BACHELOR'S PROJECT:

- ***Title:*** ***Autonomous Vehicle implementation with End to End Deep Convoluted feed forward neural network and S.L.A.M***
- *Student:* *Md Mohidul Hasan 15071717*
- *Supervisor:* *DR. Raimond Kirner*
- *Branch:* *Software Engineering*
- *Department:* *Department of Computer Science*
- *Validity:* *Until the end of winter semester 2019/20*
- *Date:* *30th April, 2019*

Acknowledgements:

First and foremost, I would like to thank my supervisor Dr. Raimond Kirner for granting me the opportunity to explore such a vast and interesting field of Machine Learning and neural computation and for always being available and pointing me to the right direction.
I would like to express my gratitude to all my friends and family for helping me make this project a reality.
Last but not the least I would like to acknowledge with much appreciation, the staff of the school of computer science, University of Hertfordshire for granting me access to the school resources at all times.

Declaration Statement:

I hereby declare that, the thesis presented, is my own work and all the sources of information in accordance with the guide line for adhering to principles of ethics when expanding an academic thesis have been cited.

I acknowledge that my thesis is subject to the rights and obligations owned by University of Hertfordshire, School of Computer Science.

April, 30th, 2019

Abstract:

The proposition of the project undertaken is to design, develop and implement an RC car employing a Raspberry pi (Computer), deep convoluted neural networks and SLAM. The objective is to successfully employ the concept of behaviour cloning within the field of vehicle operation exercising miscellaneous computer vision algorithms alongside concepts like Machine learning. A single front facing camera have been used, images from which will be passed onto the Supervised CNN to retrieve predicted steering commands. The proposed model is an illustration of supervised learning.

The report simply explains the entire developmental process of the project. The resolution of the project would be to demonstrate a computer algorithm operating a vehicle with human like precision. This will be done by employing deep end to end convoluted neural networks model proposed by NVidia (Karol Zieba NVIDIA Corporation, 2016), open source computer vision library known as open cv, and an RC car with raspberry pi and a driver board. The NVidia CNN model is able to generalize and obtain abstract models of different road features from within any image with almost human like precision with just one input parameter.

SLAM algorithms make use of LIDAR sensor, which will be the lesser focus of the project. SLAM algorithms are generally applied to retrieve a map of the environment; the vehicle is being driven into simultaneously localising itself within that map. An extended version of the Kalman filter is applied to the LIDAR data to acquire the map. Different objects located within the environment are also positioned within the map recovered.

Abbreviations:

CNN (Convoluted Neural Network)
SLAM (Simultaneous Localisation and Mapping)
LIDAR (Light Detection and Ranging)

Contents:

Introduction:

Supervised deep learning algorithms such as CNN's have brought remarkable success in the fields of computer vision and artificial intelligence. In many cases these algorithms have out-performed classical feature extraction based A.I algorithms for classification problems like semantic segmentation, object detection and labelling (Image captioning). These network of algorithms can extract features from a well-defined image with numerous levels of abstraction which would otherwise be very difficult for human beings to extract and classify employing generalised rules. With the increasingly large amount of open sourced available datasets and with the increasing power of computation employing GPU's and TPU's, Computers can train these models effectively and quickly to yield increasingly better results. In today's world this technology can be seen in most places , whether it be Google's search engine, or google news's or in medical science to classify malignant and benign tumour's or even in housing business to predict the price based on some set and defined parameters. A field that is benefitting with the emergence of this technology is the automotive engineering. Self-driving cars are being engineered by all most all of the fortune 500 companies like TESLA, WAYMO or UDACITY.

figure 1 : Udacity self-driving car engineer (Udacity)

Autonomous vehicles have already started to revolutionize transportation. Self-driving vehicles are well equipped to save lives in situations where it is not possible for a human driver to avoid the accidents purely because of the long ranged sensors like, Ultrasonic, Lidar or the Camera etc. These sensors provide vehicles with almost super human like abilities regardless of the fact that machines comprise of a far better reaction time then human beings. This is why companies like google and Udacity are providing the general public with open sourced resources for Machine learning like TENSOR FLOW or UDACITY SELF DRIVING SIMULATION and educating them to contribute towards the solutions for the problem of vehicle autonomy.

Hypothesis:

The goal of this project is to develop a low cost prototype using raspberry pi RC car and end to end deep CNN.

The car should be able to drive with a well-defined track identifying the lanes from processing the image captured by the camera mounted over the car. The track will be indicative of a simplified road. The only put parameter to the agent will be images captured by the camera. The agent will then process the image and output a steering angle based on the input. As the model accepts images as the only source of input, the thesis of this undertaking will be to design, construct and develop a model that can successfully control the RC car by providing steering commands based on appropriate input images.

A separate model has also been trained employing supervised classification algorithms to process and identify traffic signals. Similar to the model mentioned above, it will only take images as appropriate input, process it within the model and classify the image as output argument.

While obstacle avoidance and detection is a major problem when considering self-driving cars, but combining it with the steering angle as an output for the model within one single controller loop is beyond the scope of this thesis. Although, a LIDAR device will be used for obstacle avoidance and slam within a separate loop which will not affect the loop with the steering commands or the agents.

An inexpensive RC car chassis along with a cheap raspberry pi with 4 DC motors will be used. The software will be written using python machine learning libraries for the purpose extendibility.

Definitions:

Neural Network: A neural network in simple terms is defined as a network of simplified processing units that share similar functionalities with biological neurons. The processing ability is obtained by adapting to a set of training data extrapolating useful patterns and storing them as weights. (Gurney, 1997)

Feed Forward Neural Networks: Within a neural network model if there is no feedback loop or feedback directed from the output of the model towards its input, then the model can be described as feed forward neural network. (SAZLI)

CNN: A CNN also known as a convent, is a subclass of deep feed forwards neural networks designated to analysing visual data or images. (University)

Artificial Intelligence: A branch of Computer science that deals with the science and engineering of intelligent computer programs to understand human intelligence but also not confined within the domain of methodologies that are biologically inspired. (-Prakhar Swarup, 2012)

Machine Learning: According to Tom M. Mitchell "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E". (wiki, 2018)

Classification: Within machine learning and statistics, classification is defined as the process of identifying the category among a set of categories to which an observable data belongs to, based on a set of training instances inclusive of the categories defined. (wikipedia)

Supervised Learning: Supervised learning is the task of learning a specified function that maps a certain input variable to an output variable based on tuples of large input output pairs. It is a subset of machine learning. (wikipedia, 2019)

Deep Learning: Deep learning also known as deep structured learning or hierarchical learning is a segment of machine learning methods based on the number of layers within the developed model of the neural network. (wikipedia)

SLAM: Within the fields of robot mapping and odometry, navigation , SLAM is the computational model of simultaneously mapping an unknown environment while simulating the agent's location within the environment. (ross)

Project background:

Levels of Autonomous Driving:

There are five levels of autonomous driving according to National Highway and Traffic administration. (Forbes)

Level 0(No automation): Vehicle is completely driven by a human. (Forbes)

Level 1(Driver assistance): The driver is being assisted by the vehicle with certain features but not at the same time. E.g: keeping lane, controlling cruise or assisted breaking. (Forbes)

Level 2(Partial automation): The vehicle can simultaneously steer, accelerate and decelerate by itself. Although human driver is expected to do other manoeuvres when required relating to driving. E.g: Tesla Autopilot（Forbes）

Level 3(Conditional automation): The vehicle can perform driving in its entirety in most cases. The human driver is expected to take control when the system fails or the conditions are not favourable for autonomous driving. E.g: Waymo|(Google)（Forbes）

Level 4(High automation): The vehicle can perform driving in its entirety in most cases. However, the human driver is not expected to take control when the system fails or the conditions are not favourable for autonomous driving. The vehicle should be able to stop, or come at a safe position if the human driver fails to retrieve vehicle control. E.g: Google's latest Waymo project is supposed to level 4 although it is in its testing phase. (Forbes)

Level 5(Full automation): The vehicle driving system is able to navigate and drive in all cases and circumstances. (Forbes)

This thesis is aimed at developing a driving system with a level of automation between 2 and 3. Navigation will be completely autonomous while decelerating will only work when an obstacle of some sort is detected. (Forbes)

Research into hardware:

Raspberry Pi 3 b +:

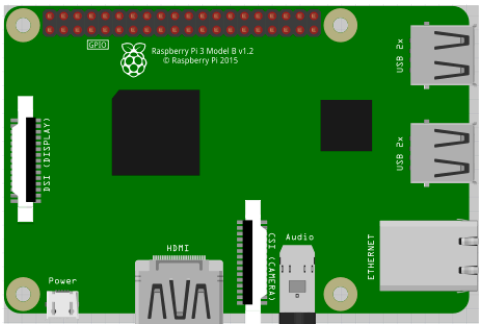


fig: raspberry pi 3 b+

A Raspberry Pi is a low cost bank card sized computer that can be plugged into any monitor or screen. It also uses general keyboard and mouse used for a desktop's.

It consists of 40 GPIO pins or header that allows the user to connect to it slave circuit and modules, sensors like the ultrasonic sensor, IR (Infrared) sensor or LIDAR. It also consists of USB slots to connect webcams or designated pi camera slot to connect to a pi camera.

Within the GPIO header different protocols like IDEPROM or UART (universal asynchronous receiver-transmitter) or I2C bus is encoded with pins like SCL, SDA, TXD, RXD.

It also consists of a 1.3 Ghz quad-core processor and 1 GB of RAM (Random Access Memory).

GPIO (General purpose input and output):

fig : Pi GPIO header

The GPIO pins in raspberry pi allow the users to interface physical devices like DC motors, ultrasonic sensors, LIDAR, camera, IR with the Linux kernel employing python programming language within the ROS (Raspian or Robot operating system). Using python RPI.GPIO library input to these physical systems can be controlled.

There are 2 pin numbering system for GPIO headers known as the BOARD pin numbering system or the BCM (Broadcom soc numbering system). BCM generally differ for different raspberry pi versions while BOARD pins follow the exact same pattern engraved within the pi GPIO header.

