Neural Networks and the last part talks about the theoretical foundations of CNNs, which are heavily used in the practical part of this thesis.

4.1 Basics of an Artificial Neural Network

Much like their biological counterparts, Artificial Neural Networks consist of independent interconnected units called neurons. These neurons compute their activation based on activations in previous layer and weights of their connections.

The main belief is that many simple units working together can add up to an intelligent whole. One of the key concepts of deep learning is *distributed representation*. This is the idea that each input to a system should be represented by many features, and each feature should be involved in the representation of many possible inputs [5]. For example, we want to build a classifier that can recognize pictures of pens, pencils and rulers and each object can be either blue, green, red or yellow. In a naive approach we would create 12 neurons where every single one of them would have to learn the concept of object identity and color. Of course, as we increase the number of objects and colors that we want to identify, the size of our model grows drastically. On the other hand, with distributed representation we can narrow the requirements down to just 7 neurons, where 3 will learn how a pen, pencil and a ruler look like, and the remaining 4 will each learn the concept of a single color like red, green, blue or yellow.

[1] Readers familiar with deep learning may skip this chapter

4.1.1 Artificial Neuron

The structure of a single neuron unit by comparison to its biological counterpart can be seen in Figure 4.1.

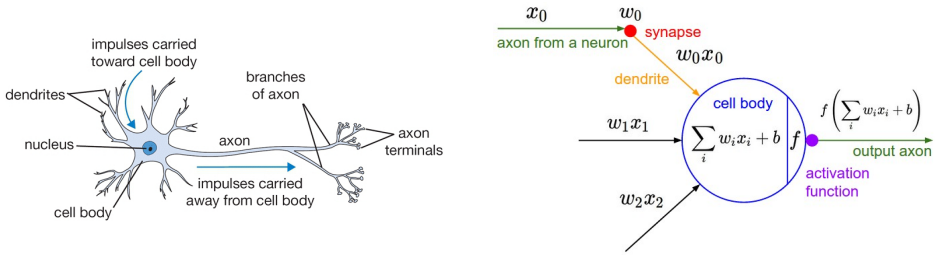


Figure 4.1: Comparison between a biological neuron (left) and a mathematical model of an artificial neuron (right) [2]

Output of a single neuron unit is calculated as shown in Equation 4.1, where x_i is the i^{th} input, w_i is its weight, b refers to the bias/threshold of the neuron and σ is the activation function.

$$y = \sigma(w_i x_i + b) \tag{4.1}$$

4.1.1.1 Activation Functions

In order to fit nonlinear patterns in input data, we need to use nonlinear activation functions. The most frequent these days are sigmoid, hyperbolic tangent and ReLU. Each has different drawbacks and is used for different purposes. The only restriction is that the function must be differentiable in order to perform gradient descent based learning.

sigmoid : $\sigma(x) = \frac{1}{1+e^{-x}}$

hyperbolic tangent : $\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$

ReLU : $ReLU(x) = max(0, x)$

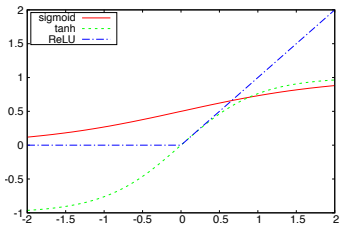


Figure 4.2: Plots of common activation functions

4.2 Feed Forward Neural Network

A feed forward neural network is a collection of neurons organized in layers. There is an *input layer*, *output layer* and the remaining layers are called *hidden layers*. If there is more than one hidden layer, the network is called a *deep neural network*.

Every neuron in each layer is connected to neurons in the following layer creating a directional acyclic graph.

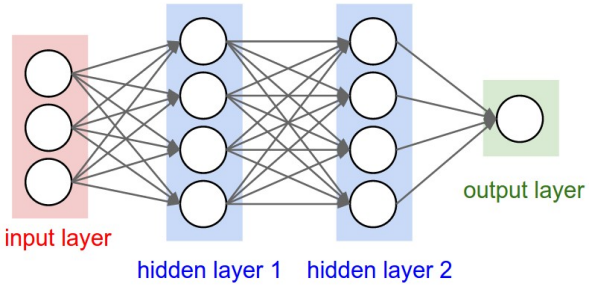


Figure 4.3: Structure of a simple Feed Forward Neural Network[2]

We denote by L, the number of layers and by $N^{(l)}$, the number of neurons in layer l. Let $l = 0$ be the input layer and $l = L - 1$ the output layer, we'll use w^l_{jk} to denote the weight of connection from k^{th} neuron in $(l-1)^{th}$ layer to j^{th} neuron in l^{th} layer. Similarly, b^l_j is the bias of j^{th} neuron in layer l. With this notation we can compute the activation a^l_j for the j^{th} neuron in l^{th} layer with the following formula

$$a^l_j = \sigma\left(\sum_k w^l_{jk} a^{l-1}_k + b^l_j\right) \tag{4.2}$$

Similarly, we can define a weight matrix W^l for weights in layer l, where the entry in j^{th} row and k^{th} column will be the element w^l_{jk} from Equation 4.2. In the same manner we can define a bias vector b^l where the values are biases of neurons in layer l. Lastly, a^l will be an activation vector whose components are activations a^l_j. The equation for an activation vector a^l can be then written as

$$a^l = \sigma\left(W^l a^{l-1} + b^l\right) \tag{4.3}$$

In each layer a non-linear transformation of the output from its previous layer is calculated. Hornik [25] proved that with the addition of hidden layers and a right set of weights and biases, FFNNs can approximate any Borel measurable function from one finite dimensional space to another at any desired degree of accuracy. This result is known as the universal approximation theorem. Unfortunately, finding those parameters is rather nontrivial.

Although [25] states that any function can be represented by a neural network with just one hidden layer and a sufficient number of units, [39] showed that this approach is usually inefficient and the same function can be represented by a much more compact deeper architecture.

4.2.1 Learning and Cost Functions

Our goal is to find the parameters $\theta = (W, b)$ based on a dataset of input and output vectors (x_n, t_n), which would minimize the difference between the output of the trained network and the real function we are trying to approximate. In other words, we need to minimize a cost function $C(\theta)$, that can be defined in several ways. Equation 4.4 displays one example, the mean squared error loss function.

$$C_{MSE}(\theta) = \frac{1}{2N} \sum_{n=1}^{N} \|y(x_n, \theta) - t_n\|^2 \tag{4.4}$$

When we use MSE as our loss function, the network can be trained using gradient-based optimization techniques [40], because every part of the network is constructed from differentiable operators. Therefore, in order to minimize the cost function and find the optimal weights W, we need to compute the gradient of the cost function with respect to the weights W, i.e.

$$\frac{\partial C_{MSE}}{\partial W} \tag{4.5}$$

and update the weights with step proportional to the negative of the gradient. This process is called backpropagation and is more thoroughly described in [41].

4.3 Convolutional Neural Network

Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers [5].

The name convolution comes from a mathematical operation called convolution. Convolution of a function f with function g is described as

$$(f * g)(t) = \int_{-\infty}^{\infty} f(x) \cdot g(t - x)\, \mathrm{d}x \tag{4.6}$$

$$(f * g)(t) = \sum_{x=-\infty}^{\infty} f(x) \cdot g(t - x) \tag{4.7}$$

in continuous and discrete cases, respectively. In literature, a convolution operation is usually denoted with an asterix. The first argument f is referred to as *input* and the second argument g as *kernel* of the convolution.

Often we want to use convolutions when we work with images. Then we have to work with a 3-dimensional input, two dimensions for width and height of the image and one dimension for the color channel. Formula 4.8 shows how to calculate convolution of n^{th} output channel of such input.

$$(I * K)(n, x, y) = \sum_{c=0}^{C} \sum_{p=\frac{-k}{2}}^{\frac{k}{2}} \sum_{q=\frac{-k}{2}}^{\frac{k}{2}} I(c, x+p, y+q) \cdot K(c, n, p, q) \tag{4.8}$$

Where $I(n, x, y)$ is the value of the n^{th} channel of a pixel at location (x, y), k is the kernel size and $K(c, n, p, q)$ represents the weights of the network.

Convolutional neural networks have several important properties which make them more suitable when dealing with images as compared to simple feed forward nets. These are sparse connectivity, parameter sharing and equivariant representations.

In traditional neural networks, every neuron interacts with every unit from its previous layer. In order to train the network, one has to perform costly matrix multiplications between layers. This leads to long learning times. On the other hand, convolutional networks leverage the correlation of nearby pixels in an image. They do so by using a kernel size k that is smaller than the size of the input image. Thanks to this sparse connectivity, we can reduce the model's memory requirements and the time it takes to perform a forward pass. Additionally, the weights of a filter applied at different locations are shared. So, instead of learning separate parameters for every part of an image, we simply learn one set. This further reduces the total number of parameters that we need to learn and allows the network to be equivariant to translation, meaning that if we move an object in the input several pixels to the left, its output will also move by the same amount. As an example, it is efficient to detect edges all over an image with a single set of parameters.

The first network which resembled modern CNNs was called Neocognitron introduced by Fukushima in 1980 [42]. In 1998, LeCunn et.al [3] came up with LeNet which was used in the field of character recognition. LeNet showed the strength and potential of CNNs and similar architectures are still being used at the moment.

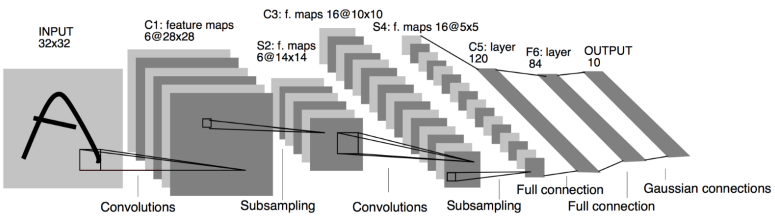


Figure 4.4: Architecture of LeNet-5 [3]

In 2014, Yamins [43] showed a similarity between the firing of neurons in human visual cortex and activations of convolutional nets, which made the researchers believe that we are on the right path with using CNNs for image recognition. Further breakthroughs continued with the ILSVRC *ImageNet* competition [44], where the main goal was to classify images with a training dataset containing an astounding 14+ million images. The first deep convolutional nets reached a test error of 16% [45]. Today's state-of-the-art convolutional network SENet has a test error 2.251% which outperforms an average person who can only score an error of 5.1% [46].

