Università
degli Studi
di Padova

Dipartimento
di Ingegneria
dell'Informazione

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA MAGISTRALE IN CONTROL SYSTEM ENGINEERING

"TITOLO"

DESIGN OF A NEURAL STEERING CONTROLLER APPLIED TO CAR TRAJECTORY TRACKING

Relatore: Prof. / Dott: Augusto Ferrante

Laureando: Elia Burattin

ANNO ACCADEMICO 2022 – 2023

Data di laurea 03/04/2023

"Shooting for the stars when I couldn’t make a killing"

(Panic! At The Disco)

Ringraziamenti

Al termine di questo lungo percorso, trovo doveroso ringraziare tutte quelle persone che mi hanno aiutato a raggiungere questo obiettivo.

Voglio ringraziare i miei genitori Marco e Cinzia e mia sorella Asia per avermi supportato, a modo loro, anche nei momenti più complicati, i quali sono spesso diventati la quotidianità.

Voglio ringraziare Gloria, la compagna con la quale per la prima volta mi sono sentito a mio agio in un ambiente in cui mi sono spesso trovato fuori luogo.

Voglio ringraziare i miei amici "Vallongani" in particolar modo Simone, Nicola ed Alessia con la quale ho condiviso tutti i miei traguardi e le mie sconfitte trovando in loro persone sempre pronte ad ascoltarmi ed aiutarmi, ed Alberto diventato ormai il mio compagno d'avventura felice di assecondarmi in ogni proposta.

Voglio ringraziare Madrid e la mia famiglia Erasmus. Ci conosciamo da poco ma insieme abbiamo trascorso un anno pieno di emozioni, instaurando legami intensi come mai mi era capitato prima. Siete una delle cose più belle che la vita mi ha dato.

In conclusione, ci tengo a ringraziare in particolar modo me stesso. Per tutte quelle volte che mi sono sentito inadatto, sempre un passo indietro a tutto ed a tutti. Tanti giorni difficili sono passati nella quale mollare sarebbe sicuramente stata la strada più facile da seguire ma, invece, a causa della mia testardaggine mi trovo oggi qui, con questo pezzo di carta in mano, a provare al mondo che dopotutto, qualcosa di buono la so fare anch'io.

Nuovamente, grazie a tutti a voi.

Contents

List of Figures

List of Tables

Abstract

The car trajectory tracking is currently an investigated problem and numerous researchers are still working on it proposing different approches based on both iterative methods and deep learning strategies. Many solutions have been developed based on Model Predictive controller and Convolutional Neural Network supplying reliable results. In this work different models are designed to solve the regression problem related to the steering angle of the vehicle based on both lateral and angular errors with respect to the desired trajectory. The work has been developed using deep learning strategies as Feedforward Neural Network and its recurrent variations as Long-short Term memory network. The used database is developed by considering as the target variable the output of the MPC previously designed by the INVETT research group of University of Alcalà. The challenging hyper-parameter tuning is performed by using both automatic tools as Optuna and observations related to previous works due to the high request in terms of computational amount of time. The comparison between the different models is performed by leveraging mainly on the Root Mean Square Error in order to give a measurement of the reliability of the prediction also in the more challenging case. In the end, the obtained results will be discussed.

The programming language adopted in the entire project is $python$ and some specialized libraries as $keras$.

Chapter 1

Introduction

1.1 General overview and motivation of the project

During the last decades the automation systems have start to enter overwhelmingly in the everyday life affecting almost each aspect from the simplest to the more critical. A sector which has witnessed a great evolution in this sense is the automotive one. The introduction of new sensors mixed to the more recent control techniques supply to the new generation vehicles a higher level of both safety and comfort. The evolution of technology led researchers and manufacturers to investigate increasingly complex problems aiming to reduce in a relevant manner the human action. On this sense, an important topic which is still actually studied is the autonomous driving. It involves different areas as collision avoidance, velocity control and so on.

One relevant aspect which covers a main role in this topic is represented by the trajectory tracking. This main area involves the reduction of both the lateral and the angular displacement of the vehicle respect to the desired path and the control of the speed of the vehicle, in order to reach the desired point in the smallest possible time by considering the limitation imposed by the rules that regulate that specific street and the surrounding environment situation as congestion rather than traffic lights and many others. Several studies have already been executed and numerous techniques have been developed by using different control strategies that became feasible with the increasing of the computational capability. Starting from the most famous control strategies as **PID** which has been in investigated in [4] moving to the newer approach as **model predictive controller** (**MPC**) which has been subject of different researches as [2] and [3] where the authors have explored its application to the investigated task obtaining great results also in real-time environment. Others previous works as [5] have studied the problem from a different point of view choosing to explore the path outlined by the introduction of **artificial neural network** (**ANN**). In this sense, the researches focus on using convolutional neural networks (**CNN**). They exploit pictures of the street as input in order to estimate the required steering angle. This solution was considered also due to the decreasing of the prices of the components as cameras which are widely installed into all the new generation vehicles. The goal of this work coincides to design a neural steering controller by considering as starting point the previously projected MPC controller which has been developed by the INVETT research group of the University of Alcalà. This controller has already been tested also in a real-time environment supplying great results considering a simplified driving condition. The previously introduced network design is going to be based on a dataset obtained by the MPC itself. This idea is motivated by the will to try to obtain the same performances bypassing the higher computational cost required by the controller at each iteration to evaluate the optimal steering angle. In fact, considering a general neural network structure, it is well-known that a higher computational power is required limiting to the training procedure. So, in line with the theory, once the NN has been trained it will compute predictions requiring a pretty lower amount of it than the MPC allowing again a real-time implementation.

1.2 Summary

In the following, a brief introduction to how the thesis work has been developed through the different sections is provided:

- In the first chapter, an introduction to the MPC and the previous researches is supplied, focusing also on both the simulating and the real-time environment considered in test phase;
- In chapter 2 the different investigated structures will be described. The description of the tuning operation and the data processing will discussed in detail. In the end, the results will be discussed;
- In the last chapter, the conclusion and the future works will be presented.

1.3 Introduction to the MPC controller

In these last decades a powerful and sophisticated control technique has been developed, it is called **Model Predictive Controller** (**MPC**). This became rapidly one of the most used controller due to its ability to perform different tasks overcoming in terms of performance other really famous model as **PID** and **state controller**. Another aspect which encourages its widespread is related to its direct consideration of the involved constrains as mechanical ones which pose limits to the range of the output of the controller.
The MPC is a model-based controller, namely it works basing its prediction on the considered model which is chosen to describe the evolution of the different states that are involved in the studied system. In view of this fact, it is easy to state that the more accurate is the considered model, the better will be the performances of the controller due to its central role. In light of this, one of the most critical aspects is to develop an accurate and reliable model of the system that is going to be investigated. Going deeper into its functioning, for each iteration, it aims to evaluate N (where N is the number of steps

inside the considered prediction horizon) control actions. This evaluation is performed by exploiting the model in order to obtain the future behaviour of the system. These information are successively used to optimize a user defined cost function respect to the input quantities. In practice, an entire sequence of input is evaluated and in the end only the first one is applied. Afterwards both prediction and optimization process will be repeated. Due to this working principle, the MPC is also called **Receding Horizon Controller** (**ROC**).

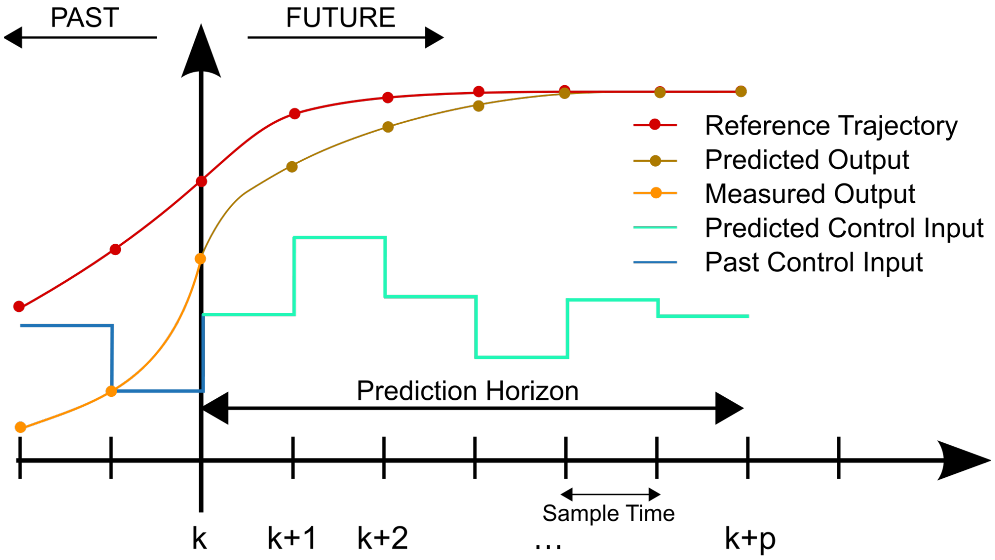


Figure 1.1: Description of functioning of the controller

1.4 Insight to the previous design controller

1.4.1 Practical assumptions

Given both the complexity of the work, some assumptions are going to be stated in order to simplify the problem. The developed MPC is constrained to work in a planar region, i.e. the control is constrained in the (x, y) plane assuming a constant altitude. This simplify the control action working only on two coordinates in the 3D-space. The velocity is assumed constant and with a low magnitude. This assumption is dictated by two main factors. The first one coincides with the control action that has to be achieved, in fact the focus is posed over the steering angle only. Considering a variable speed, two more control actions would be required, more precisely, would be necessary to introduce both a throttle and a brake control. The second constraint over the velocity is related to its magnitude as it was said before. This simplification allows to consider negligible the slip effect. In fact, assuming a dry road surface, a low velocity magnitude traduces into a small longitudinal slip. This makes negligible also the lateral slip which could rise during a curved trajectory.

1.4.2 Mathematical model

In order to define the required control structure, a mathematical model of the car is required. There are two different model that can be chosen namely the kinematic and dynamical one. The former is based on the position, the velocity and the heading neglecting completely the effects of the involved forces and in turn accelerations which represent instead the core of the dynamical model. In the previous studies, the research group opted for a kinematic model, more precisely, the choice fell on the **bicycle model**. The main advantages obtained by making this selection can be summarized in two relevant points:

- **Lower computational cost**. This characteristic is given by each kinematic system, indeed neglecting the dynamical effect, the computational cost reduces because of the lower amount of calculation required. A direct consequence is the higher facility to implement it in real-time task;
- **Higher simplicity**. The bicycle model is described in a simpler way than other model. This traduces into an easy handling.

On the other hand, this choice involves also a series of drawbacks and constraints as:

- **Lower accuracy**. Neglecting all the dynamics leads to a poorer description of the vehicle motion;

- **Motion constraints**. Exploiting this model, the velocity has to be constrained to be constant and small in magnitude simultaneously. These two constraints are direct consequence of the main goal for which the controller has been designed, in fact the controller has been developed to control the steering angle only entailing that no control action will be supplied to both brake and throttle impeding any change in term of magnitude of velocity.

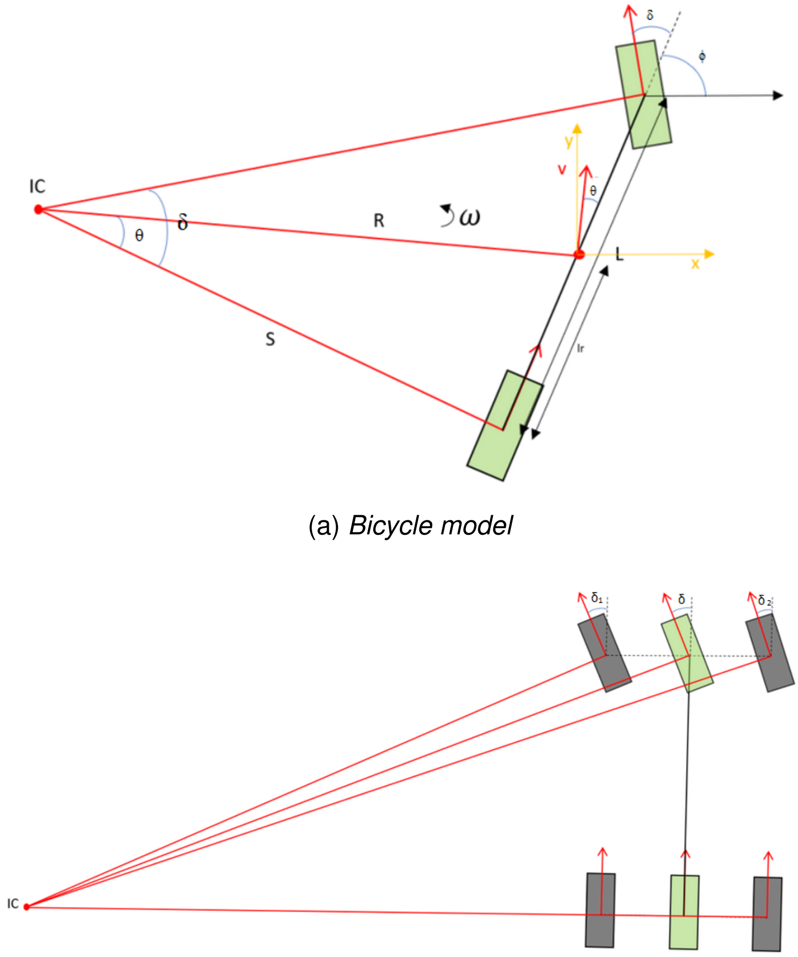
  The required small magnitude instead is required to reduce as much as possible the lateral slip.

In this work, the investigated vehicle is a car with four different wheels. It is possible to describe the automotive through the previously cited model without loss of generality. In this way, it is modeled as a bicycle, namely the forward wheels can be considered as an unique one posed in the middle point on the axis among them. Same reasoning is applied to the rear wheels.

This approximation is allowed because of the **Instantaneous Center of Rotation** (**IC**). Assuming that the car has a specific steering angle different from zero in a specific moment, it is possible to draw the orthogonal lines respect to all the instantaneous velocities in the different part of the automotive. Now, taking into account only 3 of them, namely the ones applied to the rear wheel, the center of mass and the steering wheel, the IC is defined as the point in which these three lines intersect. In the end, this model combines the steering angle of the two forward tires considering them as one. This implies that in both the cases, the vehicle is going to go revolve around the same IC but the introduced simplifications make easier the derivation of the model. The comparison among them is shown in Fig.1.2.

There are different ways to derive the required model. These depend on the considered reference frame i.e. the rear axle frame, the front axle and center of mass. In the following the last is going to be considered.

In order to evaluate the equations which describe the model, the configuration in Fig.1.2a will be considered. It is easy to see that the reference frame is



(a) *Bicycle model*

(b) *Four wheels model*

Figure 1.2: Comparison between bicycle model and real one

posed on the center of mass. In this situation, the equations related to the x and y coordinates are:

- $\dot{x}_w = v * cos(\phi + \theta)$
- $\dot{y}_w = v * sin(\phi + \theta)$

Now, the change in the orientation of the vehicle ϕ has to be evaluated. In order to do this, the radius of curvature R needs to be computed. This computation requires the evaluation of the S value which is equal to $S = \frac{L}{tan(\delta)}$. Afterwards, the R value is obtained: $R = \frac{S}{cos(\theta)} = \frac{L}{tan(\delta) * cos(\theta)}$. The final quantities to derive is the θ angle, which is easily obtained through the following formula: $\theta =$

$tan^{-1} * (l_r * \frac{tan(\delta)}{L})$. Assuming that $l_r = l_f$ and discretizing all the equations stated until now, the final model is obtained:

$$\begin{cases} \theta = tan^{-1}(\frac{l_f * tan(\delta)}{l_f + l_r}) \\ \phi = \frac{v * dt}{l_f + l_r} * cos(\theta) * tan(\delta) \\ \psi(t) = \phi + \psi(t-1) \\ x_w(t) = v * dt * cos(\psi + \phi) + x_w(t-1) \\ y_w(t) = v * dt * sin(\psi + \phi) + y_w(t-1) \end{cases} \tag{1.1}$$

where:

- δ is the tire angle of the wheels which coincides with the input signal of the system;
- θ which represents the angle between the angular velocity of the car and its longitudinal axis;
- ϕ which represents the change in the orientation of the vehicle;
- ψ is the heading of the car;
- x_w and y_w are the the Cartesian coordinate of center of mass of the car with respect to the world reference frame.

In this specific application, the control input and the output of the system are the tire angle and the cartesian position of the vehicle respectively. In Table1.1, the parameters which describe both real and simulated model are displayed.

Parameter	Simulated	Real
Rear Wheel Base l$_r[m]$	2.82	2.68
Forward Wheel Base l$_f[m]$	1.41	1
Tire max angle $\delta[rad]$	$\pm\frac{\pi}{3}$	$\pm\frac{\pi}{3}$

Table 1.1: Data used in the model

1.4.3 Description of the available controller

The available controller developed by the research group of the University is designed to solve a **multi-constrained optimization problem**. It needs a desired trajectory as input and it aims to minimize a cost function **J** which involves both the displacement between the center of mass of the vehicle and the closest trajectory point and the difference between the actual heading and the desired one:

$$J = \sum_{i=0}^{H_p} e^T * Q_e * e + \sum_{i=0}^{H_c} u^T * Q_u * u \tag{1.2}$$

where e is the vector of the errors, u is the vector of the input and the Qs matrices are the weights matrices for input and errors respectively.
The cited cost function is constrained not only by positional constrained but also by the mechanical and electrical parts of the vehicle which allows limited movements according with the manufacturer data. The prediction horizon is composed by $N = 25$ farther steps and a sequence of steering angles is computed at each iteration feeding the system with only its first element.