For the scope of this thesis the BOARD pin numbering system has been used.

Lidar:

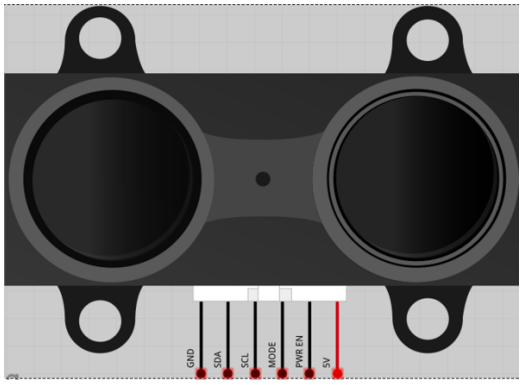


fig: LIDAR

Lidar sensors are generally used to map unknown environments with extreme precision. These sensors generally use a pulsed lased to measure the distance between the object and the sensor itself. These generated pulses are used to then measure the surface of the object with precision.

The LIDAR used for this thesis employs UART protocol as such communication between the pi and the LIDAR is very difficult to establish. It consists of 4 pins namely TX (transmitter), RX (Receiver) , VCC (Voltage) and Ground.

The data received from this device will be used to implement the SLAM algorithm. It can also be used for Object detect and obstacle avoidance.

USB – Camera:

The only source of input for our model will be images captured by the camera module mounted at the top of the vehicle. It will be connecting to one the USB ports and python imaging library alongside Open cv will be used for image manipulation, augmentation and processing.

Power Management:

A 12-volt battery is being used to power the driver board used as the HAT for the pi. Another separate 12-volt portable charger is being used to power the pi itself.

DC motors and driver Board:

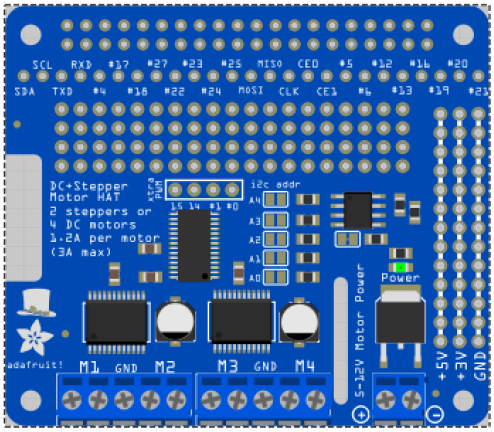


fig: Ugeek Stepper Motor Hat

A stepper motor hat with dual L298 H-bridge motor drivers are used to control 4 DC motors. This driver board used as the HAT for the pi as such interfacing with pi becomes a lot easier and maintainable.

Python scripts are run within the pi terminal to control the 40 GPIO pin header and the DC motors connected to the driver board. The driver board consists of 2 input units for each motor, which will then be used to drive the motors forwards and backwards.

Motor driver consists of PWM (Pulse width modulation) pins as well which will be used to control the speed of the vehicle.

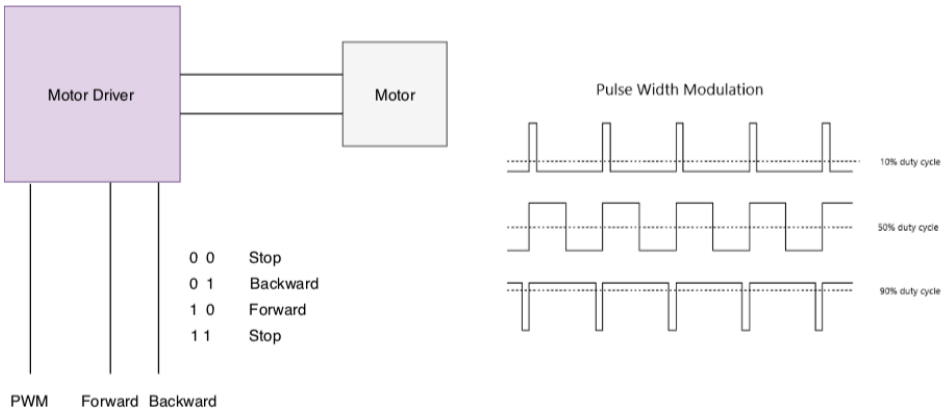


fig: Motor driver connection / Voltage control using PWM

Hardware Configuration:

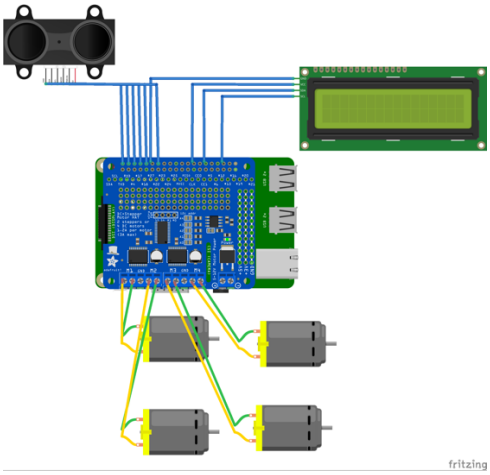


fig: Abstract diagram of the Hardware Structure of the vehicle

All the devices connect above consists of specific pin configuration. These input pins are programmed using python and rpi GPIO python library to read and write data to the registers of the driver board. The program is then placed within the pi that in turn controls the pi GPIO channels.

Sensors	TX	RX	Sensors	SCL	SDA
Lidar	14	15	LCD	5	3

Fig: Pin configuration for the sensors

PWM pins in pi:

Within pi GPIO header, there consists of specialised pins known as the PWM pins.
Generalised gpio pins cannot be used as pwm pins as their frequency is very low.
Pwm pins are those pins that consist of a higher frequency than normal pins.

Pi Setup:

The raspberry pi needs to be configured properly before it can be used. The operating system known as ROS (Raspian) needs to be installed. This is done by flashing a USB drive with the ROS DMG file. The pi is connected to a monitor, keyboard and mouse. The usb with the OS is then flashed within the micro SD card of the pi. A complete tutorial for setting up ROS on pi can be found online.

Assembling car:

A smart car chassis is being used to mount all the necessary components of the vehicle. It consists of well-defined holes used to mount the pi, mount of the camera, DC motors, Lidar and LCD.

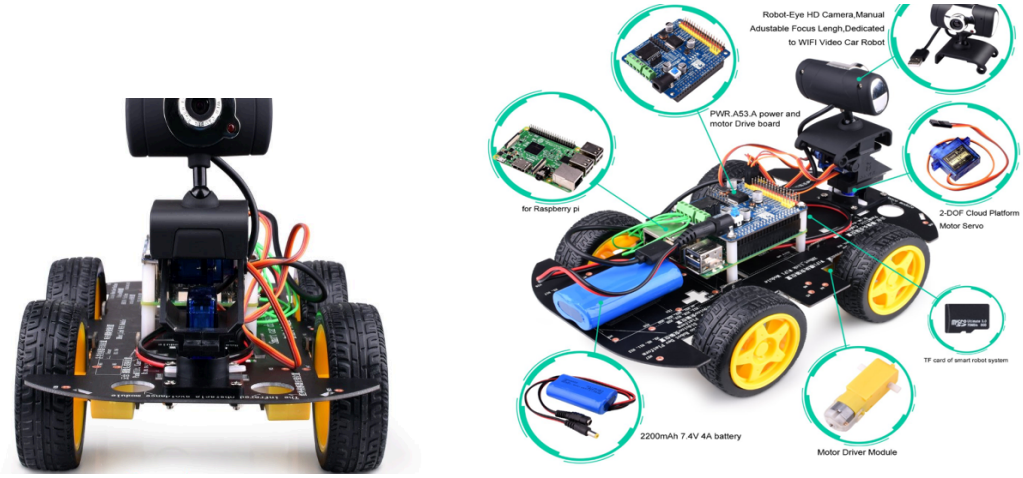


fig: Car chassis and mount

At the top front section of the car, the camera is mounted on top of a mount connected to the chassis body. The LIDAR is mounted on top of the Camera. 2 batteries, one at the front and the other at the bottom is placed.

System Analysis and design:

Ideas and concepts can sometimes become over complicated and confusing. Employing abstraction to disintegrate the idea into smaller sets of problems to make the entire development process feasible, especially in cases of complex developmental procedures. Below a set of activity, use case, sequence and class diagrams have been displayed.

Use Case Diagram:

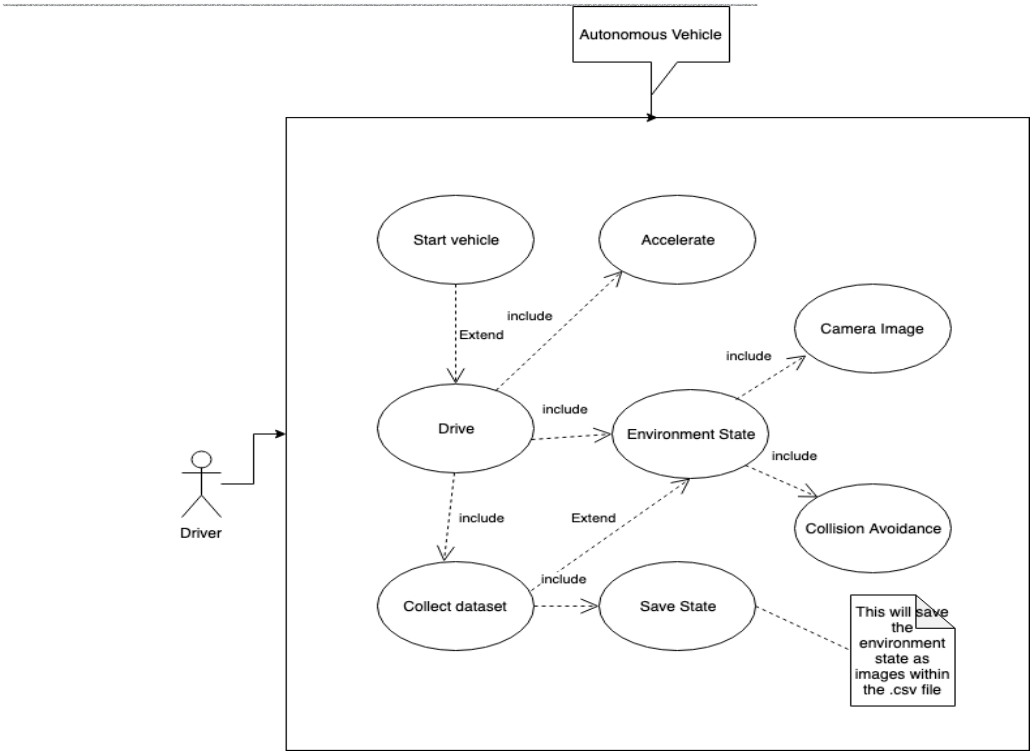


fig: Use case diagram of the proposed thesis

Sequence Diagram:

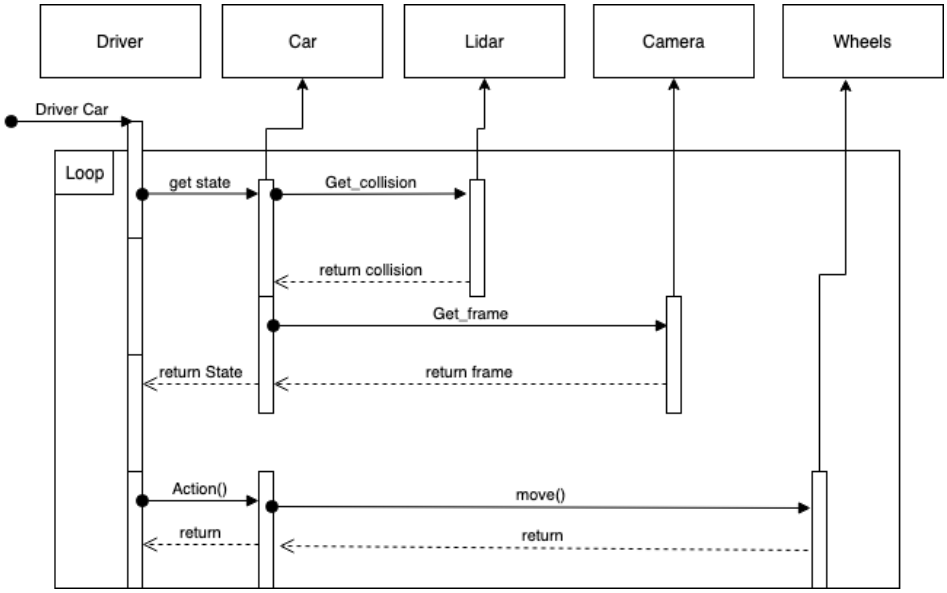


fig: Sequence diagram

Class Diagram:

These diagrams generally help with defining numerous variables and properties. But within the thesis, it'll be used and referenced as the proposed subroutines belonging to the system.

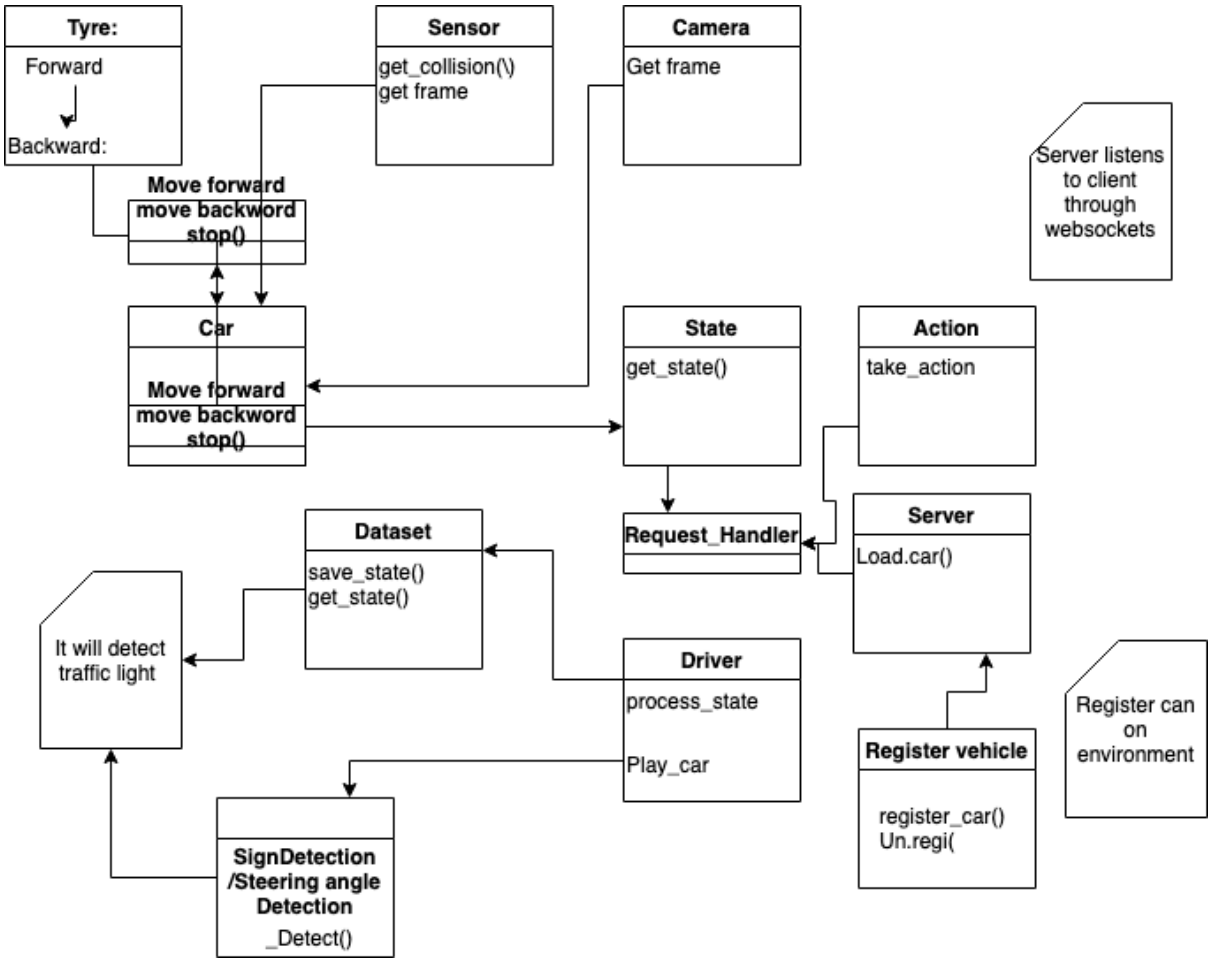


fig: Class Diagram

Activity Diagram:

The vehicle will initiate its sensors and camera after automatically discovering the network. It will then take input from the model regarding steering actions (Left, Right, Forward, Backward). The current state of the car will be generated after executing the collision avoidance function as a result of which, every time the car detects objects or obstacles it comes to a safe state of "STOP". While no obstacles or collision, the vehicle will be progressing based on the prediction steering command by the trained model (model will be discussed within the Deep Learning and AI section).

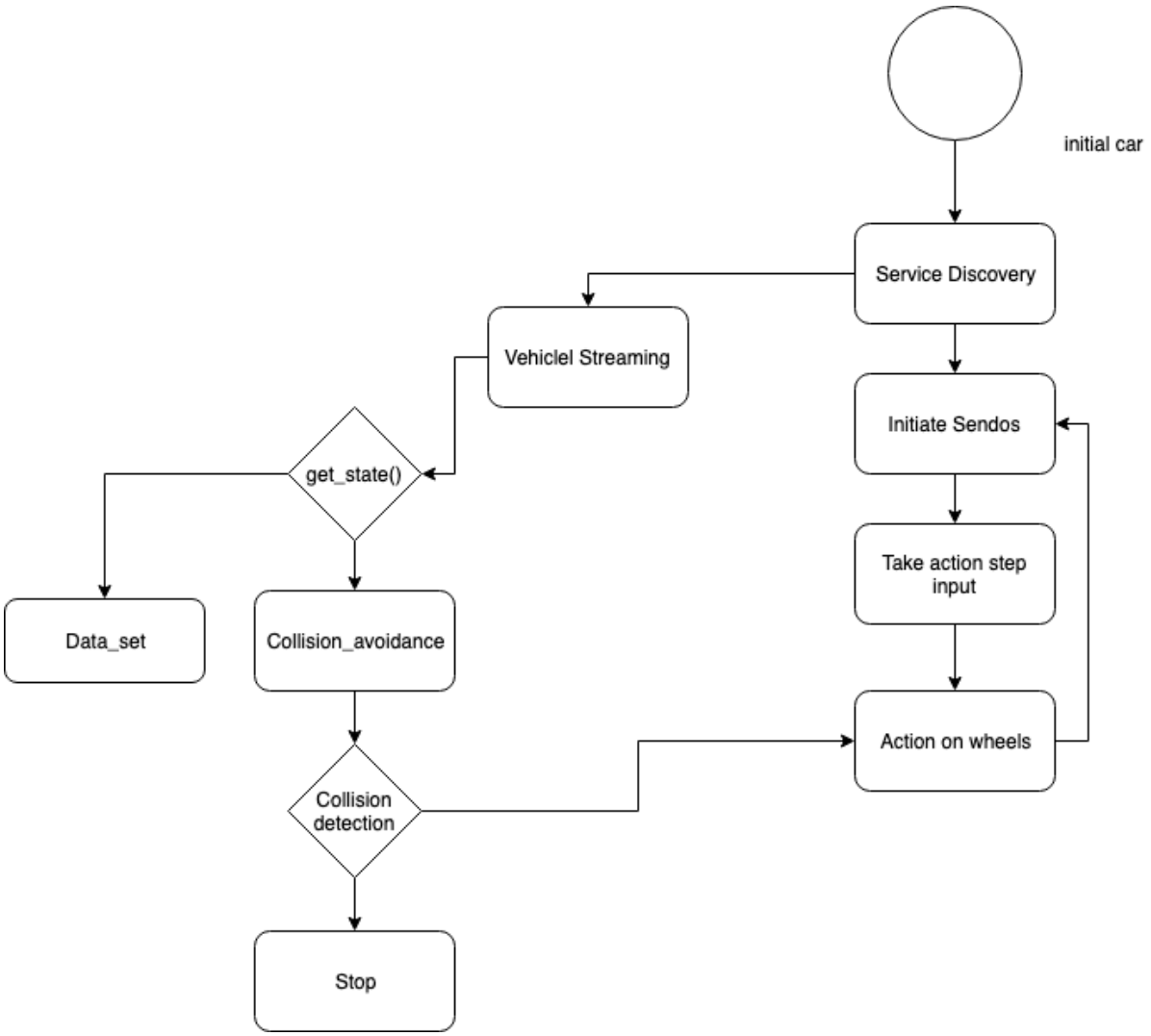


fig: Activity diagram

Software Interface:

This chapter describes in detail, all the necessary software used to establish connection and interface with different modules employed by the autonomous vehicle.