4.3.1 Pooling

Convolutional networks usually consist of convolutional and fully connected (FC) layers, which perform the high level reasoning. A typical convolutional layer has three stages – convolution stage, detector stage and pooling stage. In the first stage filters/kernels are applied in parallel to produce linear activations. These activations continue through a non-linear activation function (usually ReLU) in the detector stage. The result proceeds into a pooling layer. Pooling functions perform non-linear downsampling of the input. They do so by combining values of a bigger neighborhood of neurons into a single value that they pass to the next layer [8]. There are numerous pooling functions. Average pooling, L^2 norm pooling or max pooling, which gives the best results in practice [47]. As the name suggests, max pooling takes the maximum of neuron activations in its receptive field. An example can be seen in Figure 4.5.

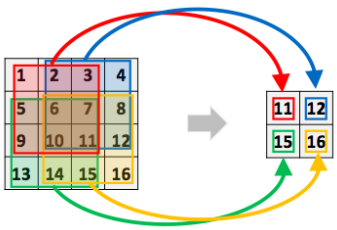


Figure 4.5: Example of max pooling with filter of size 3×3 and stride 2 [4, p. 17]

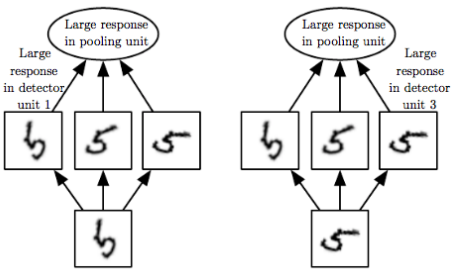


Figure 4.6: Example of learned invariances [5, p. 4]

Pooling layers allow the network to be invariant to small local translations, leveraging the fact that the pixel-precise location of a feature is much less important than its rough position relative to other features. For example, when detecting a bird in an image, we want to focus on the presence of wings, beak and other characteristics of a bird, but we care much less about the precise size and location of the aforementioned features.

4.3.2 Hyperparameters

Even though convolutional networks have less overall parametres to train than feed forward neural nets, they have more hyperparameters. We can fine-tune them to control the output size between different convolutional layers.

Depth When performing convolution we usually perform several convolution operations in parallel. Each filter learns to detect a different feature like edges or clusters of a single color.

Stride Stride is the number of pixels by which we move the filter after every step. With stride of size 1, we move the filter by one pixel at a time, which leads to overlapping of filters' receptive fields. With stride of size 2 or larger, the resulting output will be downsampled, which leads to faster training. Figure 4.5 depicts max pooling with a stride of size 2 and a kernel of size 3×3.

Zero padding Zero padding loosely means padding of the representation by zeroes on borders. When we do not incorporate zero padding, the representation naturally shrinks at each layer by one pixel less than is the size of the used filter. This can cause undesired shrinking in situations, where it is required for the input to preserve its size.

Chapter 5

The Controller

This chapter talks about the choice of model for the task of autonomous driving. An approach with a fully connected feed forward neural net is compared to convolutional networks. The last Section provides measured performance of these models and explains which one was chosen for future improvements.

5.1 The Goal of the Controller

As seen in the diagram 3.2, the task of the controller is to control the car's actuator based on the input received from its onboard camera. In other words, we try to find a mapping from camera images to steering commands. Such relationship may seem obvious for human eye, but is difficult to capture by a set of rules in a traditional programming paradigm. When we see a road that is curved, we know how much we should steer, but defining programmaticaly what a curve is, how sharp it is, how to find it in an image and what should be the correct driving command to drive through, is problematic. In a new case, we might encounter a different curve that was not described by our constraints and the program which uses a hand engineered set of rules, would fall apart. This is a good use case for using AI. Instead of describing the relationship between features in images and driving commands manually, we use supervised learning and let a neural network learn these features automatically.

As mentioned in Section 4.2, neural nets with at least one hidden layer and enough units can approximate any Borel measurable function from one finite dimensional space to another. In our case, we try to find a function that maps matrices of size $m \times n \times 3$ (input image) onto interval $(-1, 1)$ (steering angles, left $\rightarrow$ right). We can see in Figure 5.5, that when there are straight lanes on left and right side of the image, the steering angle is 0 degrees. Similarly, when pixels in upper left corner do not display any lanes, the car should probably steer right. I will explore several network architectures to capture this relationship as precisely as possible. The architecture with the best results will then be further improved in Chapter 6.

5.2 Fully Connected Network

As [27] and [48] shown, it is possible to train a simple feed forward neural network to perform steering based on visual input from camera images. Fully connected neural networks tend to have a lot of connections and parameters to tune compared to convolutional networks. We have to keep that in mind when designing the shape of the net. The larger the input will be, the more parameters will be required for the network to train, leading to larger datasets and possible over/underfitting if not enough data is collected.

Pomerleau [27] used input of size 30×32 resulting in an input layer with 960 units, although the final width of the first layer ended up having 1217 neurons, because extra data from a laser range finder was also used. On the other hand, Wang [48] used input images of size 320×120, which is considerably more (38400 input neurons). In order to reduce the input size, both authors decided to reduce the image dimensionality from 3 channels (RGB) to 1 channel. Wang opted to use grayscale images while Pomerleau used only the blue channel as it provides the highest contrast between road and non-road pixels.

5.2.1 Image Preparation

For my project, I tested input of size 50×20. Even though the GPU processing power has increased significantly in the past couple of years, in my case, there is enough data on the downscaled image that both the network and human observer should correctly classify the image. A good example of similar approach is CIFAR-100 [7], with image size of 32×32, on which modern neural nets can distinguish between 100 classes like dolphins, trucks or lobsters.

Each picture in my collected dataset goes through several steps of preprocessing. At first, irrelevant parts of the image are removed (such as upper part of the image that does not contain road lanes). As a second step, the image is rescaled to fit the neural network's input size. Lastly the image is converted to a grayscale. The visualization of the entire process can be seen in Figure 5.1.

(a) Original image (b) Cutting edges (c) Downscaling (d) Grayscale

Figure 5.1: Example of image preprocessing in training dataset

5.2.2 Network Architecture

I tried several FFNN architectures. The input layer is always the size of the input image which is 1000 neurons. Output layer is one *tanh* unit which outputs continuous values from the interval $(-1, 1)$, where -1 represents the leftmost angle that can be set in the RC car, $+1$ is the rightmost angle and 0 is for driving straight. ReLU was used as activation function for every neuron in hidden layers because of its superior performance compared to other activation functions [49]. Figure 5.2 depicts the flow of data from input to output.

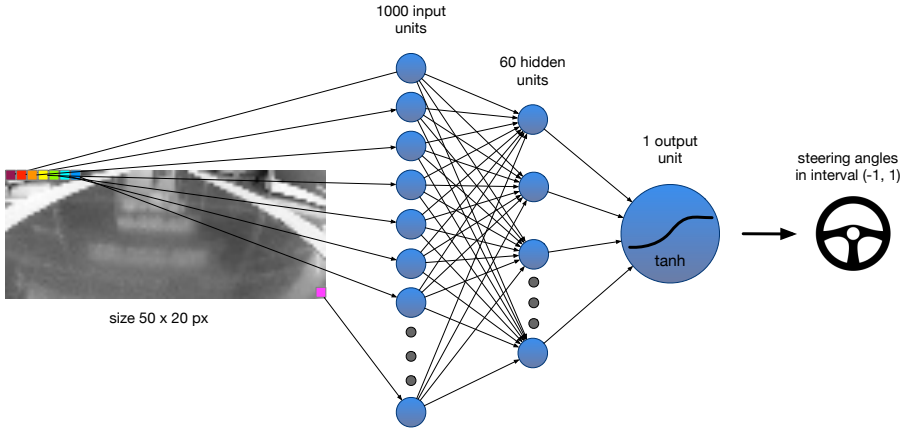
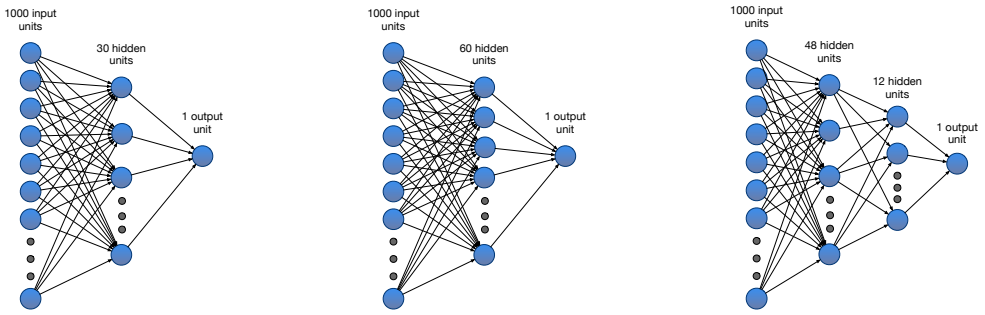


Figure 5.2: Flow of data with example architecture FFNN2

One more thing that I experimented with was the width and number of hidden layers. A good rule of thumb is to have fewer parameters than training cases. I tried 4 different architectures, first with a single hidden layer of 30 neurons, second with single layer of 60 neurons, third with 120 neurons and fourth with two hidden layer of sizes 48 and 12. Diagrams of tested architectures can be seen in Figure 5.3.



(a) Architecture 1 (b) Architecture 2 (c) Architecture 4

Figure 5.3: Tested fully connected architectures

5.3 Convolutional Networks

As I described in Section 4.3, convolutional neural networks introduce several key concepts which give them advantage in image classification. Sparse connectivity, parameter sharing and equivariant representations are one of the reasons why we have seen tremendous success in the field of image recognition in the past couple of years [8]. In this Section, I present several convolutional network architectures that I explored, when I was searching for an ideal model for my problem. At first, I tested a deeper, state-of-the-art convolutional network designed for autonomous steering of real cars. Then I compared its result to other convolutional networks of different structure.

