



M Ű E G Y E T E M 1 7 8 2

OTDK

Felügyelt tanítás alapú járműirányítás vizsgálata
Examination of supervised learning-based vehicle control

Bagoly Zoltán

2023.

Konzulens:
Fazekas Máté

Contents

Kivonat	4
Abstract	5
1 Introduction	6
1.1 Acknowledgement	6
1.2 Motivation, input and output determination	6
2 Theoretical background	8
2.1 Supervised learning	8
2.2 Neural network	8
2.3 Loss	10
2.4 Learning rate	10
2.5 Data division, overfit	10
2.5.1 Vanishing and exploding gradients	11
3 Data and Neural network creation	12
3.1 Route data	12
3.2 CarMaker	12
3.2.1 Road model	12
3.2.2 IPGDriver	12
3.2.3 Sensors	13
3.3 Data analysis	14
3.4 Data preprocessing	22
3.4.1 Train - Test - Validation ration	25
3.4.2 K-fold	25
3.4.3 Scaling	26
3.4.4 Shuffling	26
3.5 Environment	27
3.6 Structures and hyper-parameters	27
3.6.1 Input and output layers	27

3.6.2	Structure	27
4	Training parameters and Inference	32
4.1	Parameters	32
4.1.1	Learning rate	32
4.1.2	Optimizer	32
4.1.3	Learning rate scheduling	34
4.2	Inference	34
4.2.1	Loss	34
4.2.2	Breaking loss down	34
4.2.3	Individual validation cases in original order	37
5	Validating the neural network	39
5.1	Transferring a neural network from PyTorch to CarMaker	39
5.2	Using the network in CarMaker	39
5.3	Simulations	40
5.3.1	Network and human-in-the-loop	40
5.3.2	Network used to make fully self-driving car	42
5.3.3	Network supervising different driving style	42
6	Summary	46
6.1	Objectives	46
6.2	Actions taken	46
6.3	Results	46
6.4	Improvement possibilities	46
6.4.1	Further development ideas	47
	Bibliography	48

Kivonat

Az autonóm vezetésben egy jelentős probléma a környezet érzékelése és útvonal megtervezése mellett, a megfelelő beavatkozó jelek, úgymint kormányzó, gáz-, illetve fékpedál állás meghatározása. Egy újszerű megközelítés a felügyelt tanítás alapú irányítás, melynek hatalmas előnye, hogy személyre szabható, tehát a neurális háló azt a vezetési stílust fogja reprezentálni, ami a tanító adatok felvételekor jellemezte a járművezetőt. Ez különösen előnyös lehet, ha a hálót nem önvezetésre akarjuk használni, hanem a vezető szokatlan manőverét próbáljuk detektálni és csak ekkor beavatkozva elkerülni egy esetleges balesetet.

A dolgozatban részletesen bemutatásra kerül a tanító adatok generálására szolgáló eljárás, mely a CarMaker szimulációs programmal történik. Fontos lépés elemezni az adatokat egyével és azokat összevetve is és megvizsgálni, elegendőek-e ahhoz, hogy a neurális háló által megvalósítani kívánt irányítás megfelelően megtanulható-e. Ezen elemzés alapján kerültek meghatározásra a háló bemenetéül szolgáló jelek. Ezután a háló legmegfelelőbb struktúráját kerestem meg és azt különböző paraméterekkel tanítottam, keresve a legjobb eredményt adó beállításokat. A tanítások sikerességét számos különböző metrikával vizsgáltam, ezeket is szemléltetem.

A fejlesztés eredményeként elkészült háló a tanító adatokkal történő validáció után kipróbálásra került a szimulációs környezetben. Leírom, hogyan lehet egy Pythonban előállt, Pytorch könyvtárat használó neurális hálót exportálni, majd importálni MATLAB-ba, ahonnan egy Simulink blokk segítségével használni lehet a CarMaker-ben lévő jármű irányítására. Az így elért útvonal követést alkalmazva a szimulációs környezetben az autó pontosan követi a kijelölt útvonalat, a kezelőszerveket hirtelen nagy változtatások nélkül használja. Kanyarok előtt a tanító szimulációhoz hasonló ütemben fékez és azokból hasonlóan gyorsít ki. Ez demonstrálja, hogy nem pusztán egy általános pálya követést valósítottam meg, hanem a stílust is tartalmazzák a háló paraméterei. Miután a 25 km-es útvonalon végig ment, a laterális hibát összevettem a CarMaker beépített driver-ével. Azt is megvizsgáltam, hogyan alkalmazható a háló egy ember hibájának kijavítására. Erre különböző teszteseteket állítottam elő, amelyekben a háló mindig a lehető leggyorsabban visszatért ahhoz a haladási stílushoz, amin tanult. Ezt a viselkedést kívántam elérni, ez a módszer így meg tudja előzni a veszélyes szituációba való kerülést azt figyelembe véve, hogy milyenek a sofőr szokásai.

Abstract

A significant problem in autonomous driving, in addition to sensing the environment and planning a route, is the determination of appropriate control signals, such as steering angle, gas- and brake pedal positions. A novel approach is supervised learning-based control, which has the huge advantage of being customizable, so the neural net will represent the driving style that characterized the driver when recording the training data. This can be especially beneficial if the aim is not self-driving but detecting an unusual maneuver of the driver and only then intervene to avoid a possible accident.

In the paper, the procedure for generating training data is described in detail, which is carried out with the CarMaker simulation program. It is an important step to analyze the data one by one and compare them to each other and see if it is sufficient to properly learn the control that the neural net wants to implement. The determination of the signals that serve as inputs to the net was based on this analysis. I then looked for the most suitable structure for the net and trained it with different parameters, looking for the settings that would give the best results. I have examined the success of the trainings with several different metrics, which I will also illustrate.

The resulting network of the development was tested in the simulation environment after validation with the training data. I will describe how to export a neural network created in Python using the PyTorch library and then import it into MATLAB, from where it can be used to control a vehicle in CarMaker using a Simulink block. Using the path following achieved in this way, the car in the simulation environment is able to accurately follow the designated path, using the controls without sudden major changes. Before corners, it brakes at a similar pace to simulation and accelerates from them in a similar way. This demonstrates that I have not only implemented a general path following, but the style is also included in the parameters of the net. After going through a 25 km long route, I compared the lateral error to CarMaker's built-in driver. I also looked at how the net can be used to correct human error. To do this, I created several test cases and in all of them, the neural net found the fastest way to get back to the learnt driving style. This is the behavior I wanted to achieve, this method can thus prevent getting into a dangerous situation, considering what the driver's habits are like.

Chapter 1

Introduction

1.1 Acknowledgement

Firstly, I would like to take the opportunity to say thank you to Fazekas Máté - fellow of SZTAKI Systems and Control Laboratory - for the endless advises, guidelines and that he is available and helpful anytime.

1.2 Motivation, input and output determination

In the following article a solution is to be presented to the situations where a driver of a vehicle makes a mistake or is entirely incapable of driving. Some use cases are when the driver

1. does not decelerate before a corner as much as they usually do or accelerates more aggressively than usual.
2. makes an unnecessary steering movement - away from the ideal path.
3. wants to make the vehicle to drive completely autonomously.

These cases have several sub-tasks to solve. Firstly the conditions of the current situations have to be known. Information is needed of the road ahead, the state of the vehicle compared to it and anything else needed to drive a car safely. These have to be done by some sensors. We should also be able to detect not only entirely dangerous actions of the driver, actions that suggest that the driver cannot perform as usual. This might be because they are drunk or fall asleep, but less radical situations might occur as well, when they do not see how sharp a corner is, or simply cannot react to the change of the path as quickly as they should in case of a sudden change in the road ahead.

All the above need a sort of driver profile built up and a logic that creates the control actions the driver would make according to the profile. These are necessary to compare the current actions of the human to.

If we have an accurate version of these, we can use them for complete self driving mode by simply using the output of the logic as control signal.

These modes serve accordingly to one of the main motivation of self driving: making driving safer by removing human error. Apart from that, self-driving cars and advanced driver assistance systems make life more comfortable and help to save time, which is important, because engineers are some of the few, that can sell comfort and time.

A novel approach to solve such complete tasks as modelling a specific driving style of a human is to do it by deep learning. A similar study was made for the International Conference on Intelligent Robots and Systems in 2017, they created a vehicle steering system by a neural network, supplemented by a Model Predictive Control. [Vehicle steering by NN and MPC] A neural network, that learns on the data recorded by the given driver in proper condition is a promising way of solving the cases above, because it has been shown it is possible to drive by it and if it accurate enough, it has the potential to adapt properly to the driving style of the human and produces outputs according to the style in new situations.

Experimenting with the usage of neural networks was also a personal motivation of mine, since this is a dynamically changing field with many new opportunities.

Chapter 2

Theoretical background

In this chapter, I briefly go through the technical terms necessary to understand the concept and solutions of the paper for anyone interested but not proficient in the area.

2.1 Supervised learning

Artificial intelligence is intelligence demonstrated by machines, as opposed to intelligence displayed by animals and humans. OED (OUP) defines artificial intelligence as: "the theory and development of computer systems able to perform tasks that normally require human intelligence, such as visual perception, speech recognition, decision-making, and translation between languages". [AI]

Machine learning is a subset of artificial intelligence. This term refers to the algorithms that enable systems to identify patterns, make decisions, and improve themselves through experience and data without human help. [Machine Learning] [Machine Learning Algorithms]

Deep learning (also known as deep structured learning) is part of a broader family of machine learning methods based on artificial neural networks with representation learning. [Deep Learning] This means that the logic of improvement is not determined by a human, only guidelines are given.

On Figure 2.1 we can see the relationship and the simplified definition. [AI - ML - DL]

There are many kinds of deep learning. In our case, we would like to mimic a certain style, which can be produced easily in large amount via simulation, so the best fitting is the supervised learning. The goal of supervised learning algorithms is learning a function that maps feature vectors (inputs) to labels (output), based on example input-output pairs. [Deep Learning]

2.2 Neural network

The mapping is done by a deep neural network in our case. Neural nets are computing systems inspired by the biological neural networks that constitute animal brains. [Neural Network]

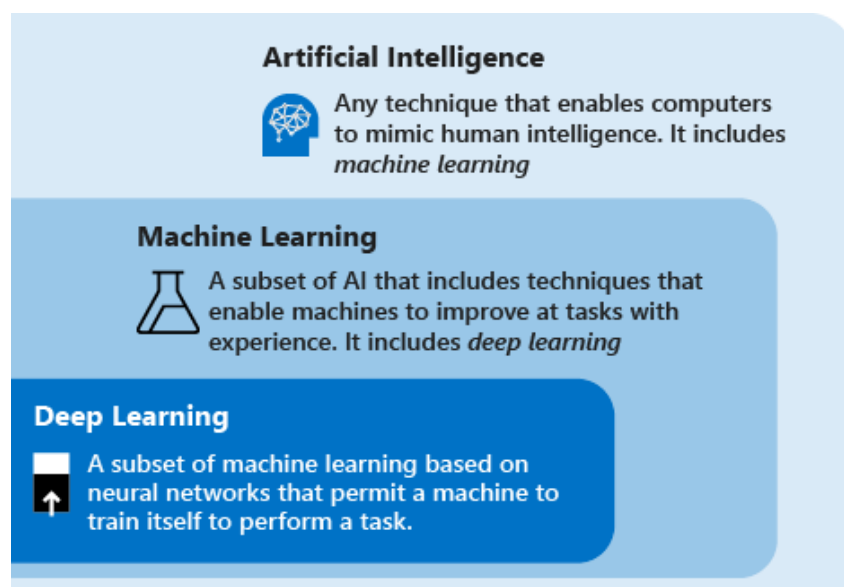


Figure 2.1: Relationship between artificial intelligence, machine learning and deep learning

They are built out of layers.

A fully connected layer - which we will use - has a set of input neurons and a set of output neurons. Each input neuron receives a numeric value. Then, each of them are connected to each of the output neurons (that is input neuron * output neuron connection). On all these connections, there are weights. The value the input neuron received is multiplied by this. The output neurons add up the numbers they get this way; and before emitting the number, they add a bias to it. This makes a layer have (input neuron * output neuron) + output neuron learnable parameters. Figure 2.2

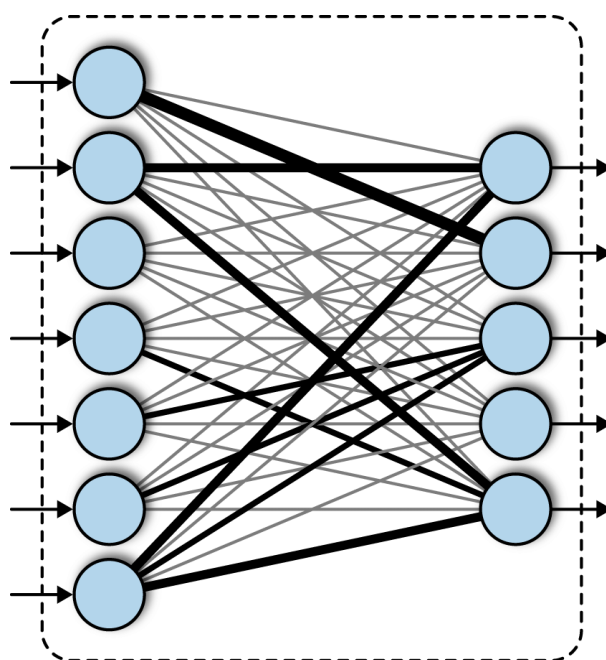


Figure 2.2: Visualization of a fully connected layer

2.3 Loss

The aim is to produce similar outputs at the end of the last layer (which gives the output of the network) to the labels (from the data we had, that we wanted to learn). Similarity is measured by the loss function.

Loss: a function that maps an event or values of one or more variables onto a real number intuitively representing some "cost" associated with the event. [Loss] It is possible to write our own loss function and use that, but PyTorch (the Python library providing us with convenient tools for deep learning) gives access to use a few prewritten ones. The loss functions goal is to be high, when the output is considered unacceptable and be lower as the net performs better.

