UNIVERSITÀ
DEGLI STUDI
FIRENZE

Scuola di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Informatica

# DEVELOPMENT OF A MONITORING METHODOLOGY FOR AUTONOMOUS VEHICLES MANAGED BY A CONTROLLER AND A SAFETY-MONITOR

TERROSI FRANCESCO

BONDAVALLI ANDREA
STRIGINI LORENZO

Anno Accademico 2018-2019

ABSTRACT

Self-Driving cars are one of the hottest topic of the decade. With the rise of *machine learning*, it seems like we are going to see cars driven by artificial intelligences on the roads very soon. The complexity of these systems require extra care due to the complexity of the environment in which they provide their service. The architectural features of these systems and the ultra-high dependability required, makes them fall in the category of *Safety-Critical Cyber-Physical System of Systems*. At the same time, the techniques used to teach to a computer how to drive a car are still being studied, but little attention is given to the *emergent behaviour* that comes from the combination of artificial intelligences and error checkers. In this work we explored this topic, putting our focus on the system in its entirety, aiming at developing a monitoring activity capable of giving an overview of the system's performances, starting from common measures of interest well known in the systems engineering community. Moreover, due to the asymmetric nature of these systems, we also focused on the long-term effectiveness of these error checkers when the neural network becomes an *"expert driver"*, wondering if these AIs can get so expert that the use of an error-checker actually becomes *detrimental* for the system in providing its service.

To conduct this study, an *Object-Detection Module* was developed using non-AI techniques, to be used as an error-checker. A *DDPG Agent* was at first trained with a *default strategy* and then retrained using different strategies, to observe how these could have an impact on both the agent's performances and the error-checker's performances.

This thesis presents an overview of the current state-of-art in the *dependability assessment and monitoring* for safety-critical systems and the issues that prevents us to deploy self-driving cars worldwide. An experimental methodology was developed, respecting all the requirements that a monitoring activity must have to be acceptable, and used to monitor these systems in a simulated environment.

The methodology proved to be an effective mean for the evaluation of the whole system in a simulated environment, by combining few metrics together we had a better understanding of its behaviour. The results collected seems to suggest that there is a correlation between *how good* the neural network is in driving a car, and the accuracy of the error-checker. This work is nothing but a starting point for further studies, aiming at understanding the complexity of these systems.

# CONTENTS

## LIST OF TABLES

## LIST OF FIGURES

7

# 1

## INTRODUCTION

Nowadays, computer systems have a central role in our society. Originally intended to be computing tools for mathematicians, thanks to the progress in technology and in engineering techniques, these systems are now used for a multitude of purposes in very different domains: from the medical field, to the avionics, without forgetting about the relatively novel *"Internet of Things"*.

The new domains in which these systems are used, required a redefinition of the concept of *"computer systems"*. Many of these systems must respect hard deadlines to fullfill their tasks, or there may be serious consequences which could result in the disruption of the infrastructures, people injuried or even in fatalities.

Whenever the system must respect hard deadlines, we refer to them as *Critical Systems*. If a failure of such systems may be catastrophic, with serious damages to the environment, the infrastructures, or human beings, we talk about *Safety Critical Systems*.[25]

There are cases in which these systems perform in a physical environment. An example of such a system is an autonomous car: a system composed by a *computer system*, in charge of doing the computations needed to fulfill the assigned task, and a physical component, in charge of interacting with the environment by change both the states of it and of the system itself.

Because these systems are so pervasive and yet so dangerous, in case of a failure, to satisfy and guarantee their (usually) ultra-high *dependability* requirements is a key factor during the design and development phase.

The *dependability* of a System is a measure of how *"trustable"* a system is, i.e. its ability to provide a **correct service**.[24]

A *failure* is an event that causes a disruption of the provided service.

## 1.1 DEPENDABILITY AND SAFETY

The *dependability* of a critical system is defined as an ensemble of *qualitative* and *quantitative* measures. Some of the most important are listed here:

- *Availability*: a function that measures the alternation between correct and incorrect service

$$A(t) = \begin{cases} 1 & \text{if a correct service is provided at time } t \\ 0 & \text{otherwise} \end{cases}$$

  $\mathbb{E}[A(t)]$, the expected value of the availibility, is the probability that the system is providing a correct service at time t

- *Reliability*: the ability of the system to provide a *continuous service*

  $R(t)$: probability of providing a correct service in the interval $[0, t)$

- *Safety*: the absence of catastrophic consequences if a failure occurrs.

- The *Safety* of a system is often defined combining other metrics such as the *Mean Time to the next Catastrophic Failure*, because the safety requirements require a quantitative measure (e.g. one failure every $10^n$) years).

  $S(t)$: probability that **no failure** occurrs in the interval $[0, t)$

  MTTF: Mean time between the recovery from a failure and the next failure.

- *Maintenaibility*: the ability for easy maintenance and repair from a failure. This measure has an influence on the Availibility.

  MTTR: Mean Time to Repair the system from a failure

- *Coverage*: the measure of effectiveness of the system's fault-tolerance mechanisms, i.e. the ability of the system to prevent, avoid or correct failures

A *threat* is a menace to the system's dependability: that is an *"event"* that makes the system provide an incorrect service. Threats may be of different kinds and come from many sources, such as wrong specification or a wrong implementation of a specification, accidents...

When considering the dependability of a system, we are interested in providing a *continuous correct service*. A *failure* is a transition from a state of *correct service* to a state of *incorrect service*. When designing a Critical System, the need to reduce the possible transitions (failures) from a state of *correct service* to a state of *incorrect service*, guides the development process.
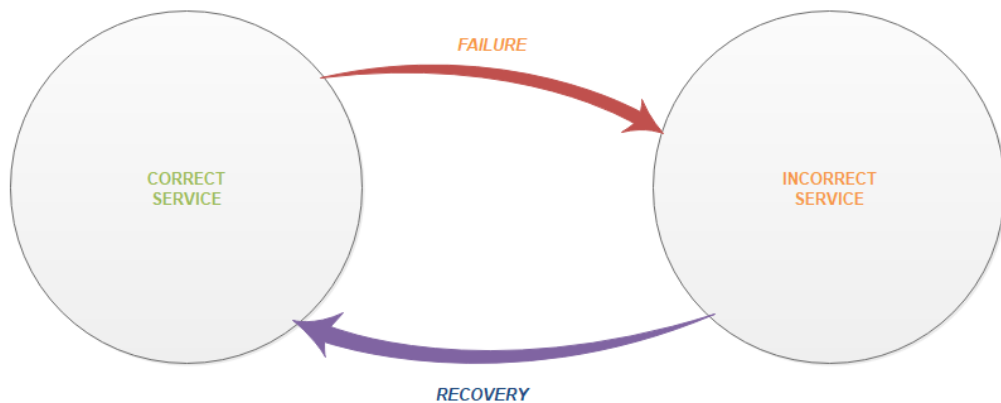


Figure 1

When it comes to *Safety-Critical Systems*, we must distinguish between *benign failures* and *catastrophic failures*. The first are failures that we can somehow accept: the system will not provid a correct service but it will still be in a safe state. The latter are the most dangerous kinds of failure because an incorrect, unsafe service will potentially have catastrophic consequences such as damages to the environment, disruption of the system's infrastructure or even fatalities, in cases where these systems and humans work in close contact. As an example, think to an autonomous car. Imagine that the car is riding in "normal" conditions and suddenly an obstacle appers in front of it. A screeching halt and a consequent interruption of the ride is indeed a failed state, but it's considered as a *benign safe*, because no one got hurt. On the other hand, a situation in which the car starts to throttle towards the obstacle (and eventually collides with it), is defined as an *unsafe failure* because people in the car could get seriously injuried (or killed).

Figure 2

To study the dependability of Safety-Critical Systems, we want to find all the possible failure causes. To do so, the academics agreed on the *fault-error-failure chain* model, which is widely accepted both by academics and practitioners:

- *Fault*: the *adjudjed or hypothesized* cause of an error

- *Error*: that part of the system state that may cause a subsequent failure

- *Failure*: the situation in which the *error* reaches the service interface, altering the service of the whole system



Figure 3

The dependability of a system comes from a set of four techniques that aims to prevent/mitigate the consequences of possible failures:

- *Fault Prevention*: means to prevent the occurence or introduction of failures

- *Fault Tolerance*: mechanisms to tolerate faults. In the case a fault happens, the system is still able to provide a correct service

- *Fault Removal*: reduction of the *number* or *severity* of faults in the system

- *Fault Forecasting*: use of statistical techniques to estimate the present number, the future incidence and the likely consequences of faults

The effectiveness of the measures adopted to achieve the dependability requirements of a system is measured through the *validation* process. The validation of the system requirements is a process that must be repeated in each phase of its development, even at the very beginning of the design phase.

There are multiple validation techniques that may be specifically selected for each phase of the system development: [26]

- *Analytical/Numerical Modeling*

    - Techniques that models the system capabilities using numerical models that have a closed form solution, i.e. the changes in a system can be described using a mathematical analytic function. Examples of such models are the *Combinatorial model* and the *State-Based model*

- *Simulation*

    - Empirical estimation of the system dependability, deployed in a simulated environment. This method allows to inject faults without damaging the real system (or prototype), to check whether a specific fault-tolerance mechanism works or not

- *Measurement*

    - When a prototype of the system is available, its executions can be observed and the measures of interest computed

It is important to keep in mind that these techniques are non-exclusive and the validation process should be performed combining all of them.

As said before, the process of validation should be done during the *whole* lifetime of a system, starting from the design phase and continuing after the deployment. Some of the methods listed above are more suitable than others in certain phases:

- *Specification*: validation is done by the specification of the dependability requirements, that can be validated using Analytical/Numerical techniques, e.g. using *Combinatorial Models*, to determine the failure conditions for the system's components. Failures are considered to be independent

- *Design*: during the design phase, it's reasonable to model the state-space of the system using *State-Based Models*. Examples of such models are *Markov Chains*, *Petri Nets* (and its evolutions)...

- *Implementation*: when the development is in an advanced phase, it may be possible to build a prototype of the system that can be directly observed to measure the effectiveness of the fault-tolerance mechanisms on the system dependability

- *Operation*: once the system is deployed, it is possible to observe how it performs in a real environment

## 1.2    SYSTEM MONITORING

To observe a system operating in its environment to collect measures and evidences about its properties, is a technique called *System Monitoring*. Nowadays it's considered a good way to measure the dependability of a system, and diverse approaches on how it has to be done were proposed in literature. In this thesis we stick with the methods described in these works [27], [28], [29]. This technique aims at constantly monitoring the system running in its *final* environment, verifying that the *observed behaviour* and *performance* meet the defined requirements. The data collected during the monitoring activity *must* be validated:

- *Offline*: data are collected and stored somewhere, while the system is running, and are analyzed afterwards

- *Online (or at Runtime)*: data are analyzed *while* they are being collected

A good monitoring technique must consider all these aspects:

- *Identification* of the relevant events, measures and features of the system, that are necessary to assess the system dependability

- *Labeling* data in order to enrich raw measurements with more informations

- *Transmission* of the data collected to the analysis node, in order to process them

- *Filtering* and *Classification* of data with respect to the measures of interest

In the definition of a monitoring activity, we call *Target System* the whole system as it is. If the monitoring activity is related to a specific hardware or software component of the system, we refer to it as *Target Component* or *Target Application*. Both practitioners and academics agree on the effectiveness of these antithetical approaches:

- *Black-Box approach*

- *White-Box approach*

The choice on which approach to use depends on how much control it's possible to have on the target system, especially on its internal implementation. If the details implementation are unknown e.g. because the monitoring activity is performed on a third-party system, only a black box approach can be adopted: after the definition of a *workload* (i.e. an input or a series of inputs, representative of a possible execution), the workload is given to the system and its outputs are observed.



Figure 4

If the internal details of the Target System are known and freely accessible, it's possible to monitor the system "from the inside", attaching *probes* directly to the system in order to observe the intermediate outputs produced by the system *during* its execution. These probes, since they are attached directly to the system's internal components, can give much more informations than the one collectable just by observing the system's outputs. If this approach provides a lot of more informations on the system's behaviour on one hand, on the other hand it requires extra care on how the monitoring activity and system probing are done. In particular, it's mandatory to adhere to these two basic rules:

- *Representativeness of Selections*: the probes must be able to observe an *adequate number of meaningful observations* to perform a good monitoring activity

- *No Intrusiveness*: the system's behaviour *must not* be affected by probing, otherwise the collected data will be meaningless



Figure 5

The design and development of the monitoring system requires additional reasoning, especially on how the probing is done. In particular, we can distinguish among:

- *Hardware Monitoring*

- *Software Monitoring*

- *Hybrid Monitoring*

A dedicated hardware circuit for monitoring is the best way to monitor a system because of the very low intrusiveness (outputs are read "on the fly" while being produced). However, systems are becoming more and more complex nowadays, making it very difficult, if not impossible, to install hardware probes. Software probes are very powerful instruments, because they have access to more informations than the ones available to hardware probes, because it's possible to know the *context* in which the specific output was produced. Software probing is done by adding instructions, specifically dedicated to data extraction and measurement,

in the *Target Process*, in the *Operative System* or by developing a new *probe process*. Hardware and software probes can be combined in a hybrid approach, with particular care on mitigating the cons of each technique.

## 1.3 MOTIVATION AND GOAL OF THIS WORK

In this work we are interested in monitoring a self-driving car composed by a Neural Network, in charge of driving the car, and a System Supervisor, a component that provides fault-tolerance mechanisms to avoid catastrophic failures.

Autonomous cars are one of the most recent and promising safety-critical systems in terms of the implications they may have on future system developments.

This kind of systems usually have ultra-high dependability requirements that are very hard to validate. Moreover, the nature of the software architecture, involving AIs and non-AI software interacting with each other, makes the task of validation even harder.

The goal of this work is to develop an experimental method to assess the dependability of such complex systems, with particular care for the problem of *what* measures are relevant for these systems and *what* are the issues that have an impact on the analysis activity. An experimental activity was conducted in a realistic simulated environment to demonstrate the concepts proposed in this thesis.

We didn't have a trained Neural Network available, nor a System Supervisor. The network was trained from scratch and a simple System Supervisor was developed as part of this work.

It's important to notice that this work doesn't aim at providing an *exhaustive* treaty on the argument, but at starting a phase of exploration of these concepts and problems, that will require further validation and study in future works.

# AUTOMOTIVE - STATE OF ART

Self driving cars are one of the hottest topics of the decade. Artificial Intelligences specifically trained to drive with machine learning techniques demonstrated that it's possible for a computer to drive cars. However, a failure of these systems may have very serious consequences that could result in people being injured, or killed. At the same time, it is a problem to certify the ultra-high dependability requirements demanded. In this chapter, today's problems regarding the safety issues related to self driving cars are reviewed, for that it was decided to conduct this study.

## 2.1 AUTONOMOUS CARS AS CPS

In order for a car to be able to drive by itself, suitable hardware and software are required. This makes autonomous cars cyber physical systems, and the possible catastrophic consequences that a failure in/of these systems can cause, make them fall under the set of critical systems.

To sense and map the surrounding environment, the system collects data from multiple sensors. Some of the most important sensors and their purposes are listed here:

- GPS
    - → High precision GPS sensors are used to estimate the exact position on the vehicle in the world

- Odometry & IMU sensors
    - → These sensors are worth for detecting changes in the position of the car and of the objects in the environment over time

- Cameras
    - → Cameras are literally the *eyes* of the system. Images captured are usually processed with image recognition software

- Lidars & Radars

    → Lidars can be seen as the evolution of conventional radars. Data combined from these sensors serve the purpose of mapping the environment and detect obstacles and objects around the car

Outputs from these sensors are combined and given to the car's control system. An abstraction of the software architecture is shown in this figure:
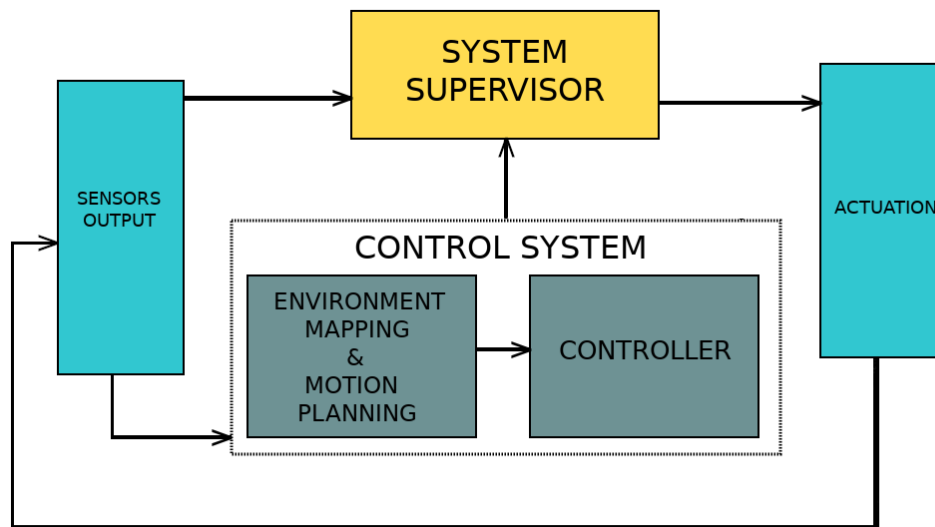


Figure 6: High-level abstraction of the system's software architecture

Data from sensors are inputs of the control system, here simplified as composed by two constituent systems: one in charge of collecting data directly from the sensors, process them in order to build an *occupancy grid*[1] to map the surrounding area and to create a physical model of the environment in order to follow the correct route to the destination without crashing. The Controller (usually composed as a combination of a Velocity Controller[2] and a Steering Controller[3]) uses these data to adjust the values on the actuators controlling the movements of the car: throttle, brake and steer.

---

1 A matrix mapping the environment, the cells $a_{i,j}$ are flagged with 0 if there's no object at coordinates $i, j$, 1 if occupied.
2 Controller in charge of adjusting the vehicle's speed
3 Controller that determines the steering angle

Due to the criticity of their task, it's mandatory to have a System Supervisor, a System in charge of detecting possible hardware failures or wrong outputs[4] from the Control System and, if needed, activate a corrective routine.

The System Supervisor is the main failure avoidance component of such systems. Of course there may be specific checks when data are processed, but the last decision is up to this system's monitor and the underestimation of its importance can lead to dramatic consequences, such as the 2018 accident in Arizona, where a woman was killed by a self-driving car during a test run.[3] Further inspections showed that the car's radar and LiDAR sensors detected the victim almost 6 seconds before the impact and it took 4 seconds circa to infer that there was an obstacle on the road and that an emergency brake was needed. However, this safety-checker was disabled during tests for "smoother rides", causing the accident.[5]

The extreme complexity of these systems raise concerns among the experts: the need to find a new point of view to study and test the safety of these systems, and the need to sensitise about safety culture.[6]

## 2.2 SAFETY AND AUTONOMOUS VEHICLES

According to a SAE International tentative to classify self-driving cars' autonomy, the level of automation can be divided in 6 tiers, ranging from 0 to 5. Level 0 means no autonomy: a human driver just drives the car; level 5 means that there's no need of human intervention at all and the car is not only capable of driving safely on the road, but it must be able to avoid catastrophic failures that may seriously harm (or kill) people. The more autonomous the car is, the higher the dependability requirements are for it to be put on public roads. It is a well known issue that demonstrating a system's dependability is not an easy task for itself, it gets even harder with ultra-high dependability systems such as these are. In adition to the problem itself, demonstrating autonomous cars' dependability has two more problems to deal with: how to safely and effectively test the system and the presence of neural networks, for which it's very hard to understand why it returned the output $y$, given the input $x$.

---

4 With *"wrong"* is intended not only outputs out of the domain space but also outputs that would cause the system to fail (e.g. causing a crash)

Lots of studies demonstrated that it's unthinkable to just test cars on the roads. One of these, that we will refer to as the RAND Study, answers the question of how many miles of driving would it take to demonstrate autonomous vehicles' reliability using classical statistical inference, saying that if autonomous cars fatality rate was 20% lower than humans', it would take more than 500 years with *"a fleet of 100 autonomous vehicles being test-driven 24 hours a day, 365 days a year at an average speed of 25 miles per hour"*.[2]

The validation of ultra-high dependability requirements for safety-critical systems is a well known problem in safety literature and has not been introduced by the advent of autonomous cars. In fact, the RAND study is nothing but a specific case of the problem considered in a work published in 1993 by Littlewood & Strigini, in which the same concepts are discussed and generalized for every ultra-high dependability system.[12]

The main problem with the RAND study approach is that future failures frequency can not be predicted just using the observed one. Not just for the quantitative results of the impossibility of it, but also because this approach can not work: an observed frequency failure of 0 would lead to optimistic (and probably harmful) predictions. Luckily, this problem is surmountable, as shown in *this*[11] work by Zhao et al.

Validating the dependability requirements of an autonomous car seems a hard task already. Things are made even harder by the fact that these cars are driven by neural networks.

In these years there is a huge growing interest in the *machine learning* sector, and this has made that a lot of progress was done in the research. It's also thanks to these progresses that autonomous cars now seem like something we can achieve, since these AIs gave surprising results with their skills, and big brands such as *Uber* and *Tesla* are putting more and more efforts in AI research. This new wave of AI research is deeply changing the way we interact with computer systems, and surprising results were achieved with neural networks.