5.3.1 PilotNet

Pilot net is a state-of-the-art convolutional net designed by NVIDIA and first presented in their *End to End Learning for Self Driving Cars* paper [29]. The network architecture is show in Figure 5.4. It consists of a normalization layer, 5 convolutional a 3 fully connected layers. Its input is a 3 channel RGB image of size 200×66. The first three convolutional layers use filters of size 5×5 and stride 2×2. Next two layers use stride 1×1 and filters with size 3×3 resulting in $64 \times (1 \times 18)$ output from the last convolution. The extracted features are then fed into 3 fully connected layers. The output layer of the network consists of one unit using *tanh* as activation function which maps the output on a continuous interval $(-1, 1)$. Altogether, there are around 250 000 trainable parameters in the network.

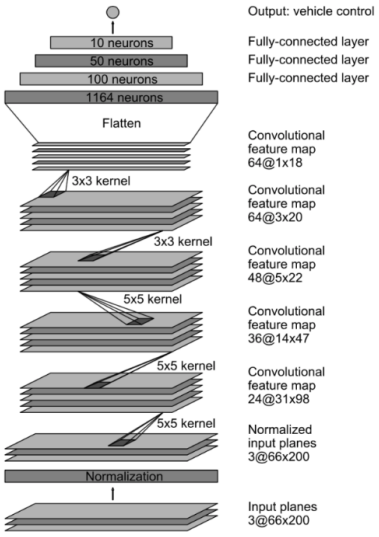


Figure 5.4: PilotNet architecture

With this architecture, NVIDIA was able to train its controller to drive autonomously 98% of the time on flat roads in Monmouth County, NJ with minimal traffic.

5.3.2 Custom CNN

Even though PilotNet was shown to be a successful solution to steering angle prediction in several reported cases, it is designed to work well on images from real world roads. In my case, I drove the car in an isolated environment using A4 papers as lane markers. These papers are in reality much bigger than lane markers on public roads. On the other hand, the part of floor outside of the test track has the same color as the road itself, giving me a small disadvantage against the real world scenario, where the road is made of asphalt and the surroundings is usually grass or at least differs in color, so the network only needs to stay in center of a black/grayish looking part of an image. The amount and variety of training data used by NVIDIA is also much higher than my dataset, making PilotNet's convolution structure too complex for my particular situation.

Therefore, I decided to experiment with other CNN models which may fit my data better. I performed an informed grid search for optimal hyperparameters using information about good convolutional net architectures ([45], [50]) to reduce the search space. I generated over 64 different models with varying amount of convolutional / fully connected layers, and filter and stride sizes. This space was further reduced to 11 candidates with 100000 to 1200000 parameters. The list of all explored architectures can be found in Table 5.1.

convolution filters	strides	FC layers	param	loss
16x3x3, 24x3x3, 48x3x3	2, 2, 2	500, 100, 25	860k	**0.0162**
16x3x3, 32x3x3, 64x3x3	2, 2, 1	100, 50, 10	730k	0.0227
16x3x3, 32x3x3, 64x3x3	3, 2, 1	100, 50, 10	300k	0.0207
16x3x3, 32x3x3, 64x3x3	3, 2, 1	200, 50, 10	570k	0.0172
16x5x5, 32x3x3, 64x2x2	2, 2, 1	100, 50, 10	860k	0.0223
16x5x5, 32x3x3, 64x2x2	3, 2, 1	100, 50, 10	290k	0.0182
24x5x5, 42x3x3, 64x2x2	2, 2, 1	100, 50, 10	870k	0.0222
16x3x3, 24x3x3, 36x2x2, 48x2x2	3, 2, 2, 1	500, 100, 25	230k	**0.0134**
16x3x3, 24x3x3, 36x2x2, 48x2x2	*3, 2, 2, 1*	*1024, 256, 32*	*1200k*	**0.0115**
24x3x3, 36x3x3, 48x2x2, 64x2x2	3, 2, 2, 1	200, 50, 10	130k	0.0178
24x3x3, 36x3x3, 48x2x2, 64x2x2	3, 2, 2, 1	500, 100, 25	300k	**0.0142**

Table 5.1: List of custom CNN architectures and their attributes - convolutional filters, strides, size of fully connected layers, total number of parameters and final validation loss. Best performing architecture highlighted (third row from bottom).

As the results imply, we can see that networks with larger fully connected part tend to perform better. Another valid observations is that lower number of filters is sufficient for good predictions. That can be explained by the fact that we mainly try to find edges between road and lanes. We don't try to detect complex objects such as faces or wheels that would require a more complex architecture. Shrinking the input with each convolutional layer and using the remaining parameter allowance for the fully connected layers turned out to be the best strategy for the network design, especially when bigger dropout probabilities were used during the training to prevent overfitting.

5.4 Model Comparison

The mean square error loss function was used to train all models. This is common for regression problems, because it punishes large deviations between predicted and recorded results. Given a feed forward neural net function $y(x, \theta)$, the mean squared error function can be described by formula 5.1, with t_n being the recorded value.

$$C_{MSE}(\theta) = \frac{1}{2N} \sum_{n=1}^{N} \|y(x_n, \theta) - t_n\|^2 \tag{5.1}$$

Adam [40], a gradient based learning method, was used for optimization. Equal levels of dropout (50%) were used to train each network. The only other form of regularization was early stopping, used mainly to reduce the extensive training times. If the network's validation loss has not improved at least by 0.005 in the last 20 epochs, the learning was stopped. Most networks converged within 40 to 80 epochs.

5.4.1 Used Dataset

The dataset was recorded through car's web interface (more in Sections 3.8, 3.6). It consists of 28 000 images that were sampled from driving video at 10 frames per second. A higher FPS would result in images that are too similar and would not provide any additional useful information. Each image has a corresponding steering angle saved in a separate `json` file.

Figure 5.5: Example images from training dataset

Before an image is fed into the network it is cropped that only the relevant part of the image is present. Anything above horizon is not important for the classification in any way and can instead introduce overfitting to objects

close to the track. This height of the cropped area was chosen by hand and can be seen as the space surrounded by red rectangles in Figure 5.5. This image is than downscaled to 100×33 for custom architectures. This size was chosen to save on training parameters while still maintaining good training possibilities as there is enough data to classify the image even through human eyes. The images are then normalized to have RGB pixel intensities between (0, 1) instead of (0, 255) in order to help with training.

The data was collected manually on two different routes in my lab. Both tracks measure between 15 and 20 meters. A small sketch of the routes is depicted in Figure 5.6. I drove on them in both directions several times in different light conditions combining variations of natural light, closed window shades, artificial yellow and white light, and combination of all mentioned above. The floor does reflect a lot of light which makes the training much more difficult, especially in situations when only yellow light is present, considering the lane markers are also yellow.

Figure 5.6: Sketch of routes used for training

The dataset also contains situations where the buggy is almost driving off the track. These cases are crucial for the network to learn how to recover from bad situations when it missclassifies and drives too close to the edge.

Angles in the dataset are well distributed with a slight bias towards going straight. The angle distributions can be seen in Figure 5.7.

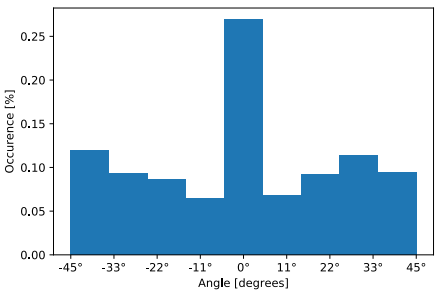


Figure 5.7: Distribution of angles in the recorded dataset

5.5 Measured Results

In this section, I briefly summarize the measured results and compare all aforementioned approaches. I used two metrics to evaluate the performance of an architecture. Those were validation loss and autonomy on on-track tests.

5.5.1 Validation Loss Tests

Figure 5.8 shows the evolving validation loss throughout training of all four tested fully connected architectures.

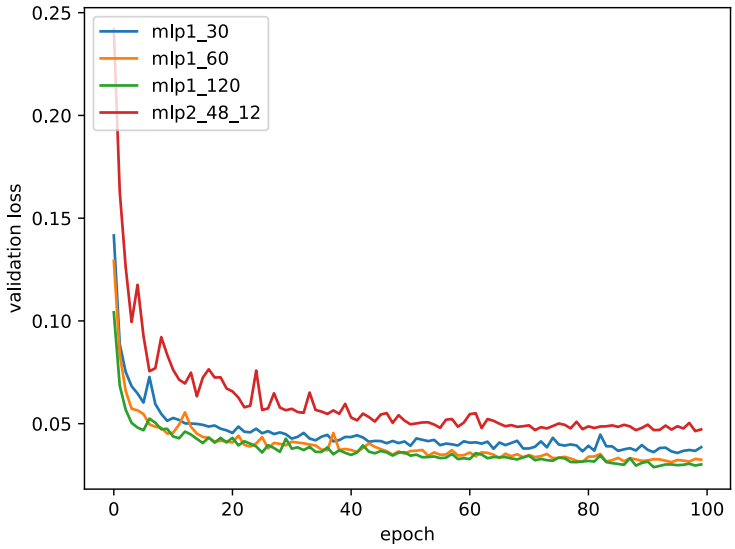


Figure 5.8: Comparison of validation loss of fully connected architectures. Blue, orange and green being networks with a single hidden layer of sizes 30, 60 and 120 respectively. The validation loss of network with two hidden layers (48 units and 12 units) is drawn in red.

As it can be seen in the graph, adding more hidden units does not seem to always increase performance. There is almost no difference between a single layer architecture with 60 and 120 units. Another important thing to notice is that spreading the neurons between two layers instead of one did not help and actually turned out to have much worse performance than using a single hidden layer. This proved the thesis that fully connected layers have difficulties with feature extraction from images and cannot overcome slight changes in input like rotations.

Finally, Figure 5.9 shows the comparison between a fully connected network (blue), NVIDIA's PilotNet (orange) and my best performing convNet architecture (green).

