Relevance of a neural network in place of a model predictive controller

Safety of a neural network equipped instead of a model predictive controller in the case of an autonomous car

Project Report

Damien Carreau
Fabio Lucas Sousa Moniz
Olivier Rescigno
Sylvain de Joannis de Verclos

Aalborg University
Department of Computer Science

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Aalborg University
http://www.aau.dk

Title:
Relevance of a neural network in place of a model predictive controller

Theme:
Scientific Theme

Project Period:
Fall Semester 2021

Project Group:
Group 6

Participant(s):
Damien Carreau
Fabio Lucas Sousa Moniz
Olivier Rescigno
Sylvain de Joannis de Verclos

Supervisor(s):
Max Tschaikowski

Page Numbers: 54

Date of Completion:
December 22, 2021

Abstract:

Model predictive control has become one of the well-established modern control methods, although it is easy to understand and implement, it has some disadvantages of requiring a large number of calculations for solving the optimization problem. On the other hand, artificial neural networks are currently developing at a fast pace, providing a possible alternative for model predictive control technology. In this project, the comparison of model predictive control and an artificial neural network is introduced with an example of a self-driven vehicle. Three types of artificial neural networks were trained using the model predictive control and validated. The time required for computation was measured and compared. A higher valid prediction was observed in certain cases. However, none of the artificial neural networks were able to decrease the amount of computation time.

Contents

Preface

This project is related to the purpose of the Computer-Science-IT-7 module and notably the Machine Intelligence (MI) lectures. This class also provided the basic theory about the cross-validation process, learning, back-propagation, square errors, and other main concepts of Neural Networks. A literature study considering Neural Network and Model Predictive Control was carried out, with the intention for the team to become familiar with the subject and amass a library of sources that could be beneficial in the future. The gathered reference work could then be used to extract information and knowledge. The team had to first gather knowledge about a specific topic before determining which approaches to employ. Once a potential solution was identified, the team conducted field tests using found out methodologies. Experiences had varying degrees of success. For example, testing includes the use of the Python module GEKKO to build the Model Predictive Control. But this proved it was not possible as it was, due to the lack of functionality required for the work. On the other hand, the reference work concerning the ACADO Toolkit was more suitable for this project. This project allowed the group to discover neural networks. Implement and evolve a machine learning algorithm into viable solutions. Through different aspects such as the creation of training data, the training itself, the validation, we will study and highlight the most successful attempts.

Aalborg University, December 22, 2021

Damien Carreau
<dcarre21@student.aau.dk>

Fabio Lucas Sousa Moniz
<fmoniz21@student.aau.dk>

Olivier Rescigno
<oresci21@student.auu.dk>

Sylvian de Joannis de Verclos
<sdejoa21@student.aau.dk>

Chapter 1

Introduction

Model predictive control (MPC) is a technology accepted in the industry for advanced control of many processes. MPC was originally developed to light on the unique control needs of power plants and petroleum refineries, it is often considered a required solution for many applications. MPC is a versatile tool, as it can deal with nonlinear behaviors and constraints, and the use of receding horizon allows it to take into account future outputs. MPC technology can nowadays be found in a wide variety of application areas including chemicals, automotive, aerospace, and many others.[3]

MPCs can be effective in the proper situation, but they have limitations that are not so often mentioned, for example, difficulty with the operation and high maintenance cost. Application of MPC controllers generally requires weeks or months of tests. [1] also, the main drawback of the MPC algorithm with a receding horizon is its high computational cost required to solve an optimization problem every few minutes or seconds.[2] Because MPC solves its optimization problem in an open-loop fashion, an artificial neural network (ANN) could potentially replace the controller.

Inspired by the human brain, ANNs are biologically inspired computational networks. AANs are the main component of machine learning and they are designed to spot patterns in data. This makes ANNs an optimal solution for classifying, clustering, and making predictions from data. Evaluating ANNs are usually more computationally efficient than computing MPCs [21], [8]. If the training of the network sufficiently traverses the state-space, then creating an ANN with a reasonable approximation of the MPC controller behavior is possible, and finally, deploying the network for the control application.

The goal of the study is to determine if an ANN can replace an MPC and still be safe and reliable. The project will take as an example the movement of a car in a two-dimensional space, where the car follows a track and can move by changing its acceleration and steering wheel orientation. Predictive techniques are commonly used to handle this type of problem but in some instances, we can also utilize machine learning solutions. The study will answer whether a trained neural network controller could replace predictive techniques.

We implement different ANNs and collate the possible solutions. To summarize, the major purpose of the research is to handle neural networks and compare their efficiency with an engineering tool, the MPC.

The method of investigation begins with a search for scientific sources. The area of investigation focuses on both topics, MPC controllers and ANN. We respond to the problem with an MPC. Then Neural Networks are trained using sets of data generated by the MPC computation. We explore three types of neural networks, one deep neural network and two recurrent neural networks (Long Short Memory and Gated Recurrent Unit). Firstly, the purpose is to find how the car would be represented, which constraints apply to it, and how the car's trajectory can be described and generated. This is why the problem should be expressed as a mathematical model, which would make it readable and analyzable. One concern is how the data-set is developed and whether or not the data quality may be improved to provide better outcomes. Another concern is in certain circumstances when time is critical. For instance, if an automobile is driving on a fast-speed road, we expect the system to be reactive due to safety concerns. The controller constantly needs to modify the velocity of the car, and turn the steering wheel. It is possible that computing (to obtain the result for both of these values) takes a considerable amount of time. This is the reason why a neural network could be utilized as a novel technology.

One can infer that an ANN is better in terms of time computation than the MPC technology because the MPC deals with nonlinear behavior and constraints. The fact is that the neural network is previously trained, and computation of output should in theory be fast. Another aspect is that in terms of safety, we cannot afford to make errors when controlling the car. If we know that MPC is a pretty sure technology, more questions are surrounding the capacity of the neural networks to be safe. In this project, we investigate and compare these aspects.

Chapter 2

Model Predictive Controller

2.1 Basics of a Model Predictive Controller

A Model Predictive Controller also called MPC is a feedback control algorithm used to solve optimization problems and make predictions. The MPC tries to solve an optimal control problem over a finite horizon. The horizon represents the number of future time steps. So at each time step, the algorithm solves an optimization problem and must find an optimal solution [18].

The MPC is based on a model, meaning that it is a mathematical representation of a real-world problem. Therefore, the model needs to reflect a real problem as accurately as possible to be realistic and reliable. The model is composed of state variables, inputs, outputs, linear time-invariant system, and constraints [17].

State variables give the precise states at each time step. Inputs are control variables that can be manipulated by the MPC and have impacts on the state variables. Outputs are just the state variables for the next step. A linear time-invariant system is a set of equations that represent the relations between our states and inputs. Solving this system allows to determine the changes of the state variables, so it is finding the outputs for any time step.

The optimization phase minimizes the errors between the reference values and the predicted values with a cost function. At the current time step, the MPC controller solves the optimization problem over the prediction horizon while satisfying the constraints. The predicted path with the smallest cost gives the optimal solution [19].

2.2 The Model

This model is based on an example using ACADO Toolkit [14], in this example, the car tries to follow a reference trajectory as accurately as possible.

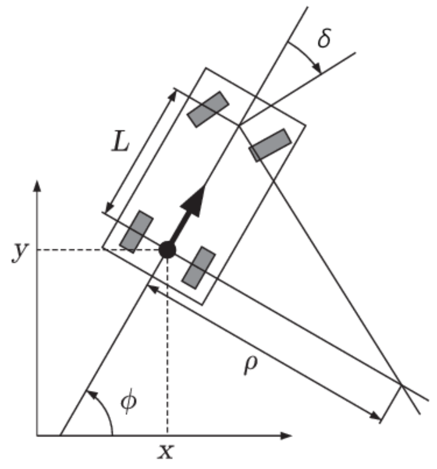


Figure 2.1: Scheme of the model taken and modified from [14]

State variable	Description
x	Position of the vehicle on the $\vec{x}$ axis
y	Position of the vehicle on the $\vec{y}$ axis
v	Velocity of the vehicle
ϕ	Car orientation
δ	Steering wheel angle of the vehicle

Table 2.1: Description of the states variables of the model

In our model the state vector is composed of five variables $S = \{x, y, v, \phi, \delta\}$, where x and y are the position of the car on the $\vec{x}$ and $\vec{y}$ axes respectively, v is the velocity, ϕ the orientation of the car and δ is the steering wheel angle, respecting the trigonometric standards *(angle $(\vec{x}, \vec{OP})$ with O the origin of the graph, P the position of the car, positive along the trigonometric direction).*. Figure 2.1 draw a visual representation of the state variable. L corresponds to the vehicle wheelbase and is constant. Table 2.1 recalls the description of the state variables. The state vector S is a vector containing all state variables, and since it contains all of them, a state vector can uniquely describe a state of the world. This vector will be used as the base of all our experiments.

Based on these state variables, we can write the time-dependent differential equations capable of representing this system where $\dot{x}$ represents the time derivative of

x, (equation 2.1). These equations describe the motion of the car at each time and, change in the state variables over time. By specifying the variables a and $deltarate$, the model is described how to evolve into the next state.

$$\begin{cases} \dot{x}(t) = v * cos(\phi) \\ \dot{y}(t) = v * sin(\phi) \\ \dot{v}(t) = a \\ \dot{\phi}(t) = v * tan(\delta)/L \\ \dot{\delta}(t) = deltarate \end{cases} \tag{2.1}$$

On this model, we apply several constraints. First, with (equation 2.2) the speed of the car is limited to the interval $[minSpeed, maxSpeed]$ where $minSpeed$ is fixed at $-5m.s^{-1}$ and $maxSpeed$ is fixed at $15m.s^{-1}$. The minimum speed can be negative, which corresponds to the reverse gear of a car.

$$minSpeed \leq v \leq maxSpeed \tag{2.2}$$

Second, the steering wheel angle is limited to the interval $[minAngle, maxAngle]$ where $minAngle$ is fixed at $-\frac{\pi}{2}$ and $maxAngle$ is fixed at $\frac{\pi}{2}$. (equation 2.3).

$$minAngle \leq \delta \leq maxAngle \tag{2.3}$$

Control variable	Description
a	Acceleration of the vehicle
$deltarate$	Variation of the steering wheel angle between two time steps

Table 2.2: Description of the control variables of the model

These differential equations, depend on state variables, but also two control variables called inputs of the system. These control variables in this model are the vector $U = \{a, deltarate\}$, where a is the car acceleration and $deltarate$ is the derivative of the steering wheel angle. So it's the angle to add to δ to update the car steering wheel angle. This vector U will be used in all our experiments.

When using a control vector U and a state vector S on the differential equations, it is possible to predict the next state of the system corresponding to the next time step. The output is a vector containing the changed state variables.

2.3 Track Generation

2.3.1 Simple Track Generation

In order to simulate the movement of a car, let us place ourselves in a 2D space as if we were looking at the vehicle from above. In order to generate a path for our car, let's use the package *scipy.interpolate* [20] in Python. This package includes functions for determining a curve's representation and evaluating it at given locations. These two methods combine to create a "smooth" curve that fits a set of irregular points.
To create an array of 40 points, let's use the rule described in equation 2.4, starting from point (0,0).

$$u(x+1) \in [u(x) - 0.25 * \sigma; u(x) + 0.75 * \sigma] \tag{2.4}$$

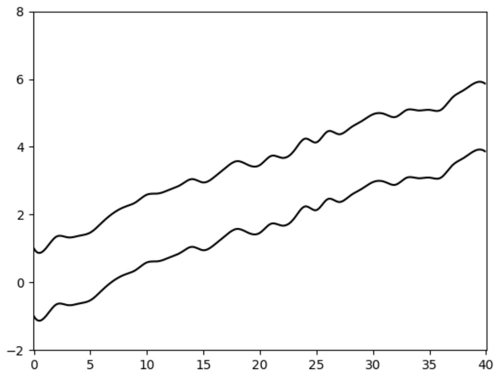


Figure 2.2: Example of a path generated by our first path generation algorithm

In this situation, the curve is created in such a way that each point does not deviate too much from its predecessor. They have a margin of freedom of σ. However, for a first simple configuration, the curve is forced to increase, hence the shift from the 0.25 interval to higher values. After submitting this table to the *scipy.interpolate* algorithm, we obtain a reference curve. In order to create a track, let us add 1 in both directions to each point to obtain a track of width 2 and two identical walls. Figure 2.2.