These tools have proven to outclass *"classic"* software solutions (intended as non-neural network) in a lot of tasks, ranging from *Object Detection*[13] to *Gaming*[14] problems, performing even better than humans.[15] The complexity of the environment in which the system performs and the need of quick decision-making procedures and fast responses to events that *cannot* be planned with *"classic"* software, make neural networks the perfect tool to achieve the task of a car being able to drive by itself, thanks to their ability to handle multiple situations that were not explicitly written in the software. However, this raises serious issues about the safety of

the whole system for many reasons.

If neural networks gave promising results on one hand, and they seem the only way to achieve goals such as autonomous cars, on the other hand it has been shown many times how weird a network's prediction can get when *minimally* perturbating the inputs[17] and how high the confidence interval can be.[16] The lack of official regulations and certifications for this kind of software, as well as the need to truly understand neural networks, is raising concerns on how dependable these systems can be and consciousness is now growing on the topic, asking for more regulations on companies developing advanced AIs.[4]

## 2.3   CONTROLLER - CHECKER PROBLEM

The interactions between the Control System and the System Supervisor are at the core of the car's movements. The Control System, or *Primary Component*, is the software performing the main computations of the system, required to drive the car. In a context like this, it's mandatory to have fault-tolerance mechanisms such as the System Supervisor, to avoid catastrophic failures. This kind of architecture is a must for these systems, due to the extremely high dependability requirements they have, in order to try to cover all the possible failures that may happen. The state space of such systems may be sketched as in figure 7.
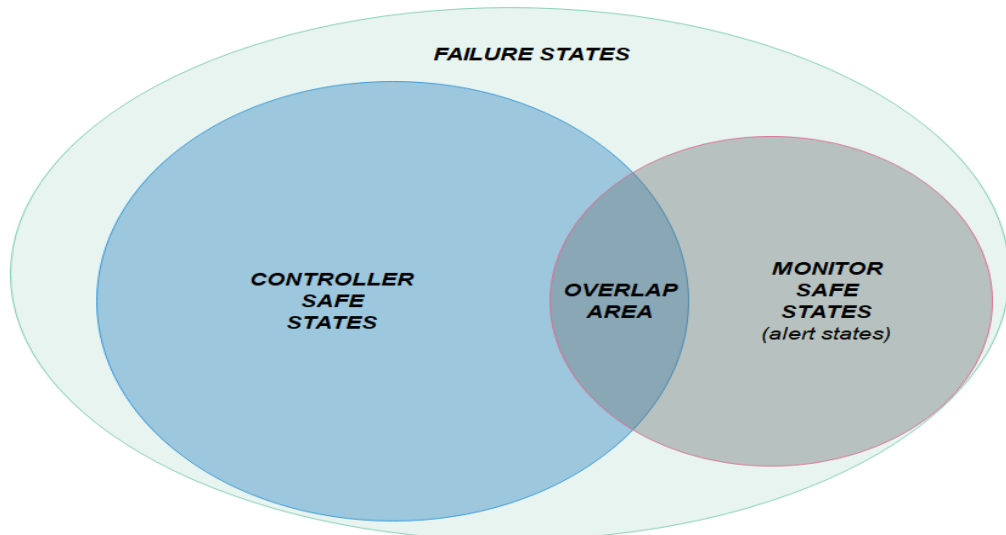


Figure 7: Sketch of the system's safe states

We consider *safe states* every state in which the Control System produces an output that would not result in a crash. If we model the system's failures in this way, the level of safety of the system can be represented as the union of the failure area covered by the Controller and the one covered by the Supervisor, with an *overlap area* in which the Supervisor is actually *detrimental* to the System performances.

Now imagine that an autonomous car is riding when suddenly an obstacle appears. If the Primary correctly detects the obstacles it should autonomously apply a safety-measure to **avoid** a transition in an *alert state*. If the controller doesn't see the obstacle, *or* detects it but keeps throttling we say that the Controller is *failed* and there is a transition from a *safe-state* to an *alert-state*, in which the failure-avoidance components turn in. The System Supervisor's duty is now to launch a corrective-routine that will put the system in a fail-safe state (e.g. by applying the safety-brake and turning off the engine). An error of the Supervisor will inevitably cause the system to fail, as a result of the failure of both components, leading in a failure state (the crash happened).

The possible behaviours of the System Supervisor can be represented using the *Confusion Matrix*, a specific table layout that allows visualization of the performance of a *classification algorithm*. This is done by creating two classes: a *Positive Class* and a *Negative Class*.

Imagine we are interested in recognizing *black sheep* in a mixed herd of white and black sheep. The class of black sheep will be the *Positive Class*. Every black sheep classified as such is a *True Positive*. The set of *white sheep* is the *Negative Class*, and each white sheep classified as such is a *True Negative*. Misclassification of a white or black sheep leads to a *False Positive* and *False Negative* respectively. [36]

This special kind of *contingency table* has two dimensions: *actual value* and *predicted value*:

|                   |          | Actual Value | |
| ----------------- | -------- | :----------: | :------: |
|                   |          | Positive | Negative |
|                   | Positive | TP | FP |
| Predicted Value   | Negative | FN | TN |

Table 1: Confusion Matrix

In the field of self-driving cars, we have to redefine *what* are the positive negative classes. Recall that we defined a *"failure"* of the Controller as

a transition from a *safe state*: a state in which the Controller can handle what's happening in the environment without crashing, to an *alert state*: a condition in which without the Supervisor, the System would inevitably go in a failure state. Hence, the Positive class is the set of events that will eventually result in a crash, while the Negative class is the set of events that are correctly managed by the Controller. For the Suervisor, we want to measure its ability to distinguish between safe states and alert states. To do so, we defined **positives** and **negatives** predictions in this way:
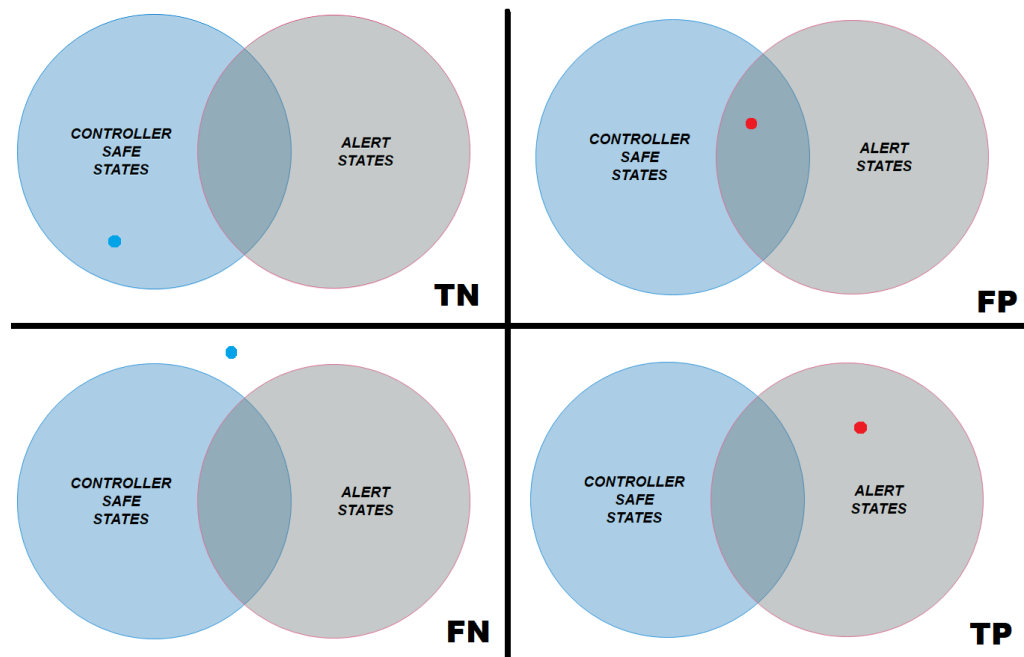
Figure 8: Graphical Representation of what true and false predictions means in the system's state space. Dots represent the current state of the System. A blue dot means that no alarm is raised, a red dot means that the Monitor raises an alarm

TP:  True Positive

- The system went in an alert state due to a Controller's failure, correctly detected and prevented by the monitor

TN:  True Negative

- The system is in a safe state and the Monitor doesn't raise any alarm

FP:  False Positive

- The system is in a safe state, but the Monitor raises an alarm

FN: False Negative

- The system is in an alert state and the Monitor doesn't detect the hazard

These raw values are nothing but a starting point to compute more complex and meaningful measures, usually presented in form of *rates*, linked with each other using statistical laws, making them equivalent from a statistical point of view and easy to compute once the amounts of correct/wrong predictions are known. Examples of some of these measures are:

- *Sensitivity (True Positive Rate):* measures the proportion of actual positives, correctly identified as such

$$TPR = \frac{TP}{TP+FN} = 1 - FNR$$

- *Specificity (True Negative Rate):* measures the proportion of actual negatives, correctly identified as such

$$TNR = \frac{TN}{TN+FP} = 1 - FPR$$

- *Miss Rate (False Negative Rate):* measures the proportion of actual positives predicted as negatives

$$FNR = \frac{FN}{FN+TN} = 1 - TPR$$

- *Fall-out (False Positive Rate):* measures the proportion of actual negatives predicted as positives

$$FPR = \frac{FP}{FP+TN} = 1 - TNR$$

These four rates can be combined as well, to compute many more measures that give an indication of the model's predictions accuracy. Here we list some of the most used measures:

- *Accuracy:* a measures used for describing *systematic errors* in predictions. A low *Accuracy* is the cause of differences between a prediction and its *"true"* value.

$$ACC = \frac{TP+TN}{TP+TN+FP+FN}$$

- *Precision:* a measures used for describing *random errors* in predictions, that is: a measure of *statistical variability* in the model used for predictions.

$$\text{PREC} = \tfrac{\text{TP}}{\text{TP}+\text{FP}}$$

- $F_\beta$-*score:* a measure of a test's accuracy expressed combining *precision* and *sensitivity*. This ranges from 0 (worst value) to 1 (best value), and it directly depends on the class defined as *positive*.

$$F_\beta - \text{score} = (1 + \beta^2) \cdot \tfrac{\text{PREC}\cdot\text{TPR}}{\beta^2\cdot\text{PREC}+\text{TPR}}$$

- *Matthew's Correlation Coefficient:* a measure of the *overall quality* of predictions, taking into account *every* cell of the Confusion Matrix. This metric ranges from $-1$ (worst value) to 1 (best value), therefore giving a wider view of the model accuracy.

$$\text{MCC} = \tfrac{\text{TP}\cdot\text{TN}-\text{FP}\cdot\text{FN}}{\sqrt{(\text{TP}+\text{FP})\cdot(\text{TP}+\text{FN})\cdot(\text{TN}+\text{FP})\cdot(\text{TN}+\text{FN})}}$$

- *ROC[5] Curve Plot:* a graphical plot representing the performance of a binary classifier, defining the x-axis by False Positive Rate and the y-axis by True Positive Rate and splitting the plot with the $y = x$ line. Values over this line represents "good" predictions, values under this line represents "worse-than-random" predictions, while values on the line represents random predictions.

It is important to keep in mind that once the true amount of positive and negative predictions are known, all these measures can be computed (and many more, depending on the need of the case study) and linked to each other, making easy to switch point of view of the analysis. In this work we want to demonstrate that with this methodology, we can approximate all these amounts that are mandatory in order to compute sophisticated other measures based on the needs of the assessment e.g. *False Discovery Rate, Threat Score, Informedness. . .*

This problem is nothing but a generalization (related to safety) of the asymmetric fault-tolerant architecture for computer systems: the idea of having a *Primary Component* that performs the main computations, and a *Primary Checker* in charge of detecting (and correcting) possible errors of the Primary.
The problem of assessing the dependability of these simpler (but still complex) systems is a well known topic in literature and was explored in different studies. In a relatively recent work published by *Popov* and *Strigini* in 2010, it is shown that the probability of a system failing on a

---

5 Receiver Operator Characteristics

specific input (or set of inputs), strictly depends on both the coverage of the Primary *and* the Primary Checker, as shown in the figure 7.[1]

In the context of self-driving cars we want the area covered by the primary to be the largest possible. This is done by intensive training of the neural networks that will control the car. As long as the network is trained *"properly"*, the control system should be able to handle most of the dangerous situations that may happen. At one point, it is possible that the Controller learns to handle the *"alert states"* covered by the System Supervisor, reducing the overall contribution to the system's safety given by the latter (figure 9).
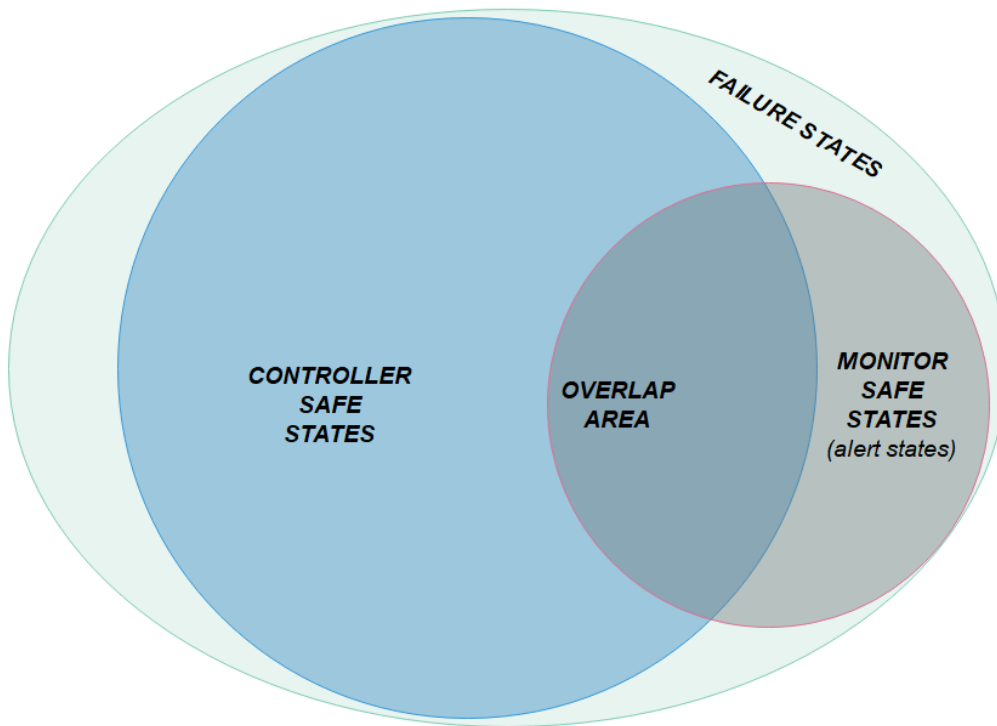


Figure 9: The states covered by the Controller are now the ones previously covered, plus some states previously covered just by the Monitor

Another possibility is that during the training, a portion of the failure area covered by the controlller becomes uncovered. This could result in a situation in which some of the previously safe states are now alarm state, representing a serious harm to the system's dependability. Since the coverage area provided by the System Supervisor can not change without changing its implementation (it doesn't "learn" automatically), a

transition to one of these states would now inevitably result in a failure (figure 10).
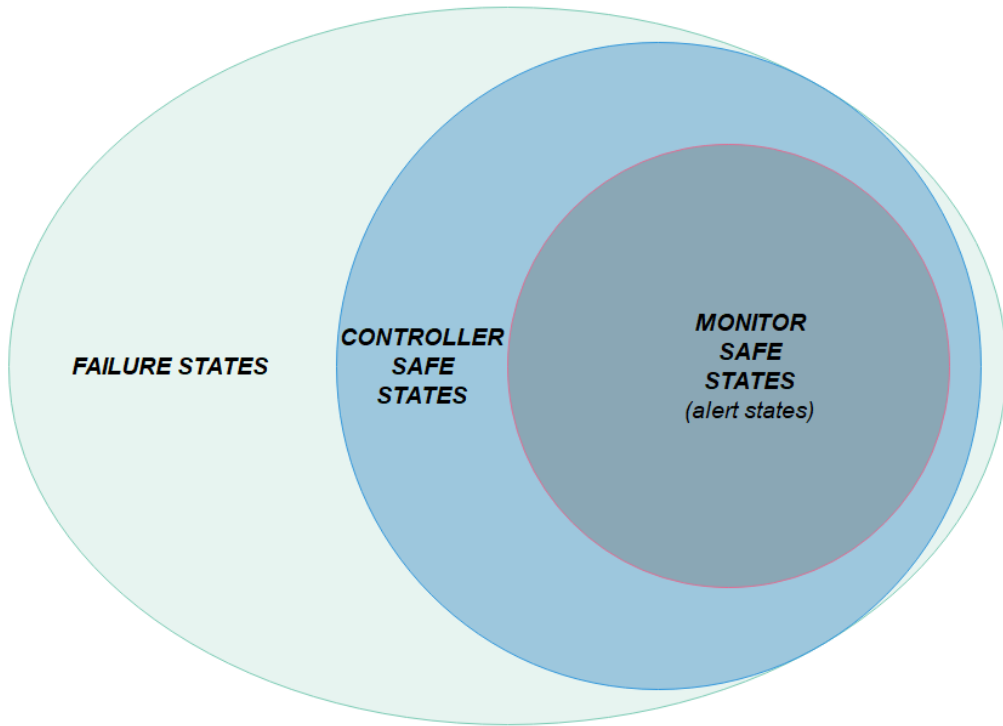


Figure 10: The Controller now covers all the states previously covered by the Monitor, but doesn't cover anymore some of the states he covered before the training

All these considerations and the lack of literature on the topic for these new systems such as autonomous cars, lead us to begin a study on the emergent behaviour resulting from the interaction of a neural network control system and a *"classic"* error checker, on what happens when the network is *taught* by a supervisor during the training and how performance metrics of these 2 components can be computed.

In the next sections we present and discuss the development and implementation of an experimental methodology to study these aspects.

# 3

## SYSTEM ANALYSIS METHOD

### 3.1 INTRODUCTION AND PRELIMINARIES

The goal of this study is to develop a first experimental methodology intended to observe emergence-related aspects coming from the interaction of a Control System and a System Supervisor, analyzing the system in a *simulated* environment.

The software architecture of an AV was simplified in two main components:

- A *Controller*: a Neural Network trained with *reinforcement learning* algorithms to drive the car

- A *Safety Monitor*: a submodule of the System Supervisor, that checks whether the car is going too fast towards an object, processing data received from a LiDAR sensor, and if that's so, apply an emergency-brake

This work focuses on exploring the topic from a different point of view and to assess its *feasibility* in an experimental, simulated environment. Due to the system being composed by two constituent systems: the Controller and the Monitor, we think that a point of view based on the emergent behaviour resulting from the interaction of these systems can improve the quality of the assessment.

In this chapter is presented and discussed a method to study the safety level of an autonomous car over time, observing the emergence resulting from the interactions of a neural network controller and a safety monitor in a simulated environment.

The proposed framework is designed with particular attention on studying the emergence resulting from the interaction of these two Consitutent Systems.

The main aspects we are interested in this first stage of exploration are:

- How the effectiveness of a monitor evolves when the neural network is learning

- Effects of training strategies on the effectiveness of a safety monitor

One of the most appealing features of neural networks is that they can be *trained* on data sets to improve their performance. One stage of training is done by collecting data over $n$ steps and updating the weights of the prediction function. The weights of the function after $i$ training stages represents the *state* of the network at **epoch i**.
A neural network will likely give good results after "enough" epochs. The harder the task, the more epochs are needed. Driving a car is a quite hard task and it's inimmaginable to save the weights of each epoch. Therefore, given a neural network N, we define a **checkpoint** as a generic epoch of N. Say we trained N for 1000 epochs. If we save the weights of the prediction function every 100 epochs, we will end up with 10 checkpoints:

$$\text{checkpoint}_1 < \text{checkpoint}_2 < ... < \text{checkpoint}_{10}$$

where $\text{checkpoint}_1$ contains the network's weights at epoch 100, $\text{checkpoint}_2$ at epoch 200 and so on.

Let's now consider a self-driving car that is being tested on the road (either real, or simulated). Its task is to ride the car the longest possible, without crashing. During the ride, the environment surrounding the car will change as it procedes in its run. It may happen that in some of the system's state, the probability of a subsequent crash becomes very high, like a pedestrian suddenly crossing the road: if and only if the action taken by the Controller will result in the pedestrian getting hit we will address it as a failure (of the controller). The same reasoning applies if the pedestrian is actually detected, and the car hits something else while trying to prevent it:

- If the Controller takes *any* action that would result in a crash, it's considered failed

- If a hazardous event happens, i.e. a situation in which the probability of observing a crash is higher than usual, the Controller is considered failed if and only if its actions will not avoid the imminent failure

In this sense, at this stage of the work, we don't distinguish between changes in the environment that raises the probability of a crash (e.g. a pedestrian crossing the street) and hazardous actions take by the Controller (e.g. a sudden steer towards a wall).

If the controller fails in the way just described, it's the Monitor's duty to run a safety-routine in order to prevent the imminent system failure.

In the case of a failure of the Controller, the Monitor not only needs to detect whether it failed or not, but it also must run a safety-routine to prevent a failure of the whole system. In this first phase of analysis, we consider the action taken by the Monitor to be always safe. This means that:

- If the Monitor executes **all** the steps in the safety-routine, the system will be in a safe-state.

- A failure of the Safety Monitor may be one of the following:
    1) The obstacle is not detected
    2) The obstacle is detected but the routine fails to terminate its execution (i.e. the detection was too late)

We consider the system failed if and only if both the Controller and the Safety-Monitor failed, as described above, leading to a crash. With this scheme in mind, the states space for this system can be modeled as in figure 10.