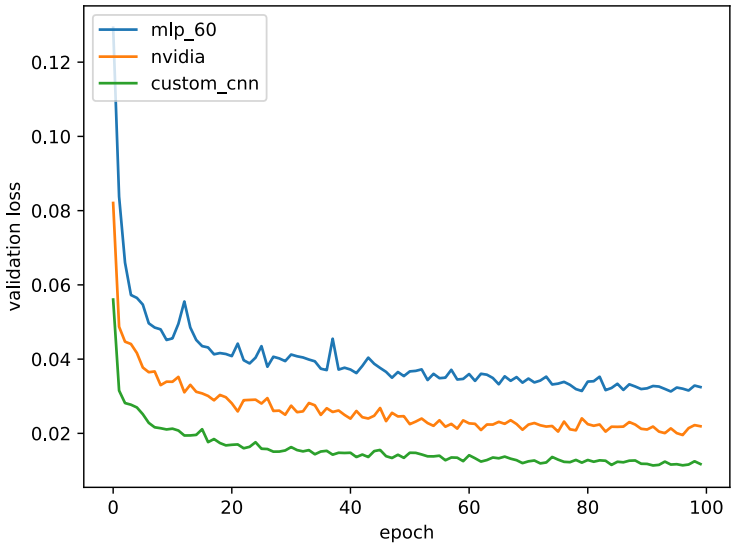


Figure 5.9: Comparison of validation loss between fully connected networks and CNNs. Blue is plain FFNN, PilotNet is in orange and my custom architecture in green.

As we can see, introducing convolutions helped to reduce the validation loss by almost a third. Interestingly, using an architecture from Section 5.3.2 helped to reduce the loss even more. This can be explained by the fact that a shallower network with fewer filters was used. The fully connected part was larger, but simultaneously big dropout probabilities (50%) helped to prevent overfitting of the decision-making part of the network.

5.5.2 On-track Tests

To measure the autonomy of the RC car, I used the same metric as was used in [29]. This metric calculates the percentage of time the car is able to drive without human intervention. Human intervention in this case means taking the RC car when it crosses the lane markings and putting it back in the middle of the track. Such intervention takes 5 seconds in my case. The overall percentage of autonomy is then calculated by formula 5.2.

$$autonomy = \left(1 - \frac{(number\ of\ interventions) \times 5}{total\ time\ [second]}\right) \times 100 \tag{5.2}$$

Figure 5.10 shows the results of on-track test on the 3 architectures from previous Sections.

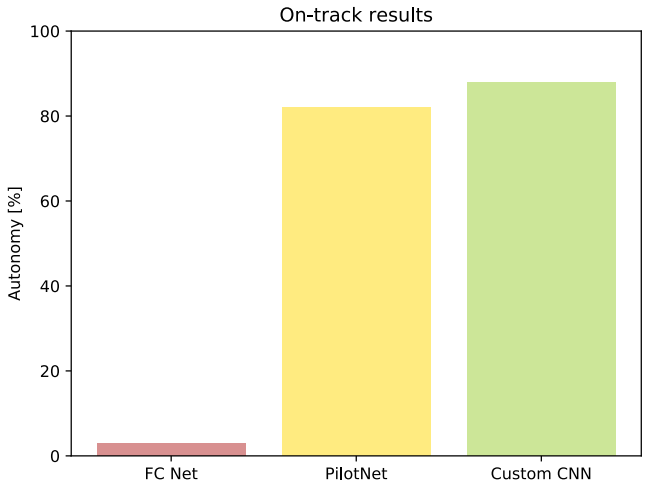


Figure 5.10: Compared on track performance. Feed forward neural network (red), NVIDIA's architecture (yellow) and my architecture (green).

All networks were tested after 30 training epochs, using batch size of 128 images. Even though the fully connected network shown relatively low validation loss, it was unable to keep on track for more than 6 seconds. Both PilotNet and my CNN architecture performed well. In 10 minutes of driving, PilotNet drove off track 22 times and therefore was autonomous 82% of the time. The best CNN architecture from Section 5.3.2 required intervention only 14 times in 10 minutes scoring the highest on this test with the score of 88%.

5.5.3 Summary

The overall best performing architecture is presented in Figure 5.11. It has shown the best results on both tests. In Chapter 6, I will focus on improving its performance further via regularization and other techniques.

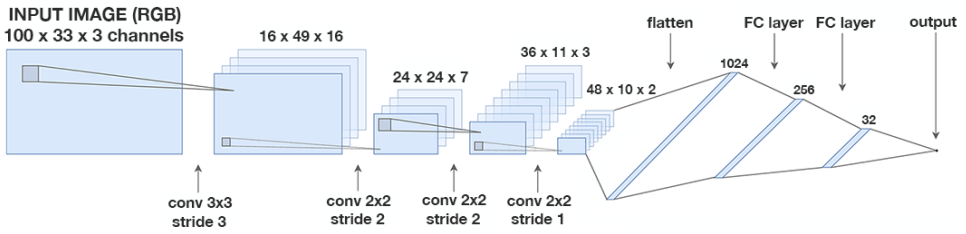
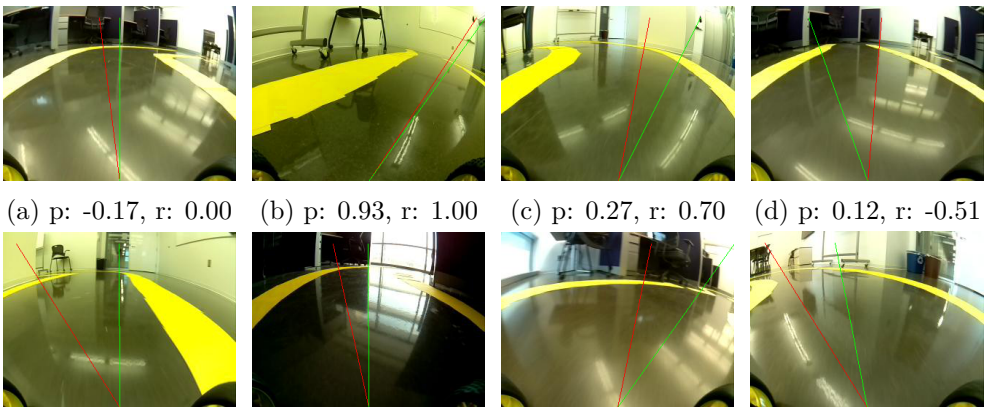


Figure 5.11: Winning CNN architecture

Some examples of its classification on the validation set can be seen in Figure 5.12.



(a) p: -0.17, r: 0.00 (b) p: 0.93, r: 1.00 (c) p: 0.27, r: 0.70 (d) p: 0.12, r: -0.51

(e) p: -0.88, r: 0.01 (f) p: 0.04, r: -0.32 (g) p: 0.28, r: 1.00 (h) p: -0.87, r: -0.27

Figure 5.12: Examples of predicted angle (p, red color) vs recorded angle (r, green color)

We can see that the network does a very good job in estimating the steering angle. Images *a* and *b* from Figure 5.12 are very close to the recorded value. Images *c*, *d* and *e* even perform a better job then I did when I was recording the dataset. On the other hand, there are still a lot of cases like *f*, *g* or *h*, when the predicted angle is not ideal. But the network's missclassification is not a big problem if it does not happen very often. The controller can recover from it by classifying new images correctly. Further, it can be seen in images *b* and *e* that the network learned how to behave in situations when the car is very close to the edge of the track.

Additionally, I would also like to point out few reasons why this network performed better than PilotNet and other architectures mentioned in Section 5.3.2. Arguably, the most crucial reason is the fact that my training dataset contains a lot of implicit noise. There are infinite ways how to drive through a curve and therefore the dataset contains a lot of similar images with different labels. This relationship is very hard to capture with a small network. One way to solve this is to train the network in simulation with a precalculated optimal path using trajectory planning algorithms [51]. For that I would have to implement a simulator with graphics similar to my real life scenario and use the mentioned planning algorithm. That would require tremendous amount of time and work unrelated to the thesis objective. Another option is to simply collect more training examples and use large enough network to try to accommodate for the existing noise. This is also what the tests shown in Section 5.3.2. Networks with larger fully connected part outperformed smaller nets when large dropout probabilities were used. To keep the number of parameters reasonable, it is possible to use fewer filters and convolution

layers. For my case it is not required to detect complex objects and layering too many convolutional layers on top of each other does not increase performance.

Chapter 6

Experiments and Testing

This chapter focuses on improving the chosen network's performance. Various approaches are tested such as collecting more data or different types of regularization. Section 6.3 discusses interpretability and visualizations techniques used to debug the model's performance. The last section 6.4 includes experiments measuring how many times per second is the system able to operate on a *Raspberry Pi* and what percentage of said time is actually taken by the network's inferring.

6.1 Collecting More Data

When trying to improve the system's performance and generalization, one of the best ways is to simply collect more data. In most situations, experimenting with different models and hyperparameters will lead to better results, but these computations are very costly. With adaptive learning rate, it takes approximately 4-6 hours to train the network on a *2013 2.6GHz i5* CPU and around 1-2 hours on *AWS p2.xlarge* instance using *NVIDIA Tesla K80* with `tensorflow` GPU acceleration. Then even a relatively small grid search within the hyperparameter space can become very expensive. In my case, when collecting training samples requires relatively little effort and time, it is the best idea to collect as much various data as possible. Specifically, my model had problems with sharper curves that the car has not seen before in the training dataset. Therefore, I decided to record more data of curves with varying curvature. I built two new tracks and created many variations of a single curve with different degrees of curvature. With the new data, the dataset should contain all most common scenarios.

Overall, additional 22 000 images were captured. The updated dataset now counts over 50 thousand images. The data was further enlarged through data augmentation techniques described in Section 6.2.1.

As expected, the resulting dataset was slightly unbalanced with a lot of angles close to 0. This could introduce bias towards driving straight. To avoid

this, I divided steering angles into bins and performed designated random sampling in order to make the dataset resemble normal distribution.

6.2 Regularization

Regularization is used to reduce overfitting [52] in machine learning models. Goodfellow et al. [5] suggest that the best approach when creating a machine learning model is to choose a large model that is capable of fitting the data and then increase its generalization via regularization. I decided to follow this approach and chose (in section 5.5.3) a network with a lot of parameters. This section aims to describe popular regularization approaches and what was their impact on training times, model performance and generalization.

6.2.1 Dataset Augmentation

One way to get more data is to generate fake examples and add them to the training set. This may sound odd, but it is actually very helpful. Images are high dimensional and can have many variations which are easy to simulate. By artificially adding reflections and shadows to the training dataset, we can simulate situations that are very hard to record manually. Enriching the dataset with this new data is not only going to make it larger, but it will also force the network to learn how to deal with unusual situations. We can think of that as a form of regularization. It will lead to both increased robustness of the model and better generalization.