2.4 Learning rate

We can lower the loss by changing the parameters of the network. We base our changes on gradient, which is computed by the backpropagation. The rate of the change is determined by the learning rate. This process basically tells us, what the ideal set of parameters would have been for that input-label pair and in which direction should the parameters move, to be in that state and by how much can we move towards that. Learning rate is a key element in training, because if we try to take too big steps, we might not find the best solution, we could jump over it. If it is too small, the training might take too long, or even worse, might get stuck in a local minimum and not find the global one. We can see these on from [Loss]

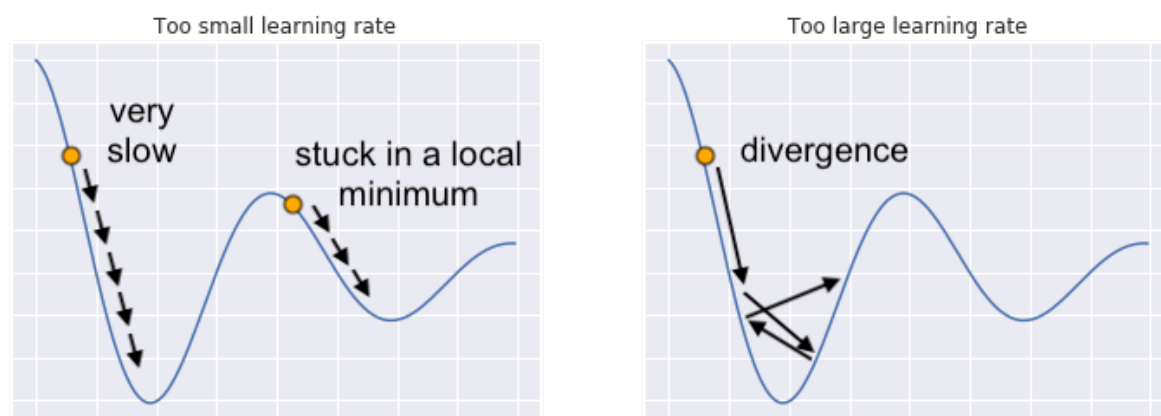


Figure 2.3: Too small and too high learning rate trying to minimise loss

2.5 Data division, overfit

While training, there should be at least 2 sets of data: 1 for training and 1 for testing. If the net updates its parameters on every data, eventually - with the good learning rate - it is going to learn every input and their correct output "by hart". This is not what we want, we would like a

generalised knowledge. To be able to monitor this, when the testing set is fed to the network, no updating should follow. This way, we can see when the train datapoints are starting to produce much lower losses than the test points - their loss will even rise not much after. This is the point when the net starts to overfit, i.e. know the training datapoints too well.

2.5.1 Vanishing and exploding gradients

Two standard problems worth mentioning are vanishing and exploding gradients. This happens, when one or few parameters are much lower or higher than the others. This will probably mean that heaps of information in the input are lost, because we either multiply them by numbers very close to 0; or because others might be so much higher they count by a much heavier weight, when computing the loss. I countered 2 cases, when this happened: the first is, when the network is too deep for the task. In this case, the first or the last layer overgrows and does most of the work, while the others are 1 or 2 scale below. The other is when a certain input is too high. Even tho the parameters are initialized on the same scale, the multiply the input numbers and if one of them is much higher or lower than the others, the chance of vanishing and exploding gradients.

Chapter 3

Data and Neural network creation

3.1 Route data

The base data for my work is a 25 km long route. X, Y and Z coordinates have been recorded by a real-life drive, which consisted of urban and rural parts as well. Thanks to this, this is a realistic representation of the general (non-highway) usage of a passenger vehicle. In the route there are uphill and downhill sections, roundabouts, longer straights combined with slight curves and areas, where there are 90° corners following each other every 30 meters.

3.2 CarMaker

3.2.1 Road model

I created the rest of the data in CarMaker, a vehicle simulation software. It is easy to load a route measurement in it and it creates a road model out of it. The user can set the number of lanes, the road markings and even the friction coefficient of it. The program generates a desired path on it. In our case, where there is only 1 lane, the path is the middle of it. A map of the whole path is on Figure 3.1.

3.2.2 IPGDriver

After I had the complete route, I could use the other features of CarMaker. It is equipped with a parameterizable, realistic driver model, that mimics the way a human would drive. The user can set the cruising speed the pilot. This does not mean the vehicle will move with this speed all the time, if it is necessary, the driver will slow down. In this program, path does not mean the vehicle is going to move directly on it, it only follows it within constraints. It is possible to set the strictness of the constraints the chauffeur will use on a scale of 0 to 1. 0 means it takes corners with the lowest possible deviation, keeping the center of mass of the car directly above the path. 1 means it even moves away from the corner before entering it, so it can follow a

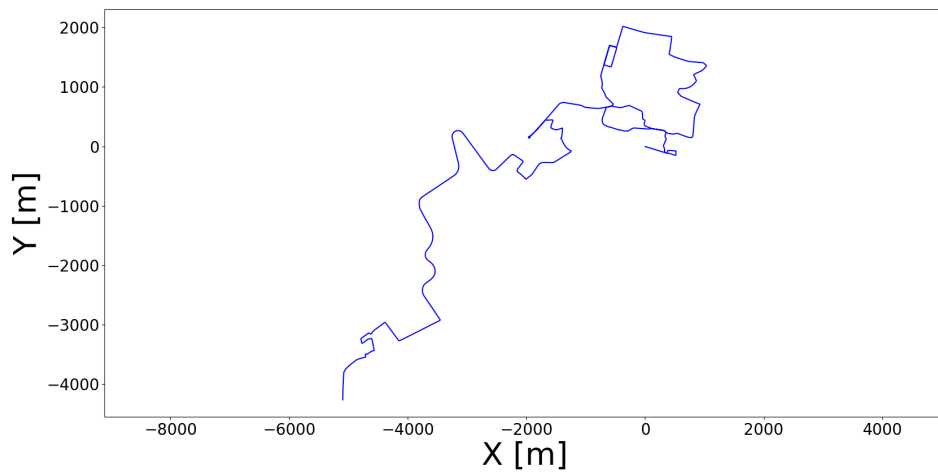


Figure 3.1: Path of the simulation

really smoothed version of the path. The difference between the two settings is visualized on Figure 3.2.

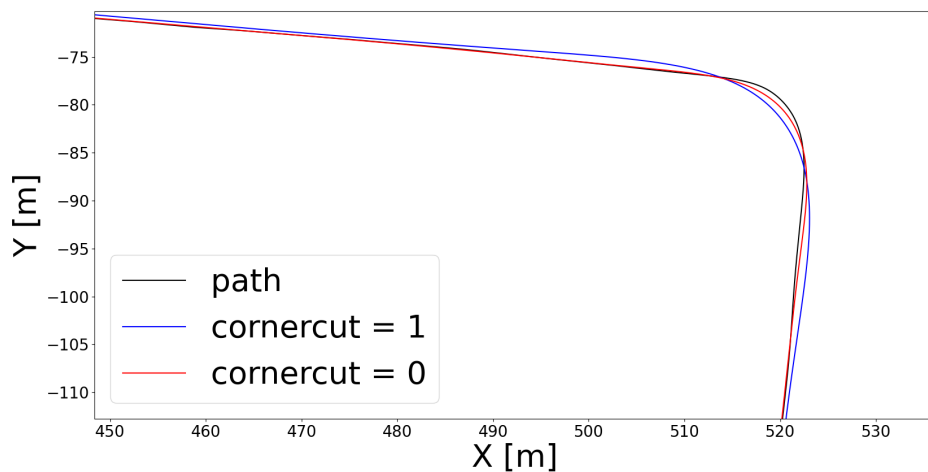


Figure 3.2: Difference between path following with cornercut being 0 and cornercut being 1

An other attribute of the driver model is how quick accelerations it uses. Braking in front of a corner, accelerating back to the desired speed, cornering, a 2 dimensional acceleration map defines how the driver will maneuver from one straight section to an other.

3.2.3 Sensors

It is also possible to equip the vehicle with numerous kinds of sensors, such as camera, lidar, radar and so on. If they are on a car in a simulation, the program generates their output and it is possible to work with them. I used the ideal road sensor, which works as follows: it outputs 2 values in every moment. The deviation distance, measured between the middle of the front

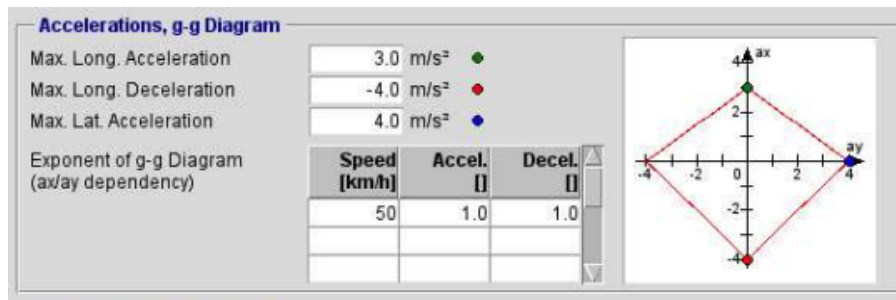


Figure 3.3: Acceleration settings for IPGDriver

axle and the closest point to it on the path and the deviation angle between the heading of the vehicle and the tangent of the path at the chosen preview distance. The way they are calculated is on Figure 3.4 (from the documentation of CarMaker) for easier understandability.

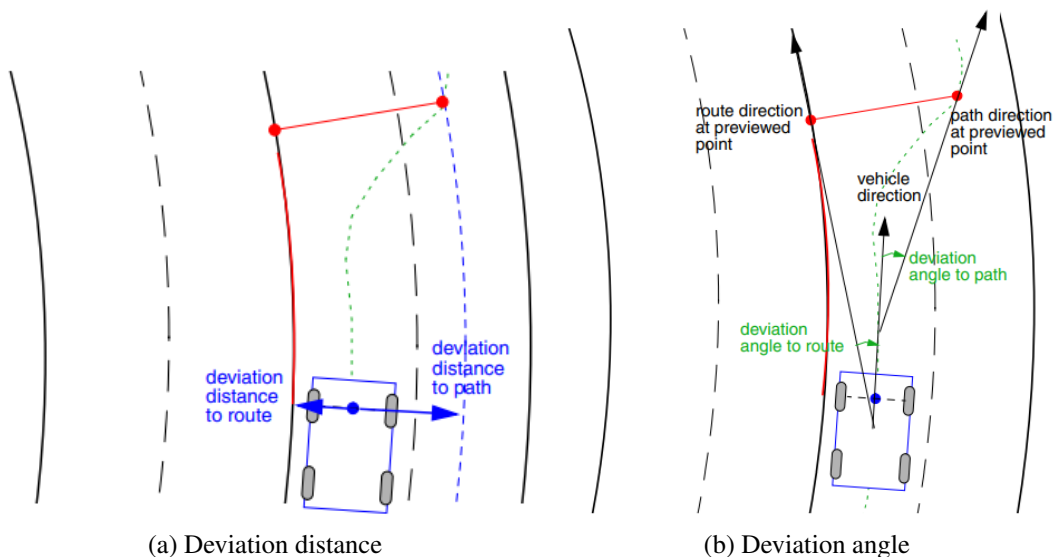


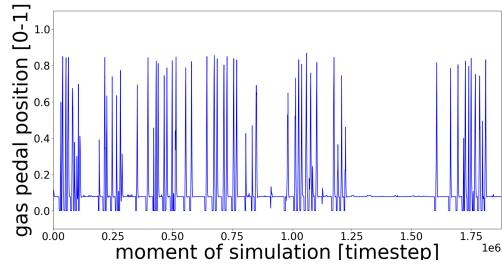
Figure 3.4: Road sensor

3.3 Data analysis

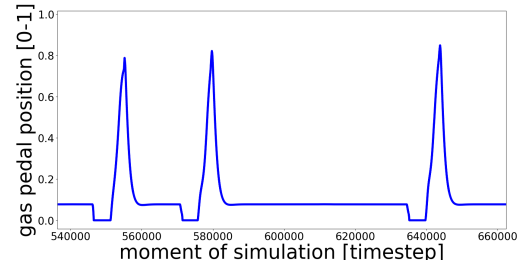
While running a simulation, saving values to a MATLAB workspace is possible. CarMaker can be connected to Simulink and with the help of some simple blocks, values from every timestep are transferred into MATLAB. I choose the length of the timestep to be 0.05 seconds. CarMaker gives access to a lot of attributes from every moment, like the angle of each wheel, the force being used on each wheel, the orientation of the vehicle and so on, the goal was to find signals that hold the information necessary for creating the control signals: gas position, brake position and steering wheel angle.

Firstly, I inspected the control signals throughout the simulation. The gas and brake pedal positions are between 0 and 1. You can see these on the graphs below. For each, there is a figure

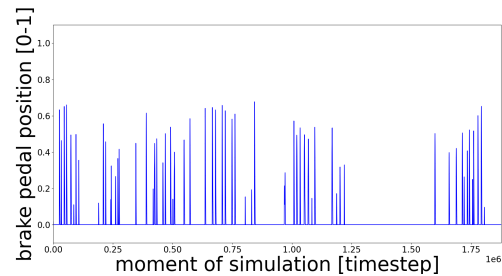
along the whole simulation to get a comprehensive understanding of the signal: Figure 3.5a and Figure 3.5a. There are zoomed in versions as well to see the characteristics of it: Figure 3.5b and Figure 3.5b.