Based on our definition, we can divide the system states in 3 sets:

- Safe States: states in which the Controller doesn't need an intervention of the Monitor

- Alert States: states in which the Monitor is required to intervene. A correct detection (and prevention) of the potential accident would result in a transition to the safe states space again

- Failure States: states in which an accident happened. It is important to notice that **not** all the situations can be detected by the Monitor. There are accidents that can not be prevented at all, in which no Monitor could save the system, therefore there is a direct transition from a *safe state* to a *failure state*
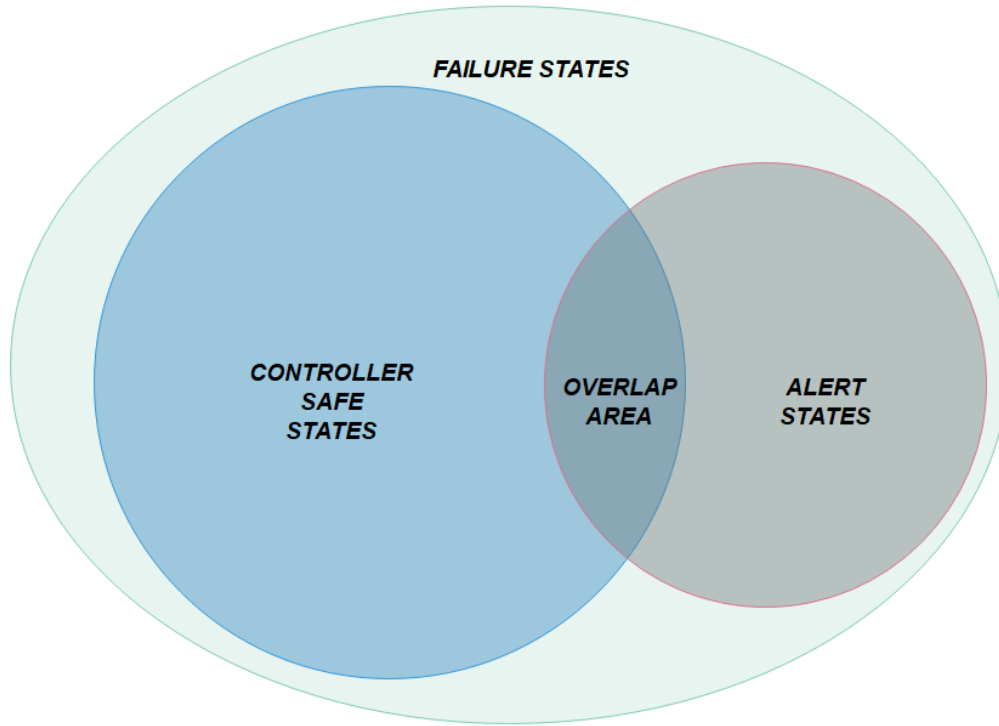
Figure 11: Representation of the system's states space

At the system level, we are interested in observing the probability of having a failure (crash) and how to minimize it. At the same time we want to observe how the effectiveness of the Safety Monitor changes when the Controller is trained over time.

In order to achieve this, $m$ checkpoints are saved for later testing and comparison. This is useful not only for checking that the network is improving during the training, but also to have a better understanding on how useful is the Monitor when the Controller becomes more and more expert. This is mandatory for the experimental activity, as different checkpoints of the same network are tested under the same identical conditions, to observe how the behaviour of the Controller changes over the training activity.

The need for multiple checkpoints is mandatory not only to test that the network is improving, but also to observe how the monitor's effectiveness changes over time. Moreover, if many checkpoints of the same network are tested in the same scenarios, it is possible to extract interesting measures and compare the behaviour of 2 checkpoints in the same situation, as we'll see in the next section.

Before the analysis can start, $n_h$ scenarios must be defined. A *Scenario* is a set of initial conditions (e.g. the spawn point of the car, seeds used in random number generators...) in which the car is intended to be tested. The pedix $h$ represents the difficulty level for the specific case. The purpose of this is that we are interested in testing the car under the same initial conditions, except for one factor, in order to have a better understanding on what makes the system fail more often. The $h$ variations should be developed with growing difficulty and at the same time they must be realistic. Given a Scenario $S$, examples of variations may be: to increase the number of cars in the scenario, or to simulate adversal weather conditions. A combination of these 2 variations should result in a *harder* variation of the scenario[1]. It's important to keep in mind that these scenarios, once defined, should not be changed as they will be used for testing all the checkpoints. A change in the settings of a scenario (except for variations) should be considered as a new one.

We hope that the results collected here will help in the process of understanding what makes a situation *"harder"* than others for such systems.

## 3.2 EXPERIMENTAL METHOD

The approach used for this exploratory work is divided in 3 phases:

- Phase 1: Given $c$ checkpoints of a neural network and $n_h$ scenarios, the Controller is tested in all the scenarios and its runs are recorded

- Phase 2: The Monitor is tested by *repeating* the runs recorded in phase 1, attaching the Safety Monitor to the system

- Phase 3: The network is retrained from the last checkpoint recorded, using different strategies to improve its performances. The new Controllers obtained and the (same) Safety Monitor are then retested in all the $n_h$ scenarios

In the first phase we are interested in assessing the goodness of the neural network and of the Monitor. This is done in 2 different steps.

In the first step, the $m$ checkpoints of the Controller are tested in all the scenarios. In this step we are mostly interested in observing how the *Reliability* of the Controller changes with respect to these checkpoints.

---

[1] It is important to point that what we think to be "harder", may in reality be easier to handle for the car.

One of the main problems when testing a neural network is that of *repeatability*. It is very unlikely that the **same** neural network, under the **same** initial conditions, will behave in the same manner in multiple runs. Due to this property of networks, it may happen that the failure mode observed in one of the runs of the scenario $s_i$ will never happen again, or the time needed to make it happen again may be very long. How the scenarios and their variations are created is fundamental to observe specific failures, however it's impossible to think to *all* the possible failure situations that may happen, for this reason we think that the scenario-variation approach may help in solving this issue, creating harder operational situations in which the factors that lead to a crash may be studied.

The repeatibility issue was solved by creating a *black box* for each run of the scenarios developed for testing. This approach is used to keep a trace of the Controller's actions, so that the specific run may be studied more fully to understand what made tha Controller fail. These data can be then used to better study what hazardous situations are covered by the Controller at a specific checkpoint $j$, and if these situations are still covered when the network is tested at the checkpoint $j + x$.

### 3.2.1   *Controller Testing*

The Controller is tested in isolation in each scenario, for each difficulty level, until a crash occurs. The situation in which no failure is recorded is still a possibility (even if the difficulty level somehow mitigates this issue). A reasonable alting criteria is out of the scope of this work and is still a problem in the academic community, however, as noted in the previous section, this problem may be solved.[11]

The reasoning behind the choice of isolating the in order to test it Controller, comes from some of the problems issued during the method development phase. The main problems are the *repeatability* and *non-intrusiveness*. As pointed before, the *repeatability* issue for the neural network is solved by creating a *black box* containing informations about the car's state in each frame. At the same time we can not think about testing the whole system at once (Controller *and* Monitor) because a safety-brake executed by the monitor will most likely change the environmental conditions for the rest of the simulation and we would not be able to compute measures about the goodness of the Controller itself. Testing the Con-

troller in isolation helps to solve these issues and it's preparatory to the
second phase.

Recording the actions taken by the Controller for *each* frame (as well as
other measures such as the vehicle's speed in that frame) make it possible
to have a great control over the data, as a single run may be repeated
many times to gather additional data if needed.

When the controller has been tested for each difficulty, in all the sce-
narios at least the followings must be computed:

- $MDBF_{i,j} = \frac{\#\,of\,faults}{meters\,travelled}$

    - Mean Distance Between Failure for the $i^{th}$ checkpoint, at the
      $j^{th}$ level of difficulty

- $MTBF_{i,j} = \frac{\#\,of\,faults}{operational\,time}$

    - Mean Time Between Failure for the $i^{th}$ checkpoint, at the $j^{th}$
      level of difficulty

- $FR_{i,j} = \frac{1}{MTBF_{i,j}}$

    – Failure Rate of the $i^{th}$ checkpoint at the $j^{th}$ level of difficulty

- $R_{i,j}(t) = e^{-FR_{i,j} \cdot t}$

    – Reliability Function of the $i^{th}$ checkpoint at the $j^{th}$ level of
      difficulty, i.e. the probability that the Controller $C_i$ is not failed
      at time $t$ when operating at difficult $j$

  - All these measures are aggregated to compute overall performance
    metrics of a Controller $i$, without being specific to the difficulty
    level, but still computed separately for deeper examination of the
    Controller's behaviour in different environmental conditions

When measuring the reliability function $R(t)$ of a system, one of the
main measure of interest is its Mean Time To Failure, because it is easy
to compute in simulated environments, and it's used to compute the
rate $\lambda$ of the exponential function. In the Automotive Sector however,
data are usually computed with respect to the travelled distance, i.e.
Mean Distance to Failure, rate of crashes per kilometers, and so on. With
our approach, if the simulated environment and the hardware running
the simulations are powerful enough to run the simulations at a fixed
time-step, it's very easy to switch the point of view on the data.

As long as the neural network is trained properly, we expect the follow-
ing disequation to hold: $R_i(t) \leqslant R_j(t)$, where $i$ and $j$ are 2 checkpoints,

with $i < j$. This can be easily verified using the approach defined above to collect the data.
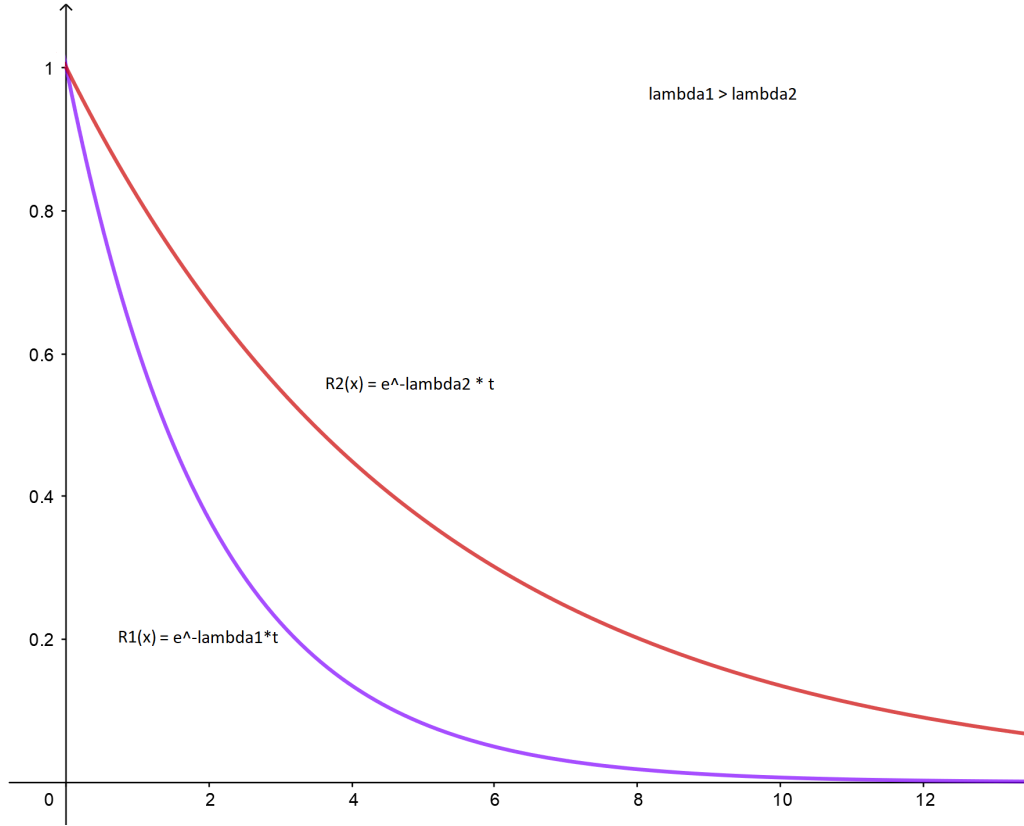


Figure 12: The Reliability function measures the probability that at time t the system is still operating. We expect that more expert drivers (i.e. more trained networks) are capable of longer runs than little trained networks

If the Simulator used for testing allows it, other data should be recorded too, in order to enhance the comprehension of the Controller's behaviour, such as:

- The car's instantaneous speed and acceleration vector at each frame

- With *what* the car crashed (e.g. a vehicle, a pedestrian, a generic obstacle...)

- Define one or more *destination goal(s)* and record whether the vehicle reached it or not

- Environmental conditions at time $t - x$, if a crash occurred at time $t$

These data are fundamental to distinguish between *"safe"* and *"catastrophic"* failures. For example, if a fence is hit at a speed, let's say, less than 10km/h, it may be flagged as a less serious crash than hitting a pedestrian at the same speed.

As the Controller learns, we expect the *Reliability Function* and the *MTBF/MDBF* to increase, since a trained driver should in principle cause less crashes than a non-expert one, therefore increasing the value of the MTBF and possibly the $y$ values of $R(t)$.

If the *black box* makes use of enhanced data (such as the ones listed above), it is also possible to observe changes in the car's behaviour with respect to the scenario and the difficulty level. For example: if we define a difficulty $k$ by doubling the number of cars in the scenarios, it may happen that it is observed an increasement on the amount of collisions against other obstacles. If that's so, the simulations should be studied more deeply to enforce the training strategy in a certain direction, as this may mean that the Controller is going to crash on walls while trying to avoid other vehicles.

### 3.2.2  *Monitor Testing*

Once the Controller's runs are recorded as described previously, the Monitor testing can start.

The goal of this phase is not only to see how good the Safety Monitor is in preventing crashes, we are also interested in observing evidences about which situations are *"hard"* for the Monitor, and which for the Controller.

As the network becomes more expert, it may happen that its behaviour evolves in a way such that the Monitor is no more able to detect imminent failures, because the Controller is now good enough to cover all the failures previously covered by the Monitor, and the hazards caused by its novel behaviour are such that the Monitor is not able to detect them.

However, the opposite is also a concrete possibility. For example, in the first epochs the Controller may drive in a "crazy" manner, e.g. with a lot of sudden, high-angle steerings and riding at high speed. The Monitor will obviously have more problems in predicting what the next state will be due to the unpredictability of the Controller's behaviour. The more the network learns, the more it will (hopefully) ride smoothly, making it easier for the Monitor to detect possible, imminent crashes.

The main problem that should be addressed here is that the Monitor effectiveness may decrease, while the network is learning, resulting in a useless component that may even be detrimental to the system's *performability*: the Controller may become good enough to be able to cover all the hazardous events covered by the Monitor at an earlier checkpoint. This may not change the whole safety of the system, if we consider the actions taken by the Monitor as inconditionally safe, but it would result in lesser smooth rides, due to the safety-brakes applied by the Safety Monitor.

The Monitor is tested as follows: the black boxes created during the Controller testing phase are used to *repeat* the runs. The initial conditions must be *identical* as well and should be saved as part of the black box in the previous stage. The runs of the Controller previously recorded are now repeated, attaching the Monitor to the System and recording the alarm raised during the run and if it was able to prevent the crash that originally occurred.

In this stage of the analysis, the major concern was that of *non-intrusiveness*. For the reasons pointed above, we can not think about rerunning the simulations just by attaching the Safety Monitor and watch how it goes. The Safety Monitor, by definition, *overwrites* the Controller's action if an alert is raised, changing the next part of the run. This is theoretically not a problem, if one can distinguish between false and true positives. However, we can not know in advance what a false positive will be without creating a software capable of sensing and mapping the environment. But this is a Safety Monitor itself and, as every tool of this kind, it will have false positives even if extremely low, therefore this approach can not solve the problem.

The idea is to record the alerts generated during the run, without enabling the safety-procedure (e.g. the brake). Since we are going to repeat exactly the runs recorded in phase 1, it's possible to know the instant of time t in which the transition to the *alert state* before the crash occurred. With this information, all the alerts raised before t are indeed false positives, or false alarms, since we know t. Enabling the Monitor to activate the safety-procedure after this instant of time, makes it possible to approximate its coverage.

As pointed in the previous chapter, to measure how good a model is in classification problems, we observe its predicted values with respect to actual values, visualized using the confusion matrix. However, for *real time critical systems* it's not easy, if not impossible at all, to measure all

of them. In particular, it's not always possible to understand when the Controller avoided a crash and the monitor *did not* raise an alarm. This makes very hard to measure the amount of true negatives because in most of the cases, this kind of situations may be seen only when a human operator analyse the specific run, introducing a level of arbitrariness when computing measures such as *true negatives*.

In particular, an input for the Safety Monitor is composed by multiple, timed inputs, that can be seen as the evolution of the environment while the system is performing its operations. For this reason, it is hard to understand whether the next state will be an alert or a safe state because we don't know *when* the series of inputs that will eventually result in a crash will begin.

This puts some limits on the metrics and the rates that can be computed for the Monitor as they require the amount of true negatives.

At the same time, the Safety-Monitor classifies the system's state *in each frame*, receiving LiDAR data, processing them, and deciding whether a safety-brake is needed or not. Following this reasoning, we can consider every frame in which the Monitor doesn't raise an alert as a *true negative*, due to the *real-timeness* of the System. This approach allows us to compute all the measures usually defined for statistic classification models.

Since the order of magnitude of *true negatives* is likely to be much bigger than the the other metrics, for the reason just described, we had to combine together true and false positives and negatives in order to have meaningful values, e.g. Precision and Accuracy. This is because false positive rate is expressed as $\frac{FP}{FP+TN}$, resulting in values with many zeroes before the first meaningful digit. Moreover, to provide an intuitive rate measure for false positives, it was decided to compute the amount of false positives over the distance travelled in a run.

The rates of correct/wrong Monitor's predictions were computed as defined in Chapter 2, and they can be used for the purpose of comparison among Checkpoints:

- Sensitivity and Miss-Rate

- Specificity and Fall-out

- Accuracy

- Precision

- Matthew's Correlation Coefficient

- False Positives per Meter

- Precision-Recall Curve

*Accuracy* and *Precision* were chosen following an ISO standard. The concept of *Accuracy* is replaced by the *Trueness* of predictions, and *Accuracy* is defined as a description of the combination of *random* and *systematic* errors, hence requiring high accuracy and precision. [37] Matthew's Correlation Coefficient is defined as the most informative metric to evaluate a *Confusion Matrix*, as it is computed as a combination of all the predictions (true and false positives and negatives) and it proved to give more accurate overall informations than $F_\beta$-score. The reason why we decided to don't consider $F_\beta$ is that, in general, it doesn't work well with unbalanced datasets providing results that are too optimistic or too pessimistic, based on the weight $\beta$. Due to the *huge* amount of *True Negatives* and since we are interested in the *overall* Monitor's effectiveness we decided to use MCC as a measure of the Monitor's performances. [38] False Positives per Meter is used to estimate how the System would behave in a context where both the Controller and the Safety-Monitor are enabled and working together, estimating the amount of wrong safety-brakes applied by the latter. Plotting the ROC[2] curve, a graphical plot used to visualize the diagnostic ability of a classifier tool, would have provided a mean to quickly visualize the Monitor's effectiveness, however this approach was not feasible. A ROC Curve is made by defining the x-axis by FPR, and the y-axis by TPR. The plot is then split by the line $y = x$: values over this line depicts the ability to identify positive cases as such, values under this line represents predictions worse than random predictions, while values on the line represents random predictions. Due to the high imbalancement of our dataset, where the false positive rate is likely going to get values lower than $10^{-2}$, due to how true negatives were defined, the plot would be flattened "on the left side".

For this reason, we decided to use the *Precision-Recall Curve* to provide a visualization mean, a curve obtained defining the x-axis by *Recall* (True Positive Rate) and the y-axis by *Precision*, using data collected for each Checkpoint in all the difficulties.

When testing the Monitor, it is advisable to *"stress"* it under detrimental conditions, in order to collect more possible evidences and linkage among

---

2 Receiver Operating Characteristic

the data. Note that this aspect is independent from the *difficulty level* defined above, as it is something related to the environment itself that may give evidences about the *situations* in which the Monitor performs well, and those in which it doesn't (e.g. think about a LiDAR sensor in a very rainy environment: the observations will likely to be fuzzy/noisy) and it is more related to *fault injection* at a higher level.

The ways in which the parameters of a Safety Monitor can be tuned really depend on its implementation, and it's up to the developing team the decision on what and how many faults inject in the software. A very simple example can be that of reducing the amount of data read by the sensors (which will result in low-quality data and a poor mapping of the environment), or introduce some noise in the observations.

Doing this step when testing the Monitor allows also to understand how to tune the internal parameters of the software in order to have its *"best"* version, that will be used in the Controller retraining phase.

### 3.2.3  *Controller Retraining*

At this point we have collected data about the behaviours of the Monitor and the Controller. In this phase we are going to study how much this values vary with respect to the learning strategy adopted, when the neural network is retrained from the last checkpoint.

As said at the beginning of this chapter, we are considering a Controller composed by a neural network, trained with reinforcement learning techniques. These kind of algorithms make use of a *reward function* to tell to the network if it's behaving *"good"* (positive value) or *"bad"* (negative value) and it is calculated at each prediction step (i.e. at each action taken by the Controller).