For my task I have decided to use several augmentation techniques.

Flipping the image horizontally This is a very easy way to double the size of training data. By simply flipping the images horizontally (and multiplying the steering angle accordingly by -1) we get new examples that we do not have to gather manually.

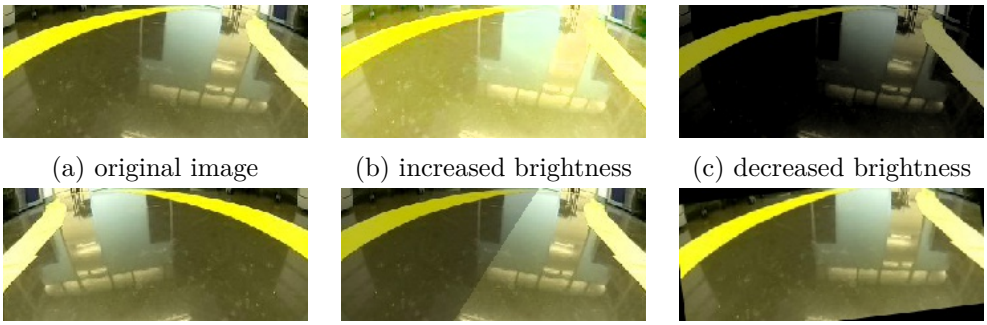
Changing brightness Slight changes in brightness and color of the image simulate different light conditions. This is implemented by randomly increasing or decreasing V channel in the HSV color model of an image.

Adding artificial shadows Random shadows were generated and added to the image to make the model invariant to actual shadows on the track.

Rotating the image The RC buggy has suspensions on both rear and front wheels. The camera is connected tightly to the chassis, but it can still wiggle. In order to simulate this behavior, slight image rotations were added to some samples from the dataset.

Examples of augmented images can be seen in Figure 6.1. The source code of mentioned methods is in file `utils.py`. The data is fed to the network through a *Keras generator*, which computes the augmentation on CPU

in real time while the GPU trains the network via backpropagation. For every batch, data is randomly sampled from the training dataset and different augmentations are applied with certain probabilities.



(a) original image (b) increased brightness (c) decreased brightness

(d) horizontal flip (e) random shadow (f) rotation

Figure 6.1: Augmentation examples

I compared three levels of augmentation – light, moderate and heavy. For light settings I only used image flipping and slight brightness changes. Moderate augmentations included more drastic brightness adjustments and smaller rotations. In the heavy case, I included shadows and amplified the effect of previous augmentations. Table 6.1 displays the effect of augmentation on validation loss during training.

epoch	**none**	**light**	**moderate**	**heavy**
10	0.0187	0.0228	0.0293	0.0325
20	0.0139	0.0176	0.0216	0.0236
50	**0.0104**	0.0126	0.0159	0.0177
100	0.0098	**0.0106**	0.0131	0.0146
200	0.0095	0.0098	0.0117	0.0116

Table 6.1: Effects of dataset augmentation on validation loss. The numbers in the table are the lowest recorded validation losses up to a specified epoch.

With the newly collected dataset, the lowest achievable validation loss seems to be around 0.009. We can see that the bigger the changes in augmentation, the longer it takes for the model to converge. Namely, the model which was trained without any augmentations was able to converge within 20 - 50 epochs. The model trained with light augmentations took almost twice that time to converge to the same validation loss. Models trained with heavy augmentations took the longest to train. They also performed worse in on-track tests than models trained with less invasive augmentation techniques. That can be explained by the fact that heavy augmentations alter the input image dramatically and it can be very difficult for the model to pick up these drastic

changes. The positive effect of the newly collected, rebalanced and augmented dataset on on-track performance is further described in the following Section 6.2.2.

6.2.2 Early Stopping and its Effect on On-track Performance

When training a neural network, one's goal is to find a configuration of weights that leads to the smallest possible generalization error. However, all standard deep neural network architectures are prone to overfitting [53] given enough time to train. Stopping the training early, before the network starts to overfit, is a widely used method to combat this problem. Shorter training times help to avoid modeling the noise in training data, leading to networks with smoother decision boundaries. Besides that, early stopping heavily reduces the required training time, making it a very efficient and highly popular form of regularization.

This section presents the results of early stopping on the on-track performance of used network. One of the goals was to find approximately how long it would take for the model to learn to drive reasonably and when would the training start to have diminishing returns. Figure 6.2 shows the relationship between training time (epochs) and the autonomy of two models, which were trained using zero and light data augmentation (sec. 6.2.1), batch size of 128, 0.25 dropout probability in fully connected layers and 0 dropout probability in convolution layers. Other forms of regularization such as L1 or L2 regularization were not used, as they did not provide any extra benefits compared to using only dropout.

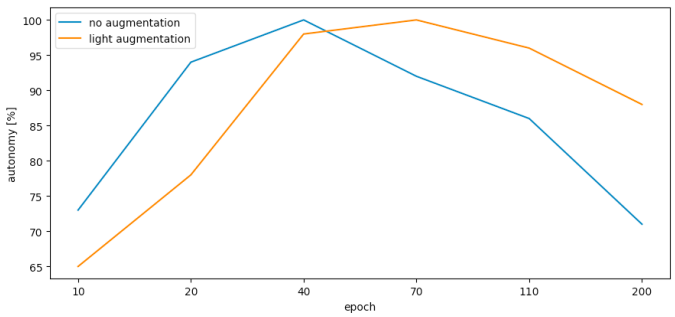


Figure 6.2: Effect of early stopping on car autonomy. Blue is the model trained without any augmentation, orange was trained with light settings.

As we can see, both models start to drive relatively well after only 20 epochs of training. At this time, the validation loss is around 0.015-0.02 for this particular scenario. Interestingly, models, with the same validation loss, that were trained on previous dataset performed worse by approximately 15%. That can be explained by the different sizes of training datasets. It is

much easier to overfit a smaller dataset than a larger one. Then even a small validation loss may result in sub-optimal autonomy on on-track tests. We can also see, that the model with stronger regularization measures takes longer to converge, but once it does, it is less prone to overfitting and performs better on average in the long run.

Around 60th epoch, the second model seems to achieve the best performance. It is able to run indefinitely on an ellipse shaped track which it has never seen before. It is also able drive on all test tracks in low speeds without ever driving off the track. When the speed is increased it starts to make mistakes in very sharp curves as the network did not learn how to move to the side of track opposite of the direction of the curve, which would give the car extra time to drive through. On the other hand, the system is able to run in 20Hz and has very fast reactions compared to humans. The fast reaction times make up for the inability to plan ahead. In some sharp curves, the car is able to get through in high speed where many drivers would fail as timing in crucial in such situations.

With longer training times, the car starts to make more mistakes. It is still able to run well on most tracks, but sometimes the network drastically missclassifies. When this happens, the wheels usually flick to the other side. Even though the next frame may be classified correctly, the time it takes for the wheels to get from one side to the other and back is too long and causes the car to drive off track, especially when higher speeds are used. This behavior starts to prevail after about 70 epochs (for the model with no augmentation), when the validation loss approaches 0.011, which is roughly equal to 85% of the lowest achievable loss.

One possible way to battle the unexpected wheel movement is to track the last couple of steering angles. Then when a drastic change occurs, we would delay steering until it is confirmed by several following classifications. This could backfire in situations when the drastic change in steering angle is actually required. But more importantly it would introduce hand engineered rules into the driving process, which I was trying to avoid from the beginning. Another possibility is to use a model with two output neurons, where one would be trained to steer left and the other to steer right. The final steering output would then be their arithmetic mean. This approach also helps the network to drive straight better. My original model uses `tanh` for the output unit. The problem is, that `tanh` is the steepest around point $[0, 0]$, making it very hard for the network to actually predict a perfect 0. That would become an issue for a real car, but my RC car is not even able to drive perfectly straight when I control it manually. Fortunately, the network knows how to get back to the center of the track, which makes up for the slight imperfections and makes the final motion look natural. To solve the flicking, I tried to implement and train an identical architecture with two output neurons. It performed equally well, but I have no exact mean to compare the driving paths of the original and the new architecture, because the optimal path

is not known in my situation (see the discussion about path planning and simulator at the end of Section 5.5.3). Unfortunately, even the new model, when trained long enough, still showed signs of undesired wheel movement. The only remaining explanation is that both models overfit when given enough time to train and missclassification happened, because the network made its decision based on pixels in the image unrelated to actual steering. That could be reflections, lights or other objects in the testing environment. Section 6.3 describes how to detect such situations and Figure 6.11 shows a concrete example of missclassification caused by an overfitted model.

6.2.3 Dropout

Dropout [54] is one of the simplest and yet most effective regularization techniques. The key idea is to randomly omit non-output units of the underlying base network during training. By doing so, we train an ensemble of thinned subnetworks which would later cooperate on the final classification, leveraging the benefits of an ensemble model. The final prediction is approximated by simply using the whole network at the test time. This process reduces overfitting by preventing the hidden units from co-adapting.

I designed and ran a set of experiments to detect the effect of dropout in convolutional layers, fully connected layers and its combination. For the purpose of this experiment, all convolutional layers get the same dropout probability, the same is done in fully connected layers. The results of the experiment can be seen in Figure 6.3.

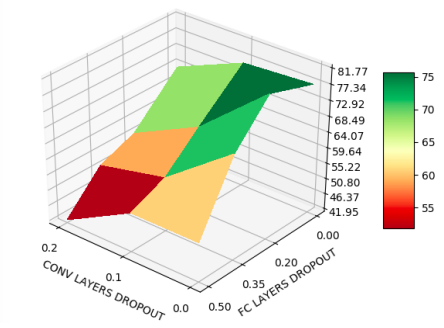


Figure 6.3: Effects of dropout on validation loss

Adding dropout to convolutional layers always seems to worsen the validation error of the network. This can be explained by the small number of connections between these layers. Adding more dropout then makes it difficult to learn weights that perform solid feature extraction. On the other hand, fully connected layers are more prone to overfitting. Here we can see that adding dropout of 0.2 probability yields the best result. Using too much dropout lead to very slow convergence. Using no dropout on the other hand resulted in expected overfitting.