System Architecture:

The main application or control loop is run within the pi. A web interface have also been implemented through which users can interact with the system. The vehicle can be controlled manually through the browser or a simulation has also been implemented, that can be used to run a pertained ML model driving a vehicle. The simulation is based on UDACITY self-driving car model.

Used Technologies:

The entire software stack has been written on python 3.6 except the front end of the web application for manual control of the vehicle. The front-end was built employing HTML, CSS, Bootstrap and JavaScript. A lot of standardised python libraries have been used for sockets and threads
Third party python libraries like Flask and Cherry have been used for backend section of the web app.
The model was trained used general python libraries like numpy, keras, sklearn, pandas and Tensorflow.

Car's Main loop:

The main loop within the main.py file runs indefinitely. At the beginning it looks for obstacles within the track. While there is no obstacles within the track the vehicle gets most recent steering angles from the model and feeds it into the actuators. Within the main loop logic is also added for 3 traffic sign's namely : stop, red light and cross sign (No road forward). A second ML model is also trained using multi class classification algorithms to predict traffic signs.

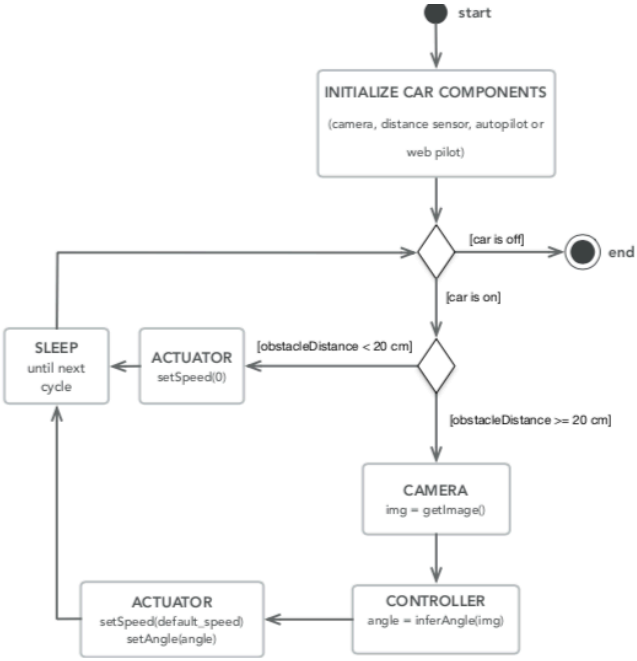


fig: Car's life cycle

While there are no obstacles detected or no traffic signs, the vehicle takes in and process output variables as steering angles from the trained model to drive autonomously within the track.

Controlling the Servo:

The camera at the front of the car Chassis is mounted on top of 2 servos'. This was done to effectively increase the degree of freedom of the camera. These servos are connected to the pwm channels on the driver board as the pi not efficient in terms of pwm channel generation.

```
#!/usr/bin/python

from Raspi_PWM_Servo_Driver import PWM
import time

# ===========================================================================
# Example Code
# ===========================================================================

# Initialise the PWM device using the default address
# bmp = PWM(0x40, debug=True)
pwm = PWM(0x6F)

servoMin = 150  # Min pulse length out of 4096
servoMax = 600  # Max pulse length out of 4096

def setServoPulse(channel, pulse):
  pulseLength = 1000000                   # 1,000,000 us per second
  pulseLength /= 60                       # 60 Hz
  print ("%d us per period" % pulseLength)
  pulseLength /= 4096                     # 12 bits of resolution
  print ("%d us per bit" % pulseLength)
  pulse *= 1000
  pulse /= pulseLength
  pwm.setPWM(channel, 0, pulse)

pwm.setPWMFreq(60)                        # Set frequency to 60 Hz
while (True):
  # Change speed of continuous servo on channel 0
  pwm.setPWM(0, 0, servoMin)
  time.sleep(1)
  pwm.setPWM(0, 0, servoMax)
  time.sleep(1)
```

fig: Example servo control code

LIDAR:

The LIDAR sensor has been used for obstacle avoidance since implementation of SLAM algorithms have become out of scope for the proposed thesis. A file named lidar.py contains all the necessary program code to retrieve desired output from within the LIDAR device. A code snippet of it has been added below for clarification purposes. Python serial package has been imported and used.

```
# -*- coding: utf-8 -*
import serial

ser = serial.Serial("/dev/ttyAMA0", 115200)

def getTFminiData():
    while True:
        count = ser.in_waiting
        if count > 8:
            recv = ser.read(9)
            ser.reset_input_buffer()
            if recv[0] == 'Y' and recv[1] == 'Y': # 0x59 is 'Y'
                low = int(recv[2].encode('hex'), 16)
                high = int(recv[3].encode('hex'), 16)
                distance = low + high * 256
                print(distance)



if __name__ == '__main__':
    try:
        if ser.is_open == False:
            ser.open()
        getTFminiData()
    except KeyboardInterrupt:   # Ctrl+C
        if ser != None:
            ser.close()
```

fig: Example Lidar code

A LIDAR device as mentioned above employs UART protocol. Within this protocol there are 2 pins that are being used, namely serial transmit (TX) and serial read (rx) . The TX pin of the LIDAR is connected to the RX pin of the pi while the RX pin of the Lidar is connected to the TX pin of the pi for establishing a successful 2 way data communication service.

LCD display:

An LCD (Liquid Crystal Display) device employs I2C bus. It is essential a protocol employed by low speed devices like microcontrollers, EEPROMs, A/D and D/A converters, I/O interfaces and other similar peripherals in embedded systems.

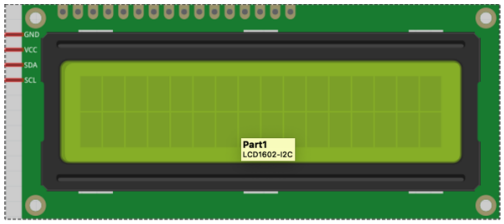


fig: LCD display

Within the LCD there are 4 existing pins known as the SCL (serial clock) , SDA (serial data) , VCC and ground. The SCL and SDA pins are connected to the corresponding SCL and SDA pins on the driver board.

Web Interface:

Highly sophisticated web development architecture known as the MVC or Model, view and controller architecture has been used to design and develop the web interface. Flask package has been used for developing the backend of the web app while bootstrap for the GUI (Graphical user interface).
Users can interact with the vehicle through the GUI. It is made possible to drive the vehicle forwards, backwords, left and right by making use of the buttons and sliders within the GUI or by using a ps3 controller. Users can also turn the recording on or off, modify speed and see the camera input in real time. Once the car is started the interface can be found at ip_address:5000.

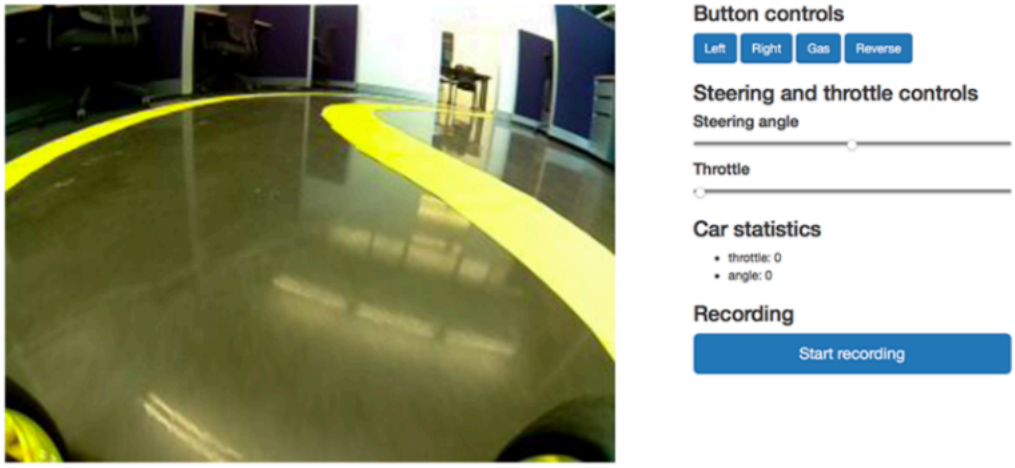


fig : frontend of the web interface

Command Line Interface:

The car can be run in 2 modes. Either in manual mode where the user has to manually drive the car around or in auto pilot mode where the controller takes control of the vehicle.
Within the auto pilot mode, a python script is used to make the vehicle run around a lane using computer vision lane detection algorithm.
On the contrary, the user has to manually take control of the vehicle and drive it using keyboard w, a, s, d keys for forwards, left, back and right.

