Ensuring Safety of an Autonomous Vehicle with a Neural Network Perception

BY

JACOPO MILONE
B.S., Politecnico di Torino, Turin, Italy, 2020

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Chicago, 2020

Chicago, Illinois

Defense Committee:

Milos Zefran, Chair and Advisor
Prasad Sistla
Andrea Bottino, Politecnico di Torino

ACKNOWLEDGMENTS

The first and biggest thanks goes to my family who supported me from day one without ever letting me down or leaving my side. Besides, I want to thank all the friends that in some way or another, shared with me the University Career, because they often pushed me to put more and more effort into what I was doing and achieve my goals. The last thanks go to my advisor, who let me work with him and made this all work possible.

JM

TABLE OF CONTENTS

CHAPTER **PAGE**

TABLE OF CONTENTS (continued)

CHAPTER **PAGE**

List of Algorithms

LIST OF FIGURES

LIST OF FIGURES (continued)

FIGURE **PAGE**

LIST OF ABBREVIATIONS

AD	Autonomous Driving
NN	Neural Network
RNN	Recurrent Neural Network
LSTM	Long Short-Term Memory (Network)
FSM	Finite State Machine
ADAS	Advance Driver-Assistant Systems
ACC	Adaptive Cruise Control.

SUMMARY

The continuous technology development both in the field of simulations and neural networks is making Autonomous Driving (AD) more and more appealing for researchers. In order to make the AD a reality that would change our lives on a daily routine basis, the work is divided into two main branches: computer vision and control algorithms improvement.

Improving computer vision means making more reliable techniques such as object detection, image segmentation, etc. On the other hand in order to improve control algorithms, which already are very reliable, the idea is to include some parallel computation into their loop for the management of unforeseen events.

The ultimate goal of the research community is to improve the safety of the autonomous vehicles so that they can overcome the already very high safety of human drivers.

CHAPTER 1

INTRODUCTION

Autonomous Driving has got very popular in the latest years. As a matter of fact a common saying among expert engineers about AD is: "The question is not if it is going to happen, but when it is going to happen?". There are many goals that the AD wants to achieve; first of all solving common issues as traffic delays and collisions caused by human lack of attention. On the other hand AD would represent a real revolution in fields like Shipping, Public Transportation, Emergency Transportation and others.

1.1 Motivations

The reason why this work has started is safety. As a matter of fact in order to make possible the life-changing revolution of making AD real on a daily basis, the vehicles must be as safe as possible. Moreover, they have to be safer than a human driver, otherwise they would not represent an upgrade in people's lives. The basic idea that lies under this thesis is that a general human driver is often able to avoid collisions and crashes because of his/her experience, which allows to anticipate other drivers' bad actions on the streets. The concept of "anticipation" is exactly what it is exploited in the work; using some of the sensors available for the installation on a vehicle, we want to acquire perception data which are processed by a Neural Network so that they can be given in a smart way to the control system of the vehicle.

In parallel, we would like to use those perception data propagating them in a probabilistic way so that a probabilistic prediction of the future instants is computed. Based on this knowledge some emergency procedures would intervene in the case of bad scenarios occurrences in the prediction.

This procedure which has been briefly explained above is coming from the concept of "monitoring" [1]. In particular the idea of being able to predict the future steps producing synthetic sensor data and propagating them is what in Figure 1 is called *belief propagation*. Looking at Figure 1 it is possible to give some definitions to better understand the motivations lying under this work:

- Safety Automaton (P): represents all the safety specifications.
- Cyber-Physical System: (A): it includes the mechanical part, which can be the vehicle, and the software used for the control of such car.
- Product Automaton ($B = A \times P$): it rapresents all the actions computed by A respecting P
- Belief Propagation: this is the part which really involves the monitoring concept. It means being able to monitor if some of the possible actions performed by A are not respecting the constraints imposed by P. This concept is what we want to achieve with the reproduction of sensor data and with their propagation through the perceptions system.

One last important aspect to highlight regards the surrounding environment: we will include in the Cyber-Physical System the model for the acquisition and processing of the data which is what we call Perception System through which the knowledge of the environment is acquired.

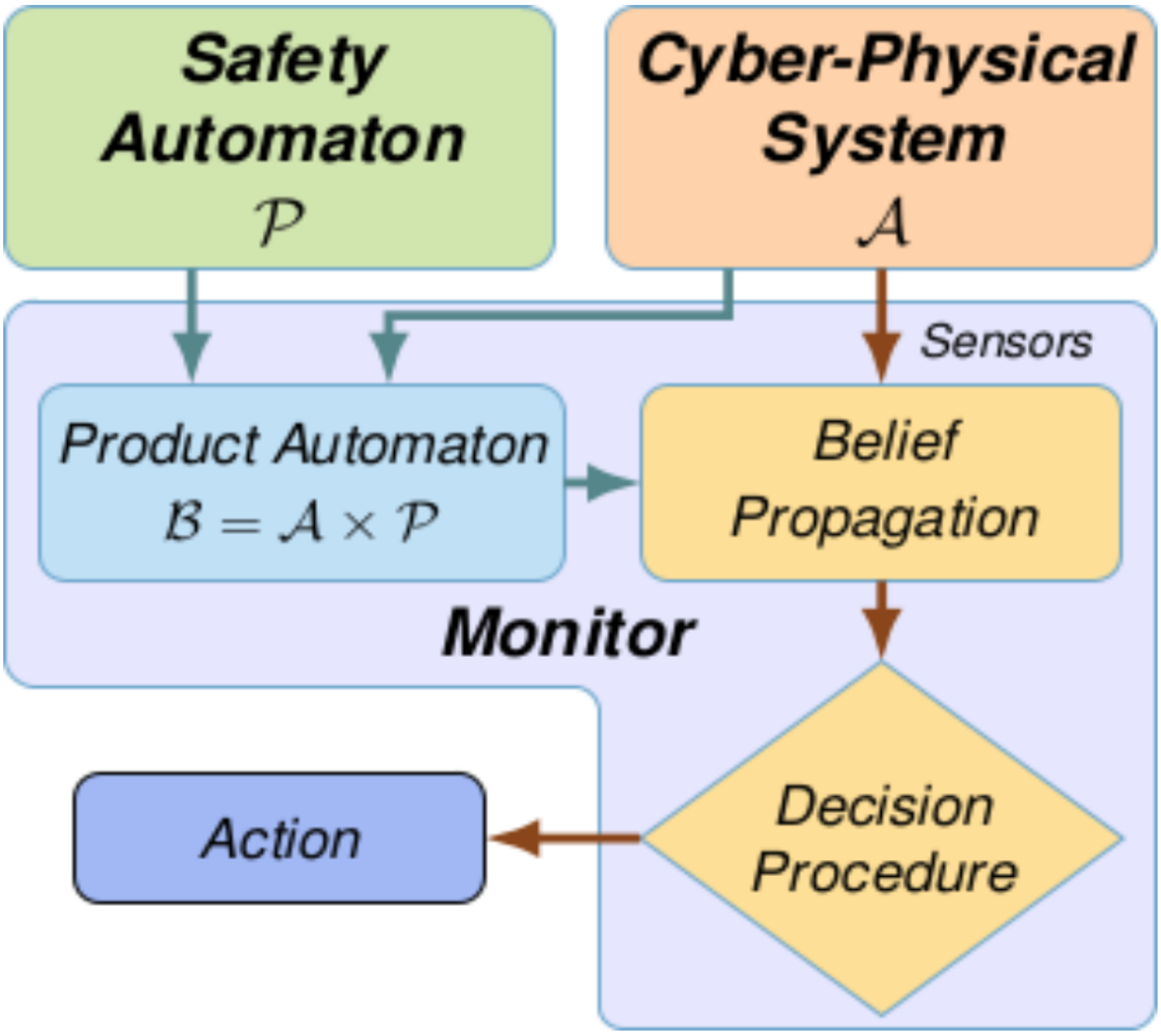

Figure 1: Scheme of the monitoring process with all the needed elements

1.2 Contributions

The principal contributions that I brought to this work, and that are explained along the dissertation are:

- Setting up the CARLA environment exploiting the Scenario-Runner and all the needed features.
- Creating a Probabilistic Model for the Leading Vehicle's Behavior useful for the data acquisition.
- Creating the Perception Model
- Developing the Vehicle Model (LSTM and FSM)
- Construct the all Prediction and Propagation framework

1.3 Thesis Structure

- Chapter 2: in this chapter a general overview on the state of the art regarding the AD research field is presented. Particular attention is focused on the concept of "Behavior prediction", giving also a brief overview of some of the existing techniques related to this subject.
- Chapter 3: here is presented the CARLA simulator, which is used as the base ground for the experiment. Being an open source simulator, CARLA gave us the possibility to exploit some features that already were created, and this is why here there is the identification of what was taken unchanged from the CARLA repository on Github, what was modified and what was created from scratch.

- Chapter 4: this chapter is dedicated to the detailed explanation of the two fundamental elements on top of which the thesis is built: controller and perception system. A little bit of background is presented and their specific functionalities in this work are explained
- Chapter 5: the development of the prediction system required to think and build several models. They are here presented and explained one by one.
- Chapter 6: the final simulation is presented and all the necessary comparison to evaluate the results are made.

Related to the thesis structure it is also crucial to explain that the thesis can be divided into two parts:

1. BUILDING THE INFRASTRUCTURE: this is the part were we had to understand how CARLA and Scenario-Runner work. We had to understand how to control the ego vehicle, how to exploit sensors and how to manage the readings of the sensors through the perception system. Chapters 3 and 4 are explaining this piece in details.
2. BUILD THE BELIEF PROPAGATION BLOCK: this is the core of the thesis and it represents the biggest contribution. Here we want to design the model of vehicles (ego vehicle and leading vehicle) as well as exploit the model for the perception in order to perform the belief propagation. These aspects are treated in Chapters 5 and 6.

CHAPTER 2

PREVIOUS WORK

2.1 Autonomous Driving Overview

The Autonomous Driving Challenge is demanding under many aspects, but in order to bring it to its full potential, and make it be the center of the revolution it is supposed to represent in everybody's lives, the concept of safety must be eviscerated under all points of view. Besides the ethical problem of "who is responsible?" in case of an accident makes the task even more essential.

According to SAE International ([2]) it is possible to divide the self-driving cars into "five levels of autonomy" to which sometimes it is possible to add the Level 0.

- Level 0: this is not a proper level, mainly because the autonomy here is practically absent, and the driver is in full control of the vehicle; however there are some alert signal that can make the human driver aware of possible dangerous occurrences.
- Level 1: this is the first authentic level. It includes all the cases where the control of the vehicle is shared by the human driver and the autonomous system. A good example of such a situation are all the ADAS such as the Adaptive Cruise Control, where the autonomy regulates the throttle and the brake in order to control the speed variation, and the human driver is in charged of the steering wheel.

- Level 2: at this stage the autonomy is starting to prevail on the driver. As a matter of fact here the control system operates on throttle, brake and steering wheel, however the human driver has to be ready to intervene if the system is about to fail. This is the reason why many systems that are catalogued in this level require the constant attention of the driver, as for example they need him/her to keep the hands on the steering wheel in order to work.
- Level 3: this stage marks the complete autonomy of the system. As a consequence the human driver here can perform other tasks while the car is driving. However he/her still has the possibility to intervene and moreover there still are some tasks (usually specified) which must be computed by the driver in order for the system to keep working properly
- Level 4: the system can take full control and the driver intervention is minimal. All the systems able to operate at this level are allowed to do so only in selected zones named "geofenced areas".
- Level 5: no human intervention is required at all.

In Figure 2 is presented a generic scheme of a self driving vehicle ([3]) which is particularly useful because it communicates the importance of the perception system that an autonomous car has to have. As a matter of fact it is of primary interest to equip the vehicle with the sensors needed for the perception and interaction with the surrounding world. Among the many devices that a vehicle might have to improve its performance, there are some of them which are essential, such as:

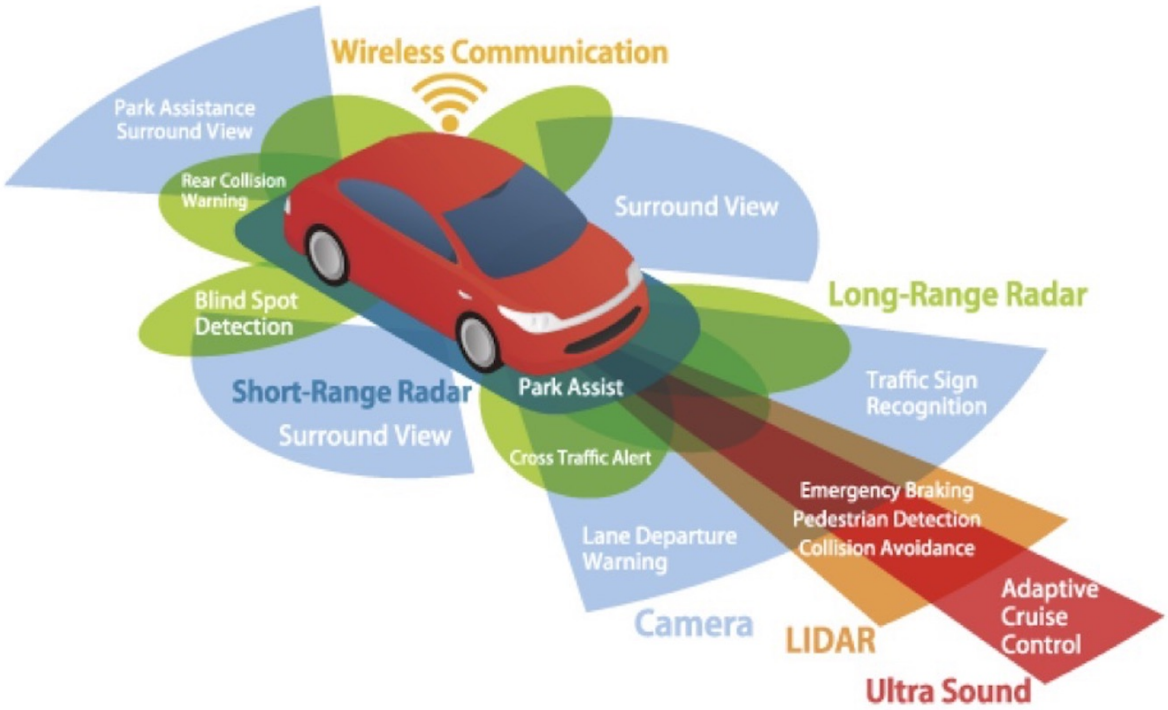

Figure 2: General scheme of a self-driving car.

- LIDAR: this is a system that exploits lasers signals and is able to create a 3D map of the environment.
- CAMERA: usually these are high-resolution cameras which can recognize colors even in conditions of low visibility; this is a very useful peculiarity because it allows, for example, to determine which is the state of a traffic light.
- RADAR: it has a function which is very similar to the LIDAR's one, except that it uses wavelengths instead of lasers. Exploited mainly because of its robustness and reliability also in bad weather conditions (rain, fog, etc.).

2.2 Behavior Prediction

The perception system of the autonomous vehicle has many margins for improvement, but what would make the car a lot safer is the concept of anticipation. This idea is strongly related to the way human drivers behave: we usually tend to base our decisions and actions on the possible behavior of other traffic participants. Of course we can do this because of our large experience on the road, but a similar concept can be implemented on an autonomous system. Following the work related to the prediction of the behavior of vehicles and pedestrians are presented.

2.2.1 Related Work

There are many studies in the field of autonomous driving which are focused on the "Behavior Prediction" theme. Following are reported some of those studies which, in one way or another, helped developing a clearer shape of this work. Koschi at al. 2017 ([4]) came up with

an idea for ensuring safety of a vehicle considering all the possible behaviors of the other traffic participants. Given that the traffic participants respect the limits imposed by the scenario, they can compute which will be the area that it can occupy, and therefore the safe trajectory of the autonomous car can be found as a consequence. They also offer a tool box named SPOT. As the authors of [4] state, they can guarantee the safety of the ego vehicle but they are also considering in all the situations all the possible actions of the traffic participants, also the least possible ones. This can make the drive style very unusual and possibly too conservative.

Gindele et al. 2010 ([5]), Chandra et al. 2019 ([6]) and Hu et al. 2018 ([7]) on the other hand were more oriented on the path that we wanted to take. As a matter of fact [6] adopts a tracking-by-detection paradigm, which is performed in two steps. The first step is the object detection which is used for the extraction of features for each road-agent. The second step consists in the prediction of the next state (velocity and position) of the agents using some kind of motion model. Finally, the work presented by [5] and [7] are exploiting the basic idea of giving probabilities to the actions of the other participants. [5] uses an approach based on a Dynamic Bayesian Network, while [7] proposes a Semantic-based Intention and Motion Prediction (SIMP) method, based on the exploitation of Mixture Density Networks, and in particular of the Gaussian Mixture Model (GMM).

CHAPTER 3

SIMULATION ENVIRONMENT

3.1 CARLA

Training and testing an algorithm implementing any kind of autonomous task in the physical world can be very hard, mainly for cost and security reasons. In order to overcome this obstacle the simulation environments are used. CARLA, which is an acronym standing for "Car Learning to Act" ([8]), is an open source urban simulation environment. It is designed over the Unreal Engine 4 (UE4) platform which guarantees a good physic design of the vehicles. The fact that it reproduces an urban driving condition is particularly interesting because it allows to create critical scenarios over which design and test some innovative algorithms.

CARLA simulates the interactions between an agent and a whole world around it which is both static and dynamic. As a matter of fact the maps are always representations of cities, with buildings, sidewalks, intersections, traffic lights, signals, etc. Besides they can be populated by other agents such as vehicles (cars, motorcycles), pedestrians and bicycles. Each of the dynamic agents can be controlled so that it follows the road without invading the other lanes, respecting signals and speed limits and trying not to collide with any other moving agent. All these scenarios can be implemented thanks to the fact that CARLA is thought to be a client-server system, where the server renders the scene that is given by the client. The client API is

implemented in Python and through it is possible to design and control an own agent which will simulate the autonomous vehicle (Figure 3).

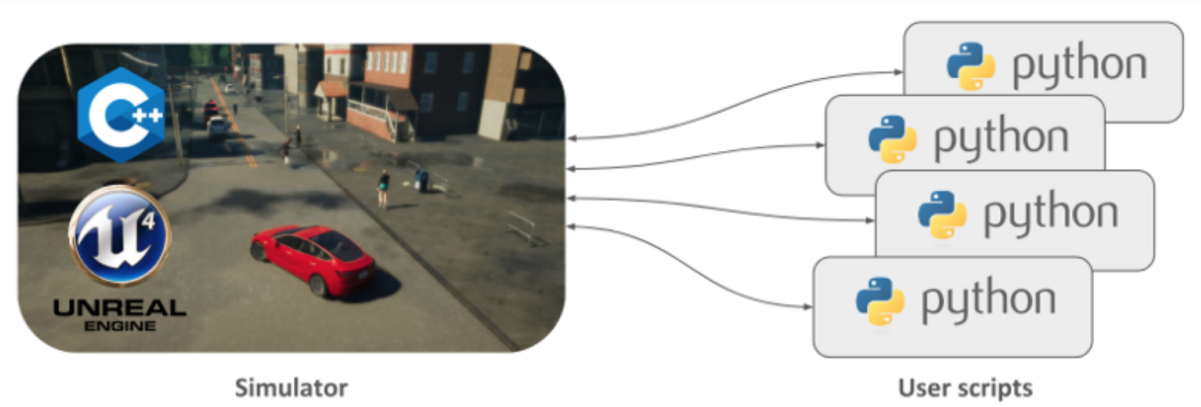

Figure 3: Server-Client communication through Python API.

Because of the autonomy concept it is important to distinguish between what the ego agent is allowed to know and what should be captured by looking at the surrounding environment. It is indeed possible to extract a lot of information such as position, velocity and acceleration about all the agents in the scene, but we must keep in mind that every algorithm should be able to work also on the real world, and therefore it is mandatory to acquire the knowledge of the set in some other way than the software itself. This is the reason why a multitude of sensors is implemented; they can be included in the simulation by being attached to the ego vehicle, and

they return a set of data that can be used in the development of the algorithm. Examples of CARLA sensor are:

- RGB Camera
- Depth Camera
- LIDAR
- IMU
- Radar: this is widely used in this work.

3.1.1 Scenario Runner

Among the many features of the simulator the Scenario Runner repository was found to be particularly useful because it was setting a very good base ground for the creation of a specific situation on which develop the work. In fact after exploring its features it appeared that it was giving us the chance to exploit some common scenarios such as "Following a Leading Vehicle". This is particularly interesting because it allows to define the behavior of the other agent(s) by using a pytree sequence and make it run on a specified map at a particular location, while the user is allowed to control the ego vehicle from his/her keyboard by running the "manual control" file. The two separated files communicate once the Unreal Engine editor is open and the scenario is supposed to end successfully if the user makes the ego vehicle behave as it should, or unsuccessfully if the user takes too much time to control the vehicle.

As it is shown in Figure 4 the interface of the manual control is particularly well configured, as it also gives a HUD section with a lot of useful information; it is indeed possible to monitor

Figure 4: Interface with the pygame window. It is evident how on the left there are a lot useful information such as: throttle value, brake value, steering angle, distance from the leading vehicle, etc.

essential values such as position and speed of the ego vehicle, throttle and brake percentage, steering angle, distance from the leading vehicle, as well as the occurrence of bad events such as collisions. In addition the presence of the keyboard interface gives the user the possibility to easily modify some general aspects; it is for example possible to change the state of the sensors toggling them into inactive if they were active and vice versa. Moreover there is the possibility to change, for example, the weather conditions. Many of them are implemented in CARLA, reproducing sunny, light rainy, heavy rainy days as well as different position of the sun creating

sunrises, sunsets or making the sun be low on the horizon line in front of the driver and reducing therefore the visibility (Figure 5 displays some of the possible choices).