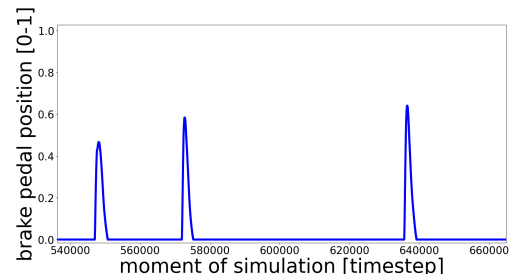
(a) Gas pedal position throughout the 25 km



(b) Gas pedal position zoomed in



(a) Brake pedal position throughout the 25 km



(b) Brake pedal position zoomed in

For most figures, where there is a zoomed in version as well, I try to choose the 540000 - 660000 moment domain, so I can show a complete picture. This section is this part of the path: Figure 3.5

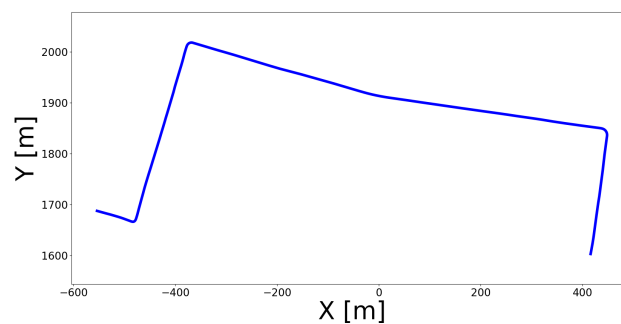


Figure 3.5: Representative path section

The wheel angle signal is the relative angle of a front wheel compared to the body. The average of the left-front and right-front values are plotted with blue. This value is always 1/13th of the steering angle in this vehicle model, so we can calculate that as well, if needed. On Figure 3.6, there is also a red line, showing the angles after sorting them. This shows that most of the simulation is on a straight line, which represents real life usage, but also warns

that using all the data points without any kind of filtering might lead to the situation where the network only learns how to drive in a straight line, because that kind of moment is clearly in heavy majority.

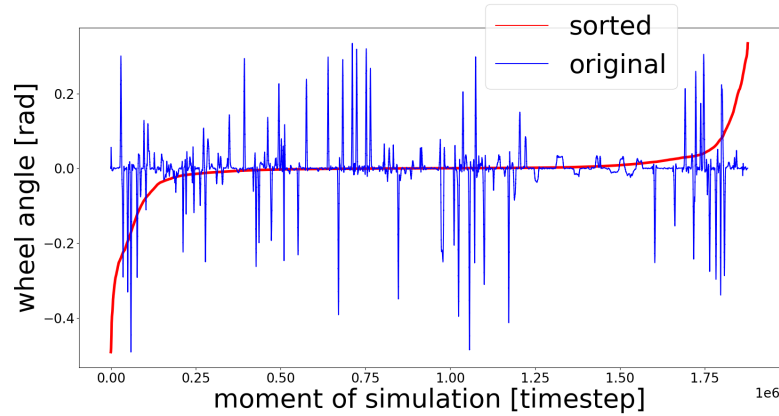


Figure 3.6: Wheel angle throughout the 25 km

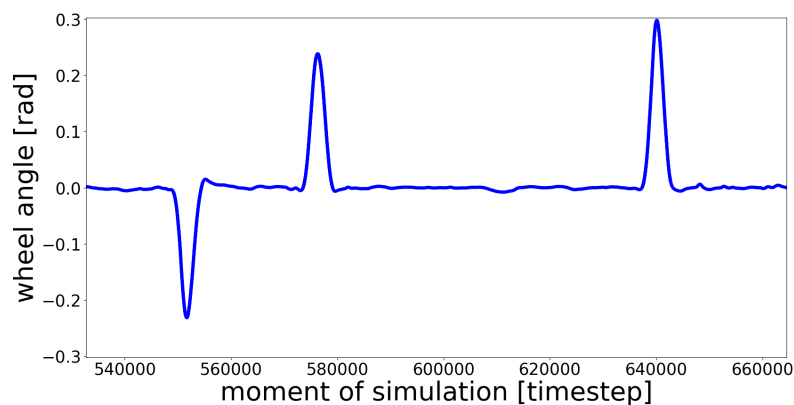


Figure 3.7: Wheel angle zoomed in

The aim is to predict these signals as accurately as possible with the neural network.

The inputs of the net need to contain the information to produce the control signals. The task is to characterize the state of the environment and the vehicle in every moment. It is clear that the current velocity is needed, without it, predicting the pedal positions would be impossible, but it is also helpful to get the wheel angle, because the vehicle reacts differently at different speeds.

Pedal positions and wheel angles were already on different scales, but velocity is a huge step higher compared to both of them, it is clear that some kind of scaling is necessary before feeding these values to the input layer of the network.

To be able to steer the vehicle, we need to know the curvature of the road. There would be a numerous ways to get this, for example knowing the coordinates of the road, getting their derivative and rotating it by the current heading of the vehicle; but the road sensor is perfect for

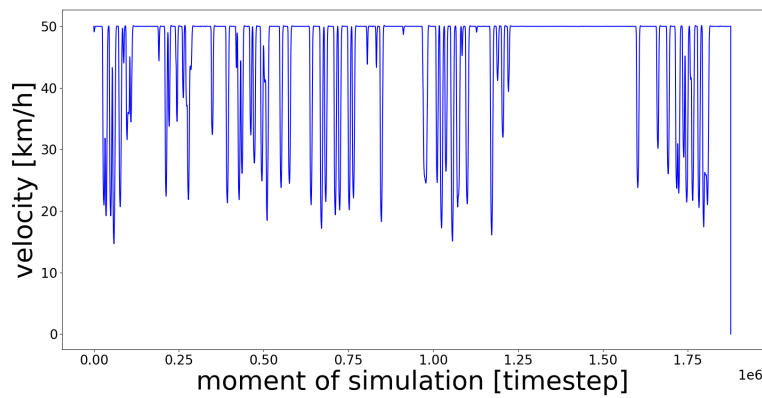


Figure 3.8: Velocity throughout the 25 km

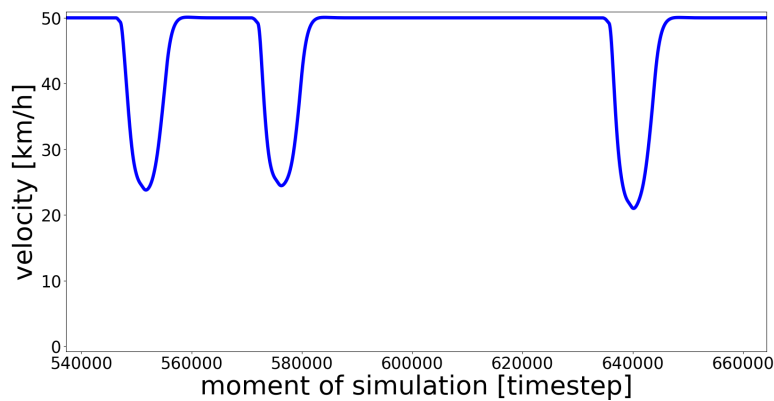


Figure 3.9: Velocity zoomed in

us, since it gives the difference between the heading and the path orientation. We can also set the preview distance, so the described point is always exactly as far ahead as we want it to be.

This signal should also give the needed information for the longitudinal control. If the preview distance is high enough, it will indicate that there is a corner ahead. This probably will be more responsible for giving information for pedal positions than wheel angle position.

On Figure 3.10 and Figure 3.11 there are plots of some road sensors deviation angle signals with different preview points and the wheel angle for reference, because the corners are probably very close to where the wheel angle is higher, since the car is not sliding.

As you can see, the road sensor with 50 meter preview (dark blue line) gives high signal long before corner entry - where the wheel (black line) angle parts from 0 - so it is probably useless for steering but more useful for braking. In contrary, the signal with 0.5 meters preview is quite close to be the inverse wheel angle. However, both are going back to 0 only at the corner exit, because the vehicle's heading at the closing stage of the corner is different from the tangent direction of the straight after, and that direction is the same 0.5 meters and 50 meters after the corner, because both are describing the straight section and in most cases there is no following corner within 50 meters.

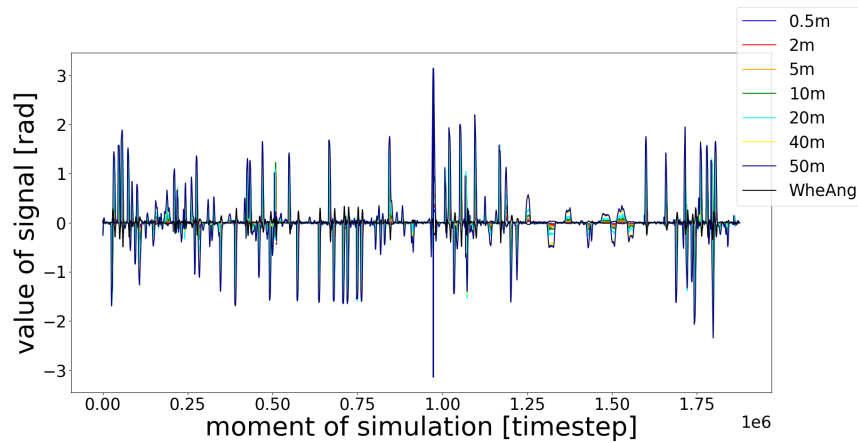


Figure 3.10: Deviation angles and wheel angle throughout 25 km

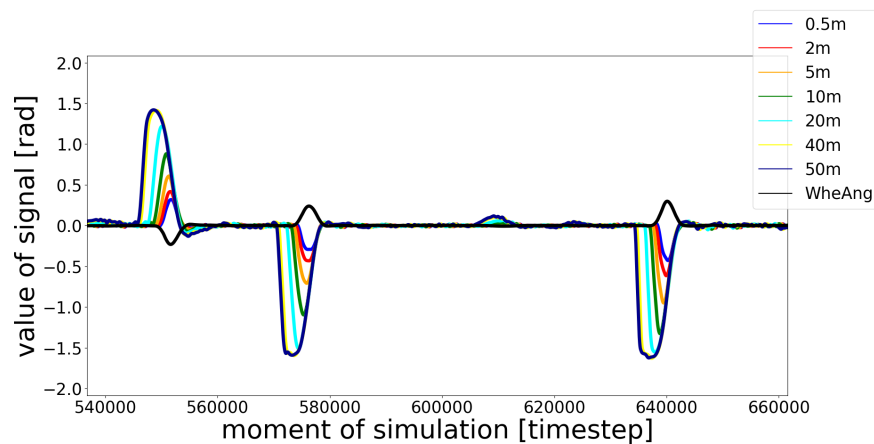


Figure 3.11: Deviation angles and wheel angle zoomed in

To be able to correct mistakes in the control and mimic the given cornercut rate, we also should know the lateral error of the vehicle compared to the path. This is the deviation distance signal of the road sensor. This value will be the same for every sensor in every moment, we can see it from the definition as well, but I also tested it to be sure. On Figure 3.12 we can see a simulation with cornercut being 0. On Figure 3.13 the signals of the different sensors are definitely plotted, but they are exactly the same, that is why only one line is visible. However, if we change the cornercut rate, the deviation distance will be massively different. As you can see on Figure 3.14 deviation distance grows with cornercut. The average of it with cornercut set to 0 is 0.080 meters, while with cornercut equal to 1 it is 0.541 meters. An other thing to take away is that the red line has more values. This is because the acceleration settings (Figure 3.3) were the same in each case, so of course, the driver model had to slow down more to take the same corner with sharper following and the same lateral acceleration.

The intuition is that these input signals have the necessary information to create the outputs.

The first version of neural network was dedicated to output only the wheel angle. The initial idea for the set of inputs for this network was to use one road sensor with an optimal preview distance to get information of the section ahead and the direction of the vehicle's movement.

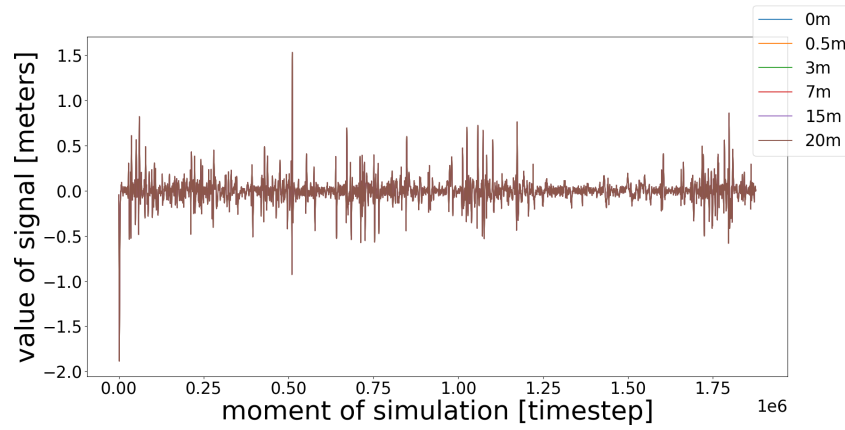


Figure 3.12: Deviation distance of a few sensors throughout 25 km

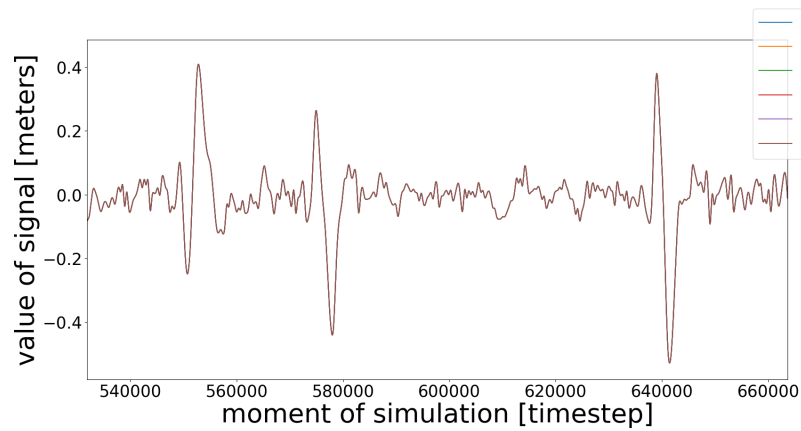


Figure 3.13: Deviation distance of a few sensors zoomed in

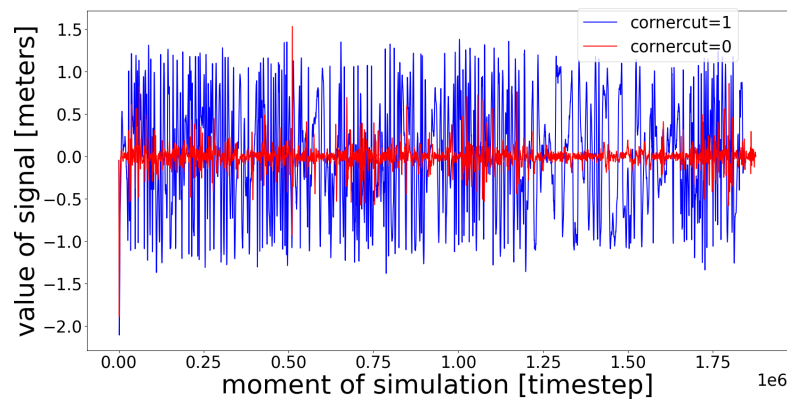


Figure 3.14: Deviation distance with cornercut 0 and 1 throughout 25 km

In order to validate this, I created an inspecting grid. The grid was 2 dimensional: deviation distance and deviation angle. Data points (moments for the simulation) were categorised based on these attributes. On the deviation distance dimension there was 24 categories and 60 on the deviation angle. This meant that every category had car states not further than 15 cm-s and 1.3° from each other for the simulation where cornercut is 0. On the X axes, there are the

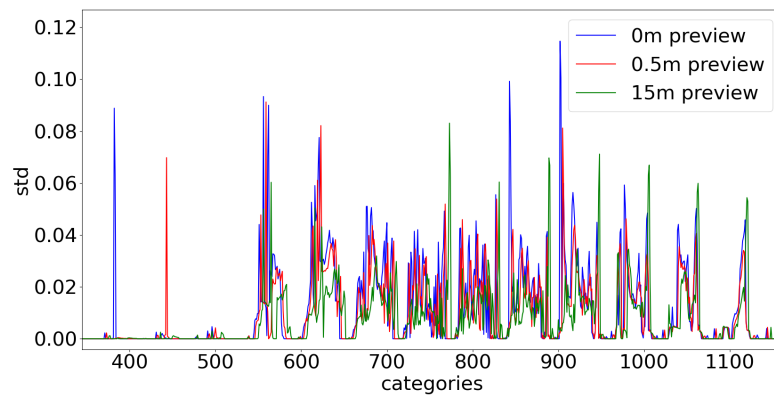


Figure 3.15: Data grid comparing 3 road sensors

different categories. There are $24 * 60 = 1440$ of them. Many of these are empty or almost empty of course, because there are really rare situations both distance- and anglewise as well. For example all the categories, where the lateral offset is more than 1 meter are going to be underpopulated. On Figure 3.15 I am showing the more populated ones.