Dataset Recording:

Collecting the appropriate training data is one of the most crucial tasks undertaken within the scope of the thesis. Not only is it important for the entirety of the thesis, but also for the CNN model which will be trained using the dataset recorded.

Dataset can be collected via the web interface making good use of the recording button or via the command line interface by running the autopilot.py script. The entire process is controlled by an object called the recorder. It creates a folder pyramid 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.

The initial training data was collected using a simulation. Udacity self-driving car simulation was built upon unity game engine. It allows the user to drive a car within a real road with extraneous circumstances, while recording the entire drive. These videos are later on processed into images and a corresponding data.csv file is created that has the images as the input value and the steering angle as the labels.

fig: Visualisation of training data, acquired from MNIST dataset

Approaches to Steering in Autonomous Driving:

Currently there exists 3 ways to deal with the problem of autonomous driving.
They are:

- Non-AI approach (Manual engineering)
- AI approach (CNN, Reinforcement Learning DQN)
- Combination of AI and non AI

Non-AI approach (Manual engineering):

The non AI approach makes use of control theory to keep the vehicle within the lane. It calculates the steering angle based on environment it is driving on using open sourced computer vision algorithms. One of the most popular methods of control theory is PID (Proportional, integral and derivative) controller. The controller works within a loop calculating the error value e (t) with every iteration as the difference between the vehicle's feedback and the next command signal. Afterwards, a correction value is calculated and applied.

The corrected value u (t) consists of 3 arguments namely proportional, integral and derivative and can be computed from the error e (t). (Zhao, et al., 2012)

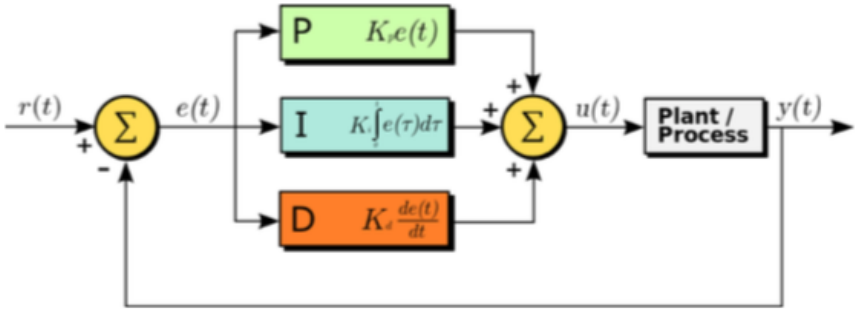


fig: Computation of the corrected value within a PID loop controller

The entire mathematical model is given below:

$$u(t) = k_p\, e\,(t) + K_i \int_0^t e\,(\tau)\, d\tau + K_d \frac{d\, e(t)}{dt}$$

Each parameter has their own coefficients k_p, k_i and k_d needs to be finely tuned for the model to perform well. An in depth analysis of the parameter tuning is out of scope for the thesis undertaken as the approach mainly interested in is the AI approach. For further details refer back to (Levine) , (Chong, et al.) , (Chandni, et al., 2017) and (Zhao, et al., 2012)

AI Approach:

Unlike the non AI approach, The AI approach does not compute the optimal steering value based on control logic and equation rather, it depends on an intelligent system to predict the best steering angle based on previously processed data.

Such agents can be developed and trained using Machine Learning specifically deep learning techniques that can process large data with the aim of recognizing and extrapolating road

features with the most precise level abstraction with the goal of predicting steering angles the vehicle should perform in order to follow the track.

With the rise of deep neural networks and fortune 500 companies like Google, waymo reshaping the idea of driving and integrating it with the field of machine learning and achieving success, new possibilities are in order. The combination of ANN's being a universal approximator including breakthrough in CNN models alongside new and improved GPU models, makes the creation of an NN – based control algorithm a reachable goal.

End to End Deep Learning:

The most standardised approach towards solving the problem of autonomous driving is to break it into several levels of abstraction such as lane marks detection, pedestrian detection, Traffic signal detection, path/motion planning and motor control. This approach to autonomous driving merges all the techniques mentioned above into one simple elegant algorithm. The image below best illustrates the two concepts:

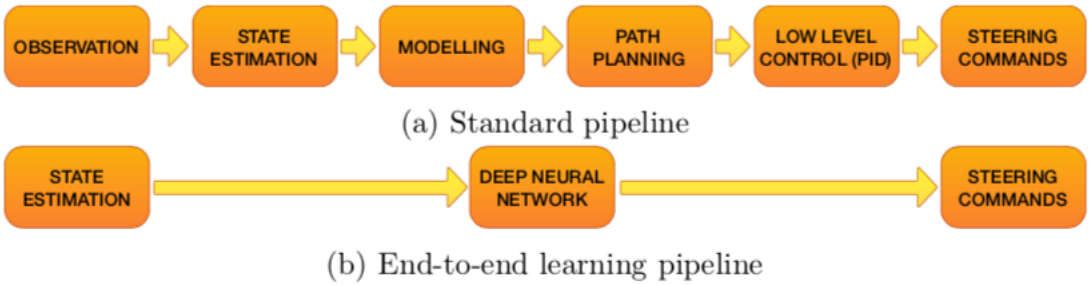


fig: Difference between modularized and AI approach

The first ever self-driving car built with neural network (ALVINN) was done by Pomerleau and it can be dated back to 1989. He employed a single NN with one deep layer composed of 29 nodes. His research greatly facilitates the idea of proposing a solution to the problem of autonomous driving employed miscellaneous end to end neural networks. In 2004 Defence Advanced Research Projects Agency (DARPA) came up with the open source project known as DAVE (DARPA Autonomous Vehicle) which is significantly small RC car project that uses NN to drive within a terrain an simultaneously avoid obstacles.

This project has revolutionised NVidia's self-driving cars project. Within the paper "End-To-End Learning for Self-Driving Cars", NVidia provides a state of the art Deep feed forward end to end CNN model built specifically for self-driving vehicles. The testing of their model displayed promise while it drove on the highway with a controlled amount of traffic.

Real world self-driving cars:

Companies that are conducting research on self-driving cars in modern times are not really interested in an end to end deep learning model as collecting large datasets of images of real world scenarios can be often times unrealistic. It was shown within a study conducted in 2016 that every case where extreme precision is required, amount of training data to train an end to end model based on that case will increase exponentially compared to the modularised model. Within the modularised model, the problem is scaled down to a number of separate modules consisting different levels of abstraction where lane track and or pedestrian detection for example are implemented using multiple ML and or manual engineering algorithms to yield the maximum accuracy.

Another important difference between real world self-driving cars and the RC car used within the scope of this thesis is in the form and size of the input. Companies like google waymo or tesla accumulate data from a wide variety of sensors to concisely precisely localise the vehicle within the environment. These accumulated data is then combined to create a visual model of the vehicles surroundings.

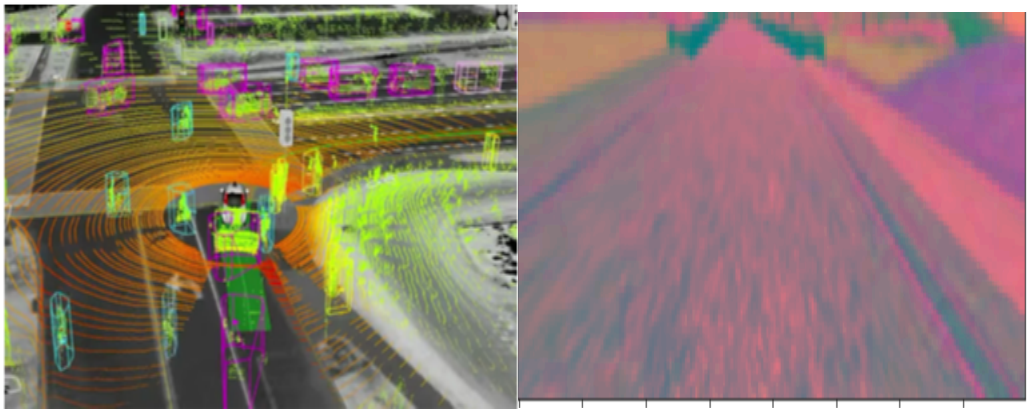


fig: Waymo Sensor Fusion/ Image segmentation with imgaug

Expensive technologies such 360-degree Lidar along with an array of cameras and ultrasonic sensors are being used to generate 360-degree model of the environment. Tesla model S uses 12 ultrasonic sensors, 5 camera's (one at the back and four facing front) and a radar sensor to track the speed of the object in front of the vehicle.

fig: tesla model S

Such vehicles also employ Bayesian SLAM algorithms with extended Kalman filters, in turn combining all the sensor data to generate the 360-degree sensor fusion image.

Thesis Model:

The Goal of this thesis will be to implement an RC car that can autonomous drive itself within an artificially designed track with no traffic. As it is very different to the real world Model, the NN architecture will not complicated by employing a modularised model. Although modularisation will still be implemented in a smaller scale by training a model to predict the steering angle and the other to predict traffic signals. It will then be integrated within the RC car's main control loop along-side certain control logic to drive the vehicle autonomously within the well-defined track. This provides an opportunity to display the strength and clarity of an end to end model when applied to the problem of autonomous driving.

Principles of Deep Learning:

This chapter aims at introducing the reader to the concepts of Deep learning and the benefits of CNN. This chapter begins by explaining the structure of an artificial neuron. Then the reader is introduced with the concepts of feed forward neural networks followed by the theoretical foundations of CNNs which will be heavily used within the practical part of this thesis.

Basics of an Artificial Neural Network :

Similar to the biological half, the artificial neural networks consist of a variety of interconnected nodes or often referred to as neurons that are able to compute their corresponding activation functions based on the activation functions of the preceding layer and their weights and biases.