Figure 5: Four examples of weather conditions, moving clockwise from top left we have: after rain (it is not rainy but the road is wet), clear sunny day, sunset, heavily rainy.

CHAPTER 4

CONTROLLER AND PERCEPTION SYSTEM

The pillar onto which is developed this all work is to improve the safety of an autonomous vehicle by anticipating the possible behavior of the other traffic participants, and in the case of bad outcome prediction such as a collision, arising an emergency behavior of the controlled vehicle to avoid the fulfillment of that specific outcome.

In order to do so the ego vehicle must be equipped with sensors that give sequences of data to be used by a controller, and obviously a controller itself. The scenario which is used to developed the algorithm is a "Following Leading Vehicle" task on a straight road; the task has to be successfully completed by the ego vehicle in complete autonomy. As a consequence we had to develop:

- Controller
- Perception System
- Prediction System

4.1 Controller

Before proceeding with the explanation of the implementation of the controller it is crucial to underline the fact that it is a base element which of course needs to be designed in order for the all experiment to work, but also that the design of such a controller is not the core of the thesis meaning that we want a reasonable implementation which however can sometimes be faulty so that we can concentrate on the development of the monitoring part.

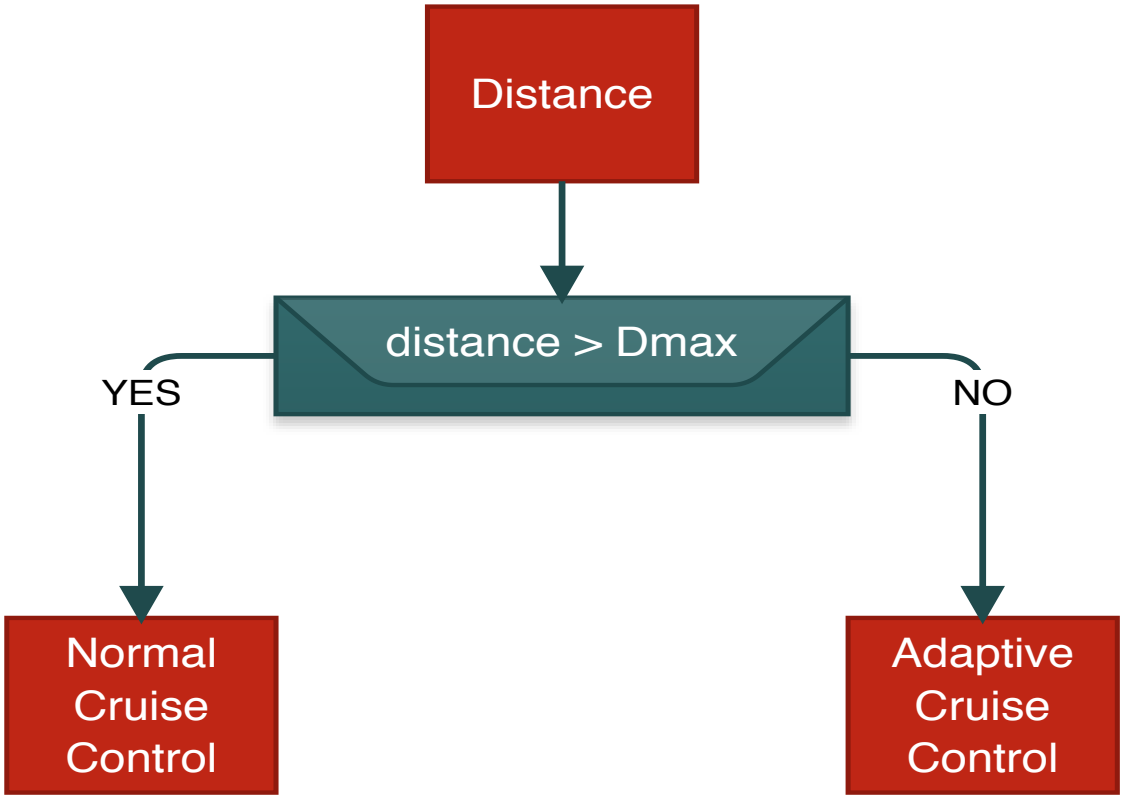

Figure 6: Top view of how the controller of our car is working

As long as the goal is the fully self-driving car, and given the specific scenario in which we operate, a good idea seemed to use a Cruise Control algorithm. This however just gives a target speed to the vehicle, that once reached, will not change because CC does not allow to modify the target speed depending on the external conditions. As a consequence we opted for the implementation of an Adaptive Cruise Control with the features specified by Shakouri et al. (2011) in [9]. This means that the ACC will operate in two different modes (Figure 6):

- Cruise Control Mode: if the distance between the ego vehicle and the leading vehicle is greater than a fixed distance. This means that if the vehicle has a reasonable amount of free road ahead of itself it can behave as there were no other traffic participants.
- Adaptive Cruise Control Mode: also addressable as "distance tracking mode", if the distance between the two vehicles is lower than the fixed value. In such a case the velocity of the leading vehicle is given to the controller which uses it as the target speed for the ego vehicle.

The controller that was adopted to implement such a behavior was a PID (Proportional-Integral-Derivative) controller, essentially because of two reasons. The first one is of course linked to its functionalities; in fact the PID controller, as suggested by the name, is a three-term controller ([10]) which is often used for ACC systems because it continuously calculates the error *e(t)* computing the difference between the desired *setpoint* (SP) and the current measured *process variable* (PV); it consequently updates its output signal *u(t)* based on the proportional, integral and derivative terms.

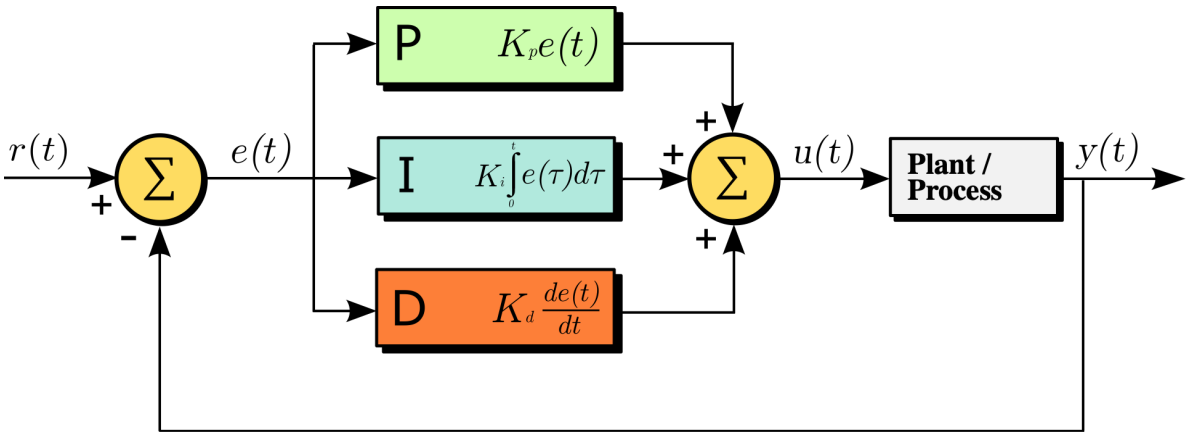

Figure 7: This is the block scheme of a PID controller. *r(t)* is in this scheme the desired *setpoint* SP; the *process variable* PV is here denoted as *y(t)*. The *error e(t)* is computed thanks to the feedback loop.

As it is further understandable looking at Figure 7 the output of the controller is computed considering the contribution of the three terms which are suitably associated to their gain factors: K_p, K_i and K_d. Below a brief explanation of each term is presented.

- Proportional: this term is linked to the difference $e(t) = SP - PV$ by the constant gain K_p. If the controller only has the proportional term the steady state error does not go to zero (Figure 8). Besides tuning the proportional gain to a high value leads to a great change in the proportional term for a given error which makes the controller very sensible but also unstable.

- Integral: this is the term which represents the sum over time of the past errors which is multiplied by the constant K_i. It is responsible of making the system reach the set point

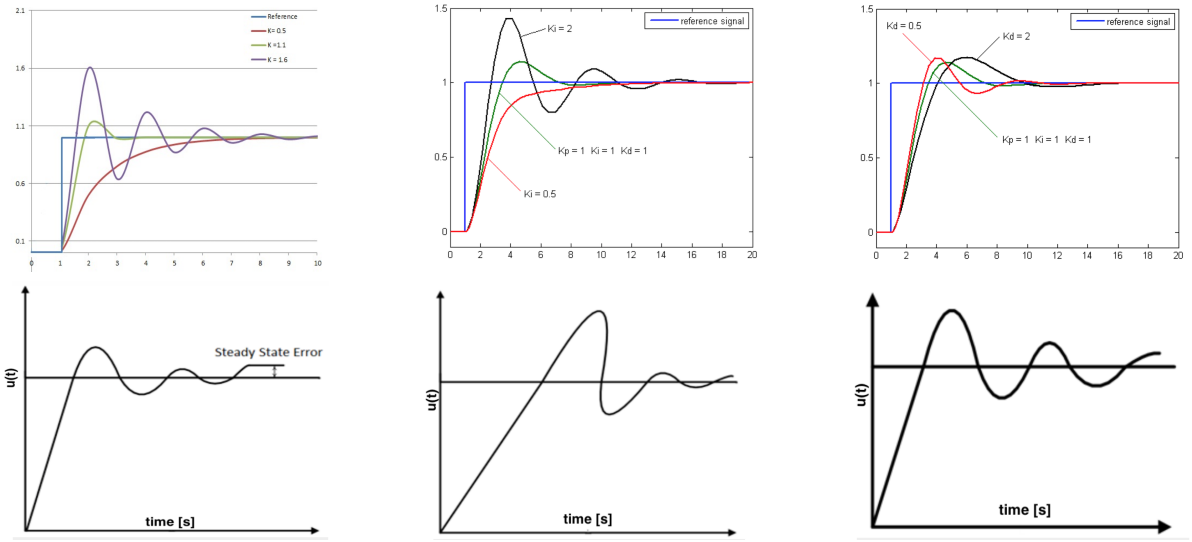

Figure 8: From left to right: (a) controller response when changing K_p; (b) controller response when changing K_i, (c) controller response when changing K_d.

faster while reducing the steady state error (Figure 8). However if the value of K_i is too high it could lead to a greater overshoot.

- Derivative: this terms computes the derivative of the error and multiplies it by the constant K_d. Tuning the K_d parameter it is possible to make the controller faster without changing the maximum overshoot (Figure 8).

The overall mathematical equation representing the input-output relation of the PID controller is:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt} \tag{4.1}$$

The effects of tuning the PID parameters are shown in Figure 8. There are several techniques that can be exploited in order to tune the PID controller, however it must be said that the second reason which led to the choice of this controller is strongly related to the tuning aspect. As a matter of fact in CARLA the PID controller is used for controlling the agents in the various scenarios and is already designed and tuned. They use two associated PID controllers:

- Later PID controller: which controls the steering angle
- Longitudinal PID controller: which controls the velocity of the vehicle

Recalling that the goal here is to have an implementation of an ACC, we took into account only the Longitudinal Controller and we modified it in order to make it suitable for our task. As a matter of fact the control output $u(t)$ in the CARLA implementation is passed as the *throttle* value which is bounded between 0 and 1 (0% - 100%). Nevertheless to let the ego vehicle behave in complete autonomy we must be able to suddenly decrease its speed when the distance from the leading vehicle is less than the set value and the latter is strongly decreasing in speed for whatever reason, and of course we must also be able to stop the car if the one in the front is still. To achieve this result the output of the controller is modified so that it is now bounded between -1 and 1 and its value is exploited as follows:

- if $u(t) == 0$ both *throttle* and *brake* are set to zero
- if $u(t) > 0$ *throttle* is set to the $u(t)$ and *brake* is set to zero
- if $u(t) < 0$ *throttle* is set to zero while *brake* is set to $\left|u(t)\right|$

The code related to the implementation of the controller is reported in Appendix G.

4.2 Perception System

In order to give to the PID controller the velocity of the leading vehicle as the target speed to be used as *set point*, it is obviously mandatory to be able to get its value "looking" at the environment around our self-driving car. Responsible for accomplishing this task is the Perception System. It is built over the following principal elements:

- Sensors: the original Scenario Runner file was again modified, this time by adding the radar as a new sensor to be used in the simulation. The choice of exploiting the radar is explicable considering the two following reasons:

    1. The application of this work is mainly urban and as a consequence considering an average radar (as it is doable in CARLA), without necessarily distinguish between long range radar and short range radar ([11]), is going to be fine as the distances between vehicles will never be too large.
    2. Using a radar instead of a Camera for example guarantees robustness against bad weather, as a matter of fact the perception of a camera can be altered by rain and even completely canceled in presence of thick fog. On the contrary the radar perception being based on wavelengths is robust to atmospheric conditions. To further remark this concept, the data acquired for the training and testing of the Classification Neural Network are taken switching among the weather conditions allowed by CARLA.

The CARLA implementation of the radar (reported in Appendix H) is very intuitive as it prints on the simulation all the detected points with different colors: red if they are moving towards the radar, white if they are still with respect to the radar and blue if they are moving away from the radar. An example of this implementation is reported in Figure 9.

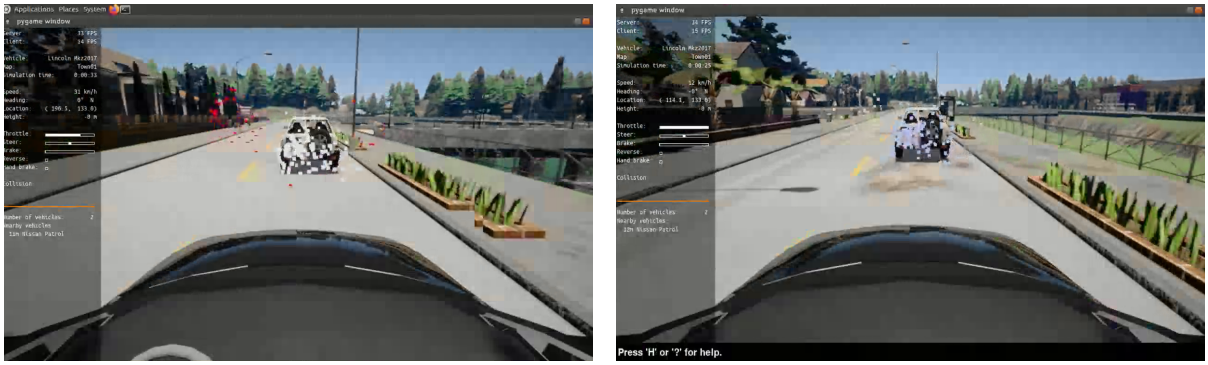

Figure 9: Radar detected points: red if moving towards the radar; white if still with respect to the radar; blue if moving away from the radar.

- Classification Neural Network: once the radar readings are available we must be able to use them properly, in particular we have to recognize which of the detected points are belonging to the leading vehicle, which are belonging to the background and if there is any other dynamic object in the scene. It has to be said that the radar returns a list of points, where each of them has four values representing its velocity, altitude, azimuth and depth all with respect to the radar. In order to do so we developed a simple NN with

the purpose of giving it a single point and getting as the output the classification of that point. In order to improve the efficiency of the network and also to make the relationship between the input and what the network had to learn to provide the correct output, we decided to give as input an 8x1 array made by the point to be classified (4x1) and the current state of the ego vehicle in that exact moment. This current state is evidently also a 4x1 array whose elements represent: the velocity of the car and its position given in the three Cartesian coordinates.

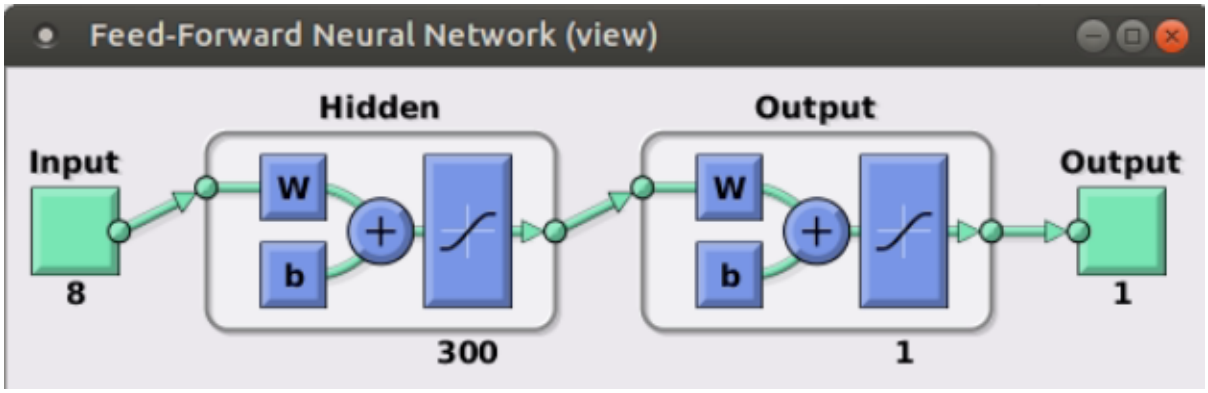

Figure 10: Structure of the Classification Neural Network. As it appears it is a Feed Forward Fully Connected Neural Network.

As it appears from Figure 10 the neural network was developed using the Neural Network Toolbox on MATLAB, and it was trained with the "trainrp" algorithm using the Nvidia GPU.

The training algorithm "trainrp" stands for *Resilient Backpropagation* and it is used especially when the activation functions of the multi-layer NN are the ones defined as "squashing"; these functions are addressed in such a way because they take an infinite input range and give a finite output range. They are usually sigmoid and hyperbolic tangent functions. In our network is indeed used the MATLAB activation function *tansig* which is the equivalent of the common *tanh*, but has a different implementation which makes it better for computational reasons. The choice of the activation function was forced by the network purposes as the output should be 1 for classifying points belonging to *leading vehicle*, 0 for *background* points and -1 for other *dynamic points*. Consequently the choice of the training algorithm was somehow forced, and as it turns out it was also the right one because it brings to the best network performance.

The necessity of coming up with the Resilient Backpropagation when using "squashing" activation functions comes from the fact that if using a usual steepest descent algorithm, the gradient can be very small in magnitude and therefore cause a small change when weights and biases are updated even if those are far from their optimal values ([12]). For such a reason only the sign of the derivative is taken into account and the magnitude has no place at all. In fact the weights and biases are increased of a certain amount named δ_{inc} whenever the derivative has the same sign for two successive iterations; on the contrary

they are decreased by the quantity δ_{dec} if the sign of the derivative oscillates; finally the update amount remains the same if the derivative is null. Moreover the weights and biases can change of a lower quantity if the sign of the derivative keeps oscillating, and they can change of a greater amount if the sign keeps remaining the same for several iterations.

The NN was tested and the accuracy was computed considering some kind of threshold:

- if $output < -0.5$ the classification is considered as -1 which means *dynamic point*
- if $-0.5 < output < 0.5$ the classification is considered to be 0, therefore *background*
- if $output > 0.5$ the classification is considered to be 1, and so *leading vehicle*

In such a way the overall accuracy of the Classification Neural Network is 99.2620%, which is a very satisfying result, achieved with a training dataset composed by 137445 samples, and a test dataset of 24255 samples (about 15% of the all set), which usually is a reliable way to divide the available data. The specific code related to the all design of the Classifier are presented in Appendix A.

- Hierarchical Algorithm: once we got to this point we were able to manage the readings of the radar sensor and classify them into three different lists by use of the Classification Network. We therefore obviously concentrated on the *leading vehicle* list because the ultimate goal remains to give to the PID controller the right information about velocity and distance of the leading vehicle. It seems reasonable to consider that the point of the leading vehicle which is of more importance for us is the closest one. As a consequence

given the *leading vehicle* list we choose among all the points the one with the lower value of distance and from it we take all the information that generalize the leading vehicle.

Running the simulation using the algorithm developed up to this point gives us as a feedback a couple of aspects:

1. The simulation is basically working: the self driving car starts, respects all the constraints, follows the leading vehicle adapting its speed and eventually it often stops without crashing into the still vehicle in front.
2. The ego vehicle *often* stops: this means that randomly the overall outcome of the simulation can be positive or negative. This is due to the fact that all the vehicles and agents are implemented in CARLA following a realistic model (PhysX Vehicles) which makes them behave slightly differently depending on small factors, like delays in the engine response etc. These are responsible for differences in the reached velocities and in the distances causing therefore some conditions under which the response of the controller is not good enough and brings the autonomous car to collide in the leading one.

This outcome is actually a good one because it shows that the algorithm is working, but also that there is margin for improvement to be hopefully brought by the introduction of the Prediction System.