6.3 Visualization and Interpretability

While deep networks achieve great results on a wide variety of computer vision tasks, the inability to decompose them into smaller, intuitive and understandable segments makes them very hard to interpret. One often looks at a trained neural network like on a black box, because there are no implicitly written rules describing how the network infers its output. By adding more layers and using deeper models, we sacrifice interpretability for greater performance and robustness. So when the network fails to perform well, it is often very problematic to find out why. There could be multiple reasons — not enough data, too much noise in the data, the network can underfit, overfit or the task might be too complex for the specific network architecture.

This problem naturally calls for a solution as it is highly unsatisfactory to not be able to identify, why our model performs poorly. In this section I describe several popular visualization techniques that I used to learn more about the inner structure of my model.

6.3.1 Visualizing Filters

The first thing that I wanted to examine were filters. They tend to be most interpretable on the first convolution layer that interacts directly with the input image data. Filters in this layer learn how to detect edges or changes in color. Filters in deeper layers combine information of previous neurons to detect corners and other more structured objects.

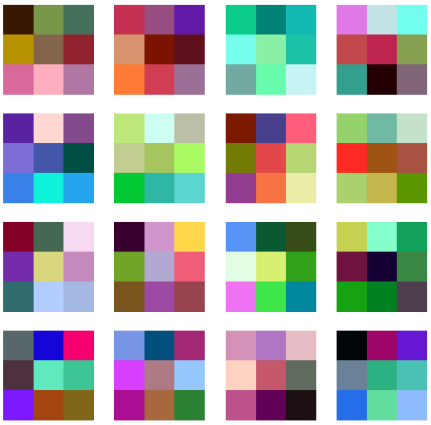


Figure 6.4: Colored version without interpolation

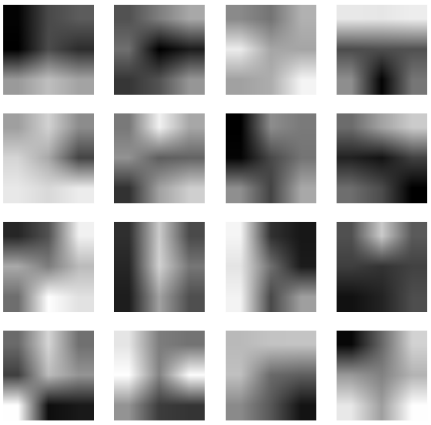


Figure 6.5: Grayscale version of green channel with bilinear interpolation

Figure 6.6: Visualization of trained filter in first convolutional layer.

Visualizing the weights is useful, because a properly trained network usually has smooth filters without any noisy patterns [2]. The noise could indicate

either overfitting or a network that was not trained long enough. As Figure 6.6 shows, most of the sixteen filters of the first convolution layer look relatively smooth. We can see both color and edge detecting filters that the network was able to pick up during training.

6.3.2 Visualizing Activations

Another straight-forward technique is to show the layer activations during forward pass. We can obtain them by multiplying the learned filters with input data and applying the activation function, in this case ReLU. What we get is a set of features that is further passed to the next layer, which repeats the same process. During the pass we can extract these activations and display them to see how each filter modifies the input image to extract features that are important for correct classification.

Figures 6.7 and 6.8 display activations of first two convolutional layers on images from the dataset.

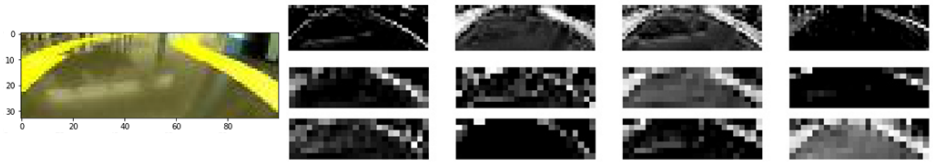


Figure 6.7: Layer activations for steering right. Images in the first row are activations of the first layer, the remaining rows show activations of the second layer.

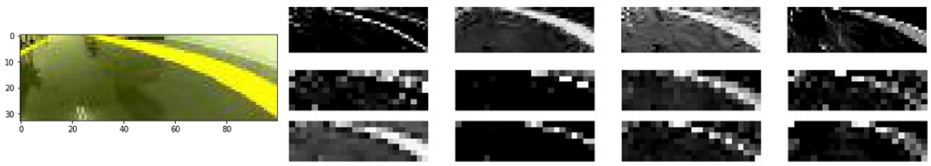


Figure 6.8: Layer activations for steering left. Images in the first row are activations of the first layer, the remaining rows show activations of the second layer.

The four images in the first row correspond to the first layer and are of size 16×49. The remaining eight images are examples of second layer activations with size 7×24. The activations clearly show that the network learned to detect lane marks. It does so by having larger activations (white color) for parts of the image related to the edges of the track. The top left image was created by a filter that probably learned to detect color transitions from yellow to black (edge between the track and lane marks).

6.3.3 Occlusion Maps

When evaluating a convNet, one criterion is whether the model is truly identifying the real location of important features in the image, or if it is making decisions based on other unrelated regions. In my case, the goal is to make the model detect mostly lane marks.

We can find the regions of an image that the model relies on the most by systematically occluding different portions of the input image by black window and monitoring changes in the regressed angle [55]. If the output changes dramatically, the occluded window contains information that is important for the steering decision. Formally, we can describe such binary occlusion map O with following equation:

$$O_{i,j} = \begin{cases} 0, & \text{if } abs(\hat{y}_{i,j} - y) > \epsilon \\ 1, & \text{otherwise} \end{cases} \tag{6.1}$$

Where y is the predicted angle without any occlusion and $\hat{y}_{i,j}$ is the regressed angle when the center of the occluding rectangle is placed at location (i, j) of the input image.

This process is relatively straight-forward and does not require any knowledge of the network's architecture. In fact, we only use the network's output. The major downside of this technique is its computational cost. To detect important regions in the image we need to create numerous sliding windows and for each window we need to perform a forward pass and record the output. The inserted black rectangles could also introduce new unwanted features to the image that may alter its classification.

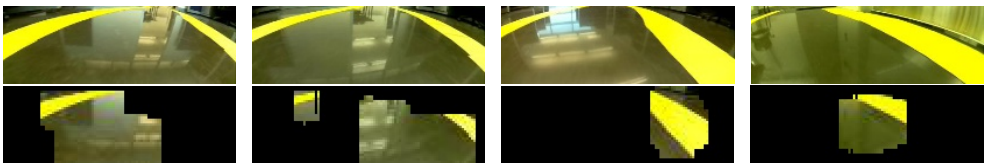


Figure 6.9: Examples of occlusion maps. Top row contains the original images, while the bottom row shows corresponding occlusion maps. Left two samples utilize sliding windows of size 15×15, the right two 7×7.

Figure 6.9 shows occlusion maps of four images from the recorded dataset. The top row shows original images and the bottom row displays occlusion maps on top of the original images. The parts of an image that do not largely affect the regressed angle are covered in black, while the rest is visible. We can see that some parts of the image that contain lane marks are chosen as important but some are not. Similarly, the network could also leverage the central part of the image in a way, that if it contains a large continuous part of the track and no lane marks, the probable decision is to drive straight. Another drawback

of this method is, that it does not tell us anything about why were those parts chosen. For more insight into that, I decided to implement grad-CAM.

6.3.4 Grad-CAM

Grad-CAM [6] is an extension of original work of Zhou *et al* [56], that leveraged global average pooling (GAP) layer proposed in [57] to enable CNNs to have localization ability despite only being trained on image-level labels without any localization information. Both CAM and Grad-CAM do so by visualizing the importance of each pixel in the overall inferring process. Grad-CAM is also class-discriminative, meaning that the map produced for class A highlights only class A regions of the image. That cannot be said about other visualization techniques as Deconvolution [55] or Guided Backpropagation [58].

With CAM, one would take output of the last convolutional layer, make a spatial average of that via global average pooling and feed it into a softmax for classification. Large weighs of the softmax imply important features, which are then multiplied by the corresponding convolution output. Unfortunately this process requires the network to have a specific architecture. It can only contain conv layers followed by a global average pooling layer. Viable approach is to remove all fully connected layers from my pretrained model, append GAP and softmax, freeze the weights of trained convolution layers and fine-tune just the softmax weights. But by doing so, I would completely ignore the fully connected layers, which is not ideal.

Grad-CAM, introduced in March 2017, generalizes CAM and is applicable to most CNN model families, because it does not require any change in the network architecture. It does so by combining feature maps and the gradient information that flows into the last convolution layer. As displayed in Figure 6.10, to calculate the class c discriminative localization map $L^c_{Grad-CAM} \in \mathbb{R}^{w \times h}$ of width w and height h, we need to compute $\frac{\partial y^c}{\partial A^k}$ – the gradient score y^c (before softmax is applied) with respect to feature maps A^k of a convolutional layer [6]. The gradients are then average-pooled to get neuron importance weights α^c_k:

$$\alpha^c_k = \frac{1}{Z} \sum_i \sum_j \frac{\partial y^c}{\partial A^k_{ij}} \tag{6.2}$$

The weights α^c_k represent importance of the k^{th} feature map for the target class c. The final localization map is then calculated as a weighted linear combination of the weights and feature maps with following formula:

$$L^c_{Grad-CAM} = ReLU\left(\sum_k \alpha^c_k A^k\right) \tag{6.3}$$

The ReLU is applied, because we only look for features that have a positive impact on the class c, meaning the increase of their intensity also increases y^c.

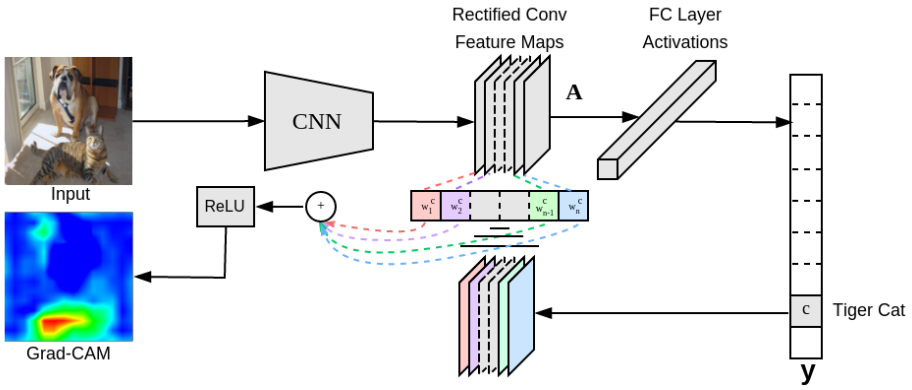


Figure 6.10: The process of computing Grad-CAM for an image [6].