On the Y axes, is the standard deviation of the wheel angles (in radian) in the category.

As we can see, for a huge amount of situations, despite the inputs are looking rather similar, the outputs have quite high variance. When the std is 0.04, the variance is 0.0016. This would be the value that we could get if every wheel angle in a group would differ from the average angle by more than 2° . To improve this, I supplemented the input set by the current velocity. The grid became 3 dimensional, having 6 slots for different velocities in each of the former categories - meaning there has to be less than 6 km/h difference between 2 data points to be in the same box. This divides the data into $24 * 60 * 6 = 8640$ groups. The result is on Figure 3.16. This does help, the average deviation is lower. This means that the velocity of the car has

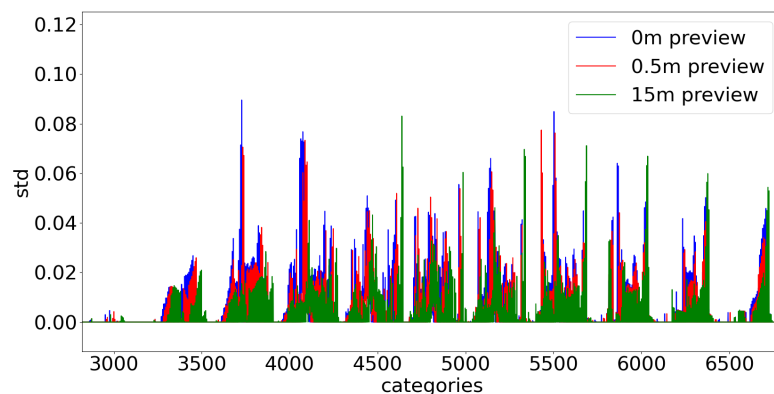


Figure 3.16: Data grid comparing 3 road sensors in different velocity states

useful information for the steering part of the task as well.

To understand, why are there differences, I plotted the wheel angles of categories, where standard deviation was higher. An example is is on Figure 3.17 This particular box was created

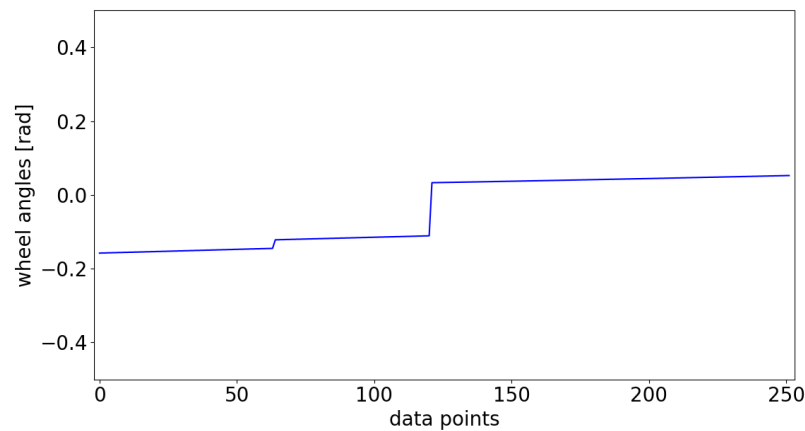


Figure 3.17: Wheel angles from the same category

by the road sensor with preview distance set to 0 m. The data points in have deviation distance between 46 cm-s and 31 cm-s to the left of the path; and have 1.2° and 2.5° deviation angles to the right. The velocity was between 20.5 km/h and 26.5 km/h. The lowest wheel angle value is -9.01° the highest is 3.00° . The average in this category is 2.45° , the standard deviation is 0.0895. We can guess that there are 3 different situations, when this kind of state occurs, that is why there are 2 steps in the plot.

To understand how such different situations can realise, I made plots of those moments, where there are high differences in wheel angle within a box.

On the 1st kind of diagram Figure 3.18 we can see the moment, where the smallest wheel angle was needed in the category. The red line is the path, the red dot is the center of the car. The orange dot marks the position of the sensor, this is the middle of the front axle of the car. The black dot shows which direction are the wheels pointing at. In this case, the wheel angle is slightly to the right. The brown dot points out the part of the path, where the deviation angle is calculated at, the preview point.

The 2nd kind is were similar, this is the biggest wheel angle in the category. Figure 3.19 The blue line: path, blue dot: CoG of car, lightblue: sensor position and black is the wheel direction again. In this case, it is slightly left. On the 3rd diagram Figure 3.20, there are plots of multiple road sensors deviation angle values in the minimal and maximal moments. The red is the minimal, the blue is the maximal. This proves that the information of the road ahead is in the collective of the sensors much more accurately, than in only one sensor, indifferent to where the preview point is. These plots convinced me that it is worth using multiple road sensors at the same time even if it is only used for the steering.

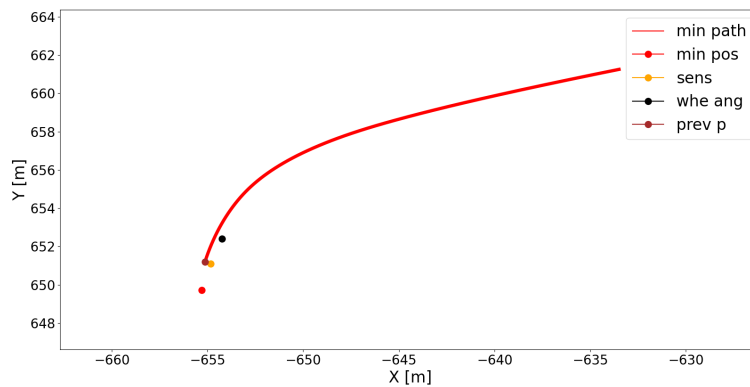


Figure 3.18: Situation with minimal wheel angle in the category

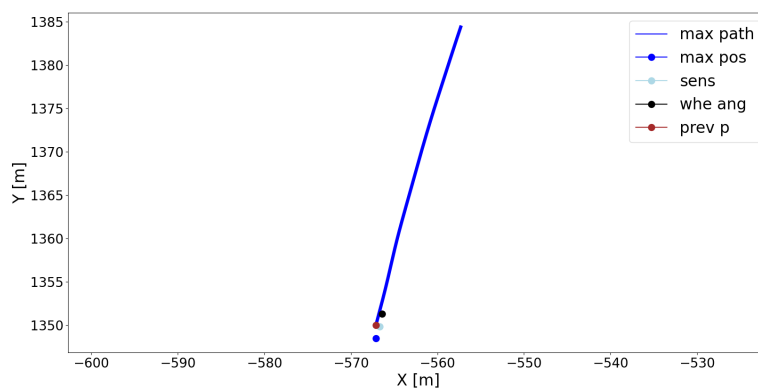


Figure 3.19: Situation with maximal wheel angle in the category

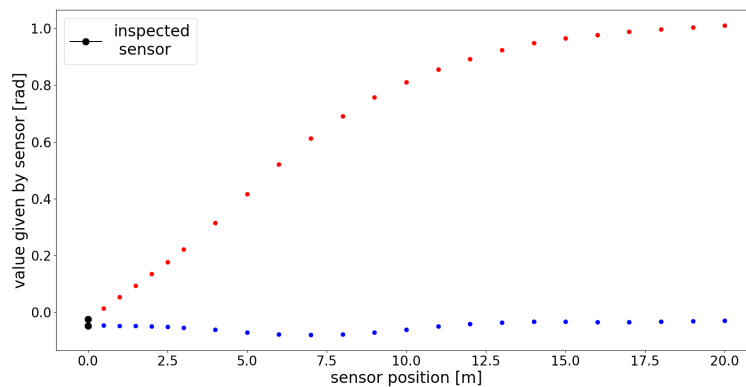


Figure 3.20: Values of many sensors from that moment

3.4 Data preprocessing

CarMaker supports MATLAB, so we can access the values of every moment by a Simulink block and we can send them into the MATLAB workspace. From there it is easy to save the values to a .mat file and load them into a python code. I made a script, which is dedicated to form the data into a shape that is usable for deep, supervised learning and then save it.