1.4.4 Required equipment for realized real-time test

The real test is performed using a commercial Citröen C4 with no low-level access to any of the actuators which is made available by the University. This one had to be equipped with different tools necessary in order to store data and in turn compare the real behaviour with the experimental one. The required instrumentation are composed by:

- a monitor and a PC in order to run the control action. Moreover it guarantees the possibility to run the real simulation and the virtual one simultaneously;
- a GPS module in order to collect data related to the actual position of the vehicle which are going to be compared with the considered reference;

- a battery in order to supply energy to all the electronic components.

- MPU 9255 which is the inertial mass unit(IMU) and a BMP180 altimeter;

- Control Area Network (CAN) bus to allows communicatio between the device inside the vehicle.

1.4.5 Description of the simulation

As already introduced, during the design phase, the controller has been tested in a simulating environment considering the assumptions stated in 1.4.1. In this environment, all the performances could be evaluated with a good approximation of the real world ones taking advantage of an user-designed map which recreates the neighbourhood of the University. The described map has been developed by the INVETT research group staff. The same neighborhood has been used in order to perform the real simulations once the controller supplied reliable results during the simulations.

In order to replicate the trajectory tracking task, a trajectory has to be used. Also in this case, it has been developed by the same research group. This one has been developed considering the CARLA simulator world reference frame which has been used to define the simulating environment. The simulator will be introduced in the next section. On the other hand, in the real world the exact reference frame coincides with the satellite one which supplies information which are described by means of **GPS** coordinates. In order to take into account this change of reference frame, the coordinates have to be adjust in order to match the real ones and no more the ones required for simulation. This adjustment coincides with a simple addition of an offset concerning both the axis namely x and y. Talking about the real simulation, the required trajectory has been stored considering the previously introduced geo-location system.

1.5 Simulating environment

1.5.1 CARLA simulator

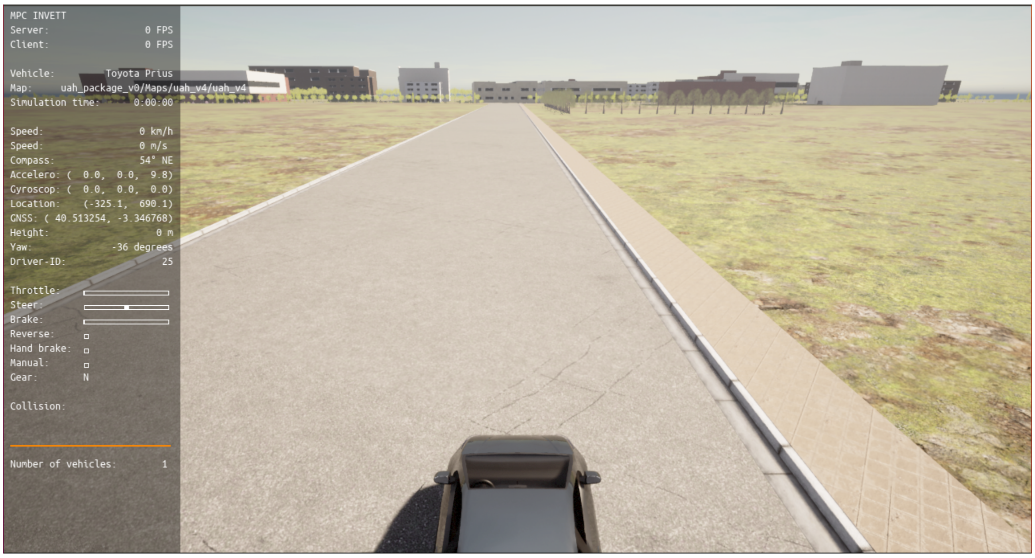
In this section, CARLA simulator has been introduced. It is an open-source autonomous driving simulator which leverages on a server/client system where the simulator clearly coincides with the server. Through its API, is possible to consider all the main components needed to obtain as realistic as possible environment, namely:

- Sensors. Vehicles rely on them to dispense information of their surroundings. In CARLA they are a specific kind of actor attached the vehicle and the data they receive can be retrieved and stored to ease the process. Actually different sensor could be represented as radar, cameras and so on;
- Traffic manager. A built-in system that takes control of the vehicles besides the one used for learning. It is used to recreate urban-like environments with realistic behaviours. In the performed simulation, the traffic manager is set in order to avoid presence of obstacles as person or other vehicle;
- Recorder. This feature is used to reenact a simulation step by step for every actor in the world;
- ROS bridge and Autoware implementation;
- Open assets. CARLA provides different maps for urban settings with control over weather conditions and a blueprint library with a wide set of actors to be used;
- Scenario runner. In order to ease the learning process for vehicles, CARLA provides a series of routes describing different situations to iterate on.

The use of the simulator allows to perform reliable and detailed simulations reproducing disparate environment from the more chaotic as the congest urban streets to the highway. The different defined classes with which is equipped make it a useful tool to replicate all the possible situation which can occurs on a real street.

It bases its own functioning on the definition of different objects called actors. In CARLA world an actor is defined as every objects involved in the simulation as vehicles, pedestrians, sensors and other kind of obstacles.

In this specific task, the simulation has taken into account only the main vehicle, the sensors attached on it and the fixed obstacles as sidewalks, roads signs and the buildings. No moving obstacles as pedestrian and others vehicles have been considered. The main goal in fact was related to the trajectory tracking task, so unnecessary sources of troubles have been avoided. They will be taken in consideration in future researches.



```
MPC INVETT
Server:                 0 FPS
Client:                 0 FPS

Vehicle:          Toyota Prius
Map:     uah_package_v0/Maps/uah_v4/uah_v4
Simulation time:      0:00:00

Speed:                 0 km/h
Speed:                 0 m/s
Compass:               54° NE
Accelero: (  0.0,  0.0,  9.8)
Gyroscop: (  0.0,  0.0,  0.0)
Location:    (-325.1,  690.1)
GNSS: ( 40.513254, -3.346768)
Height:                   0 m
Yaw:              -36 degrees
Driver-ID:                 25

Throttle:
Steer:
Brake:
Reverse:    □
Hand brake: □
Manual:     □
Gear:       N

Collision:

Number of vehicles:         1
```

Figure 1.3: Display of CARLA simulator

Shared memory system implementation

The shared memory system is a specific configuration which allows two or more different scripts to exchange information among them.

The system has been implemented using the **sysv_ipc** library available in python. The main tools considered from this library are $ftok()$, $SharedMemory()$, $attach()$ and $detach()$. The first one is used to obtained an unique key based on two parameters namely the **path** and the **id**, where the former is the memory location in which the involved scripts share the required information and the latter indicates an identifier instead. The $SharedMemory()$ is an object constructor which defines the desired shared memory and allocates it. The last two functions are used to attach and detach the created SharedMemory object respectively. In this way, the defined memory location will be used to share information among the different codes.
In this work, there are only two communicating scripts which are going to be introduced more in detail in the next paragraph.

1.5.2 Program structure

In this paragraph the program structure has been introduced. As it was said, two different scripts have been developed:

- mpc_control_gateway.py. It is the one that directly communicate with the simulator, so, in this context it is the client. It is entrusted to create the **World** object which define the used map, to choose the climate conditions and furthermore it manages all the actors involved in the simulation.

  This script provides also a visual representation of the simulation by the definition of the **HUD** (Head-up display) object.

  In the end, it applies the control action obtained from $gateway\_mpc.py$ by using the shared memory system introduced before;

- gateway_mpc.py. This second program is the responsible of the evaluation of the control input and its delivery to the previously introduced program in order to apply it to the designated actor.

These two work in parallel by communicating between them by using the shared memories system. It is important to underline that the former is the one that

initialized the shared memory allocation and so, running the latter before their allocation causes the rise of an error associated to the absence of the allocated memory and the subsequent crash of the program.
Going deeper in the program structures, starting to the first one, as it was already said, inside it all the classes used to initialize and perform the simulation are introduced. More precisely:

- the shared memory are allocated to allow the exchange of data between the client and the server during all the simulation;
- the $World()$ class. It is the most important class, in fact it is used to define the simulating environment through its World object. Through this class, all the main aspects of the simulation can be managed as the definition of the actors or the detection of all possible collisions by using specialized sensors forcing the re-initialization of the simulation considering the defined initial conditions for example. In the end the information are made available to the users by displaying all the required information through the HUD object;
- the $HUD()$ class. The **Head-Up Display** (**HUD**) is a display also present in the new generation vehicle which supplies information about the road helping the driver to be focus on the street. In this specific case it supplies visual information related to the considered state vector and the evaluated errors;
- several classes in order to define all the involved sensors. They coincide with collision, line invasion, Gnss, IMU, radar and cameras;
- the $KeyboardControl()$. This class allows the user to interact with the simulation associating certain action with the keys of the keyboard.

This code is also entrusted to apply the control action. This operation is performed inside a "loop for ever" routine named $game\_loop()$. This one takes a set of data, which are inserted from console by the user, in order to initialize

the desired vehicle, its sensors and more in general the world. After this initialization part, inside this routine, the required state vector is obtained and subsequently shared. In the end, the control action is received and consequently applied.
In this case, the state vector is composed in the following way:

- x and y positions;
- heading angle;
- linear velocity;
- steering angle;
- angular velocity;
- acceleration;
- throttle;
- collision flag. This is a boolean used to report if a collision has happened.

To define the actors, $carla.Actor()$ constructor has to be used. The main actor of the simulation is the vehicle which has to be controlled by mean of steering angles. In order to manage the control action, a $carla.VehicleControl()$ is defined. This one is used to obtain the actual control input and also to apply the new one in the subsequent iteration using CARLA's $apply\_control$ method which performs this operation modifying the physical quantities as the throttle, the brake and the steering angle. As it was stressed until now, the control will be applied on the steering angle only. This means that both throttle and brake will remain fixed to zero during the entire simulation.
Moving now to the second involved program, it is entrusted to both develop and supply the evaluated control action. It requires some quantities which are stored within a defined state vector in order to evaluate the control action. It receives it by leveraging on the memory system. The data have to be processed

to obtain the best possible control input taking into account all the constraints inserted in the model. To perform this operation, a script which describes the iterative optimization process managed by the MPC has been developed. It will be discussed later. An important fact to consider lies in the data used for the prediction, in fact, as it was said before, a state vector is required. This one cannot be directly supplied to the controller as starting condition because of it represents the actual state. During the MPC's routine, the car is driving and so the state vector changes. This will lead to an optimal control action related to a past position. In order to prevent this false prediction, the future state conditions are evaluated using the bicycle model. In this way, the optimization is obtained basing on one step forward displacement of the vehicle.

Now, looking at 1.1 it is easy to observe that the a time displacement dt is required. It depends on the execution time of the iterative process of the MPC which is not equal for each iteration. The best strategy to cope with this problem is to fix dt to an arbitrary value. It is quite intuitive that a trade-off between the minimum control time and the maximum acceptable between two different iteration has to be find. After some trials, it was set to 50ms.

Chapter 2

Design of the neural controller

2.1 Neural controller

A neural controller is a control structure based on the implementation of a neural network. This one has to be trained, validated and tested in order to evaluate both its efficiency and reliability.

The scheme is the one reported in Fig.2.1, where $NN(s)$, $G(s)$ and $H(s)$ are the neural controller, the plant and the feedback gain respectively, $R(s)$, $\delta(s)$ and $[X(s), Y(s), \psi(s)]'$ are the input, the tire angle and the output. This last one is composed by the x and y position and the yaw angle. The z coordinate is omitted according to the practical assumptions. The structure follows a feedback fashion which aims to achieve the desired trajectory points.

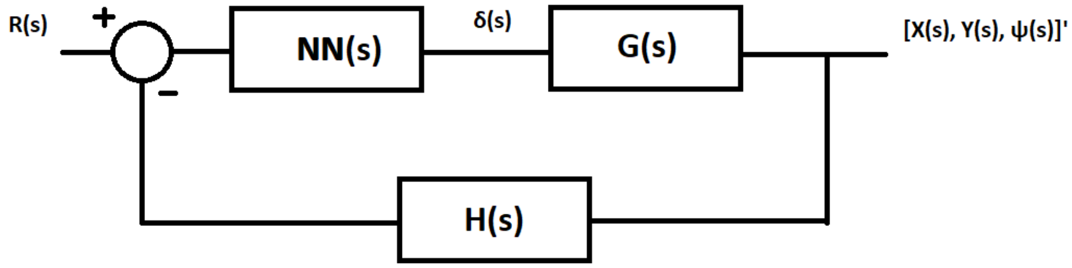


Figure 2.1: General system with neural controller

In general, the structure aims to obtain a function which describes the target variables by means of input ones. This function will be learnt by leveraging the so called training data which is defined as a set of samples picked randomly from the available database.

It is well known that an empirical method to design a generic neural network does not exist. This is due to its black-box structure which characterized this architecture. Considering this fact, the entire design will be performed by a trail and error approach.

2.1.1 Development of the database used for training

The design operation requires a database which supplies both training and validation data. This one has been supplied by the University. It is obtained by storing the data related to the MPC controller application. Talking about the test set, it is considered associated to the previously introduced path which describes the a neighbourhood of the University. This will be used in future in case of reliable results for real-time test considering the same equipment describes in previous chapter. This choice has been made to approximate as much as possible the real environment as it was made for the MPC design.

2.2 Multilayer perceptron

As first design strategy, the Multilayer perceptron (**MLP**) has been chosen. This kind of architecture is a special case of fully-connected Feedforward neural network. It is composed considering three different types of layers namely:

- **Input layer**. It is the layer which contains the input features. It is the only layer that does not own an activation function;
- **Output layer**. It supplies the output of the structure as the name suggests.
- **Hidden layer**. It is one of the hyper-parameter that need to be tuned. They are the core of the structure, indeed they have the main role to catch

the relation among the input features and the target. By increasing them, the complexity of the pattern that the structure can described increase too.

Talking about the last layer type, in general they are N, each of them contains with M neurons. The neurons are another important part of the structure. Each one is equipped with an activation function. This works following the structure reported in Fig.2.2. Every input is multiplied for its own weight and afterwards all the obtained results plus the bias component are summed. The result will be passed as input to the specific activation function which activates the specific unit depending on the obtained final value.

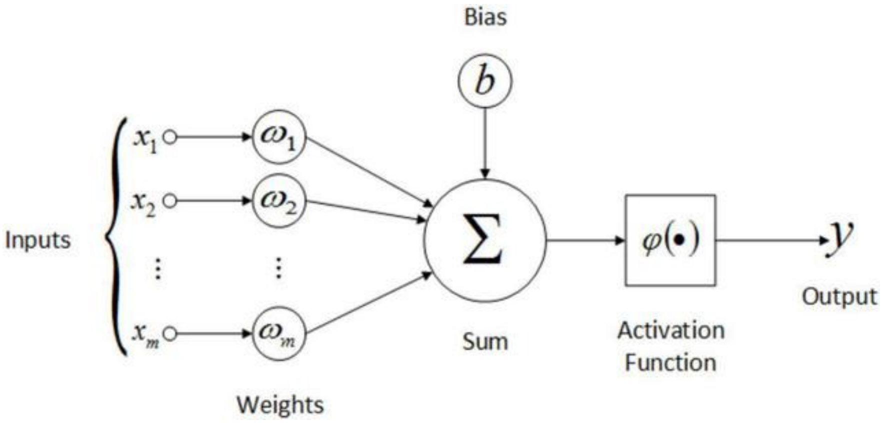


Figure 2.2: General structure of a neuron with activation function

There exist a lot of different activation function which are designed for different configurations. A brief description of some of them is given in 3. Coming back to the main discussion, both M and N have to be tuned as many others hyper-parameters which condition the behaviour of the net. Normally, this operation is the most challenging because of the required amount of time. This is a direct effect of the trial and error approach which is the only method to tune a general neural network. The needed time is strictly related to number of models that have to be trained in order to obtain a satisfying number of comparisons among different configurations. Some ways to tackle the cited drawback involve different measures which base on both hardware implementation and the use of specialized libraries. From the hardware point of view, the implementation of Graphics Processors Unit **GPU** can bring relevant improvements supplying higher computational power respect to the single CPU. A second good practice involves the use of specialized libraries available using **Python3**. These ones are going to be introduced farther.

Before to proceed with the training of the model, a validation has to be performed on it to guarantee its generalization capability. The validation is going to be evaluated considering the **cross-validation** technique. This strategy implies the split in two different groups, namely train and validation set. It is usually performed by using a more sophisticated version of this technique called **K-fold cross validation**. The employing of this specific algorithm demands to split the dataset in K folds where, at every iteration, $K-1$ are chosen among them as training set. The remaining one is used as validation set instead. When a generic iteration ends, in the subsequent one, the same split is performed but selecting a different fold as validation set causing the choice of a different training one as well. This procedure is repeated until each fold has been used to validate the model. In the end, the average of all the obtained losses is computed. This approach supplies a more reliable analysis on the performances of the model. This statement is straight consequence of the complete use of the available samples, indeed proceeding leveraging on this strategy, the possibility to choose a "lucky" training set is completely avoided ensuring more generalization capability.