The training function needs the following parameters, that are recorded at each algorithm iteration:

a) Action taken by the Network

b) State of the network in which the action was taken

c) Reward given for the action taken in a certain state

In this phase we want to discover how training strategies for the neural network may affect the overall behaviour of the system, with particular attention on the effectiveness of the Safety Monitor.

To start an exploration in this topic, we defined 4 strategies and the outcomes we expect after the training is completed. These outcomes are *"forecasted"* in the sense that we don't know how the network will react to a change in the training strategy, nor if the observe behaviour will be the supposed one.

The strategies developed in this method are mostly based on the reward function and on the actions taken by the network:

S1) The reward function is now more punitive when the car hits something, while braking is slightly more rewarded, provided that the Car won't stop moving

S2) The Safety Monitor is attached to the system and, if an alert is raised, the action of the network is replaced by the Monitor's action (the safety-brake)

S3) The network's action is replaced by the monitor's if it raises an alert. The network is given a positive reward for behaving like the Monitor

S4) The training step is stopped and the network is given a *negative* reward if an alert is raised

Using S1 we expect at least the Controller's mean time/distance between failures to be lower. Giving a more negative reward for collisions and a little positive reward for braking (if the car doesn't stop moving) should force the car to prefer a brake instead of a swerve (that could result in a *new* hazardous situation), therefore improving its time between failures and its travelled distance.

The purpose of S2 is to *"teach"* to the network to brake whenever an alarm is raised by the Monitor. A possible outcome is that the alert states previously covered by the Monitor become now safe states. However the Monitor's false positives will inevitably have an impact on the system's availability, caused by erroneous safety-brakes.

S3 is pretty much similar to S2. We expect that giving a positive reward when behaving like the Monitor may increase the learning speed.

S4 is the most promising strategy and it may give the most interesting results. Giving a negative reward and alting the training step when the Safety Monitor raises an alarm, regardless if it's a true or false positive, should force the network to completely *avoid* the situations in which the Monitor intervenes. Essentially we expect that the effectiveness of the

safety component will be drastically reduced.

After the four Controllers are trained enough, the new systems are tested as in phase one. The same measures listed before are approximated for the Controllers and the Monitor and the results are compared with the original checkpoints.

# 4

## METHOD IMPLEMENTATION AND RESULTS

In this chapter the tools used, the software infrastructure and method implementation and the results collected during the analysis are reviewed.

A DDPG Agent[1] was trained to drive in an urban environment. Checkpoints of the network's state during the training were recorded for the purpose of comparison. These checkpoints of the network were then tested with and without a simple safety-monitor in order to provide a new point of view to study AV's behaviours.

### 4.1 TOOLS AND SOFTWARE

#### 4.1.1 *Carla Simulator*

In order to have a realistic environment, with accurate physics simulation and data sensors, the open-source simulator CARLA[7], developed by researchers at the University of Barcelona, was used. This simulator was developed with the purpose of offering an environment where AI agents can be trained to drive, with high control of the simulation parameters and the simulation of realistic sensor, which can be tuned to increase or decrease data quality, or to inject faults.

CARLA is developed with a client-server architecture in mind. The *server* is basically a game, developed with *Unreal Engine 4* in C++. C++ performances are with no doubt essential to the functionality of the server: not only the environment must be simulated (including movements of pedestrians/vehicles, weather simulation...), but also all the data needed from the sensors attached to the system.

---

1 An agent whose actions are taken by a neural network trained with the reinforcement learning algorithm DDPG

Figure 13: CARLA logo at carla.org

CARLA is currently at version 0.9.7 and huge improvements are done at every release, gaining more attention from the experts for its realism. Unfortunately, when this study started, CARLA 0.9 was recently released and the tools needed for our work couldn't be found online. Thanks to the quantity of work done for the last *stable* version of CARLA, 0.8.4 was used at first.

Versions prior to 0.9 have some limitations on the control one has of the simulations parameters and on the data collectable from it. This doesn't impede our study, but of course limited in some way the informations on the environment and system. Some of these problems are still present in later versions of the simulator, but most of them were solved in the transition from 0.8 to 0.9.

One of the main problems found was with the coordinate systems. Before version 0.9, developers were using UE4's default coordinates system which is left-handed, while the standard is considered to be right-handed. This looks like not a big deal since things could be easily solved by applying a transformation matrix. However, due to performance issues

(a Python client should do the real-time processing of *loads* of data at each timestep, resulting in considerable slowdowns as a result of all the processes running at the same time), it was decided to stick with the developers' decision and convert the data during analysis phase.

Unfortunately, this version of CARLA has only 4 sensors available, which were all used during the experiments. They can be easily accessed via the Python APIs provided:

- Cameras
    - The *scene final* camera provides a view of the scene (just like a regular camera)
    - The *depth map* camera assigns RGB values to objects to perceive *depth* of the environment
    - A *semantic segmentation* is used to classify different objects in the view by displaying them in different colors, according to the object's class

- Ray-cast based Lidar
    - Light Detection and Ranging is use to sense the environment and measures distance from objects by illunating the target with laser beams and measuring the time reflected light needs to "go back" to the sensor

The three cameras were used during the training phase of the network. Three *scene final* cameras are attached to the car to actually *see* the environment (one on the front and one per side). The *depth map* camera allows the car to get a colormap of the distances from objects in the scenario. The *semantic segmentation* provides image classification features by querying the server for ground-truth values. This is with no doubt a semplification of a real system, where the most powerful image-classification softwares are essentially other neural networks trained separately. At the same time a misclassification can be considered as an error of the control system: if the safety monitor detects the possible hazard it will not "correct" the misclassification but it must react fast and safely to avoid the possible consequences of it, therefore this simplification won't have an impact on the overall method.

A ray-cast based Lidar is the only other sensor available for this version of CARLA. Parameters of this sensor can be easily tuned to simulate real lidars such as the *Velodyne LiDAR* or to simulate faults such as low data

quality, noisy data or data loss... This data are generated using the *Point Cloud*[20] format

In the simulations, due to the high hardware resources requirements to simulate a real LiDAR, a slightly modified version of the *Velodyne64 LiDAR* is implemented with the following parameters:

- Channels = 64

  - The number of laser beams used by the system. These lasers are distributed over the vertical axis. The more the lasers are, the more accurate will be the scannings

- Range = 75m

  - Lasers' range in meters

- Rotation Frequency = 15 Hz

  - This parameters define the rotation frequency (in Hz) of the scanning beams.

- Points Per Second = 1.000.000

  - The actual number of points generated each frame by the sensor

- Vertical FOV bounds (height = 24m, low = -2m. Distances are relative to the position of the sensor)

  - Maximum and minimum height of the scannings

The simulator provides Python APIs not only to modify sensors, but also to have a great control on what is being simulated, such as seeds definition for the spawning points and the behaviours of pedestrians and vehicles, and on the state of *"actors"* in the scene such as their position, their speed.... All these data are directly provided by the simulator with ground-truth values. These kind of measurements can be simulation-related, such as the simulation time-step, or the FPS. Actors-related measurements include for example vehicles' speed, intensity of collisions (if any) and the 3D acceleration vector.

While this tool was being tested a problem was found with the LiDAR sensor data that is still not resolved.[8] The bug resulted in the vehicles' bounding boxes to be distorted when moving, providing very poor data. After some research an improved version of the simulator was found,

where the bounding boxes for each vehicle were redefined to provide accurate data.[9] As of today, developers are working on this issue but it is still unresolved without manual intervention on the source code.

### 4.1.2 *Controller Implementation*

The main component needed for the Controller implementation is the neural network that will be in charge of deciding the action to perform to drive the car. We looked for a framework with the following characteristics

1  Training code must be available

2  No critical issues in the codebase

3  Provide an environment to interface the network with CARLA

4  Provide a default training strategy

After analyzing all the machine-learning related projects for CARLA, we chose the reinforcement-learning framework *Coach*, developed by *Intel AI Lab*.[10]

This framework satisfies all the requirements listed above: it is distributed as an *editable Python package*. The development team behind this project assure us the quality of the product. Moreover, several presets are available as a starting point. Two of these presets offer an interface to the CARLA simulator, as well as a default training strategy, which is exactly what we required.

These presets implement the *Deep Deterministic Policy Gradient* algorithm, proposed in 2015.[19] This algorithm proved to perform well in tasks such as car driving, as shown in the original paper, and it's specifically adapted to perform in environments with continuos action space, such as the one we are considering.

P1) The first preset uses a single, front camera to perceive the environment, and the other data augmentation cameras provided by CARLA. The agent resulting from using this preset will likely not have any sense of depth created by the regular camera and will rely solely on the depth camera

P2) This preset uses all the cameras available in CARLA and have a much more interesting architecture since the car is equipped

with three regular cameras (Front, Left, Right) and all the data augmentation cameras available in CARLA, listed in the previous section

The usage of the data augmentation cameras (depth and semantic segmentation) allows us to ignore objects misclassifications (e.g. labeling a pedestrian as a light pole) as well as some other easinesses such as computing distances from objects. As pointed in the previous section, this is indeed a simplification of a real architecture, where object detection modules are neural networks themselves, that must be trained in advance and tested separately. However, again, a misclassification will probably cause the Controller to behave in unexpected manners that the Safety Monitor must be able to detect and possibly correct.

Unfortunately, the first preset suffers from an issue that causes the car to stop throttling, while being positively rewarded[18]. It would have been interesting to test this preset as well, to understand what's the impact on the effectiveness of the Safety Monitor with respect to the quantity of data available to the network.

It's important to notice that our goal *is not* to build the *"perfect"* agent, nor the perfect autonomous cars. The codebase was studied but, due to reasons of time, we could not inspect all the details of the provided implementation. This framework was used as an example to demonstrate the concepts described in this work.

### 4.1.3    *Safety-Monitor Implementation*

In order to start the experimental activity, we needed a software capable of processing LiDAR data to map the environment and sense the obstacle. Moreover, this software needs to be *real-time* and capable of interacting with CARLA. The latter is obvious: whenever an alert is raised, the *"brake"* command must be sent to the CARLA server. The first required some reasoning: one could think to record the LiDAR data in advance and then run the simulations feeding the Monitor with these data. Unfortunately this approach can not work: this comes from the fact that we are not only interested in having a 100% prediction accuracy, but also in the *goodness* of the safety-measure adopted by the Monitor. The same measurement in two different simulations could give *slightly* different results, that could cause the Monitor to behave in a completely different way than the way it could have behaved using the *real time* data produced during

the simulation. If the accuracy of measurements is strictly dependent on the simulator used for the experiments (CARLA is improving at every release on this side), we could not ignore the effects that a brake can have on a single run.

In order to develop a decent Safety-Monitor, a research on the best, *"not-neural-network"* techniques was conducted, as well as a research on the open-source instruments available. The choice fell on the *Point Cloud Library*[22], an open-source library specifically dedicated to the processing of Point Cloud data, developed using C++ to achieve great performances when processing huge amount of data. This library was first released in 2011[21] and it's been improved at every release, also thanks to the big community testing and debugging the new features.

This library offers a multitude of functions that implements the most known techniques for Point Cloud processing. The steps to follow in order to develop an object-detection module are the following:

1) Downsampling

    → The data generated by a single scan can contain more hundreds of thousands records, with possibly a lot of redundancy and noise. A downsample is usually necessary as a preliminary step to discard all the *"useless"* data. After some reasoning, the Voxel Grid Filter was chosen for this step

2) Ground Segmentation

    → After the downsampling (if needed), the first mandatory step is to filter the data that are useless for the purpose of object detection, such as points relative to the ground. These point must be filtered in order to separate the ground from the objects we want to detect. In our implementation this is done using the RANSAC[2] algorithm[? ], a technique to separate *"inliers"* (i.e. data whose distribution is characterized by the model's parameters) from *"outliers"* (i.e. data that don't have a representation in the chosen model)

3) Clustering

    → This final step is required to effectively identify what is an object in the scene, and what data points are relative to the same object. This can be done using clustering algorithms

---

2 Random Sample Consensus

based on the distance among points. We chose the Euclidean Clustering Algorithm, a range search algorithm based on the euclidean distance between points, with the assumption that *very dense* points represent the same object

4) Object Tracking and Avoidance

$\rightarrow$ Objects detected in the first three steps must be tracked over time in order to recognize if two objects observed in two consecutive steps, are actually the *same* object, usually using physical models to predict their behaviour over time, such as the Kalman Filter, and combine this model with a failure-prevention routine

Unfortunately, implementing a Kalman Filter for such a complex model would have been too much time-consuming, alting our study. Therefore we simplified the object-detection and the failure-prevention routine in this way:

- Only data *in front of* the car are recorded. This is a huge semplification of the model, as the Monitor now can only detect obstacles ahead of the car. However, this will for sure have an impact on the effectiveness of the Safety Monitor, but the concepts explained in the previous section still holds

- The failure-prevention routine implemented takes inspiration from the Responsibility-Sensitive Safety model proposed by Mobileye, an Intel company.[23] If an object is detected, its speed relative to the system is computed. If the distance travelled by the system in 1 second plus the braking distance is greater than the distance travelled by the object plus the distance between the system and the object, a safety brake is applied.

Since the Safety-Monitor is developed in C++, both for taking advantage of the Point Cloud Library and due to performance reasons, an infrastructure to exchange data with CARLA was developed.

The software is composed of a C++ server that receives LiDAR Point Clouds from the CARLA client. This data are processed as in the steps above (with the semplification described) and, if needed, an alert is sent back to the client. If the message received by the Monitor is an alert, the actions of the Controller are ignored and a brake is performed.

The object detection module is inspired by an open-source project developed by Engin Bozkurt[30]. The codebase was deeply modified to

adapt the detection algorithm parameters to fit our needs and to interact with CARLA.

We are conscious that this Monitor will likely not achieve "state of art" performances and it's likely to produce a lot of *false positives*. This aspect may have some consequences on the traning efficacy, but the concepts explained in this work still holds.

## 4.2 EXPERIMENTAL ACTIVITY

In this section is described how the methodology developed in the previous chapter was implemented, and the technical problems found during the implementation.

The Controller was trained in an urban environment, with the default strategy provided by *Coach* in order to generate four checkpoints to be subsequently tested.

Scenarios were generated using the 152 spawn points[3] provided by Carla. For each spawn point, a default setting and three variations of it were generated:

h0) *Default Setting*: the map is generated using the *same* conditions in which the Controller was trained, with 30 pedestrians and 15 cars

h1) *Pedestrians Setting*: the map is generated increasing the number of pedestrians from 30 to 60

h2) *Vehicles Setting*: the map is generated increasing the number of vehicles in the environment from 15 to 30

h3 ) *Pedestrians and Vehicles Setting*: the map is generated merging h1 and h2, resulting in an environment with 60 pedestrians and 30 vehicles

For repeatibility, and consistency across the variations, 152 couples of unique seeds[4] were generated, each of them was associated to a starting point. This allows to use the same seed, for the same starting point, using different variations.

To enrich the informations about the System's overall performances, the car is also given a *Destination Goal* to reach, which is also recorded for repeatibility. If the Goal is reached, the success is recorded and the

---

3 Coordinates in the map in which the car will start its ride.
4 These seeds are used in the generation of pedestrians and vehicles

System is given a new destination to reach.

In general, it may require a huge amount of time for a crash to happen. For this reason, in order to demonstrate the concepts explained in this work, it was defined a maximum running time of 15 minutes, after which the System's task is considered succeeded.

In the first phase of the analysis, four checkpoints generated were tested according to the methodology described in section 3. Since the *Coach* project is open-source, the codebase was freely available. This allowed us to use *software probes* to monitor the Controller's activity. The source code was modified adding some instructions to write all the needed data into different file for each run:

- The starting point and seeds of the Scenario

- Actions taken by the Controller

- If there is a collision, with what

- If a goal is reached, record it and the new goal coordinates

Modifying the source code may in general result in a worsening of the overall performances, and the accuracy of results. Fortunately, the CARLA simulator provides an option to run the simulations at a *fixed-time step*. This was set at its minimum: 10 FPS[5], to assure that all the data received from the server are *timely* and *correct*. This also serves as a mean for *repeatibility* of tests: a *variable time-step* may introduce randomness in data since we have no control over it.

The first training phase and the Controller testing required circa 1 month and a half to be performed.

The second step was to test the effectiveness of the Safety-Monitor for each checkpoint.

The source code was once again modified to create two separate, concurrent processes: one takes LiDAR data from the CARLA server and sends them to the Safety-Monitor server; the other waits for the response of the Safety-Monitor to check whether a brake is needed.

The LiDAR data are generated by the CARLA server at each frame. The workload for data generation relies solely on the CPU, resulting in very much slower execution times. Unfortunately, CARLA suffers from a

---

5 Frames Per Second

bug which causes data to be inaccurate, if FPSs are lower than 10. This required a powerful machine to guarantee the lower bound of 10 FPS. The source code was once again modified to insert software probes in the codebase to gather informations about alarms raised by the monitor.

Runs recorded in the first step are now repeated observing the Safety-Monitor's behaviour. All the alerts raised during the run are considered *false positive*. The emergency brake is enabled 2 seconds before the collision, to estimate the amount of *true positives*.

This approach, allows us to estimate the goodness of the Safety-Monitor in its operational environment.

After the first phase is concluded, data gathered are processed to compute the measures listed in the previous Chapter:

- *Controller's Checkpoints*:
    - Mean Distance Between Failures
    - Mean Time Between Failures
    - Failure Rate
    - Reliability

- *Safety-Monitor*:
    - Confusion Matrix of predictions
    - True/False Positives/Negatives Rates
    - Accuracy
    - Precision
    - Matthew's Correlation Coefficient
    - False Positives per Meter

At this point, training is resumed from the last checkpoints using the strategies described before, and the measurement process is repeated identically.

Some of the strategies defined requires the training to be done *with* the Safety-Monitor, resulting in very low execution times. This phase took approximately 3 months because of the heavy computation load needed to generated LiDAR data, after which the training was stopped.

The numerical results collected can be compared to check if the network is learning properly and how the Safety-Monitor's effectiveness changes when the network improves.

Data collected were processed using *Python* scripts, to aggregate data in order to produce metrics of the *overall* performances of the Controller and the Safty-Monitor. Data aggregation is fundamental for this kind of activity. At the same time, due to the complex nature of these systems and the fact that many hazardous events are *rare events*, it is *necessary* to study specific runs individually and to separately compare their performances before aggregating data as well (e.g. compare performances for the same Controller in each difficulty level).

In the next section we present collected results mostly in aggregated form, due to the huge amount of data collected. However, there are considerations on *"raw"* data that can not be shown in this work, but are attached separately for examination.

## 4.3    RESULTS

### 4.3.1    *Phase 1 - Controller Testing*

The data collected are saved in separate files for each run and processed using Python scripts, computing the measures listed in Chapter 3.

The measures collected are compared to check the goodness of the Controller and changes in the effectiveness of the Safety-Monitor over the neural network's checkpoints.

Since the Checkpoints were tested in the *same* scenarios, the distances travelled for each scenario were plotted to check if meters travelled are increasing and if there are scenarios somehow *"favorable"* or extremely *"hard"* for the system. This first step allows testers to filter between critical scenarios and easy scenarios. Ideally, given a Scenario $x$, if the distance travelled by the Controller in $x$ is lower than the average distance travelled in all the checkpoints, there may be two cases:

1) In the case that the Controller always travels the same distance in all the checkpoints before crashing, there may be a hazard that the Controller has not learned to handle

2) In the case the distance travelled fluctuates in a certain (small) interval, the happening of a crash may depend from the inital conditions

Regardless of the case, with this approach it's easy to see which scenarios make the Controller crash soon, and they can be examined individually with more attention.

It is important to keep in mind that if a trend is observed when testing different Checkpoints, e.g. a scenario in which the Controller *always* performs extremely good or extremely bad, an analysis of the specific case is needed, in both of the cases.

The reason is obvious for "bad" runs, for the reason listed above. It may seem counter-intuitive, but repeated "good" runs in the same scenario requires particular attention. An analysis of the correlation between performances and initial conditions is mandatory, and it may be demonstrated or confuted with this approach. A more complex case is when the System finds a *"lucky pattern"*, e.g. a run in which the car just follows the same pattern without finding any car on the road.