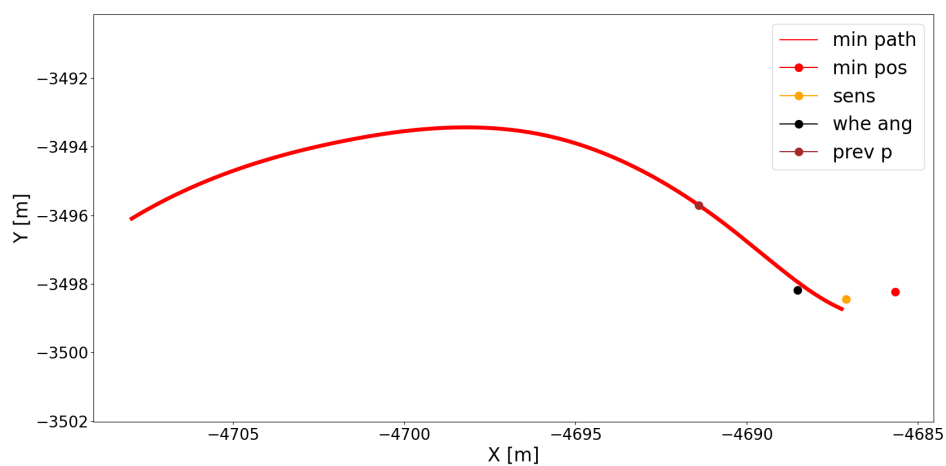


Figure 3.21: Situation with minimal wheel angle in the category

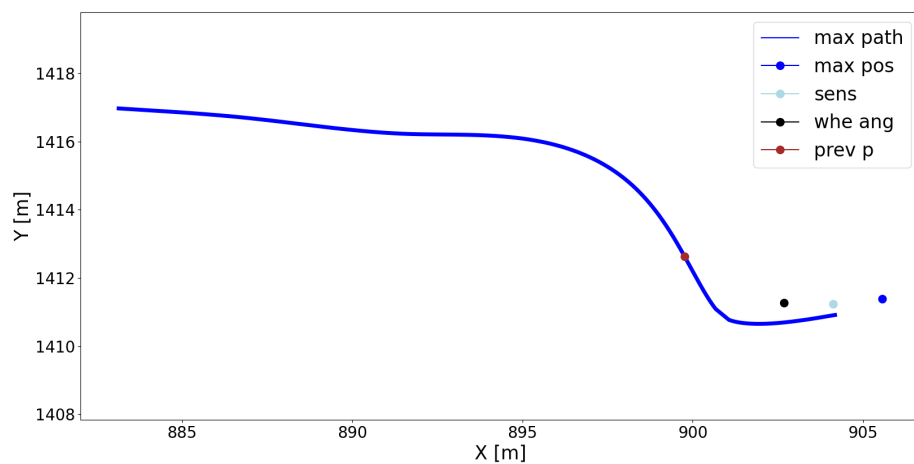


Figure 3.22: Situation with maximal wheel angle in the category

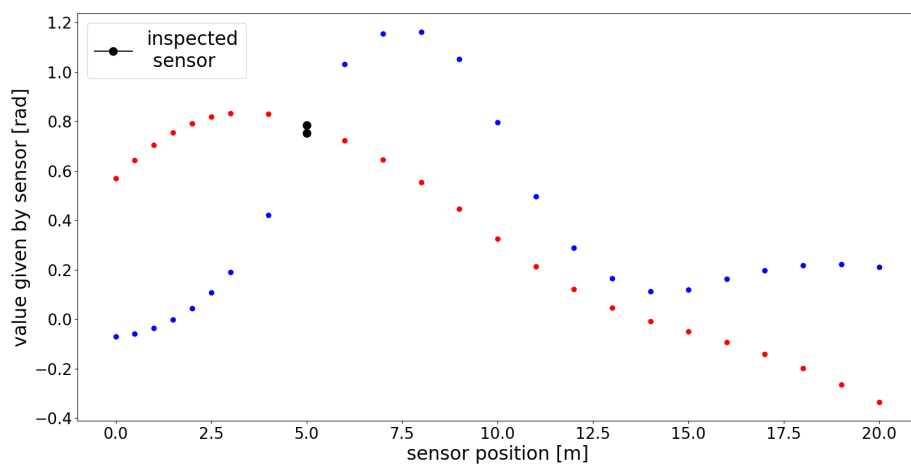


Figure 3.23: Values of many sensors from that moment

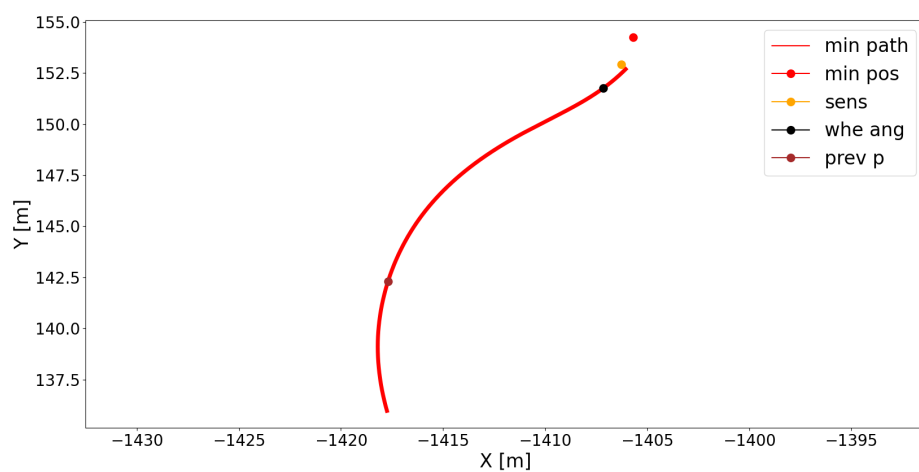


Figure 3.24: Situation with minimal wheel angle in the category

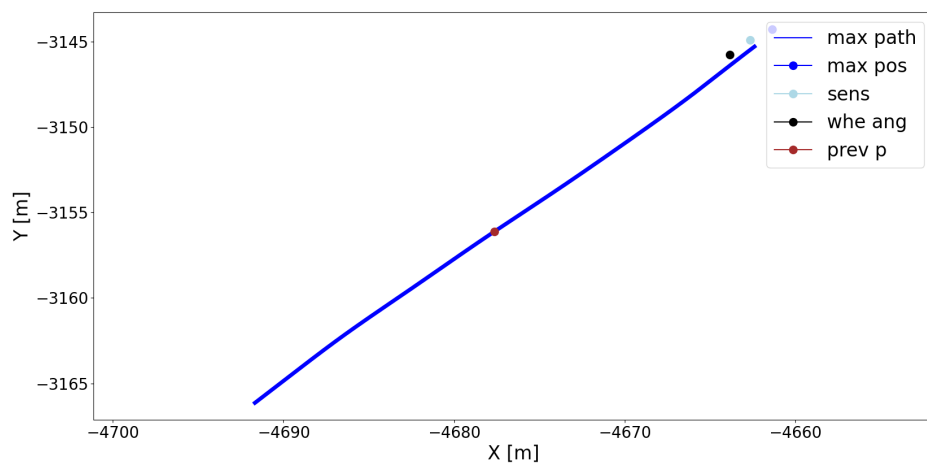


Figure 3.25: Situation with maximal wheel angle in the category

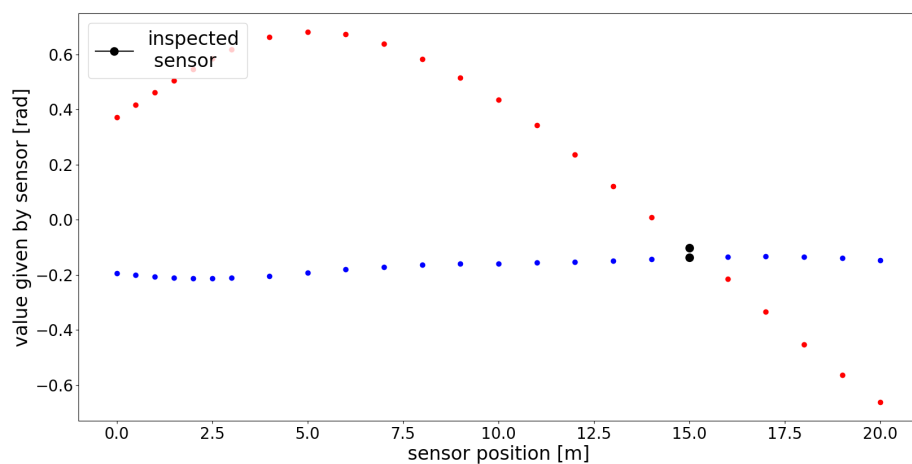


Figure 3.26: Values of many sensors from that moment

Every data point describes a single moment from the 25 km long simulation, consisting of the input: 1 deviation distance value, each sensor's deviation angle value and the velocity; and the label: the average front wheel angle and the gas and brake pedal positions.

3.4.1 Train - Test - Validation ration

As I already mentioned, timestep is set to 0.05 seconds, so around 36,000 datapoints are created with a simulation. These have to be allocated to 3 sets:

1. Training data: most of the data has to be here. The neural network is going to see these repeatedly while training and we will update the parameters of it based on the loss on these input-output pairs.
2. Testing data: a smaller proportion of the dataset. These are used to monitor the course of the training, by regularly testing the knowledge of the net on them. (Although much less often than the training data.) One must not update parameters while testing. This is to make sure the net uses generalized knowledge, not just optimizes parameters on tests as well as training data.
3. Validation data: another small set, also not used for updating. This one however is only used once, after training. This is needed because we are setting the hyper-parameters between trains to get the smallest possible loss on test data, but it is possible to set up the hyper parameters in such a way that they work well specifically on the test set.

There are conventions for the rates between these, a popular one is 85% train, 10% test, 5% validation. However this is practical for regular usage, because 10% and 5% are enough to be sure the knowledge is general, I used a different one. I sacrificed training data to have more validation. This is because I output control signals and later I will show that it is possible to put them back into CarMaker and drive the car with them. To be able to have a long enough validating section, where most type of corners are present, it divided the data to 60% train, 20% test and 20% validation parts. Figure 3.27

The validating part is a separate section of the 25km long path, so when the vehicle is driven on it, we know for sure, no parameter updating was made anywhere near. The training and testing moments are mixed, from every 4 points, 1 is a test, the others are train points. This creates a representative test set, however brings the danger of having train points too similar to test points and to make us not sure if the test is well done because the generalisation is good or there are too similar train inputs. This is also why we have the validation set.

3.4.2 K-fold

Of course these test and validation sections can still be specific, they are less than 3rd of the whole data combined. This brings the danger of not realising that in some situations the network



Figure 3.27: Train test and validation sets throughout the 25km:
green: validation, red: train, blue: test

gives unacceptable outputs, because they are unrepresented in the non-training sets. We can look at the loss on the train data, but that might be low solely because of overfitting. A better idea is to implement K-fold testing. This means that the whole data is divided to K parts and every set of hyper-parameters are trained K times, in each case the testing data is a different part. In my implementation, K was 5. When a set of parameters resulted in low losses in all trainings, I chose a different set to validate and retrained the network to be sure nothing is specialized.

3.4.3 Scaling

Evidently, the values to learn from differ heavily in turns of scale. The pedal positions are between 0 and 1, the angles are between $-\pi$ and π the velocity however is usually above 10 (even in m/s as the CarMaker outputs it). This kind of difference can cause gradient vanishing or exploding while training, so scaling the inputs and outputs to the same level is necessary. I experimented with the -1 to 1 region, but in the end following the convention of scaling between -3 and 3 worked better. It is also important to mention that although wheel angle is between $-\pi$ and π by definition of vehicle model, we should use what we learnt from inspecting the data: it hardly ever has higher absolute value than 0.3. When making the training-testing-validating data, I upscale wheel angle by 10. Deviation angles are multiplied by 3. The reason for this difference is on Figure 3.11. The pedal positions are not scaled up by 3, as definitions would suggest, but by 4. The training is successful when is done by 3, but the loss on these kind of data is significantly better if I use 4. I multiplied deviation distance by 5 and velocity by 0.4. This is why it is essential to look at the diagrams at the beginning of this section.

This of course means that the neural network will give its outputs as close as possible to the scaled outputs, based on the scaled inputs. When we use the network in a different environment, on different data, we have to scale them as well, before feeding them to the input layer and multiply the output by the same numbers to make the net work as intended.

3.4.4 Shuffling

After every datapoint's each value is scaled and all points are categorised to be train, test or validation data, we have to shuffle each category. This is because the network might not see the whole category, but just a small batch of it, update parameters based on it and move on to a next batch. What happens if we do not shuffle is that the network learns the track's first part

and then overwrites it seeing the 2nd part and so on in each epoch, because the characteristics changed as we got further on the track. If we shuffle, each batch will have multiple kind of datapoints from every part of the track, making it easier to generalise. After this is done, I saved the portions into Numpy files, so I can load them each time I want to train.

3.5 Environment

The neural network was created in python, using the PyTorch library, since it gives a convenient interface in creating layers, connecting them, writing forward functions and setting up an environment for training and evaluating nets. I worked in Spyder and PyCharm.

3.6 Structures and hyper-parameters

3.6.1 Input and output layers

The input is an $1 * (N + 2)$ sized tensor where N is the number of sensor used. These values are the deviation angles of each sensor, plus the deviation distance and the velocity. The label is a $1 * 3$ sized tensor: wheel angle, gas, brake. These can be the inputs and outputs of the network. This means that a fully connected layer used for taking the inputs of this task must have (number of sensors) + 2 input neurons and a fully connected layer used for giving the outputs should have 3 output neurons.

3.6.2 Structure

The approach was to solve the problem by a fully connected neural network. This type of net is often avoided because the number of parameters grow quickly to a point where the training gets too demanding in time, but after some trials I concluded that the optimal network size is not too large to train on my laptop in a reasonable time.

The way I tried to find the best size and shape is as follows. I used networks consisting of layers of the same size.

First, I fixed the number of input neurons per layer (The number of output neurons are equal to that in each layer to make it possible to put them after each other) at 32 (this will be the width of the network), since this is the closest to powers of 2 to the number of inputs - and it makes the GPU work quicker if it operates with such numbers - and tried networks with increasing number of layers like this. It became clear that different nets work best with different parameters, e.g. learning rate, so the first step was to find a good enough parameter set for a network to be able to judge its performance. The details of my way of doing this task and the results are in the next chapter, because after finding the best hyper-parameter set, I started the process over, but with smaller steps, because I already knew the region from finding it here.

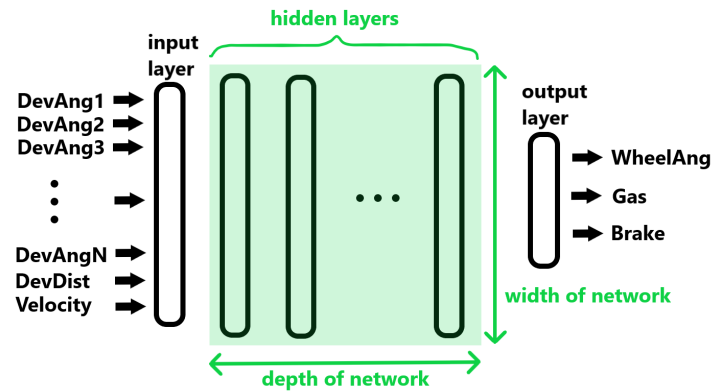


Figure 3.28: Generic network - we are trying to find the ideal depth and width

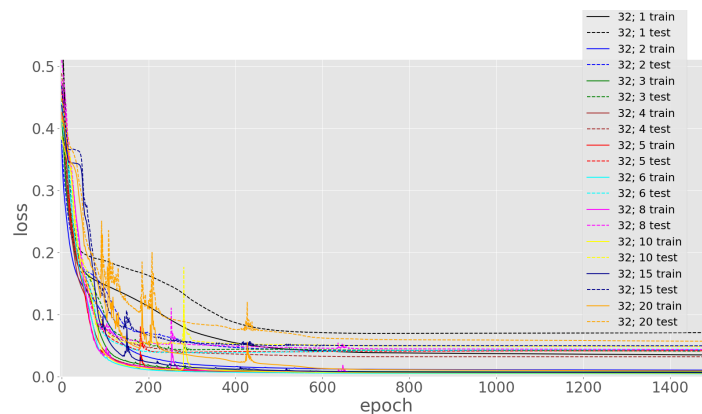


Figure 3.29: Mean Squared Error throughout the training of 32 wide nets

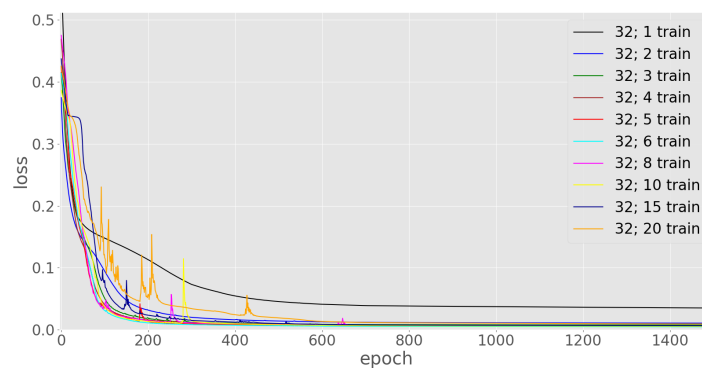


Figure 3.30: Train Mean Squared Error throughout the training of 32 wide nets

Based on results visible on Figure 3.29, Figure 3.30 and on Figure 3.31 it looked like the optimal depth is around 3-5 hidden layers plus the input and output layers. This resulted in the conclusion that generally, the rather shallow networks work best on this task, which is not impossible, because although deeper networks have more parameters, so in theory they also have the potential to reach the same knowledge as the more shallow ones (we can simply imagine this by creating a deep network of the shallow net and then layers with parameters like

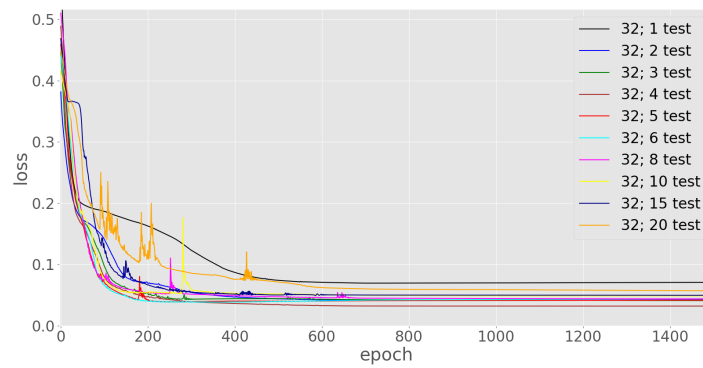


Figure 3.31: Test Mean Squared Error throughout the training of 32 wide nets

this: all weights are 1, all biases are 0), in practise, deeper network are more prone to gradient vanishing or exploding, so they are harder to train to the same level on some tasks. We can see this better on Figure 3.32 and Figure 3.33.