2.3.2 A More Realistic Track

This time, we will generate a track that looks like a realistic road with curves. To accomplish this, we go from the equation 2.4 to the equation 2.5. Where $r1$ and $r2$ are random values with $r1 \in [1,3]$ and $r2 \in [1,4]$.

$$u(x+1) = \begin{cases} u(x)/2 + r1, & \text{if } x \leq 10, \\ u(x)/2 + r2, & \text{otherwise} \end{cases} \tag{2.5}$$

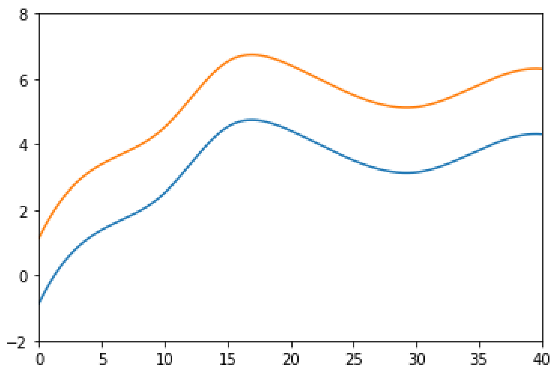


Figure 2.3: Example of a generated track with turns

In this situation, the curve is created in such a way that some irregular points are generated thanks to r1 and r2. The method of constructing the table is simply to add the last ordinate value divided by two with a random value. Also for the first ten abscissa values, the random value is generated in a smaller interval to avoid excessive growth and to tend to be more realistic. This method, therefore, allows a table of different values to be obtained.
After submitting this table to the algorithm *scipy.interpolate*, we obtain a "smooth" reference curve. This reference curve attempts to simulate the path of a road as closely as possible. So to create a track, let's add 1 in both directions at each point to obtain a track of width 2 and two identical walls as in figure 2.3.

2.4 Code of the Model Predictive Controller

2.4.1 ACADO Toolkit

ACADO Toolkit [7] is open-source software for automatic control and dynamic optimization. This software is used to solve control problems and optimization problems. It provides a large variety of algorithms as nonlinear optimization algorithms. ACADO Toolkit is implemented as self-contained C++. Generally, the toolkit is used to programming in C++, however, it can provide extensions like a python extension that interprets python objects as C++ code. It can be used to implement a MPC after defining the model in a language that is specific to ACADO. That is, defining state variables, parameters, constraints, and differential equations. See section 2.2. In our case, we will use ACADO packaging in Python and will allow us to determine in an optimized way the acceleration and deltarate to apply to the car to best follow a reference trajectory.

2.4.2 Python Package Generation

ACADO is flexible and can be modified to adapt to many model predictive control problems. The ACADO Toolkit provides an object OCP (optimal control problem) which permits to generate MPCs and to specify an optimization algorithm. In this section, we describe how is the OCP object configured.

Firstly, we describe the state variables and control variables.

```
DifferentialState x;
DifferentialState y;
DifferentialState v;
DifferentialState phi;
DifferentialState delta;

Control a;
Control deltarate;
```

Secondly, the differential equation of the model is defined.

```
DifferentialEquation f;
// model equations
f << dot(x) == v*cos(phi);
f << dot(y) == v*sin(phi);
f << dot(v) == a;
f << dot(phi) == v*tan(delta)/L;
f << dot(delta) == deltarate;
```

The OCP object is created with the start time 0 and the end time $N * DT$. N is the horizon and DT is the time interval between two steps.

```
OCP ocp(0, N * DT, N);
```

Then, the differential equation and constraints of the system are registered in the OCP. The absolute value of acceleration can oscillate between 0 and 1. Similarly, absolute values of *delta* and *deltarate* oscillate between 0 and respectively 90 and 45 degrees.

```
ocp.subjectTo( f );
ocp.subjectTo( -1.0 <= a <= 1.0 );
ocp.subjectTo( -M_PI/2 <= delta <= M_PI/2 );
ocp.subjectTo( -M_PI/4 <= deltarate <= M_PI/4 );
```

Here the state variable δ is not very important. As a reminder, the δ variable describes the orientation of the car's wheels. Indeed, between two steps, δ will modify the trajectory of the car but through ϕ. Moreover, it is directly controlled by the control variable *deltarate* describing its derivative. Thus, as our differential equations describe, it is sufficient to optimize the ϕ variable as a function of *deltarate* to implicitly define the state of δ. Thus, in the definition of our optimization functions, we do not declare δ as a variable to be optimized.

The least-square method is used to minimize acceleration and *deltarate* values. To implement this, we create two diagonal matrices. The matrix WN, of size 4x4, corresponding to the number of state variables, describes the new state of the car after applying the optimized MPC outputs during the next N steps. The matrix W, of size 6x6, corresponding to the number of state variables plus the number of control variables, will represent the costs generated by the solution. The MPC is then generated and the Gauss-Newton algorithm is used for minimization.

```
Function rf  << x << y << v << phi  << a << deltarate;
Function rfN << x << y << v << phi;
BMatrix W = eye<bool>(rf.getDim());
BMatrix WN = eye<bool>(rfN.getDim());
ocp.minimizeLSQ(W, rf);
ocp.minimizeLSQEndTerm(WN, rfN);
OCPexport mpc( ocp );
mpc.set(HESSIAN_APPROXIMATION, GAUSS_NEWTON);
```

After exporting this MPC and declaring it as a Python package named *acado*. We can use it via the function *acado.mpc(...)* by parsing variables like the current state, the desired trajectory for the next N steps. This function returns the optimized state vector S and the manipulated vector U (which contains the optimized values for *deltarate* and a).

2.4.3 Compute the Reference Trajectory

Section 2.3, explained the creation of tracks, this returns two arrays of points containing the x and y coordinates of the path. But this is not sufficient for the MPC to interpret the trajectory to be followed. It is, therefore, necessary to reformat this data.
Firstly, the MPC describes steps as a function of time. Our interval between two consecutive steps is set to $DT = 0.1$ seconds. Thus, we need to split our path into a succession of points separated by our DT time step. 0.1 seconds allows us to have a fairly precise cutting, and therefore to represent reality faithfully without overloading the MPC and thus increasing its execution time.
Moreover, for each point, we must specify to the MPC what is the orientation of the car we want and also its speed (Figure 2.4). To do this, at each point, we indicate the orientation of the car as the angle of the tangent to the track at that point. For the speed, we describe a desired speed equal to $target\_speed = 5km/h$. This speed is low but avoids too much deviation from the track. If the track is not too curved at this point, then we set the optimal velocity at this point as *target_speed*, otherwise, we decrease this optimal value so the vehicle doesn't move too far from the

track. To know if the track is too curved, we look at the difference in orientation of the vehicle between two successive states. If this is greater than the maximum variation between two states (set at $\frac{\pi}{4}$ in section 2.2) then the car must slow down or it will leave the desired path. We then save these values in four arrays, one for each reference value, $cx, cy, c\phi, cv$ for the state variables x, y, ϕ, v respectively. In other words, $c\phi$ represents the reference curvature of the track at each point, cv represents the theoretic speed at each point of the track, and so on. These four variables correspond to what we call the reference trajectory of the track.

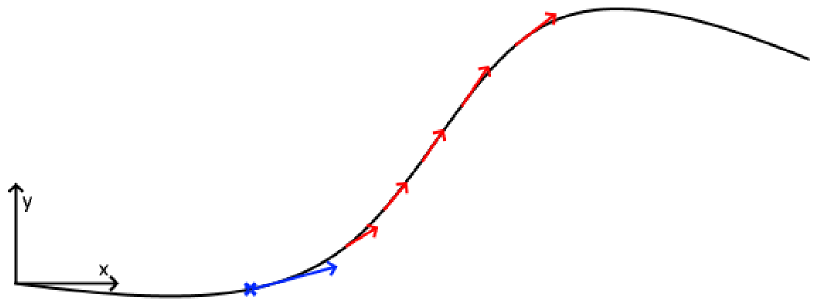


Figure 2.4: Representation of the reference trajectory. The black curve represents the result of our trajectory generator. The reference trajectory is a list of values storing the position (x and y), orientation (ϕ) and velocity (v) that the vehicle is supposed to have at each point. The interval between two points is $DT = 0.1s$. If the blue vector represents the current state, then the blue vector plus the five red vectors represent the variable $stateRef$ for a horizon $N = 5$.

2.4.4 Simulation of the Model Predictive Controller

On our curves, we use the pseudo-code 1 to obtain the MPC predictions. The predicted path is stored in the variable $trajectory$, the commands to be applied are stored in a vector $command$.

At first, we generate a track *(line 9)* then we generate the reference value for each state variable and store them in $cx, cy, c\phi$ and cv *(line 10)*.
On *line 11*, we create the initial state which has the value of x equal to the first value of x of the reference path. Same for y and ϕ, the first orientation of the car is the same as the first orientation of the track. However, for the velocity, we start with an initial velocity of zero.
Then we apply a *while* loop (*line 13*) to iteratively increment the state. We apply a time limit ($time < MAX\_TIME$ condition). To cover the 200 meters of the track, we limit it to 1000 seconds. And we describe the step between 2 states as a time difference of 0.1 seconds (DT variable). Furthermore, if the prediction arrives at

the end of the reference path before our time limit, we exit the loop ($S.x < cx[-1]$ condition).
For each step, (*line 14*) we create a matrix $stateRef$ which stores the current state vector and the reference values for the next N steps. In figure 2.4 you can see a representation of $stateRef$ where the blue vector is the current state and the five red vectors correspond to the next $N = 5$ reference values. Note that even if the current state is not on the reference path, the next N reference states are unchanged even if they are unreachable by the car in the short term. They represent the states the car is supposed to be in.
We then give the MPC the current state and this matrix $stateRef$ and it returns the new state vector associated with the control vector that optimizes all conditions (*line 15*). These state and control vectors are stored in the list *trajectory* and *control* to create the predicted path and controls (*lines 16* & *17*).
In figure 2.5, you can see the prediction of the MPC for a 40 meters track. It is almost identical to the reference track with a maximum deviation of about 25cm.

Algorithm 1 Pseudo code used to obtain an MPC prediction

```
1: procedure MAIN
2:     import acado
3:     $MAX\_TIME \leftarrow 1000$
4:     $N \leftarrow 5$
5:     $DT \leftarrow 0.1$
6:
7:     $time \leftarrow 0$
8:     $trajectory \leftarrow$ List of vector
9:     $commands \leftarrow$ List of vector
10:    $x, y \leftarrow$ generate the coordinates of a 200m track
11:    $cx, cy, c\phi, cv \leftarrow$ generate the reference value for each variable at each point
       of the track using $x$ and $y$
12:    $S \leftarrow$ Vector($x = cx[0]$, $y = cy[0]$, $\phi = c\phi[0]$, $v = 0.0$)
13:    while $time < MAX\_TIME$ and $S.x < cx[-1]^1$ do
14:        $stateRef \leftarrow$ reference trajectory for the N next steps using $cx, cy, c\phi$ and $cv$
15:        $S, U \leftarrow$ acado.mpc($S$, $stateRef$)
16:        $trajectory$ add $S$
17:        $commands$ add $U$
18:        $time$ += $DT$
19:    end while
20: end procedure
```

1: The index -1 represents the index of the last element of the array.

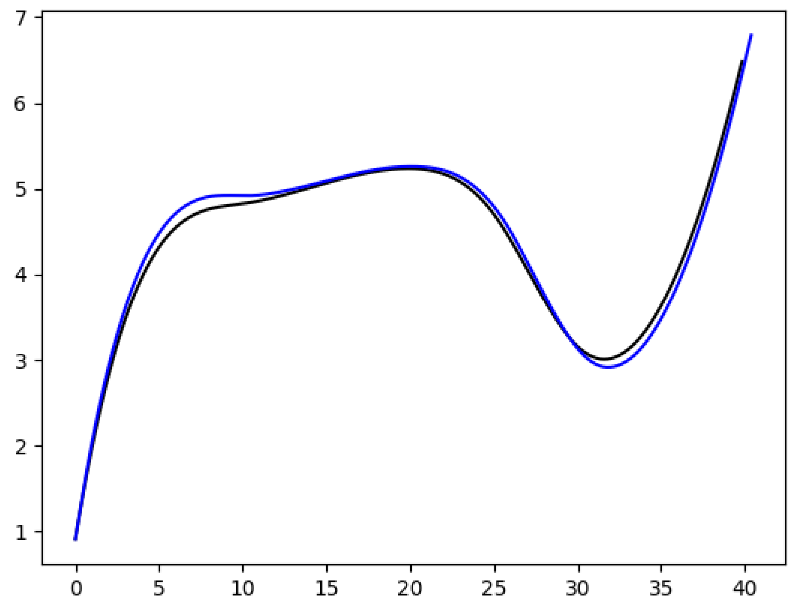


Figure 2.5: MPC prediction for a 40 meter track. The black line is the reference trajectory, the blue line is the MPC prediction.