The underlying idea is that many independent simple unit working together towards a singular goal can often mimic the existence of an intelligent whole much like the ants in many ways. One of the major concepts of deep neural networks is distributed representation or abstraction. The idea is that each input of a system is denoted as a collection of many features and each feature is considered to be a representations of numerous inputs. For example, to construct a classifier that can recognize books, copies and rubbers and each object can only be either red, green or blue. In an attempt to solve the problem, 6 neurons can be created where they can learn the concept of being an object or colour. Every time the number of objects and colour that needs to be identified is increased, the size of the model exponentially increases as well.

Whereas with distributed representations we can scale it down to just 6 neurons where 3 neurons learn the physical shape or aesthetics of an object while the rest learn the method of classifying between colours.

Artificial Neurons:

A comparison between an artificial neuron with a real neuron is discussed below:

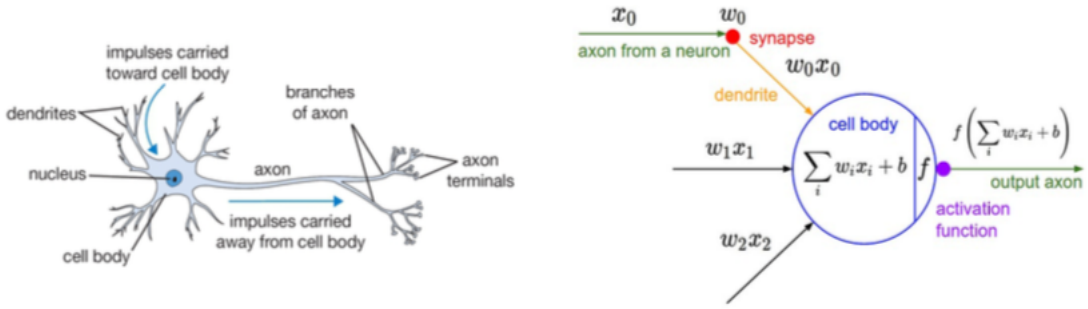


fig: a comparison between a biological unit of brain (left) and an artificial neuron (right)

The output of a single artificial neuron is calculated in the equation below, where x is the input, w is the weight associated with the input, b refers to the bias unit or threshold of that specific neuron and σ is the activation function of that neuron.

$$y = \sigma\,(w_i x_i + b)$$

Activation Functions:

In order to fit a non-linear set of input data, a non-linear activation function needs to be used. The most popular activation functions used in today's world are the sigmoid, relu (rectified linear unit), tanh (hyperbolic) and the elu function. They all have their own short comings and are used in various circumstances to fit various purposes. These functions need to be differentiable for performing gradient 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)

ELU: ELU (x) = $\alpha \times (e^{x} - 1)$

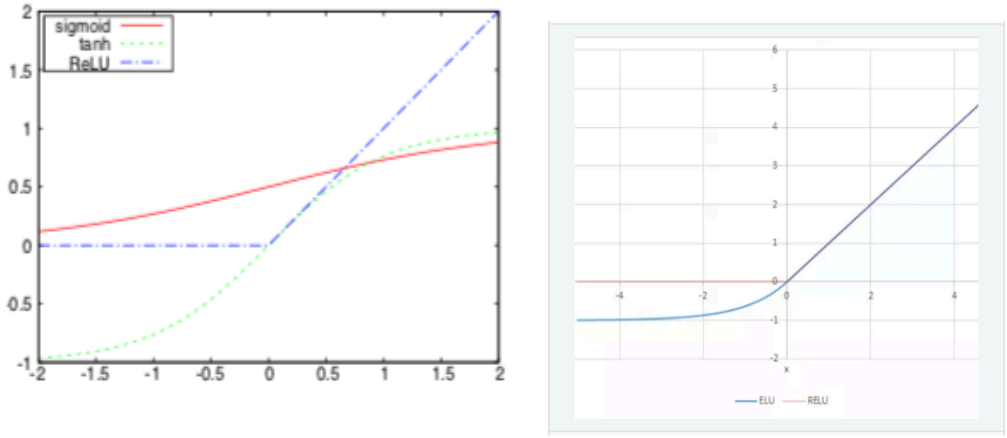


fig: Plots of common activation function

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.

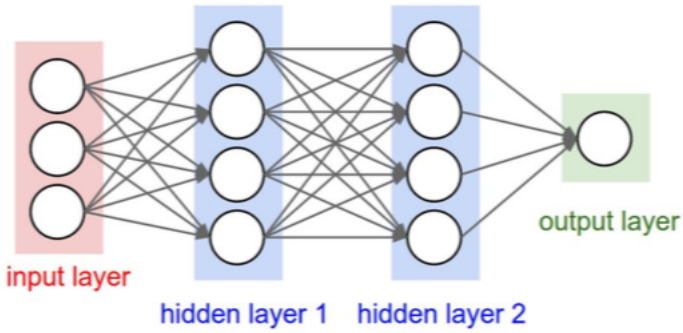


fig: Structure of a simple Feed Forward Neural Network

L denotes the number of layers and $N^{(l)}$ denotes the number of neurons in layer l. Let l = 0 be the input layer and l = L-1 the output layer, 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,

$$a^i_j = \alpha \left(\sum_k w^l_{jk} \; a^{l-1}_k + b^l_j \right.$$

Similarly, a weight matrix W^l is defined for weights in layer l, where the entry in j^{th} row and k^{th} column will be the element $w^l_{j\,k}$ from Equation 4.2. In the same manner a bias vector b^l can be defined 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,

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

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 de- sired 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.

Learning and Cost Functions:

Our goal is to find the parameters θ = (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(θ), that can be defined in several ways. The following Equation displays one example, the mean squared error loss function

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

When using MSE as the 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.

$$\boldsymbol{\frac{\partial \, C_{MSE}}{\partial w}}$$

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].

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

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

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

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 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 colour channel. Formula 4.8 shows how to calculate convolution of n^{th} output channel of such input.

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

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.

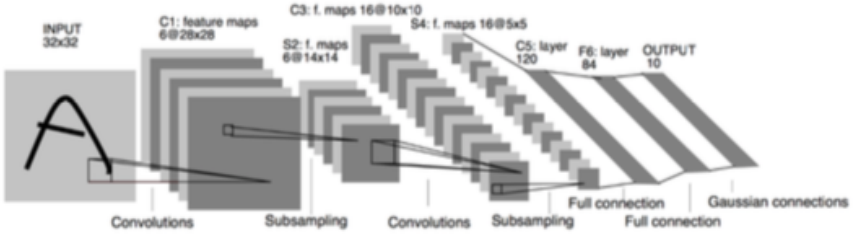


fig: Architecture of LeNet-5

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].

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 down sampling of the input. They do so by combining values of a bigger neighbourhood 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 the figure below:

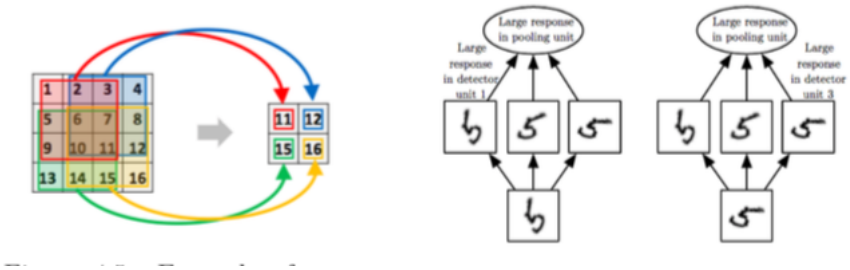


fig: Example of max pooling with filter of size 3 ◊ 3 and stride 2 [4, p. 17] (left) Example of learned invariances [5, p. 4] (right)

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.

Hyper parameters:

Even though convolutional networks have less overall parameters to train than feed forward neural nets, they have more hyper parameters. 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 colour.

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 down sampled, 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.

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.

The Goal of the Controller:

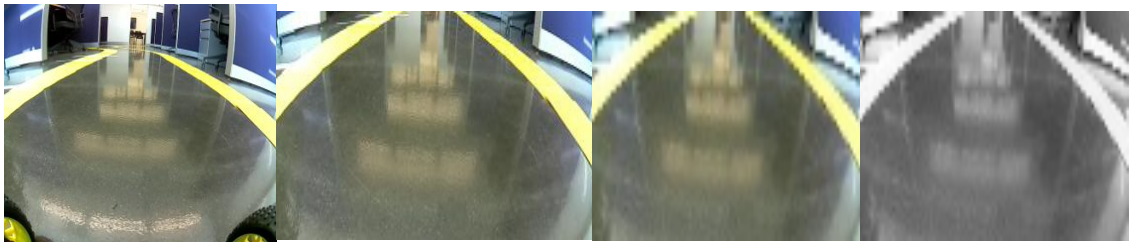
The task of the controller is to control the car's actuator based on the input received from its on-board camera. In other words, we try to find a mapping from camera images to steering commands. Such relationship may seem obvious for human eye, but is difficult to capture by a set of rules in a traditional programming paradigm. When we see a road that is curved, we know how much we should steer, but defining programmatically 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 * n * 3 (input image) onto interval (-1, 1) (steering angles, left and 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.

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/under fitting 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.

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 pre-processing. 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 the following figure



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

Figure 5.1: Example of image pre-processing in training dataset

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]. The following figure depicts the flow of data from input to output.