2.2.1 Definition of predictors and output variables

As it has been said in the previous sections, a specific database is available. This one has been developed storing data related to a MPC controller applied to steering control. Basing on it, the choice of the input features has to be explored, in fact the output is already fixed as the steering angle $\delta(t)$. As first attempt, the choice fell on the following quantities:

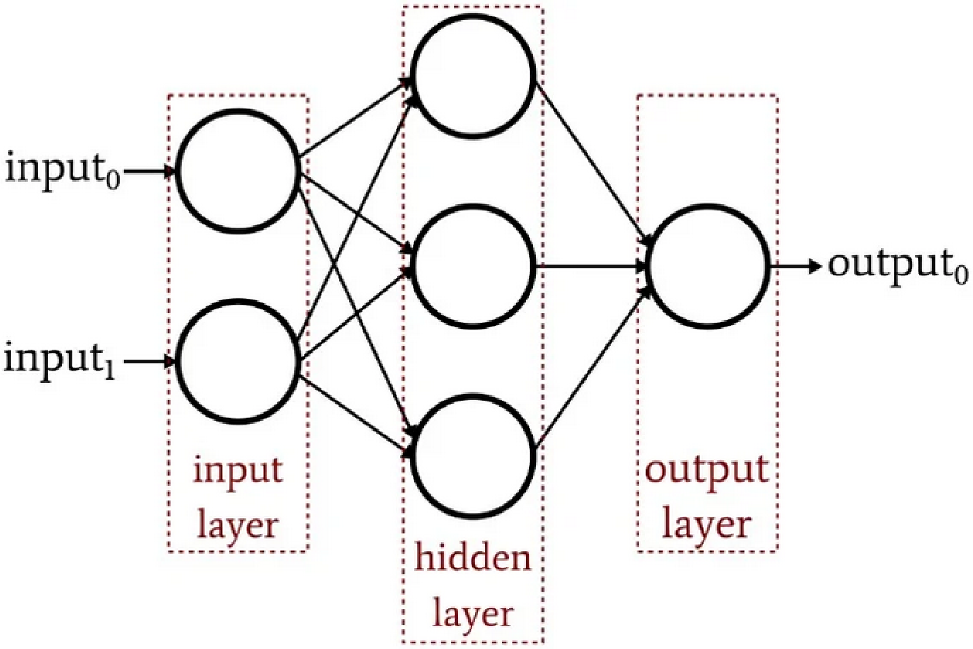


Figure 2.3: Multilayer perceptron

- lateral error (e_l), which is defined as the distance between the center of mass (**CoM**) of the vehicle and the closest trajectory point;
- angular error (Ψ), which is described as the angle between the sagittal plane of the car and the yaw angle of the closest trajectory point.

A graphical representation of the two introduced errors is supplied in Fig. 2.4. These two quantities have been selected focusing on two aspects that deeply characterized the investigated problem:

- ideally the CoM of the vehicle would lie on each trajectory point which compose the required path. This involves a lateral error as small as possible, ideally equal to zero, at every instant;
- in order to keep following the desired trajectory, the heading of the vehicle has to be instantaneously aligned with the trajectory itself, otherwise the car will move away from it. The required goal can be achieved reducing as much as possible Ψ.

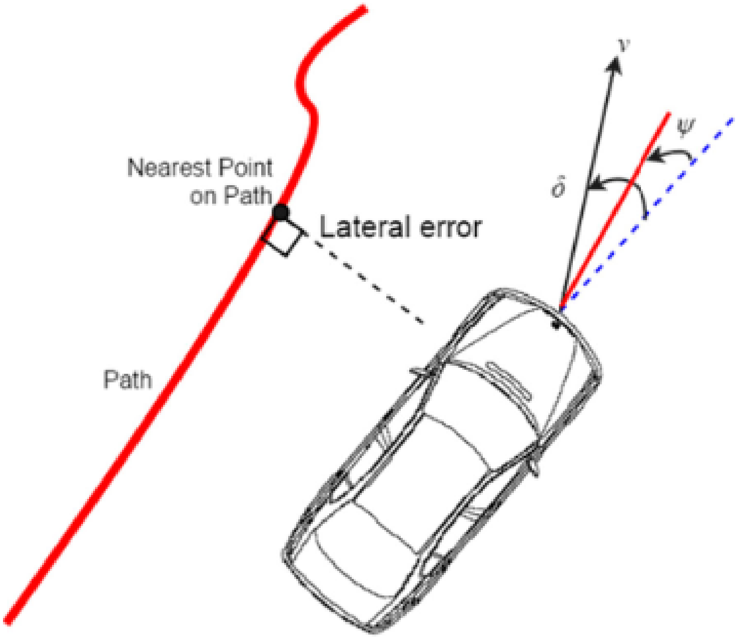


Figure 2.4: Lateral and angular error over a car model

A further consideration which suggests these two input has been extrapolated from the iterative process of the **MPC**. The controller, in fact performs the steering evaluation basing on it which aims to estimate a sequence of four angles which minimize the number of steps needed to enter in trajectory. In a nutshell, it will choose the values which reduce as fast as possible the two introduced errors considering the applied constraints.

2.2.2 Data pre-processing

Before to dive in the training procedure, some operations over the data are required. The first coincides to remove the duplicates. This is carried out using the built-in python module $pandas$, more precisely the $DataFrame$ class that allows to store the data in table-like structures and subsequently drop the duplicates by calling the method $drop\_dulicates()$ which removes the equal rows. In this way the redundancy is removed. This has been performed basing on the fact that equal samples would not introduce new information to the model. The second operation involves scaling the input in a standard range which normally coincides with [0, 1] or [-1, 1]. This operation is necessary to make all the features significant for the prediction. In fact different inputs could vary within different range and the effort of a specific variable could be underestimated due to its relative magnitude respect to the others. So, mapping all the features in the same range helps to avoid the rise of this problem. The scaling operation is easily performed using $sklearn.preprocessing.MinMaxScaler()$. This method allows to impose the scaling interval. As first attempt, [0, 1] has been considered. The described operation follows this formula:

$$x_{norm} = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{2.1}$$

where $x = [e_l, \Psi]$ depending on the quantity that is going to be normalized.

Design of the network

Once all the pre-processing operation are concluded, the database has to be split in the previously introduced sets. Accomplishing also this task, the focus can move on the design phase. Starting from the analysis of the trajectory tracking problem, it can be seen as a regression problem where each neuron output can be described as below:

$$Y = W * X + b \tag{2.2}$$

where:

- X is the **nx1** input vector where n is the number of features. As first instance it will be composed by e_l and Ψ only;
- Y output. It is a scalar quantity coinciding with the steering angle;
- W is the **1xn** weighting vector;
- b is the bias vector;

Considering the regression fashion of the problem, the best choice for the activation function in the output layer is the **linear** activation function. This one is described by the simple equation $g(x) = x$ and so it supplies the advantage of does not impose a restricted output interval. An additional motivation to choose it lies in the **backpropagation** algorithm which is used to update both the weights and the biases. It covers the most important part of the learning process coming in succession to **forward propagation** where, given a batch containing a certain number of samples, the predictions related to each of them are evaluated using 2.2. Once the prediction is obtained, a loss metric is computed comparing the prediction with the respective ground true. Basing on it, both weights and biases will be update in order to minimize that error. To achieve this goal, the partial derivatives respect to them of the loss function are evaluated. In the end, the update is performed by considering the inverse direction indicated by the obtained vector gradient and an other parameter called **momentum** used to avoid to get stack in a local minima. This operation is managed by an **optimizer**. The introduced linear activation shows a constant gradient which implies that it does not contribute to the update process limiting its utility to simply sum up the inputs of the output node supplying the final sum as result.

Once both the input and output structures have been identified, the focus has to move on weights, biases and their initialization which play a critical role during the learning procedure. As first aspect, is important to define which is the role of the first two during the involved phase. The weights have to evaluate the importance of a specific predictor respect to a specific neuron. From a geometrical point of view, considering for sake of simplicity ReLU as activation in a scalar case, the variation of the single weight converts into a change of slope. On the other hand, the bias parameter coincides with a vertical shift of it. The scalar example is shown in Fig2.5. Now, going deeper into the initialization topic, it needs to be performed carefully, otherwise some troubles could rise. For example, if the weights are initialized with too high values, the initials layers will probably learn more than the others leading to the **exploding gradient problem**. This is related to the excessive increasing in magnitude of the gradient which involves a big update of the weights at each iteration. In that kind of situation, the network could be unable to interpret the results due to the limited computational power of the device supplying NaN as output. On the other hand, if they are initialized with a too small values the opposite problem could appear, namely the **vanishing gradient**. In this opposite case, the gradient would become smaller and smaller performing the backward propagation process. This involves only a slight change in the weights update or in the worst case its totally absence. In the end, both the problems have to be avoided to guarantee the developing of a reliable controller.

There are different possibilities to perform the initialization. The choice is strictly dependent on the used activation function as is described in the following:

- the **Glorot initialization** which selects the initials guesses using a normal distribution $X \sim \mathcal{N}(0, \frac{2}{fan_{in}+fan_{out}})$ or an uniform one $X \sim \mathcal{U}(0, \frac{2}{fan_{in}+fan_{out}})$, where fan_{in} and fan_{out} are defined as the input to one layer and the number of neurons in that layer respectively. This method is suggested if the structure is equipped with $sigmoid$ or $tanh$ as activation function;

- the **He initialization**. This one follows the previous idea but in this case the normal distribution is $X \sim \mathcal{N}(0, \frac{2}{fan_{in}})$ and the uniform $X \sim \mathcal{U}(0, \frac{2}{fan_{in}})$. This modification is applied in order to cope with the non-differentiation which characterize ReLU around zero. This peculiarity makes this initialization really suitable for models which use ReLU-based activations;

- the **LeCun**. This initialization method aims to obtain the same distribution around all the net. It can be proved that going deeper in the net the variance could decrease if the weights are initialized with a too small guess or remain stacked if they are initialized to high. This modifies the variance between each output of the layers. Then, the LeCun formulation gives the solution to this problem normalizing it by using $X \sim \mathcal{N}(0, \frac{1}{fan_{in}})$

Studies related to the effectiveness of the different initialization was observed and discuss in [12].

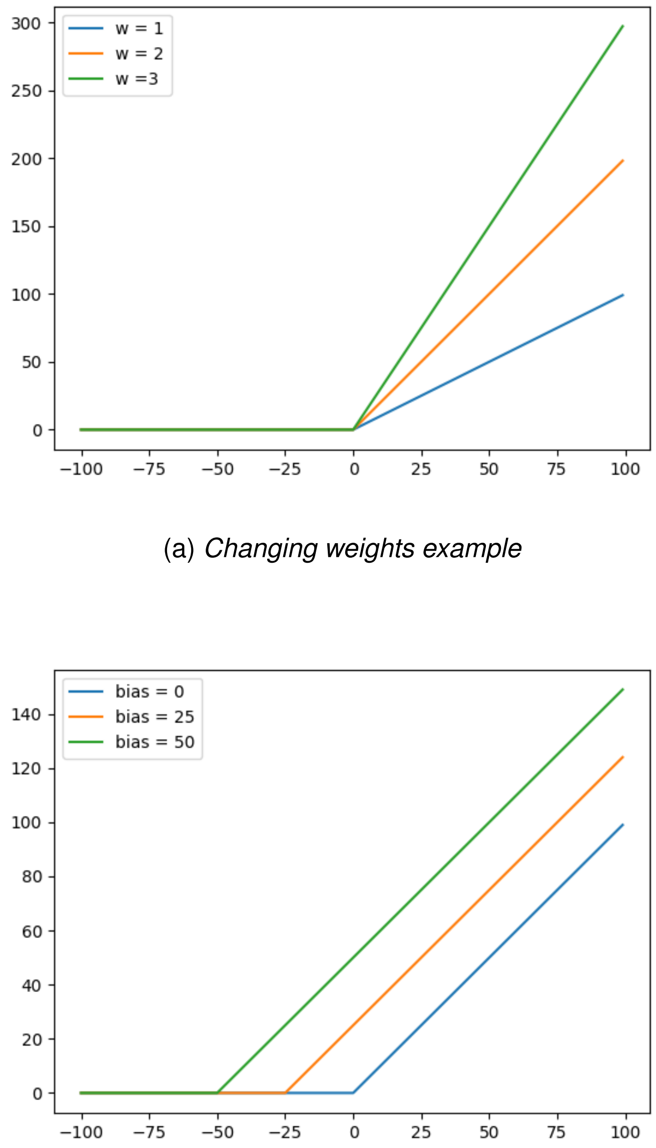
Now, concentrating on bias component, also this one has to be initialize in a proper way. In general, each component is set to 0 and, as for the weights, tuned during the training process. In literature some authors advise to slightly change this value fixing it to a small positive number. With this approach, all the nodes will be activated in the first iteration. Anyway, from previous studies it seam that this variation does not supply relevant improvements enforcing the idea to initialize them to zero.

Concluding the discussion over the initialization procedure, others important topics to investigate coincide with both the loss function and the optimizer. Coping with the regression problem, the **accuracy** cannot be evaluated because of it is strictly related to the classification problem, then the choice of the former fell on the"**mean squared error**" (**MSE**). It describes an error, then the final goal of the structure will be minimize that quantity. It is defined by the following formula:

$$MSE = \frac{\sum_i^N (y_i - \hat{y}_i)^2}{N} \tag{2.3}$$

where y_i is the i-th true value and $\hat{y}_i$ is the related prediction. Its strength lies on an unfair penalization, indeed following a parabolic description, the higher is the error, the heavier is the penalization. This will affect the update operation by applying a hefty one when a bad prediction is performed.

Talking about the optimizer instead, **Adam** has been chosen. It is a first guess based on the general good results obtained by its employing in different cases. The net is developed from scratch by using the $keras$ module which is directly supplied by python. This one is developed considering a **Sequential()** architecture which could be obtained easily using the cited python's library. It allows to insert layers in sequence as the name suggests. All the parameters have been set leveraging on the observations made until now. Anyway there are hyper-parameters as the number of hidden layers and the number of neurons



(a) *Changing weights example*

(b) *Changing bias example*

Figure 2.5: Scalar example of changing weights and bias in a ReLU function

that cannot be chosen apriori through analysis.

2.2.3 Hyper-parameter tuning

As it was already said, the hyper-parameter tuning is the most important aspect to consider and at the same time the most challenging. As first step, it is important to observe if some assumptions can be stated on them. This will help to reduce the number of configurations that have to be tried which strictly depends on the number of different hyper-parameters leading to a smaller amount of time required by the trial and error approach. A few assumptions have been already done in the previous paragraphs as the chosen activation functions, the optimizer and the cost function. Others can be done:

- using ReLU as activation, the He_initialization is taken into acount;
- in [7], the author advises to fix the Adam optimizer momentum parameter as $\beta_1 = 0.9$ and $\beta_2 = 0.999$;
- the number of epochs is fixed to 1000. This number can be fixed bigger basing on **Early stopping** function. This is a callable function that can truncate the training if there are not improvements after a certain number of epochs. This number is arbitrary selected.

In the remaining cases it is not possible to hypothesize a specific value without performing different trials. So, the remaining hyper-parameters need to be tuned, namely: number of layers, neurons per layer, batch size, learning rate of the optimizer and the dropout probability. The last one characterizes a special layer called **Dropout** used to apply regularization. It was introduced in [6] as a modern method to avoid overfitting in deep learning models. It works turning off some neurons of a specific layer with a certain probability p that has to be tuned. Using this strategy, the learning process becomes noisier, namely at each iteration different neurons turn off defining a different configuration helping into generalize. An important aspect to underline about this technique lies on difference between training and test. In fact, the dropout acts only on the training set. This variation is managed supplying a balancing term in the test case.

In order to tune that amount of parameters, a specialized python's library named $optuna$ was used. It has been recently developed and it allows to perform the required tuning in an almost automatic way. The structure of the script is organized in three steps:

1. development of a callable function which defines the model;
2. definition of an objective function;
3. optimization of the introduced objective function with respect of the hyper-parameters.

Starting from the first point, the neural network will be defined using $keras$ library then the required callable function has to be developed involving the $keras$'s functions. It has to be developed considering its dependency from the investigated parameters. This dependency can be defined by using $optuna.trial.Trial$ object which allows to fix a set of guesses which will be considered in order to initialize the different models. Theoretically there are infinite values which can be chosen, then some observations are required to reduce the variation range of each parameter. Considering both the number of layers and neurons, their range have been selected almost randomly trying to check if it is possible to obtain a simple structure which does not implies an enormous amount of training time.

Focusing on the batch size, in our case, the available dataset contains several steering angles described by an unbalanced distribution as it can be observed from 2.6. It is easy to observe that angles related to curved trajectory are lower in number respect to the straight one. In conclusion, a sufficient big batch size is needed to assure the presence of them in the batches.