Chapter 3

Neural Network Introduction

3.1 Structure of a Neural Network

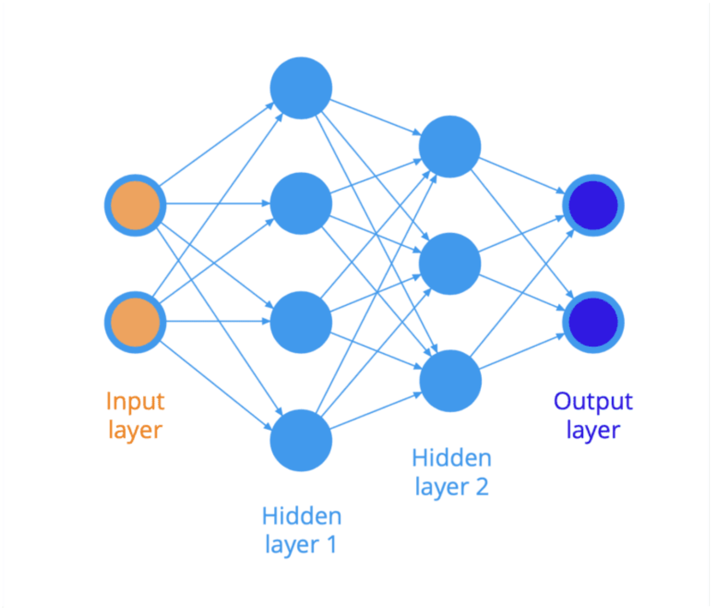


Figure 3.1: Scheme of a Neural Network [5]

An ANN is a computational model that predicts outputs based on inputs. An artificial neural network is biologically inspired by the human brain. It is a collection of nodes called neurons, neurons are linked together by connections with a weight value. These neurons are organized in layers : the input layer, the hidden layers, and the output layer. Moreover, nodes have a value, which is information or knowledge.

3.2 Transmission of Knowledge in a Neural Network

The goal of a neural network is to share information across the network, then each neuron performs operations to make the data flow through the network. The information can be carried across the connection between two nodes by the product of nodes' value and the weight of the corresponding connection. The nodes of a layer that contain knowledge send it to the nodes of the next layer, so these receiving neurons sum the input value into an output using a specific function called the activation function. The output value can serve as an input to the next layer. Equation (3.1)

$$output = a(\sum_{i=0}^{n} x_i * w_i) \tag{3.1}$$

The operations performed by each neuron can be illustrated as follows. Figure 3.2

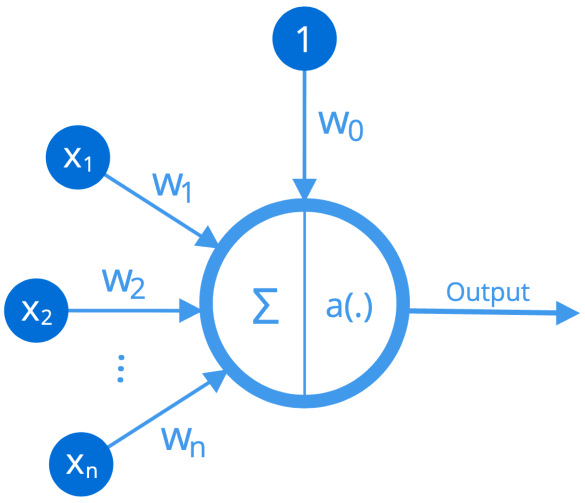


Figure 3.2: Operations performed by a neuron [5]

3.3 Forward and Backward Propagation

In a neural network, information can propagate in a forward direction or a backward direction, there exist different algorithms to realize this Forward / Backward propagation. Moreover, the goal of a neural network is to find the set of weights that fit best the patterns in the data, then a loss or cost function needs to be used because with this cost function it's possible to calculate the margin of error between expected and actual value.

$$J = \frac{1}{n}\sum_{i=1}^{n}(y - \hat{y})^2 \tag{3.2}$$

In the equation of the cost function [11], n is the number of training samples, y is the expected output, and $\hat{y}$ is the actual output. This equation simply squares the difference between every network output and expected output, and takes the average.

The aim is to minimize a cost function, thus forward and backward propagation are necessary steps. Each time that the network is traversed in the forward direction, a prediction and a cost value are generated, the backward propagation algorithm uses gradient descent to learn the weights and calculate the gradient of this error to changes in the weights. For that, we need to set a learning rate which is a small positive value that controls the magnitude of change of the parameters at each run. The learning rate impacts the training time of the neural because if it is too small, it will take a longer time to find a minimal set of weights and if it is too big, the optimal solution will never be reached[5].

Then repeating the Forward/Backward propagation through the neural network update the parameters to minimize the cost function[11].

$$w_{t+1} = w_t - \alpha\frac{\partial J}{\partial w} \tag{3.3}$$

$$b_{t+1} = b_t - \alpha\frac{\partial J}{\partial b} \tag{3.4}$$

In the equation of the updating rules for weights (3.3) and the bias (3.4), it simply finds the new weight by adding the actual weight with the corresponding gradient and for the bias, it adds the actual bias by the corresponding gradient.

3.4 The Learning Rate

The learning rate is considered a hyper-parameter for Deep Learning algorithms. A hyper-parameter can be defined that way *"**We define a hyper-parameter for a learning algorithm A as a variable to be set prior to the actual application of A to the data, one that is not directly selected by the learning algorithm itself**"*[4]. The learning rate is used to update the weights, Equation (3.3). This parameter is crucial to find the new set of weights in order to minimize the loss function, moreover, it also impacts the training time because it has effects on the speed convergence to a minimum set of weights. Thus an essential step before training a neural network is to initialize the learning rate with an appropriate value that is not too large to find a local minimum and not too small to reduce the number of updates and computations.

3.5 Types of Neural Networks

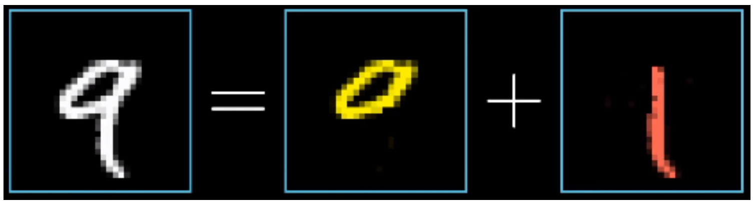
Deep neural networks
The deep neural network is the most common. Each node in one layer is connected to each node in the next. [12]

Convolutional neural networks
Convolutional neural networks allow convolution to be applied to input data or data in a hidden layer. They do not interpret the data one by one, but rather a set of inputs defined by the convolution mask. This type of network is most often used for image recognition as it allows pixels to be interpreted in groups which are easier to identify patterns. [12]

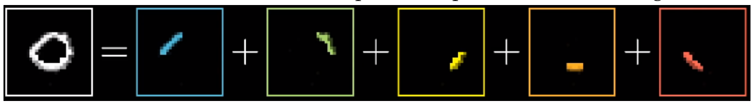
Recurrent neural networks
Recurrent neural networks allow functions to be simulated in a temporal sequence. Indeed, for a prediction, the network will add to the inputs of the previous prediction. This can be used in our case for prediction on a path. However, it will be necessary to be careful when changing the route, that the previous predictions for the previous path do not influence the new prediction. [12]

3.6 The Layers

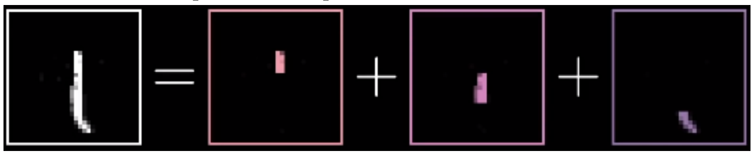
The intuition of a hidden layer is to compute an intermediate calculation. To be more concrete, let's use an example of number recognition as described in [6]. If the goal is to recognize the number '9' with a neural network with two hidden layers, we can imagine the following situation.
As shown in the figure 3.3, to represent a nine, we draw a loop at the top and a long line on the right. So, to recognize the number nine, we can imagine that the neural network will first try to recognize these shapes. And recursively, to recognize a loop or a long line, it will first try to recognize a short line, as shown in figures 3.3b and 3.3c. Thus, we can imagine that the first hidden layer will try to discern small lines. The second hidden layer will try to join them together to create larger shapes. And finally, the output layer assembles these shapes to deduce the input figure.



(a) The number nine has a loop at the top and a line on the right



(b) A loop can be interpreted as a sum of five small lines



(c) A long line can be interpreted as the sum of three short lines

Figure 3.3

In the case of image recognition, it is very easy to imagine these intermediate steps. But in our model, it is much more difficult. The MPC, starting from a reference trajectory, projects the current state onto the next N steps by applying the previously predicted outputs. Using an error function, in our case the sum of squared errors, it adapts these outputs to be as close as possible to the desired trajectory. So there is nothing recursive about it. So, applying our intuition of hidden layers, we don't need many in our model.

3.7 The Nodes

To understand the impact of neurons, let's take a simple example. Suppose we have two inputs $x1$ and $x2$ bounded on $[-1;1]$. We will use only one hidden layer with the ReLU (Rectified Linear Unit : ReLu(x) = max(0,x)) activation function and the output layer will have a single neuron equipped with the softsign function. This description is summarised in table 3.1 with setting $T = 1$ (T represent the number of nodes on the hidden layer). A representation of this network can be found in figure 3.4.

$$\begin{aligned} output &= softsign(w_3 * relu(w_1 * x_1 + w_2 * x_2 + b_1) + b_2) \\ &= \begin{cases} softsign(b_2) & \text{if } w_1 * x_1 + w_2 * x_2 + b_1 \leq 0 \\ softsign(w_3 * (w_1 * x_1 + w_2 * x_2 + b_1) + b_2) & \text{otherwise} \end{cases} \end{aligned} \tag{3.5}$$

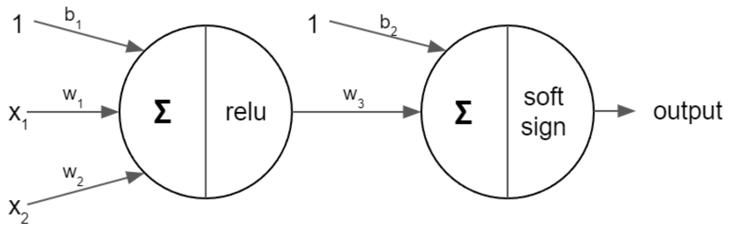


Figure 3.4: Representation of the first network use in the subsection 3.7

DNN Structure	
Type	Fully_connected
Number of layer	1
Number of nodes on the input layer	2
Number of nodes on the hidden layer	T
Number of nodes on the output layer	1
Activation function of the hidden layers	ReLU function
Activation function of the output layer	SoftSign function

Table 3.1: Configuration of the deep neural network of section 3.7. T corresponds to the number of node on the hidden layer, it will change to demonstrate the effect of adding nodes.

We can therefore write the output equation 3.5.

Here we see two cases. This is thanks to the ReLU function which distinguishes between the case where the argument is greater than zero or not. In all cases, this argument represents a plane of space (we recognise an equation of the type z = ax+by+c where a, b and c are constants). Thus, by drawing on figure 3.6a the surface associated with this network, we find two parts corresponding to two planes passed into a softsign function. It is therefore possible to describe simple functions. For example, by replacing the ReLU and softsign activation functions with the identity function, it is possible to describe a linear function since a plane equation will be obtained directly. We can now look at the consequences of adding a second neuron on the hidden layer, figure 3.5. Here is the output equation 3.6.

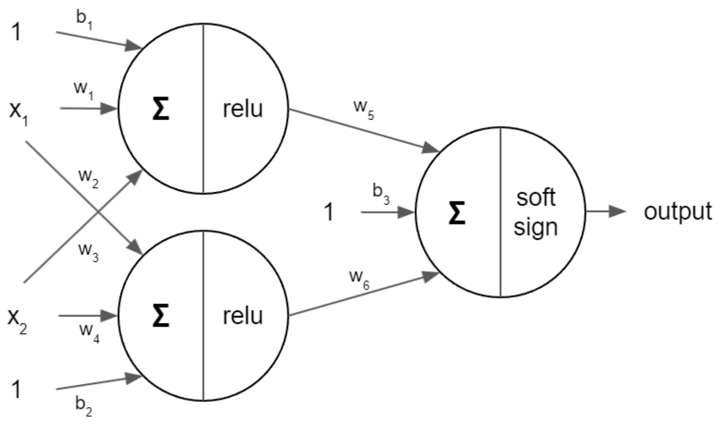


Figure 3.5: Representation of the second network use in the subsection 3.7

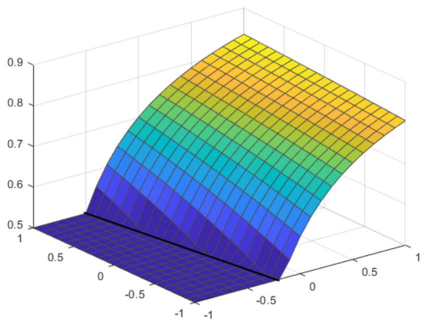
$$X_1 = w_1 * x_1 + w_3 * x_2 + b_1 \tag{3.6a}$$

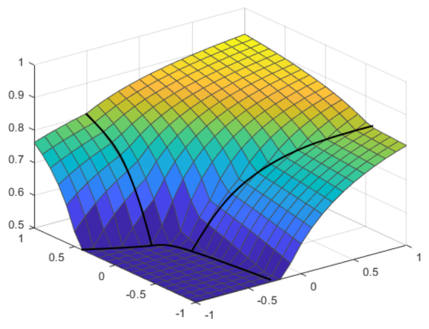
$$X_2 = w_2 * x_1 + w_4 * x_2 + b_2 \tag{3.6b}$$

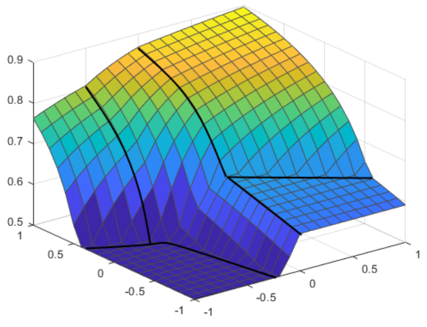
$$\begin{aligned} output &= softsign(w_5 * relu(X_1) + w_6 * relu(X_2) + b_3) \\ &= \begin{cases} softsign(b_3) & \text{if } X_1 \leq 0 \text{ and } X_2 \leq 0 \\ softsign(w_5 * X_1 + b_3) & \text{if } X_1 \geq 0 \text{ and } X_2 \leq 0 \\ softsign(w_6 * X_2 + b_3) & \text{if } X_1 \leq 0 \text{ and } X_2 \geq 0 \\ softsign(w_5 * X_1 + w_5 * X_2 + b_3) & \text{otherwise} \end{cases} \end{aligned} \tag{3.6c}$$

Now we have four different cases. Indeed, each neuron can be either greater or less than zero. We can therefore deduce that for T neurons, there will be 2^T different cases. (As long as we stay with the ReLU function). Thus, in figure 3.6b, we see four distinct planes passed through the $softsign$ output layer activation function.