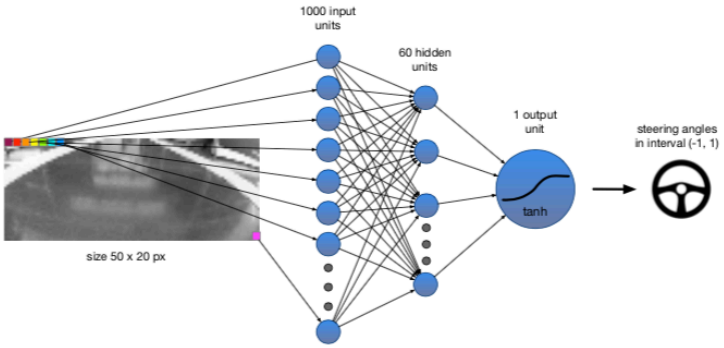
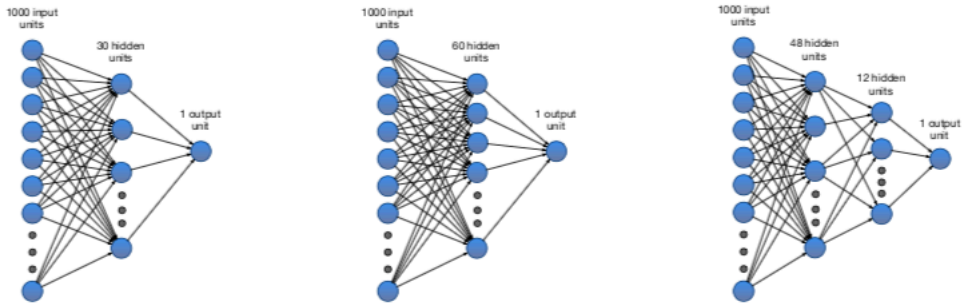


fig: Flow of data with example architecture FFNN2

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 the following figure



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

fig: Tested fully connected architectures

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 net- work designed for autonomous steering of real cars. Then I compared its result to other convolutional networks of different structure.

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 ◊ (1 * 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.

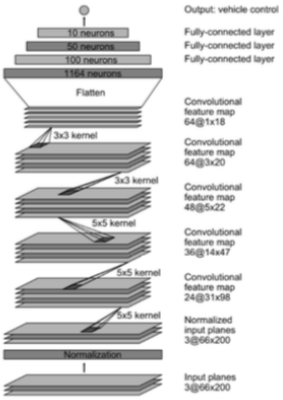


fig: PilotNet architecture

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

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 colour 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 different in colour, so the network only needs to stay in centre of a black/greyish 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 hyper parameters 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 following table:

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**

fig: 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.

Model Comparison:

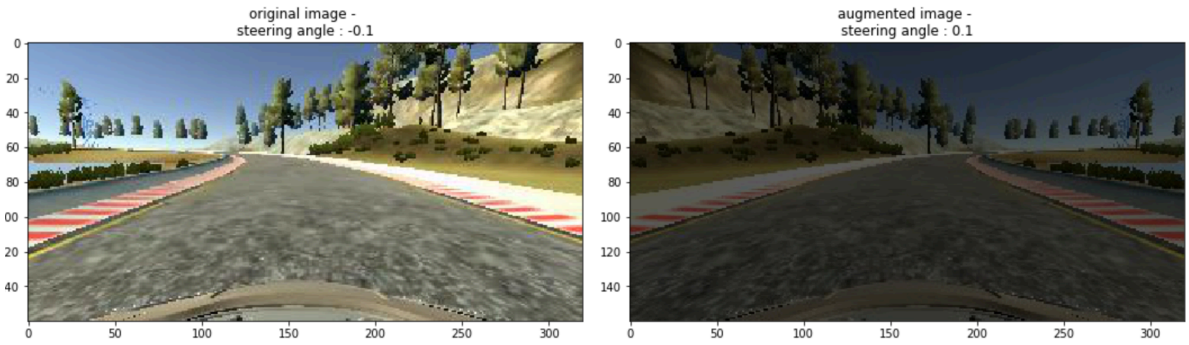
The mean square error loss function was used to train all models. This is common for regression problems, because it punishes large deviations between predicted and recorded results. Given a feed forward neural net function y(x,θ), the mean squared error function can be described by formula 5.1, with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

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

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.

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.



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 the figure below. 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.

fig: Sketch of routes used for training

The dataset also contains situations where the buggy is almost driving o the track. These cases are crucial for the network to learn how to recover from bad situations when it misclassifies 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 the figure below

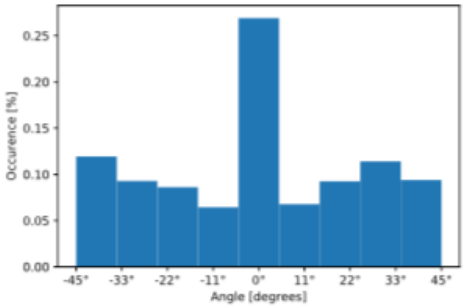


fig: Distribution of angles in the recorded dataset

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.

Validation Loss Tests:

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

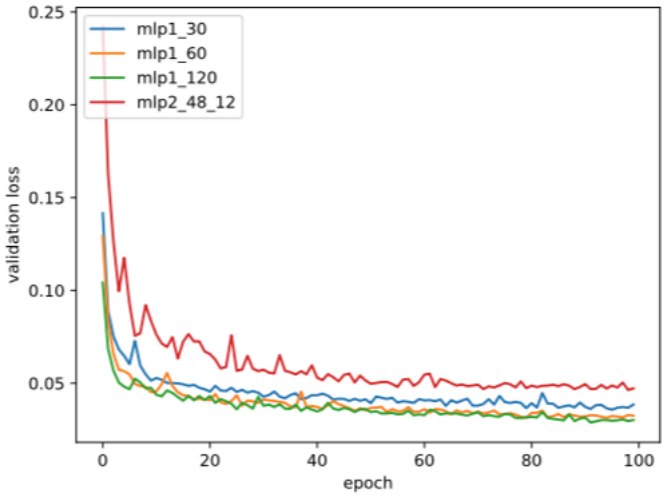


Figure: Comparison of validation loss of fully connected architectures. Blue, orange and green being networks with a single hidden layer of sizes 30, 60 and 120 respectively. The validation loss of network with two hidden layers (48 units and 12 units) is drawn in red.

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 below shows the comparison between a fully connected net- work (blue), NVIDIA's PilotNet (orange) and my best performing convNet architecture (green).

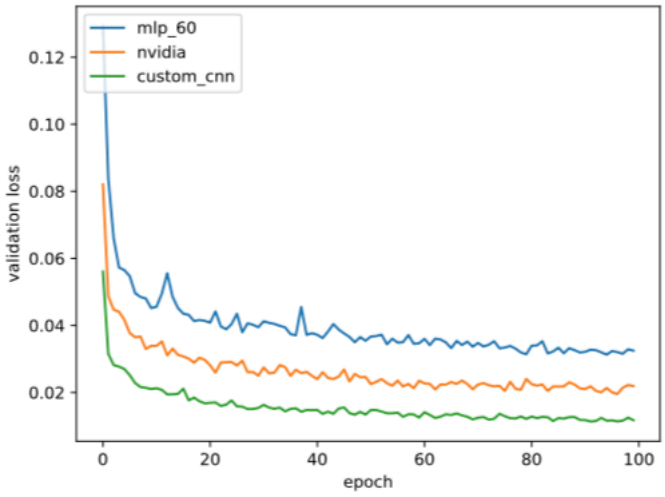


fig: Comparison of validation loss between fully connected networks and CNNs. Blue is plain FFNN, PilotNet is in orange and my custom architecture in green.

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.

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 below.

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

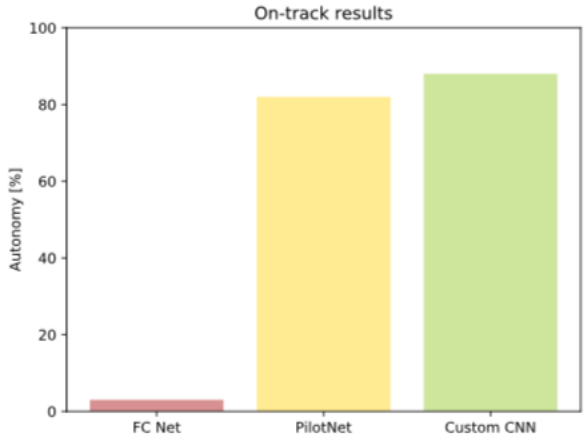


fig: Compared on track performance. Feed forward neural network (red), NVIDIA's architecture (yellow) and my architecture (green).

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 on 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%.

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.

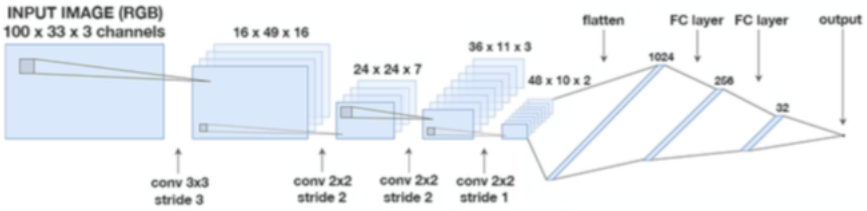


fig: Winning CNN Architecture

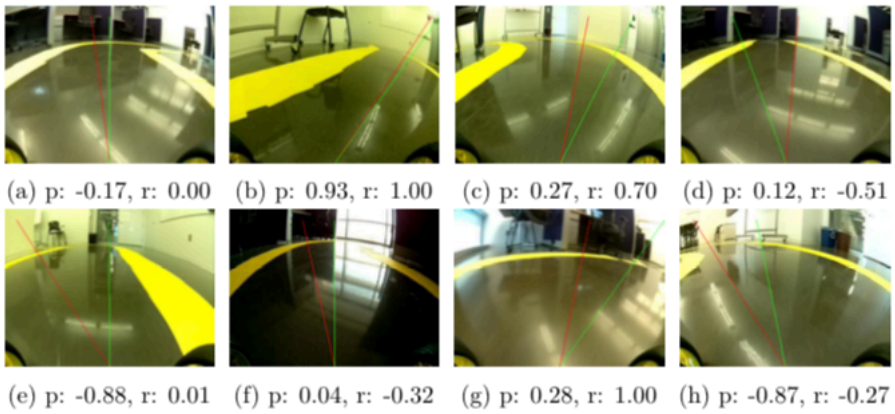


fig: Examples of predicted angle (p, red colour) vs recorded angle (r, green colour)

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 than 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 misclassification 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 recalculated 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.

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. A discussion on interpretability and visualizations techniques used to debug the model's performance is done within this section. 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.

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 hyper parameters 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 tensor flow GPU acceleration. Then even a relatively small grid search within the hyper parameter 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.