The default learning rate in the $Adam$ case is fixed to 0.001, so considering a

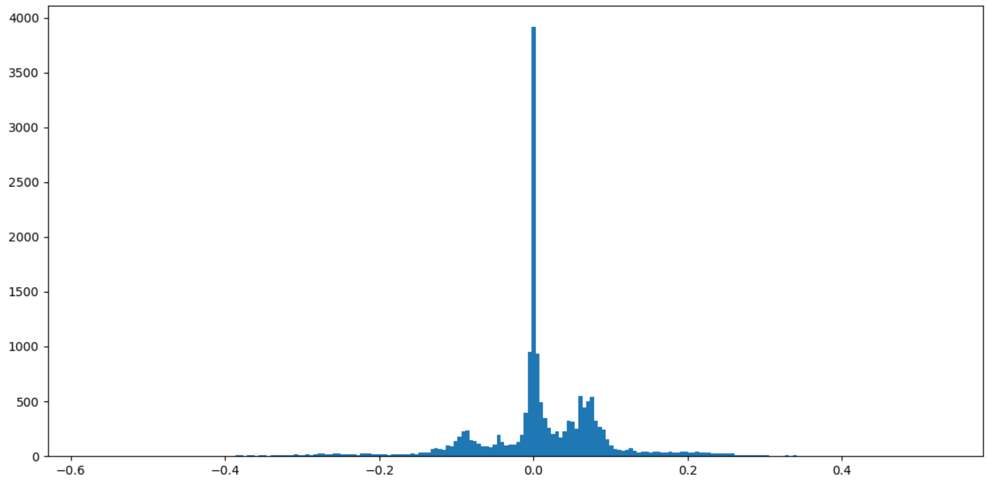


Figure 2.6: Density distribution of the target variable

surrounding floating interval can be a good initial guess. In this case a sampling step is needed.

The dropout rate is supposed to be inside $\{0.3, 0.4, 0.5\}$ set avoiding to impose a too high probability causing the introduction of a great quantity of noise

In the end the set of guesses is so defined:

$$\begin{cases} number\_nodes \in \{128, 256, 512, 1024\} \\ number\_layers \in \{1, 2, 3\} \\ learning\_rate \in [0.001, 0.1] \\ dropout\_rate \in \{0.3, 0.4, 0.5\} \\ batch\_size \in \{1024, 2048\} \end{cases} \tag{2.4}$$

Guesses for $batch\_size$ have been introduced but remembering that it is not involved in the model creation but in the fitting operation.

In order to define the both discrete sets and continuous intervals, $optuna$ supplies two operator: $suggested\_float()$ which allows to define a continuous floating interval and $suggested\_categorical()$ which allows to define the required sets by storing them in a list structure.

Both the hidden and the dropout layers have to be insert by taking into account the introduced guesses. The dropout layers are supposed to be insert only among the hidden layers avoiding to add it just after both the input and output ones.

Proceeding with the definition of the objective function, it will involves the error evaluated on the validation set. the best parameters will be the one the ones which lead to the minimum score by means of loss function. This operation is performed by using a **K-fold-cross validation**. The standard value are 5 and 10 but these implies to train K model for each trial, namely:

$$number\_of\_models\_to\_train = K * number\_of\_trails \tag{2.5}$$

How it can be stated the number of models to train is proportional to both number of folds and trials. A big number of trials allows to test a wider number of combinations. In light of this, to reduce the total required time, it was chosen to adopt a smaller value for K by assuming that the dataset is big enough to guarantee a good accuracy on the performances evaluation. In the end, the choice fell on $K = 5$. Then, for each model a specific metric averaged respect to the different folds will be evaluated. There are different metrics that are normally used to evaluate the performance of a model. The more used are in general the **Mean Absolut Error** (**MAE**) and the **Root Mean Squared Error** (**RMSE**). Both are interesting because of they supply a measurement of the error in the same unit measure that characterized the target value, in this case **radiants**. Giving a more detailed description, the former is described by the following equation:

$$MAE = \frac{\sum_i^N |y_i - \hat{y}_i|}{N} \tag{2.6}$$

where y_i is the i-th true value and $\hat{y}_i$ is the associated prediction. The main characteristic of this metric lies into weight in an equal way both small and big errors. On the other hand, **RMSE** is simply the square root of the **MSE** which formula is reported in 2.3. This leads into weight in an uneven manner the different errors penalizing more the higher ones. In the presented work the unbalance distribution which describes the target variable lead to prefer the RMSE because of, in this way, the bad prediction will contribute in a more relevant way to the error computation. In the end the objective function will be the mean of the validation **RMSE** over the 5 folds.

Last step coincides with the optimization. This one is performed considering the built-in function of optuna only. In fact it allows to define a so called $Study$ object through a specific constructor which requires two parameter called $solver$ and $pruner$ which are both used to speed up the tuning operation. This allows faster evaluations respect to the classical $GridSearchCV()$ method for example.

Looking at the more practical aspects, the define objective function is an error function and then it is quite intuitive aiming to minimize it. This aspect has to be specified inside the $optimize()$ method applied to the Study object. In the end, the set of parameters which will supply the lowest mean validation RMSE will be selected.

Going deeper in the fitting procedure, at the beginning of each trial, the initial weights of the first defined model are stored and afterwards they are set to every initialized new model at the start of each new fold iteration. This is done to avoid different starting weights within the same attempt in order to rely on the same initial conditions for each trained model inside it.

2.2.4 Starting point

In this section, all the aspects discuss until now have been used to perform a first attempt.

In order to start with the design procedure, as first step the database has to be uploaded and divided into input and target vector. After this operation, the input data have been scaled as specified in the previous paragraph. On the other hand, the output has been used without considering any kind of preprocessing. This choice follows from the same reasoning made for the input, indeed they are scaled to guarantee a balanced contribution of each component to the prediction. Now, the output is one-dimensional, consequently there is no relevant motivation to scale it.

Once the data have been re-arranged in the required shape, the output distribution shown in 2.6 has been observed again. Only this one has been investigated omitting an accurate analysis of the input ones for the following reason: the available database contains samples related to the MPC predictions. As it was highlighted in the first section, this controller has been tested supplying accurate angles estimation. In light of this, it is straightforward that the magnitude of the target will be significantly different from zero only when at least one between lateral or angular error is non-zero. So, studying the target distribution involves an implicit study of the input joint one. The indicated distribution is displayed using histogram which shows it by means of bins where the choice of their number represents the most important aspect to consider in this analysis. In fact, a too small number could lead to an under-estimated visual description inserting too many values in the same bin. On the other hand, a very large number involves the reverse problem namely leading to a very sparse histogram, so a trade-off has to be found. From literature there are different algorithms to perform the selection. In this case, the **Freedman-Diaconis' Rule** is considered. It leverages on **Interquartile Range** (**IQR**) which is so defined:

$$IQR = Q_3 - Q_1 \tag{2.7}$$

where Q_3 and Q_1 is the third and the first interquartile respectively. The final formula is so obtained:

$$bin_{size} = 2 * \frac{IQR}{\sqrt[3]{n}} \tag{2.8}$$

From 2.8, the number of bins results:

$$bin_{number} = \frac{X_{max} - X_{min}}{bin_{size}} \tag{2.9}$$

In the end, the resulting histogram coincides with the already shown in Fig2.6.

At first glance it is easy to note that the distribution is an unbalanced one showing the majority of the data around zero. This may be a warning to take in consideration for future analysis.

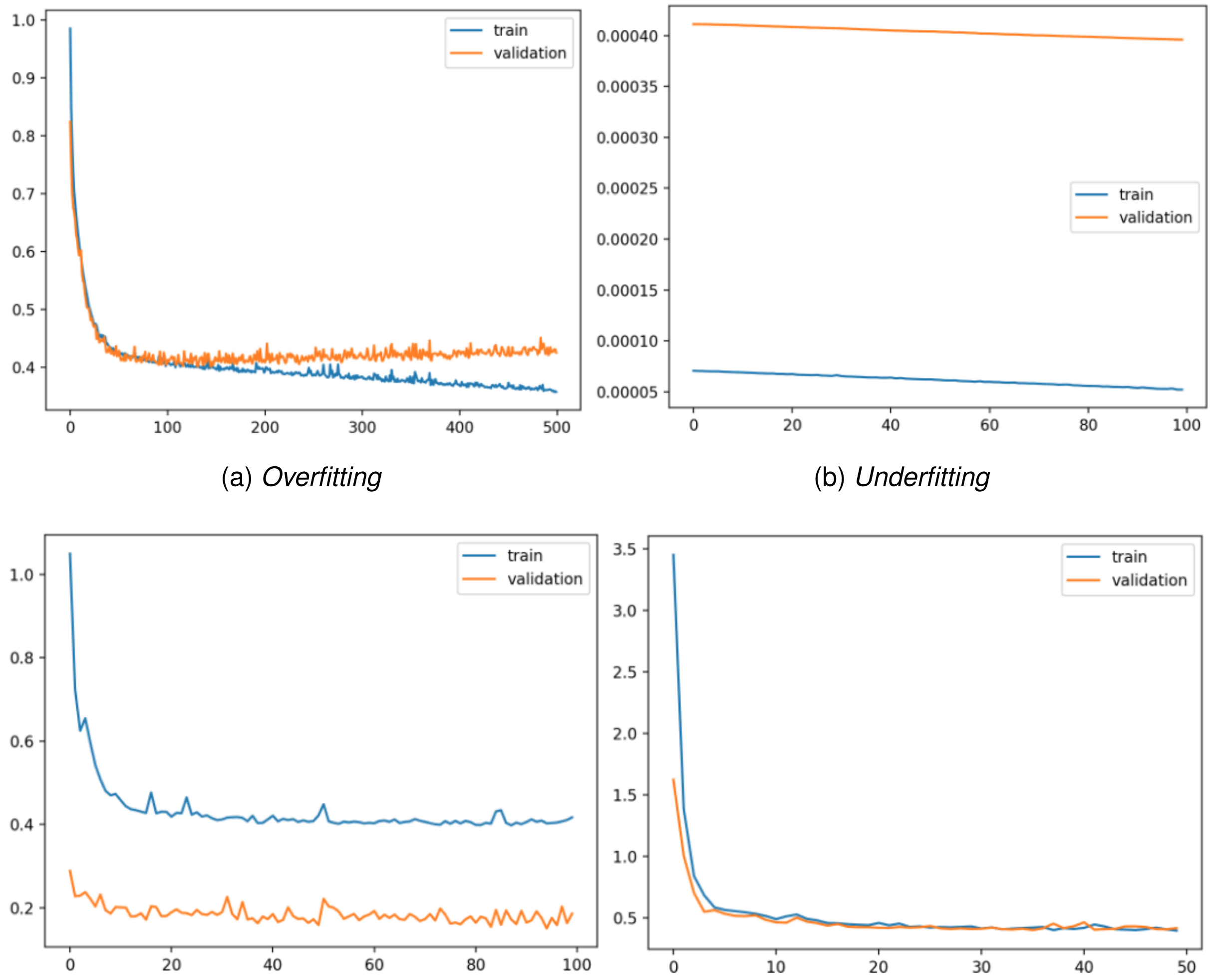
Proceeding now with the hyper-parameter tuning, it is performed as it was widely explained in the previous section.

After the parameters have been tuned and the best configuration has been set, the network has to be finally trained . Both training and validation set are required. The latter is used to check the evolution of the loss function over a set of unseen data. This allows to both manage the training by stopping it if no-improvements are observed for an arbitrary big time window and check if some relevant problems as overfitting or underfitting are risen. This can be performed by plotting the so called **learning curve** which allows a visual comparison between training and validation loss function evolution through the different epochs. Some examples are shown in Fig.2.7. They coincide with the more common cases. Considering the unbalanced nature of the distribution, a careful definition of the two sets has to be executed in order to make them similar. This avoids the possibility to fall into a "lucky" case which could lead to a non-representative validation set.

In the end, the respective learning curve has been checked (Fig2.8). It does not show critical aspects suggesting a good fit. The only aspect to discuss concern the lower magnitude of the validation loss respect the training one. This behaviour seems to be strictly related to the application of the dropout layer, in fact it is applied on the training set only, making more demanding the associated prediction. Anyway, this does not involve any type of trouble.

2.2.5 Sobol's sensitivity analysis

Sensitivity analysis (**SA**) is a powerful tool which allows to prove if a specific predictor is useful to describe the variation of the output. This analysis can be divide in two categories: **Local SA** and **Global SA**. Looking at the former, it bases on the study of the system response considering the effect of each



(a) *Overfitting*

(b) *Underfitting*

(c) *Validation set easier than training one*

(d) *Good fit*

Figure 2.7: Focus on different learning curves

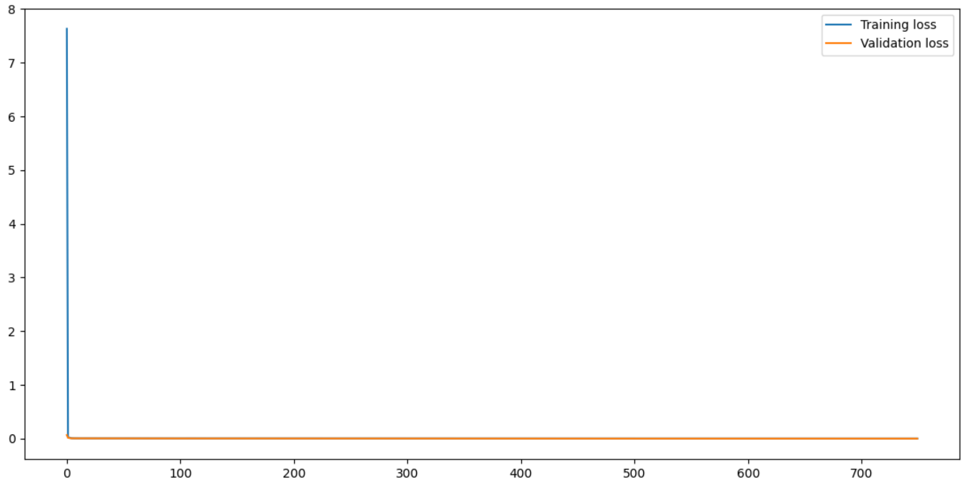


Figure 2.8: Learning curve considering only the two errors as features

single input only. The main drawback lies in the lack of information on the joint input contribution. On the other hand, the Global SA covers this aspect. In light of this,only the global fashion will be considered.

There are different ways to perform the previous introduced analysis, in the following Sobol's formulation will be presented and used on the designed models. It leverages on the output variance decomposition into summands of variances which has to be functions of the input parameters, namely considering the output variance $V(Y)$, where $Y = f(X_1, X_2, ..., X_D)$ assuming D number of features, it can be decomposed as function of the predictors in the following way:

$$V(Y) = \sum_i V(f(X_i)) + \sum_i \sum_{j>i} V(f(X_i, X_j)) + \cdots + V(f(X_1, X_2, \ldots, X_D) \quad (2.10)$$

Basing on 2.10, the different Sobol's indexes can be easily evaluated. By means of these, some metrics can be defined in order to supply information on the contribution of every single predictor and their joint interaction. They will be introduced farther. Coming back to the indexes evaluation and description, they are described by means of orders which depends on the number of considered interacting variables. For example, the first order index can be obtained as:

$$S_i = \frac{V_{X_i}(E_{X\sim i}(Y|X_i))}{V(Y)} \tag{2.11}$$

which describes the effect of the i-th niput feature on the variation of the total output variance. In the same way, the higher order indeces can be evaluated.

As it was said before, some metrics have to be evaluated. The more important are **Total sensitivity** and the **First-order sensitivity**. The latter coincides with the first order Sobol index, the former instead quantifies the importance of one variables considering both its single and joint contributions. It can be computed:

$$S_{T_{X_i}} = \frac{E_{X\sim i}(V_{X_i}(Y|X_{\sim i}))}{V(Y)} \tag{2.12}$$

It can be observed that the higher is the similarity between S_i and $S_{T_{X_i}}$, the lower is its interacting effect .

From a practical point of view, to compute the introduced metrics a sequence of samples is required. This sequence is created using the procedure defined by $Saltelli$ which is a Monte-Carlo based method. The computational cost is proportional to the number of samples that will be created. This number is arbitrary but a good trade-off between computational cost and a sufficient big number of samples is required in order to obtain both fast performances and reliable results. In the end, the defined sequence will be given as input to the trained model and the analysis will be performed supplying as final result all the required metrics. In this specific case, it coincides with Total sensitivity only. Talking about a possible threshold to choose if a specific predictor has to be considered or not, a rule of thumb is to consider as important each one which shows a total effect greater or equal than 0.05. Anyway, also the values under that score could bring some benefits providing a higher level of redundancy.

Keras implementation

The script has been coded basing on already existing python library $SALib$. It is a specialized library that is used to perform the required sensitivity analysis. Its execution is performed by defining a customize-function called $sensitivity\_analisys()$ which can be summarized in four steps:

1. definition of $problem$ structure. It requires three parameters namely the number of predictors, their names and their variation ranges. In this case, these ranges have been set according to the chosen scaling interval;

2. creation of the samples. They are created using the $sobol.saltelli$ method which defines a set of samples basing on Saltelli's formulation starting from the previous defined problem structure;

3. prediction. The investigated model is fed with the created samples and the related predictions are obtained;

4. performing the sensitivity analysis. Feeding $sobol.analyze()$ method with the problem structure and the predictions previously obtained by the model, both the total and the first sensitivity will be evaluated.