Thus, if we use three neurons, figure 3.6c, it is possible to describe eight different planes through the $softsign$ function. Continuing this reasoning, the more neurons we add, the more planes we can describe through the $softsign$ function and the more we will be able to describe any function. In the figure 3.6d, you can see the output of an identical network with 500 neurons on its hidden layer.
Article [13] formally confirms this intuition. It shows that a neural network with hidden layers equipped with the ReLU activation function can describe diverse and complex functions. Moreover, it establishes that a single hidden layer can produce conclusive results after efficient training.


(a) T = 1


(b) T = 2


(c) T = 3

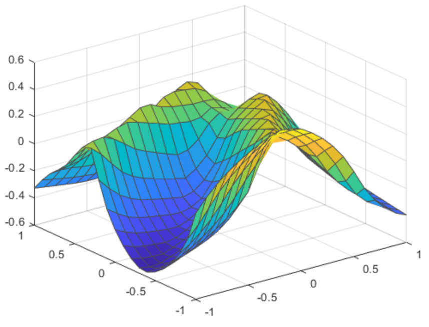

(d) T = 500

Figure 3.6: Representation of the output of the neural networks describe in the section 3.7.
T represent the number of neurons on the hidden layer. The black line highlight the different parts. The weights of the networks are chosen randomly to demonstrate the effect of adding nodes to the hidden layer.

Chapter 4

Deep Neural Network

4.1 Training Data

In order to train our neural network, and to predict its response on new trajectories, we need to define the inputs. Previously in the section 2.2, the MPC predicts the movements to be performed according to four parameters :

- $stateRef$ which is the current state plus the N next reference states one the reference trajectory. N describing the horizon of the MPC. It's a matrix of size $(N+1) \times 4$. Each column corresponds to the states variables x, y, ϕ and v. See an example of $stateRef$ on figure 2.4.
- The current state vector itself $S = \{x, y, v, \phi, \delta\}$. Array of size 5.
- The last output prediction of the MPC : $U = \{a, deltarate\}$. The MPC returns a prediction for each N step so it is represented by two arrays, one for a and one for $deltarate$ with a length of $N+1$ each. (The actual output and the N next).

These inputs are good for describing a state and its environment. For our neural networks, we will keep the first two points. We will not keep the last prediction because our neural network will not give us any hints for the next states.
The inputs to our neural networks will therefore be 29 values. By setting a horizon $N = 5$, we will have 24 values corresponding to $stateRef$ and 5 values for the current state.
To train our networks, we will use supervised learning. That is, we will give the network input and the output it is supposed to obtain, and by repeating this operation a certain number of times, the network will be able to imitate the desired behavior. Thus, as we wish to mimic the behavior of an MPC, we will use the predictions of the MPC (section 2.4.4) as the reference output.

4.2 First Experiment

To implement the neural network, we will use the *TFLearn* library for Python [16], based on *TensorFlow* [15] which is the reference library for machine learning nowadays.

Let's start with a deep neural network. Our first layer consists of 29 nodes, one for each input described in the section 4.1. Our output layer has 2 nodes, one for acceleration and one for delta rate. We can try to put 5 fully connected hidden layers, each composed of 500 nodes. To train our model, we use the data from 50 tracks of 200 meters in length, which represents a dataset of around 92.500 states (A 200m track contains in average 1.850 states : $50 \times 1.850 = 92.500$). Each of these is linked to the output given by the MPC for that state.

As it is our first try, we choose several epochs that equate to one. An epoch is the number of times the network will use the dataset to train. If there are not many examples, it may be useful to increase the number of epochs. But it can also work against you as the network may be too good with your dataset than with never seen data.

For the backpropagation algorithm, we use the *mean_square* loss. This means that the cost function will be the sum of the squares of the errors of each output. As this is a sum of all outputs, we need them all to have the same range. For example, if we have one output with a range of 0 to 1 and a second with a range of 0 to 1000, even for the same percentage error on both parameters, this will not have the same impact on the cost function. Indeed, for a percentage error of 5% on both outputs, the first will have an impact of 0.05 while the other will have an impact of 50.

In our model, the acceleration belongs to the interval $[-1; 1]$ while the delta rate is defined between $[-\pi/4; \pi/4]$. We will therefore normalize the delta rate between -1 and 1 so that the *mean_square* loss is fair for both outputs. Furthermore, as it is normalized between -1 and 1, we can use the softsign function (figure 4.1) as the activation function on the output layer. Thus, the output will always be between -1 and 1.

In table 4.1, you can find the full description of the network.

To perform a neural network prediction, we use an algorithm similar to the one described in pseudo-code 2. It is almost identical to Algorithm 1 used for the MPC predictions.
In figure 4.2, we can see the result of our neural network on a 200 meter track that

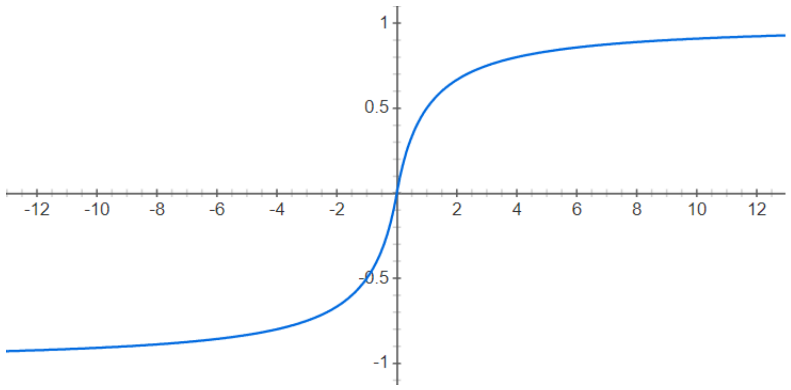
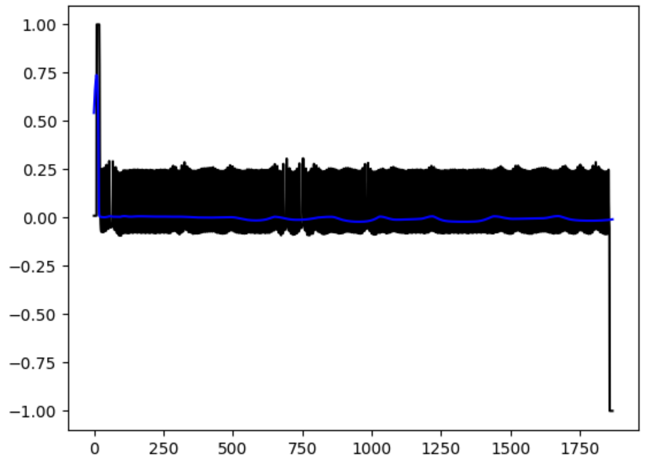


Figure 4.1: softsign(x) = $\frac{x}{|x|+1}$

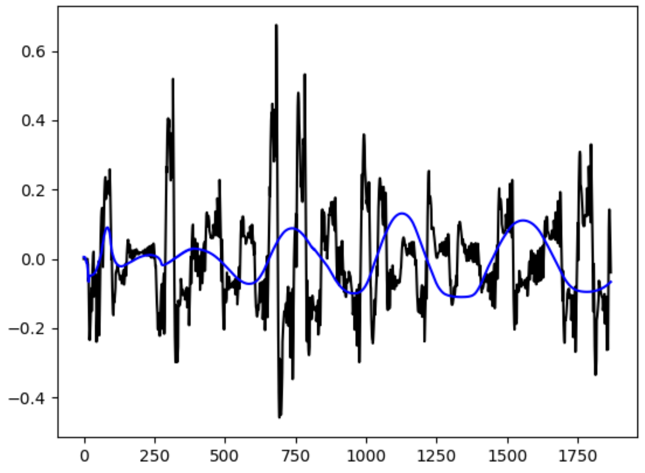
DNN Structure	
Type	Fully_connected
Number of layer	5
Number of nodes on the input layer	29
Number of nodes on the hidden layer	500
Number of nodes on the output layer	2
Activation function of the hidden layers	ReLU function
Activation function of the output layer	SoftSign function
Number of epoch	1
Loss function	Mean Square Error
Optimizer	Adam
Learning rate	0,0001

Table 4.1: Configuration of the deep neural network of section 4.2

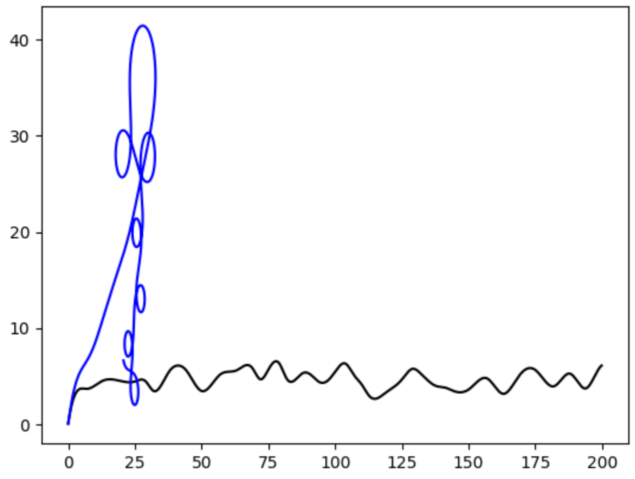
was part of its dataset. In the first figure 4.2a, we see that the output of the MPC is unstable. From one step to the next, it jumps from -0.1 to 0.25 and then back to -0.1. Our neural network is therefore not able to follow it and averages it out. To improve this reference track, we can smooth the change in acceleration by putting a condition on the derivative of the acceleration. But this may have an impact on the accuracy of the MPC. We will explore this idea further in the section 5.
In the following figure 4.2b, as for the acceleration, the neural network prediction for delta rate is a kind of average. But here it is more accurate.
Figure 4.2c, shows only the trajectory of the car if we apply the predicted outputs.



(a) Comparison of the acceleration output as a function of time (10 units on the x-axis correspond to 1 second)



(b) Comparison of the deltarate output as a function of time (10 units on the x-axis correspond to 1 second)



(c) Comparison of the trajectory path

Figure 4.2: Simulation of our first model (table 4.1) on a 200 meter track. In each figure, the black line represents the reference trajectory given by the MPC and the blue line the prediction of the neural network.

Algorithm 2 Pseudo code used to obtain a neural network prediction

```
procedure MAIN
    $MAX\_TIME \leftarrow 1000$
    $N \leftarrow 5$
    $DT \leftarrow 0.1$

    $time \leftarrow 0$
    $neural\_network \leftarrow$ load a previously trained neural network
    $trajectory \leftarrow$ List of vector
    $commands \leftarrow$ List of vector
    $x, y \leftarrow$ generate the coordinates of a 200m track
    $cx, cy, c\phi, cv \leftarrow$ generate the reference value for each variable at each point
  of the track using $x$ and $y$
    $S \leftarrow \text{Vector}(x = cx[0],\ y = cy[0],\ \phi = c\phi[0],\ v = 0.0)$
    while $time < MAX\_TIME$ and $S.x < cx[-1]^1$ do
        $stateRef \leftarrow$ reference trajectory for the N next steps using $cx, cy, c\phi$ and $cv$
        $S, U \leftarrow$ neural_network.predict($S, stateRef$)
        $trajectory$ add $S$
        $commands$ add $U$
        $time$ += $DT$
    end while
end procedure
```

1: The index -1 represents the index of the last element of the array.

4.3 Second Experiment

As we saw in section 3.6, it is possible to describe a complex and non-linear function, as is our case, by multiplying the number of neurons on the hidden layers. The intuition of putting 500 neurons on each layer in the neural network of section 4.2 was, therefore, a good idea. We will keep this parameter. At first sight, our model has nothing recursive, no apparent intermediate calculation. So we will choose a single hidden layer for this new network. We will come back to the number of hidden layers in the cross-validation section 4.4.