To apply this procedure to steering angle regression, I had to include few modifications. High gradients do not contribute to any class, but are instead related to steering right and negative gradients contribute to steering left. To capture this relation, I divided the steering angles into three brackets (-1, -0.2), (-0.2, 0.2) and (0.2, 1) (left, straight, right) and implemented custom loss function similar to [59]. Results of Grad-CAM localization maps can be seen in Figure 6.11.

(a) Localization maps of second convolutional layer.

(b) Localization maps of last convolutional layer.

(c) Localization maps produced by overfitted models.

Figure 6.11: Examples Grad-CAM localization maps on recorded images.

The top two rows of images were obtained from the second and last convolutional layer of the best performing model, that was trained with light data augmentations, 0.2 dropout in FC layers and stopped early at 0.0015 validation loss. We can see that parts of the image belonging to lane marks have high activations. It is more visible in the second row, where big chunks

of red and yellow cover most of the lanes while unrelated parts remain blue. This network display correct behavior and we can be more sure that it is not overfitting our data.

On the other hand, the third row consists of localization maps calculated with a model that used no augmentation or regularization and was let to train for over 400 epochs. We can see in the left and right images of the bottom row, that the network made its classification based on objects that are not related to steering. On the right image it is a floor reflection of a chair, and on the left legs of a whiteboard. This would not be a problem, if the car was to drive only on a single track, but if we were to move the car somewhere else, it would not drive correctly.

6.4 Testing the Pi's Performance

Lastly, I measured the performance of *Raspberry Pi* and its suitability for the task. Since the final neural network architecture is quite complex, we could encounter delays in reaction times of the system. Even though the *Pi* is relatively small, it still has a lot of computing power. It is equipped with Broadcom BCM2837 SoC, a quad-core ARM Cortex-A53 running at 1.2GHz, 1GB of RAM and a Broadcom VideoCore IV. Unfortunately *tensorflow* does not support GPU acceleration for *Raspberry Pi*, so I could not leverage that opportunity.

The tests were run on Raspbian [60] with linux kernel 4.9.5, Python 3.6 and tensorflow 1.1. The system was running for 5 minutes and the Table 6.2 displays the measured results.

	min [ms]	**max [ms]**	**avg [ms]**
image prep.	2.8	4.5	3.4
NN infering	11	43	21
total loop time	18	55	26

Table 6.2: Measured times (minimum, maximum and average case) for a single run loop.

We can see, that one decision loop takes approximately 26ms to execute. That includes everything from capturing the image from camera, preprocessing it, feeding it into the network, translating the network output into car commands and then passing them to the actuator. To reduce the loop's execution time, the images are captured on a separate thread in order to not block the main thread. The web interface also runs on its own thread, so the biggest bottleneck is the network's inferring, which takes up around 81% of the execution time. The system is then able to run safely on 10, 20 and 30Hz.

I observed, that even at 10Hz, the car is able to run smoothly on moderate speeds. NVIDIA's DAVE2 [29] system runs at 30Hz, but their hardware is much more powerful. Having equivalent performance as their system indicates, that the *Raspberry Pi* is sufficient for this task and reducing the loop time further is meaningless. On the other hand it is possible to run the system on a less powerful computer, to save on costs and battery. But then we would lose the ability to use tensorflow, so the program would have to be largely rewritten to work without it.

Conclusion

In this thesis, I described how to use deep convolutional networks to control a prototype of a simplified self-driving RC car. I built the vehicle from scratch using custom hardware parts. Then I designed a multithreaded system that controls it and provided a light web and command line interface which users can use to interact with the car. Lastly, I equipped the car with autopilot controller based on state-of-the-art convolutional neural network architecture.

- The first chapter focused on the current state of autonomous driving and approaches that are used in the industry. I mentioned their pros and cons and explained the reasons for choosing an end-to-end training approach.
- Chapters two and three described how to actually create such car. What are the hardware requirements and how to write software, that will be able to control it.
- In the fifth chapter, the Controller, I came up with various neural network architectures that could perform well on the task of autonomous driving. I measured their performance and chose the best for additional improvements.
- The last part of the thesis consisted of several experiments aimed to improve and validate the car's autonomy. Enriching the training dataset and using proper forms of regularization turned out to yield the best results.

The system learned to steer autonomously when running on fixed speed. It can drive on simplified flat roads without traffic only using camera as its input. In the end, the car was able to drive indefinitely on a circular loop and a set of three flat tracks that it had never seen before. The biggest challenges I had to face were the absence of driving simulator and the problem of overfitting.

Future work

Even though the car works well in the lab and does not drive off track, there is still a lot of room for improvement.

- The path it sometimes takes is not ideal. That is caused mainly by the implicit noise in the training data, because I was unable to collect a dataset with optimal paths. Section 5.5.3 describes this problem in more detail. One option would be to create a driving simulator and implement path planning algorithms to gather a better driving dataset. That would also simplify the mapping between input images and steering angles, allowing for smaller networks with fewer parameters. Another option might be to use evolution to force the car to take better paths resulting in an increased average speed.

- The system cannot regulate speed, because it was only trained on steering data. When I started driving, I realized that there is no information about the actual speed of the car. The only input I had was the list of buttons I was pressing whilst collecting the data. Buying a speed sensor, attaching it to the *Pi* and training a new network with two outputs would solve this.

- Because driving is a continuous task, we could use recurrent neural networks like LSTM to capture the sequential relation between neighboring images. Then in situations where the camera is saturated from looking directly in a source of light, the system would use information from previous frames to estimate the decision for the more difficult frame. Changes in speed in the preceding frames could also indicate that the car is approaching a maneuver. The network could learn these relationships and improve its driving.

- To make the system more robust, a much larger dataset of different track types and different road conditions would need to be collected. This could be an interesting use-case for generative adversarial networks [61]. The driving task could also become harder by introducing intersections, signs, traffic or pedestrians. That would require a modular approach with specific parts designed for object detection. Ultrasonic sensors and radar could then be used to gather more complex information about the car's surroundings.

Bibliography

[1] Motors, T. Tesla Autopilot. 2017, [Online]. Available from: https://www.tesla.com/autopilot

[2] Karpathy, A. Stanford CS231n - Convolutional Neural Networks for Visual Recognition [Online]. University Lecture, January 2018. Available from: http://cs231n.github.io/neural-networks-1/

[3] LeCun, Y.; Bottou, L.; et al. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, volume 86, no. 11, November 1998: pp. 2278–2324.

[4] Gudi, A. *Recognizing Semantic Features in Faces using Deep Learning*. Master's thesis, University of Amsterdam, 9 2014.

[5] Goodfellow, I.; Bengio, Y.; et al. *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[6] Selvaraju, R. R.; Das, A.; et al. Grad-CAM: Why did you say that? Visual Explanations from Deep Networks via Gradient-based Localization. *CoRR*, volume abs/1610.02391, 2016, 1610.02391. Available from: http://arxiv.org/abs/1610.02391

[7] Krizhevsky, A. Learning Multiple Layers of Features from Tiny Images. 05 2012.

[8] Krizhevsky, A.; Sutskever, I.; et al. ImageNet Classification with Deep Convolutional Neural Networks. *Commun. ACM*, volume 60, no. 6, May 2017: pp. 84–90, ISSN 0001-0782, doi:10.1145/3065386. Available from: http://doi.acm.org/10.1145/3065386

[9] He, K.; Zhang, X.; et al. Deep Residual Learning for Image Recognition. *CoRR*, volume abs/1512.03385, 2015, 1512.03385. Available from: http://arxiv.org/abs/1512.03385

[10] Vinyals, O.; Toshev, A.; et al. Show and Tell: A Neural Image Caption Generator. *CoRR*, volume abs/1411.4555, 2014, 1411.4555. Available from: http://arxiv.org/abs/1411.4555

[11] Fang, H.; Gupta, S.; et al. From Captions to Visual Concepts and Back. *CoRR*, volume abs/1411.4952, 2014, 1411.4952. Available from: http://arxiv.org/abs/1411.4952

[12] Redmon, J.; Divvala, S. K.; et al. You Only Look Once: Unified, Real-Time Object Detection. *CoRR*, volume abs/1506.02640, 2015, 1506.02640. Available from: http://arxiv.org/abs/1506.02640

[13] Badrinarayanan, V.; Kendall, A.; et al. SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation. *CoRR*, volume abs/1511.00561, 2015, 1511.00561. Available from: http://arxiv.org/abs/1511.00561

[14] Long, J.; Shelhamer, E.; et al. Fully Convolutional Networks for Semantic Segmentation. *CoRR*, volume abs/1411.4038, 2014, 1411.4038. Available from: http://arxiv.org/abs/1411.4038

[15] Wu, Y.; Schuster, M.; et al. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *CoRR*, volume abs/1609.08144, 2016, 1609.08144. Available from: http://arxiv.org/abs/1609.08144

[16] Esteva, A.; Kuprel, B.; et al. Dermatologist-level classification of skin cancer with deep neural networks. *Nature*, volume 542, Jan. 2017: pp. 115–. Available from: http://dx.doi.org/10.1038/nature21056

[17] Jung, I.-S.; Thapa, D.; et al. Neural Network Based Algorithms for Diagnosis and Classification of Breast Cancer Tumor. In *Computational Intelligence and Security*, edited by Y. Hao; J. Liu; Y. Wang; Y.-m. Cheung; H. Yin; L. Jiao; J. Ma; Y.-C. Jiao, Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, ISBN 978-3-540-31599-5, pp. 107–114.