Test of the model

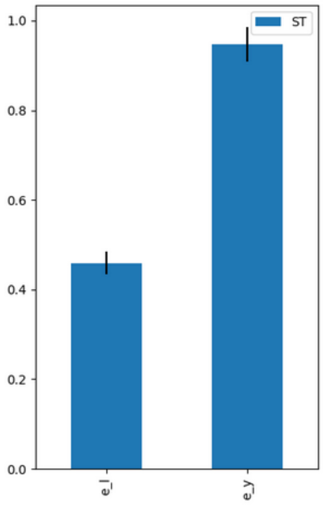
As last step, the controller is finally tested on the test data. Observing the sensitivity analysis shown in 2.9, it can be stated that both the predictors play a significant role in the estimation of the target variable. Looking at the results showed in Fig.2.10 a and b which give a visual representation of the prediction error and where it arise respectively, it is possible to observe a disequilibrium between straight trajectory related angles and the curved one in terms of prediction error suggesting a low generalizing capability regarding that specific pattern. This assumption is validated even further computing the RMSE related to the two different patterns of the road:

RMSE TRAINING	RMSE TEST	RMSE STRAIGHT	RMSE CURVED
0.0322045	0.0342640	0.0114904	0.1274342

Table 2.1: Result using ReLU with only two features

From the results in 2.1, it is straightforward to note this kind of underfitting, indeed the error is 10 times higher. This problem might be related to a low information rate supplied by the actual input predictors.

A further analysis involved the prediction vs true plot In Fig2.10 c. In case of a perfect prediction, all the blue points would lie on the red line with unitary slope. From the graph, the most critical working areas can be detected. It underlines that the structure do not seem able to estimate the entire required range of angles, remaining stacked in a smaller region which excludes the boundary values. This result strengthen the previous observation which stated that the input variables are not sufficient to describe the output one. A possible way to tackle this drawback could be supplied by adding more predictors which could help in to obtain its better description. The process which involves the extrapolation of new predictors is called **feature construction**.



(a) *Sensitivity analysis*

Figure 2.9: Sensitivity analysis

2.2.6 Features construction

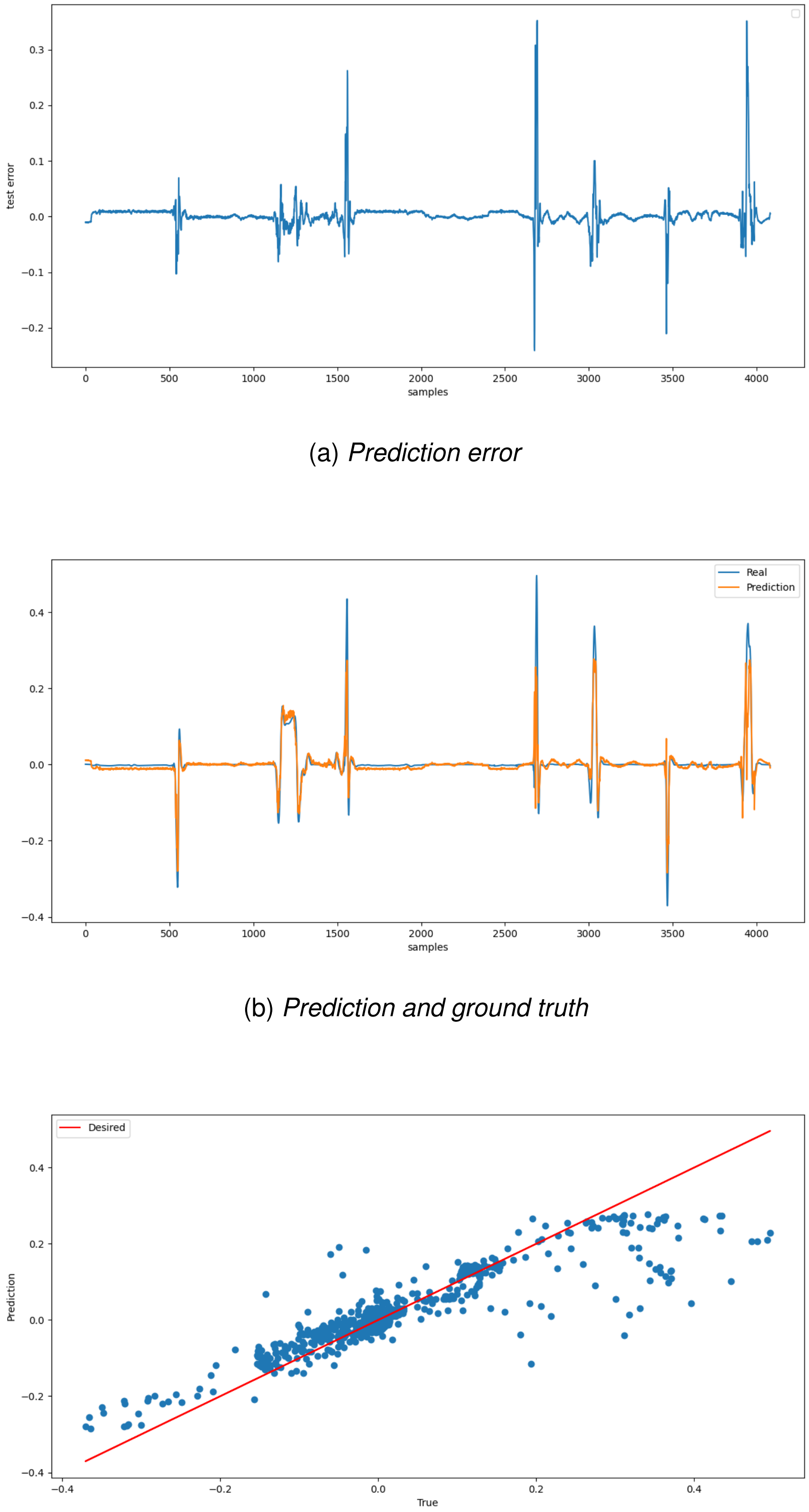
In this section features construction has been investigated. It belongs to feature engineering techniques and it involves the construction of new predictors basing on the already available ones. This might lead to perform better in term of prediction error guaranteeing a high generalizing capability using the additional information rate introduced by the new features.

The choice of these variables fell on the first derivative of the two errors. The intuition behind it bases on the effectiveness of the controller. Reasoning on a generic iteration and assuming that the vehicle is approaching a straight trajectory, at the steady-state the vehicle will be exactly on the desired point with both the errors almost equal to zero. When a change in the pattern of the street is encountered, namely a change of lane is required or it is simply entering a curve, an increment related to at least one error will be detected. In that situation, the error is changing in magnitude causing the increment of the magnitude of its derivative too. In light of this, these variations indicate that the steer need to be adjusted. Then the controller might be consider this new information in order to increase the accuracy in the evaluation of the control action. Considering the discrete equation of the errors:

$$e(k) = e(k-1) + \Delta e(k) * T_s \tag{2.13}$$

where e indicates the investigated error at the specific time, Δe its variation and T_s the sampling time. The 2.13 leverages on the **backward difference**. This approach has been selected because of in a real-time environment both the actual and the previous errors will be available adopting a buffer of memory which stores the past measurements. So it has been preferred respect to the forward version which requires an estimation of the future errors. Rearranging the previous equation, the derivative can be evaluated:

$$\Delta e(k) = \frac{e(k) - e(k-1)}{T_s} \tag{2.14}$$



(a) *Prediction error*

(b) *Prediction and ground truth*

(c) *Prediction vs ground truth*

Figure 2.10: Results over test data

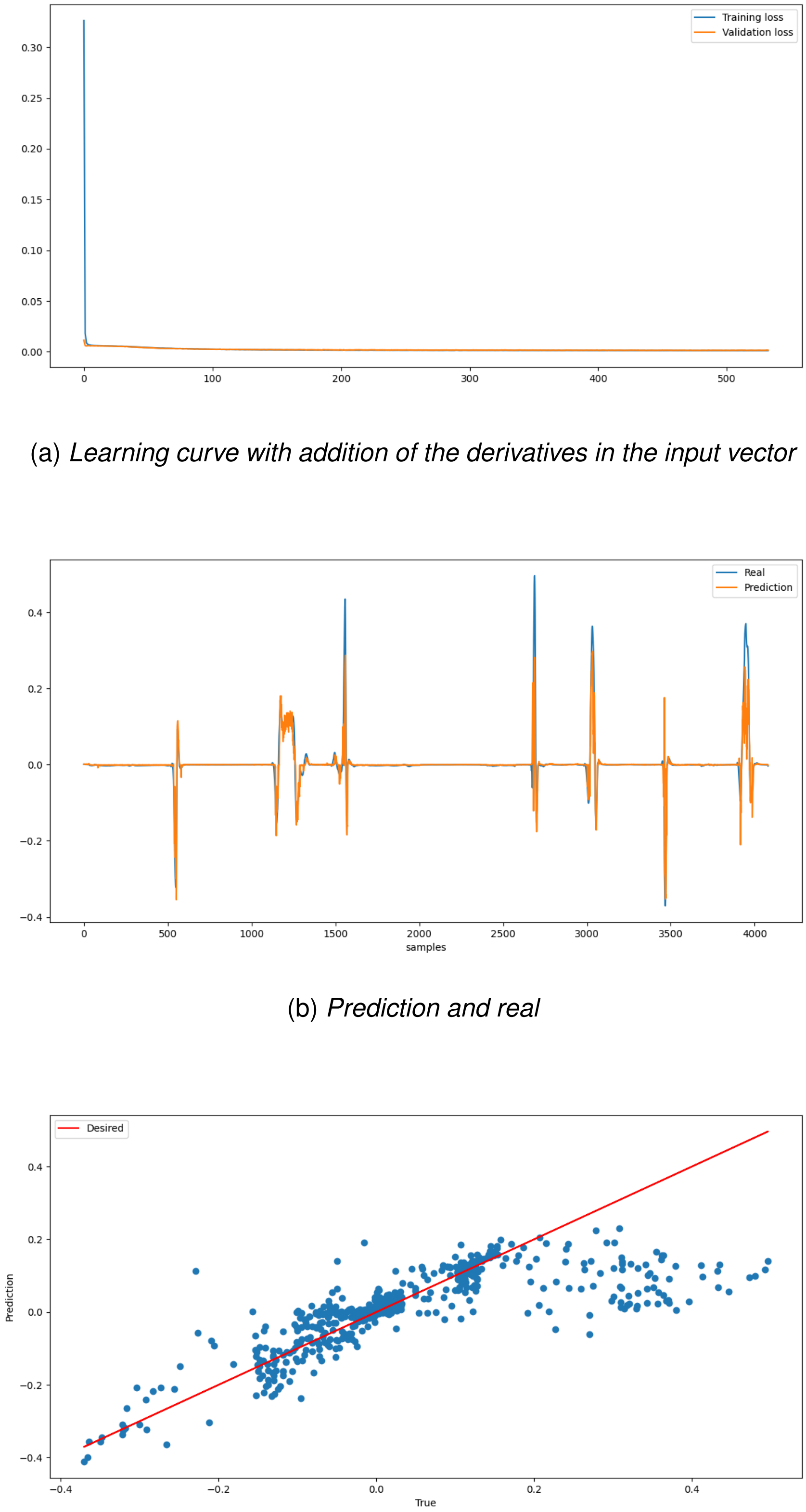
Basing on 2.14, both the derivatives are evaluated and afterwards the new input vector is obtained by stacking them by columns.
As in the previous case, the best structure has to be evaluated and so the hyper-parameters have to be tuned. The same procedure as before is used. The results are shown in Fig2.11 and Fig2.12. As before, the first investigated graph is the one coinciding with the learning curve. This one suggests also in this case that no problem has occurred as overfitting or underfitting. Looking instead at the sensitivity analysis, it seems that the new variables help into explain part of the output variance confirming its contribution in terms of extra information rate supplied to the structure. Taking into consideration the results shown in Fig2.11, the main problems can be underlined:

- Looking at the negative half plane, the entire range can be covered now showing the effectiveness of the applied features construction. Anyway, the predictions supply again poor results in term of prediction errors related to the more challenging pattern;

- Concentrating on the positive half plane, no improvement are observed.

As it was said, the positive angles shows the most relevant issues and so they have been investigated in order to find the root cause of this behaviour. A possible motivation can coincide with the scaling range in which the input is constrained. The intuition is supplied by the following consideration: assuming the case in which one of the two or both errors are positive, the computed steering angle will be negative in order to compensate them and so moving the vehicle closer to the desired trajectory. More in general, the control output has to contrast the rise of the errors. This observation lead to reconsider the scaling interval. In the end, it will be modified into [-1, 1]. In order to obtain this new mapping, the 2.1 need to be slightly modified in:

$$x_{norm} = 2 * \frac{x - x_{min}}{x_{max} - x_{min}} - 1 \tag{2.15}$$



(a) *Learning curve with addition of the derivatives in the input vector*

(b) *Prediction and real*

(c) *Prediction vs real*

Figure 2.11: Test results adding derivatives to input

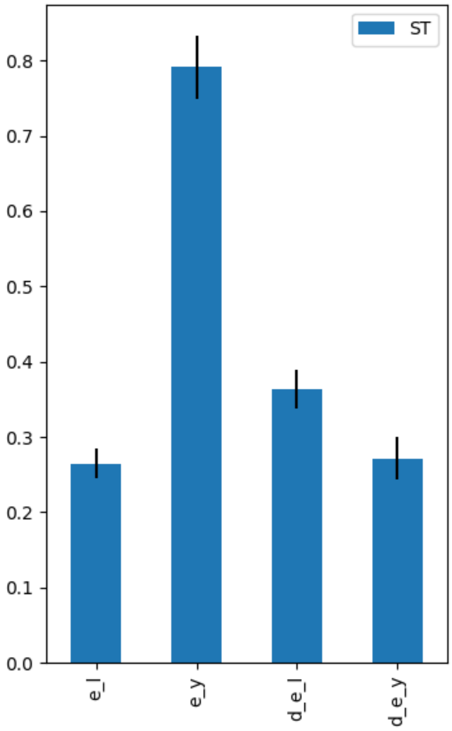


Figure 2.12: Sensitivity analysis considering the derivatives of the errors as input

Focusing now on improving the performances related to the curved trajectories, a possible way to cope with them bases on manipulating the database. Until now, different approaches have involved scaling the input values within different intervals and the addition of some new features. How it can be easy state, all these approaches do not involve particular focus on the statistics of samples within the database. As it was widely stressed, the target variable shows a really unbalanced distribution. Considering this, the next step will involve its more accurate analysis because of it is now suspected to be the main reason of those poor performances. The investigated histogram was previously plot and it coincides with Fig2.6. How it is easy to observe, the majority of the data assume values really close to zero. This could be the motivation behind a better estimation of the angles related to the straight trajectory. A first possible solution coincides with an **under-sampling** operation. It consists into remove samples within the more populated bins. They have been removed by performing the choice in a random way. Once this operation has been performed, the obtained distribution is checked (Fig2.13). Now, it seems to be more balanced than before even though a poor number of samples related to the boundary values are available. Anyway, a new tuning operation is performed, this one take into account all the considerations made until now in terms of predictors and scaling range.

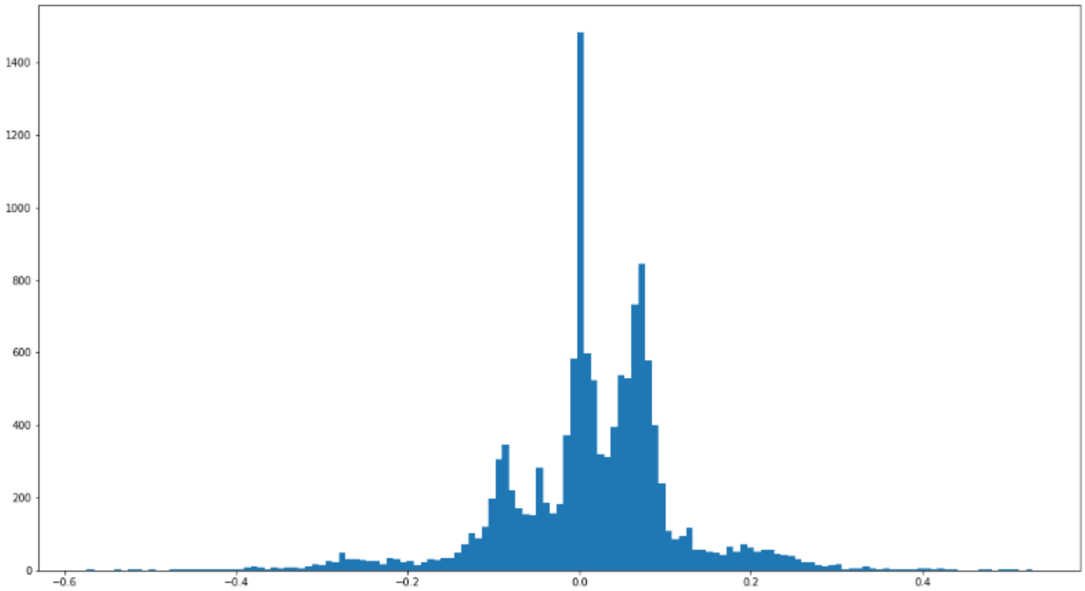


Figure 2.13: Distribution after under-sampling

Starting again with the tuning procedure and the subsequent testing operation, it can be observed is that the model is able to cover the complete target range but again showing the same problem the it was encountered so far. Others trials involved the change of the activation function of the model, more precisely adopting some variations of the ReLU. It was done in order to prevent the **dying ReLU** problem. This drawback derives by non-activation of the nodes, indeed in presence of negative values the final result associated to a specific node might be under the threshold. If this situation involves a great number of them, then the model would become too sparse making it not able to learn anymore leading to the previously introduced problem. In the specific application, there might be point in which both the errors and their derivative could be lower than zero, so the information supplied by that specific samples will be lose. During the years, the researchers have been developed a lot of new activation function in order to cope with this drawback. The most used are ReLU-based activations function which consist into slightly modifications constrained in the negative half plane. In the following, one of them is going to be tested namely, the **LeakyReLU**. Another trial is going to be performed considering a recently developed function, i.e. the **Swish** which respect to previous cited, involves the sigmoid function. The results are shown in Table2.2.