To summarise, we therefore take a network with 29 entries representing the reference trajectories over the next 5 states $cx, cy, c\phi$ and cv as well as the current state x, y, ϕ, v and δ. There is a hidden layer of 500 neurons with the ReLU activation function. The output layer has two neurons, one for acceleration and one for delta rate, and is equipped with the softsign function. In the table 4.2, you can find the full description of our network. Listing 4.1 shows the code used to create the network.

```python
# input_size = 29
# layers = 1
# nodes = 500
# output_size = 2
net = tflearn.input_data(shape=[None, input_size, 1])
for _ in range(layers):
    net = tflearn.fully_connected(net, nodes, activation='relu')
net = tflearn.fully_connected(net, output_size, activation='softsign')
net = tflearn.regression(net, optimizer='adam', loss='mean_square',
   learning_rate=0.0001)
model = tflearn.DNN(net) # We create de DNN (Deep neural network)
[...] # Loading and shaping X and y
model.fit(X, y, n_epoch=40)
```

Listing 4.1: Python code use to create the neural network of section 4.3

DNN Structure	
Type	Fully_connected
Number of layer	1
Number of nodes on the input layer	29
Number of nodes on the hidden layer	500
Number of nodes on the output layer	2
Activation function of the hidden layers	ReLU function
Activation function of the output layer	SoftSign function
Number of epoch	40
Loss function	Mean Square Error
Optimizer	Adam
Learning rate	0,0001

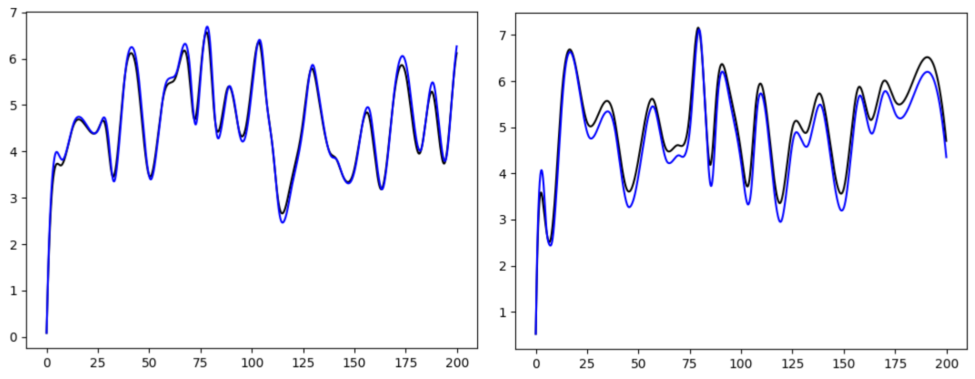
Table 4.2: Configuration of the neural network of section 4.3

We will use the same dataset as in the previous network containing about 92,000 states. As this model is much simpler than the previous one, the learning process is fast. So we set an epoch of 40. This means that the network will use the dataset 40 times to train itself.

In the figure 4.3a, we can see the prediction of the neural network on a path it has already seen. And on 4.3b, on a path, it has never seen.

We can see that the network is quite efficient. It is noticeable that the predicted path for a track it has already seen is closer to the reference than on a path it has never seen. This is the problem called overfitting. By using 40 epochs, the risk is that our model will perform very well on the dataset but much worse on paths it has never seen before. It becomes somewhat specialized on the dataset. To avoid this problem, we can simply generate a larger dataset or even an infinite dataset that is continuously generated as it is used.

As with our first model in Figure 4.2, Figure 4.4 shows a comparison between the outputs given by the neural network and the predictions of the MPC for one track. The trajectory described by the car associated with its commands is shown in figure 4.3a. It can be seen that for the acceleration (figure 4.4a), as the variations of values are too important from one step to the next, the network is not able to retranscribe faithfully the MPC outputs. In chapter 5, we will come back to an option to overcome these abrupt changes. The network remains close to zero and varies very little. On the 4.4b figure, the outputs of the neural network correspond more to the outputs of the MPC. It should be noted that as the acceleration is beside zero, the travel time of the track is greater than with the MPC. Therefore, the commands of the network are time-shifted with respect to the MPC and terminate later.



(a) On a track it has already seen **(b)** On a track it has never seen

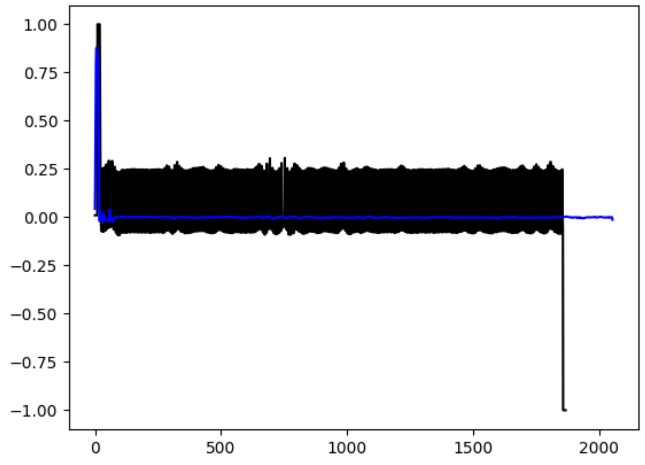
Figure 4.3: Prediction of trajectories with the neural network of the section 4.3.
The black line is the reference trajectory, the blue line is a prediction of the network.

4.4 Cross Validation

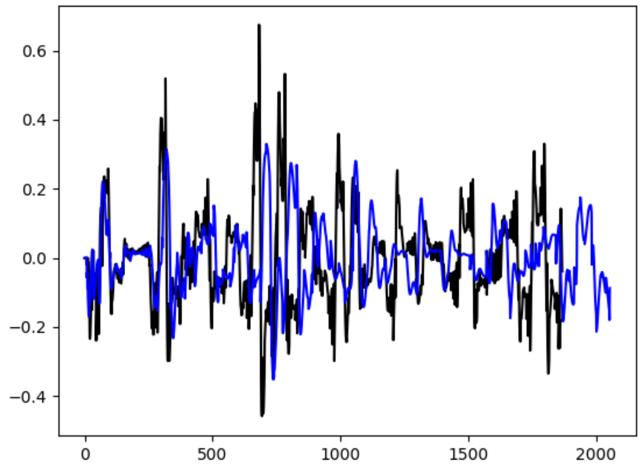
In order to compare networks, we need criteria. We train our network on a data set and then simulate the results on a validation set. For each track prediction, we need to check whether it is consistent with what is expected. That is, whether it follows the reference curve.

To do this, we will look at four things:

- **Is the route described by the prediction fast enough?** If the travel time to the end of the reference route is more than 1.5 times the time taken by the MPC, we consider the prediction invalid.



(a) Comparison of the acceleration output



(b) Comparison of the deltarate output

Figure 4.4: Comparison of the controls of the second model (table 4.2) and the MPC on a 200 meters track as a function of time (10 units on the x-axis correspond to 1 second). It is the same track as figure 4.2. The prediction of the neural network on this track is shown in 4.3a. In each figure, the black line represents the reference trajectory given by the MPC, and the blue line the prediction of the neural network.

- **Does the prediction arrive close to the arrival of the reference?** In our experiment, the reference path is 200 meters long, so we will check if the prediction of the neural network ends well beyond 195 meters. If this requirement is not met, the prediction will be considered invalid.
- **Does the prediction turn around?** Our path generator only produces paths in one direction, with increasing x's. This means that for a given x, there will always be one and only one corresponding point on the reference track. There is therefore no reason to turn back. If this requirement is not met, the

prediction will be considered invalid. (Here, this criterion is due to our track generator, it is quite possible to imagine different situations)

- **Does the prediction go too far off track?** For each x of the track, we will calculate the deviation from the prediction and the reference. If the maximum deviation is above 2% of the length of the track, we consider the prediction as invalid. In our case, the length is 200m so we allow a maximum deviation of 4 meters. 2% seems to be a good compromise. It allows to limit the track departure of the car and thus to guarantee its safety. But it is not too restrictive for the MPC or our neural networks to validate this condition.

These conditions are interesting because they can be checked during the prediction. The prediction corresponds to the *while* loop in pseudo-code 2. By testing the conditions during each iteration of the loop, it is, therefore, possible to exit the loop before these exit conditions are reached and thus save execution time. If the prediction is turned around at the beginning, we can stop the calculation and consider the path invalid. The same goes for the maximum deviation criterion. This gives us good time efficiency.

However, in our experiments is possible that the validation set is harder than the data set used for training. This means that the validation set can have much more complicated paths than the training set. And so, as our network will never have seen this kind of path, it will fail. Thus, we will use the cross-validation method, figure 4.5.

Assuming a large dataset, we will separate it into M parts. We will train the model on $M - 1$ parts and then validate with the remaining set. And we're going to do that for each part. That is, each part will be the validation set in turn. We can then average the results for each validation set and give a score to the neural network studied.

For our experiments, we will take a dataset consisting of 250 tracks of 200 meters in length and an M equal to 5. Thus we will have training on approximately 370 000 states multiplied by the number of epochs chosen. And a validation on about 90 000 states: 20% of the total dataset (480 000 states).

In the table 4.4, we can see the results of the cross-validation method for 25 different networks. Table 4.3 recalls the description of the networks. These are percentages obtained by summing the number of valid predictions divided by the total number of predictions. It shows the results for an epoch worth 10. In order to have reliable results, we performed five cross-validations. Thus the results displayed are averages over 25 identical neural networks and this simulation took around 60 hours.

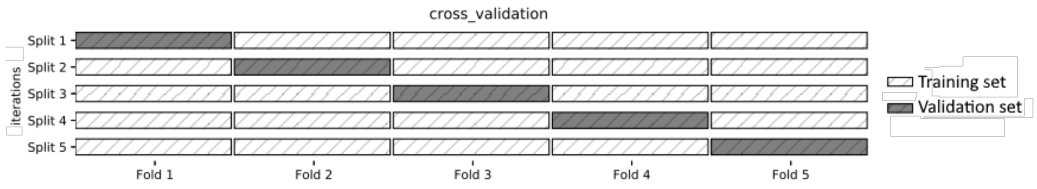


Figure 4.5: Cross-validation concept with a dataset separated into 5 parts

DNN Structure	
Type	Fully_connected
Number of nodes on the input layer	29
Number of nodes on the output layer	2
Activation function of the hidden layers	ReLU function
Activation function of the output layer	SoftSign function
Number of epoch	1
Loss function	Mean Square Error
Optimizer	Adam
Learning rate	0,0001

Table 4.3: Configuration of the deep neural network used for cross-validation

		Number of nodes on each layer				
		50	150	250	350	450
Number of hidden layers	1	6.0	11.6	7.6	12.8	4.8
	2	0.4	0.0	0.0	0.0	0.0
	3	0.0	0.0	0.0	0.0	0.0
	4	1.2	0.0	0.0	0.0	0.0
	5	0.0	0.0	0.0	0.0	0.0

Table 4.4: Percentage of valid predictions for an epoch of 10. Each value is an average of 25 networks validating on 50 channels each. Running time: 60 hours

The majority of the percentages are zero. Therefore, we cannot conclude anything. Only the first two rows have non-zero values, even if they are very low. This is due to the learning rate, as there are more layers, and the learning rate is small, a lot of data is needed to train the network correctly. Our dataset or the number of epochs is therefore too small or the learning rate must be increased.
After some experiments, by increasing the learning rate, the neural networks perform less well than in the table 4.4. They fail to converge to a correct model. We will therefore leave the learning rate low (0.001) but increase the number of epochs. In table 4.5, you can find the results of our second simulation for an epoch worth 40.

Using the table 4.5, we can see that the percentage decrease the more layers there

		Number of nodes on each layer				
		50	150	250	350	450
Number of hidden layers	1	62.0	41.2	36.7	31.5	34.2
	2	40.5	52.7	37.9	37.0	43.0
	3	18.1	17.1	18.3	15.7	11.3
	4	16.0	14.5	10.0	10.9	5.3
	5	4.8	2.0	5.6	6.5	2.3

Table 4.5: Percentage of valid predictions for an epoch of 40. Each value is an average of 25 networks validating on 50 channels each. Running time: 100 hours

		Number of nodes on each layer				
		50	150	250	350	450
Number of hidden layers	1	[42.6 ; 81.4]	[24.9 ; 57.5]	[17.5 ; 55.9]	[14.3 ; 48,7]	[16.0 ; 52.4]
	2	[25.4 ; 55.6]	[36.2 ; 69.2]	[21.8 ; 54.0]	[21.2 ; 52.8]	[26.5 ; 59.5]
	3	[5.4 : 30.8]	[8.0 ; 26.2]	[9.4 ; 27.2]	[5.1 ; 26.3]	[1.6 ; 21.0]
	4	[5.1 ; 26.9]	[5.1 ; 23.9]	[3.0 ; 17.0]	[3.5 ; 18.3]	[3.0 ; 17.0]
	5	[0.0 ; 9.6]	[0.4 ; 3.6]	[0.0 ; 12.8]	[1.9 ; 11.1]	[0.4 ; 4.2]

Table 4.6: 95% confidence interval for the values of table 4.5

are. We cannot deduce a general truth for our study. Because we can see that the results are all better than with 10 epochs (table 4.4). Therefore, training the network more produces better results. Maybe by training more networks with many layers and many neurons on each layer, we will get better results. But we can conclude that networks with one hidden layer can get correct results with less training time than larger neural networks. This conclusion coincides with the results of [13] which classifies neural networks with a hidden layer, equipped with the ReLU activation function, as "universal approximators".