CHAPTER 5

PREDICTION SYSTEM

The Prediction System required to come up with several ideas from scratch and therefore a whole chapter is dedicated to it. The following models and behavior had to be thought of and generated:

- Leading Vehicle Random Behavior Model
- Vehicle Model for Next State computation
- Probabilistic Prediction of leading vehicle's Next Mode using a Finite State Machine
- Neural Network for Synthetic Radar Reading Reproduction

The reasons why all these models had to be developed are basically two. The first one is that the leading vehicle behavior must be implemented in such a way that some actions occur with a reasonable probability; this means that the probability of suddenly stopping must be present but also very low, as well as the chances of accelerating or decelerating which have to reproduce a common behavior of a vehicle in the real world. The second reason is that the final Prediction Neural Network that we want to create, takes as input the following data (some we already have, others have to be computed):

1. Current Radar reading
2. Ego Vehicle Current State

3. Ego Vehicle Next State

4. Leading Vehicle Current State

5. Leading Vehicle Next State

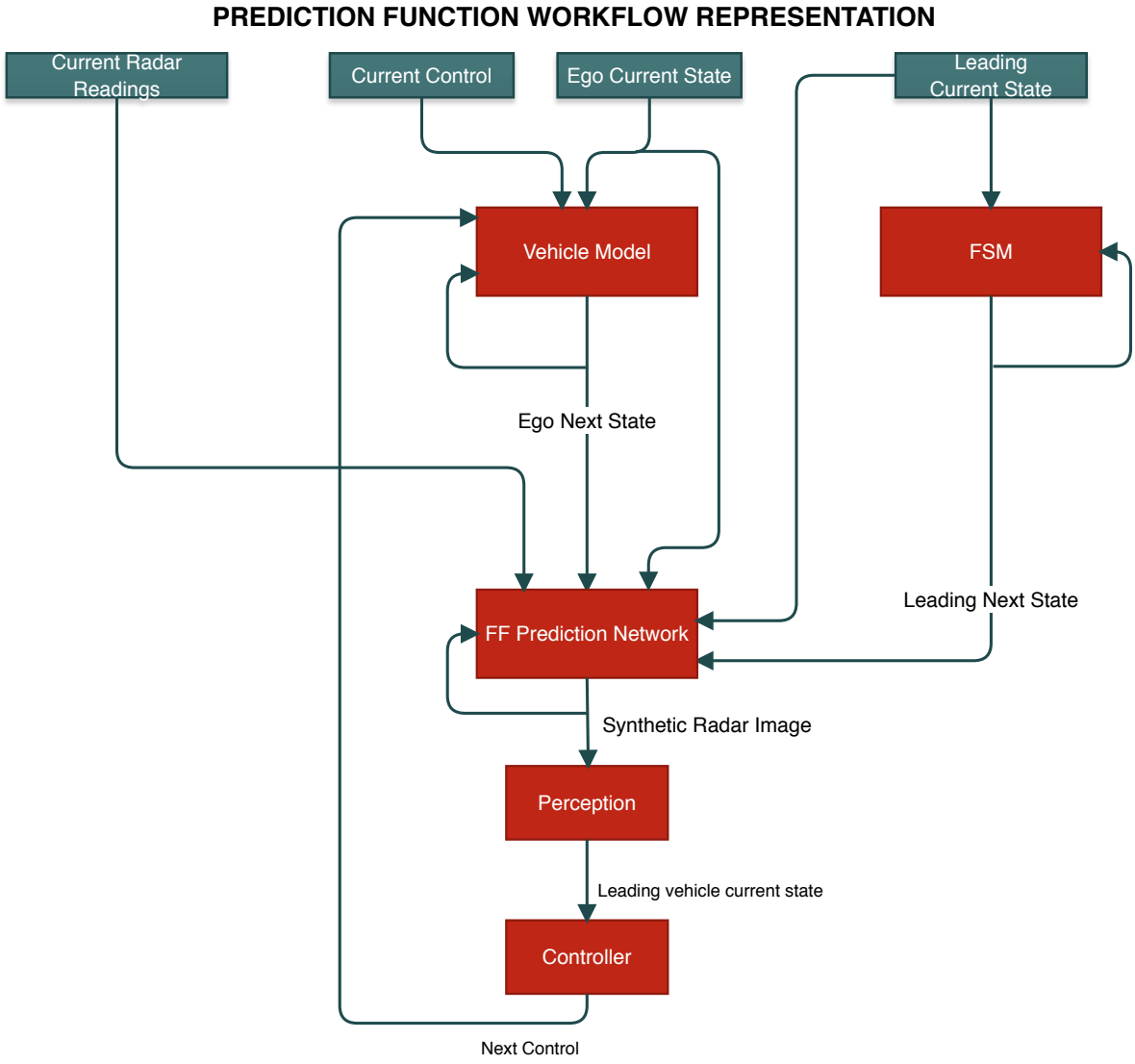

Figure 11: Loop for the prediction and propagation part. It highlights all the needed models and also all the information that will be given to the final neural network for the synthetic radar reading reproduction.

5.1 Leading Vehicle Random Behavior Model

This is a very simple model which has been thought in order to implement the behavior of the leading vehicle in the most randomized way, keeping however its plausibility. The leading car has indeed to have a predominant behavior alongside with others which should occur with a much lower probability.

In order to accomplish such goal the simple symmetry properties of the Standard Normal curve have been exploited, using a random variable as the "decision maker". The variable (named *behavior variable*) is in fact initialized to be random under the Standard Normal curve, and then the all possible behaviors are associated to the probability of that variable to assume a determined value. The pseudo codes reporting the algorithm implementation are presented in Appendix I. Following the plots reporting how the velocity of the leading vehicle is varying depending on the different behaviors are reported (Figure 12).

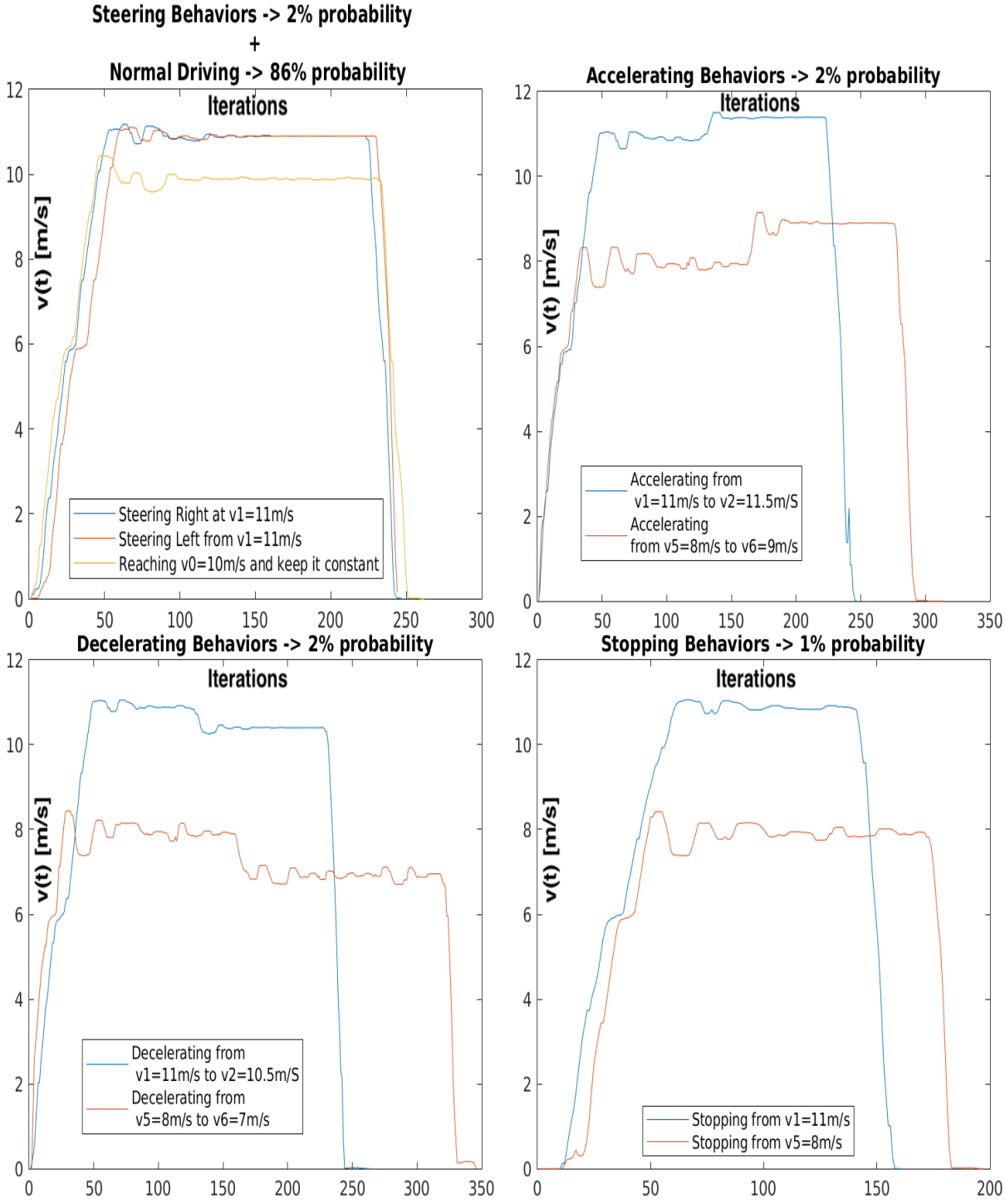

Figure 12: All the possible leading vehicle's velocity profiles depending on the different behaviors

5.2 Vehicle Model for Next State computation

One of the requisites needed to give as an input to the final Prediction Network is indeed the Next State of the ego vehicle, which means its next velocity and position when the CARLA simulator has computed a whole control iteration. At this point we know the current velocity and position of our self-driving car, and we also know the values of throttle and brake which are produced from the PID controller. As a consequence we have to be able to recreate the dynamic model of the vehicle so that the following relationship is respected:

$$x(t+1) = f(x(t), u(t)) \tag{5.1}$$

where $x(t)$ is the *current state*, $u(t)$ is the *throttle, brake* information.

In order to achieve this result the problem has been divided into two parts which can be represented by Equation 5.2 and Equation 5.3:

$$v(t+1) = f_v(v(t), u(t)) \tag{5.2}$$

$$pos(t+1) = \frac{1}{2}\frac{\Delta v}{\Delta t}(\Delta t)^2 + v(t)\Delta t + pos(t) \tag{5.3}$$

Doing so allows us to concentrate on the computation of the next velocity $v(t+1)$ and once we know this value we can use it to compute the next position simply by using the kinematic equation reported in Equation 5.3. It is important to underline that in our case Δt is the time needed by CARLA for going through a whole iteration.

5.2.1 LSTM for Velocity Prediction

Based on the information we have up to this point, and obviously on the needs related to what the function $f(v(t), u(t))$ in Equation 5.2 must fulfill, the best way to proceed seems to be by exploiting Recurrent Neural Networks, and in particular Long Short-Term Memory (LSTM) Networks. This decision if mainly the result of the fact that in order to develop a model able to reproduce Equation 5.2 we should have considered the complete model of a car in a very detailed manner, including the following pieces:

- Engine Model: needed to map the throttle value into the fuel/air flow going into the engine itself, producing a variation on the rpm.
- Transmission Model: the rpm of the engine are giving different torque values depending on the gear the car is currently driving at. Besides the automatic transmission (present in our CARLA vehicle) was complicating the relationship even more.
- Body of the Car Model: the four wheels had to be taken into account of course, alongside with the all friction forces depending on the specific type of car, in order to obtain a value of acceleration usable for our purposes.

Instead of going through this process it has been decided to use a LSTM Network which fits incredibly well for our purpose. As a matter of fact LSTMs are able to learn how a dynamic system behaves and mostly they are originally designed for managing sequences and time series data, which are exactly the kind of data set we have to deal with.

LSTMs are a kind of RNN developed to solve the vanishing gradient and exploding gradient problem which basically are the reasons why normal RNN cannot remember Long-Term Dependencies and therefore why they do not work as expected in theory. LSTM on the contrary can handle such information.

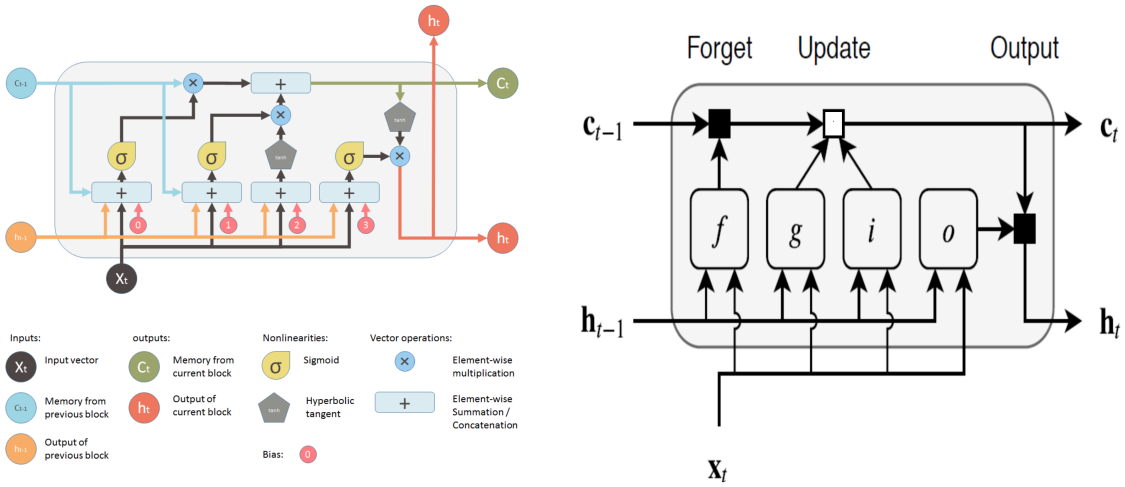

Figure 13: Left:Architecture of a LSTM Layer; Right: diagram highlighting the dependencies between gates, input, hidden state and cell state.

In Figure 13 two diagrams of the structure of the LSTM are are presented (from [13] and [14]). Looking at the picture we can see that there are three inputs:

1. c_{t-1}: Cell State at previous time step
2. h_{t-1}: Hidden state at previous time step

3. x_t: Input at current time step

The Cell State is where the memory of the LSTM is kept, the Hidden State is the output of the LSTM at time t and is also given back as an input with a feedback. Besides we can see four gates which are, looking at the picture, from left to right: *forget gate*, *cell candidate*, *input gate*, *output gate*. At last there are of course the weights and biases of the trained network which are:

$$W = \begin{bmatrix} W_i \\ W_f \\ W_g \\ W_o \end{bmatrix}, R = \begin{bmatrix} R_i \\ R_f \\ R_g \\ R_o \end{bmatrix}, b = \begin{bmatrix} b_i \\ b_f \\ b_g \\ b_o \end{bmatrix},$$

and are equally divided to perform all the multiplications needed for computing the next Hidden State and Cell State as stated below:

$$i_t = sigmoid(W_i x_t + R_i h_{t-1} + b_i) \tag{5.4}$$

$$f_t = sigmoid(W_f x_t + R_f h_{t-1} + b_f) \tag{5.5}$$

$$g_t = tanh(W_g x_t + R_g h_{t-1} + b_g) \tag{5.6}$$

$$o_t = sigmoid(W_o x_t + R_o h_{t-1} + b_o) \tag{5.7}$$

From Equation 5.4 to Equation 5.7 is reported how the output of each gate of the LSTM is computed. Considering now these four outputs and indicating with $\otimes$ the element wise

multiplication between matrices and vectors, it is possible to compute the Cell State and Hidden State at time step t, as it is reported in Equation 5.8 and Equation 5.9:

$$c_t = f_t \otimes c_{t-1} + i_t \otimes g_t \tag{5.8}$$

$$h_t = o_t \otimes tanh(c_t) \tag{5.9}$$

Looking at the set of equations above it is clear how the memory of the previous valuable information keeps running inside the LSTM and is therefore understandable how it can predict the output x_{t+1} given the input x_t of a time series data set.

As a consequence of this reasoning our own network has been developed, again as in the first case of the Classification Network, on MATLAB. The data set on which the training and testing are performed is acquired directly from CARLA; the sequence is given to the network without its last value for the input, while it is given without its first value as the expected output to be used for the supervised learning. In such a way the correspondence $t \rightarrow t+1$ is maintained. Several simulations are carried out and a set of 1872 input-output pairs are acquired, where 90% of them are used for training and 10% for testing. The input is given by:

$$input = \begin{bmatrix} v_t \\ throttle_t \\ brake_t \end{bmatrix},$$

while the output is just the value v_{t+1}.

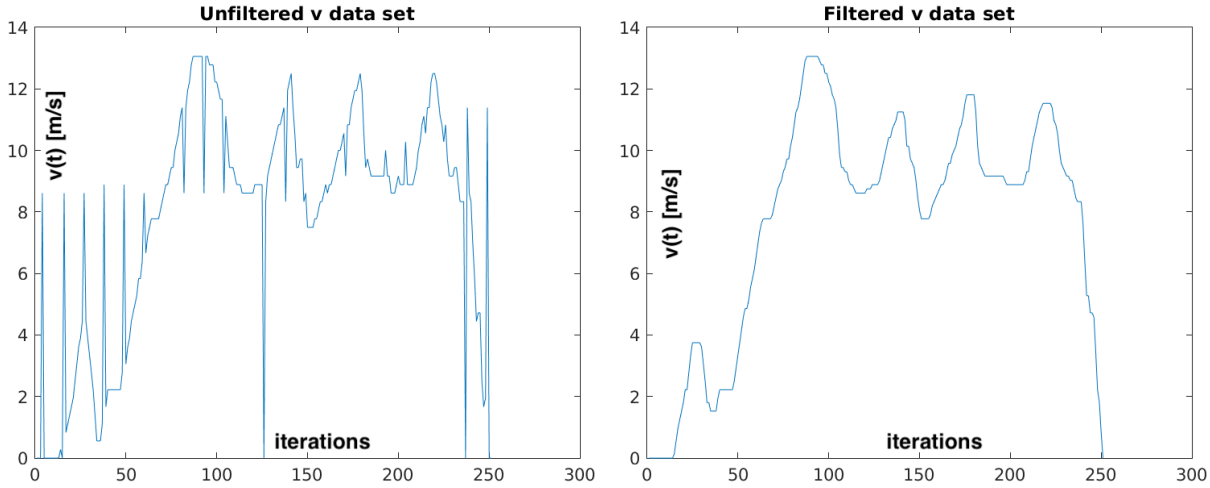

Figure 14: Left: unfiltered velocity data set; Right: filtered velocity data set.

At this point a problem in the velocity data set was detected: it was indeed full of spikes, given probably by a bug in the CARLA acquisition method which were making the signal very noisy and as a consequence the prediction was not as good as expected. As it can be seen in Figure 14 the data were filtered using a 10th order median filter (such high order of the filter is due to the very noisy data available), which means that each value of the data set $v(k)$ was considered with respect to the values down to $v(k-5)$ and up to $v(k+4)$; in such a way a neighbourhood for each value $v(k)$ is created, and the value is replaced with the mean value of that neighbourhood.

The whole network structure however is not just the LSTM, but on the contrary can be divided into the following layers:

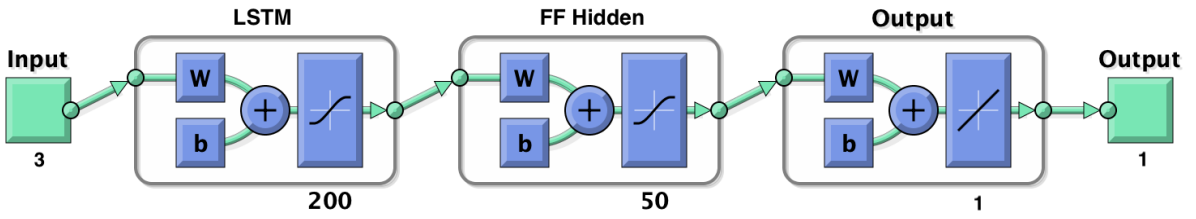

Figure 15: Structure of the all network: lstm plus feedforward network

- Sequence Input Layer: this is just the layer that takes the input and sets how many features are in fact given; in our case three features as stated in *input* above.
- LSTM Layer: this is where the recursive LSTM architecture is. In our case the number of hidden unites is 200. which means that the Hidden State and Cell State are 200x1 arrays.
- Fully Connected Layer (50 neurons): this is the classical fully connected layer which maps the 200x1 Hidden State array into a 50x1 vector.
- Fully Connected Layer (1 neuron): as long as we want to predict just the value of the velocity we need a scalar as an output and therefore we use this layer.
- Regression Layer: output layer computing the RMSE of the network.

The network is trained with such a structure for 500 epochs using Adaptive Moment Estimation Optimizer (Adam); the training process and the result obtained on the testing are shown in Figure 16 and Figure 17 respectively.

The obtained result is very satisfying as long as the predicted value is very close to the expected value for the whole test set (187) samples; as a matter of fact the accuracy computed

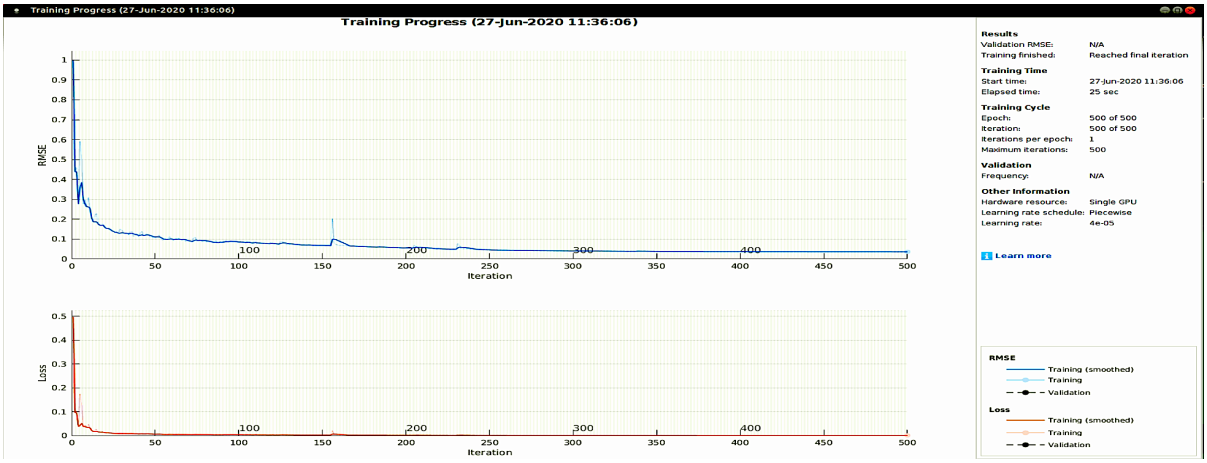

Figure 16: Training Process of the overall LSTM network. The value of RMSE and loss are computed for all the 500 epochs.

considering a prediction to be right with just 1m/s of margin is 99.4652%. This value is furthermore supported by the bottom graph in Figure 17 as we can see how just one value is slightly outside of the 1m/s boundary.