ACTIVATION	RMSE TRAINING	RMSE TEST	RMSE STRAIGHT	RMSE CURVED
ReLU	0.0254973	0.0255928	0.0118308	0.0900076
LReLU	0.0256583	0.0255710	0.0131886	0.0871378
Swish	0.0245721	0.0258981	0.0148386	0.0847619

Table 2.2: Comparison in performance between ReLU and its variations

It has to be highlighted that both LeakyReLU α parameter and β parameter which describes the swish function have been set to 0.1 and 2.5 respectively by using the usual trial and error approach. Also the SELU function has been tested but it was excluded from this presentation due to the bad obtained results.

From the results reported in 2.2, training and test loss show quiet close values confirming a good fitting. Basing on this, it can be stated that no overfitting has occurred. An other important result concerns the improvements in term of prediction error showed by all the models regardless of the used activation function confirming the previous intuition which assumed that the main cause of the problem was the disequilibrium in the target distribution. An interesting behaviour can be observed comparing the ReLU and the swish results where the latter shows better performance on the curved path to spite of the straight one. This underlines an higher generalizing capability. Anyway, the error keeps high requiring the research of new strategies to cope with the drawback.

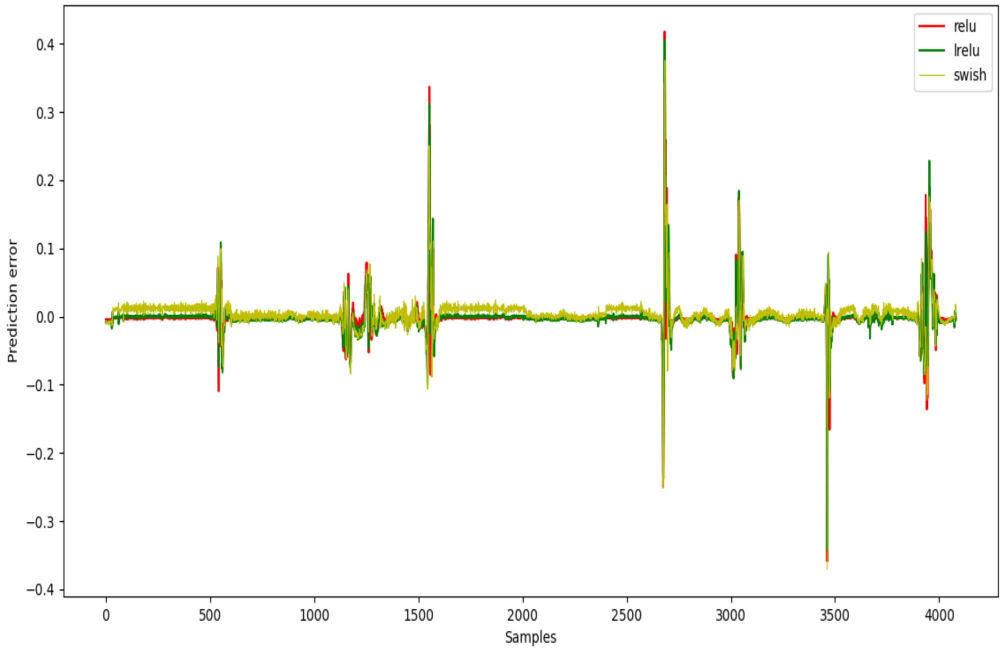


Figure 2.14: Comparison between errors obtained by ReLU, LeakyReLU and Swish

2.2.7 Weighted regression

The under-sampling operation has supplied some improvements in the target estimation concerning also the most problematic case. Anyway, they are not sufficient to guarantee a real time implementation. In light of this, a new way to tackle the issue has been introduced. Two different approaches can be investigated: **over-sampling** and **sampling weight**. Talking about the former, it consists into add artificial samples related to less frequent data. These ones can be obtained by simply adding multiple copies of random selected samples from the specific class. The latter instead works directly on the algorithm which define the loss function to spite of modify the database. This topic has been widely explore concerning the classification fashion but in the regression case there are not many rigorous researches. Two of them as [9] and [10] have been investigated. The substantial difference between the two fashions lies in the crossing lines among different classes. In fact, in the classification case these classes are well defined supplying clear bounds among them. In the regression case instead, the involved variables belong to a continuous interval. So, in this case the boundary lines have to be obtained by attempts. The choice for the different classes selection is obtained contemplating the results provided by the **Freedman-Diaconis' Rules**.

Once the different bins are defined, weights related to them need to be evaluated. This is going to be investigated in next paragraphs.

Going deeper into required algorithm analysis, the idea is to base on the standard MSE by applying a simple variation described below:

$$WMSE = \frac{\sum_{i \in \mathcal{B}} w_i * (y_i - \hat{y}_i)^2}{|\mathcal{B}|} \tag{2.16}$$

where $\mathcal{B}$ and $|\mathcal{B}|$ are the actual batch and its cardinality respectively. w_i instead represents the i-th weight related to the i-th steering angle and it depends on the bin in which the specific angle has been inserted.

In this way, each sample will provide a different contribute in weight update.

2.2.8 Kernel density estimation

In [9], an interesting strategy to tackle the problem is introduced. The authors highlight the main difference between the classification problem and the regression one supplying results based on two dataset as CIFAR-100 ([13]) and IMDB-WIKI ([14]), a discrete and a continuous one respectively, managing them in order to base the study on the same label distribution. In the former case seems that the empirical density probability represents a good quantity on which construct the required weights basing this statement on the high score obtained from the Pearson's correlation between the density of the label and the prediction error distribution. On the other hand, in the continuous space involved in the second database, the same metric does not provide a satisfying score. This dissimilarity seems caused by the following property: in the continuous space, a part of the information related to a generic label is supplied by its neighbours. So the required weights have to be evaluated considering this dependency. It can be taken in consideration by using the **Kernel Density Estimation** (**KDE**). It is a non-parametric density estimation technique which works by using the so called **kernel function**. It is a similarity function which behaviour depends on the chosen kernel type. According to the introduced paper, the choice has to fall on a symmetric kernel as gaussian or laplacian. In the investigated case, the former is going to be selected. It is described through the following equation:

$$K(y, y') = e^{-\frac{||y-y'||^2}{2*\sigma^2}} \tag{2.17}$$

where σ is the standard deviation. It is a hyper-parameter that need to be tuned.

From the 2.17, it is easy to state that the closer are two samples, the higher will be the respective output of the kernel function. The result can vary inside the interval [0, 1] where the boundary values can be reached if the considered samples are overlapped one on the other or if they are infinitely far respectively. The next step involves the convolution between the kernel and the label distribution, where the labels are described by means of bins. The authors called this operation **Label Distribution Smoothing**.

$$\tilde{p}(y') \triangleq \int_{\mathcal{Y}} K(y, y')n(y)dy \tag{2.18}$$

where $n(y)$ is the number of appearances of label y in the training set and $\tilde{p}(y')$ is the obtained density function for label y'.

As last step, the weights are obtained considering the reciprocal of the estimated density:

$$w(y) = \frac{\alpha}{\tilde{p}(y)} \quad \forall y \in \mathcal{Y}_{TRAIN} \tag{2.19}$$

where α is a hyper-parameter that has to be tuned.

The result of this computation is displayed in Fig2.15. Comparing it with Fig2.13 it is easy to observe that this last one represents a smoother version of the former.

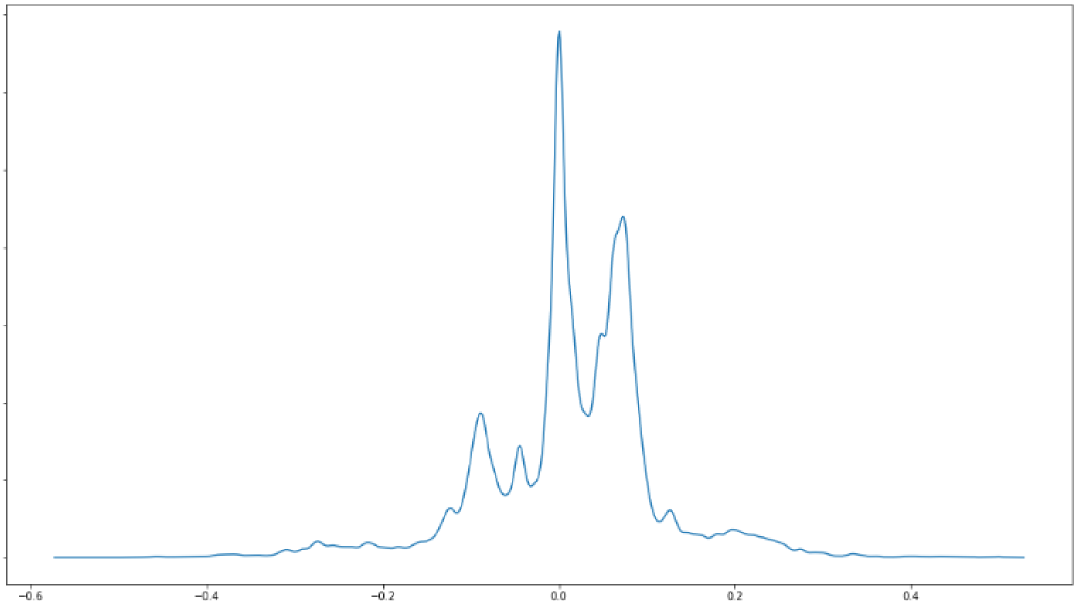


Figure 2.15: Estimated target density

Keras implementation

The main structure bases on the already introduced MLP, namely its keras implementation does not change. The only modification involves the loss function implementation. Keras model can be equipped with one among several standard loss functions but, in this case, the required one is not among them and so it is necessary to define a **custumize loss function**. In order to implement 2.19, it has to be defined to perform the following operations:

1. at each iteration, the required weight has to be searched among all the available ones. In order to speed up this process, an Nx2 matrix has been developed, where N is the number of training samples and the two columns are the sorted target data and the associated weights respectively. The sorting operation is performed by using **quick sort** as sorting algorithm which is the standard implemented by the $numpy$ library. This pre-process has been done in order to apply **binary search** as searching algorithm which is one of the fastest developed.

   So, at each iteration, the weights associated to the target value in the considered batch has to be found. In the end, a vector containing the required weights is obtained. This operation has to be performed for each batch iteration. An important practical fact to consider is the following: the data are described by **float64** data type but the standard used by keras is **float32**, so it is important to change in in order to avoid the rise of errors induced by casting operation. This can be done with $tf.keras.backend.set\_floatx('float64');$

2. to evaluate the weighted loss, the squared difference of every sample need to be computed an subsequently stack in a vector. Afterwards, the matrix product between it and the vector of the weights need to be evaluated. In the end, the final results is obtained by averaging it over the batch size;

3. In the end, the evaluated loss is returned in order to allow the backpropagation execution.

The loss function has to be defined in an understandable by the model way. This can be obtained by coding using $keras.backend$. Another important aspect to consider coincides with the right procedure to supply it to the model. In this specific case an extra parameter has to be considered, namely the previously introduced Nx2 matrix. To perform this operation, a wrapping function is required. More in detail, the keras models are developed in order to manage only the true and the prediction vector as parameters. So a way-around to introduce also an extra parameter consists into wrap this function inside another one which will consider the mentioned parameter. Once the model has been trained, it will be stored to make it available for future usage. In order to load it, the used customized cost function has to be explicitly communicated to the model in the loading routine. This operation has to be make by passing to keras' $load\_model()$ function a dictionary where the required loss is specified, namely: $custom\_objects = loss:'Weighted\_loss(weight\_list)'$, where Weighted_loss and weight_list are the customize loss function and the previously introduced matrix which connects target with relative weights respectively.

Focusing on the weights evaluation, as first step, the output samples have to be split in bins which have to be evaluated by considering **Freedman-Diaconis' Rule**. Afterwards, the kernel function has to be applied. It is obtained by using $sklearn$ library which defines different ones. As said before, the focus is posed to gaussian kernel also known as **radial basis function**. This can be defined using rbf_kernel method supplied by the cited library. It returns an NxN matrix where N is the number of different labels . In the end, the integral operation is performed by means of sums. Finally, the weights are obtained multiplying the inverse of the result of the last sum for a hyper-parameter α.

Results

In the following the obtained results have been shown and discussed. Actually only the test errors related to the two different patterns have been considered. From table2.3, it can be observed that the implementation of this weighting strategy supply some benefit in term of balancing between angles associate to different trajectory patterns. Focusing on the ReLU activation function, it supplies the worst results regardless of the α value. On the other hand, the LeakyReLU and the swish seem to make the controller able to achieve a better generalizing capability confirming the previous statement. Anyway, the obtained improvements are not so significant. The root cause of this behaviour could be the guessed the linear law 2.19 which defines the linear relation between the weights and α which involves a linear dependency also between α and the relative distances among the samples. In light of this, a different relation which enhances these relative distances between samples has been tried.

ACTIVATION	RMSE STRAIGHT	RMSE CURVED
	$\alpha = 10$	
ReLU	0.0144296	0.1023206
LReLU	0.0105826	0.0807136
Swish	0.0189600	0.0994699
	$\alpha = 50$	
ReLU	0.0101265	0.0987373
LReLU	0.0115377	0.0865663
Swish	0.0166121	0.0888863
	$\alpha = 100$	
ReLU	0.0105826	0.0968148
LReLU	0.0112007	0.0878945
Swish	0.0168775	0.0855665

Table 2.3: Results with different α values

2.2.9 Exponential weight

As it was said, the focus will move on a different weighting function which will highlight the difference in number of samples inside each bin. The idea coincides to apply weights described by an exponential law, namely:

$$w(y) = e^{-\alpha * p(y)} \quad \forall y \in \mathcal{Y}_{TRAIN} \tag{2.20}$$

where both α and $p(y)$ are positive quantities. The $p(y)$ in 2.20 has to be defined in order to describe the frequency of a specific label. Basing on this, two different candidates have been investigated namely:

- again $\tilde{p}$ evaluated in 2.18;
- the **empirical probability** of each label.

In both cases, the used quantity has been scaled between [0, 1]. In this way, problem as exploding gradient caused by an excessive increment in magnitude of the gradient will be avoided.

Empirical probability

the empirical probability expresses the probability of a generic label through the below equation:

$$p(y) = \frac{n(y)}{|\mathcal{Y}_{TRAIN}|} \quad \forall y \in \mathcal{Y}_{TRAIN} \tag{2.21}$$

where $n(y)$ indicates the number of samples described by y label within the training set.
In the end, the final weights have been obtained by normalizing 2.21 by using 2.1.

Results

In this section, the results obtained basing the definition of the errors on the exponential law introduced so far have been discussed . They are reported in Tables 2.4 and 2.5.
Also in this case the analysis has been restricted to the usual metrics applied to both the straight and the curved trajectory. Looking at 2.4 and 2.5, it can be observed that counterintuitively no improvement is revealed in comparison to the previous case rejecting the last formulated hypothesis.
In light of the experimental results obtained so far, the MLP seems unable to reach the required generalizing level performing poorly in the most critical cases regardless of the several employed strategies. Basing on this statement, in the next sections, a different structure as **Recurrent Neural Network** which has the ability to handle time correlated data. Both an insight on the motivations which lead to this change of structure and its complete description will be supplied starting from the next section.

2.3 Recurrent Neural Network

In this section a different strategy to cope with the regression problem is investigated. It exploits again a neural network structure, more precisely a recurrent

ACTIVATION	RMSE STRAIGHT	RMSE CURVED
$\alpha = 1$		
ReLU	0.0106763	0.0872975
LReLU	0.0113600	0.0858577
Swish	0.0150122	0.0945417
$\alpha = 10$		
ReLU	0.0108702	0.0915685
LReLU	0.0109896	0.0879226
Swish	0.0157582	0.0973542

Table 2.4: Results with different α values using exponential weight with KDE

ACTIVATION	RMSE STRAIGHT	RMSE CURVED
$\alpha = 1$		
ReLU	0.0110020	0.1107410
LReLU	0.0117412	0.0886954
Swish	0.0162116	0.0875768
$\alpha = 10$		
ReLU	0.0108616	0.1060140
LReLU	0.0110081	0.0836828
Swish	0.0144464	0.0946854

Table 2.5: Results with different α values using exponential weight with empirical probability

model. The main difference which characterized this one from the previously studied feed-forward configuration resides in the ability to manage the temporal relation between successive data. In specific cases as the sequential tasks, an important factor to consider is how the investigated quantity is influenced from its past values. This operation cannot be performed by a multi layer perceptron which is a memory-less structure. In order to overcome this problem, the recurrent neural networks have been developed.