Chapter 5

Alternative Dataset

This chapter discusses about creation of more relevant datasets in order to achieve better results while training neural networks. This part is unable to produce useful results but we think it would still be of interest to present this area of our investigations. Some details of the research are presented in the appendix.

5.1 Post Processing

As stated in section 4.2, acceleration a and $deltarate$ of the vehicle, generated by the MPC, are subject to important variations from a state to another. For example, in figure 4.2a and 4.2b, MPC outputs are unstable and the neural network predictions (blue) struggle to keep pace with the black curve. Section 4.2 suggested that the data extracted from the MPC could be smoothed. To perform this and get more accurate data, there are two options. Either modify the MPC generation and add constraints or perform post-processing on the data. It is possible to add constraints in the MPC parameters. For example, $\dot{a}$ the variation of acceleration between two steps could be regulated. We could state that $\dot{a}$ should be lower than a defined value. Nevertheless, the MPC is working well and other parts of the project are based on it. Such a modification could impact the progress of the other sections of the project. Therefore, post-processing with averaging appears to be a less risky process.

The aim of the post-processing is to make a and $deltarate$ more readable and easier to compare with the neural network results by the creation of an alternative dataset. In the alternative dataset, every a and $deltarate$ values are replaced by an average of themselves and their neighbors' values. Thus, the variable n is defined, it represents the number of neighbors values used in the average.

Figures 5.1a and 5.1b show data (acceleration and deltarate) before (black curve) and after (blue curve) post processing, with n equals to 11, for a given dataset.

To modify outputs values of the MPC could imply modifying the trajectory of the car. Indeed, each state depends on the previous state values and the values of acceleration and delta rate. Figure 5.1c is the trajectory computed by the MPC, with (blue curve) and without (black curve) post-processing. The red curve is the reference trajectory.

To decide which value to choose for n, more investigations are provided in the appendix section 9.1. The section establishes that $n = 5$ is suitable. n is selected in a way that it can both : keep a realistic shape (close to the previous trajectory) and maximize variations of a and $deltarate$.

To create the dataset, we generate trajectories on which post-processing is applied. Nevertheless, such modifications imply that some trajectories have averaged a and $deltarate$ which are too far from reality. We do not want these trajectories to be in the dataset. These trajectories are named outliers. The appendix section 9.2 details how they are removed from the dataset.

5.2 Comparison of the Alternative Dataset

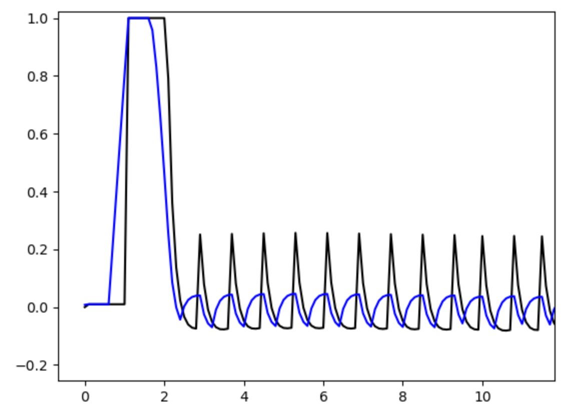
The goal of this part is to see if a neural network trained with less changing data can produce better outcomes.

The dataset consists of five post-processed training sets (generated as described in section 5.1) with approximately fifty trajectories each. Fifty trajectories equate to around 100000 states. To make a good comparison, the same configuration as in section 4.4 is used. The train is realized across forty epochs. The neural Network considers a trajectory acceptable if the track deviation is less than 2% of the track length. A total of 25 neural networks, with a different number of layers and nodes, are trained. The number of layers utilized ranges from one to five, with each layer containing 50, 150, 250, 350, or 450 neurons.

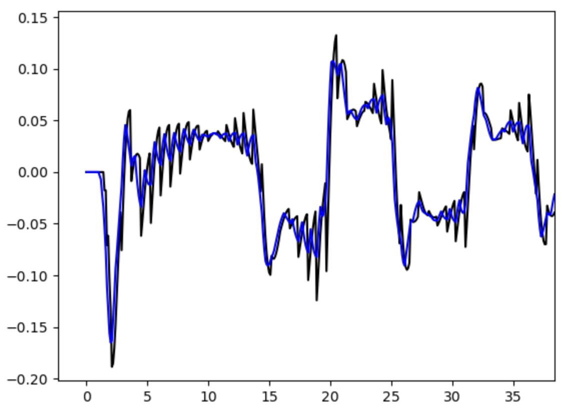
The table 5.1 indicates the percentage of valid predictions for each of the networks. For each configuration, neural networks fail to generate a relevant prediction of the car path. The conclusion could be that reducing the variation of outputs does not improve the MPC accuracy. Moreover, it degrades the quality of the neural networks that are trained. These results could be explained by the fact that the reference trajectory's acceleration and delta rate are influenced by more than just the next N reference trajectory steps. N is the horizon length equates to 5. Indeed, the smoother acceleration and delta rate is averaged over the next five MPC output values, and each output is dependent on the reference trajectory's N following steps. It indicates that the tenth following step of the reference trajectory could alter smoothed acceleration or delta rate, even though this information is not sent to the Neural Network. The neural network inputs contained in the variable $stateRef$ the only information until the fifth next step of the reference trajectory (section 2.4.3).

		Number of nodes on each layer				
		50	150	250	350	450
Number of hidden layers	1	0.0	0.0	0.0	0.0	0.0
	2	5.3	1.0	0.0	0.0	0.0
	3	0.0	0.0	0.0	0.0	0.0
	4	0.0	0.0	0.0	0.0	0.0
	5	0.0	0.0	0.0	0.0	0.0

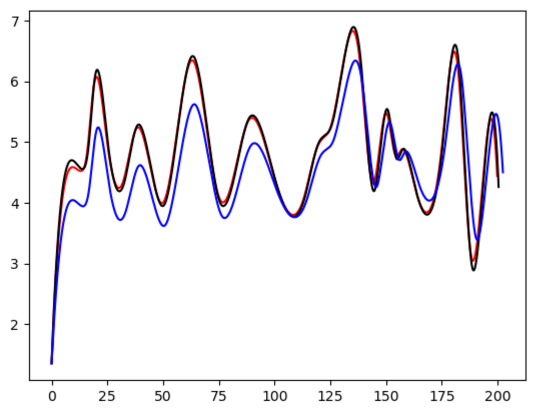
Table 5.1: Percentage of valid prediction with a set of post processed data ($n = 5$), and for an epoch of 40.



(a) Variations over the time (s) of a before (black) and after (blue) smoothing, $n = 11$



(b) Variation over the time (s) of $deltarate$ before (black) and after (blue) smoothing, $n = 11$



(c) Reference trajectory (red) and trajectories of a vehicle before (black) and after (blue) post processing, $n = 11$

Figure 5.1

Chapter 6

Recurrent Neural Network Experiment

6.1 Two Types of Recurrent Neural Networks

The recurrent neural network can be a good option because the predictions are based on the previous predictions, indeed the operations are similar to the operations made by an MPC which calculates the next inputs in the function of the previous inputs. A recurrent neural network needs to store foregoing inputs to be effective, there exist two specialized types of recurrent layers which have memory, the Long Short-Term Memory called LSTMs and the Gated Recurrent Unit called GRUs.

6.1.1 Long Short-Term Memory layer

An LSTMs is composed of a cell state and different gates. A cell state can be compared to the memory of the network, then the gates can choose to remove or add some information into the memory. An LSTMs counts three gates to protect the cell state, each of them has a sigmoid layer and a pointwise multiplication operation that allow the gate to determine the quantity of information it has to let go through the network based on the outputs of the sigmoid layer [10]. Figure 6.1 shows the structure of a LSTMs layer and its components.

The process of an LSTMs unit is different from a process of a common neural network unit because to predict the best output, it needs more intermediate steps. That is why in an LSTMs unit, some decisions are made by the gate layers. The first decision is to know which information needs to be removed, this decision made by the first sigmoid layer is called the forget gate layer.

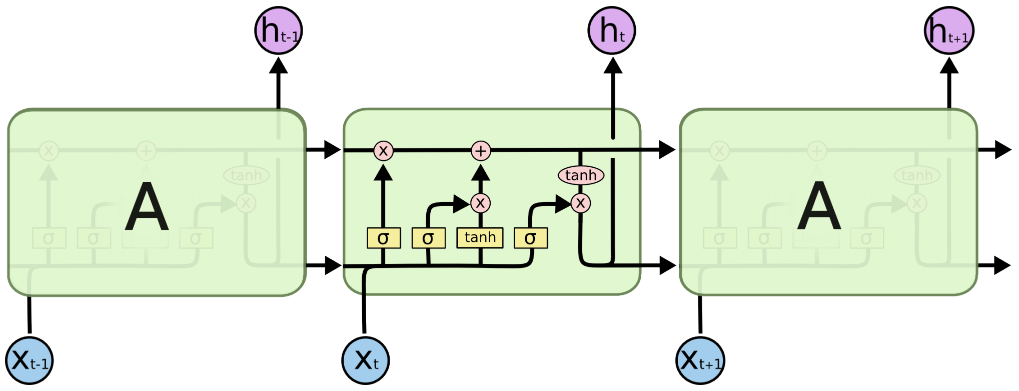


Figure 6.1: Operations performed by an LSTMs unit taken from [10]

The next operation is to choose which information needs to be added to the cell state, this operation requires two phases. First, a sigmoid layer called the input gate layer decides which values need to be updated. In the second phase, a tanh layer creates a vector of new candidate values.

Now the old cell state needs to be updated with the new candidates' values and forget the information that is not needed anymore. The last step is to get the output values, then we need to select only the chosen values in the state cell based on applying the cell state to a sigmoid layer which decides what parts of the cell state are going to be the final output.

The following equations describes the tanh function (6.1) and the sigmoid function (6.2).

$$tanh(x) = 2 * sigmoid(2x) - 1 \tag{6.1}$$

$$sigmoid(x) = \frac{e^x}{1 + e^x} \tag{6.2}$$

6.1.2 Gated Recurrent Unit Layer

The other variant is the GRUs, the workflow in GRUs is practically the same as in LSTMs. GRUs counts two gates an update gate and a reset gate, moreover, it does not have a memory cell state, this unit uses directly the hidden state that stores the sequence of information (Figure 6.2). Basically, the update gate keeps the hidden state up to date, so this gate is responsible for determining the amount of previous information that needs to pass along to the next state. Likewise, the reset gate decides how much of the previous state need to be stored [10].

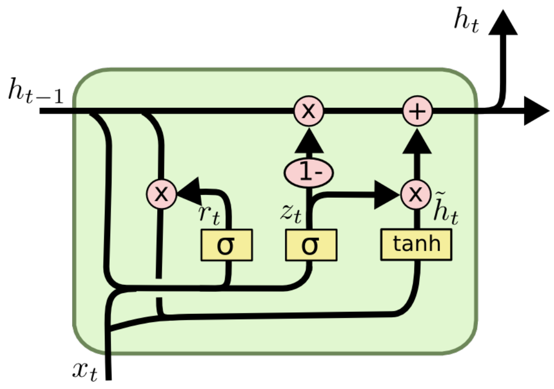


Figure 6.2: Operations performed by a GRUs unit [10]

6.2 Experiments on Recurrent Neural Networks

6.2.1 The Parameters

Now we need to try some recurrent neural networks and observe how they perform to find a good model. As we saw in sections 3.6 and 3.7, the number of layers does not need to be high, moreover, only one layer suffices. In our case, we need a large number of nodes to be able to fit as accurately as possible the output data as the output data is pretty difficult to fit as seen in figures 4.3a and 4.3b. Also, we observed that a deep neural network with only one hidden layer can perform well, that is why a neural network composed of one recurrent layer with a large number of a node should be sufficient to adapt any functions.

The number of nodes is the parameter that decides the learning capacity of the neural network and therefore the more the number of nodes is high, the best it can fit difficult data and functions. We already know that choosing 500 as the number of nodes can give us accurate predictions as seen in the section 4.3. Then a recurrent layer with 500 nodes must be good enough to obtain good results.

The choice of the optimizer is also important because the optimizer defines the actions performed during the gradient descent and can accelerate the training time and have more precise results. In our case, the Adam optimizer is well-suited like it is based on a stochastic gradient descent that computes adaptive learning rates for different parameters from estimates of first and second moments of the gradients. The name Adam is derived from adaptive moment estimation. Adam is robust and well-suited to a wide range of non-convex optimization problems in the field of machine learning [9].