At this point we have successfully modelled the relationship expressed in Equation 5.2 and we just have to compute the variation in the velocity between two consecutive time steps as:

$$\Delta v = v_{t+1} - v_t \tag{5.10}$$

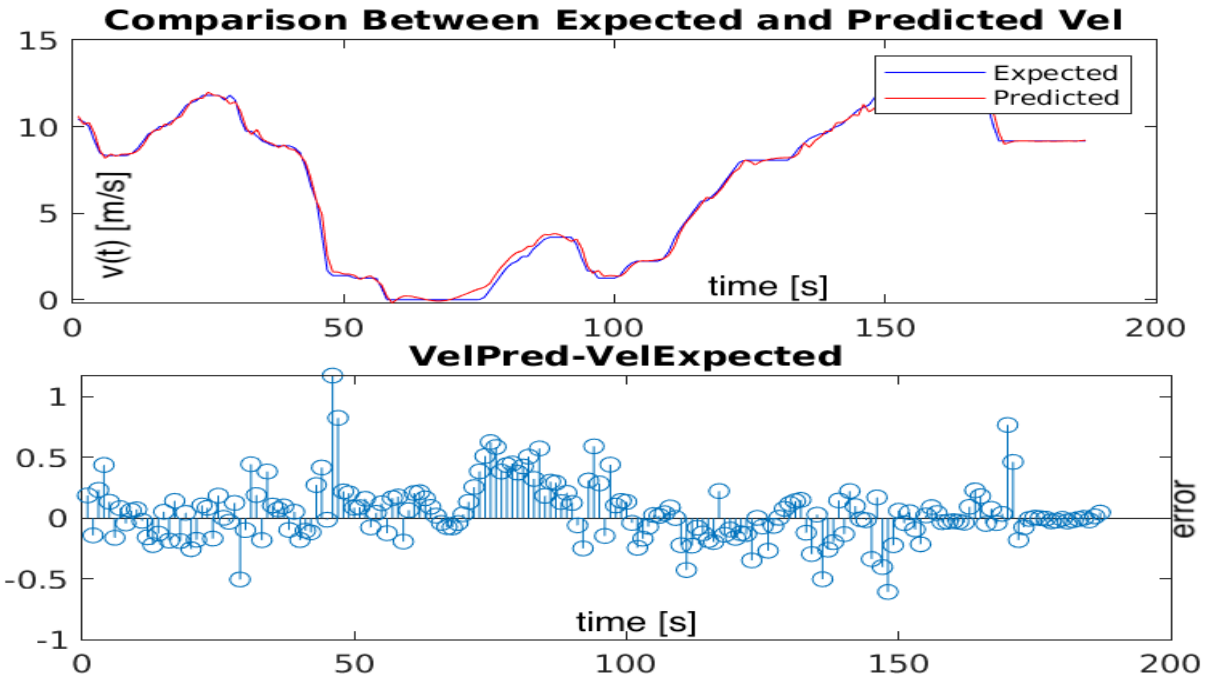

Figure 17: The picture shows the comparison between the value predicted by the network and the expected value. As it can be seen, the result is very good as the shapes of the two series are very similar and besides their difference is never higher than 1.5 $\frac{m}{s}$.

and plug Δv into Equation 5.3 to have the next position. Finally, the next state of the ego car is computed, and it is given by:

$$ego\ next\ state = \begin{bmatrix} v_{t+1} \\ x_{t+1} \\ y_{t+1} \\ z_{t+1} \end{bmatrix}$$

The code developed for training and testing the network is reported in Appendix B.

5.3 Probabilistic Prediction of leading vehicle's Next Mode using a Finite State Machine

In the previous section the whole process for predicting the next state of our self-driving car is shown and explained. It is therefore clear that now it is mandatory to find a way to compute the next state also for the leading vehicle. We have to remember that for time step t we have the current state of the ego vehicle and the radar reading that if processed (classification network) can give us the notion of relative velocity and position of the leading vehicle with respect to the ego car. We therefore need these set of information also for the prediction part.

The behavior of the leading vehicle, as explained in section 5.1 is implemented in a random way, and obviously this piece of information must not be known by the autonomous car system. Given the circumstances the problem is: how to be able to know how the car in front of us will behave in the future? To solve this issue a probabilistic approach is chosen and a Finite State Machine (FSM) is implemented. As usual throughout this all work we tried to take into account the most plausible driving actions and transitions (Figure 18) to somehow reproduce what would happen in real life on the road. The actions that we believe are a good base for make this as realistic as possible are:

- Normal Driving
- Stop
- Decelerating
- Accelerating

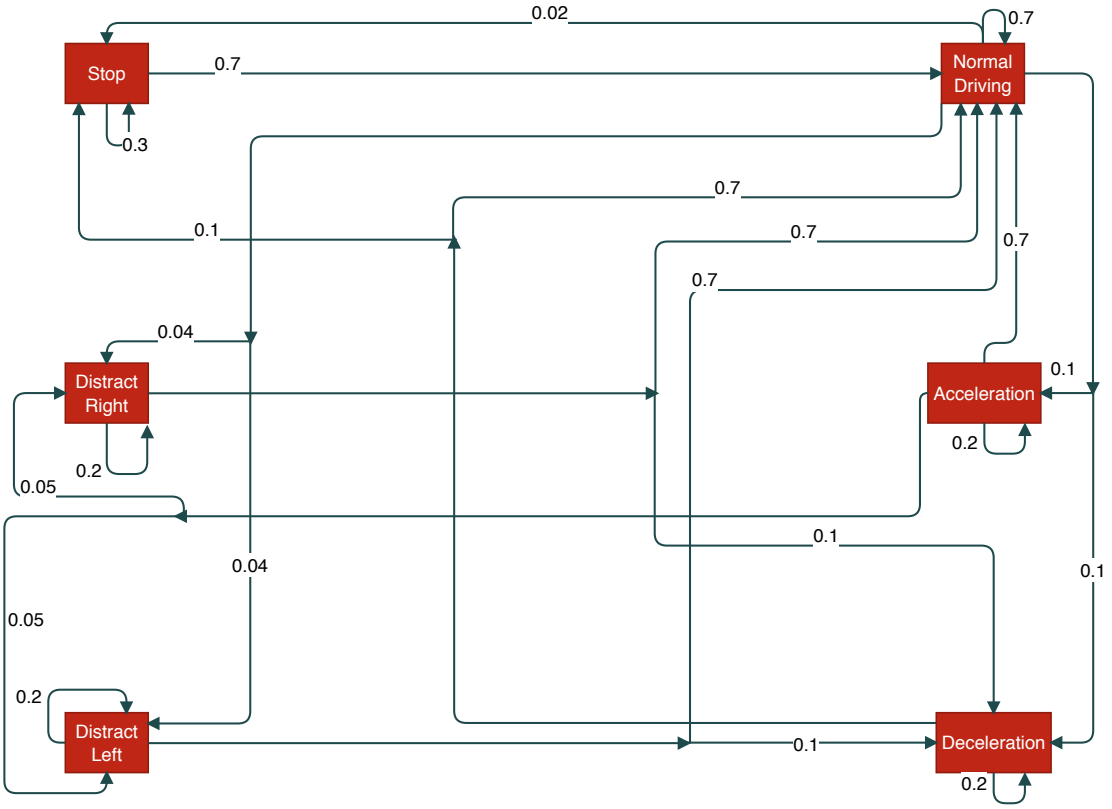

Figure 18: Finite State Machine Diagram: it shows all the modes with all the possible transitions associated with their probability. If a transition is not shown it means that its probability is null.

- Distract Right: as long as we are considering a vehicle proceeding in front of us on a straight road, we must take into account also the possibility of a bad driver steering and eventually invading the opposite lane or hit the sidewalk

- Distract Left: same concept as above.

As a consequence for the computation of the next state of the leading vehicle we proceeded in the following way:

1. Based on the knowledge of the ego vehicle state and on the selected point of the radar reading we can compute the velocity of the leading vehicle as :

$$leading\ vehicle\ velocity = relative\ velocity + ego\ vehicle\ velocity$$

.

2. The same concept can be applied for the position as:

$$leading\ vehicle\ position = relative\ position + ego\ vehicle\ position$$

.

3. Once we have the state (velocity + position) of the leading vehicle we can compute its current mode, meaning which is the FSM mode in which the vehicle is. In order to do so the velocity and the azimuth value are compared with their value at time step $t - 1$.

4. Now having the current mode we can use it to be passed to the FSM (implemented with the idea of a switch-case structure) which will give us the next mode of the vehicle, meaning that we will have the knowledge of what the action of the leading car will be. A specific variation in the velocity Δv (and variation in the azimuth for Distract Left and Distract Right mode) is associated for each transition. In order to choose this value,

which is arbitrary and can be easily tuned, it has been considered that given that the Δt between two consecutive time steps is of 1s, and that the maximum speed reached in the simulation is about 45km/h because we are considering a urban scenario. The final choices were:

- Normal Driving $\rightarrow \Delta v = 0 m/s$
- Accelerating $\rightarrow \Delta v = 0.5 m/s$
- Decelerating $\rightarrow \Delta v = -0.5 m/s$
- Stop $\rightarrow \Delta v = -current\ velocity$
- Distract Right $\rightarrow \Delta azimuth = 1$
- Distract Left $\rightarrow \Delta azimuth = -1$

In such a way becomes very easy to compute the next state of the leading car knowing its current state and how its velocity and azimuth are changing using the usual kinematic equations. Once the next state is computed it is also possible to get to know which will be the next relative velocity and distance between the two vehicles simply by subtracting the two next states. The code presenting the implementation of the FSM is reported in Appendix J.

5.4 Neural Network for Synthetic Radar Reading Reproduction

As it is stated at the beginning of this chapter, the final goal is being able to create the complete radar reading, which is a 75x4 matrix, using a neural network. The inputs of the network should be: *current radar reading*, *ego vehicle current state*, *ego vehicle next state*, *leading vehicle current state* and *leading vehicle next state*; it is therefore clear that once got to this point we developed all the necessary models needed for having all the inputs, and it is possible to proceed with the network. The network chosen and designed in this case is a Feed Forward Neural Network, designed in MATLAB, composed by 5 layers as shown in Figure 19.

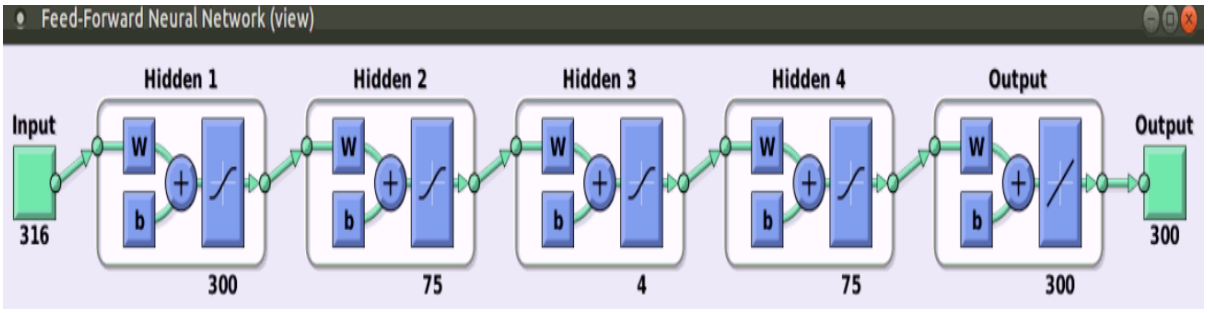

Figure 19: Structure of the Prediction Neural Network. As it is visible it is a Feed Forward network with 5 layers.

The input layer of the network is 79x4 matrix (75x4 is the current radar reading and the four states have 4x1 dimension), which is vectorized to be a 316x1 array; the output wants to be the radar reading corresponding to the next step of the simulation and must be a 300x1

(which is nothing but the vectorization of the 75x4 expected matrix). Given this mandatory features for the network, the structure is designed in order to try to reproduce the encoding-decoding behavior. As a matter of fact the layers respectively made of the following number of neurons: 300, 75, 4, 75, 300. The idea laying behind such a choice is to try to encode the input extracting the most relevant feature and then decode that information to reconstruct the whole radar reading. The activation functions are hyperbolic tangents for all the layers in order to extract the important information without letting the output value explode, and a linear activation function for the output layer, because the values must represent velocities and positions and cannot be flatten between boundaries. The training algorithm is "trainrp" which is the same one used for the Classifier.

The network is trained on 2075 samples and tested on 230. The accuracy is computed again considering an output value to be right if its difference with the expected value was less than one. In such a way the average accuracy on the 230 test samples is of 63.1565%. The accuracy is referred as "average" because of course the network has to predict 300 values for each of the test inputs and it was not constant for each of them, reaching peaks of 84.34% and downs of 52%. The code related to the design of this network is presented in Appendix C.

The accuracy here is not great, as generally speaking 63% is not a good result for a network. However it has to be considered the fact that here for each input the network is trying to predict a huge number of values and has therefore a lot more sources of error. In addition to this we must say that what really matters about the output is: how does it behave in the loop of the

simulation? Obviously to find it out, we are going to include it in the process and see what we get.

CHAPTER 6

FINAL SIMULATION

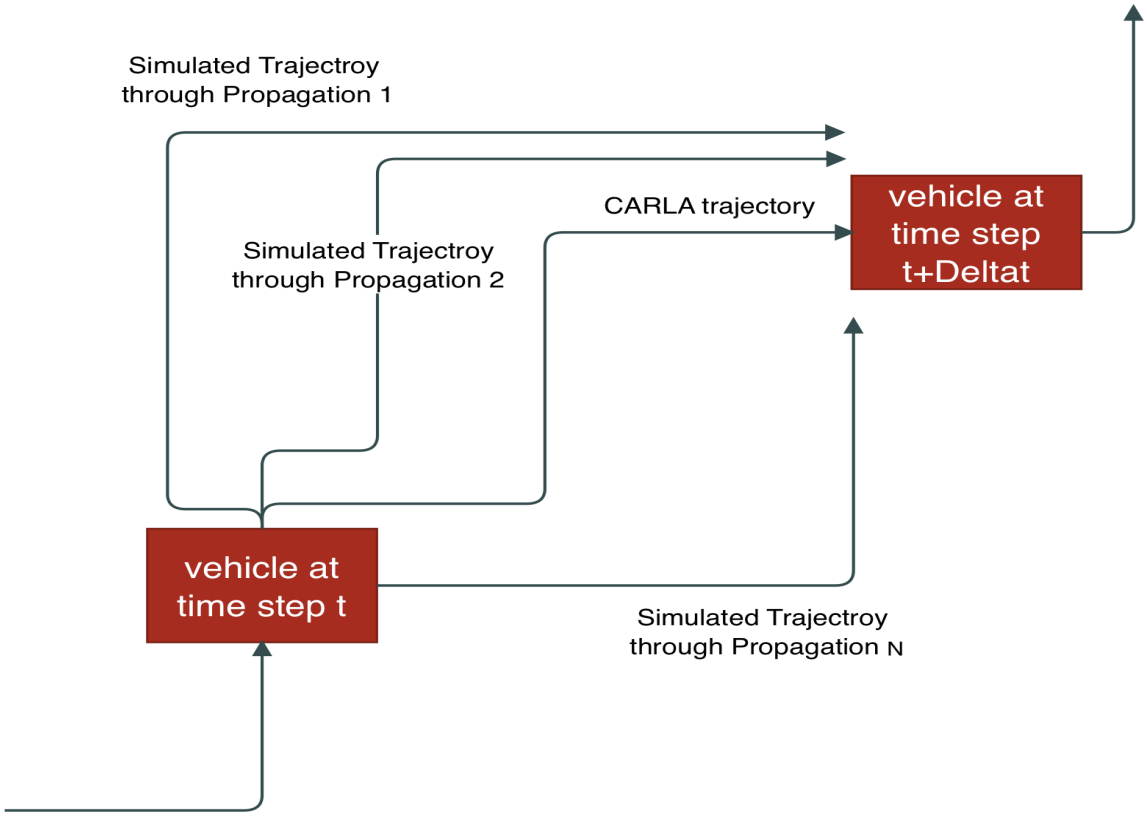

Figure 20: In our work the CARLA simulation is considered to be the reality and it is the main timeline. On top of it for each time steps N predictions propagating the future for Δt seconds can be computed and among them some should be close to the real one.

The ultimate goal of this work is to put all the pieces together in order to perform a simulation able to recreate the flowing of real life conditions. For such a reason we are going to consider the CARLA simulation as if it were reality, on top of which all the computations and predictions are made (Figure 20). The CARLA settings are changed so that the interval between two consecutive displayed images is of 1s and for each reading of the sensors, in particular of the radar, the following actions are performed:

1. The Radar Reading goes through the Classifier. Besides the ego vehicle current state is computed.
2. The points classified as belonging to the leading vehicle are selected and among them the one with the lower value of distance is taken into account (leading vehicle current state). This is because that is the point of the leading car that the autonomous vehicle is going to hit in case of a collision.
3. With the selected point the knowledge of relative velocity between the vehicles and their relative distance is acquired. The relative velocity is crucial for the computation of the leading vehicle velocity.
4. Velocity and distance values are given to the control part which depending on the second decides to act as normal Cruise Control or as Adaptive Cruise Control, in which case the first is used as reference velocity in the PID controller. From this step come throttle and brake values.

5. The current radar reading, ego and leading vehicle current states are passed to the Prediction Function which computes the next states for both vehicle as well as the next radar image as explained in the previous chapter. The function can than be included in a loop so that the predicted values can be considered to be the current ones and the prediction can be propagated N steps ahead.

The process explained above is nothing but the general workflow of the simulation which explains which actions are taken and when. Figure 21 reports a diagram useful for the visualization of such a sequence of operations.

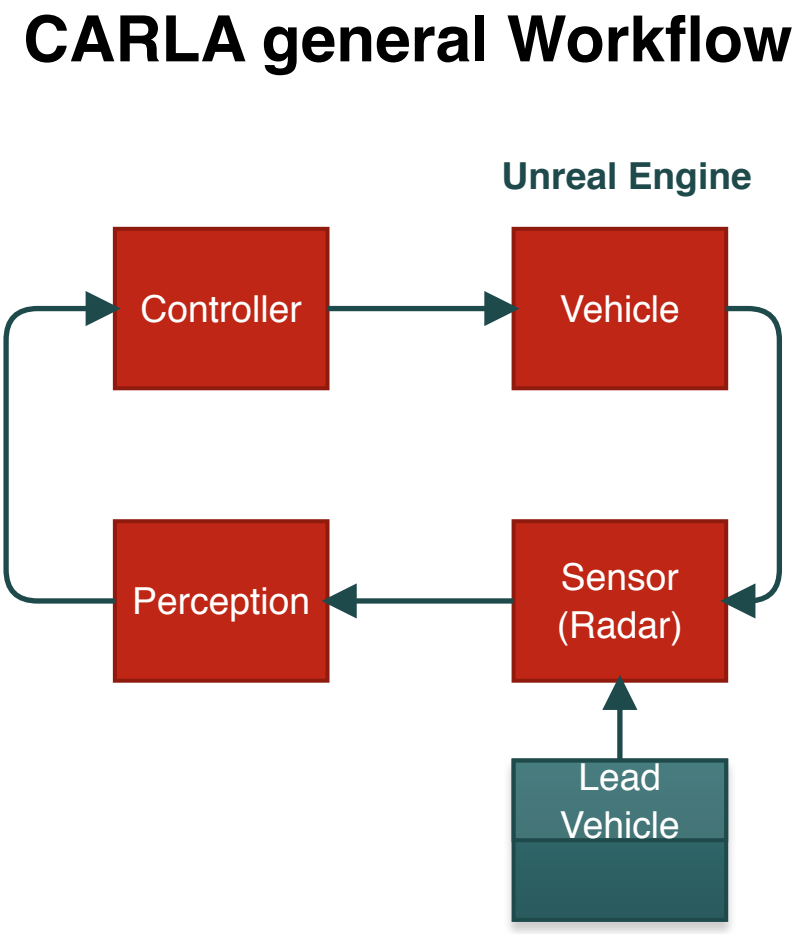

Figure 21: Flowchart representing the general workflow of the Final Simulation.

In this specific case we thought it would have been good enough to propagate the prediction for 10 steps. This is indeed what it is implemented in the code, where the 10 steps propagation is performed for each time step t. What could be checked to understand how well this system works

is the correspondence between the computed next relative velocity and distance throughout the FSM and through the processing of the Synthetic Radar Image. As a matter of fact they should be correlated and if the models are good enough they should be close as long as the first one is given to the final Feed Forward Network as an input.