Looking at charts of distances travelled in each scenario for Checkpoint 1 and 4 at *standard difficulty* below, it's obvious that the order of magnitude is quite different between the two checkpoints (C1 never travels more then 400 meters, while C4 has several runs longer than 1 km) and in a couple scenarios it has been reached the maximum length defined for these experiments. Another very interesting fact observed with this approach, is that the lower bound on distance travelled doesn't change for C2...4 and it's stuck at 22 meters in *all* the difficulties. A deeper analysis on travelled distances and running times shows that there are runs in which the Controller doesn't make any progress, as an indicator that the event causing a crash may be of a particular kind, which is hard for the Controller to handle. Results of this kind are with no doubt interesting in understanding if the shortest runs have some environmental conditions in common, which may cause the Controller to perform badly.
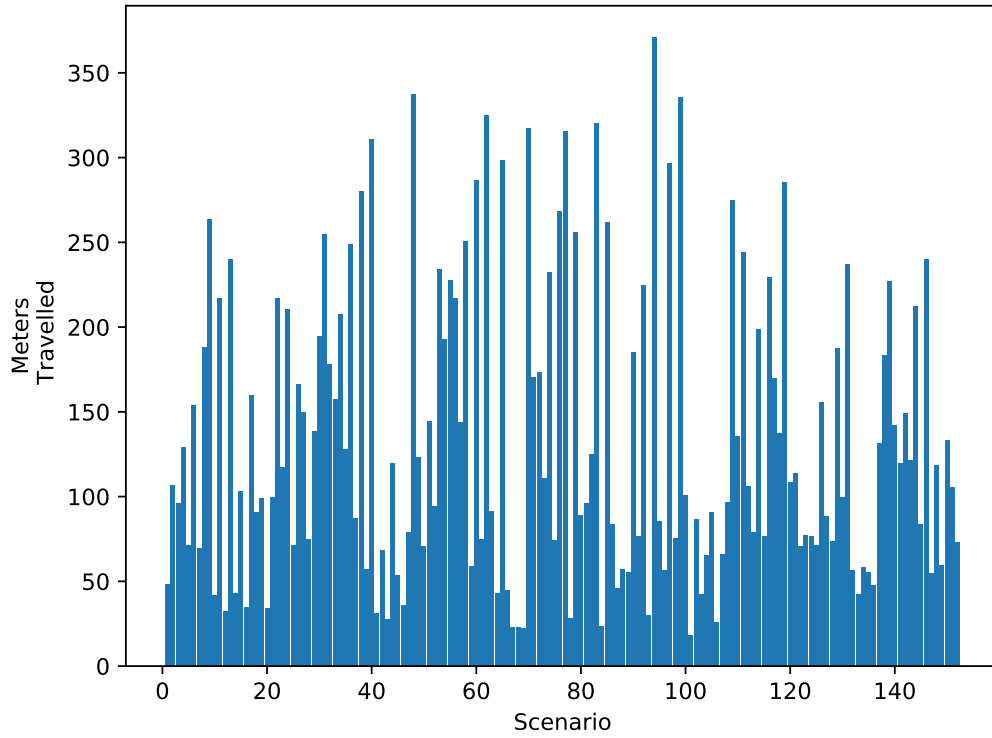
Figure 14: Meters travelled by Controller C1 (first Checkpoint) in each scenario at standard difficulty
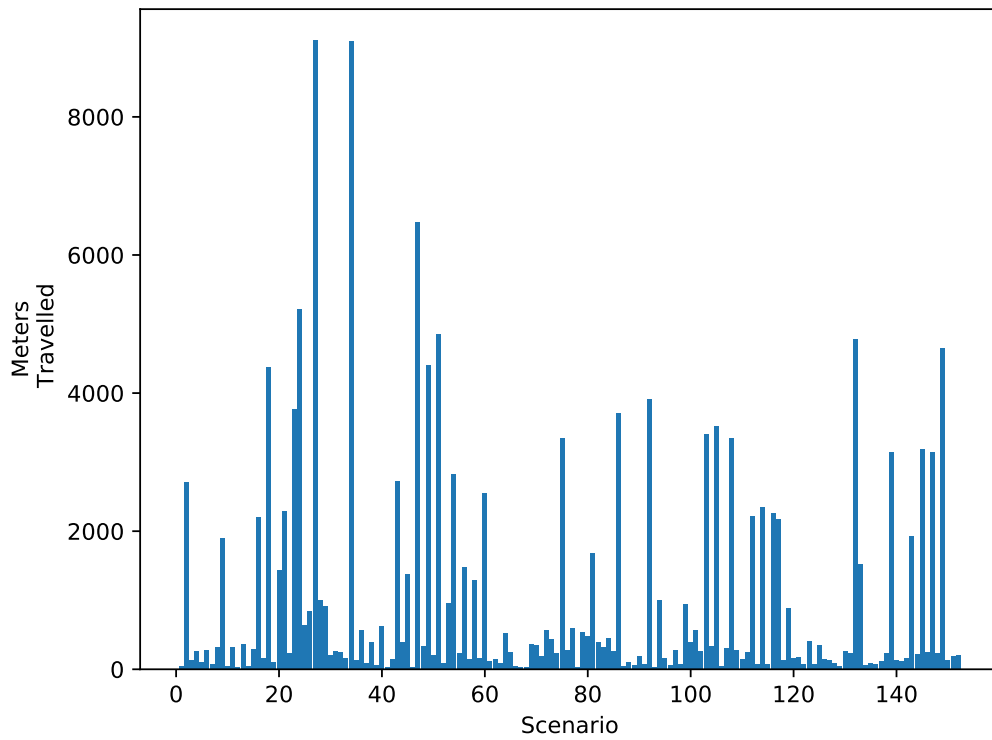


Figure 15: Meters travelled by Controller C4 (fourth Checkpoint) in each scenario at standard difficulty

The use of a *fixed time-step* when running the simulations, i.e. the simulation time elapsed between two steps of the simulation is fixed and known, gives us knowled of the interval of time between two consecutive actions: observing how many actions the Controller has taken it's possible to compute the *true* total time of a single run.

In this way, we are able to easily approximate the *Reliability Function* for each Checkpoint, based on the average MTTF in each level of difficulty.
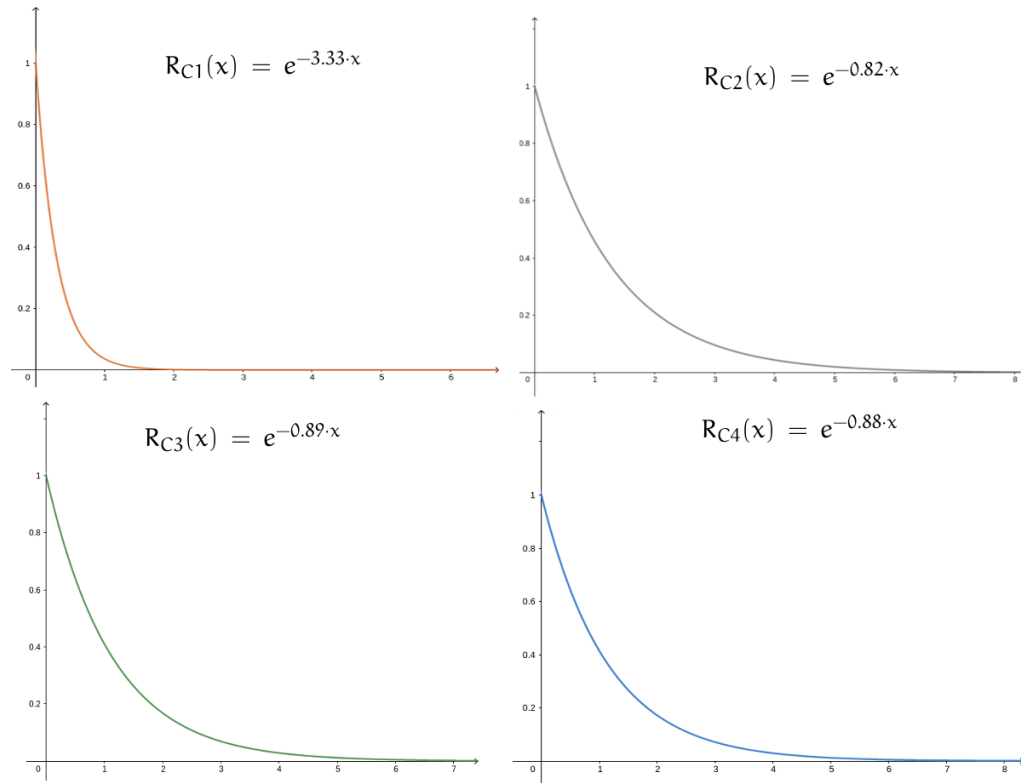


Figure 16: Graphics that shows the probability y of the system being operational, after x minutes of operation

As we expected, the MTTF of the first checkpoint is very low (20 seconds) and increases over Checkpoints. Surprisingly, the second Checkpoint seems to have better performances (higher MTTF) than Checkpoints 3 and 4. However, analyzing a subset of the runs in which the second Checkpoint achieved very good performances in terms of time to failure and distance travelled, it can be seen that its driving is way more dangerous than the other two, resulting in a bad driving style in which most of the times the car avoided a crash by steering at the very last moment. As another indicator of the fact that C2 is not really better than C3 and C4

is that it reached *Goals Destinations* fewer times than C3/C4, as another evidence that the *apparent* higher safety is likely to be a result of *"lucky patterns"* in the sense defined above (Goals reached charts are shown in *Section 3*).

Fortunately, as we will show with the results collected in phase 3, the Controller managed to overcome this problem.

The same behaviour is observed for distances travelled by each Checkpoint: longer runs are actually runs in which the car travels more meters. This assure us that the system is not *"cheating"*: a longer execution time may come from the car not moving at all. Computing the average distance travelled by each Checkpoint in each difficulty level, allows us to approximate the Reliability Function in terms of kilometers travelled before crashing.

Moreover, the observed progresses in the *mean distance to failure* seems to follow the progresses of the *mean time to failure*, which is another assurance of the correspondence between the two metrics.
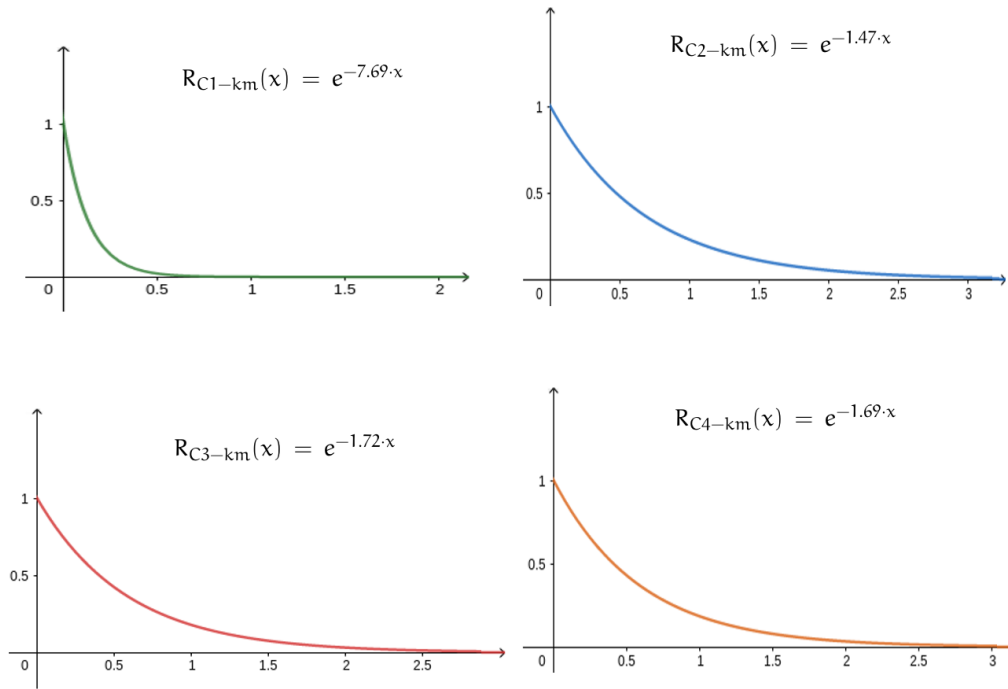


Figure 17: Graphics that shows the probability y of the system being operational, after x kilometers

As another mean to analyze the Controller's behaviour over Checkpoints, since CARLA provides means to record the kind of object with

which a collision occured, rates of kind of object collided with respect to total collisions are computed for each Checkpoint, for each level of difficulty. This procedure serves multiple purposes:

1) To monitor whether the Car crashes with obstacles, indicating that it tends to go off-road by crashing with walls, light poles, fences...

2) To monitor how the Car reacts to diverse level of difficulty, e.g. if we increase the number of pedestrians and we observe for example less collisions with pedestrians, it may be an indicator that the System is capable of avoiding collisions with people.

3) Understand the System's ability to avoid crashes with specific "objects" (pedestrians, vehicles and generic obstacles)



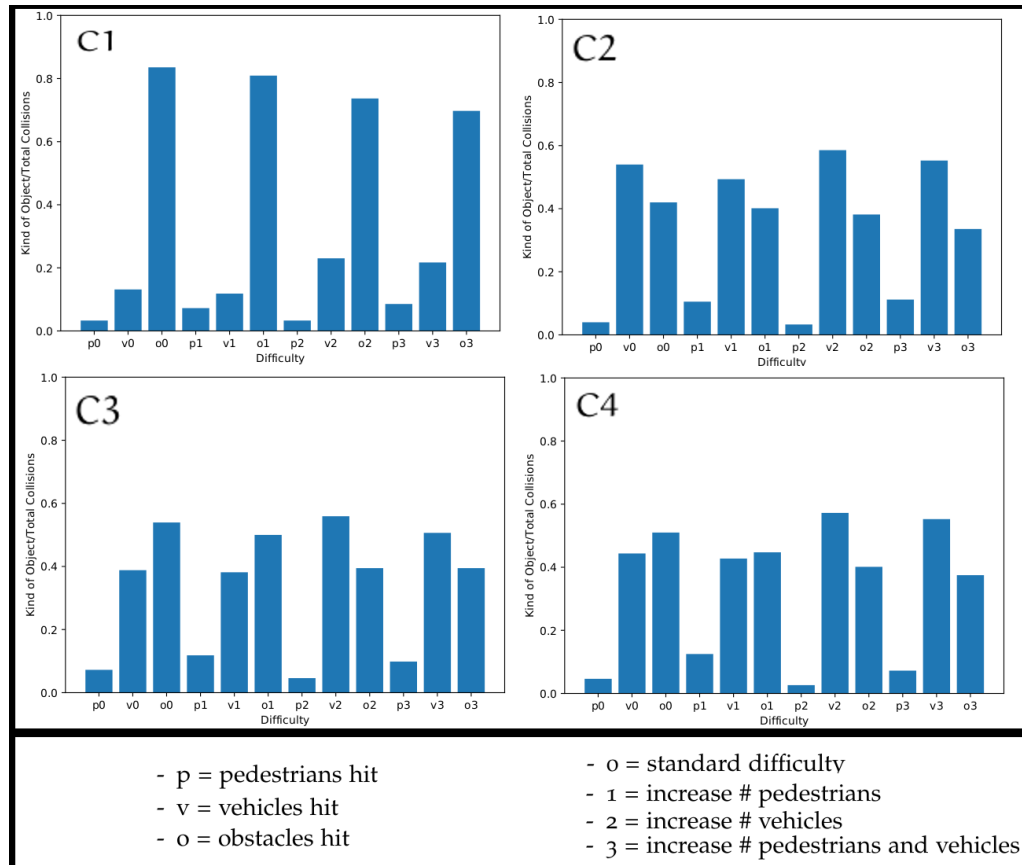Figure 18: Charts that shows $\frac{\#\,kind\,hit}{Total\,Collisions}$ for Checkpoints $1\ldots4$, in each difficulty level

These diagrams are useful to have an overlook of how the System performs when changing the difficulty of the Scenario while being trained

and to observe correlations between the kinds of collision and the Checkpoints. As we could expect, the first Checkpoint mostly collides with off-road obstacles, indicating poor driving skills of the Controller.

It also seems that there is a tendency in which the ratio of collisions with generic obstacles follows the overall Checkpoint's performances. Observing this diagram and those of the Reliability Function, it can be seen that C1 (lowest MTTF), tends to crash against obstacles, C2, which had the best performances in terms of MTTF and MDTF, has the lowest ratio of collisions against obstacles. C3 increased the rates of these collisions, coeherently with its worsening of the MTTF/MDTF, which is subsequently lowered by C4.

The four levels of difficulty defined, even if quite simple, proved to be a good way to assess the System's performances in environments that are different from the training environment. MTTFs and MDTFs are sensibly lower in difficulties 3 and 4 (increasing the number of vehicles, increasing the number of pedestrians and vehicles) giving evidences that situations of dense traffic tend to be harder to handle. At the same time increase/decrease of the hit rates with a specific kind of object, follows the features of the difficulty chosen. This suggests that the crashes observed are strictly dependent on the environmental conditions. Increasing the number of pedestrians resulted in more collisions with these *"objects"* and increasing the number of vehicles shows the same evidence. Increasing the amount of pedestrians and vehicles shows that collisions with other cars is still the main cause of a crash, giving evidences that dense traffic is more likely to create harder situations for the Controller.

These aspects are useful for the definition of more complex difficulties. These measures also allows to detect correlations between the increase/decrease of kind of collisions.

With the methodology defined for the first phase, combining the definition of the Scenarios, the difficulty levels and the metrics chosen, allows to observe many features of the Controller without actively observing its executions. Moreover, combining the few metrics chosen it is possible to have an overlook of possible correlations between environmental conditions, initial conditions, and Checkpoints.

### 4.3.2  *Phase 2 - Monitor Testing*

After the Controller has been tested, and its runs are recorded, it's possible to test the Safety-Monitor in order to approximate its prediction accuracy, but also how it affects the safety of the whole System.

As said before, LiDAR data are collected at each frame and sent to the Safety-Monitor. These data are then processed using the algorithm described in the *Monitor Implementation* section and a message is sent back to the Control System, containing informations about whether braking or not. There is no synchronization between these 2 components, as it is in real systems of this kind: it would not be possible for the Controller to wait for the message of the Safety-Monitor at each frame, as it would result in jerky runs.

The *amounts* of True Positives, True Negatives, False Positives and False Negatives are computed using the procedure described in the previous chapter[6]:

- The Controller runs recorded in the previous phase are repeated attaching the Safety-Monitor to the car

- The Safety-Monitor is allowed to generate alerts, but the safety brake is not performed until the istant of time t in which the System enters in an *alert state*, defined as in Chapter 3.

  - The choice of t is a critical parameter for the accuracy of measurements, as a choice of an *"early"* t may cause a brake for an event that the Controller actually handled, and a *"late"* t may result in a failure of the Safety-Monitor just because the safety brake was enabled too late. Ideally, the safety brake should be enabled as soon as there is a transition from a *safe state* to an *alert state*. Unfortunately it's not always possible (if not possible at all in complex runs) to understand when this transition happened. In this work, t was approximated observing the average speed of the car wrt speed limit and the time needed to process LiDAR data and get the answer from the Safety-Monitor, and was set to 2 seconds

- All the alerts generated before t are considered *false positives*

---

6 We know that *rates* for these measures are linked to each other, but some of the aggregated measures (e.g. *Accuracy*) explicitly requires *each* of these quantities

- All the frames in which the Monitor makes a prediction *and does not* raise an alert, are considered *true negatives*

- If the System managed to survive for the maximum time defined for a single run, but the Safety-Monitor raises an alert after time t, this alert is considered a *false positive*

- A successfull avoidance of the crash (if happened) because of the brake caused by the Safety-Monitor, is considered to be a *true positive*

- If a crash happens, regardless of whether the Safety-Monitor raised an alarm or not, it's classified as a *false negative*

- Once the amounts of true/false positives/negatives are known, they can be combined in more refined measures

The method developed in this work seems a promising mean not only to observe the Monitor's behaviour, but also to understand and validate the hypothesis on the Controller's behaviour.

Analyzing table 1, it seems like the Monitor's effectiveness is influenced by the Controller's reliability. If on one hand the True Positive Rate is stable around 0.75 (except for C2, which anyway exhibited a weird behaviour in its runs), on the other hand we can see how the *Precision* of the Monitor drops after C2 (recall that *Precision* in this setting measures the rate of crash avoided with respect to all the alerts raised), raising some concerns on the long-term effectiveness of the Safety-Monitor. The drop in *Precision* is an indicator of the increase in *false positives* (i.e. false alarms), which makes us wonder on the fact that the Monitor is not able to classify the System's states correctly, due to the novel behaviour of the Controller. This hypothesis is enforced by looking at the MCC of each Checkpoint, that follows the same behaviour of *Precision*, decreasing as the Controller gets trained. As the MCC is a measure of the *overall* performance of the Monitor, we are interested in seeing whether it will decrease again in phase 3, after training the Controller again.

As we expected, this Monitor tends to produce many false positives (stabilizing around 1 false positive per meter). This doesn't affect the effectiveness of the monitoring methodology described, but raises some concerns on the effectiveness of a training strategy using *this* Monitor, and makes it possible to understand how the System would behave when

both components (the Controller and the Safety-Monitor) works together, i.e. with a lot of "wrong" safety-brakes caused by the Safety-Monitor's false alarms.

|      | C1     | C2     | C3     | C4     |
|------|--------|--------|--------|--------|
| TPR  | 0.75   | 0.82   | 0.75   | 0.77   |
| FNR  | 0.25   | 0.15   | 0.25   | 0.23   |
| FPR  | 0.0029 | 0.0029 | 0.0062 | 0.0069 |
| TNR  | 0.9971 | 0.9971 | 0.9938 | 0.9931 |
| ACC  | 0.995  | 0.992  | 0.993  | 0.992  |
| PREC | 0.63   | 0.14   | 0.17   | 0.15   |
| MCC  | 0.68   | 0.34   | 0.35   | 0.34   |
| FPPM | 0.52   | 1.13   | 1.02   | 1.12   |

Table 2: Prediction Rates of the Safety-Monitor for Checkpoints 1-4

Plotting the *Precision-Recall Curve* in figure 19 provides a quick visualization mean to observe the performance drop of the Safety-Monitor, performing well with a *"dumb"* Controller, while becoming almost disruptive with more trained Controllers.

Looking at the curves, what surprises us about the Monitor is that, even if its performances are not good, its best performance is observed at the *hardest* difficulty, i.e. the setting where the amount of pedestrians and vehicles in the scenarios is increased. After C1, the second best performances are observed with the second hardest difficulty for the Controller, i.e. the setting where the amount of vehicles is increased. The correlation between what's *"hard"* for the Controller and what for the Monitor can not be treated exhaustively in this study, however, plotting these curves showed an unexpected behaviour, motivating us to study more deeply this aspect in future works.