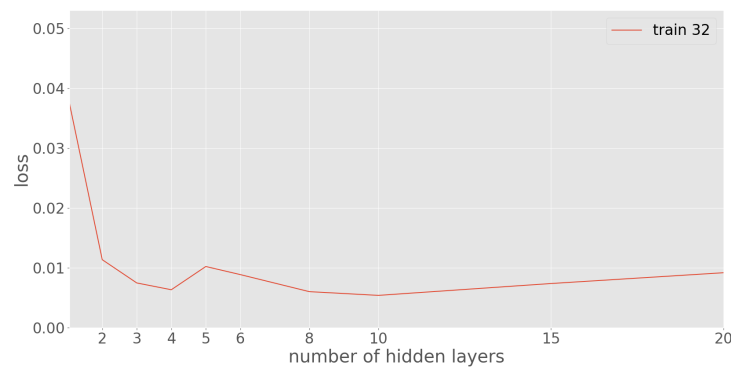


Figure 3.32: Train Performance of best networks of different shapes with 32 width

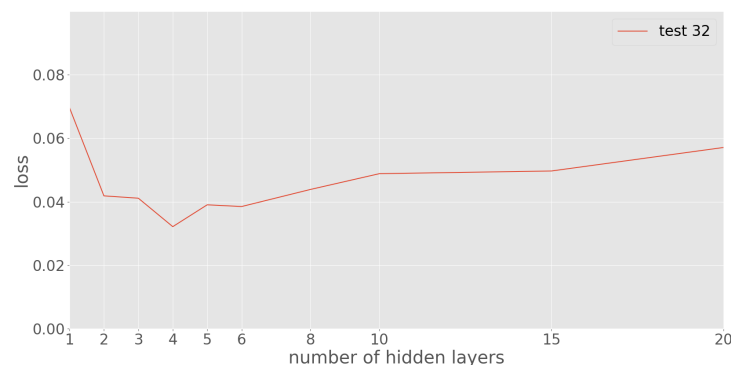


Figure 3.33: Test Performance of best networks of different shapes with 32 width

On the X axes, we see the depth of the networks, on the Y, the the average loss of the 10 best epochs (so the 10 best versions of that shape from the whole training). It is necessary to

pick out these best versions and not just use the last 10, because some might overfit, meaning the loss raises towards the end. This is also the reason I saved the networks regularly while training, not just at the end of it. The different coloured lines are different depths. This kind of chart is easier on the eye than having to look at countless loss graphs throughout the training.

After finding a seemingly optimal depth, I tried increasing and decreasing the number of neurons in the layers. Obviously, every new shape had their own research for initial learning rate and for many of them I tried different scheduling technics and optimizers to be sure. This way I found the best working width for the network to be 32 with the help of these figures: Figure 3.34 and Figure 3.35

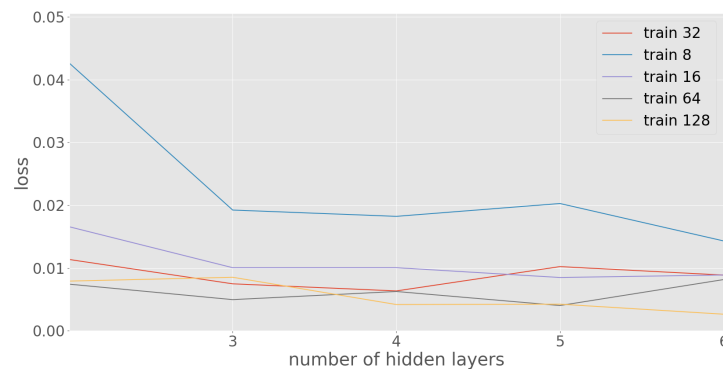


Figure 3.34: Train Performance of best networks of different shapes

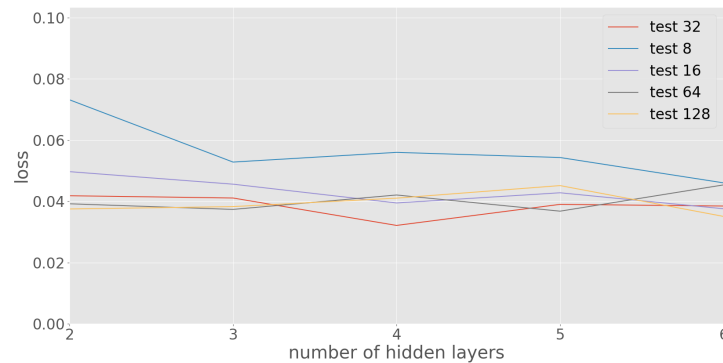


Figure 3.35: Test Performance of best networks of different shapes

As we can see, the test performance of the 32 x 4 net is the best and the train performance of it is the 2nd best, so it has potential to learn and generalises well enough already, so in conclusion, the 32 wide, 4 deep neural network seemed the best. It goes trough the data 1500 times in 9 minutes and 30 seconds on my laptop and has 7683 trainable parameters:

$$\begin{aligned}
 \text{NumberOfParams(layers)} &= \sum_{\text{layer=inputLayer}}^{\text{out putLayer}} \text{weights} + \text{biases} = \\
 &\sum_{\text{layer=inputLayer}}^{\text{Out putLayer}} \text{layer}_{\text{inputNeurons}} * \text{layer}_{\text{out putNeurons}} + \text{layer}_{\text{out putNeurons}}
 \end{aligned}$$

This network is what I try to optimize by tweaking the training parameters to lower the test performance even more.

Chapter 4

Training parameters and Inference

4.1 Parameters

In this section, I show how I iterated towards the best set of training parameters of a given net structure. These parameters influence, how the learning parameters of the net change over time in training. They can be initialized manually by the initialization function of PyTorch. [PyTorch]

4.1.1 Learning rate

The definition of learning rate is in the Theoretical background part. Here I want to mention that I experienced that higher learning rates work better with network with fewer parameters. The bigger networks perform better with smaller learning rates.

What happens when the learning rate is too high is that the net can jump into a local minimum and can not get out of it, as I also already discussed. In my case to realisation when, because the learning rate was high, and most moments of the training data are on straights as we stated in the inspection on the wheel angle signal. These two meant that the network found the local minimum by outputting 0 for wheel angle, 0.06 for gas position (the amount of gas needed to counter friction on the road and air resistance at 50 km/h speed on a straight line without going uphill or downhill in a Tesla model S, our simulation car) and 0.0 for brake position. It is not possible to gain a lower loss by changing huge amounts on the parameters, so that stays the end knowledge. We can achieve this with any network and a too high learn rate.

4.1.2 Optimizer

I looked up a few different optimizers, and read articles about them, the Adam seems to be the most effective in most cases. It has been proposed in [Adam], this is what PyTorch references in its help for Adam, this picture (Figure 4.1) is from it. The obvious parameters of Adam are the

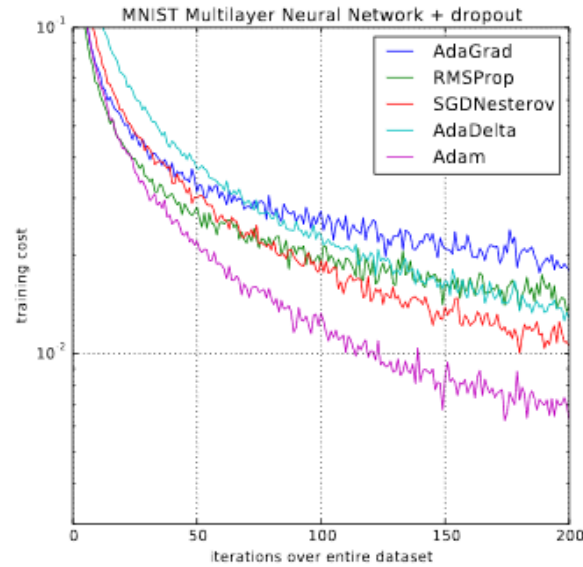


Figure 4.1: Optimizer comparison

1. parameters set, which is the set of neural network parameters we want Adam to work on. In most cases, this is all the parameters of the net;
2. learning rate (default is 0.001)

The less obvious, which you might not even use are the

1. betas, which are coefficients used for computing running averages of gradient and its square (default is (0.9, 0.999))
2. epsilon, which is the term added to the denominator to improve numerical stability (default is $1e-8$), i.e. to avoid division by 0.
3. weight decay, which is L2 penalty in this implementation (default: 0)
4. amsgrad, (boolean, optional) to determine whether to use the AMSGrad variant of this algorithm from the paper [AMSGrad]

For this task, the setup of the optimizer was generalizable along different net structures:

1. the optimizer should work on all the parameters;
2. the learning rate around 0.01 - 0.0005, depending of the size of the network;
3. the default betas are perfect;
4. there is no need to set a special epsilon;
5. weight decay around 0.0001
6. amsgrad=False;

4.1.3 Learning rate scheduling

The learning rate scheduling were less generalizable, the principle I followed in changing how it operates: gamma (the number - between 0 and 1 - we multiply the learning rate with it time to time) close to 1. There is a convention of multiplying by 0.1, but that was always a step too large for me and it just made the loss curve entirely flat, meaning the rate was too small to get out of the local minimum.

We do this multiplication along the training based on the step size. This gives the frequency of the scheduling. The ideal is to do it when the learning curve flattens. In our case, the first part was always steep, and in most cases it lasted around 300 epochs. This is where the learning rate needed to be changed first. However, step size set to 300 did not work in many cases, because the curve changed dynamics after it. My solution to this was to measure the change in test loss while the training is going and when there was a 30 epoch long period, when the losses were within 0.005, that is when I used the gamma.

4.2 Inference

In the following section, I will show the metrics that helped me deduce which network is the best from a given set.

4.2.1 Loss

The most obvious indicator of the network's knowledge is the loss. I used the mean squared error as loss, meaning for every batch of data, the net outputs 3 values - wheel angle, gas, brake - per data point and we know the label for that data point. For that data point, the loss is:

$$l(x, y) = L = \{l_1, \dots, l_N\}^T, l_n = (x_n - y_n)^2$$

If there are multiple data points in a batch, the losses of those are averaged. This is a good representation by 1 single value. The first adjustment on it needed is to break it into 2: training and testing loss. This is important, as already explained in the theoretical background section, but it is not enough.

4.2.2 Breaking loss down

Testing and validation loss can be low without the actual knowledge of the net being great for many reasons. These cases can be detected if we further break our indicator.

Loss on separate data types and separate moments

One example came, when I had a well performing network, which could steer a car successfully, but did not output pedal positions yet. (Much like in [Velocity predicting by NN]. Their

control was also partially via neural network, they predicted the vehicle speed by it; or in [Engine modelling by NN], they modelled a DC motor by a neural network.) I made the 1st version of the accelerating and decelerating network with 3 outputs and the loss (which is an average, so it should not go higher just because there are more output values) stayed low on the 1st training, however the car went off the route in the simulation environment. I created an inference file, where the trained network gives outputs for every data point and the label-output pairs can be inspected separately.

When I examined the numbers, I saw what I semi-expected before training. The wheel angles are alright, but the pedal positions were all wrong - around 0.5, indifferent to the input. This was possible, I knew that just because the input has the information to steer and I found a method to train the network to transform the information to steering this is not necessarily going to work for pedals as well, but I did not understand how this did not show in the loss. I created an other function, that computes the loss for each output datatype and it also sets them in order of the original simulation, so originally neighbouring moments are next to each other. This shows how the ultimate loss is distributed on the output data types. In the problematic case, the ultimate loss was equal to wheel angle loss, while pedal losses were practically 0. While checking and making sure I calculated their loss well, I realised the problem: in the preprocessed data, the pedal positions were in a significantly lower scale, than the wheel angles. This meant that low loss was achievable without having to get close to pedal positions, because $(0.5 - \text{labelPos})^2$ was really low for every label position compared to the wheel angle loss. From here I knew I had to scale up these data types. After scaling trainings resulted in plots like this: Figure 4.2, Figure 4.3 and Figure 4.4. Here, the pedal positions give a higher part of the loss, so there is potential that the network optimises on them. Scaling the input solved this problem and showed another one.