Regularization

Regularization is used to reduce overfitting [52] in machine learning models. Good fellow 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.

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 colour of the image simulate different light conditions. This is implemented by randomly increasing or decreasing V channel in the HSV colour 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 behaviour, slight image rotations were added to some samples from the dataset.

Examples of augmented images can be seen in the figure below. The source code of mentioned methods is in file utils.py. The data is fed to the net- work 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.

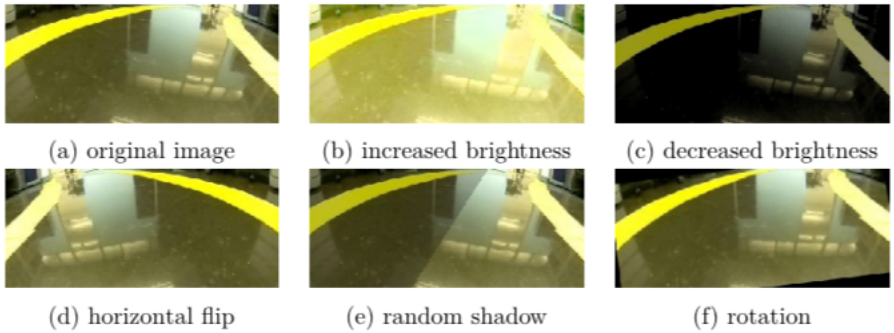


fig: Augmentation examples

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 below 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

fig: 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 eect of the newly collected, rebalanced and augmented dataset on on-track performance is further described in the following Section.

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 over fit, is a widely used method to combat this problem. Shorter training times help to avoid modelling 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 below shows the relationship between training time (epochs) and the autonomy of two models, which were trained using zero and light data augmentation, 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.

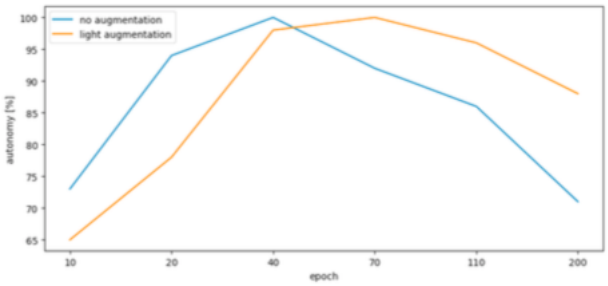


fig: Effect of early stopping on car autonomy. Blue is the model trained without any augmentation, orange was trained with light settings.

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 over fit 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 with- out ever driving o 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 misclassifies. 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 on track, especially when higher speeds are used. This behaviour 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 centre 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 over fit when given enough time to train and misclassification 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 misclassification caused by an over fitted model.

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 below.

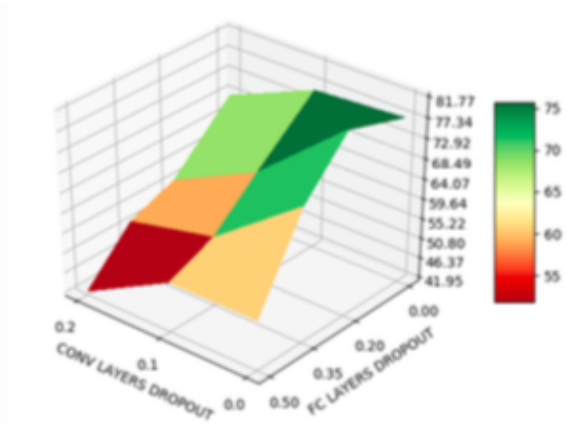


fig: Effects of drop out on validation loss

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.

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 understand- able 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 under fit, over fit 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.

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 colour. Filters in deeper layers combine information of previous neurons to detect corners and other more structured objects.

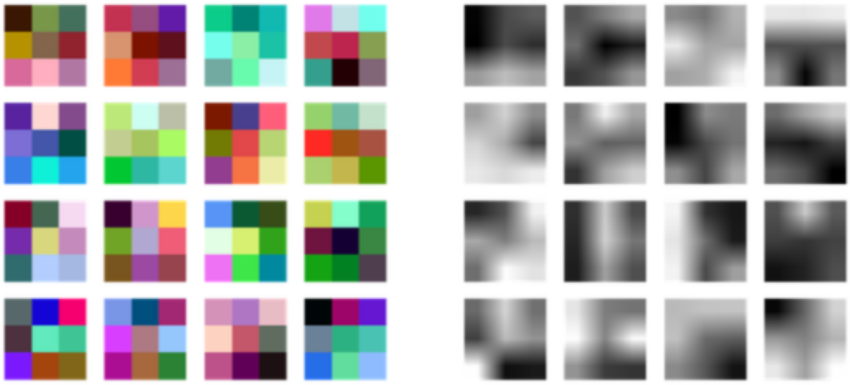


fig: (left) Coloured version without interpolation (right) Grayscale version of green channel with bi linear interpolation
fig: feature map visualisation

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 above shows, most of the sixteen filters of the first convolution layer look relatively smooth. We can see both colour and edge detecting filters that the network was able to pick up during training.

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 below display activations of first two convolutional layers on images from the dataset.

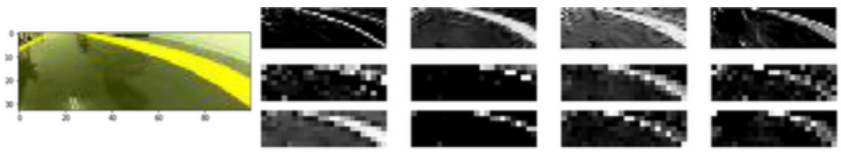


fig: Layer activations for steering right. Images in the first row are activations of the first layer, the remaining rows show activations of the second layer.

fig: 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 colour) 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 colour transitions from yellow to black (edge between the track and lane marks).

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:

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

Where y is the predicted angle without any occlusion and $\hat{y}_{i,j}$ is the regressed angle when the centre 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.

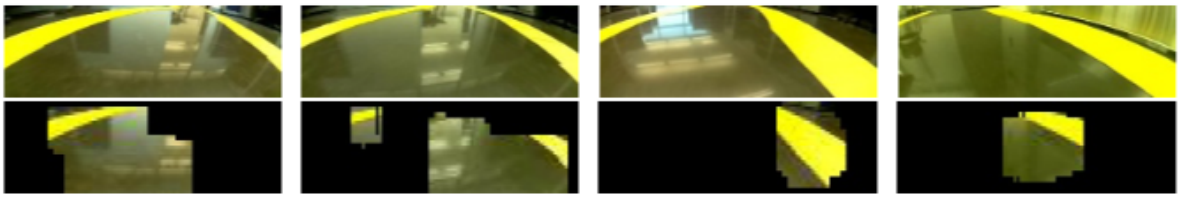


fig: Examples of occulation maps

Figure above 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.

Grad-CAM

Grad-CAM [6] is an extension of original work of Zhou *et al* [56], that lever- aged 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 $L^c_{grad-cam} \in_R w*w$ 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_k^c:

$$\alpha_k^c = \frac{1}{Z} \sum_i \sum_j \frac{\partial y^c}{\partial A_{ij}^k}$$

The weights α_k^c 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:

$$L^c_{Grad-CAM} = ReLU\left(\sum_k \alpha_k^c A^k\right)$$

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.

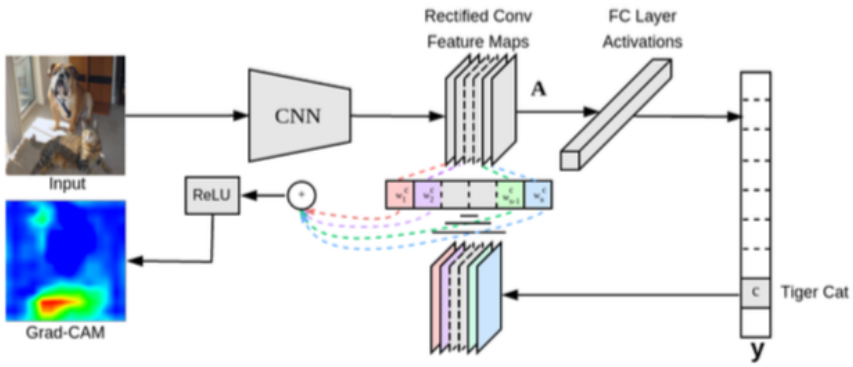
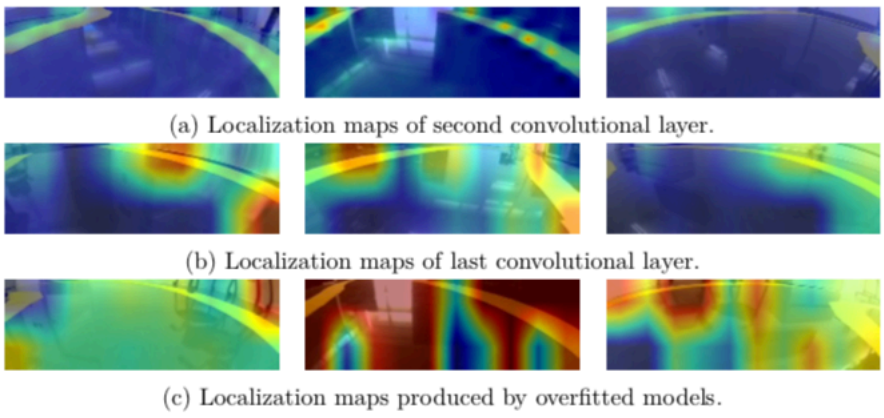


fig: The process of computing Grad-CAM for an image [6].

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 below:



(a) Localization maps of second convolutional layer.

(b) Localization maps of last convolutional layer.

(c) Localization maps produced by overfitted models.

fig: Examples Grad-CAM localisation maps

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 displays correct behaviour 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.

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 below 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

fig: Measured times

We can see, that one decision loop takes approximately 26ms to execute. That includes everything from capturing the image from camera, pre-processing 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.