6.1 First Comparison

In order to see how our all system performed, some simulations are run and for a specific instant the 10 steps ahead prediction is saved and plotted so that the comparison between FSM and synthetic radar reading can be displayed. Five predictions for the same time steps are reported; they are all different because they depend on the probabilistic behavior of the FSM. In Figure 22 the distance comparison for all the five of them is reported. This is a very explanatory image; as a matter of fact the prediction coming from the Neural Network reproduction of the radar image is not too far from the one computed by the FSM. However between time steps 20 and 30 (which means in the middle of the third prediction), the FSM is telling us that there will be a crash. The distance becomes indeed negative, which means that the ego vehicle is overcoming the leading car, but this is not possible considering the scenario settings, leaving us with the only possible conclusion of a predicted collision. Unfortunately our synthetic image reproduction system is pretty disappointing here because it is completely missing the prediction of the crash.

In order to improve a little bit this result we can try to make the FSM model less aggressive, reducing the variation in the velocity in case of a stopping condition to $\Delta v = \frac{-current\ velocity}{2}$,

and developing an analytical model able to create a radar reading from the FSM prediction value.

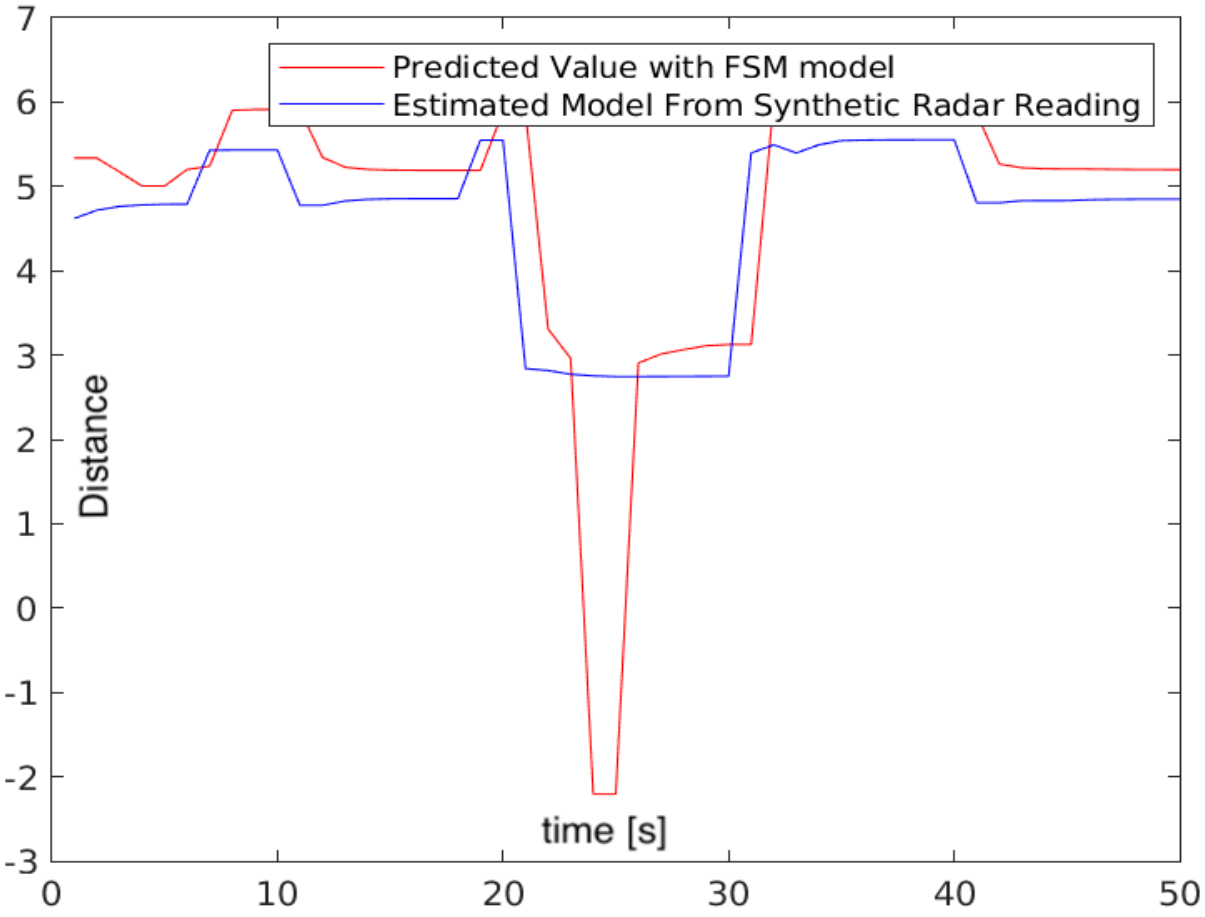

Figure 22: Comparison between the distance evolution predicted by the Finite State Machine and by the Synthetic Radar Image generation. Five predictions of 10 steps each are reported.

6.2 Second Comparison - Analytical Model

The idea lying behind this modification is that we can have a simple analytical model which is not going to take into account the randomness of the radar, but which however will be very consistent with the FSM prediction, so that it can be used as backup if the Neural Network prediction is completely off. The basic reasoning upon which this model is built is:

- Background points will remain background, and in doing so the new values associated to them will be just the negative next velocity of the ego vehicle
- Leading vehicle points will remain leading vehicle points; this means that it is possible to assign to them the state coming from the FSM.

The addition of this analytical model changes a little bit the workflow of the simulation which is reported in Figure 23. The result that we got after this changes were made to the system is shown in Figure 24. It is very good to notice a couple of things here regarding the last mentioned picture above:

- The Analytical Model output is perfectly coincident with the FSM prediction. This is very good because it means that it can be used with a very high reliability
- The crash that again is predicted here by the FSM is actually happening in the main timeline of CARLA. As a matter of fact according to the FSM the collision is going to happen at time instants 3, 4 and 5. As it is visible in the sequence of images reported in Figure 25 the crash is actually happening at the fifth step.

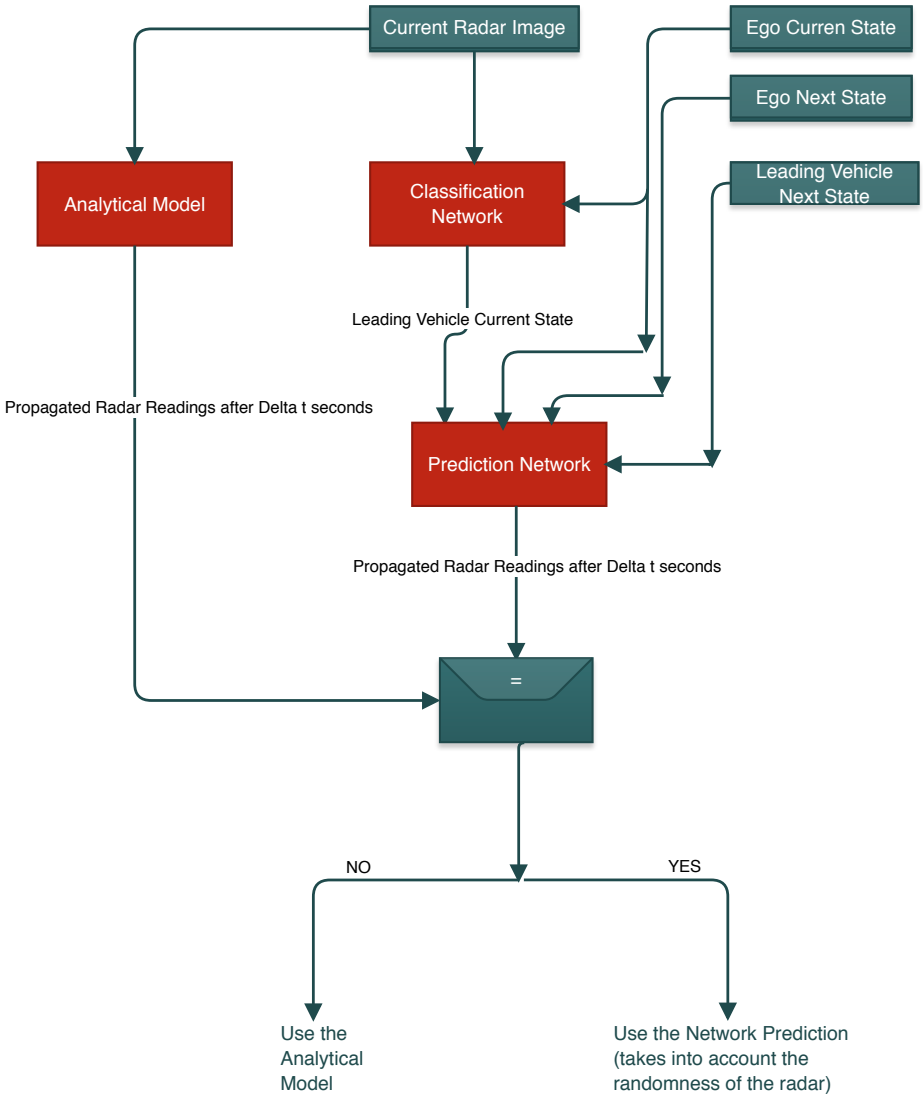

Figure 23: Modified workflow when the Analytical Model is taken into account. This flowchart is simplified with respect to the one shown in Figure 15. This is because what really has to be highlighted here is the comparison between Analytical Model and Synthetic Radar Image Estimation.

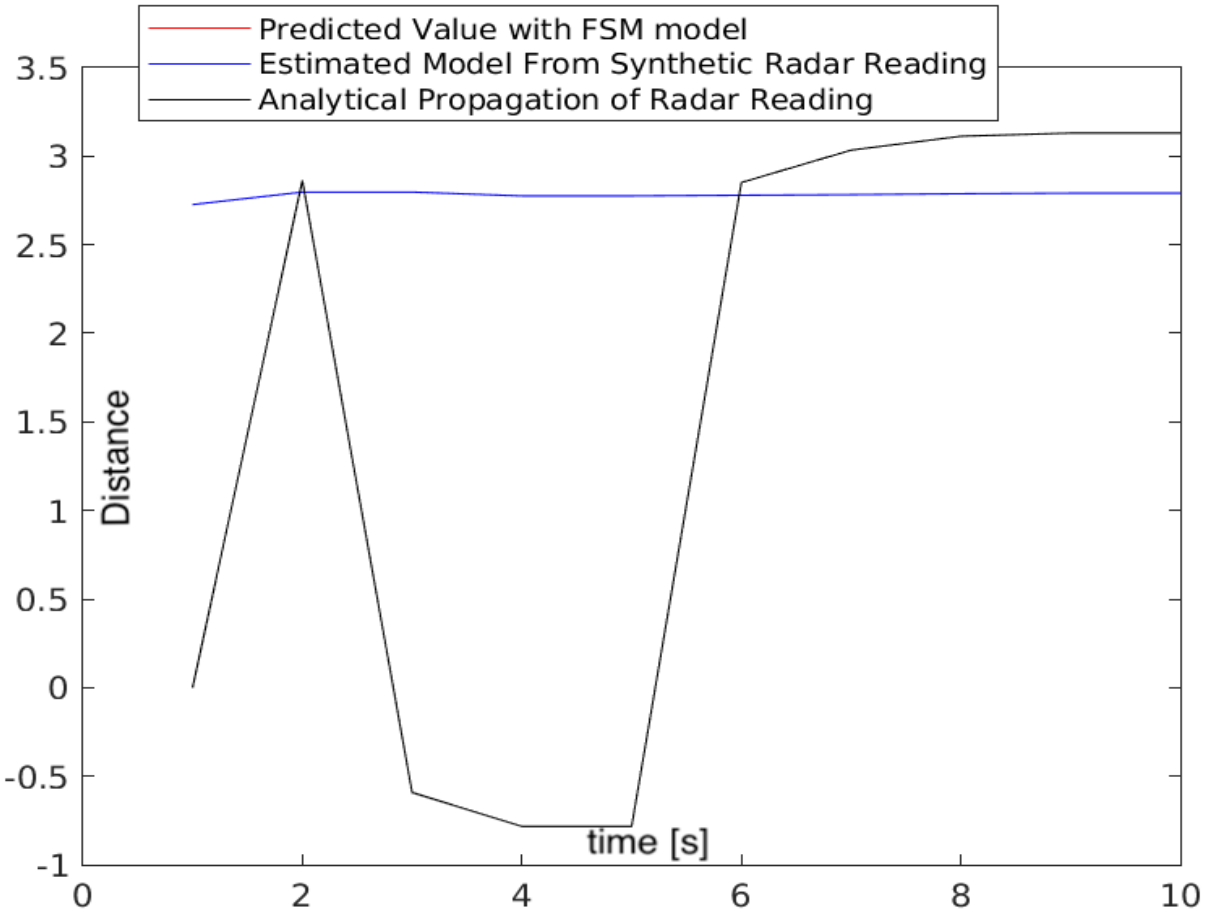

Figure 24: Comparison between FSM prediction, Analytical Model and Neural Network prediction. As it appears from the picture the Analytical model output is coincident with the FSM prediction.

Figure 25: Collision Sequence.

It must be underlined how as it was for Figure 22, also for Figure 24 the neural network prediction is pretty poor in the sense that is completely missing the collision predicted by the FSM. As long as the aim of this work is to develop a monitoring system being able to predict the possible faults of the controller by reproducing the radar images, some new strategies have to be found.

The first possible cause of such a behavior of the network could be that in the data set the number of crashes was very low; this is why the idea to improve the performances could be to retrain the network using this time a data set full of crashes.

6.3 Third Comparison - Retraining the Network

Considering that the Feed Forward Network used for the prediction of the next radar reading has to be retrained with a data set including a large number of collisions, the behavior of the ego vehicle is set so that for each simulation a crash occurs. Doing so we have to be aware of the fact that we do not want the network to be able to detect just one specific crash, and therefore we have to add some variance to the simulations in terms of velocity of the leading car and the point where it has to stop which, given the settings, would also be the point where the vehicles will collide. The data are acquired running the following simulations:

1. Stopping from $v = 15\frac{m}{s}$ at 120m from the next intersection
2. Stopping from $v = 15\frac{m}{s}$ at 130m from the next intersection
3. Stopping from $v = 15\frac{m}{s}$ at 110m from the next intersection
4. Stopping from $v = 16\frac{m}{s}$ at 115m from the next intersection

5. Stopping from $v = 15.5\frac{m}{s}$ at 125m from the next intersection
6. Stopping from $v = 16.5\frac{m}{s}$ at 135m from the next intersection
7. Stopping from $v = 15\frac{m}{s}$ at 165m from the next intersection

This new set of data is composed by 1077 samples. The network is trained with the exact same training options as before (look at Section 5.4). The new average accuracy is slightly better than before as it reaches 65.3178%.

The network is subsequently included again in the simulation and the leading vehicle behavior is left so that a crash occurs.

As Figure 26 shows, the retraining procedure brought really good effects to our network. Both the velocity and distance estimated by the network are following what the FSM is predicting, and in particular we can see in the top picture how the network is actually predicting the collision. The second thing that can be observed looking at Figure 26 is that the collision predicted by the network seems to be much lighter than the one predicted by the finite state machine. This however is not necessarily a bad thing, because we have to remember that the model associated to the FSM is still somehow aggressive and probably the Synthetic Radar Reading Estimation is closer to the real evolving of the states. This speculation is confirmed in a sense by looking at the simulation itself where the entity of the impact and the value of velocity at the collision instant reported on the HUD seem to confirm our belief.

In Appendix D and Appendix E the code for the prediction function and for the overall control loop are respectively reported.

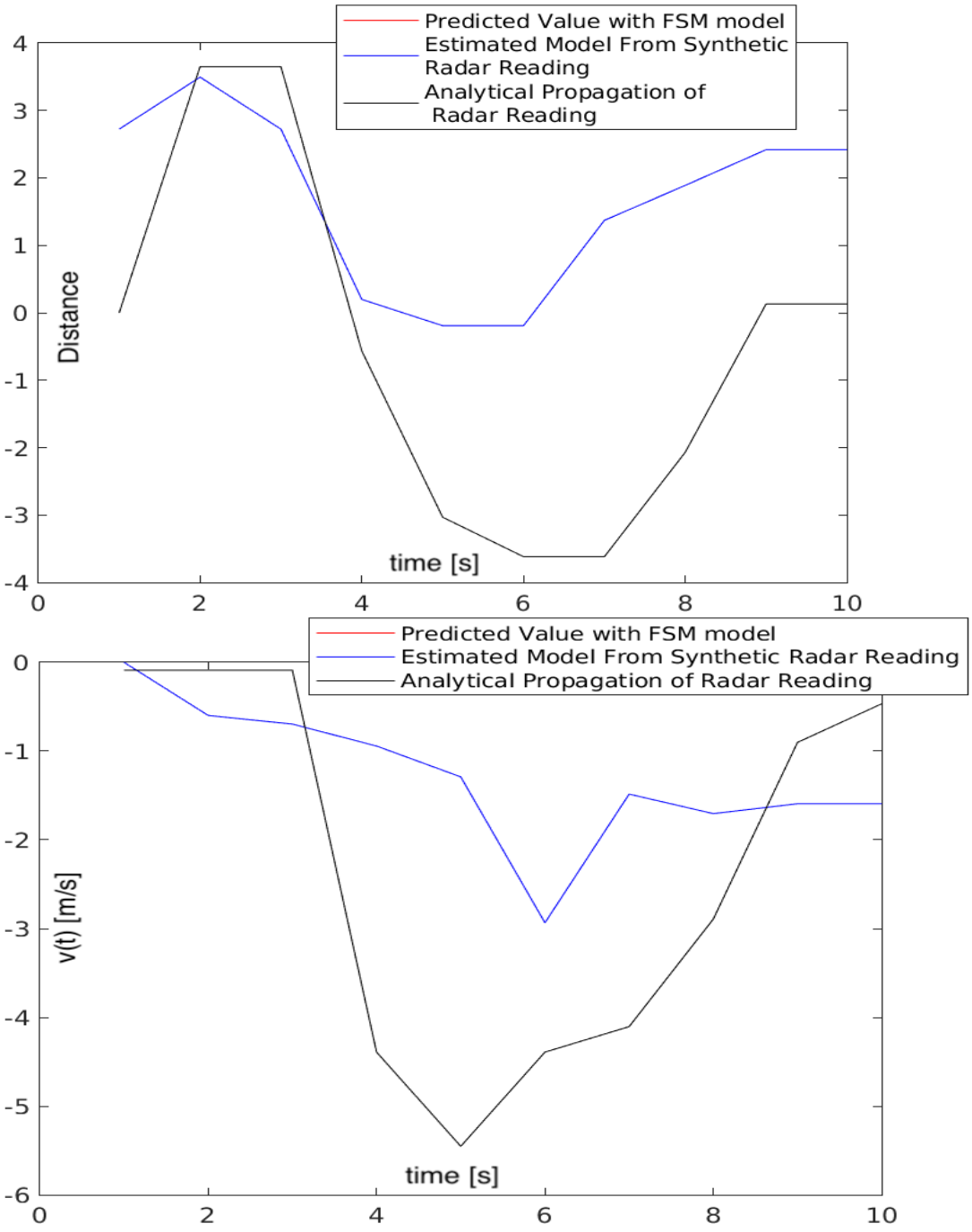

Figure 26: Top Picture: comparison in the distances between FSM, Analytical and Synthetic Reading Estimation; Down Picture: comparison in the velocity between FSM, Analytical and Synthetic Reading Estimation.

CHAPTER 7

CONCLUSION

This work provided a way to monitor what could be a possible dangerous situation in the future. In such a way it is possible to act in advance and overcome the fault of the controller exploited by the autonomous system. In doing so we are giving a good way to improve the safety of the vehicle alongside with driving style which is close to the human one, and especially which is not too conservative in all the scenarios.

7.1 Contributions

In order to properly resume what has been done regarding this thesis the following list is reporting the main personal contributions:

- The Carla environment was set up and successfully exploited taking advantage of some of the features and of the Scenario-Runner
- The behavior of the leading vehicle has been implemented in a probabilistic way allowing to acquire the several data sets needed for all the networks training.
- The all Perception System was developed, processing the Radar readings with the Classification Neural Network.
- The models both for ego vehicle (LSTM) and leading vehicle (FSM) are though and implemented.
- The Prediction and Propagation framework for the Belief Propagation is created

- The final result is satisfactory as it is possible to predict a possible crash in advance (5 seconds) and therefore increasing the safety of the vehicle.

7.2 Future Work

Obtaining a satisfying result is telling us that monitoring and performing Belief Propagation is an efficient way for the safety increase, and as a consequence there are some upgrades that can make this idea work even better:

- Training the Network for the Synthetic Radar Reading reproduction with a more diverse data set able to include more crashes as well as normal driving situations would probably increase the performance.
- The FSM could be studied more in details so that both the transitions and the assignments of the variations of velocity and azimuth can be made more humanly relatable.
- The Perception System could be enlarged including other sensors such as Cameras and LIDARs
- The scenario can be expanded to a more complex one considering more traffic participants (both vehicles and pedestrians).