These curves are consistent with the data collected and analyzed, as the Safety-Monitor performs well with C1, while almost the same performances were observed for C2...4, with slightly better performances with C2.
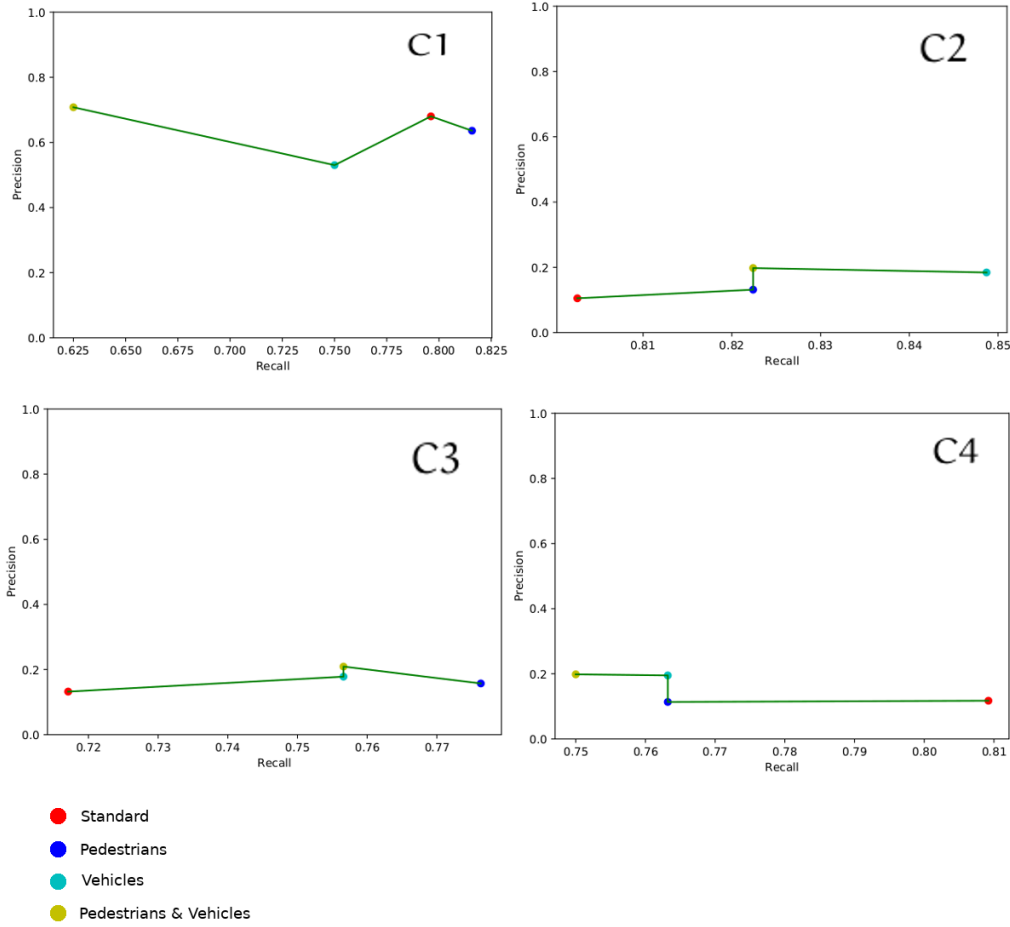
Figure 19: Precision-Recall Curve for Checkpoints C1...4, using values observed in each difficulty level

In phase 1, we had to manually rerun and observe some of the best runs for Checkpoint 2, to understand the values of the Safety-Monitor's predictions. If we look at the percentage of false positives causes showed in table 3, we can see that an excessive speed is the main cause of Monitor's failures for C2.[7] Another interesting aspect that can be observed collecting data in this way, is that it seems like the more the network learns to handle new situations, the more the Safety-Monitor tends to produce more false positives and, above all, the situations that will eventually result in a crash, are of a new kind that the Monitor is not able to

---

7  FAST SPEED: the car's speed was over 40km/h when the crash happened.
   SLOW SPEED: the car's speed was lower then 5 km/h when the crash happened

detect (SM ERROR).

The measures listed in table 2 are just demonstrative of how this method works and how, with few measures, a lot of informations and evidences about the System's behaviour can be extracted. Depending on the actual hardware available and the simulation environment, more measures can be extracted and combined to enrich these informations.

|              | C1  | C2  | C3  | C4  |
|:------------:|:---:|:---:|:---:|:---:|
| SM ERROR     | 80% | 41% | 39% | 57% |
| SUDDEN STEER | 0%  | 9%  | 24% | 12% |
| FAST SPEED   | 19% | 48% | 36% | 30% |
| SLOW SPEED   | 1%  | 2%  | 1%  | 1%  |

Table 3: Percentage of *false negatives* causes for C1 . . . C4.

As we hypothesized at the beginning of this work, the Monitor effectiveness seems to be influenced by *how* the Controller behaves. Most importantly, data suggests that the prediction accuracy of the Monitor is not directly dependent on how long the Controller is trained. Retraining the Network again with different strategies and apply the same monitoring method for the Controllers thus obtained may confirm or deny this fact.

### 4.3.3  *Phase 3 - Retraining and Retesting*

If *phase 2* already gave some hints about the fact that the effectiveness of the *same* Safety-Monitor is likely to change when applied to different Checkpoints, the re-training with different strategies and the performance comparison of the Controllers trained with such strategies and the one we took as the default strategy, will be central in confirming or denying this hypothesis.

In this phase, we took C4, and trained it again using these five strategies:

S0) The default strategy provided by *Coach framework* and that was used to train C1 . . . 4, where the reward function computed based on the overall neural network performances (collisions, time spent off-road or off-lane, travel speed. . . )

S1) The reward function is now more punitive when the car hits something, while braking is slightly more rewarded, provided that the Car won't stop moving

S2) The Safety Monitor is attached to the system and, if an alert is raised, the action of the network is replaced by the Monitor's action (the safety-brake)

S3) The network's action is replaced by the monitor's if it raises an alert. The network is given a positive reward for behaving like the Monitor

S4) The training step is stopped and the network is given a *negative* reward if an alert is raised

resulting in $\{C5_{S0}, C5_{S1}, C5_{S2}, C5_{S3}, C5_{S4}\}$.

*- On the Failure of Strategies S2, S3, S4:*

Unfortunately, the strategies that used the Safety-Monitor did not give the results we expected. Specifically, S3 and S4 turned to make the Controller to not move, while S2 makes the Controller stop after very few meters travelled if there is an object in front of the car, even if very far. All these Checkpoints doesn't get an unintentionally high reward for their behaviour, as *not moving* is given a negative reward, hence the case in which the neural network got stuck in a *local minimum* may still be an hypothesis, but our conjecture is that the training activity was too short.

This is almost surely caused by the very high false positives of the Monitor. A long training conducted in this way taught the car to not move in order to avoid the alerts raised by the Safety-Monitor and, even if these strategies takes into account the eventuality in which the Car does not move, punishing it with a negative reward, this was not enough to prevent the car from standing still. We believe that the *reward functions* for the *DDPG algorithm* used for the training activity are well specified, in the sense that all the positive and negative behaviours were taken into account when designing it and istantaneous rewards are designed to be well-balanced, as there are no *unintentional high reward*. However, the complexity and multitude of situations that a network must be able to handle in order to drive a car, makes it very difficult to tune all the reward parameters in order to prevent outcomes like the one we had. Most of all, we can not be sure whether this behaviour is a result of *how*

the reward function was designed or if a *longer* training was needed to observe interesting behaviours.

The biggest issue in the training activity is the extreme time consumption, and it's not thinkable to train $n$ agents with $n$ strategies and wait for them to be trained to analyze which strategy gave the best result.

As shown in several works, the tuning of the network's parameters largely depends on *rules-of-thumb* and *heuristic techniques*. However, these approaches has proven to be time-consuming (trial-and-error) and *error prone* (heuristics can not always model efficiently the complexity of these systems). Moreover, at the current state of knowledge, trial-and-error seems like the only way to assess the goodness of the reward function. This problem, which is specific to these kind of algorithms, is still unsolved since *reinforcement learning* is still being studied, and there are no official regulations for self-driving cars. [35]

The design of the *reward function* for these relatively new algorithms is a hot topic in the academic community, and researchers are working on methods that aims at understanding *how* to specifiy a *good model* for the learning agent. [31] [32] [33]

At the same time, as said before, AVs training require lots of data that are often unavailable or insufficient than those of conventional training techniques. There are several proposals on frameworks to be used to guide the design of decision-making policies for reinforcement learning, however most of them seems to *speedup* the learning process rather than helping on how to introduce new learning parameters for these networks. [34]

An interesting approach would be to develop stochastic models to guide us in the tuning of the reward function parameters, in order to perform a *calibration activity* prior to the retraining phase of the Controller. In this sense, the method developed in this work would be have a strong impact on the development of these models, as the performance measures for the Safety-Monitor were approximated effectively (e.g. the high false positives rate of the Safety-Monitor was a concern for us when training the Controller with the Monitor).

In general, it's not clear *a priori* what value the reward function should assign to each state. We also must consider that we are working in a *continuous space state*, which is hard to model and makes hard to consider all the possible events and transitions that may happen, making this task even harder.

Unfortunately, the complexity and the novelty of these systems and techniques, still requires more research to solve these problems, and the development of a *stochastical model* to guide us in the parameter tuning will be of uttermost interest in future works.

*- Performance Comparison and Analyis of S0 and S1:*

The failure of the strategies that relied on the Monitor, doesn't allow us to analyze how the effectiveness of the Monitor is influenced by those strategies. However, the monitoring activity can still be performed on $C5_{S0}$ and $C5_{S1}$ to observe:

a) How the effectiveness of the same Safety-Monitor is influenced by the Controller's effectiveness

b) If harshening the punishments (negative rewards) during the training has an impact on both the Controller's and Monitor's performances

The monitoring activity is basically a repetition of *phase 1* and *phase 2*. The same measures are computed and aggregated in the same form for the purpose of comparison.

$C5_{S1}$ performed significantly better than $C5_{S0}$ both in terms of MTTF and MDTF and both of them outclassed $C1\ldots4$. This reassure us on the fact that the System is actually travelling and higher MTTF is not a result of the car not moving, as happened for $C5_{S2\ldots4}$.

This is an interesting result, from the moment that the training period was the same for both Controllers (someone could argue that $C5_{S1}$ was trained for more time, but this is not the case), a more punitive strategy applied after a certain training phase gave good results in terms of running times and travelled distances, as shown in figures 20-21 which plots the Reliability Functions of $C5_{S0}$ and $C5_{S1}$ related to operational time and travelled distance.
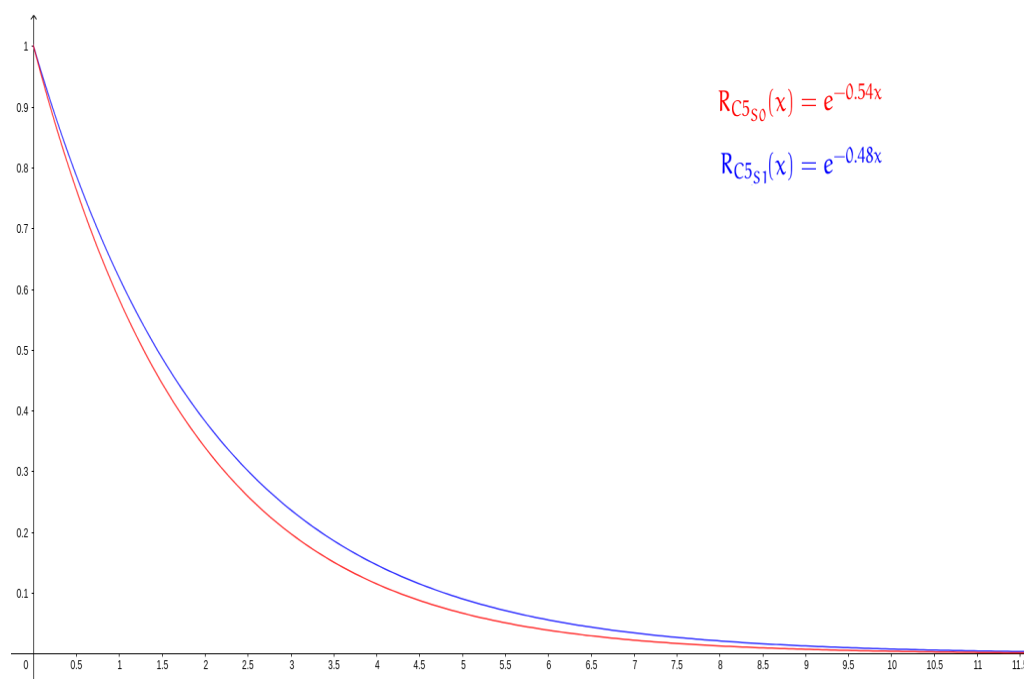
Figure 20: Reliability Function (minutes of operation) comparison between $C5_{S0}$ and $C5_{S1}$
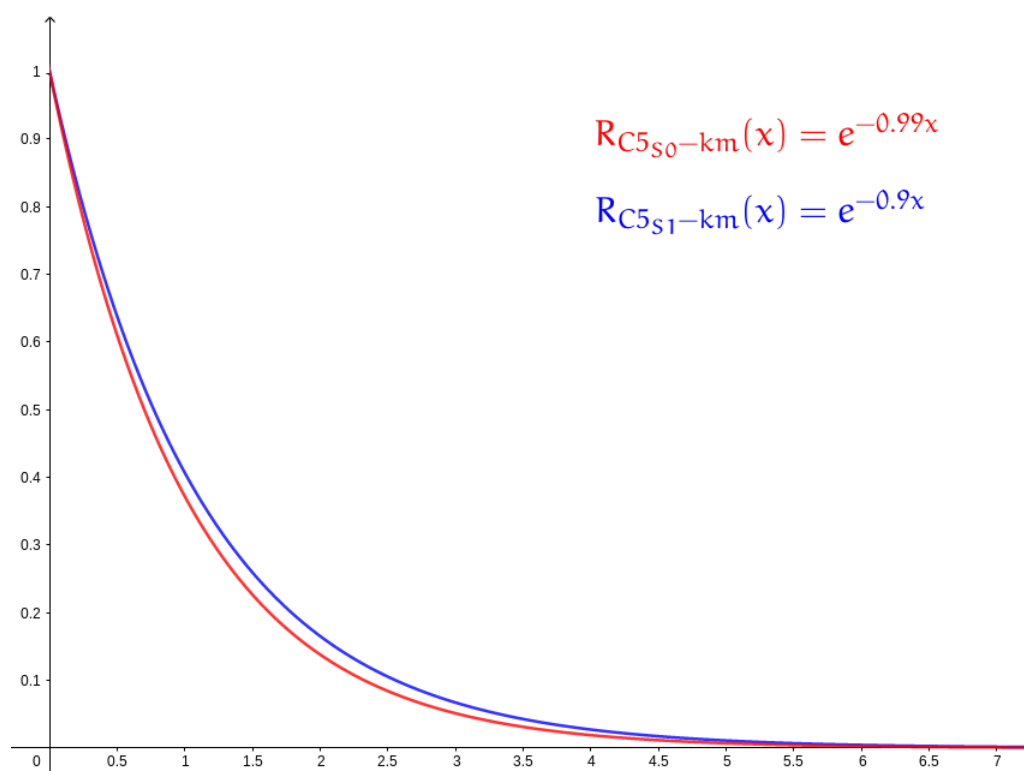
$$R_{C5_{S0}}(x) = e^{-0.54x}$$

$$R_{C5_{S1}}(x) = e^{-0.48x}$$



Figure 21: Reliability Function (kilometers travelled) comparison between $C5_{S0}$ and $C5_{S1}$

$$R_{C5_{S0}-km}(x) = e^{-0.99x}$$

$$R_{C5_{S1}-km}(x) = e^{-0.9x}$$

Comparing travelled distances charts (figures 22-27), we can see that both $C5_{S0}$ and $C5_{S1}$ have good performances compared to $C1\ldots4$, as both of them have an average distance travelled of circa 1 km, while C4 did approximately 500 m. $C5_{S1}$ reached the max run legth 3 times, and in general did longer runs than $C5_{S0}$ which reached max length only once. This is validated by the Reliability Functions computed both in term of time in operation and distance travelled, as $R_{C5_{S1}} > R_{C5_{S0}} > R_{C4}$.

Looking at $C5_{S1}$, if on one hand it had the best overall performances in terms of MTTF/MDTF, there are many scenarios in which it fails earlier than C4, failing even before the lower bound of 22 seconds previously observed. This can be seen very clearly looking at the right side of charts (approximately after scenario 120), where if on one hand $C5_{S0}$ is learning how to handle some of those scenario (while of course still failing in others, as it would have required too much time to wait for it to master all of them), $C5_{S1}$ shows a total different behaviour and in general it performs worse than both C4 and $C5_{S0}$ in those scenarios.
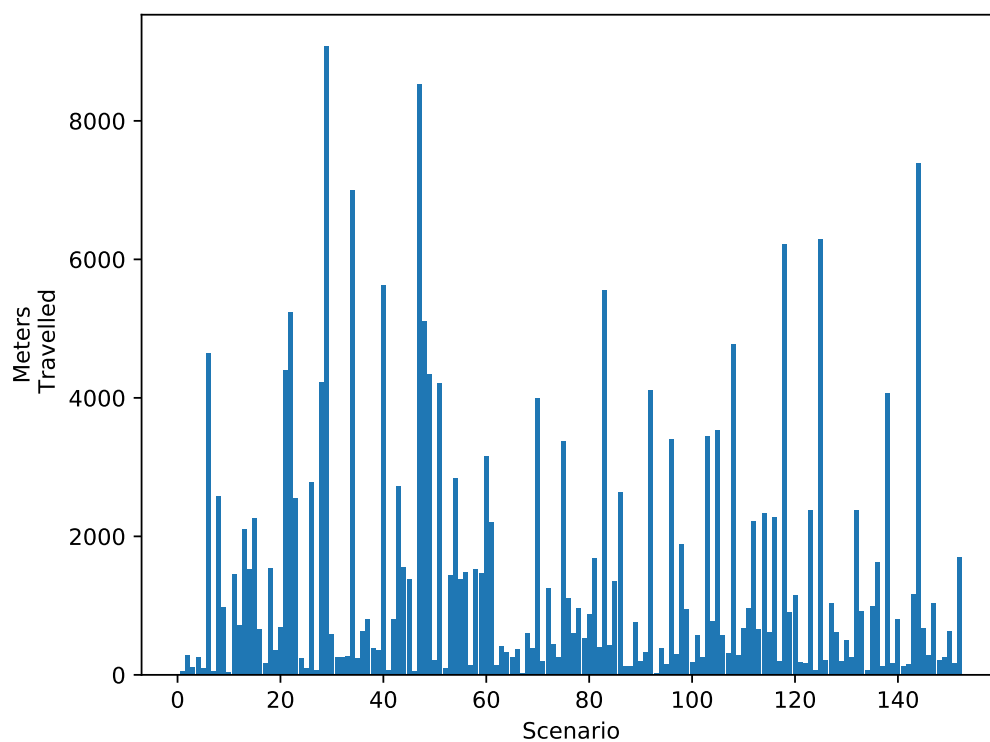
Figure 22: Meters travelled by $C5_{S0}$ in each scenario at standard difficulty
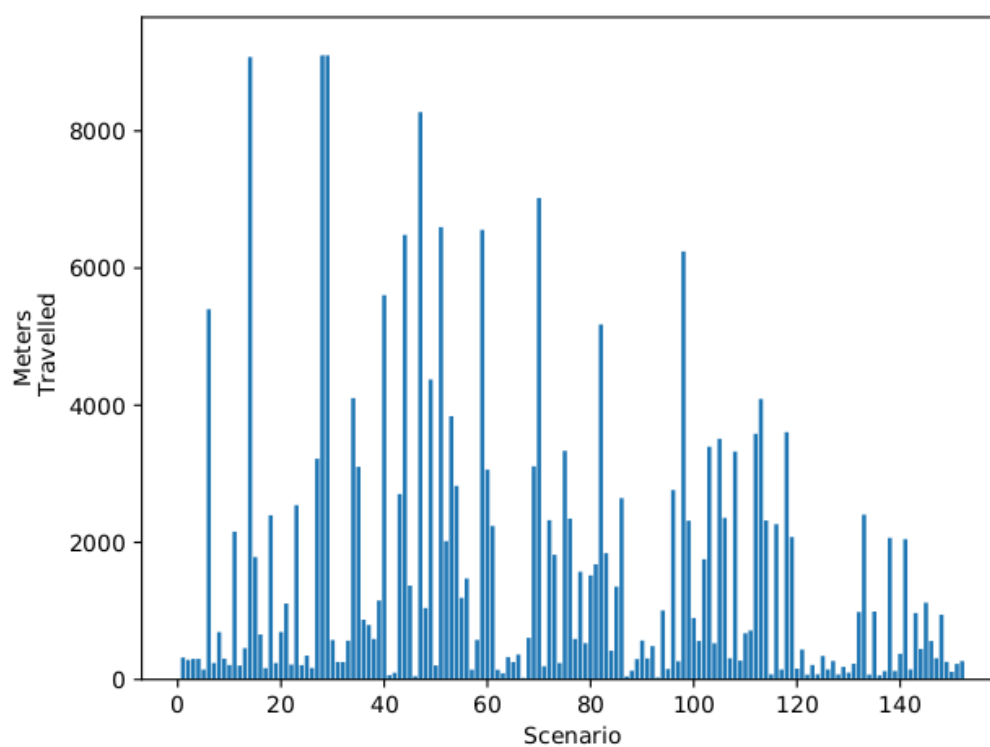