The idea to move to this different strategy is given two main facts:

1. some important information are supplied by the evolution through time

of the steering angle. In light of this, knowing that the multiperceptron structure is not suited to handle these kind of features, the best solution requires to move to recurrent model;

2. observing carefully the results obtained so far, it has been stated that all the trained models show inability to learn the pattern which described the curved trajectory. Considering this observations, involving an entire sequence as input of the model might help it to cope with this drawback;

3. from the results obtained so far it can be stated that the required angles depend on the trajectory that the vehicle is following, more precisely, the idea is that the target variable is related to the pattern of the path that the vehicle is approaching. Basing on this intuition, it can be described by means of evolution of errors by using sequential data as new input. These sequences can be obtained by storing data during the driving.

2.3.1 Basic structure

The basic structure is the so called **Recurrent network**. As the previous investigated model, it is described by means of layers and units. Actually they are more sophisticated in order to manage the past information and more precisely the correlation between them and the actual one. A visual description is reported in Fig.2.16. Going deeper in the structure, the main parts are:

- X_t. It represents the input at time t;
- O_t. It is the output of the unit at time t;
- h_t an h_{t-1}. These are the hidden states at time t and t-1 respectively. They encode the time dependency from the previous step;
- $activation\ function$. It has the same purpose as the FFN.

Another important difference to highlight lies into the learning process. It is well-known that a feedforward network bases it on the **backpropagation** algorithm. In the actual case, it has to be transformed in order to handle the time

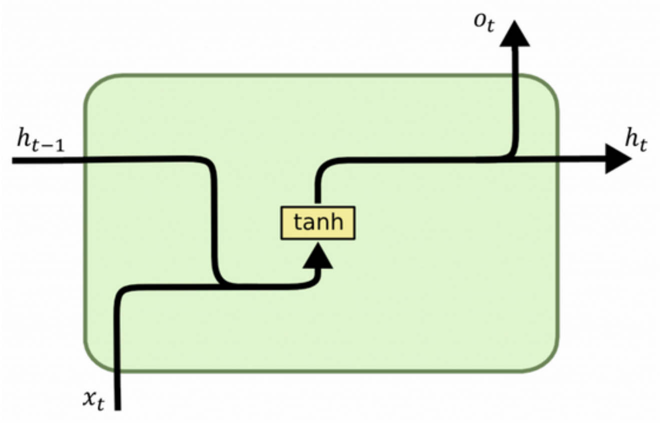


Figure 2.16: Unit of RNN

relation involved into the sequential data. This variation is called **Backpropagation Through Time** (**BPTT**). Its working principle is based on two step:

1. unroll of the network during the forward pass computing and accumulating the error through the timesteps;
2. Roll-up the network updating the weights.

these steps are repeated for each epoch. An important drawback associated to the introduced procedure is related to the high computational cost, indeed one update is performed after the computation of the error on the entire sequence. So, if this last is too long, the computational time will increase dramatically. A possible solution is supplied by using a variation of the standard algorithm named **Truncated Backpropagation Through Time** which exploits a limited number of time steps to perform the update.

This kind of unit is not suited to handle long sequences. Indeed, too long sequences as input mixed to the BPTT lead to a **vanishing gradient** problem. In order to address this limitation, a more sophisticated structure has been developed which is named **Long-Short Term Memory**.

2.3.2 Long-short Term Memory structure

As it was said in the previous paragraph, this new structure is introduced to solve the vanishing gradient problem which affects the standard recurrent network. The LSTM unit scheme is shown in Fig.2.17. How it can be observed, the new design presents 3 new components called gate. They are described in the following:

- **Forget gate**. It is equipped with a sigmoid function which take both the actual input and the previous hidden state as input. It supplies an output value between zero and one which represents how much has to be remembered from the previous state C_{t-1};
- **Input gate**. This is composed by two different stages. The first is a sigmoid function which takes the same input as the forget gate. It decides which value to update. The second stage is equipped with a tanh activation function which is fed as before. Its output is used to regulate the network, namely if add or not information to the previous cell state. In the end, the two are combined and sum to the cell state;
- **Output gate**. Its structure is quite similar to the input gate one, indeed it is equipped with both a sigmoid and a hyperbolic-tangent function. The former is used to choose which part of the cell state has to be taken into account and in which quantity to obtain the output. The latter instead is used to map C_t between $[-1, 1]$. By multiplying the output of both sigmoid and tanh, the hidden state at time t is obtained.

It is straightforward to state that the cell state at time t is obtained by combining the effects of the forget and the input gate with the previous state.
This structure allows to perform different kind of prediction basing on the input and output structures. They are show in Fig.2.18. In the investigated case, the required set up is the **many-to-one** which involves the knowledge of an entire sequence to estimate only one target variable.

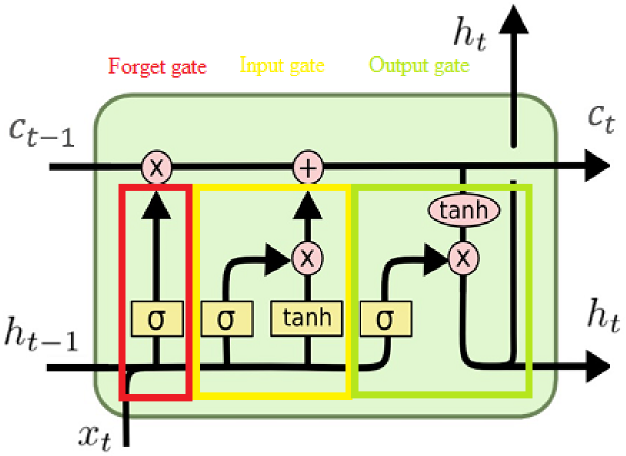


Figure 2.17: LSTM unit structure

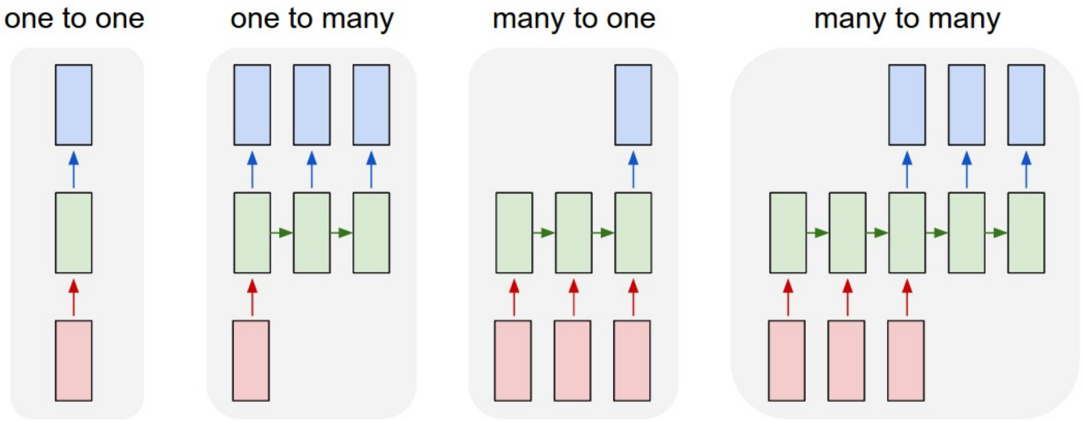


Figure 2.18: Prediction allowed by LSTM network

Keras implementation

Also in this case, the structure is implemented by using keras library. As in the MLP case, the library supplies the possibility to insert an LSTM layer by the following line of code: $tensorflow.keras.layers.LSTM()$. The layer requires some input parameters in order to be defined. In the following only the most important are going to be describe and analyzed to supply also some intuitions about the hyper-parameters to tune:

- **units**. As in the MLP case;
- **activation** and **recurrent_activation**. They are the activation function and the gate one respectively. According to the previous section they are set as hyperbolic-tangent and sigmoid;
- **kernel_initializer** and **bias_initializer**. They define the initialization of both weights and bias for each units. Basing on the observation made in the previous section, using tanh as activation function, the more suited weight initialization follows a glorot formulation. In this case keras use **glorot_uniform** as default. The bias instead will be initialize to zero;
- **dropout**. Used to regularize the output as ever;
- **return_sequences**. This parameter has to be taken into consideration when more than one LSTM layer is used in sequence. Each LSTM layer requires an input of the shape ($num\_timesteps$, $num\_features$). Setting it equal to $True$, the output of that specific layer will exactly match this shape.
- **stateful**. This parameter defines the way in which the training is performed. More precisely, setting it to $False$ the network will learn in a **memoryless** way. This means that after each batch execution and the respective update of the weights, the memory will be deleted. In a nutshell, the structure will learn from different episodes described by the

different sequences. On the other hand, setting it to $True$, the net will learn using a **memoryful** approach. In light of this, it is mandatory to consider a sorted set of sequences. A practical aspect to underline coincides with the manual erasing memory operation that has to be applied at the end of each epoch, indeed keras does not perform it automatically requiring an external for loop.

In the end, a $Dense$ layer with only one neuron equipped with a $linear$ activation function is insert in order to obtain a single output prediction.

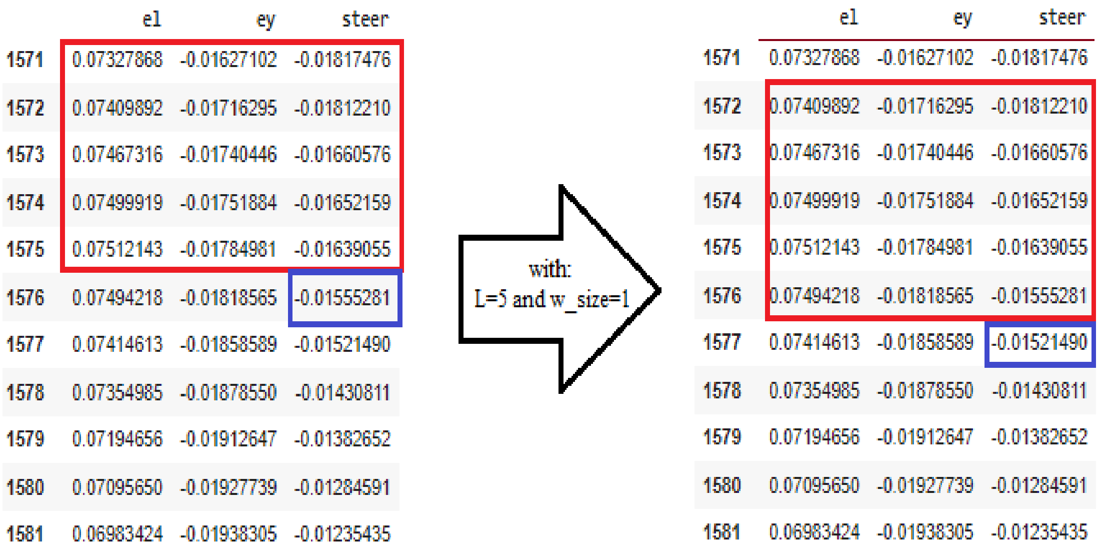
Dataset creation

As each neural network, also in this case an appropriete database is required in order to perform the supervised learning. As it was described, the recurrent models are designed to handle sequential data. Basing on this, the available dataset need to be modified. A first possibility coincides to split the data in sequences of a certain length and associate them with their relative single output. This strategy involves a relevant reduction in the number of samples, in fact its new number will be a fraction of the chosen length. For example, considering a length equal to 2, the number of available data will be halved. In order to cope with this issue, the so called **sliding window** approach has been used. This method involves the following steps:

1. The features have to be divided by columns;

2. A sequence length L is arbitrary selected and afterwards, starting from the beginning of each column, a number of data equal to it is picked creating a new sample;

3. Now, a shift of w_{size} is applied and others L values will be selected creating again a new sample. This operation has to be performed in order to explore the entire database.



	el	ey	steer
1571	0.07327868	-0.01627102	-0.01817476
1572	0.07409892	-0.01716295	-0.01812210
1573	0.07467316	-0.01740446	-0.01660576
1574	0.07499919	-0.01751884	-0.01652159
1575	0.07512143	-0.01784981	-0.01639055
1576	0.07494218	-0.01818565	-0.01555281
1577	0.07414613	-0.01858589	-0.01521490
1578	0.07354985	-0.01878550	-0.01430811
1579	0.07194656	-0.01912647	-0.01382652
1580	0.07095650	-0.01927739	-0.01284591
1581	0.06983424	-0.01938305	-0.01235435

	el	ey	steer
1571	0.07327868	-0.01627102	-0.01817476
1572	0.07409892	-0.01716295	-0.01812210
1573	0.07467316	-0.01740446	-0.01660576
1574	0.07499919	-0.01751884	-0.01652159
1575	0.07512143	-0.01784981	-0.01639055
1576	0.07494218	-0.01818565	-0.01555281
1577	0.07414613	-0.01858589	-0.01521490
1578	0.07354985	-0.01878550	-0.01430811
1579	0.07194656	-0.01912647	-0.01382652
1580	0.07095650	-0.01927739	-0.01284591
1581	0.06983424	-0.01938305	-0.01235435

Figure 2.19: Example of sliding window considering L=5 and w$_{size} = 1$

A clear explanation is supplied by Fig,2.19 where the red square represents the input sequence and the blue one the next steering angle to estimate.

As for the previous design, the input data requires to be scaled. The actual structure is equipped with a $tanh$ activation, so the data are going to be scaled within the interval [-1, 1] in order to match the activation function output range. Actually, also the output has been scaled in contraposition to the precedent analysis. This choice has been made taking into account that each prediction will be used as input in the successive iteration. In light of this, it is reasonable to map it in the same range of the input steering data. Once the estimation has been obtained, it has to be de-normalized to move back to its initial range in order to guarantee an understandable comparison with the MLP.

Posing the focus on the design phase, as usual the first operation aims to define the train, validation and test set. These are no created by shuffling the data because they have to be sorted to construct the required sequences. Once the split is finished, the data are scaled and in the end the sequences are created.

Talking about both L and w_{size}, they are two new hyper-parameters which need to be tuned. In order to avoid further complications, the latter is chosen apriori and fixed equal to 1.

Training and test

Also in this case, the model needs to be trained and tested. The training procedure follows the same framework used until now. The tuning of the hyperparameters have been performed again by using intuition, previous results and a trial and error approach. As it was underlined in the previous subsection, the activation function, the weights and the biases initialization have been already selected.

Considering the batch size, it has to be considered big enough to contain sequences associated to different trajectory patterns, then as first assumption the batch size will be fixed equal to 1024.

Moving on the number of epochs, a big number of them will be considered basing again on the **Early stopping** mechanism.

In the end, also in this case the focus will be placed mainly on both number of units and layers, the regularization and the learning rate. Focusing on the regularization, some initial trials suggest to avoid the implementation of the dropout layers due to the poor results obtained by using them. As consequence others kind of regularization have been explored as **lasso** and **ridge**. In general, keras' architectures implement it to the weights, biases or the output of the activation functions. In addition, a new type of regularization is usable as **recurrent regularization** which is exclusive for recurrent structure and it aims to regularize the hidden states of the cells. For all the introduced techniques, three different ways to apply them can be used namely:

- **L1 regularization**. It coincides with **Lasso** regularization;
- **L2 regularization**. In this case the **Ridge** regression is going to be considered;
- **L1L2 regularization**. This method involves the combination of the previous two in order to exploit both their advantages. It is named **Elastic**

Net.

The last technique requires the tuning of two hyper-parameters which indicate how much each specific regularization is applied.
Respect to the MLP design, an extra value is going to be considered, namely the length of the sequence L. It straightforward notice that it affects abruptly the training velocity because of the bigger is the sequence, the higher is the number of weights and biases to update after each epoch. In the following the results obtained with different L values have been shown.

SEQUENCE LENGTH	RMSE TRAINING	RMSE TEST	RMSE STRAIGHT	RMSE CURVED
20	0.0168200	0.0150113	0.0063409	0.0428815
30	0.0103110	0.0101135	0.0040736	0.0290985
40	0.0128902	0.0133436	0.0052961	0.0384431
50	0.01663450	0.0152300	0.0057616	0.0441532

Table 2.6: Comparison in performance between different sequence length

From the obtained results it can be observed that this method outperforms the previous one confirming the hypothesis that the evolution of the trajectory described by means of sequences involving both errors and steering angles supply a higher information rate which leads to a better estimation of the target variable. Looking more in detail the table 2.6, it seems that $L = 30$ supplies the best results in terms of RMSE involving also an improvements on the same metric applied to the two different investigated until now trajectory patterns. Anyway, the error in some points is not negligible and this involves the impossibility to test the controller in a real-time application requiring more investigation on the topic in order to achieve a more reliable model.