APPENDICES

Appendix A

CLASSIFICATION NEURAL NETWORK RELATED CODE

The first code is used for the processing of the data for training and testing the network.

```
clear all
close all
clc

%import of the radar detetction files
location_radar_detection = '/media/disk3/cvrl/home/carla/scenario_runner/
    save_file/radar_reading_for_classifier';
ttds_radar_detetction = tabularTextDatastore(location_radar_detection,'
    FileExtensions',{'.txt'});
ttds_radar_detetction.TreatAsMissing = 'NA';
ttds_radar_detetction.MissingValue = 0;
ttds_radar_detetction.ReadSize = 'file';
ttds_radar_detetction.VariableNames = {'Vel','Alt','Azi','Dep'};

%import of files with the expected output information
location_output = '/media/disk3/cvrl/home/carla/scenario_runner/save_file/
    relative_vel_file';
ttds_out = tabularTextDatastore(location_output,'FileExtensions',{'.txt'});
ttds_out.TreatAsMissing = 'NA';
ttds_out.MissingValue = 0;
```

Appendix A (continued)

```
ttds_out.ReadSize = 'file';
ttds_out.VariableNames = {'Output'};

%import the files with ego current state
location_ego_cs = '/media/disk3/cvrl/home/carla/scenario_runner/save_file/
    ego_current_state_classifier';
ttds_ego_cs = tabularTextDatastore(location_ego_cs,'FileExtensions',{'.txt'});
ttds_ego_cs.TreatAsMissing = 'NA';
ttds_ego_cs.MissingValue = 0;
ttds_ego_cs.ReadSize = 'file';
ttds_ego_cs.VariableNames = {'Ego_CS'};

%just a quick checking of the length
if length(ttds_radar_detetction.Files) == length(ttds_out.Files) &&...
        length(ttds_radar_detetction.Files) == length(ttds_ego_cs.Files)
    N1 = length(ttds_out.Files);
end

%the following loop reads the radar detection files and the expected
%outputs files which are aquired with simulations performed with clear
%sunny weather and creates the inputs and the targets
for i = 1:N1
    file_radar_detection = read(ttds_radar_detetction);
    file_out = read(ttds_out);
    file_ego_cs = read(ttds_ego_cs);

```

Appendix A (continued)

```
43    radar_detection_mtx = file_radar_detection{:,:};
44    out_mtx = file_out{:,:};
45    ego_cs_mtx = file_ego_cs{:,:};
46
47    %if the detected points are less than 75 (which is the maximum) we set
48    %all the remaining lines to zero so that the input size will always be
49    %of the same dimension
50    if length(radar_detection_mtx) < 75
51        radar_detection_mtx(length(radar_detection_mtx)+1:75,:) = zeros(75-
      length(radar_detection_mtx),4);
52    end
53
54    for n = 1:75
55        if  abs(radar_detection_mtx(n,1)-out_mtx(1)) < 1
56            %lead_vehicle_point
57            label_mtx(n,:) = 1;
58        elseif  abs(radar_detection_mtx(n,1)-out_mtx(2)) <= 1 || sum(
      radar_detection_mtx(n,:) == zeros(1,4)) == 4
59            %background poitn
60            label_mtx(n,:) = 0;
61        else
62            %other active element point
63            label_mtx(n,:) = -1;
64        end
65    end
66
```

Appendix A (continued)

```
    %input and target and label
    x(:,i) = reshape(radar_detection_mtx',[],1);
    target(:,i) = out_mtx(1);
    label(:,i) = label_mtx;
    ego_current_state(:,i) = ego_cs_mtx;
end

%%
[rows_x, columns_x] = size(x);
[rows_label, columns_label] = size(label);

for ind_columns = 1:columns_x
    intermediate_input_mtx = reshape(x(:,ind_columns),[4,75]);
    intermediate_label_mtx = label(:,ind_columns)'; %this becomes 1x75
    intermediate_ego_cs_mtx = ego_current_state(:,ind_columns).*ones(4,75);
    intermediate_input_classifier = [intermediate_input_mtx;
   intermediate_ego_cs_mtx];
    if ind_columns ==1
        input_classifier =  intermediate_input_classifier;
        label_classifier = intermediate_label_mtx;
    else
        input_classifier = [input_classifier  intermediate_input_classifier];
        label_classifier = [label_classifier intermediate_label_mtx];
    end
end

```

Appendix A (continued)

```
save input_classifier
save label_classifier
```

The second code is the actual one used for the training and testing of the network.

```
clear all
close all
clc

load input_classifier
load label_classifier

rng('default'); %in order to guarantee repetability

%%
N_tot = size(input_classifier,2);
N_train = fix((N_tot*85)/100);%8500;
N_test = N_tot-N_train;

x_train = input_classifier(:,1:N_train);
label_train = label_classifier(:,1:N_train);
x_test = input_classifier(:,N_train+1:N_train+N_test);
label_test = label_classifier(:,N_train+1:N_train+N_test);

%%
classifier_net = feedforwardnet([300]);

```

Appendix A (continued)

```
classifier_net.performParam.regularization = 0.1; %regulariration for the mse
classifier_net.performParam.normalization = 'none';
classifier_net = init(classifier_net); %initialization of weights and biases

classifier_net.divideFcn = 'dividerand';
classifier_net.divideParam.trainRatio = 0.9;
classifier_net.divideParam.valRatio = 0.1;
classifier_net.divideParam.testRatio = 0;

classifier_net.trainFcn = 'trainrp';
classifier_net.trainParam.epochs = 10000;

classifier_net.inputs{1}.processFcns={'mapminmax','fixunknowns'};
classifier_net.outputs{2}.processFcns={};

classifier_net.layers{1}.transferFcn = 'tansig';
classifier_net.layers{2}.transferFcn = 'tansig';

classifier_net = train(classifier_net, x_train, label_train,'useGPU', 'yes','
    showResources','yes');
y_train = classifier_net(x_train, 'useGPU', 'yes','showResources','yes');
perf = perform(classifier_net, label_train, y_train);

y_test = classifier_net(x_test,'useGPU', 'yes','showResources','yes');
save classifier_net;

```

Appendix A (continued)

```
%%
right_class = 0;
for i = 1:N_test
    diff = abs(y_test(i) - label_test(i));
    if diff < 0.5
        right_class = right_class + 1;
    end
end

classifier_accuracy_test = (right_class/N_test)*100;
```

Appendix B

LSTM NEURAL NETWORK FOR EGO VEHICLE VELOCITY PREDICTION RELATED CODE

Again here the first code is for the processing of the data

```
clear all
close all
clc

%import of the throttle and brake values
location_sim1 = '/media/disk3/cvrl/home/carla/scenario_runner/save_file/
    input_for_forecasting_2';
ttds_sim1 = tabularTextDatastore(location_sim1,'FileExtensions',{'.txt'});
ttds_sim1.TreatAsMissing = 'NA';
ttds_sim1.MissingValue = 0;
ttds_sim1.ReadSize = 'file';
ttds_sim1.VariableNames = {'IN'};

N1 = length(ttds_sim1.Files);

%the following loop reads the radar detection files and the expected
%outputs files which are aquired with simulations performed with clear
%sunny weather and creates the inputs and the targets
for i = 1:N1
```

Appendix B (continued)

```
    file_throttle_brake = read(ttds_sim1);

    throttle_brake_mtx = file_throttle_brake{:,:};

    throttle_brake_current_state_input(:,i) = throttle_brake_mtx;
end

figure(1);
plot(throttle_brake_current_state_input(1,:));
title('Unfiltered v data set');

throttle_brake_current_state_input(1,:) = medfilt1(
   throttle_brake_current_state_input(1,:),10);
figure(2);
plot(throttle_brake_current_state_input(1,:));
title('Filtered v data set');
```

The second code is the actual implementation of the Network

```
clear all
close all
clc

load throttle_brake_current_state_input
rng('default');

data = throttle_brake_current_state_input(:,1:end);
```

Appendix B (continued)

```
9
10 numTimeStepsTrain = floor(0.9*size(data,2));
11
12 dataTrain = data(:,1:numTimeStepsTrain+1);
13 dataTest = data(:,numTimeStepsTrain+1:end);
14
15 mu = mean(dataTrain,2);
16 sig = std(dataTrain,0,2);
17
18 dataTrainStandardized = (dataTrain - mu) ./ sig;
19 %%
20
21 XTrain = dataTrainStandardized(:,1:end-1);
22 YTrain = dataTrainStandardized(1,2:end);
23
24 numFeatures = size(XTrain,1);
25 numResponses = size(YTrain,1);
26 numHiddenUnits = 200;
27
28 layers = [ ...
29     sequenceInputLayer(numFeatures,'Normalization','none', '
    NormalizationDimension', 'auto')
30     lstmLayer(numHiddenUnits,'OutputMode','sequence')
31     fullyConnectedLayer(50)
32     fullyConnectedLayer(numResponses)
33     regressionLayer];
```

Appendix B (continued)

```
34
35 miniBatchSize = 20;
36 options = trainingOptions('adam', ...
37     'MaxEpochs',500, ...
38     'GradientThreshold',1, ...
39     'InitialLearnRate',0.005, ...
40     'LearnRateSchedule','piecewise', ...
41     'LearnRateDropPeriod',125, ...
42     'LearnRateDropFactor',0.2, ...
43     'ResetInputNormalization', false,...
44     'Verbose',0, ...
45     'Shuffle','never',...
46     'Plots','training-progress');
47
48 lstm_forecasting = trainNetwork(XTrain,YTrain,layers,options);
49 save lstm_forecasting;
50 %%
51 %standardize the training data using the same parameters as the training
52 %data
53
54 dataTestStandardized = (dataTest - mu) ./ sig;
55 XTest = dataTestStandardized(:,1:end-1);
56
57 lstm_forecasting = predictAndUpdateState(lstm_forecasting,XTrain);
58
59 YPred = [];
```

Appendix B (continued)

```
numTimeStepsTest = size(XTest,2);
for i = 1:numTimeStepsTest
    [lstm_forecasting,YPred(:,i)] = predictAndUpdateState(lstm_forecasting,
    XTest(:,i),'ExecutionEnvironment','cpu',...
                                                        'MiniBatchSize',1)
    ;
end

YPred = sig(1).*YPred + mu(1);
YTest = dataTest(1,2:end);

right_pred = 0;
diff = abs(YPred-YTest);
for j = 1:size(diff,2)
    if diff(:,j) < 1
        right_pred = right_pred + 1;
    end
end

accuracy_test = (right_pred/length(diff))*100;

for k = 1:size(YTest,1)
    figure(k);
    subplot(2,1,1)
    plot(YTest(k,:),'b');
    hold on;
```

Appendix B (continued)

```
84      plot(YPred(k,:),'r');
85      legend('Expected','Predicted');
86      title('Comparison Between Expected and Predicted Vel');
87
88      subplot(2,1,2)
89      stem(YPred(k,:)-YTest(k,:));
90      title('VelPred-VelExpected');
91  end
```

Appendix C

FEED FORWARD PREDICTION NEURAL NETWORK FOR SYNTHTIC RADAR READING REPRODUCTION RELATED CODE

As usual this first piece of code is related to the processing of the data

```
clear all
close all
clc

%import of the current radar detetction files
location_radar_detection = '/media/disk3/cvrl/home/carla/scenario_runner/
    save_file/input_for_final_prediction_net/radar_reading';
ttds_radar_detetction = tabularTextDatastore(location_radar_detection,'
    FileExtensions',{'.txt'});
ttds_radar_detetction.TreatAsMissing = 'NA';
ttds_radar_detetction.MissingValue = 0;
ttds_radar_detetction.ReadSize = 'file';
ttds_radar_detetction.VariableNames = {'Vel','Alt','Azi','Dep'};

%import of the ego vehicle current state
location_ego_current_state = '/media/disk3/cvrl/home/carla/scenario_runner/
    save_file/input_for_final_prediction_net/ego_current_state';
ttds_ego_current_state = tabularTextDatastore(location_ego_current_state,'
    FileExtensions',{'.txt'});
```

Appendix C (continued)

```
ttds_ego_current_state.TreatAsMissing = 'NA';
ttds_ego_current_state.MissingValue = 0;
ttds_ego_current_state.ReadSize = 'file';
ttds_ego_current_state.VariableNames = {'ego_current'};

%import of the ego next state
location_ego_next_state = '/media/disk3/cvrl/home/carla/scenario_runner/
    save_file/input_for_final_prediction_net/ego_next_state';
ttds_ego_next_state = tabularTextDatastore(location_ego_next_state,'
    FileExtensions',{'.txt'});
ttds_ego_next_state.TreatAsMissing = 'NA';
ttds_ego_next_state.MissingValue = 0;
ttds_ego_next_state.ReadSize = 'file';
ttds_ego_next_state.VariableNames = {'ego_next'};

%import of the leading current state
location_leading_current_state = '/media/disk3/cvrl/home/carla/scenario_runner/
    save_file/input_for_final_prediction_net/leading_current_state';
ttds_leading_current_state = tabularTextDatastore(
    location_leading_current_state,'FileExtensions',{'.txt'});
ttds_leading_current_state.TreatAsMissing = 'NA';
ttds_leading_current_state.MissingValue = 0;
ttds_leading_current_state.ReadSize = 'file';
ttds_leading_current_state.VariableNames = {'leading_current'};

%import of the leading next state
```

Appendix C (continued)

```
location_leading_next_state = '/media/disk3/cvrl/home/carla/scenario_runner/
    save_file/input_for_final_prediction_net/leading_next_state';
ttds_leading_next_state = tabularTextDatastore(location_leading_next_state,'
    FileExtensions',{'.txt'});
ttds_leading_next_state.TreatAsMissing = 'NA';
ttds_leading_next_state.MissingValue = 0;
ttds_leading_next_state.ReadSize = 'file';
ttds_leading_next_state.VariableNames = {'leading_next'};

if length(ttds_radar_detetction.Files) == length(ttds_ego_current_state.Files)
    && ...
        length(ttds_radar_detetction.Files) == length(ttds_ego_next_state.Files
    ) && ...
        length(ttds_radar_detetction.Files) == length(
    ttds_leading_current_state.Files) && ...
        length(ttds_radar_detetction.Files) == length(ttds_leading_next_state.
    Files)
    N_files = length(ttds_radar_detetction.Files);
end

for i = 1:N_files
    file_radara_readings = read(ttds_radar_detetction);
    file_ego_current_state = read(ttds_ego_current_state);
    file_ego_next_state = read(ttds_ego_next_state);
    file_leading_current_state = read(ttds_leading_current_state);
    file_leading_next_state = read(ttds_leading_next_state);
```

Appendix C (continued)

```
58
59      radar_readings_mtx = file_radara_readings{:,:};
60      ego_current_state_mtx  = file_ego_current_state{:,:};
61      ego_next_state_mtx  = file_ego_next_state{:,:};
62      leading_current_state_mtx  = file_leading_current_state{:,:};
63      leading_next_state_mtx  = file_leading_next_state{:,:};
64
65      radar_readings(:,i) = reshape(radar_readings_mtx',[],1);
66      ego_current_state(:,i) = ego_current_state_mtx;
67      ego_next_state(:,i) = ego_next_state_mtx;
68      leading_current_state(:,i) = leading_current_state_mtx;
69      leading_next_state(:,i) = leading_next_state_mtx;
70  end
71
72  save radar_readings
73  save ego_current_state
74  save ego_next_state
75  save leading_current_state
76  save leading_next_state
```

The following code is on the other hand regarding the actual implementation of the network.

```
1  clear all
2  close all
3  clc
4
5  rng('default'); %guaranteeing reproducibility
```

Appendix C (continued)

```
6
7  load radar_readings
8  load ego_current_state
9  load ego_next_state
10 load leading_current_state
11 load leading_next_state
12
13 data = [radar_readings; ego_current_state; ego_next_state;
       leading_current_state; leading_next_state];
14
15 numTimeStepsTrain = floor(0.9*size(data,2));
16
17 dataTrain = data(:,1:numTimeStepsTrain+1);
18 dataTest = data(:,numTimeStepsTrain+1:end);
19
20 XTrain = dataTrain(:,1:end-1);
21 YTrain = dataTrain(1:300,2:end);
22
23 XTest = dataTest(:,1:end-1);
24 YTest = dataTest(1:300,2:end);
25
26 prediction_net = feedforwardnet([ 300 75 4 75 ]);
27
28 prediction_net.performParam.regularization = 0.01; %regulariration for the mse
29 prediction_net.performParam.normalization = 'standard';
30
```

Appendix C (continued)

```
prediction_net = init(prediction_net); %initialization of weights and biases

prediction_net.divideFcn = 'divideint';
prediction_net.trainFcn = 'trainrp';
prediction_net.trainParam.epochs = 1000;
prediction_net.trainParam.max_fail = 10;

prediction_net.divideParam.trainRatio = 0.9;
prediction_net.divideParam.valRatio = 0.1;
prediction_net.divideParam.testRatio = 0;

prediction_net = train(prediction_net, XTrain, YTrain, 'useGPU', 'yes','
    showResources','yes');
out_train = prediction_net(XTrain,'useGPU', 'yes','showResources','yes' );
perf = perform(prediction_net, YTrain, out_train);

YPred = prediction_net(XTest,'useGPU', 'yes','showResources','yes' );
save prediction_net;

right_pred = zeros(size(YPred,1),size(YPred,2));
diff = abs(YPred-YTest);
for j = 1:size(diff,2)
    for k = 1:size(diff,1)
        if diff(k,j) < 2
            right_pred(j) = right_pred(j) + 1;
        end
```

Appendix C (continued)

```
56      end
57      accuracy_test(j) = (right_pred(j)/size(diff,1))*100;
58  end
59
60  accuracy_mean_on_test = mean(accuracy_test);
```

Appendix D

PREDICTION FUNCTION RELATED CODE

```
class PredictionClass(object):

    def __init__(self, vehicle, world):

        self._vehicle = vehicle
        self._world = world
        self._PID_control = PIDLongitudinalController(self._vehicle, K_P=1.0, K_D=0.0, K_I=0.0, dt=0.03)
        self._control = carla.VehicleControl()

    def run_prediction(self, ego_vehicle_current_state, leading_vehicle_current_state, current_radar_reading, percepted_vel_list, leading_vehicle_current_mode, DeltaT):

            global hidden
            global cell
            normal_cruise_velocity = 45.0
            ego_next_state_list = []
            leading_next_state_list = []
            leading_vehicle_next_mode_list = []
            max_distance = 6
```

Appendix D (continued)

```
19          ego_pos = self._vehicle.get_location()
20          v = self._vehicle.get_velocity()
21          v_ms = int(1 * math.sqrt(v.x**2 + v.y**2 + v.z**2))
22
23          # CONTROL PART NEEDED FOR HAVING THE CONTROL OUTPUT WHICH LEADS TO
    THE ACCELERATION
24          leading_vehicle_relative_velocity = leading_vehicle_current_state[0]
25          leading_vehicle_azimuth = leading_vehicle_current_state[2]
26          leading_vehicle_distance = leading_vehicle_current_state[3]
27
28          leading_vehicle_velocity = v_ms + leading_vehicle_relative_velocity
29
30          if leading_vehicle_distance > max_distance:
31              self._control.throttle = self._PID_control.run_step(
    normal_cruise_velocity)
32              self._control.brake = 0.0
33          elif leading_vehicle_distance < max_distance:
34               if (leading_vehicle_velocity*3.6) >= 1.0:
35                   front_vehicle_vel = int(leading_vehicle_velocity*3.6)
36                   if self._PID_control.run_step(front_vehicle_vel) == 0.0:
37                       # if the output is zero no action inrequired
38                       self._control.throttle = self._PID_control.run_step(
    front_vehicle_vel)
39                       self._control.brake = self._PID_control.run_step(
    front_vehicle_vel)
40                   elif self._PID_control.run_step(front_vehicle_vel) > 0:
```

Appendix D (continued)

```
41                    # if the output is > 0 the car needs to speed up so the
    throttle is controlled while the brake is kept to zero
42                    self._control.throttle = self._PID_control.run_step(
    front_vehicle_vel)
43                    self._control.brake = 0.0
44                elif self._PID_control.run_step(front_vehicle_vel) < 0:
45                    # if the output is < 0 the car needs to slow down so the
    brake is controlled while the throttle is kept to zero
46                    self._control.throttle = 0.0
47                    self._control.brake = abs(self._PID_control.run_step(
    front_vehicle_vel))
48            elif 0.0 <= (leading_vehicle_velocity*3.6) < 1.0:
49                front_vehicle_vel = 0.0
50                self._control.throttle = 0.0
51                self._control.brake = abs(self._PID_control.run_step(
    front_vehicle_vel))
52        # EGO VEHICLE KINEMATIC MODEL
53        input_to_lstm_net = [v_ms, self._control.throttle, self._control.
    brake]
54
55        update_lstm, ego_next_vel = RCNN.complete_lstm_network(
    input_to_lstm_net, input_weight, recurrent_weight, bias_lstm, hidden, cell,
    mean_value, standard_dev, weight_fc1, bias_fc1, weight_fc2, bias_fc2)
56
57        update_lstm = list(update_lstm)
58        hidden = update_lstm[0]
```

Appendix D (continued)

```
59          cell = update_lstm[1]
60
61          ego_next_pos_x = 0.5*((ego_next_vel-v_ms))*(DeltaT) + v_ms*DeltaT +
    ego_pos.x
62          ego_next_pos_y = ego_pos.y
63          ego_next_pos_z = ego_pos.z
64          ego_next_state = [ego_next_vel, ego_next_pos_x, ego_next_pos_y,
    ego_next_pos_z]
65
66          ego_next_state_list.append(ego_next_state)
67
68          # predicting next mode of the leading vehicle using the Finite State
    Machine
69          leading_vehicle_next_mode = FiniteStateMachine.current_mode_checker(
    leading_vehicle_current_mode)
70          leading_vehicle_next_mode_list.append(leading_vehicle_next_mode)
71          # here I am associating to each next mode a defined velocity
    variation, whihc should be a plausibla one. Given this variation I can
    compute leading vehicle next velocity and thanks to it using the kinematic
    eqautions I can compute the next distance between ego vehicle and leading
    vehicle.
72          DeltaV = 0
73          DeltaAzimuth = 0
74          if leading_vehicle_next_mode == 'Normal Driving':
75              DeltaV = 0
76          elif leading_vehicle_next_mode == 'Accelerating':
```