Finally, we need to fix the value of the learning rate, *"**A default value of 0.01***

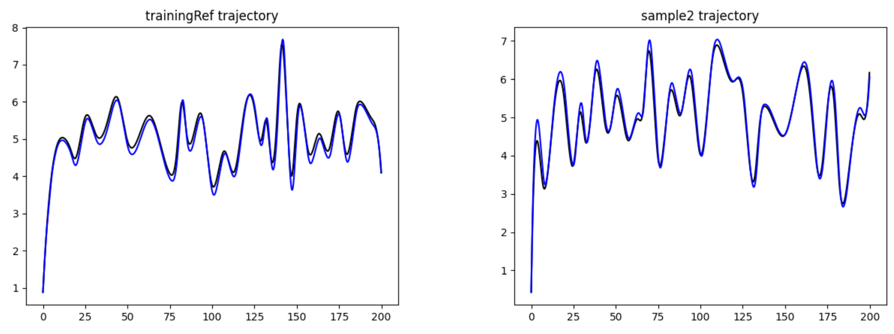
typically works for standard multi-layer neural networks" [4]. Then we decide to fix the learning rate at 0.001 because is small enough to find a local minimum.

6.2.2 Experiment with a Long Short-Term Memory Layer

Let's try to implement some recurrent neural networks using an LSTMs layer of a large number of nodes. In this experiment, the neural network is composed of one input layer of 29 nodes corresponding to the reference trajectory over the next 5 states $cx, cy, c\phi$ and cv, one LSTMs layer of 500 nodes, and one output layer of 2 nodes for the acceleration and the delta rate with a tanh function as activation function. The dataset used in this experiment is the same dataset used in Section 4.3, so we use the data from 50 tracks of 200 meters, which represents a dataset of around 92.500 states. The structure of this network is described in the table 6.1. Here is the code used to program this network :

```
def rnn(input_size, output_size):
    # Network building
    net = tflearn.input_data(shape=[None, input_size, 1])
    net = tflearn.lstm(net, 500)
    net = tflearn.fully_connected(net, output_size, activation='tanh')
    net = tflearn.regression(net, optimizer='adam', loss='mean_square'
    , learning_rate=0.001)

    # Define model
    model = tflearn.DNN(net)
    return model
```

Listing 6.1: Code for the recurrent neural network using 1 LSTM layer



(a) On a track it has already seen **(b)** On a track it has never seen

Figure 6.3: Prediction of trajectories with the Adam optimizer
The black line is the reference trajectory, the blue line is a prediction of the network.

This experiment is positive, as seen in the figures 6.3a and 6.3b, the predicted curve follows rather well the reference trajectory.

RNN Structure	
Type	LSTM
Number of layer	3
Number of nodes on the input layer	29
Number of nodes on the hidden layer	500
Number of nodes on the output layer	2
Inner activation function of the hidden layer	Sigmoid function
Activation function of the hidden layer	Tanh function
Activation function of the output layer	Tanh function
Number of epochs	10
Loss function	Mean Square Error
Optimizer	Adam
Learning rate	0,001

Table 6.1: Configuration recurrent neural network using 1 LSTM layer

6.2.3 Experiment with a Gated Recurrent Unit Layer

Let's try to implement a recurrent neural network using a GRUs layer. In this experiment, the neural network is composed of one input layer of 29 nodes corresponding to the reference trajectory over the next 5 steps $cx, cy, c\phi$ and cv, one GRUs layer of 500 nodes, and one output layer of 2 nodes for the acceleration and the delta rate with a tanh function as activation function. The dataset used in this experiment is the same dataset used in the Section 6.2.2 above. The structure of this network is described in the table 6.2. The code to program a recurrent network using a GRUs layer is not really different than the code for an LSTMs layer because it's just the type of the layer that changes.

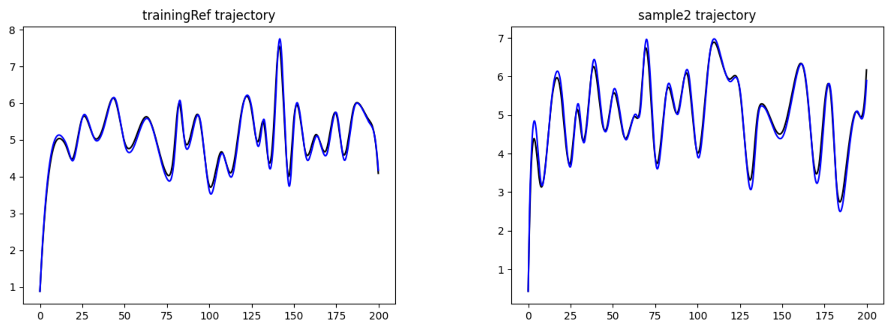
```python
def rnn(input_size, output_size):
    # Network building
    net = tflearn.input_data(shape=[None, input_size, 1])
    net = tflearn.gru(net, 500)
    net = tflearn.fully_connected(net, output_size, activation='tanh')
    net = tflearn.regression(net, optimizer='adam', loss='mean_square'
    , learning_rate=0.001)

    # Define model
    model = tflearn.DNN(net)
    return model
```

Listing 6.2: Code for the recurrent neural network using 1 GRU layer

This experiment is positive as seen in the figures 6.4a and 6.4b. The predicted path follows well the reference trajectory.

RNN Structure	
Type	GRU
Number of layer	3
Number of nodes on the input layer	29
Number of nodes on the hidden layer	500
Number of nodes on the output layer	2
Inner activation function of the hidden layer	Sigmoid function
Activation function of the hidden layer	Tanh function
Activation function of the output layer	Tanh function
Number of epochs	10
Loss function	Mean Square Error
Optimizer	Adam
Learning rate	0,001

Table 6.2: Configuration recurrent neural network using 1 GRU layer



(a) On a track it has already seen

(b) On a track it has never seen

Figure 6.4: Prediction of trajectories with the GRU of the section
The black line is the reference trajectory, the blue line is a prediction of the network.

6.2.4 Performance Analysis

Now, based on these experiments in Sections 6.2.2 and 6.2.3, we can know that both the LSTMs and GRUs are able to perform well and obtain good results. Thus the next part is to make more experiments on both of them and evaluate them to know the accuracy of their predicted path and the frequency of their valid predictions.

6.3 Cross Validation for Recurrent Neural Network

In order to evaluate the recurrent neural network using an LSTMs layer or using a GRUs one, we decide to run some tests on both of them. These tests will be structured in the same manner as Section 4.4. These experiments are made only on neural networks composed of one layer and different numbers of nodes for a question of training time, moreover they are trained for 5 epochs because this number of epochs is sufficient to obtain appropriate results.

Let's run the cross-validation algorithm on several networks using an LSTMs layer and a GRUs layer. The dataset is the same as the one used in the experiments in Section 4.4, so 80% of the dataset is used for training and 20% is used for validation. The results for the neural networks using a LSTMs and GRUs layer for 5 epochs are in the tables 6.3 and 6.4. It describes the percentage of valid predicted path for each neural networks configuration and the confidence interval for each configuration.

	Number of nodes				
	50	150	250	350	450
LSTM	55.2	69.68	85.04	68.56	75.28
GRU	80.16	58.8	66.72	55.06	50

Table 6.3: Percentage of valid predictions for an epoch of 5. Each value is an average of 25 networks composed of 1 LSTM layer or 1 GRU layer validating on 50 tracks. Running time for LSTM : 108 hours and Running time for GRU : 118 hours

	Number of nodes				
	50	150	250	350	450
LSTM	[41.3 ; 68.7]	[56.5 ; 80.9]	[76.1 ; 94]	[56.3 ; 80.9]	[64.3 ; 86.2]
GRU	[69.5 ; 90.8]	[45.6 ; 72]	[53.4 ; 81.1]	[43.2 ; 66.8]	[30.9 ; 69.1]

Table 6.4: 95% confidence interval for the values of Table 6.3

In the table 6.3, for each configuration 25 networks have been trained, then an average of the valid predictions obtained for each configuration is made. For the LSTMs, we can observe that for 50 nodes, more than half of the predictions are valid. The percentage increases for 150 nodes and reaches almost 70%. Again with 250 nodes the percentage of valid path predicted is rather high with 85%. After when using 350 nodes, the percentage decreases, moreover for the last configuration, the validity percentage is higher than 75%. We can interpret these results in this way, a value of 50 nodes is not high enough to obtain really good results, all the percentages except the one at 50 nodes are quite good, furthermore 250 nodes are for 5 epochs of training the best configuration and finally with 350 and 450 nodes a higher value for the number of epochs can probably obtain better results than an epoch of 5.

Similarly, for the GRU, we can observe that with 50 nodes, it reaches a validity percentage of 80%. However, with 150 nodes, the percentage decreases to less than 60%. Again, with 250 nodes, the percentage is above 65% before dropping to 50% for 350 and 450 nodes. On this basis, we can say that for 5 epochs, the best configuration is to fix the number of nodes at 50, moreover, we can assume that 5 epochs are probably not sufficient for a configuration with more nodes since all the percentages are lower than the one with 50 nodes.

To conclude, a neural network using an LSTMs layer is overall better than another one using GRUs, all the validity percentages are better with LSTMs except for the configuration of 50 nodes. The neural network using LSTMs is rather reliable as we can see in Table 6.3, all the validity percentages are almost at 70% except with 50 nodes. On the other hand, the GRUs shows bad results and therefore seem to be an invalid alternative to replace an MPC.

Chapter 7

Time Efficiency

Now that we have studied several options, several types of neural networks, we can compare these solutions with each other. During our experiments, we saved the networks with good results. If the cross-validation results were above 80%, we saved the network.

To quote our neural networks, we will use the following notation: *dnn*_1_50_40_96. Where *dnn* is the type of network. DNN stands for Deep Neural Network, LSTM for Long Short-Term Memory, and GRU for Gated Recurrent Unit. The next value 1 represents the number of hidden layers. Then 50 is the number of nodes on the hidden layers. 40 is the number of epochs used to train the network. Finally, 96 is the percentage of valid prediction of this network given by the cross-validation.

After generating a random path via our generator (section 2.5). We will measure the time required for each network and also for the MPC to predict the path to be taken. Furthermore, we will analyze whether the prediction is valid or not to have a percentage of confidence in the results. For example, even if a network is five times faster than the others, if only 50% of its predictions are valid it is not very conclusive. The analysis time of the path is not included in the prediction time. The results are presented in the table 7.1.

These results were obtained on 425 different predictions. As we selected good neural networks, the percentages of valid predictions are quite good. It can be seen that the MPC does not have a 100% score. And as we used data created by the MPC to train our networks, it will be hard to get better results than the MPC.

In terms of time efficiency, MPC is efficient with an average prediction time over a 200 meters track of 1.5 seconds. Then comes the deep neural networks with an average time between 2.5 and 3 seconds with results equivalent to the MPC. The

Name	Percentage of valid prediction (%)	Average prediction time (s)	Time comparison to the MPC in percent (%)
MPC	99.53	1.46	-
dnn_1_5_40_96	97.41	2.65	181.5
dnn_1_50_40_100	99.53	2.59	177.4
dnn_1_250_40_100	99.53	2.37	162.4
dnn_1_450_40_100	100.00	2.51	171.9
dnn_1_500_70_86	84.94	3.00	205.5
dnn_2_150_40_90	88.00	2.72	186.3
dnn_2_150_40_92	94.59	2.48	169.9
dnn_2_250_40_98	93.65	2.63	180.1
dnn_2_450_40_92	92.47	3.06	209.6
lstm_1_150_10_100	98.59	7.26	497.3
lstm_1_250_10_96	100.00	9.48	649.3
lstm_1_250_10_98	98.53	9.39	643.2
lstm_1_350_10_100	100.00	14.47	991.1
lstm_1_450_5_94	40.47	30.65	2,099.3
gru_1_50_5_100	69.41	7.38	505.5
gru_1_150_5_98	84.00	7.98	546.6
gru_1_250_5_100	99.76	9.45	647.3

Table 7.1: Comparison of prediction time for several networks and the MPC. These values are based on a total of 425 predictions of 200 meters long tracks.

GRU recurrent networks come second to last, taking five times longer than the first by about 8 seconds. Moreover, their percentage of valid predictions is much lower, so they are not reliable. Finally, we have the LSTM networks with times between 7 and 30 seconds.

Thus, it seems that the MPC is very efficient even if it has to take into account, and therefore calculate, several states in the future. As the networks are trained with data from the MPC, they can never theoretically outperform the MPC itself in terms of prediction. Thus, even though the networks do not need to optimize on future states but only rely on a fixed mathematical model (once the training is finished), they are not more efficient in terms of time but still interesting with a similar order (less than 10 times slower). However, we find that recurrent networks (LSTM and GRU) are not suitable for our problem because their time efficiency is far from what we are looking for and the reliability of their prediction is not better or worse.