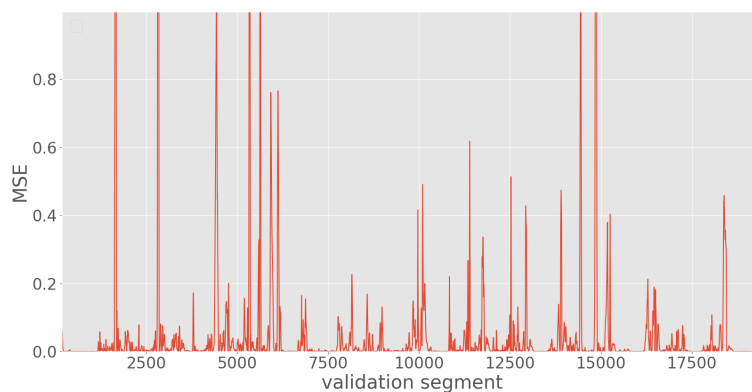


Figure 4.2: Loss on gas in each validation moment

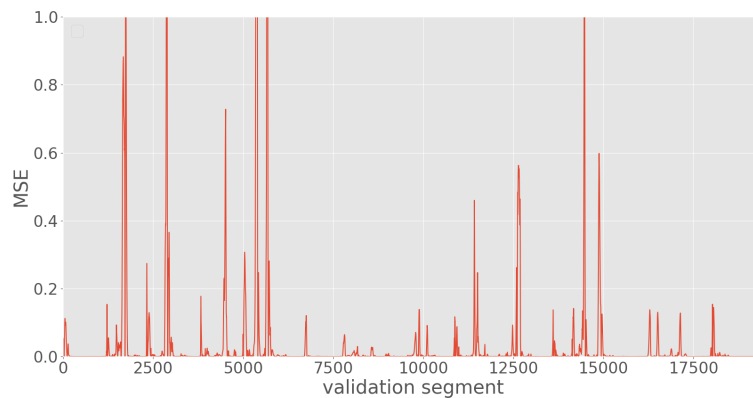


Figure 4.3: Loss on brake in each validation moment

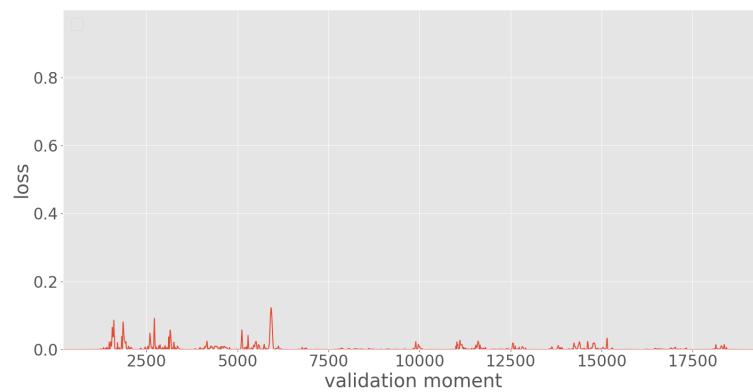


Figure 4.4: Loss on wheel angle in each validation moment

Inspecting the validation set

It is also useful to break loss down to test data points. Two cases can produce the same loss, but might have very different causes: a) The loss is evenly spread across the test cases. This shows the knowledge of the network well. b) The loss is generally low, but for some data points, it is a lot higher. If the latter option occurs, the net performs well, but not in special cases. Looking at the standard deviation of the loss is not enough, because on a validation data set e.g. an unknown route section, if the deviation is high, the first question is at which part of the section are the higher losses at.

One example for this was when the wheel angle signals were great again, the gas loss was also relatively low (even tho it was calculated on higher numbers, because it was scaled up), but not the brake loss. I tried to look at individual moments and the outputs of the network on them, but could not find situations where huge brake loss would be generated and the vehicle stayed on the road in the simulation. I started inspecting the loss of validation set along time compared to the label wheel angle to know where corners were. This instantly showed the problem. In straights, corners, corner exits and late corner entries, the braking position was alright, but at

the beginning of the brakezone, the network's output was 0. On these few moments, the break loss was huge. I realised this was because for steering, road sensors with preview distance up to 20 meters were more than enough, but for some corners, the breaking starts earlier than 20 meters before cornering. Of course the network had no chance of braking in time, there was no input suggesting it should. Loss along time: Figure 4.5 As we can see here, there is no significant rise in loss on corner entry (however exits could be better). After I increased the number of sensors on the car, some looking ahead by even 35 meters, this problem was solved, but a different one approached.

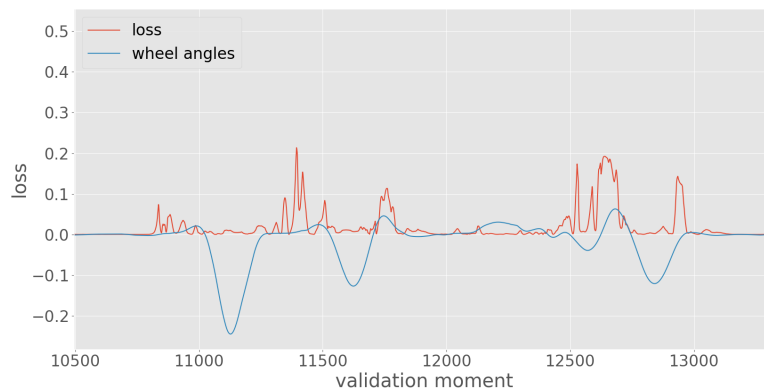


Figure 4.5: Loss on validation datapoints

4.2.3 Individual validation cases in original order

The validation losses were all low, so I tested how it behaves in the simulation. It braked in time, accelerated when possible and steered right on the path everywhere, except in one corner, where it applied the brakes until it stopped. I realised that if I resort the shuffled validation dataset, I could see sections of it together and that might help understanding this. The reason that corner was such a special case is that there was another corner within 30 meters. The only few inputs, where 2 sets of sensors showed high deviation angles: some with close preview distance (the first corner) and some with the highest preview distances (the second corner). Also, the 2nd set's output were huge deviation angles, because these were 2 right hand turns, so the angle error was around 180° . The network probably had 2 sets of neurons for these sets of sensors, mostly responsible for braking and all of them were activated, that is why such a high brake pedal position was applied. This 1 case did not affect the loss over the whole batch, so this situation was not optimised. To counter this edge case, I created more situations like this by adding 3 more sensors, with preview distance 40m, 45m and 50m. With this, the same logic would have produced much higher loss, so the network was forced to optimise on the 2 close corner situations (plural now, because there were multiple parts of the track, where there were 2 corners within 50 meters).

Figure 4.6, Figure 4.7 and Figure 4.8 are a 3 figures with the validation part of the track and the signals of the original simulation i.e. label and the output of the network - prediction - scaled back to match the original.

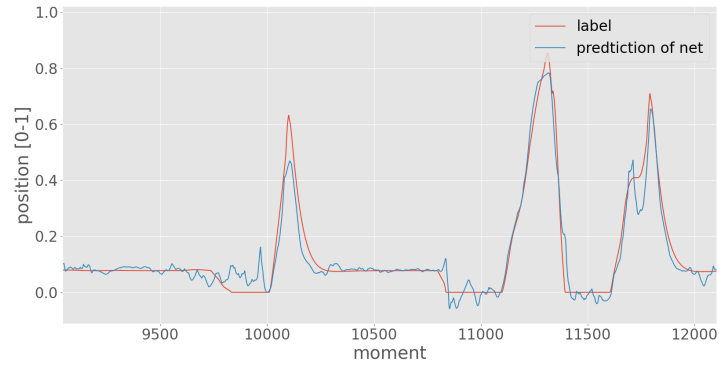


Figure 4.6: Gas signal comparison in validation dataset

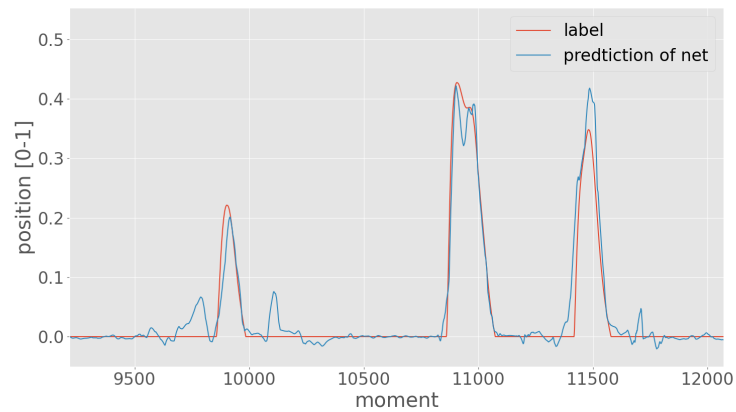


Figure 4.7: Brake signal comparison in validation dataset

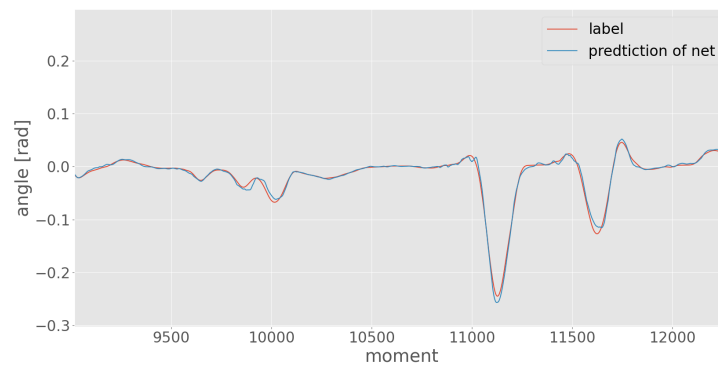


Figure 4.8: Wheel angle signal comparison in validation dataset

Chapter 5

Validating the neural network

In this chapter, I write about the way the neural network can be used in a vehicle simulator and the possible situations and modes to use the it in.

5.1 Transferring a neural network from PyTorch to CarMaker

When I had a trained neural network, which performed well under every test I could create in Python, the next step was to use its knowledge in the simulated world.

The way to do it is to export the model to an onnx file. This is easily done in any PyTorch, using `torch.onnx.export`. The onnx file can be imported in several different platforms including MATLAB 2020b and any newer version by `net = importONNXFunction(...)` and it contains the structure of the network as well as the parameters. The problem is that CarMaker, our target environment supports MATLAB 2020a and some older versions, but nothing newer, so a simple line of importing and another to use it does not work. The way around this is to load the network in a new version of MATLAB, then get the parameters from the returned value by `net.Learnables.1stlayerWeight` and save them in a `.mat` file. The reason for saving them from MATLAB and not from Python is convenience for the next step: it is possible to create a neural network of chosen structure in an older version of MATLAB which is compatible with CarMaker. In the initializer function the parameters can be initialized by some random number generalizers and custom functions as well. In a custom function of a layer, I just need to load the `.mat` file, chose the right set of weights and biases to the given layer and the network will have the trained parameters.

5.2 Using the network in CarMaker

From this point it is rather straightforward, because if there is a `net` variable in the workspace, it is possible to create a function file, which gets an input and uses the `net` to determine an output. This file can be used in Simulink via an Interpreted MATLAB function block. CarMaker gives

access to the vehicle control through Simulink, so from here the only thing to do is to find the steering, gas and brake signals and swap them to the outputs of the block.

If the target is to use network to correct mistakes of a human driver, one more step is needed. That is to decide when and how to interact. For the steering input, I decided to make the system a bit more complex than just taking the control away from the human in a situation, in which their behaviour is considered incorrect. This would only be manageable in a steer by wire system anyway. Instead, I used the force feedback of the steering wheel. The difference between the steering angle computed by the net and the angle given by the human is given to a PI controller as the error in every time step. The output of controller is the force, which will be applied to the steering wheel in the hand of the driver. The parameters are set, so the driver can swerve around the desired path with ease, but as the difference gets higher, the force will try to turn the wheel in the right position. If the pilot lets go of the wheel completely, the system manages to follow the path without oscillation.

For the pedals, no such possibility is available, so the simpler option was implemented. If the differences pass a threshold, the pedal positions are given to the vehicle model from the network. This is true for the use case where the accelerator signal is greater or smaller than the computed; and when the brake signal is smaller. The 4th- when the brake is greater - is treated differently in the sense that if the chauffeur is allowed to brake harder than the network would account for the chance they would want to go slower or stop even though the road would allow to go (for example an animal stepping in front of the vehicle).

5.3 Simulations

In the section below, here are some shorter and longer situations to demonstrate how the network performs.

5.3.1 Network and human-in-the-loop

Below are 2 situations, where the vehicle is equipped with the neural network to provide additional safety for the cases, when the driver is not avoiding a crash.

Dangerous actions

In the first example we see an easy case. Figure 5.1 The red line is the steering wheel angle which controls the front wheels. This is steered by the human and has force feedback from the neural network. The road is completely straight. The driver has control, holds on the steering wheel. Then falls asleep and steers to the right at the 680th moment of the simulation, making the lateral error (in meters on the figure) move away from 0. This results in the network outputting a high steering angle and the PI controller outputting a high force, since the difference is huge between the 2 angles. Not much after (where there red line stops growing in the negative

direction) this force takes the steering wheel out of the sleeping human's hand and steers the car back on the middle of the road.

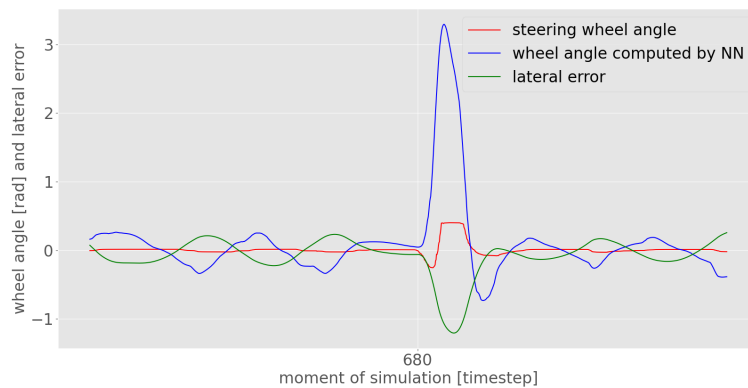


Figure 5.1: Human veers to the right, NN collects before crash

In the second example Figure 5.2 the path ahead consists of 2 corners. The driver does not see they are there, so they do not brake, they stay at 0.06 gas pedal position and 0.0 brake pedal position. It is possible to take those corners with 50 km/h speed, however the supervising network was trained on data made by this particular driver, who is a lot more cautious so the supervising system is able to detect the problem and slow down the vehicle carefully, in time. The green line shows the switch in control. When the value of it is 1, there could even be a

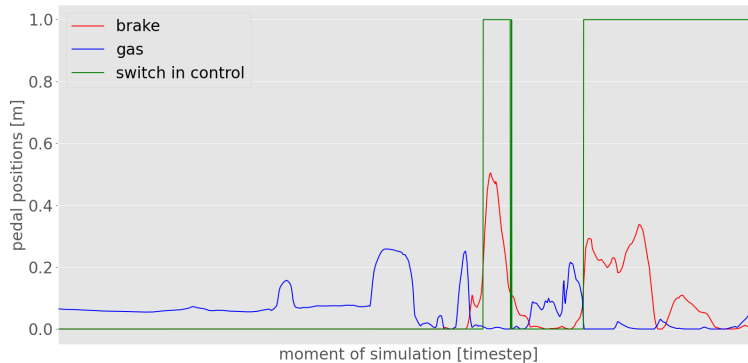


Figure 5.2: Human approaches corner without slowing down

pedal failure and the gas could be pushed all the way down to the bottom, the car would not accelerate, the neural network has control.