Figure 23: Meters travelled by $C5_{S1}$ in each scenario at standard difficulty

Figure 24: Meters travelled by C4 in each scenario at standard difficulty



Figure 25: Meters travelled by $C5_{S0}$ in each scenario at standard difficulty

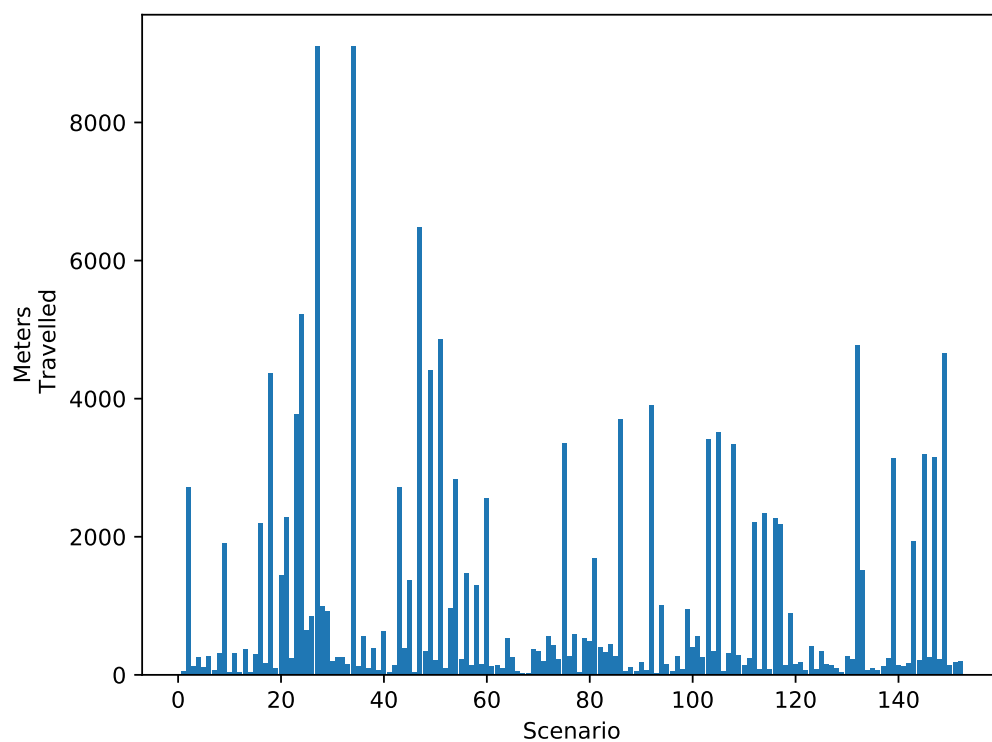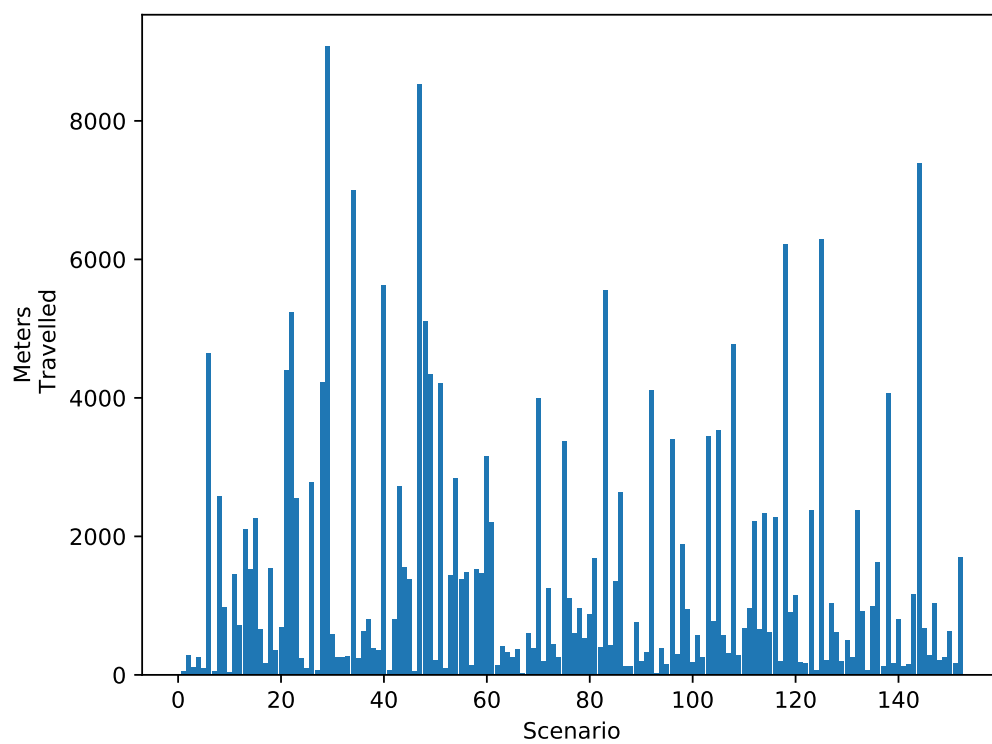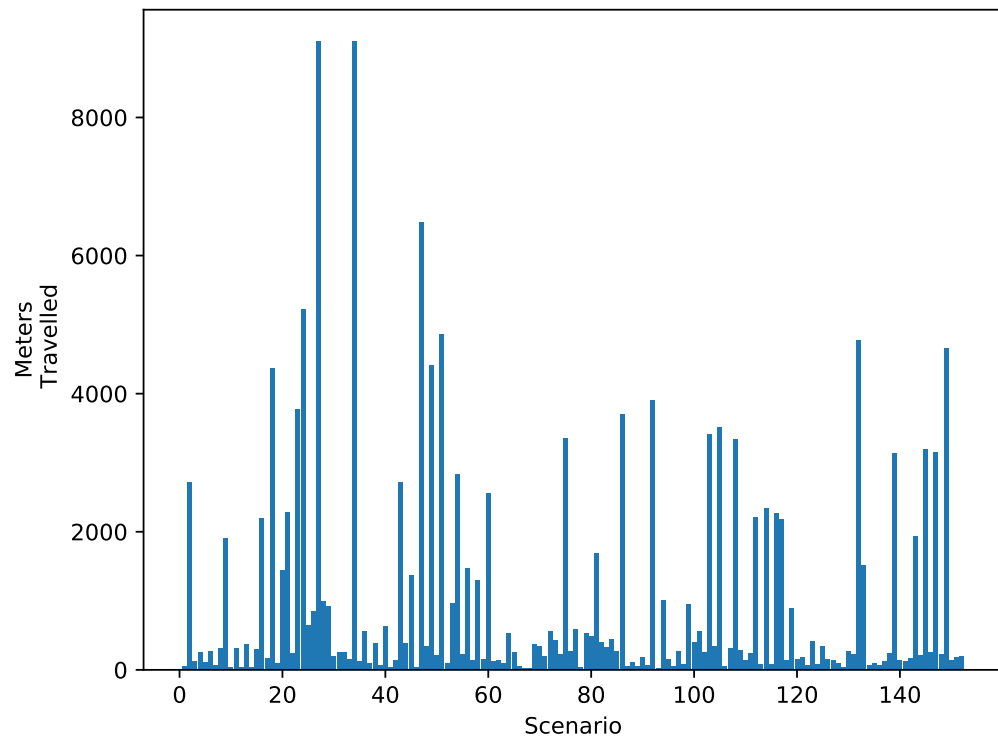Figure 26: Meters travelled by C4 in each scenario at standard difficulty
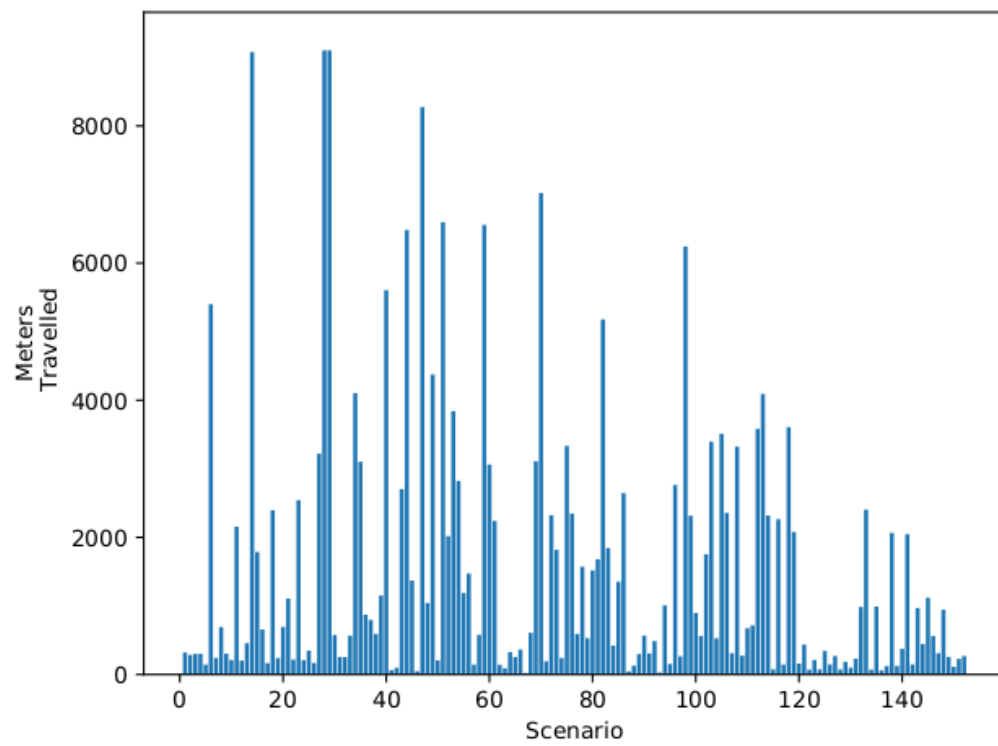


Figure 27: Meters travelled by $C5_{S1}$ in each scenario at standard difficulty

To have a better understanding on how these Controllers behaves, we look at how many times they reached a goal in each difficulty level (figure 28), and combine this information with *MTTFs* and *MDTFs*.

What confirms our hypothesis on C2's performances is that it reaches few goals compared to the others. This validate the hypothesis that the longer *MTTFs* and *MDTFs* were a result of *"lucky"* runs.



Figure 28: Goals reached charts for each Checkpoint, in each difficulty level

Moreover, $C5_{S1}$ chart is more balanced in terms of goals reached per difficulty. A fair objection would be that C4 reached its goals more times, but the average distance travelled is significantly lower than both $C5_{S0}$ and $C5_{S1}$, moreover, as an indicator of the fact that C4 **is not** better than $C5_{S0}$ & $C5_{S1}$, by an analysis of raw data it was observed that:
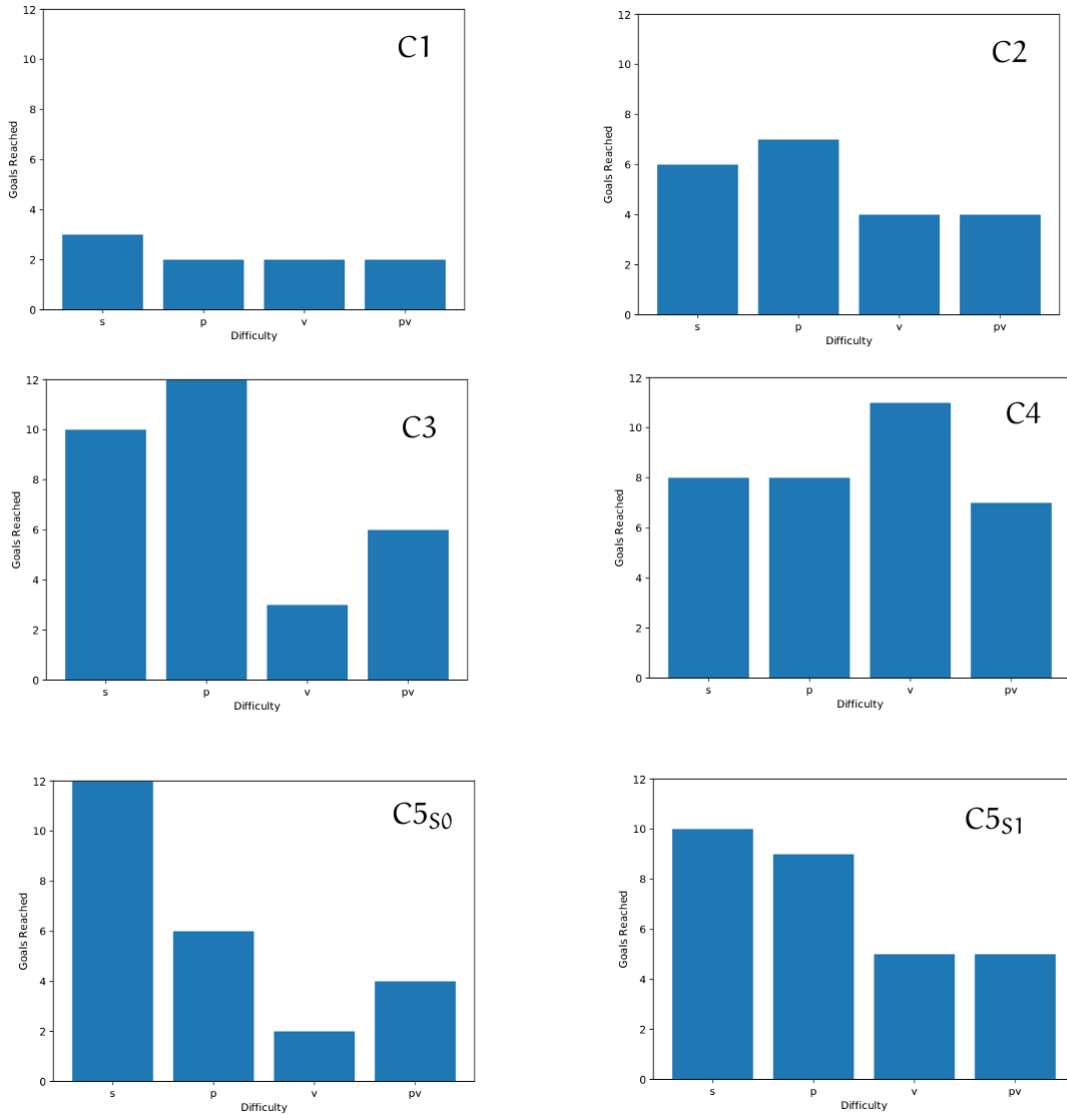
- $C5_{S1}$ is capable of reaching *more* goals in the *same* run, differently from the other Checkpoints

- C4 reaches some goals in scenarios $> 120$, where the unexpected performance drop is observed for $C5_{S1}$

In addition to this, $C5_{S0}$, which is the *"natural continuation"* of C4 as it is obtained from C4 using the *same* training strategy, performs very good at *standard difficulty* while performing much worse in the others, fact validated by comparing times to failure in each difficulty as $MTTF_{C5_{S0}h_0} >> MTTF_{C5_{S0}h_{1...3...3}}$. This seems to suggest that the training strategy adopted has a strong influence on how the System will perform in conditions different from the training ones. Collected data confirm this, as looking at raw data, we observe that $MTTF_{C5_{S1}h_i} > MTTF_{C5_{S0}h_i}$ as a proof that $C5_{S1}$ performs better in harder conditions. Moreover, the differences observe on the average travelled distances are lower than the ones observed for *all* the other Controllers, which again reassure us on this fact.

As done for $C1 \ldots 4$, hit rates of collisions with specific kinds of object wrt to total collisions are computed and shown in figure 28. Those for $C5_{S0}$ and $C5_{S1}$ looks similar, as hit rates take close values. The drop of collision rate against *obstacles* reassure us on the fact that both Checkpoints are now learning to stay on-road. To confirm this fact, a subset of runs was analyzed and it was observed that the main cause of collisions with generic obstacles is caused by the Car actually trying to avoid other cars and pedestrians. The same phenomenon was observed for Checkpoints C3, C4, where huge increasement of obstacles hit was linked to an increase of SUDDEN STEER errors for the Monitor. More evidences about this fact will be probably given by analyzing the runs with the Safety-Monitor attached to the System.
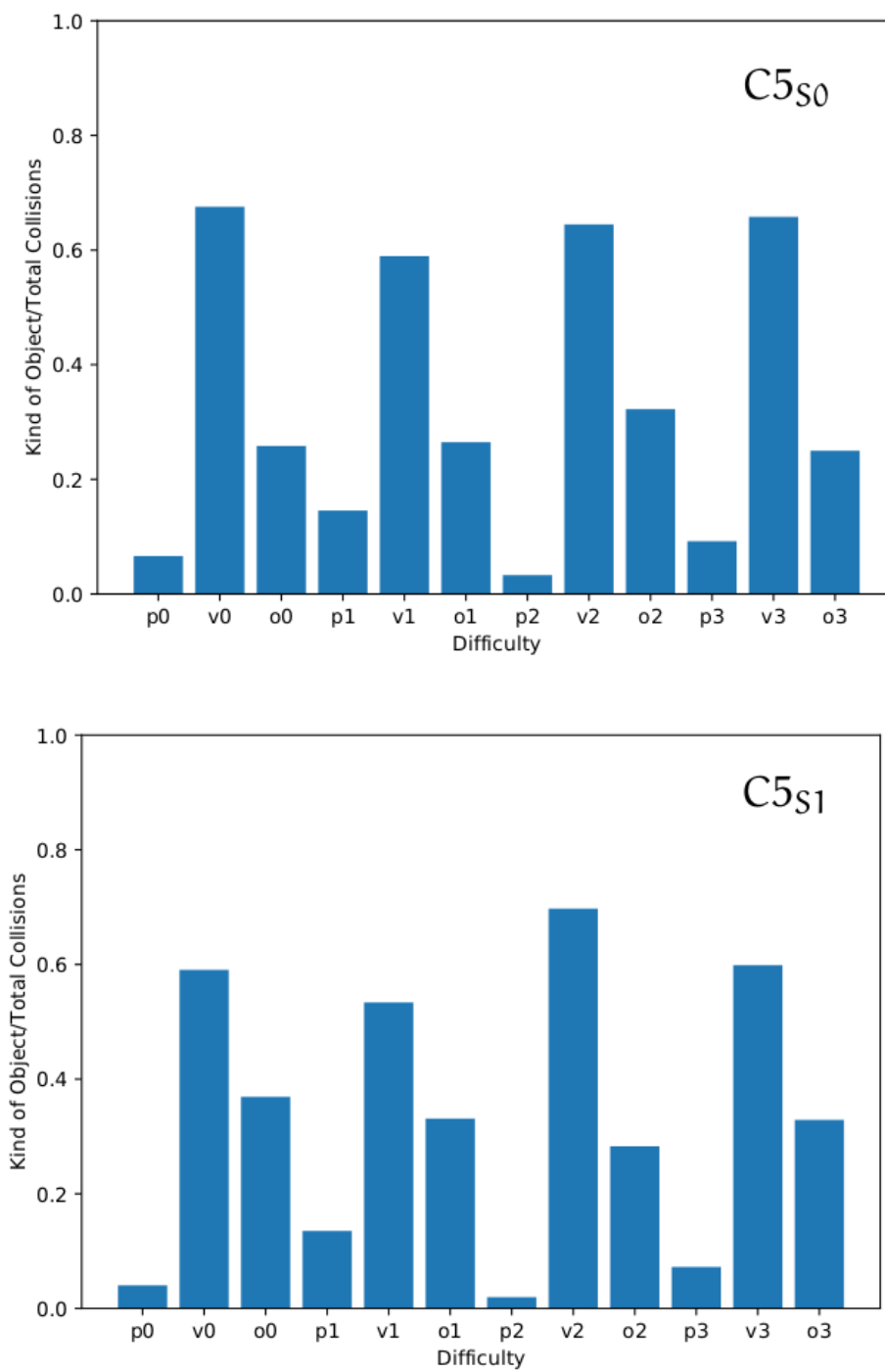
Figure 29: Hit rates for $C5_{S0}$ and $C5_{S1}$

*- On the Effectiveness of the Safety-Monitor:*

The analysis of the Safety-Monitor performances isn't different from what we did in *Phase 2*.

This second assessment of the Safety-Monitor seemed promising in conferming the fact that the effectiveness of the Safety-Monitor strongly depends both on the *training strategy* adopted and on the Controller's performances.