Chapter 8

Conclusion

Our study focused on the effectiveness of a model predictive controller (MPC) against neural networks. The MPC is a widespread technology nowadays but relatively expensive in terms of performance because it must simulate the future to optimize its choices in the present. On the other hand, neural networks are based on a fixed mathematical model that allows a constant time response. In the first part, we have defined an MPC based on an example using the ACADO Toolkit library. After supervised learning on different neural networks with data obtained by an MPC, machine learning allowed to obtain similar results. The three types of networks we tried: Deep Neural Network, Long Short-Term Memory, and Gated Recurrent Unit, proved to be less reliable than the MPC. Their predictions were often worse and their execution time more than doubled.

Thus, our study shows that it is possible to imitate the behavior of an MPC with neural networks. The more you train a network with a large and diverse data set, the more likely the network will be able to reproduce the desired commands. However, the temporal performance is not conclusive. Therefore, it is not possible to replace an actual MPC with a neural network. As the reaction time is longer, the commands could arrive too late and cause the vehicle to crash risking the safety of passengers. Furthermore, if the network is trained from the data of an MPC, it will ideally only mimic the MPC but never be better.

Nevertheless, in reality, the model of a vehicle is not as simplistic as the one in our example. It will be described by much more complex equations and with more state variables. Thus, on a real model, the MPC will tend to see its execution time increase enormously due to the difficulty of the problem. Whereas neural networks, for a given model, have an almost constant prediction time. Only the training time will be increased. Consequently, the results announced here are to be qualified with the simplicity of our model.

Chapter 9

Appendix

9.1 Alternative dataset : define post processing parameter n

Input and output generated by the MPC have a lot of variations from one step to another. The fact is that these oscillations could be difficult to predict by the neural network, due to the amount of oscillation. A way to improve the data (before testing if the results are more accurate) is to smooth the MPC outputs (acceleration and delta rate). We discuss a way to improve the data generated. The solution is to convert MPC outputs values as an average of the n following values. In this appendix section, we explain how n is chosen.

n is the number of neighbors values used in the averaging of a and $deltarate$. On one hand, n should not be too high because its values would depend too much on the neighbors, data accuracy would be lost. For example, the blue curve of figure 9.1 represents the trajectory with $n = 50$. It does not follow the MPC computed trajectory (black curve). On the other hand, n should be high enough to smooth the values. If n is low, even though variations will be less important, the curve will still be very rough.

To determine n, an analysis can be performed where values should be tested on a large sample of trajectories. For instance, in figure 5.1c, $n = 11$ looks to fit with the particular trajectory used because the gap between the two curves is small. Nevertheless, it could also be inadequate with the rest of the dataset. Therefore, the following values of n are chosen and will be compared : 3, 5, 11, 13, 25, 51, 101. For each value, a dataset of one thousand trajectories is generated. For each trajectory, the post-processing on delta rate and acceleration is performed (with the current value of n). Then, trajectory with post-processing is compared to trajectory without post-processing.

In the table 9.1 results are presented for each value of n. The fluctuation decrease on a (named fda) represents how much the variation of acceleration is reduced and is calculated on equation 9.1. This rate is established by the calculation of how the variation of acceleration between neighbor post-processed states is reduced in comparison to acceleration without post-processing. The final value is the mean of this rate on the complete sample.

$$fda = \sum_{j=1}^{s\_size} \sum_{i=1}^{n\_steps} 1 - \frac{|pp\_a_j(i+1) - pp\_a_j(i)|}{|in\_a_j(i+1) - in\_a_j(i)|} \tag{9.1}$$

In equation 9.1, n_steps is the number of steps found by the MPC to reach the arrival point, for the current trajectory. s_size is the size of the sample. pp_a is the post-processed value of acceleration. in_a is the initial value of a, without post-processing.

The fluctuation decrease on $deltarate$ is calculated similarly. In the equation 9.1, acceleration values are replaced by delta rate values.

In table 9.1, $distance$ is for each round of the sample, the sum of the euclidean distance between points of post processed trajectory and points of the initial trajectory. Mean, median, standard deviation, min and max values of distance in the sample are calculated.

n	Fluctuation decrease on a (%)	Fluctuation decrease on $deltarate$ (%)	Median distance (m)	Mean distance (m)	Standard deviation distance (m)	[Min;Max] distance (m)
3	43%	27%	0.06	1.00	3.1	0.00;33.4
5	66%	48%	0.17	2.50	7.5	0.00;64.3
11	83%	64%	0.76	6.19	15.6	0.03;99.3
13	87%	68%	2.6	8.4	16.8	0.82;106.6
25	93%	78%	53.05	53.24	15.1	10.8;105.8
51	96%	85%	104.8	92.9	27.6	6.0;123.0
101	97%	92%	114748	117307	69012	12894;252987

Table 9.1: Test different values of n with a sample of 1000 MPC outputs.

The aim is to decrease fluctuation of a and $deltarate$ to smooth the curve of their evolution and facilitate comparison of the MPC and the Neural Network. Simultaneously, an acceptable distance between new trajectory without and with post-processed values have to be kept. As we can see in the table, the more the value of n is important, the more fluctuation on a and $deltarate$ decrease, but the more *distance* is important.

It can be observed that the value $n = 5$ could fit better others. It looks to be a good compromise because it decreases 66% of the fluctuation of a and 48% of the fluctuation of $deltarate$. Moreover, the median distance with the initial value is still reasonable ($0.17m$). The fact is that the median distance grows exponentially with n. For example, the median distance with $n = 13$ is 2.6. It is more than twenty times more important than with n equates 5.

Once the value of n is chosen, the goal is to build a relevant dataset with smoothed values that could be trained and tested in a neural network. Even if a safe value of n is chosen, the mean of the distance is still disturbed by some trajectories which take a completely wrong shape. For example, if $n = 11$, the maximal distance is 99.3 which is too important and means that the smoothed curve deviated from the initial trajectory. Another example with n equals 50 is represented in figure 9.1.

We do not want these bad predictions to be part of the dataset. It could distort the training of the neural network. The section 9.2 explains how these elements, which can be named outliers, are removed from the dataset.

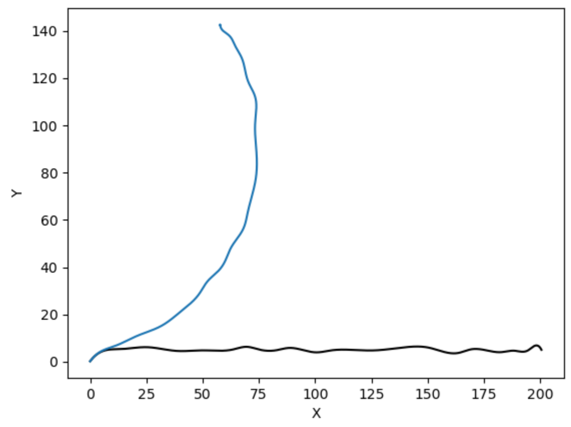


Figure 9.1: Trajectory of the car with post processed data (blue) with $n = 50$. The black curve is the MPC computed trajectory before post processing

9.2 Alternative dataset : remove outliers

Trajectories that are too far from reality are named outliers. To remove these outliers, a *distance exclusion value* is defined. It is a limit. For each trajectory in the dataset, the distance to the MPC computed trajectory is compared with the *distance exclusion value*. If the distance of the smooth trajectory to the MPC trajectory is under the limit, the smooth trajectory is accepted in the dataset. If the distance is above the limit, a smooth trajectory is considered an outlier and is rejected from the dataset.

In the table 9.2, different values of the distance exclusion value are tested (0.1, 0.2, 0.3, 0.5, 1, 5 and 10). The dataset used is a sample of 1000 trajectories, as in table 9.1. For any given values of n and a set of *distance exclusion value*, the percent of outliers in the sample has been calculated. The more the limit is high, the less the percent of outliers is important. For instance, if $n = 11$ and the limit equals 0.3, then 99% of the data is removed from the dataset while with a limit equal to 1, 32% of the dataset is removed. Moreover, the more the parameter n (introduced in section 5.1) is important, the more the quality of the dataset is bad. For example, given a limit equal to 0.2, if $n = 3$, 18% of the data is outlier while if $n = 11$, a hundred percent of the data is the outlier.

	Distance exclusion value						
n	**0.1**	**0.2**	**0.3**	**0.5**	**1**	**5**	**10**
3	23%	18%	17.5%	17%	15%	6%	2%
5	84%	39%	23%	19%	17%	12%	8%
11	99%	97%	93%	78%	32%	16%	14%
13	100%	100%	100%	100%	99%	17%	15%
25	100%	100%	100%	100%	100%	100%	100%
51	100%	100%	100%	100%	100%	100%	100%
101	100%	100%	100%	100%	100%	100%	100%

Table 9.2: % of excluded trajectories given n and for a set of *distance exclusion values*. The sample contains 1000 trajectories.

The aim is to get an acceptable number of outliers and minimize the distance with the MPC trajectory. The *distance exclusion values* 0.3 looks interesting because if $n = 5$, only 23% of the data is outlier. Moreover, it ties up with the conclusion of section 9.1 because the value $n = 5$ looks the best compromise. Once *distance exclusion* and n values are chosen, the neural network can be trained with the generated dataset, and results with or without smooth (on acceleration and delta rate) can be compared.

Bibliography

[1] Control Arts Alan Hugo Senior Applications Engineer. *Limitations of Model Predictive Controllers*. English. URL: `http://www.controlartsinc.com/Support/Articles/LimitationsOfMBC.pdf`.

[2] *An event-triggered Model Predictive Control scheme for freeway systems*. English. URL: `https://www.researchgate.net/publication/261459801_An_event-triggered_Model_Predictive_Control_scheme_for_freeway_systems`.

[3] *An Overview Of Industrial Model Predictive Control Technology*. English. URL: `https://www.researchgate.net/publication/2773527_An_Overview_Of_Industrial_Model_Predictive_Control_Technology`.

[4] Yoshua Bengio. *Practical Recommendations for Gradient-Based Training of Deep Architectures*. English. URL: `https://arxiv.org/pdf/1206.5533.pdf`.

[5] Jason Brownlee. *Calculus in Action: Neural Networks*. English. URL: `https://machinelearningmastery.com/calculus-in-action-neural-networks/`.

[6] *But what is a neural network? | Chapter 1, Deep learning*. English. URL: `https://www.youtube.com/watch?v=aircAruvnKk`.

[7] B. Houska et al. *ACADO Toolkit User's Manual*. English. URL: `https://acado.github.io`.

[8] Di Wu Benoit Boulet Franc¸ois Bouffard Geza Joos Huiliang Zhang Sayani Seal. *Data-driven Model Predictive and Reinforcement Learning Based Control for Building Energy Management: a Survey*. English.

[9] Diederik P. Kingma and Jimmy Ba. *arXiv:1412.6980*. English. URL: `https://arxiv.org/abs/1412.6980`.

[10] Christopher Olah. *Understanding LSTM Networks*. English. URL: `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`.

[11] Terence Parr and Jeremy Howard. *The Matrix Calculus You Need For Deep Learning*. English. URL: `https://arxiv.org/pdf/1802.01528.pdf`.

[12] *Practical Machine Learning Tutorial with Python Introduction*. English. URL: `https://pythonprogramming.net/machine-learning-tutorial-python-introduction/`.

[13] Poorya Mianjy Raman Arora Amitabh Basu and Anirbit Mukherjee. *Understanding Deep Neural Networks with Rectified Linear Units*. English. URL: http://www.optimization-online.org/DB_FILE/2016/12/5780.pdf.

[14] *Real-time Model Predictive Control (MPC) with ACADO and Python*. English. URL: https://grauonline.de/wordpress/?page_id=3244.

[15] *TensorFlow*. English. URL: https://tensorflow.org.

[16] *TFLearn: Deep learning library featuring a higher-level API for TensorFlow*. English. URL: http://tflearn.org/.

[17] *The explicit linear quadratic regulator for constrained systems*. English. URL: https://www.sciencedirect.com/science/article/abs/pii/S0005109801001741.

[18] *Understanding Model Predictive Control, Part 1: Why Use MPC?* English. URL: https://ch.mathworks.com/videos/understanding-model-predictive-control-part-1-why-use-mpc--1526484715269.html.

[19] *Understanding Model Predictive Control, Part 2: What Is MPC?* English. URL: https://ch.mathworks.com/videos/understanding-model-predictive-control-part-2-what-is-mpc--1528106359076.html.

[20] *User guide : Interpolation (scipy.interpolate)*. English. The SciPy community. URL: https://docs.scipy.org/doc/scipy/reference/tutorial/interpolate.html.

[21] Nils C. Hamacher Wolfgang R. Huster Adel Mhamdi Ioannis G. Kevrekidis Alexander Mitsos Yannic Vaupel Adrian Caspari. *ARTIFICIAL NEURAL NETWORKS FOR REAL-TIME MODEL PREDICTIVE CONTROL OF ORGANIC RANKINE CYCLES FOR WASTE HEAT RECOVERY*. English. URL: https://www.orc2019.com/online/proceedings/documents/76.pdf.