[18] of Transportation, U. D. Automated Driving Systems - A Vision for Safety. September 2017, [Online; posted September-2017]. Available from: https://www.nhtsa.gov/sites/nhtsa.dot.gov/files/documents/13069a-ads2.0_090617_v9a_tag.pdf

[19] LLC, W. Waymo Self-Driving Car Project. 2017, [Online]. Available from: https://waymo.com

[20] Waymo. Waymo's fully self-driving vehicles are here. November 2017, [Online; posted 7-November-2017]. Available from: https://medium.com/waymo/with-waymo-in-the-drivers-seat-

fully-self-driving-vehicles-can-transform-the-way-we-get-around-75e9622e829a

[21] Levine, W. S. *The Control Handbook, (Three Volume Set) (Electrical Engineering Handbook)*. Boca Raton, FL, USA: CRC Press, Inc., second edition, 2010, ISBN 1420073664, 9781420073669.

[22] Chong, G.; Li, Y. Trajectory Controller Network and Its Design Automation Through Evolutionary Computing. 01 2000: pp. 139–146.

[23] Chandni, C.; V V, S.; et al. Vision based closed loop pid controller design and implementation for autonomous car. 09 2017.

[24] Zhao, P.; Chen, J.; et al. Design of a Control System for an Autonomous Vehicle Based on Adaptive-PID. volume 9, 07 2012: p. 1.

[25] Hornik, K.; Stinchcombe, M.; et al. Multilayer Feedforward Networks Are Universal Approximators. *Neural Netw.*, volume 2, no. 5, July 1989: pp. 359–366, ISSN 0893-6080, doi:10.1016/0893-6080(89)90020-8. Available from: http://dx.doi.org/10.1016/0893-6080(89)90020-8

[26] Lucas, S.; Kotas, G. NVidia's GPU Microarchitectures. University Lecture, Fall 2017. Available from: http://meseec.ce.rit.edu/551-projects/fall2017/1-5.pdf

[27] Pomerleau, D. A. Alvinn: An autonomous land vehicle in a neural network. In *Advances in neural information processing systems*, 1989, pp. 305–313.

[28] Muller, U.; Ben, J.; et al. Off-road obstacle avoidance through end-to-end learning. In *Advances in neural information processing systems*, 2006, pp. 739–746.

[29] Bojarski, M.; Del Testa, D.; et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[30] Xu, H.; Gao, Y.; et al. End-to-end learning of driving models from large-scale video datasets. *arXiv preprint*, 2017.

[31] Shalev-Shwartz, S.; Shashua, A. On the Sample Complexity of End-to-end Training vs. Semantic Abstraction Training. *CoRR*, volume abs/1604.06915, 2016, 1604.06915. Available from: http://arxiv.org/abs/1604.06915

[32] Himmelsbach, M.; Mueller, A.; et al. LIDAR-based 3D object perception. In *Proceedings of 1st international workshop on cognition for technical systems*, volume 1, 2008.

[33] Durrant-Whyte, H.; Bailey, T. Simultaneous localization and mapping: part I. *IEEE Robotics & Automation Magazine*, volume 13, no. 2, 2006: pp. 99–110, doi:10.1109/MRA.2006.1638022.

[34] Propwashed. ESC Buyers Guide. 2017, [Online]. Available from: https://www.propwashed.com/esc-buyers-guide/

[35] Ltd, E. NAVIO2 - Autopilot HAT for Raspberry Pi Powered by ArduPilot and ROS. 2018, [Online]. Available from: https://emlid.com/navio/

[36] Community, T. D. DonkeyCar. 2018, [Online]. Available from: http://www.donkeycar.com

[37] SunFounder. PCA9685 16-Channel 12-Bit PWM Servo Driver. October 2016, [Online; posted October-2016]. Available from: https://www.sunfounder.com/pca9685-16-channel-12-bit-pwm-servo-driver.html

[38] Otto, c. B., Mark. Bootstrap - The world's most popular mobile-first and responsive front-end framework. 2018, [Online, 2.2.2018]. Available from: http://getbootstrap.com

[39] Bengio, Y.; LeCun, Y. Scaling Learning Algorithms towards AI. In *Large Scale Kernel Machines*, edited by L. Bottou; O. Chapelle; D. DeCoste; J. Weston, MIT Press, 2007. Available from: http://www.iro.umontreal.ca/~lisa/pointeurs/bengio+lecun_chapter2007.pdf

[40] Kingma, D. P.; Ba, J. Adam: A Method for Stochastic Optimization. *CoRR*, volume abs/1412.6980, 2014, 1412.6980. Available from: http://arxiv.org/abs/1412.6980

[41] LeCun, Y.; Bottou, L.; et al. Efficient BackProp. In *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*, London, UK, UK: Springer-Verlag, 1998, ISBN 3-540-65311-2, pp. 9–50. Available from: http://dl.acm.org/citation.cfm?id=645754.668382

[42] Fukushima, K. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, volume 36, no. 4, Apr 1980: pp. 193–202, ISSN 1432-0770, doi:10.1007/BF00344251. Available from: https://doi.org/10.1007/BF00344251

[43] Yamins, D. L. K.; Hong, H.; et al. Performance-optimized hierarchical models predict neural responses in higher visual cortex. *Proceedings of the National Academy of Sciences of the United States of America*, volume 111, no. 23, 06 2014: pp. 8619–8624, doi:10.1073/pnas.1403112111. Available from: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC4060707/

[44] Russakovsky, O.; Deng, J.; et al. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, volume 115, no. 3, 2015: pp. 211–252, doi:10.1007/s11263-015-0816-y.

[45] Mishkin, D.; Sergievskiy, N.; et al. Systematic Evaluation of Convolution Neural Network Advances on the ImageNet. 05 2017.

[46] Fridman, L. MIT 6.S094 - Lecture 1: Deep Learning [Online]. January 2018. Available from: `https://selfdrivingcars.mit.edu`

[47] Scherer, D.; Müller, A.; et al. Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition. In *Artificial Neural Networks – ICANN 2010*, edited by K. Diamantaras; W. Duch; L. S. Iliadis, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, ISBN 978-3-642-15825-4, pp. 92–101.

[48] Wang, Z. Self Driving RC Car. 2016, [Online]. Available from: `https://zhengludwig.wordpress.com/projects/self-driving-rc-car/`

[49] Glorot, X.; Bordes, A.; et al. Deep Sparse Rectifier Neural Networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, *Proceedings of Machine Learning Research*, volume 15, edited by G. Gordon; D. Dunson; M. Dudík, Fort Lauderdale, FL, USA: PMLR, 11–13 Apr 2011, pp. 315–323. Available from: `http://proceedings.mlr.press/v15/glorot11a.html`

[50] Iandola, F. N.; Moskewicz, M. W.; et al. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *CoRR*, volume abs/1602.07360, 2016, `1602.07360`. Available from: `http://arxiv.org/abs/1602.07360`

[51] Wang, D.; Qi, F. Trajectory Planning for a Four-Wheel-Steering Vehicle. In *ICRA*, 2001.

[52] Lawrence, S.; Lee Giles, C.; et al. Lessons in Neural Network Training: Overfitting May be Harder than Expected. 01 1997: pp. 540–545.

[53] Geman, S.; Bienenstock, E.; et al. Neural Networks and the Bias/Variance Dilemma. *Neural Comput.*, volume 4, no. 1, Jan. 1992: pp. 1–58, ISSN 0899-7667, doi:10.1162/neco.1992.4.1.1. Available from: `http://dx.doi.org/10.1162/neco.1992.4.1.1`

[54] Srivastava, N.; Hinton, G.; et al. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, volume 15, 2014: pp. 1929–1958. Available from: `http://jmlr.org/papers/v15/srivastava14a.html`

[55] Zeiler, M. D.; Fergus, R. Visualizing and Understanding Convolutional Networks. *CoRR*, volume abs/1311.2901, 2013, 1311.2901. Available from: http://arxiv.org/abs/1311.2901

[56] Zhou, B.; Khosla, A.; et al. Learning Deep Features for Discriminative Localization. *CoRR*, volume abs/1512.04150, 2015, 1512.04150. Available from: http://arxiv.org/abs/1512.04150

[57] Lin, M.; Chen, Q.; et al. Network In Network. *CoRR*, volume abs/1312.4400, 2013, 1312.4400. Available from: http://arxiv.org/abs/1312.4400

[58] Springenberg, J. T.; Dosovitskiy, A.; et al. Striving for Simplicity: The All Convolutional Net. *CoRR*, volume abs/1412.6806, 2014, 1412.6806. Available from: http://arxiv.org/abs/1412.6806

[59] Gildenblat, J. Visualizations for regressing wheel steering angles in self driving cars. 2016, [Online]. Available from: https://jacobgil.github.io/deeplearning/vehicle-steering-angle-visualizations

[60] Foundation, T. R. P. Raspbian. 2018, [Online]. Available from: http://raspbian.org/FrontPage

[61] Goodfellow, I. J.; Pouget-Abadie, J.; et al. Generative Adversarial Networks. *ArXiv e-prints*, June 2014, 1406.2661.

Appendix A

Acronyms

AI Artificial Intelligence

ALVINN Autonomous Land Vehicle in a Neural Network

API Application Programming Interface

ARM Advanced RISC Machine

AWS Amazon Web Services

CAM Class Activation Maps

CNN Convolutional Neural Network

CPU Central Processing Unit

ESC Electronic Speed Controller

DAVE DARPA Autonomous Vehicle

DARPA Defense Advanced Research Projects Agency

FFNN Feed Forward Neural Network

FPS Frames Per Second

GAP Global Average Pooling

GPU Graphical Processing Unit

GUI Graphical user interface

HAT Hardware Attached on Top

HSV Hue Saturation Value

ILSVRC ImageNet Large Scale Visual Recognition Challenge

LSTM Long Short-term Memory

ML Machine Learning

NHTSA National Highway Traffic Safety Administration

NN Neural Network

PID Proportional–Integral–Derivative

PWM Pulse-Width Modulation

RAM Random Access Memory

RC Radio Controlled

ReLU Rectified Linear Unit

REST Representational State Transfer

RGB Red Green Blue

SLAM Simultaneous Localization And Mapping

STL Stereolithography (file format)

Appendix B

Contents of enclosed CD

- `readme.txt` ..... CD contents description
- `examples` ..... directory with short video examples
- `src`
  - `deeprcar` ..... directory with source codes
  - `thesis` ..... directory with LaTeX source codes of the thesis
- `text` ..... directory with thesis text files
  - `DP_Ungurean_David.pdf` ..... the thesis text in PDF format