Appendix D (continued)

```
77              DeltaV = 0.5
78          elif leading_vehicle_next_mode == 'Decelerating':
79              DeltaV = -0.5
80          elif leading_vehicle_next_mode == 'Stop':
81              DeltaV = -(leading_vehicle_velocity/2)
82          elif leading_vehicle_next_mode == 'Distract R':
83              DeltaAzimuth = 1
84          elif leading_vehicle_next_mode == 'Distract L':
85              DeltaAzimuth = -1
86
87          leading_next_vel = leading_vehicle_velocity + DeltaV
88          leading_next_altitude = leading_vehicle_current_state[1]
89          leading_next_azimuth = leading_vehicle_azimuth + DeltaAzimuth
90
91          leading_pos = ego_pos.x + leading_vehicle_distance
92          leading_next_pos = 0.5*DeltaV*DeltaT + leading_vehicle_velocity*
    DeltaT + leading_pos
93
94          leading_next_distance = leading_next_pos - ego_next_pos_x
95          leading_next_relative_vel = leading_next_vel - ego_next_vel
96
97          leading_next_state =  [leading_next_relative_vel,
    leading_next_altitude, leading_next_azimuth, leading_next_distance]
98
99          leading_next_state_list.append(leading_next_state)
100
```

Appendix D (continued)

```
101             # HERE I WANT TO TEST THE NETWORK THAT PREDICTS THE NEXT RADAR
    READING
102         input_prediction_net = np.append(current_radar_reading, np.array(
    ego_vehicle_current_state).reshape(1,4), axis=0)
103         input_prediction_net = np.append(input_prediction_net, np.array(
    ego_next_state).reshape(1,4), axis=0)
104         input_prediction_net = np.append(input_prediction_net, np.array(
    leading_vehicle_current_state).reshape(1,4), axis=0)
105         input_prediction_net = np.append(input_prediction_net, np.array(
    leading_next_state).reshape(1,4), axis=0)
106
107         next_radar_reading = FFNeuralNet.prediction_radar_reading_network(
    input_prediction_net, input_ymax_pred, input_ymin_pred, input_xmax_pred,
    input_xmin_pred, weight1, bias1, weight2, bias2, weight3, bias3, weight4,
    bias4, weight5, bias5, output_ymax_pred, output_ymin_pred, output_xmax_pred,
     output_xmin_pred)
108
109         predicion_net_lead_vehicle_state =  PredictionClass(self._vehicle,
    self._world).run_propagation(next_radar_reading, ego_next_state)
110
111         analytical_lead_vehicle_state = PredictionClass(self._vehicle, self.
    _world).analytical_model(ego_vehicle_current_state,
    leading_vehicle_current_state, current_radar_reading, ego_next_state,
    leading_next_state)
112
```

Appendix D (continued)

```
113         return [ego_next_state, leading_next_state, next_radar_reading,
    leading_vehicle_next_mode, predicion_net_lead_vehicle_state,
    analytical_lead_vehicle_state]
114
115     def run_propagation(self,next_radar_reading, ego_next_state):
116
117         setted_ego_vel = 45.0
118         max_d = 6
119         leading_vehicle_points_list = []
120         background_points_list = []
121         active_point_list = []
122         v = self._vehicle.get_velocity()
123         v_ms = int(1 * math.sqrt(v.x**2 + v.y**2 + v.z**2))
124
125         for row in next_radar_reading:
126             classification_input = np.append(row[:,None], np.array(
    ego_next_state).reshape(-1,1), axis=0)
127             classifier = FFNeuralNet.classification_network(
    classification_input,input_ymax_class, input_ymin_class, input_xmax_class,
    input_xmin_class, input_w_mtx_class, input_b_vec_class, layer1_w_mtx_class,
    layer1_b_vec_class)
128
129             if -0.5 < classifier < 0.5:
130                 background_points_list.append(row)
131             elif 0.5 < classifier < 1:
132                 leading_vehicle_points_list.append(row)
```

Appendix D (continued)

```
133             elif -1 < classifier < -0.5:
134                 active_point_list.append(row)
135 
136         if len(leading_vehicle_points_list) != 0:
137             cnt = 0
138             for row in leading_vehicle_points_list:
139                 dist = row[3]
140                 if cnt == 0:
141                     min_dist = row[3]
142                     relative_vel = row[0]
143                     azimuth = row[2]
144                     leading_vehicle_current_state = row
145                 else:
146                     if dist < min_dist and dist != 0:
147                         min_dist = dist
148                         relative_vel = row[0]
149                         azimuth = row[2]
150                         leading_vehicle_current_state = row
151                 cnt = cnt + 1
152             return leading_vehicle_current_state
153 
154     def analytical_model(self, ego_vehicle_current_state,
    leading_vehicle_current_state, current_radar_reading, ego_vehicle_next_state
    , leading_next_state):
155 
156         leading_vehicle_points_list = []
```

Appendix D (continued)

```
157         background_points_list = []
158         active_point_list = []
159         analytical_radar_reading = []
160         next_background_points = []
161         next_leading_vehicle_points = []
162
163         for row in current_radar_reading:
164           classification_input = np.concatenate((row,
    ego_vehicle_current_state), axis=0)
165           classifier = FFNeuralNet.classification_network(
    classification_input,input_ymax_class, input_ymin_class, input_xmax_class,
    input_xmin_class, input_w_mtx_class, input_b_vec_class, layer1_w_mtx_class,
    layer1_b_vec_class)
166
167           if -0.5 < classifier < 0.5:
168              background_points_list.append(row)
169              new_vel = -ego_vehicle_next_state[0]
170              new_alt = ego_vehicle_current_state[1]
171              new_azi = ego_vehicle_current_state[2]
172              new_dist = ego_vehicle_current_state[3]
173              new_row = [new_vel, new_alt, new_azi, new_dist]
174              next_background_points.append(new_row)
175              analytical_radar_reading.append(new_row)
176           elif 0.5 < classifier < 1:
177              leading_vehicle_points_list.append(row)
178              new_row = leading_next_state
```

Appendix D (continued)

```
179                 next_leading_vehicle_points.append(new_row)
180                 analytical_radar_reading.append(new_row)
181             elif -1 < classifier < -0.5:
182                 active_point_list.append(row)
183 
184         if len(next_leading_vehicle_points) != 0:
185             cnt = 0
186             for row in next_leading_vehicle_points:
187                 dist = row[3]
188                 if cnt == 0:
189                     min_dist = row[3]
190                     relative_vel = row[0]
191                     azimuth = row[2]
192                     analytical_lead_vehicle_state = row
193                 else:
194                     if dist < min_dist and dist != 0:
195                         min_dist = dist
196                         relative_vel = row[0]
197                         azimuth = row[2]
198                         analytical_lead_vehicle_state = row
199                 cnt = cnt + 1
200             return analytical_lead_vehicle_state
```

Appendix E

OVERALL CONTROL LOOP RELATED CODE

```
1    def _parse_PID_control_NNperception(self, world):
2
3        max_d = 6 # this is the maximum distance between ego vehicle and lead
    vehicle in meters
4        setted_ego_vel = 45.0 # this is the maximum velocity of the ego vehicle
     to be kept by it if the distance is greater than the max_d setted above
5        global points
6        global starting_flag
7        global percepted_vel_list
8        global loop_cnt
9        global reference_time
10       global mode_changing_counter
11       global leading_vehicle_mode_list
12       global save_image_index
13       global even_numbers_list
14       global start_time
15       global DeltaT
16       global time_list
17       global simulation_start_time
18       global actual_trigger_time
19       front_vehicle_vel = 0.0
```

Appendix E (continued)

```
20          v = world.vehicle.get_velocity() # current vel of ego vehicle in m/ms
21          kmh = int(3.6 * math.sqrt(v.x**2 + v.y**2 + v.z**2)) # current vel of
    ego vehicle in km/h
22          t = world.vehicle.get_transform()
23          vehicles = world.world.get_actors().filter('vehicle.*') # list of
    vehicles in the simulation
24          distance = lambda l: math.sqrt((l.x - t.location.x)**2 + (l.y - t.
    location.y)**2 + (l.z - t.location.z)**2) # method to compute the distance
    between other vehicle and ego vehicle
25          mutal_d_form_vehcile = [(distance(x.get_location()), x) for x in
    vehicles if x.id != world.vehicle.id] # distance between ego vehcile and
    leading vehicle
26          self._PID_control = PIDLongitudinalController(world.vehicle, K_P=1.0,
    K_D=0.0, K_I=0.0, dt=0.03)
27
28          v_ego_ms = kmh/3.6
29          ego_pos = world.vehicle.get_location()
30          input_for_forecasting_list = [v_ego_ms]
31
32          ego_current_state = [v_ego_ms, ego_pos.x, ego_pos.y, ego_pos.z]
33
34 #with the following method I am making all the input matrices to be 75x4
    otherwise they could have a lower number of rows depending on the points
    detected from the radar
35          points_mtx = np.array(points)
36          if len(points_mtx) < 75:
```

Appendix E (continued)

```
37          new_rows = 75-len(points_mtx)
38          append_mtx = np.zeros((new_rows,4), dtype=np.float)
39          points_mtx = np.append(points_mtx, append_mtx, axis=0) # this is the
     75x4 current radar reading
40
41          #current_radar_reading = points_mtx.flatten('F').reshape(-1, 1) #here I
     am flattening the points matrix and making it a column vector that can be
    feed as the input to the network
42
43          # here I use the classification neural network because I want to
    classify the current radar readings into points belonging to the leading
    vehicle and points belonging to the background. If the output of the network
     is 0 => the point belongs to the background, while if the output is 1 =>
    the points belongs to the leading vehicle.
44          background_points_list = []
45          leading_vehicle_points_list = []
46          active_point_list = []
47          for row in points_mtx:
48              classification_input = np.concatenate((row, ego_current_state),
    axis=0)
49              classifier = FFNeuralNet.classification_network(
    classification_input,input_ymax_class, input_ymin_class, input_xmax_class,
    input_xmin_class, input_w_mtx_class, input_b_vec_class, layer1_w_mtx_class,
    layer1_b_vec_class)
50
51              if -0.5 < classifier < 0.5:
```

Appendix E (continued)

```
52                background_points_list.append(row)
53            elif 0.5 < classifier < 1:
54                leading_vehicle_points_list.append(row)
55            elif -1 < classifier < -0.5:
56                active_point_list.append(row)
57
58 # Following there is the control part. First of all I consider only the points
     that are classified as "leading vehicle". Besides I choose among those
     points the one that has the minor distance, because in theory is the fist
     one my ego vehicle will touch in case of a crash. I take from that point the
      distance and the relative velocity and I use them in the control.
59 # First of all I make a starting process which will make the simulation start
     and will bring the ego vehicle close enough to the leading vehicle, making
     the ego vehicle reach a velocity between 10 and 12 km/h. Besides when the
     distance is bigger that a setted value "max_d" the ego vehicle responds to a
      normal cruise control at a setted velocity, while is the distance is lower
     than max_d an adaptive cruise control comes in, and the ego vehicle tries to
      reach the velocity of the leading vehicle which is return from the radar
     readings. Besides cheking if the previous values of velocity are somehow
     similar to the current value I am trying to get rid of possible misreadings
     and misclassifications.
60
61        throttle_brake_list = []
62        acceleration_list = []
63
64        if len(leading_vehicle_points_list) != 0:
```

Appendix E (continued)

```
65          cnt = 0
66          for row in leading_vehicle_points_list:
67              dist = row[3]
68              if cnt == 0:
69                  min_dist = row[3]
70                  relative_vel = row[0]
71                  azimuth = row[2]
72                  leading_vehicle_current_state = row
73              else:
74                  if dist < min_dist and dist != 0:
75                      min_dist = dist
76                      relative_vel = row[0]
77                      azimuth = row[2]
78                      leading_vehicle_current_state = row
79              cnt = cnt + 1
80
81          absolute_vel = 3.6*(relative_vel + v_ego_ms) #kmh
82
83          distance_list.append(min_dist)
84          percepted_vel_list.append(absolute_vel)
85          loop_cnt = loop_cnt + 1
86
87          # FSM FOR NEXT MODE (PROBABILISTIC MODEL OF LEADING VEHICLE)
88          # Before proceeding with the control part I need to establish which
    is the current mode of the leading vehicle among: Normal Driving,
    Accelerating, Decelerating, Stop, Distract Left and Distract Right. In order
```

Appendix E (continued)

```
 to do I am going to compare the absolute velocity at this instant with the
absolute velocity in the previous instant.
89          if len(percepted_vel_list) == 1:
90              leading_vehicle_current_mode = "Accelerating"
91              leading_vehicle_mode_list.append(leading_vehicle_current_mode)
92              reference_time = time.time()
93              Tmin_list.append(2)
94              mode_changing_counter = 0
95          elif (time.time() - reference_time) >= Tmin_list[
mode_changing_counter]:
96              if percepted_vel_list[loop_cnt-1]-percepted_vel_list[loop_cnt-2]
< -1:
97                  leading_vehicle_current_mode = "Decelerating"
98                  leading_vehicle_mode_list.append(leading_vehicle_current_mode)
99                  reference_time = time.time()
100                 Tmin_list.append(2)
101                 mode_changing_counter = mode_changing_counter + 1
102             elif percepted_vel_list[loop_cnt-1]-percepted_vel_list[loop_cnt
-2] > 1:
103                 leading_vehicle_current_mode = "Accelerating"
104                 leading_vehicle_mode_list.append(leading_vehicle_current_mode)
105                 reference_time = time.time()
106                 Tmin_list.append(2)
107                 mode_changing_counter = mode_changing_counter + 1
108             elif -0.5 < percepted_vel_list[loop_cnt-1] < 0.5:
109                 leading_vehicle_current_mode = "Stop"
```

Appendix E (continued)

```
110                 leading_vehicle_mode_list.append(leading_vehicle_current_mode)
111                 reference_time = time.time()
112                 Tmin_list.append(1)
113                 mode_changing_counter = mode_changing_counter + 1
114             elif leading_vehicle_current_state[2] > 1:
115                 leading_vehicle_current_mode = "Distract R"
116                 leading_vehicle_mode_list.append(leading_vehicle_current_mode)
117                 reference_time = time.time()
118                 Tmin_list.append(1)
119                 mode_changing_counter = mode_changing_counter + 1
120             elif  leading_vehicle_current_state[2] < -1:
121                 leading_vehicle_current_mude = "Distract L"
122                 leading_vehicle_mode_list.append(leading_vehicle_current_mode)
123                 reference_time = time.time()
124                 Tmin_list.append(1)
125                 mode_changing_counter = mode_changing_counter + 1
126             else:
127                 leading_vehicle_current_mode = "Normal Driving"
128                 leading_vehicle_mode_list.append(leading_vehicle_current_mode)
129                 reference_time = time.time()
130                 Tmin_list.append(3)
131                 mode_changing_counter = mode_changing_counter + 1
132         elif (time.time() - reference_time) < Tmin_list[
    mode_changing_counter]:
133             leading_vehicle_current_mode = leading_vehicle_mode_list[loop_cnt
    -2]
```

Appendix E (continued)

```
134             leading_vehicle_mode_list.append(leading_vehicle_current_mode)
135
136         # here I am checking how much time it takes to execute one iteration
    of the all function "Keyboard Control"
137         spot = time.time()
138         time_list.append(spot)
139         if len(time_list) > 1:
140             delta_t_execution = time_list[loop_cnt-1] - time_list[loop_cnt-2]
141         else:
142             delta_t_execution = time_list[loop_cnt-1] - simulation_start_time
143
144         new_ego_state_list = []
145         new_leading_state_list = []
146         estimated_leading_state_list = []
147         analytical_lead_state_list = []
148         prediction_trigger_time = time.time()
149         if 75.0 <= (prediction_trigger_time-simulation_start_time) <= 76.5:
150             actual_trigger_time = time.time()
151             for n in range(10):
152                 if n == 0:
153                     ego_next_state, leading_next_state, next_radar_reading,
    new_mode, estimated_leading_vehicle_netx_state,
    analytical_lead_vehicle_state = PredictionClass(world.vehicle, world).
    run_prediction(ego_current_state, leading_vehicle_current_state, points_mtx,
     percepted_vel_list, leading_vehicle_current_mode, delta_t_execution)
154                     new_ego_state = ego_next_state
```

Appendix E (continued)

```
155                     new_leading_state = leading_next_state
156                     new_radar_reading = next_radar_reading
157                     new_ego_state_list.append(new_ego_state)
158                     new_leading_state_list.append(new_leading_state)
159                     estimated_leading_state_list.append(
    estimated_leading_vehicle_netx_state)
160                     analytical_lead_state_list.append(
    analytical_lead_vehicle_state)
161                     print('Saving at', ego_pos.x)
162                 else:
163                     if estimated_leading_vehicle_netx_state is None:
164                         new_ego_state, new_leading_state, new_radar_reading,
    new_mode, estimated_leading_vehicle_netx_state,
    analytical_lead_vehicle_state = PredictionClass(world.vehicle, world).
    run_prediction(new_ego_state, new_leading_state, new_radar_reading,
    percepted_vel_list, new_mode, delta_t_execution)
165                         new_ego_state_list.append(new_ego_state)
166                         new_leading_state_list.append(new_leading_state)
167                         estimated_leading_state_list.append(
    estimated_leading_vehicle_netx_state)
168                         analytical_lead_state_list.append(
    analytical_lead_vehicle_state)
169                     else:
170                         new_ego_state, new_leading_state, new_radar_reading,
    new_mode, estimated_leading_vehicle_netx_state,
    analytical_lead_vehicle_state = PredictionClass(world.vehicle, world).
```

Appendix E (continued)

```
run_prediction(new_ego_state, estimated_leading_vehicle_netx_state,
new_radar_reading, percepted_vel_list, new_mode, delta_t_execution)
                    new_ego_state_list.append(new_ego_state)
                    new_leading_state_list.append(new_leading_state)
                    estimated_leading_state_list.append(
estimated_leading_vehicle_netx_state)
                    analytical_lead_state_list.append(
analytical_lead_vehicle_state)
                print('Saving at', ego_pos.x)

            if len(new_ego_state_list) != 0:
                new_index = len(os.listdir('/media/disk3/cvrl/home/carla/
scenario_runner/save_file/testing_prediction_net/predicted_states/sim2/
ego_states'))
                points_file = open("/media/disk3/cvrl/home/carla/
scenario_runner/save_file/testing_prediction_net/predicted_states/sim2/
ego_states/ego_prop%s.txt" % new_index, "w")
                for row in new_ego_state_list:
                    np.savetxt(points_file, [row])
                points_file.close()
                new_index = len(os.listdir('/media/disk3/cvrl/home/carla/
scenario_runner/save_file/testing_prediction_net/predicted_states/sim2/
leading_states'))
                points_file = open("/media/disk3/cvrl/home/carla/
scenario_runner/save_file/testing_prediction_net/predicted_states/sim2/
leading_states/leading_prop%s.txt" % new_index, "w")
```

Appendix E (continued)

```
185                 for row in new_leading_state_list:
186                     np.savetxt(points_file, [row])
187                 points_file.close()
188                 new_index = len(os.listdir('/media/disk3/cvrl/home/carla/
    scenario_runner/save_file/testing_prediction_net/predicted_states/sim2/
    estimated_lead_states'))
189                 points_file = open("/media/disk3/cvrl/home/carla/
    scenario_runner/save_file/testing_prediction_net/predicted_states/sim2/
    estimated_lead_states/leading_prop%s.txt" % new_index, "w")
190                 for row in estimated_leading_state_list:
191                     np.savetxt(points_file, [row])
192                 points_file.close()
193                 new_index = len(os.listdir('/media/disk3/cvrl/home/carla/
    scenario_runner/save_file/testing_prediction_net/predicted_states/sim2/
    analytical_lead_state'))
194                 points_file = open("/media/disk3/cvrl/home/carla/
    scenario_runner/save_file/testing_prediction_net/predicted_states/sim2/
    analytical_lead_state/leading_prop%s.txt" % new_index, "w")
195                 for row in analytical_lead_state_list:
196                     np.savetxt(points_file, [row])
197                 points_file.close()
198 
199         else:
200             for n in range(10):
201                 if n == 0:
```

Appendix E (continued)

```
202                     ego_next_state, leading_next_state, next_radar_reading,
new_mode, estimated_leading_vehicle_netx_state,
analytical_lead_vehicle_state = PredictionClass(world.vehicle, world).
run_prediction(ego_current_state, leading_vehicle_current_state, points_mtx,
 percepted_vel_list, leading_vehicle_current_mode, delta_t_execution)
203                     new_ego_state = ego_next_state
204                     new_leading_state = leading_next_state
205                     new_radar_reading = next_radar_reading
206                     new_ego_state_list.append(new_ego_state)
207                     new_leading_state_list.append(new_leading_state)
208                     estimated_leading_state_list.append(
estimated_leading_vehicle_netx_state)
209                     analytical_lead_state_list.append(
analytical_lead_vehicle_state)
210
211                 else:
212                     if estimated_leading_vehicle_netx_state is None:
213                         new_ego_state, new_leading_state, new_radar_reading,
new_mode, estimated_leading_vehicle_netx_state,
analytical_lead_vehicle_state = PredictionClass(world.vehicle, world).
run_prediction(new_ego_state, new_leading_state, new_radar_reading,
percepted_vel_list, new_mode, delta_t_execution)
214                         new_ego_state_list.append(new_ego_state)
215                         new_leading_state_list.append(new_leading_state)
216                         estimated_leading_state_list.append(
estimated_leading_vehicle_netx_state)
```

Appendix E (continued)

```
217                     analytical_lead_state_list.append(
analytical_lead_vehicle_state)
218                 else:
219                     new_ego_state, new_leading_state, new_radar_reading,
new_mode, estimated_leading_vehicle_netx_state,
analytical_lead_vehicle_state = PredictionClass(world.vehicle, world).
run_prediction(new_ego_state, estimated_leading_vehicle_netx_state,
new_radar_reading, percepted_vel_list, new_mode, delta_t_execution)
220                     new_ego_state_list.append(new_ego_state)
221                     new_leading_state_list.append(new_leading_state)
222                     estimated_leading_state_list.append(
estimated_leading_vehicle_netx_state)
223                     analytical_lead_state_list.append(
analytical_lead_vehicle_state)
224
225         # Here starts the real control loop. in which the PID controller is
taken into account and where we are switching between normal cruise control
to adaptive cruise control
226         if (kmh <= 1 and starting_flag == 0):
227             self._control.throttle = self._PID_control.run_step(
setted_ego_vel)
228             self._control.brake = 0.0
229             if kmh > 0.2:
230                 starting_flag = 1
231         else:
232             if min_dist > max_d:
```

Appendix E (continued)

```
233                 self._control.throttle = np.clip(self._PID_control.run_step(
    setted_ego_vel), 0.0, 1.0)
234                 self._control.brake = 0.0
235             elif min_dist < max_d:
236                 if absolute_vel >= 1.0:
237                     front_vehicle_vel = absolute_vel
238                     if self._PID_control.run_step(front_vehicle_vel) == 0.0:
239                         # if the output is zero no action inrequired
240                         self._control.throttle = self._PID_control.run_step(
    front_vehicle_vel)
241                         self._control.brake = self._PID_control.run_step(
    front_vehicle_vel)
242                     elif self._PID_control.run_step(front_vehicle_vel) > 0:
243                         # if the output is > 0 the car needs to speed up so
    the throttle is controlled while the brake is kept to zero
244                         self._control.throttle = self._PID_control.run_step(
    front_vehicle_vel)
245                         self._control.brake = 0.0
246                     elif self._PID_control.run_step(front_vehicle_vel) < 0:
247                         # if the output is < 0 the car needs to slow down so
    the brake is controlled while the throttle is kept to zero
248                         self._control.throttle = 0.0
249                         self._control.brake = abs(self._PID_control.run_step(
    front_vehicle_vel))
250                 elif 0.0 <= absolute_vel < 1.0:
251                     front_vehicle_vel = 0.0
```

Appendix E (continued)

```
252                         self._control.throttle = 0.0
253                         self._control.brake = abs(self._PID_control.run_step(
    front_vehicle_vel))
```

Appendix F

HOW TO RUN CARLA AND SCENARIO-RUNNER

This Appendix is explaining how to reproduce the simulation that I created for this work.