Looking at tables 4 and 5, we can see that the Monitor attached to $C_{1...4}$ and $C5_{S0}$ have a *True Positive Rate* of approximately 0.75, with the exception of C2, that acted as an outlier in our study. The fact that the Monitor with $C5_{S0}$ has the same TPR of the previous Checkpoints is a good result, and a promising starting point for further examinations. However, to claim this fact, more data and analysis are needed and will be a central point in our next studies.

|      | $C5_{S0}$ | $C5_{S1}$ |
|------|-----------|-----------|
| TPR  | 0.74      | 0.67      |
| FNR  | 0.26      | 0.33      |
| FPR  | 0.0058    | 0.0059    |
| TNR  | 0.9942    | 0.9941    |
| ACC  | 0.994     | 0.994     |
| PREC | 0.10      | 0.09      |
| MCC  | 0.28      | 0.25      |
| FPPM | 0.99      | 0.89      |

Table 4: Prediction Rates of the Safety-Monitor for Checkpoints $C5_{S0}$, $C5_{S1}$

As a proof that true and false positive and negative rates were not enough to understand the Monitor's performances, we can again look at the values of *Precision* and *MCC*. Analyzing these measures, the decrease in both of them immediately catches the eye. If Precision decrease relatively slowly, the drop in MCC is considerable, passing from $\approx 0.35$ observed in C2, C3, C4 to 0.28 and 0.25 observed in $C5_{S0}$ and $C5_{S1}$ respectively. The decrease in the MCC worries us, as this monotonic decrease could make the MCC to become negative (i.e. actually *detrimental* to the System, making wrong predictions only), if the Controller gets trained again.

|      | C1     | C2     | C3     | C4     | $C5_{S0}$ | $C5_{S1}$ |
|------|--------|--------|--------|--------|--------|--------|
| TPR  | 0.75   | 0.82   | 0.75   | 0.77   | 0.74   | 0.67   |
| FPR  | 0.0029 | 0.0029 | 0.0062 | 0.0069 | 0.0058 | 0.0059 |
| FPPM | 0.52   | 1.13   | 1.02   | 1.12   | 0.99   | 0.89   |
| ACC  | 0.995  | 0.992  | 0.993  | 0.992  | 0.994  | 0.994  |
| PREC | 0.63   | 0.14   | 0.17   | 0.15   | 0.10   | 0.09   |
| MCC  | 0.68   | 0.34   | 0.35   | 0.34   | 0.28   | 0.25   |

Table 5: Main measures comparison for every Checkpoint

The most interesting fact is that the Monitor performed worse with $C5_{S1}$ which is actually the Controller that had the best performances in terms of MTTF/MDTF. The fact that this last Checkpoint was obtained from C4 by *changing* the training strategy, seems to motivate our initial hypothesis both on the long-term effectiveness of the Safety-Monitor, and the influence the training strategy has on it. Another interesting fact is the drop in TPR.

We plot again the *Precision-Recall Curve* for the Safety-Monitor attached to $C5_{S0}$ and $C5_{S1}$, and we can see that the Monitor is definetely useless and almost disruptive to the System. From these plots, the Monitor worsening when operating with $C5_{S1}$ can be immediately seen, as it ranges from 0.65 to 0.685, while with $C5_{S0}$ it ranges from 0.7 to 0.8, consistently with C1...4. Moreover, these plots confirm that, even if its performances are still bad, that the Monitor has overall better performance with difficulties h2, h3, the two hardest difficulties defined, as it was observed with the previous Checkpoints.
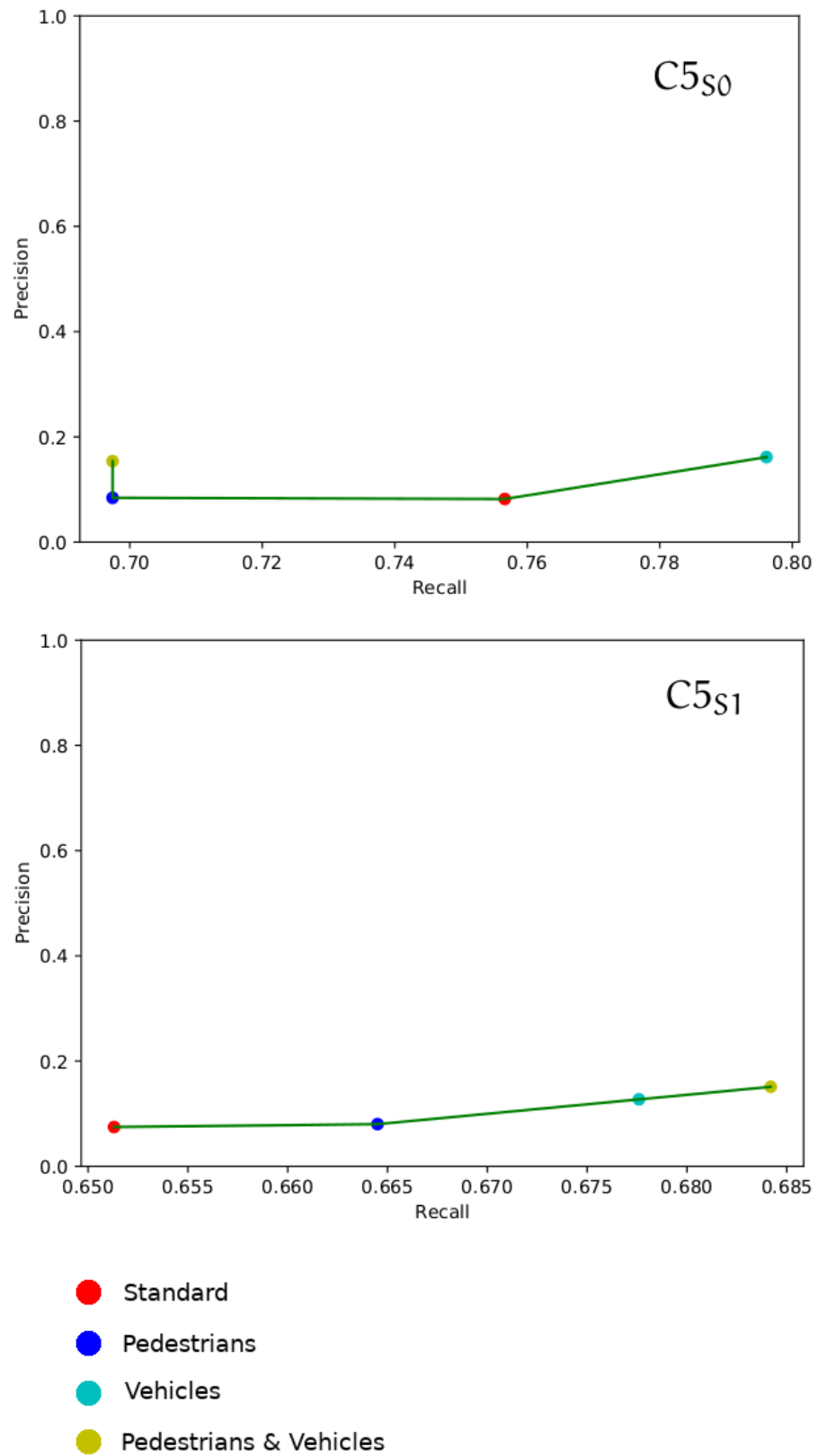
Figure 30: Precision-Recall Curve for Checkpoints C1...4, using values observed in each difficulty level

Although we could not plot the ROC curve, for the reasons explained in Chapter 3, the *Precision-Recall Curve* proved to be an effective visualization mean for this kind of systems, where there is a huge imbalancement between the positive and negative classes. The performance drop of $C5_{S1}$ would have not been that immediate with the "classic" ROC curve, as the False Positive Rates all have values lower than $10^{-2}$, resulting in curves flattened to the left side of the plot. Moreover, due to this issue, it would have been harder to understand how the Monitor performs in each difficulty. Ddespite the missing ROC plot, with our methodology we were still able to correctly define and measure all the rates needed to evaluate a classification model. What surprises us, looking once again at table 5, are the TPR and FPR values observed for $C5_{S0}$ and $C5_{S1}$: even if the Monitor had close FPR values with both of them, the drop in TPR observed in $C5_{S1}$ seems like a speedup towards the "worse-than-random area" (enforced by the MCC decreasing monotonically), motivating the fact that the Controller is now creating more situations that the Monitor is not able to detect (also motivated by the fact that it C5S1 shows the lowest FPPM ever, excluding C1). It is true that a deviation from the mean value is observed with C2 as well, but the other measures were somehow consistent and got close values. $C5_{S0}$ has the same TPR of all $C1\ldots4$ and it was trained with the same strategy of $C1\ldots4$. This doesn't happen with $C5_{S1}$, but at the same time, we can not claim this as a fact, as more experiments are needed. Again, this result may be a starting point for future works.

Analyzing crash causes listed in table 6, we have another interesting result. The percentage of *"unknown"* causes, labeled as generic errors of the Safety-Monitor is higher for $C5_{S1}$ than $C5_{S0}$, which is the main cause of crashes for the first, while the latter tends to crash at high speed. This may ring a bell on the fact that $C5_{S1}$, which now we can confirm performed better than *all* the other Checkpoints, gets into situations which are more complex to handle. This is a very important point to analyze. A fair objection would be that our Safety-Monitor is not good enough to detect the new kinds of hazard caused by the novel Controller's behaviour. It is true that a more sophisticated Monitor would be able to detect and avoid more complex hazardous events, and it may even show a SM-ERROR rate of 0% when tested with a controller $C_i$, but what we are pointing out in this work is that *we can not assure* that if $C_i$ is trained to drive in *new environments*, obtaining $C_{i+j}$, the SM-ERROR rate will still be 0%.

|  | $C5_{S0}$ | $C5_{S1}$ |
|---|---|---|
| SM ERROR | 28% | 50% |
| SUDDEN STEER | 14% | 13% |
| FAST SPEED | 58% | 37% |
| SLOW SPEED | 0% | 0% |

Table 6: Percentage of *false negatives* causes for $C5_{S0}$, $C5_{S1}$.

The high percentage of FAST SPEED errors observed in $C5_{S0}$ also provides a deeper explanation on the higher collision rate against pedestrians and vehicles: in many scenarios the System is goind so fast that it is not able to detect the possible pedestrians/vehicles on its road. At the same time, the similarity in the observed hit rates assure us that both Checkpoints are learning to don't go off-road and raises more questions on *how* the definition of the reward function causes behaviours that were not specifically planned or considered, when designing the training strategy.

# 5

## CONCLUSIONS

In this work we explored the topic of how to perform a monitoring activity on self-driving cars, controlled by a Control System (Controller) and a System Supervisor (Safety-Monitor).

This study is meant to be a first step on the analysis of *how a monitoring activity should be performed for such systems*, as they require particular care due to their complexity, and the critical issues of the environment in which they operate. Autonomous Vehicles themselves are not a new kind of systems, since they are commonly employed in military actions. However, the fact that self-driving cars will work at close contact with people and that the urban environment requires extra care in handling the moltitude of events (hazardous or not) that may happen. Moreover, we questioned if there are relationships between the effectiveness of the Safety-Monitor component and the Controller component, when the latter gets trained for long periods, using different strategies.

In the first sections, we reviewed the main concepts of these activities, such as

- The definition of *dependability* and *safety*, which are central to Safety-Critical Systems, are seen in the light of self-driving cars

- The problems that prevent us to deploy self-driving cars in urban environments at this level of knowledge

- How a wrong/poor/optimistic *dependability assessment* of these systems may have tragic consequences when these systems are deployed

- The problems of studying the observed emergent behaviour resulting from the interaction of two main Constituent Systems

- If, and in what way, the Safety-Monitor effectiveness is influenced by the performances of the Controller, in terms of:

- Length of the training period

- Strategies adopted during the training

The development of this first methodology has taken into account the tools available for this kind of activities. Since most technologies in this field are private and proprietary, we had to look for open-source equivalents, demonstrating that it's not easy to perform these analyses in non-professional environments.

CARLA is with no doubt the best self-driving car simulator available, but it's not perfect. The LiDAR sensor bug[8] would have prevented us to do this study and if it wasn't for the project developed by Zhuang[9], it would have been *unfeasable* because our only alternative would have been to use *ground-truth* data, which would have been pointless for this work.

If on one hand, *Coach framework* is developed by *Intel AI Labs*, it suffers from the common issues often found in open-source project, the most evident being one of the two agents provided for CARLA, which stopped moving after few stages of training in *phase 1* (therefore unrelated to the use of Safety-Monitor). This bug makes us wonder about the presence of other issues in the design of the training algorithm which may had an influence on the behaviours observed for $C5_{S2...4}$.

Another issue that mines the usability of this framework is that the RAM is not correctly freed, eventually filling it and causing a crash of the running processes. This required us to monitor the training execution and restart it at every crash.

The monitoring activity was developed keeping in mind the main properties of:

- Repeatibility

  - By recording the Controller's execution in files, saving the action taken by the Controller in each frame and the seeds used in RNGs[1], the Destination Goals and the environmental parameter of each scenario

- Non-Intrusiveness

  - Code instrumentation, as said, may produce non-real time data or wrong measurements, due to the client-server architecture

---

1 Random Number Generators

used. CARLA solves this problem by allowing users to run simulations at a *fixed time step*. However the hardware must be powerful enough to achieve the time step required

- Representativeness

    - Representativeness of *test-cases* is one of the central problems when monitoring self-driving cars, because hazardous events may be so complex and so many that is impossible to consider them all. The definition of scenarios and difficulty level helps in this sense, providing many different sitations starting from the same scenario

    - Representativeness of the *measures* computed for the Safety-Monitor raised issues on the nature of the dataset we are working with. The choice of the measures to be used on the Confusion Matrix was explained and motivated, as well as the exclusion of the others

- Feasibility

    - The research on the best tools to be used for this activity was not easy, due to the multitude of solutions proposed that in reality have lots of practical limitations. However, the tools used in this work allowed us to perform this activity. At the same time we can not ignore the issues found in these projects

If the *feasibility* property is undermined by the presence of issues in the tools used, the methodology developed still proved to be an effective way of assessing the reliability of Autonomous Vehicles allowing us to approximate all the common measures of interest for these systems, without sticking to common metrics used in the evaluation of neural networks, e.g. the *Loss Function*, which indeed are powerful and mandatory measures to compute to check that the neural network is actually learning, but doesn't give any evidence at all on the goodness of the System in its entirety.

This methodology proved to be an effective mean to study the behaviour of the whole System, focusing on well-known measures used to assess a system's dependability and the Safety-Monitor's performances. The definition of *Scenarios* and *difficulties* is very useful in the understanding of what environmental parameters cause a degradation on System's performances and in the generation and analysis of *"uncommon"* situations that not always can be taken into account when designing *ad-hoc*

*safety-cases*

The main conjecture that made us start this work, was if "static" (in the sense that they don't learn from new data) error-checkers' performances were depending on the performances of neural networks and if ad hoc training strategies have an influence as well. From the data collected it looks like the Monitor's effectiveness is likely to change with expert agents, motivating us to continue to study in deep this conjecture.

The failure of those strategies that used the Safety-Monitor to guide the training (S2, S3, S4) did not give us evidences on this fact, and this problem was already discussed in the previous chapter. This failure may be caused by two factors:

1) Misspecified reward function

2) The training period was not long enough

The fact that the agents trained were not getting unintentionally high reward for standing still, makes us think that the neural network needed more time to learn what the Safety-Monitor is trying to teach. A higher *simulation time step* could have helped us, making *simulation time* running faster than *real time*. However, this required ultra-high performance hardware, therefore we could not train the models for more time, nor retrain the networks adjusting the reward parameters.

At the same time, the Safety-Monitor proved to be *less* effective when operating in combination with the Checkpoint which had the *best* performances and, most of all, it was the one obtained from Checkpoint C4 by changing the training strategy, while when operating with the Checkpoints trained with strategy S0 (C1, C2, C3, C4, $C5_{S0}$) it had the same TPR (0.75), with the exception of C2. If on one hand, C2 was definetely a special case, for the reasons listed in the previous chapter, at the same time we can not exclude the same hypothesis for $C5_{S1}$. In future works, we want to continue the training of $C5_{S1}$ to confirm this fact.

This work demonstrated the power of the use of a simulated environment for the assessment self-driving cars, taking into account and solving many of the problems found when approximating the measures of interest for these systems, and common problems found in a monitoring activity.

In future works, our interest will be:

- Developing stochastical models based on data collected with this monitoring methodology, to guide us in the tunning of the reward function parameters

- Continuing the training of C5$_{S0...4}$ to:

    a) Collect more evidences about the effectiveness of the Safety-Monitor

    b) Have a better understanding of the failure causes of strategies $S2...4$

- Applying the same monitoring methodology using more powerful tools (e.g. proprietary software, pretrained neural networks, better Safety-Monitor)

# BIBLIOGRAPHY

[1] Peter Popov, Lorenzo Strigini - *Assessing Asymmetric fault-tolerant Software* (Cited on page 28.)

[2] Nidhi Kalra, Susan M. Paddock - *Driving to Safety: How Many Miles of Driving Would It Take to Demonstrate Autonomous Vehicle Reliability?* (Cited on page 22.)

[3] Arizona 2018 Uber Incident - *https://en.wikipedia.org/wiki/Death_of_Elaine_Herzberg* (Cited on page 21.)

[4] Elon Musk declarations - *https://techcrunch.com/2020/02/18/elon-musk-says-all-advanced-ai-development-should-be-regulated-including-at-tesla* (Cited on page 23.)

[5] Uber incident Preliminary Report - *https://www.ntsb.gov/investigations/AccidentReports/Reports/HWY18MH010-prelim.pdf* (Cited on page 21.)

[6] Philip Koopman - *http://safeautonomy.blogspot.com/* (Cited on page 21.)

[7] CARLA - *http://carla.org/* (Cited on page 47.)

[8] CARLA LiDAR Bug - *https://github.com/carla-simulator/carla/issues/1212* (Cited on pages 50 and 88.)

[9] Improved LiDAR CARLA - *https://github.com/ZhuangYanDLUT/carla* (Cited on pages 51 and 88.)

[10] Nervana Systems - Coach - *https://github.com/NervanaSystems/coach* (Cited on page 51.)

[11] Xingyu Zhao, Valentin Robu, David Flynn, Kizito Salako, Lorenzo Strigini - *Assessing the Safety and Reliability of Autonomous Vehicles from Road Testing* (Cited on pages 22 and 36.)

[12] Bev Littlewood, Lorenzo Strigini - *Validation of Ultra-High Dependability for Software-based Systems* (Cited on page 22.)

[13] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, Piotr DollÃ¡r - *Focal Loss for Dense Object Detection* (Cited on page 22.)

[14] DeepMind - https://deepmind.com/research/case-studies/alphago-the-story-so-far (Cited on page 22.)

[15] Google AI defeats human Go champion - *https://www.bbc.co.uk/news/technology-40042581* (Cited on page 22.)

[16] Anh Nguyen, Jason Yosinski, Jeff Clune - *Deep Neural Networks are Easily Fooled: High Confidence Predictions for Unrecognizable Images* (Cited on page 23.)

[17] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, Rob Fergus - *Intriguing properties of neural networks* (Cited on page 23.)

[18] Coach Issue - *https://github.com/NervanaSystems/coach/issues/428* (Cited on page 52.)

[19] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, Daan Wierstra - *Continuous control with deep reinforcement learning* (Cited on page 51.)

[20] Point Cloud Format - *https://en.wikipedia.org/wiki/Point_cloud* (Cited on page 50.)

[21] Point Cloud Library Info - *https://en.wikipedia.org/wiki/Point_Cloud_Library* (Cited on page 53.)

[22] Point Cloud Library - *http://pointclouds.org/* (Cited on page 53.)

[23] Responsibility-Sensitive Safety - *https://www.mobileye.com/responsibility-sensitive-safety/* (Cited on page 54.)

[24] J.C. Laprie - *Dependability - its attributes impairments and means, Predictability Dependable Computing Systems* (Cited on page 9.)

[25] J.C. Knight - *Safety Critical Systems: Challenges and Directions, Proceedings of the 24th International Conference on Software Engineering* (Cited on page 9.)

[26] Andrea Bondavalli - *L'analisi quantitativa dei Sistemi Critici, Fondamenti e Tecniche per la Valutazione - Analitica e Sperimentale - di Infrastrutture Critiche e Sistemi Affidabili* (Cited on page 13.)

[27]  G.J. Nutt - *Tutorial: Computer system monitors* (Cited on page 14.)

[28]  B. Plattner - *Real-time execution monitoring* (Cited on page 14.)

[29]  B.Plattner, J. Nievergelt - *Special feature: Monitoring program execution: A survey* (Cited on page 14.)

[30]  Engin Bozkurt - *https://github.com/enginBozkurt/LidarObstacleDetection* (Cited on page 54.)

[31]  Erik Talvitie - *Learning the Reward Function for a Misspecified Model* (Cited on page 71.)

[32]  Ellis Ratner, Dylan Hadfield-Menell, Anca D. Dragan - *Simplifying Reward Design through Divide-and-Conquer* (Cited on page 71.)

[33]  Nicolas Heess, Greg Wayne, David Silver, Timothy Lillicrap, Yuval Tassa, Tom Erez - *Learning Continuous Control Policies by Stochastic Value Gradients* (Cited on page 71.)

[34]  Yingjun Ye, Xiaohui Zhang, Jian Sun - *Automated Vehicle's behavior decision making using deep reinforcement learning and high-fidelity simulation environment* (Cited on page 71.)

[35]  Sampo Kuutti, Saber Fallah, Richard Bowden, Phil Barber - *Deep Learning for Autonomous Vehicle Control - Algorithms, State-of-the-Art, and Future Prospects* (Cited on page 71.)

[36]  Confusion Matrix, Wikipedia - *https://en.wikipedia.org/wiki/Confusion_matrix* (Cited on page 24.)

[37]  ISO 5725-1:1994 - *Accuracy (trueness and precision) of measurement methods and results* (Cited on page 42.)

[38]  Davide Chicco, Giuseppe Jurman - *The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation* (Cited on page 42.)