In the third example, the driver is not able to steer the car in the corners, so they let go of the steering wheel. The force feedback turns the wheel alone and even though the steering wheel has a deadzone, the vehicle takes the corners successfully. Figure 5.3

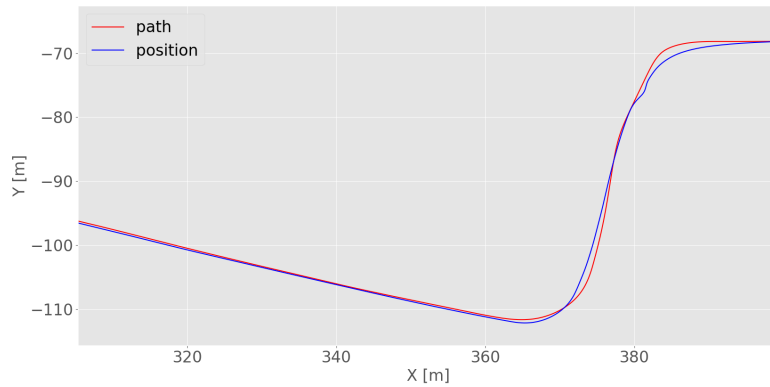


Figure 5.3: Position compared to path in corners taken by NN

5.3.2 Network used to make fully self-driving car

Correcting mistakes and avoiding dangerous situations is one thing, but as stated in the motivation, the network should be able to control the vehicle on its own. The IPGDriver (maximal longitudinal acceleration set to $3m/s^2$ accelerating, $-3m/s^2$ decelerating; maximal lateral acceleration set to $3m/s^2$ and cornercut is 0) drives around the validation set with average deviation distance 0.08 meters. The neural network produces 0.09 meters on the same 5 km long section.

This difference is small enough that it is really hard to tell which driving mechanism controls the car if we see them in a simulation, so I am satisfied by the result.

5.3.3 Network supervising different driving style

The one thing left to test is the hardest problem. Supervising and correcting a driver even if they do not make a massive mistake, but act differently to when they are completely capable and control the vehicle exactly as they want to. This requires the net to be able to identify different driving styles, not just dangerous movements. If this task is managed, the net can be used to detect even possible danger, long before sudden action is needed and actual possibility of damage is present.

Cornercut

Firstly, I wanted to test the cornering style. For reference, I ran a simulation with an IPGDriver alone, cornercut set to 0. This resulted in a position sequence. This sequence's average lateral error was 0.10 meters (of course not 0, we can see in the data inspection chapter that even cc set to 0 driving is not exactly on the path). Then another simulation, where cornercut is 1. The acceleration values were the same. The average lateral error was 0.58 meters in this case.

Then, the corner cutting driver was supervised by the neural network which learned on the same corner cutting rate. This simulation resulted in 0.58 meters lateral error, the net never really interfered, just like we would wish.

Finally the same driver was supervised by the network taught on a cornercut equal to 0 data. This resulted in a lateral error 0.35 meters. Of course, this could not be exactly 0.10 meters, because that would mean the system gives absolutely 0 freedom to the driver, it would not even let us swerve around a pothole. I ran the simulation with the PI controller having exactly the same parameters to when I used it to correct human mistakes, but let the human use the wheel correctly. This setup feels normal when moving the vehicle around the path, but applies heavier forces when we try to run off the road and restrains the cornercut equal to 1 style by a large margin successfully.

Accelerations

To test this, I changed the settings of CarMaker so that the steering is done by given force, not by angle (which was read and used from either the joystick, or the network, when it was alone).

Then, I could set up different styled IPGDriver-s and supervised them with different styled neural networks.

First, an IPGDriver (maximal longitudinal acceleration set to $6m/s^2$ accelerating, $-6m/s^2$ decelerating; maximal lateral acceleration set to $6m/s^2$ and cornercut is 0) was created and for supervisor, a network taught on a simulation made by a more restrained IPGDriver (maximal longitudinal acceleration set to $3m/s^2$ accelerating, $-3m/s^2$ decelerating; maximal lateral acceleration set to $3m/s^2$ and cornercut is 0) was used. Fed their wheel angle output in the PI controller and gave the output of it to the vehicle model to steer according to it.

In the second test, the same IPGDriver drove, but a different neural network supervised it, which was taught on the aggressive IPGDriver's simulation. We would like this network not to interfere too much, since nothing extraordinary happens in the driving style compared to when it was learning.

To verify this, I plotted a simulation where there was no supervision, only the 6, 6, 6 acceleration, 0 cornercut IPGDriver.

On Figure 5.4 we can see that there is no difference in the steering, each of them follow the cornercut set to 0 determined line. This is expected, the wheel angle usually produces the smallest loss, the parameters and scaling I found allow the network to predict it well, if the velocity is around the right value, there can not be a problem in steering.

On Figure 5.5 we can see that the acceleration is exactly the same in the non-supervised simulation to the one where we expected the network to supervise, but not interact. This is great. The blue dotted line however shows much more cautious breaking and accelerating, but in rather straight sections, it matches the other two cases. This is again exactly what we tried

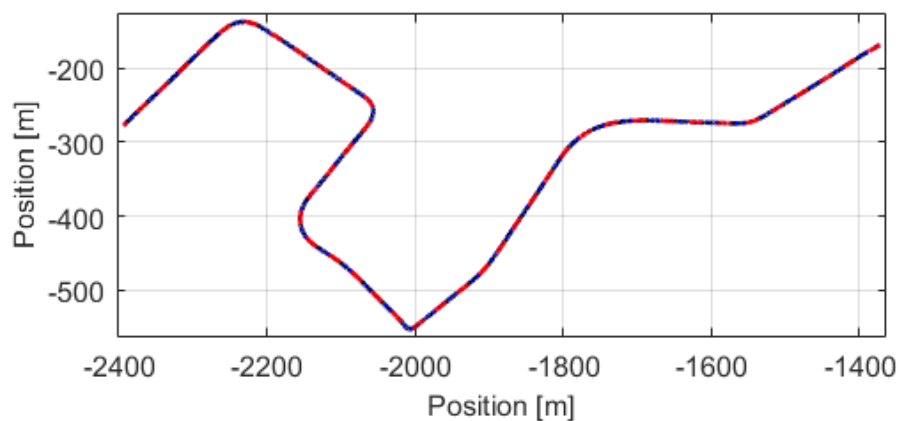


Figure 5.4: Vehicle positions

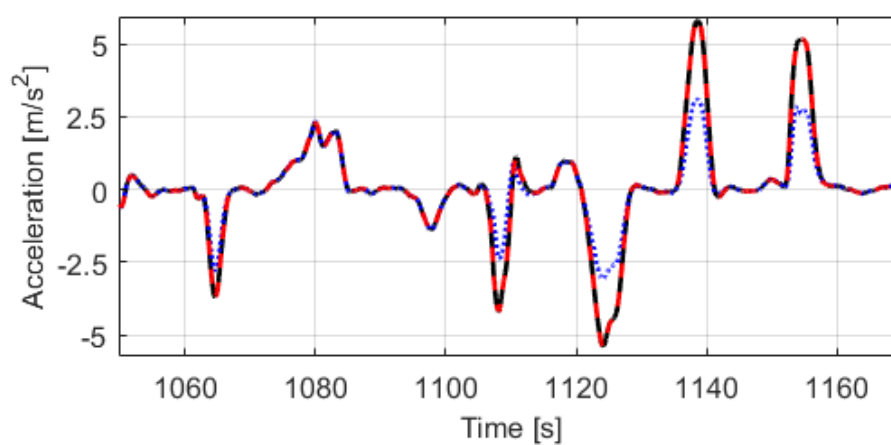


Figure 5.5: Acceleration differences between the 3 runs

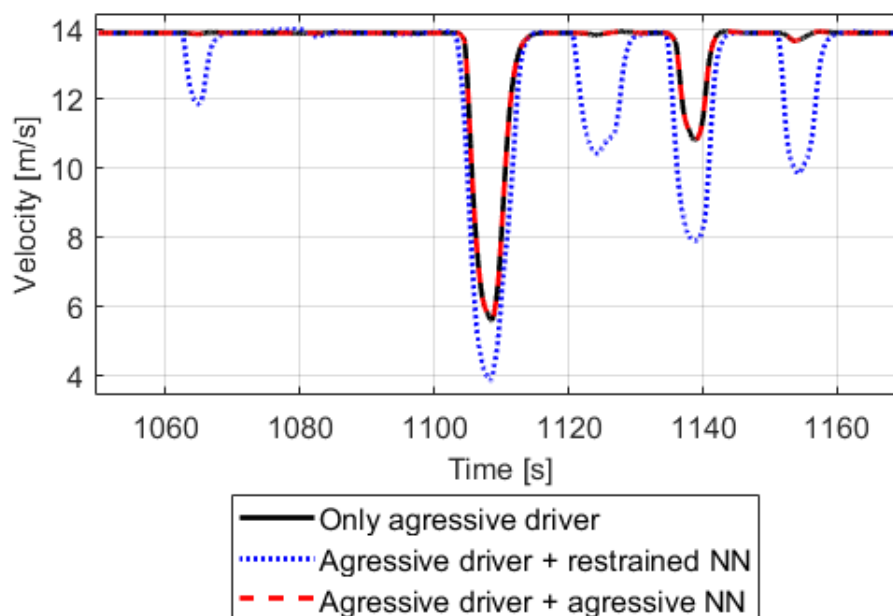


Figure 5.6: Resulting velocity differences between the 3 runs

to accomplish, the red line stays within $+6$ and $-6m/s^2$ and the blue one is between $+3$ and $-3m/s^2$.

On the last figure, we see that all 3 simulations keep the 50 km/h on the straights and accelerate back to it after corners, but the restrained network slows down the aggressive IPGDriver much more, to the nice and conservative style it has learnt on.

Chapter 6

Summary

6.1 Objectives

To summarize the paper, we need to look at the motivation. The aim was to create a solution which is able to correct human mistakes, even if they are not directly leading to an accident, but are just unusual. The solution moreover needed to be able to control a car completely on its own, on unknown track as well.

6.2 Actions taken

After careful inspection of the input and output possibilities, I chose the target signals, created an environment, where I could run 10s of trainings at once, saving the results and the resulting networks into easily and quickly loadable files. Found a well performing set of hyperparameters and fine tuned its training to achieve the best version of it.

I also created numerous ways of inspecting the results, so every kind of weakness is exposed.

6.3 Results

The resulting neural network is able to do the tasks listed at the beginning consistently, even on a track with differing characteristics. The loss can always be lower, this network's weakness are the pedal positions, but as the validating simulations show, the behaviour is more than acceptable.

6.4 Improvement possibilities

It would be great to go into more detail what kind of situations cause higher loss, separating situation types and inspecting them would be a good way of doing it. Maybe it would be

beneficial to filter the dataset before training, it is possible that because the easier kind of sections, the straights are outnumbering the corners, unused potential is left in the network.

6.4.1 Further development ideas

It would also be amazing to apply this solution on a road car. This might happen if SZTAKI finishes the Lexus it develops and equips with all kinds of sensors and beacons and they will have the capacity to allow me to try my results out at ZalaZone [ZalaZone], where the road allows us to create signals similar to the ideal road sensor's. The simulations ran by myself could run more than 2 times quicker than real time, even while saving data, so giving outputs in time should not be a problem.

Lastly, I would like to try, how this solution can be used to help a motor racing team:

a driver completes laps around a track with a few setups (this can of course be a simulated driver in a simulated race team as well). The network learns on this dataset, acquires the style of the pilot. This is necessary, because every driver is different and there are many occasions in the motor sport world, where the engineers say a setup is the best, but the pilot simply can not extract it's potential. Then the network could drive the same track with thousands of different setups, and tell which direction should the team move with the setup, without using rubber, fuel and risking the pilot's well being on a racetrack.

Bibliography

Books and papers

- [Adam] Adam: A Method for Stochastic Optimization, by Diederik P. Kingma and Jimmy Lei Ba, 2015.
- [Machine Learning] Machine Learning, by Sashank J. Reddi, Satyen Kale and Sanjiv Kumar
- [AMSGrad] On the Convergence of Adam and Beyond, by Tom Mitchell, McGraw Hill
- [Machine Learning Algorithms] Machine Learning Algorithms: A Review, by Ayon Dey
- [Overfit] Overfitting in adversarially robust deep learning, by Leslie Rice, Eric Wong, J. Zico Kolter
- [Deep Learning] LeCun, Yann; Bengio, Yoshua; Hinton, Geoffrey, „*Deep Learning*”, 2015.
- [Deep Learning] Stuart J. Russell, Peter Norvig, „*Artificial Intelligence: A Modern Approach, Third Edition*”, 2010.
- [Neural Network] Yang, Z.R.; Yang, Z., „*Comprehensive Biomedical Physics*”, 2014.
- [Loss] Raschka, Sebastian, „*Python machine learning : machine learning and deep learning with python, scikit-learn, and tensorflow 2.*”, 2019.
- [Engine modelling by NN] P. K. Sinha, L. M. Hadjiiski, K. Mutib, „*Neural Networks In Autonomous Vehicle Control*”
- [Velocity predicting by NN] A. Haddoun, M. E. H. Benbouzid, D. Diallo, R. Abdessemed, J. Ghouili and K. Srairi, „*Neural Networks In Autonomous Vehicle Control*”, 2008.
- [Vehicle steering by NN and MPC] G. Garimella, J. Funke, C. Wang and M. Kobilarov, „*Neural network modeling for steering control of an autonomous vehicle*”, 2017.

Internet

- [ZalaZone] ZalaZone webpage
URL: <https://zalazone.hu/> (2022. November)
- [AI] Definition of AI
URL: <https://www.oed.com/viewdictionaryentry/Entry/271625> (2022. November)
- [AI - ML - DL] AI - ML - DL from Microsoft
URL: <https://learn.microsoft.com/en-us/azure/machine-learning/concept-deep-learning-vs-machine-learning> (2022. November)
- [Loss] Loss and too high and too low learning rate
URL: <https://towardsdatascience.com/the-learning-rate-finder-6618dfcb2025> (2022. November)
- [PyTorch] PyTorch initialization
URL: <https://pytorch.org/docs/stable/nn.init.html> (2022. November)