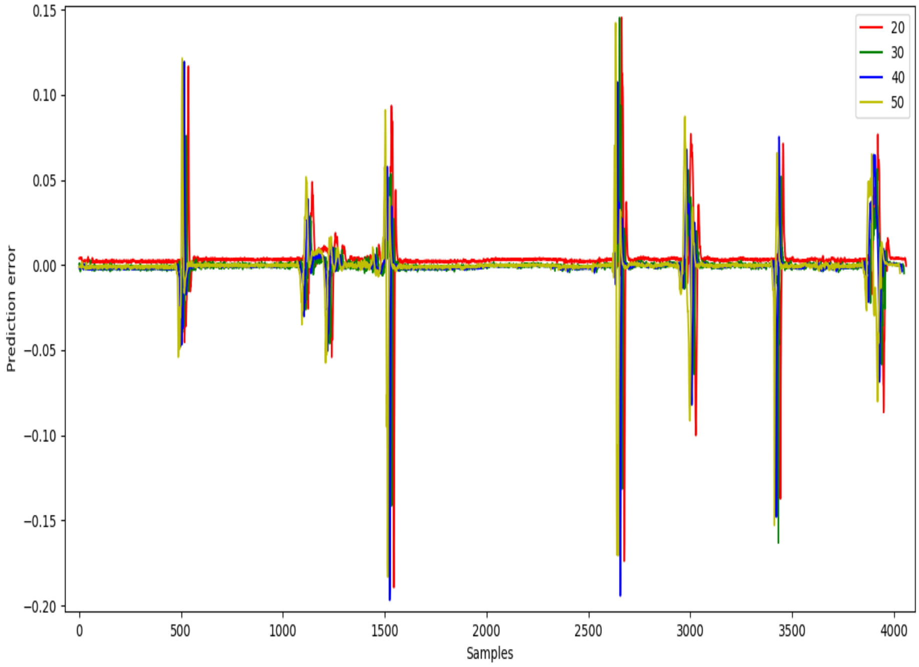


Figure 2.20: Comparison of errors considering different sequence length

Chapter 3

Conlcusion and future works

Through this work, different neural network structures have been investigated trying to emulate the behaviour of a previously design MPC controller in light of the great results shown by its implementation. During the entire work, the focus has been posed on two structures namely the multilayer perceptron and the recurrent neural network. Considering the MLP, several configurations have been trained and tested underlining the difficulty of each of them to achieve a sufficiently high generalization capability required to obtain reliable performances on different trajectory patterns. Different strategies based on features construction, manipulation of the database performing scaling and under-sampling operation and sampling weight have been implemented without obtaining relevant improvements in terms of RMSE. On the other hand, the recurrent architecture outperforms the previous structure leading to halving the prediction error shown in the worst case. This result highlights the correlation between the required steering angles and the trajectory followed by the car described by means of previous errors and past angles. In spite of this significant improvement, the actual performances do not allow its real-time implementation because of the non-negligible error mainly involved in the curved trajectories.

In light of this, the future works will focus on a deeper investigation of this structure by both considering the addition of new features and the implementation of variations of the LSTM unit as the **Gathered Recurrent Unit** (**GRU**) which bases on the same working principle but simplifying the gates structures in order to reduce the required computational cost. In addition, a different approach could involve the simultaneous application of both the MPC and this last controller using the former to cope with the most critical cases only in order to partially reach the initially proposed goal.

Appendix A - Activation functions

In this appendix different cost functions have been introduced. These play an important role in the working principle of each deep learning structure, indeed they are the ones which both manage the activation of each neuron and allow the update of weights and biases by allowing the application of the backpropagation algorithm during the training process.

ReLU

The first relevant function is the **Rectified Linear Unit**, simply known as **ReLU**. It is so defined:

$$ReLU(x) = \begin{cases} x, & if \quad x >= 0 \\ 0, & otherwise \end{cases} \tag{3.1}$$

A visual representation is shown in Fig.3.1 This is one of the most used activations due to a several number of advantages that it has introduced, for example:

- it is computationally cheap;
- it avoids the problems related to both saturation and vanishing gradient which affect both sigmoid and tanh.

The main drawback related to ReLU application involves the sparsity introduce by the function itself. This phenomena is called "**dying Relu**". The root cause which lead to the arise of this issue lies in the possible negative quantities which can enter a specific neuron leading to its non-activation.

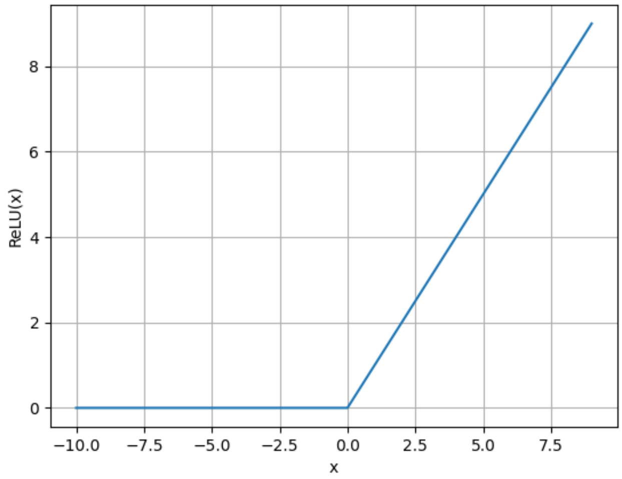


Figure 3.1: ReLU activation function

Leaky-ReLU

As it was said in the previous paragraph, the main problem related to ReLU concerns dying ReLU phenomena. The only way to cope with this drawback involves the introduction of new activations which consider also the negative quantities. It is clear that they have to be defined in order to be non-linear. A first possible solution has been introduced by the **Leaky Rectified Linear Unit**, or **Leaky-ReLU**. Its mathematical description is reported in the following:

$$LeakyReLU(x) = \begin{cases} x, & if \quad x >= 0 \\ a * x, & otherwise \end{cases} \tag{3.2}$$

It differs from the previous one only for the negative side in which a different slope is introduced. This slope is equal to a which is a parameter that has to be tuned. By introducing this slight variation, the saturation problem disappear, indeed also the negative inputs are mapped into a non-zero value which guarantees the suppression of the problem. Also in this case the function is computationally cheap.

The disadvantages related to this new function can be noticed in the excessive reduction of the sparsity of the weighting matrix. In fact in ReLU case, some neurons will remain inactive. This phenomena helps in reducing the required computational cost during each prediction if and only if a restricted number of inactive neurons is involved. Applying LeakyReLU, no sparsity will be introduced.

In the end, the visual description are reported in Fig3.2

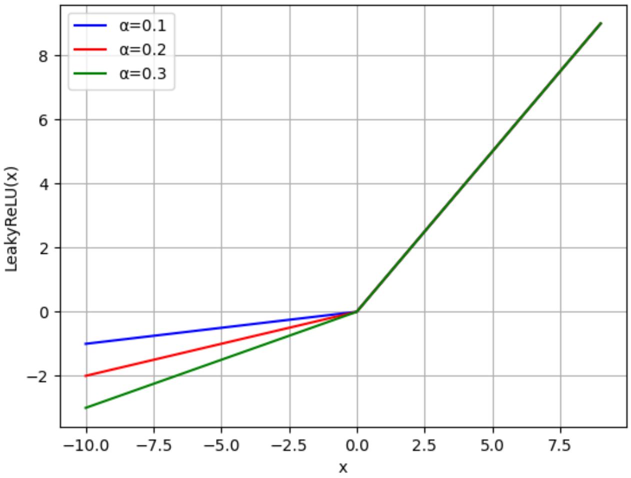


Figure 3.2: LeakyReLU activation function

SELU

The next function is the **Scaled Exponential Linear Unit** (**SELU)**. Also in this case the differences involve the left part of the cartesian plane only, i.e.:

$$SELU(x) = \lambda * \begin{cases} x, & if \quad x >= 0 \\ \alpha * (e^x - 1), & otherwise \end{cases} \tag{3.3}$$

where both λ and α are fixed values stated in the related paper [11] The main advantage introduced by it is related to its internal normalization property.

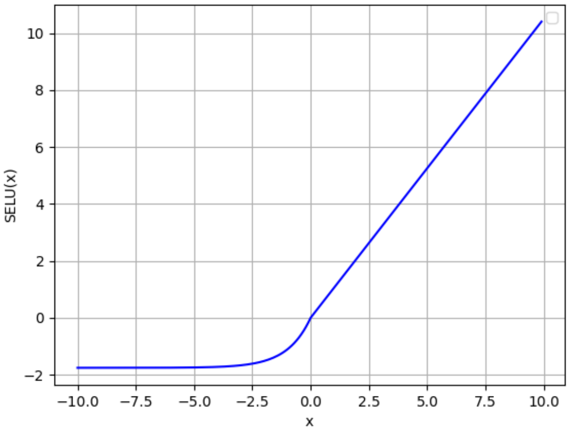


Figure 3.3: SELU activation function

Swish

It has been developed by Google's research team. Respect to the previously introduced transfer function, this one has a different structure which does not base on the ReLU only. Its mathematical description is:

$$Swish(x) = x * sigmoid(\beta * x) = \frac{x}{1 + e^{-\beta * x}} \tag{3.4}$$

This introduced a set of improvements, namely it cannot cause dying neuron and it cannot be affected by vanishing/exploding gradient problem. On the other hand, some critical aspects are shown. Indeed, it is more computationally expensive than the other cited function. Moreover, from [8] its instability is stated.

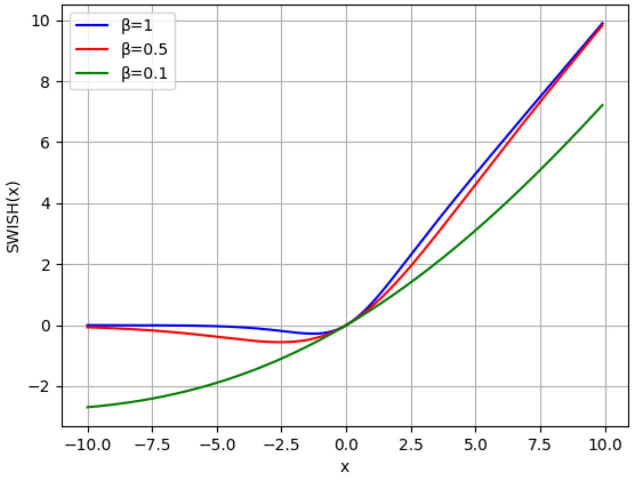


Figure 3.4: Swish activation function

Linear

This activation function is the easier one and it is used only for the output neuron in regression problem because of its constant derivative which does not contribute into the update of the weights preserving the right output. Its mathematical expression coincides with the line equation. A visual description is shown in Fig3.5

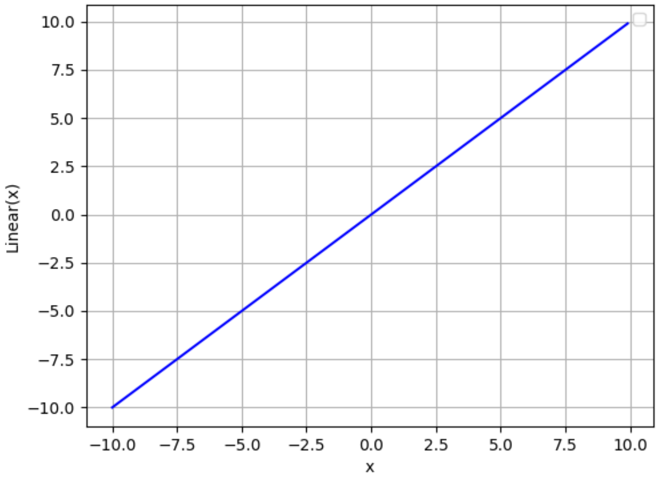


Figure 3.5: Linear activation function

Appendix B - Code

```
def create_model(trial, input_dim):
    layer = trial.suggest_categorical("layer", [1, 2, 3])
    lr = trial.suggest_float("learning_rate", 1e-3, 1e-1, log=False,
              step=0.005)
    units = trial.suggest_categorical("units_1", [128, 256, 512, 1024])
    dropout = trial.suggest_categorical("dropout", [0.3, 0.4, 0.5])
    params = {layer:'layer', learning_rate:'lr', units:'units',
              dropout:'dropout'}

     return model_definition(params)


import tensorflow as tf
from tf.keras.optimizer import Adam
from tf.keras.layers import Dense, Dropout

def model_definition(params, input_dim):
    adam = Adam(learning_rate = params['learning_rate'])
    model = Sequential()

    if params['layer'] == 1:
         model.add(Dense(input_dim=input_dim, units=params['units'],
                  kernel_initializer='he_uniform', activation='relu'))
         model.add(Dropout(params['dropout']))
    elif params['layer'] == 2:
        model.add(Dense(input_dim=input_dim, units=params['units'],
                 kernel_initializer= 'he_uniform', activation='relu'))
        model.add(Dropout(params['dropout']))
        model.add(Dense(units = params['units_1'],
               kernel_initializer= 'he_uniform', activation='swish'))
        model.add(Dropout(params['dropout']))

    else:
        model.add(Dense(input_dim=input_dim, units=params['units'],
               kernel_initializer= 'he_uniform', activation='swish'))
        model.add(Dropout(params['dropout']))
        model.add(Dense(units = params['units'],
               kernel_initializer= 'he_uniform', activation='relu'))
        model.add(Dropout(params['dropout']))
        model_.add(Dense(units = params['units'],
               kernel_initializer='he_uniform', activation='relu'))
        model.add(Dropout(params['dropout']))

    model.add(Dense(units=1, kernel_initializer='he_uniform',
    activation='linear'))
    model.compile(optimizer=adam, loss='mse', metrics=['rmse'])

    return model

import tensorflow as tf
import tf.keras.backend as K
import binary search

def Weighted_loss(weight_list):
   def loss(y_true, y_pred):
        weights = []
        for true in y_true:
            serached_index = binary_search(weight_list[:, 0],
                             K.get_value(true))

            weights.append(weight_list[serached_index, 1])


        square_differences = K.reshape(K.square(y_true - y_pred),
                                       (-1, 1))
        weight_tensor = K.reshape(tf.convert_to_tensor(weights),
                                  (1, -1))

        w_mse = K.dot(square_differences, weight_tensor)/len(weights)

        return w_mse[0]

    return loss
```

Bibliography

[1] Schwenzer, M., Ay, M., Bergs, T. et al. Review on model predictive control: an engineering perspective. Int J Adv Manuf Technol 117, 1327–1349 (2021).

[2] Jiangfeng Nan, Bingxu Shang, Weiwen Deng, Bingtao Ren, Yang Liu, MPC-based Path Tracking Control with Forward Compensation for Autonomous Driving, IFAC-PapersOnLine, Volume 54, Issue 10, 2021, Pages 443-448, ISSN 2405-8963.

[3] Shuping Chen, Huiyan Chen, Dan Negrut. Implementation of MPC-Based Trajectory Tracking Considering Different Fidelity Vehicle Models[J]. Journal of Beijing Institute of technology, 2020, 29(3): 303-316.

[4] Farag, W. Complex Trajectory Tracking Using PID Control for Autonomous Driving. Int. J. ITS Res. 18, 356–366 (2020).

[5] D. V. P. Mygapula, A. S, S. V. V. V and S. K. P, "CNN based End to End Learning Steering Angle Prediction for Autonomous Electric Vehicle," 2021 Fourth International Conference on Electrical, Computer and Communication Technologies (ICECCT), Erode, India, 2021, pp. 1-6, doi: 10.1109/ICECCT52121.2021.9616875.

[6] Srivastava, N.; Hinton, G.E.; Krizhevsky, A.; Sutskever, I.; Salakhutdinov, R. Dropout: A simple way to prevent neural networks from overfitting. J. Mach. Learn. Res. 2014, 15, 929–1958.

[7] Diederik P. Kingma, Jimmy Ba, Adam: A Method for Stochastic Optimization, published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

[8] Prajit, R.; Zoph, B.; Le, Q.V. Swish: A self-gated activation function. arXiv 2017, arXiv:1710.05941.

[9] Yuzhe Yang, Kaiwen Zha, Ying-Cong Chen, Hao Wang, Dina Katabi, Delving into Deep Imbalanced Regression. 38th International Conference on Machine Learning (ICML 2021), Long Oral, 2021.

[10] Steininger, M., Kobs, K., Davidson, P. et al. Density-based weighting for imbalanced regression. Mach Learn 110, 2187–2211 (2021).

[11] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, Sepp Hochreiter, Self-Normalizing Neural Networks. Advances in Neural Information Processing Systems 30 (NIPS 2017).

[12] He K, Zhang X, Ren S, Sun J (2015) Delving deep into rectifiers:Surpassing human-level performance on imagenet classification.In: Proceedings of the 2015 IEEE International Conference onComputer Vision (ICCV), IEEE Computer Society, USA, ICCV'15, p 1026-1034.

[13] Krizhevsky, A., Hinton, G., et al. Learning multiple layers of features from tiny images. 2009.

[14] Rothe, R., Timofte, R., and Gool, L. V. Deep expectation of real and apparent age from a single image without facial landmarks. International Journal of Computer Vision, 126(2-4):144–157, 2018.