1. Open the first tab in the terminal. Go to the CARLA folder and run:

$$./CarlaUE4.sh$$

2. Open a second tab in the terminal. Go to the Scenario-Runner folder and run:

$$python3\ scenario\_runner.py\ --scenario\ MyScenario2\_1\ --reloadWorld$$

   Pay attention to the fact that if any other scenario wants to be run, it is simply necessary to change the $MyScenario2\_1$ into whatever is desired. This is the file that sets the scenario settings and controls the other actors behaviors.

3. Open a third tab in the terminal. Again in Scenario-Runner folder run:

$$python3\ PID\_control\_class\_pred.py$$

   This is the file that controls the ego vehicle. There are others and any of them can be modified.

Appendix G

IMPLEMENTATION OF PID LONGITUDINAL CONTROLLER IN PYTHON

```
class PIDLongitudinalController(object):
    """
    PIDLongitudinalController implements longitudinal control using a PID.
    """

    def __init__(self, vehicle, K_P=1.0, K_D=0.0, K_I=0.0, dt=0.03):
        """
        :param vehicle: actor to apply to local planner logic onto
        :param K_P: Proportional term
        :param K_D: Differential term
        :param K_I: Integral term
        :param dt: time differential in seconds
        """
        self._vehicle = vehicle
        self._K_P = K_P
        self._K_D = K_D
        self._K_I = K_I
        self._dt = dt
        self._e_buffer = deque(maxlen=30)

```

Appendix G (continued)

```
    def run_step(self, target_speed, debug=False):
        """
        Execute one step of longitudinal control to reach a given target speed.
        :param target_speed: target speed in Km/h
        :return: throttle/brake control in the range [-1, 1]
        """
        current_speed = get_speed(self._vehicle)

        if debug:
            print('Current speed = {}'.format(current_speed))

        return self._pid_control(target_speed, current_speed)

    def _pid_control(self, target_speed, current_speed):
        """
        Estimate the throttle/brake of the vehicle based on the PID equations
        :param target_speed:  target speed in Km/h
        :param current_speed: current speed of the vehicle in Km/h
        :return: throttle control in the range [0, 1]
        :return: brake control in the range [-1, 0]. When applying the brake
    remember to consider
         the absolute value so that the brake is in [0, 1] as expected in Carla
     control
        """
        _e = (target_speed - current_speed)
        self._e_buffer.append(_e)
```

Appendix G (continued)

```
45
46          if len(self._e_buffer) >= 2:
47              _de = (self._e_buffer[-1] - self._e_buffer[-2]) / self._dt
48              _ie = sum(self._e_buffer) * self._dt
49          else:
50              _de = 0.0
51              _ie = 0.0
52          # the return function has been modified so that it returns value
    between -1 and 1. In such a way if the output is > 0 the throttle is going
    to be controlled, while if it is < 0 the brake is going to be controlled
53          return np.clip((self._K_P * _e) + (self._K_D * _de / self._dt) + (self.
    _K_I * _ie * self._dt), -1.0, 1.0)
```

Listing G.1: Longitudinal PID Controller.

It is notable how the function returned at the last line of the code above is exactly the expression reported in Equation 4.1

Appendix H

IMPLEMENTATION OF THE RADAR IN PYTHON

```
class RadarSensor(object):
    def __init__(self, parent_actor, hud):
        self.sensor = None
        self._parent = parent_actor
        self.velocity_range = 7.5 # m/s
        radar_world = self._parent.get_world()
        self.debug = radar_world.debug
        bp = radar_world.get_blueprint_library().find('sensor.other.radar')
        bp.set_attribute('horizontal_fov', str(35))# original str(35)
        bp.set_attribute('vertical_fov', str(20))# original str(20)
        self.sensor = radar_world.spawn_actor(
            bp,
            carla.Transform(
                carla.Location(x=2.8, z=1), # original z=1.0
                carla.Rotation(pitch=0.6)), # original pitch=5
            attach_to=self._parent)
        # We need a weak reference to self to avoid circular reference.
        weak_self = weakref.ref(self)
        self.sensor.listen(lambda radar_data: RadarSensor._Radar_callback(
weak_self, radar_data))

```

Appendix H (continued)

```
21          def toggle_radar(self):
22              if self.sensor is None:
23                  self.radar_sensor = RadarSensor(self.player)
24                  #self.radar_sensor = RadarSensor(display)
25              elif self.sensor is not None:
26                  self.sensor.destroy()
27                  self.sensor = None
28
29          @staticmethod
30          def _Radar_callback(weak_self, radar_data):
31              global points
32              self = weak_self()
33              if not self:
34                  return
35              # To get a numpy [[vel, altitude, azimuth, depth],...[,,,]]:
36              points = np.frombuffer(radar_data.raw_data, dtype=np.dtype('f4'))
37              points = np.reshape(points, (len(radar_data), 4))
38
39              current_rot = radar_data.transform.rotation
40              for detect in radar_data:
41                  azi = math.degrees(detect.azimuth)
42                  alt = math.degrees(detect.altitude)
43                  # The 0.25 adjusts a bit the distance so the dots can
44                  # be properly seen
45                  fw_vec = carla.Vector3D(x=detect.depth - 0.25)
46                  carla.Transform(
```

Appendix H (continued)

```
                    carla.Location(),
                    carla.Rotation(
                        pitch=current_rot.pitch + alt,
                        yaw=current_rot.yaw + azi,
                        roll=current_rot.roll)).transform(fw_vec)

                def clamp(min_v, max_v, value):
                    return max(min_v, min(value, max_v))

                norm_velocity = detect.velocity / self.velocity_range
                #range[-1, 1]
                r = int(clamp(0.0, 1.0, 1.0 - norm_velocity) * 255.0)
                g = int(clamp(0.0, 1.0, 1.0 - abs(norm_velocity)) * 255.0)
                b = int(abs(clamp(- 1.0, 0.0, - 1.0 - norm_velocity)) * 255.0)
                self.debug.draw_point(
                    radar_data.transform.location + fw_vec,
                    size=0.075,
                    life_time=0.06,
                    persistent_lines=False
```

Listing H.1: Radar Implementation in CARLA.

Appendix I

LEADING VEHICLE RANDOM BEHAVIOR IMPLEMENTATION

Algorithm 1: Leading Vehicle Random Behavior Pseudo Code

```
behavior variable = np.random.randn()
q995 = 2.575 (1% probability)
q990 = 2.325 (1% probability)
q980 = 2.055 (2% probability)
q970 = 1.882 (2% probability)
q960 = 1.751 (2% probability)
q950 = 1.645 (2% probability)
q940 = 1.554 (2% probability)
q930 = 1.476 (2% probability)
if behavior variable < −q995 or behavior variable > q995 then
    Starting
    Reaching v1 = 11m/s (39.6km/h)
    Suddenly stopping at 120m from the intersection (middle of the simulation)
else if behavior variable < −q990 or behavior variable > q990 then
    Starting
    Reaching v5 = 8m/s (28.8km/h)
    Suddenly stopping at 120m from the intersection (middle of the simulation)
else if behavior variable < −q980 or behavior variable > q980 then
    Starting
    Reaching v1 = 11m/s (39.6km/h)
    Accelerating from v1 to v2 = 11.5m/s (41.6km/h) at 120m from the intersection
      (middle of the simulation)
    Keeping v2 until the car stops at the next intersection
```

Appendix I (continued)

Algorithm 2: Leading Vehicle Random Behavior Pseudo Code - Continue(1)

```
else if behavior variable < −q970 or behavior variable > q970 then
    Starting
    Reaching v1 = 11m/s (39.6km/h)
    Braking from v1 to v3 = 10.5m/s (37.8km/h) at 120m from the intersection
      (middle of the simulation)
    Keeping v3 until the car stops at the next intersection
else if behavior variable < −q960 or behavior variable > q960 then
    Starting
    Reaching v5 = 8m/s (28.8km/h)
    Accelerating from v5 to v6 = 9m/s (32.4km/h) at 120m from the intersection
      (middle of the simulation)
    Keeping v6 until the car stops at the next intersection
else if behavior variable < −q950 or behavior variable > q950 then
    Starting
    Reaching v5 = 8m/s (28.8km/h)
    Braking from v5 to v7 = 7m/s (25.2km/h) at 120m from the intersection (middle of
      the simulation)
    Keeping v7 until the car stops at the next intersection
else if behavior variable < −q940 or behavior variable > q940 then
    Starting
    Reaching v1 = 11m/s (39.6km/h)
    Steering to the right keeping v1 at 120m from the intersection (middle of the
      simulation)
    Keeping v1 until the car stops at the next intersection
```

Appendix I (continued)

Algorithm 3: Leading Vehicle Random Behavior Pseudo Code - Continue(2)

```
else if behavior variable < −q960 or behavior variable > q960 then
    Starting
    Reaching v1 = 11m/s (39.6km/h)
    Steering to the left keeping v1 at 120m from the intersection (middle of the
     simulation)
    Keeping v1 until the car stops at the next intersection
else
    Starting
    Reaching v0 = 10m/s (36km/h) and keeping v0 until the car stops at the next
     intersection
end
```

The whole code is here omitted because it is pretty intuitive but also very long, and is therefore replaced with the correspondent pseudo code.

Appendix J

FSM IMPLEMENTATION CODE

```
1
2  class FiniteStateMachine(object):
3
4      def transition_from_normal_driving(transition_variable):
5          p_stop = 2.323
6          p_dr = 1.881
7          p_dl = 1.64
8          p_acc = 1.28
9          p_dec = 1.035
10         if transition_variable < -p_stop or transition_variable > p_stop:
11             next_mode = 'Stop'
12         elif transition_variable < -p_dr or transition_variable > p_dr:
13             next_mode = 'Distract R'
14         elif transition_variable < -p_dl or transition_variable > p_dl:
15             next_mode = 'Distract L'
16         elif transition_variable < -p_acc or transition_variable > p_acc:
17             next_mode = 'Accelerating'
18         elif transition_variable < -p_dec or transition_variable > p_dec:
19             next_mode = 'Decelerating'
20         else:
21             next_mode = 'Normal Driving'
```

Appendix J (continued)

```
22          return next_mode
23
24      def transition_from_accelerating(transition_variable):
25          p_dr = 1.96
26          p_dl = 1.64
27          p_acc = 1.03
28          if transition_variable < -p_dr or transition_variable > p_dr:
29              next_mode = 'Distract R'
30          elif transition_variable < -p_dl or transition_variable > p_dl:
31              next_mode = 'Distract L'
32          elif transition_variable < -p_acc or transition_variable > p_acc:
33              next_mode = 'Accelerating'
34          else:
35              next_mode = 'Normal Driving'
36          return next_mode
37
38      def transition_from_decelerating(transition_variable):
39          p_stop = 1.64
40          p_dec = 1.03
41          if transition_variable < -p_stop or transition_variable > p_stop:
42              next_mode = 'Stop'
43          elif transition_variable < -p_dec or transition_variable > p_dec:
44              next_mode = 'Decelerating'
45          else:
46              next_mode = 'Normal Driving'
47          return next_mode
```

Appendix J (continued)

```
48
49      def transition_from_stop(transition_variable):
50          p_stop = 1.03
51          if transition_variable < -p_stop or transition_variable > p_stop:
52              next_mode = 'Stop'
53          else:
54              next_mode = 'Normal Driving'
55          return next_mode
56
57      def transition_from_distract_left(transition_variable):
58          p_dec = 1.64
59          p_dl = 1.03
60          if transition_variable < -p_dec or transition_variable > p_dec:
61              next_mode = "Decelerating"
62          elif transition_variable < -p_dl or transition_variable > p_dl:
63              next_mode = "Distract L"
64          else:
65              next_mode = 'Normal Driving'
66          return next_mode
67
68      def transition_from_distract_right(transition_variable):
69          p_dec = 1.64
70          p_dr = 1.03
71          if transition_variable < -p_dec or transition_variable > p_dec:
72              next_mode = "Decelerating"
73          elif transition_variable < -p_dr or transition_variable > p_dr:
```

Appendix J (continued)

```
74              next_mode = "Distract R"
75          else:
76              next_mode = 'Normal Driving'
77          return next_mode
78
79      def current_mode_checker(string):
80          switcher = {
81              "Normal Driving": FiniteStateMachine.
    transition_from_normal_driving(np.random.randn(1, 1)),
82              "Accelerating": FiniteStateMachine.transition_from_accelerating(np
    .random.randn(1, 1)),
83              "Decelerating": FiniteStateMachine.transition_from_decelerating(np
    .random.randn(1, 1)),
84              "Stop": FiniteStateMachine.transition_from_stop(np.random.randn(1,
     1)),
85              "Distract L": FiniteStateMachine.transition_from_distract_left(np.
    random.randn(1, 1)),
86              "Distract R": FiniteStateMachine.transition_from_distract_right(np
    .random.randn(1, 1)),
87          }
88          return switcher.get(string, "nothing")
```

Listing J.1: Python implementation of the FSM with the reproduction of a simple switch-case structure.

CITED LITERATURE

1. Sistla, A. P., Zefran, M., and Feng, Y.: Monitorability of stochastic dynamical systems. In Computer Aided Verification (CAV), 2011.

2. International, S.: Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles. `https://www.sae.org/standards/content/j3016_201806/`, [online; accessed 6/15/20].

3. Dividend, L.: Self-driving car technology: How do self-driving cars work? `https://www.landmarkdividend.com/self-driving-car/`, [online; accessed 6/15/20].

4. Koschi, M. and Althoff, M.: Spot: A tool for set-based prediction of traffic participants. In 2017 IEEE Intelligent Vehicles Symposium (IV), pages 1686–1693, July 2017.

5. Gindele, T., Brechtel, S., and Dillmann, R.: A probabilistic model for estimating driver behaviors and vehicle trajectories in traffic environments. In 2010 13th International IEEE Annual Conference on Intelligent Transportation Systems, September 2010.

6. Chandra, R., Bhattacharya, U., Randhavane, T., Bera, A., and Manocha, D.: Roadtrack: Tracking road agents in dense and heterogeneous environments. arXiv preprint arXiv:1906.10712, 2019.

7. Hu, Y., Zhan, W., and Tomizuka, M.: Probabilistic prediction of vehicle semantic intention and motion. In 2018 IEEE Intelligent Vehicles Symposium (IV), pages 307–313, October 2018.

8. Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., and Koltun, V.: CARLA: An open urban driving simulator. In Proceedings of the 1st Annual Conference on Robot Learning, pages 1–16, 2017.

9. Shakouri, P., Ordys, A., Laila, D. S., and , M. A.: Adaptive cruise control system: Comparing gain-scheduling pi and lq controllers. In Proceedings of the 18th World Congress The International Federation of Automatic Control Milano (Italy) August 28 - September 2, 2011, September 2011.

CITED LITERATURE (continued)

10. Wikipedia: Pid controller. https://en.wikipedia.org/wiki/PID_controller, [online; accessed 6/16/20].

11. Lenhard, T.: Commercial radar sensors and applications. AMA Conferences 2017 – SENSOR 2017 and IRS2 2017, 2017.

12. MathWorks: Resilient backpropagation. https://www.mathworks.com/help/deeplearning/ref/trainrp.html, [online; accessed 6/20/20].

13. Yan, S.: Understanding lstm and its diagrams. https://medium.com/mlreview/understanding-lstm-and-its-diagrams-37e2f46f1714, [online; accessed 6/25/20].

14. MathWorks: Long short-term memory networks. https://www.mathworks.com/help/deeplearning/ug/long-short-term-memory-networks.html, [online; accessed 6/25/20]).

15. Chao, Q., Jin, X., Huang, H., Foong, S., Yu, L., and Yeung, S.: Force-based heterogeneous traffic simulation for autonomous vehicle testing. In 2019 International Conference on Robotics and Automation (ICRA), pages 8298–8304, May 2019.

16. Broadhurst, A., Baker, S., and Kanade, T.: Monte carlo road safety reasoning. In IEEE Proceedings. Intelligent Vehicles Symposium, 2005., pages 319–324, September 2005.

17. Eidehall, A. and Petersson, L.: Statistical threat assessment for general road scenes using monte carlo sampling. IEEE Transactions on Intelligent Transportation Systems, 9(1):137–147, February 2008.

18. Schamm, T. and Zöllner, J. M.: A model-based approach to probabilistic situation assessment for driver assistance systems. In 2011 14th International IEEE Conference on Intelligent Transportation Systems (ITSC), pages 1404–1409, November 2011.

19. Kuhnt, F., Schulz, J., Schamm, T., and Zöllner, J. M.: Understanding interactions between traffic participants based on learned behaviors. In 2016 IEEE Intelligent Vehicles Symposium (IV), pages 1271–1278, June 2016.

20. Diehl, F., Brunner, T., Le, M. T., and Knoll, A.: Graph neural networks for modelling traffic participant interaction. In 2019 IEEE Intelligent Vehicles Symposium (IV), pages 695–701, August 2019.

CITED LITERATURE (continued)

21. Ontanòn, S., Lee, Y.-C., Snodgrass, S., Winston, F. K., and Gonzalez, A. J.: Learning to predict driver behavior from observation. The AAAI 2017 Spring Symposium on Learning from Observation of Humans Technical Report SS-17-06, 2017.

22. Ivanov, R., Weimer, J., Alur, R., Pappas, G. J., and Lee, I.: Verisig: verifying safety properties of hybrid systems with neural network controllers. In arXiv:1811.01828v1 [cs.SY], 2018.

23. Junges, S., Jansen, N., Dehnert, C., Topcu, U., and Katoen, J.-P.: Safety-constrained reinforcement learning for mdps. In arXiv:1510.05880v1 [cs.SE], 2015.

24. Emero: How self-driving cars work – a simple overview. `https://emerj.com/ai-sector-overviews/how-self-driving-cars-work/`, [online; accessed 6/15/20].

VITA

NAME	Jacopo Milone
EDUCATION	
	Master of Science in "Electrical and Computer Engineering, University of Illinois at Chicago, December 2020, USA
	Specialization Degree in " Mechatronic Engineering ", Jul 2020, Polytechnic of Turin, Italy
	Bachelor's Degree in Electronic Engineering, Jul 2018, Polytechnic of Turin, Italy
LANGUAGE SKILLS	
Italian	Native speaker
English	Full working proficiency
	A.Y. 2019/20 One Year of study abroad in Chicago, Illinois
	A.Y. 2018/19 Lessons and exams attended exclusively in English
	A.Y. 2017/18 One Semester of study abroad in Knoxville, Tennessee
	2017 - IELTS examination (7.0/9)
SCHOLARSHIPS	
Fall 2019	Italian scholarship for TOP-UIC students