UNIVERSITÀ
DEGLI STUDI
FIRENZE

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

TITOLO IN ITALIANO

TITLE IN ENGLISH

TERROSI FRANCESCO

BONDAVALLI ANDREA
STRIGINI LORENZO

Anno Accademico 2018-2019

ABSTRACT

Abstract

INDICE

ELENCO DELLE TABELLE

ELENCO DELLE FIGURE

1

INTRODUZIONE

Sistemi informatici ormai ovunque (Cosa sono, esempi)

1.1 CYBER-PHYSICAL SYSTEMS OF SYSTEMS

1.2 DEPENDABILITY AND SAFETY

- Dependability
- Safety
- fault tolerance - fail safe
- Considerazioni generiche sul perche‘ della tesi

2

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 in these kind of systems may have very serious consequences that could result in people being injuried, or killed. At the same time, it is a problem to certify the ultra-high dependability requirements of these systems. 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 computer 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:

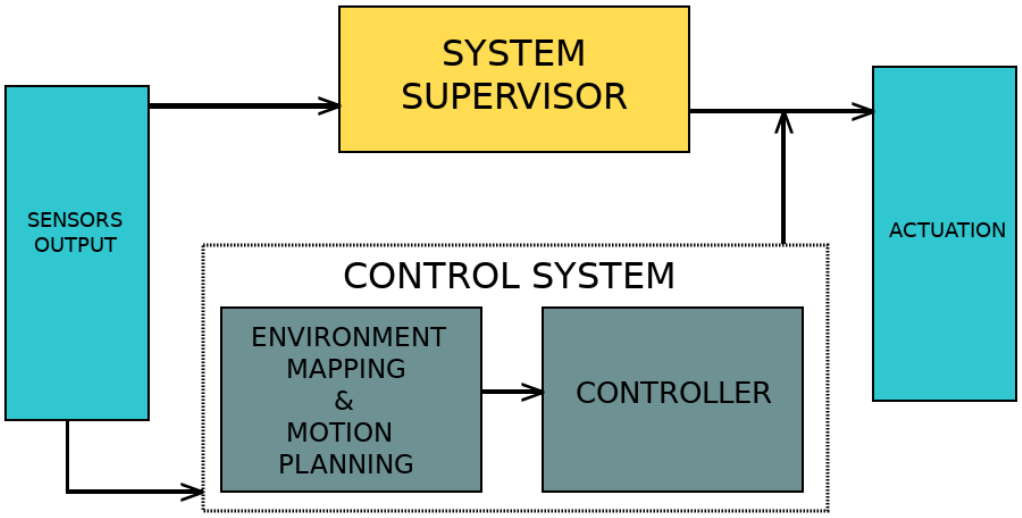

Figura 1: High-level abstraction of the system's software architecture

Data from sensors are inputs of the control system, here simplified as compose 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

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.

without crashing. The Controller (usually composed of a Velocity Controller[2] and a Steering Controller[3] uses these data to adjust the values on the actuator controlling the movements of the car: throttle, brake and steer.
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.[3] Further inspections showed that the car's radar and lidar data 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 way te system's safety is studied and tested must be looked from a new point of view and 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 form 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 well known 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

2 Controller in charge of adjusting the vehicle's speed
3 Controller that determines the steering angle
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)

more problems to deal with: how to safely and effectively test the system and the need for neural networks to achieve the task.

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 mils 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.[10]
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 of this approach can not work: an observed frequency failure of 0 would lead to optimistic (and possibly harmful) predictions. Luckily, this problem is surmountable, as shown in *this*[9] 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 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 names 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*[11] to *Gaming*[12] problems, performing even better than humans.[13] 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[15] and how high the confidence interval can be.[14] The lack of official regulations and certifications of these kinds 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 interaction between the Control System and the System Supervisor is 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 in this way:

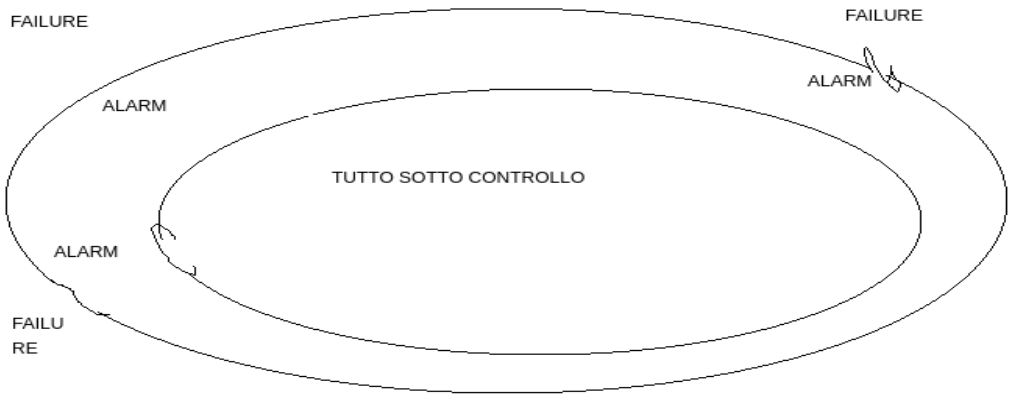

Figura 2: Sketch of the system's safe states

We consider *safe states* all the states in which the Control System produces an output that would not result in a crash.

Imagine that an autonomous car is riding when suddenly an obstacle appears. If the Primary correctly detects the obstacles it should apply a safety-measure to **avoid** a transition in an *alert state* (e.g. a safety-brake). If the controller doesn't see *or* detects the obstacle but keeps throttling, 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 not done by the controller 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). 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

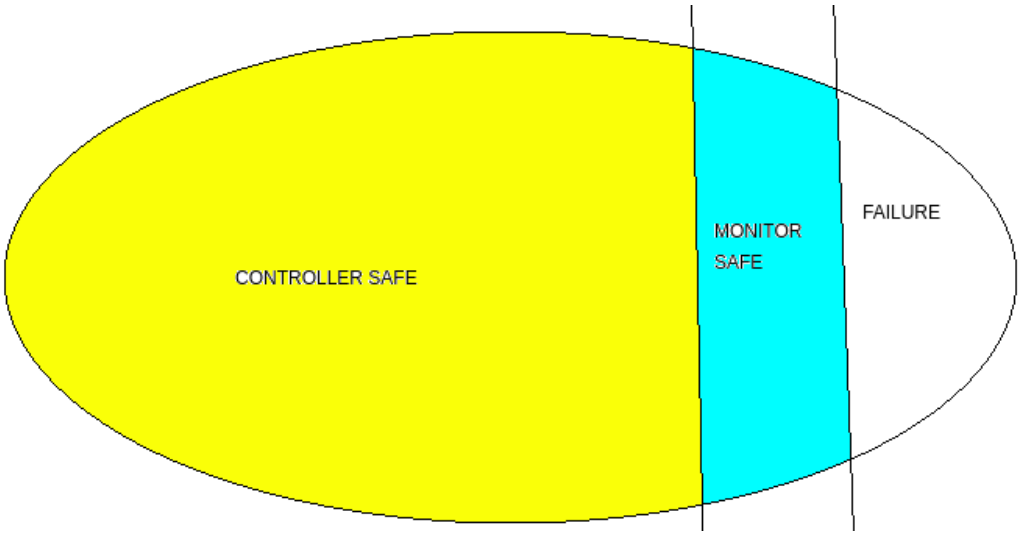

Figura 3: Visualization of safety areas

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 does the main computations, and a *Primary Checker* in charge of detecting (and correcting) error 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

specific input (or set of inputs), strictly depends on both the coverage of the Primary *and* the Primary Checker, as shown in the figure above.[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 areas covered by the Controller and the Supervisor will eventually overlap, reducing the overall contribution to the system's safety given by the latter.

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 the overall dependability of the system is increased, but some of the previously safe states are now alarm state. 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.

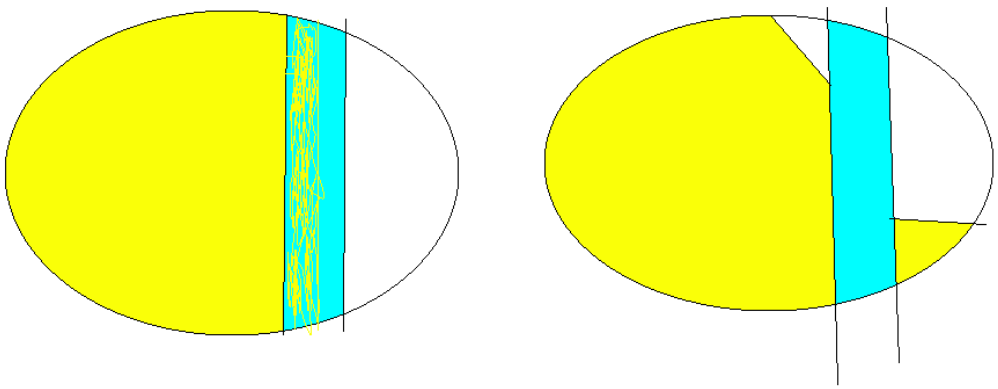

Figura 4: How the coverage of the system may vary when the network is trained

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, 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:

$$\texttt{checkpoint}_1 < \texttt{checkpoint}_2 < \ldots < \texttt{checkpoint}_{10}$$

where $\texttt{checkpoint}_1$ contains the network's weights at epoch 100, $\texttt{checkpoint}_2$ 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 would 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 avoid 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 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.

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, c 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 condition, to observe how the behaviour of the Controller changes in the first stages of the training.

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 change over time. Moreover, if many checkpoints of the same network are tested under the same scenarios, it is possible to extract interesting measures, 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 condition but 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 and the Safety Monitor are tested
- Phase 2: The network is retrained from the last checkpoint recorded, using different strategies to improve its performances. The new networks obtained and the (same) Safety Monitor are then retested in all the n_h scenarios
- Phase 3: Data Analysis and Comparison

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 c 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.[9]

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 *repeatibility* and *non-intrusiveness*. As pointed before, the *repeatibility* 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 Controller 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 scenarios 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

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 λ 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 following 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.

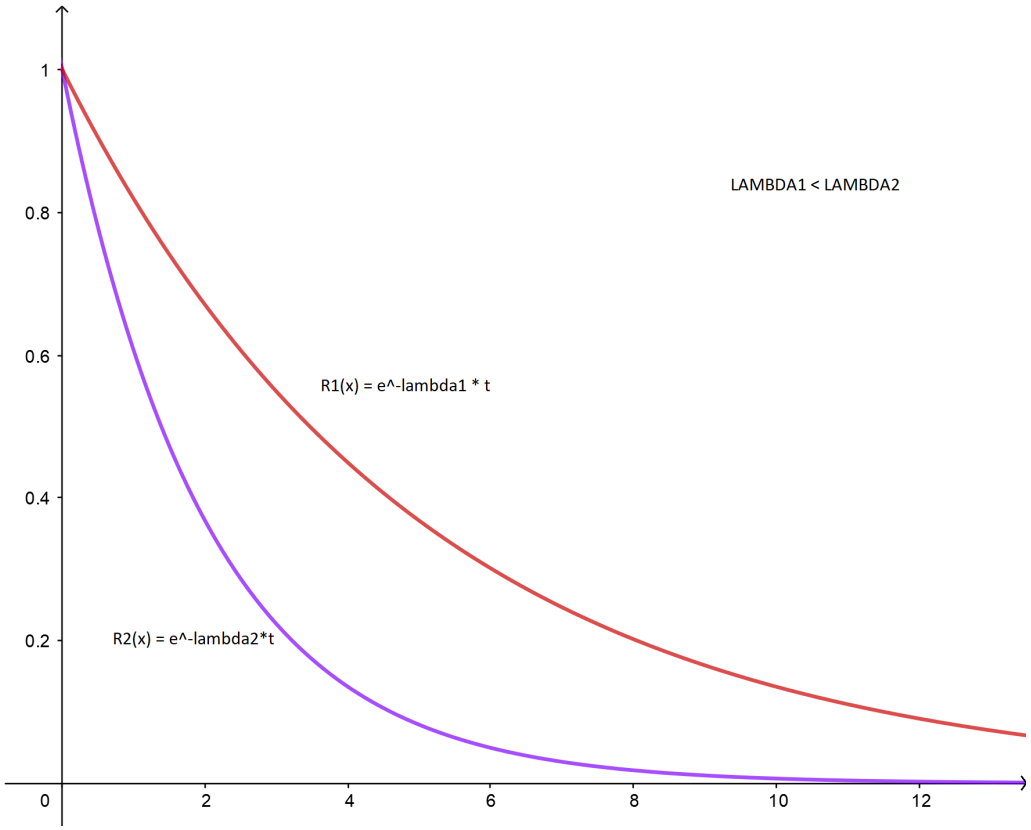

Figura 5: The Reliability measures the probability the at time t the system is still operating. We expect that more expert drivers (i.e. more trained networks) are capable of longer runs than little trained networks

If the Simulator used for testing permits 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...)
- 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. 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 create 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 an 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 possible to observe its true coverage.

Recall that what we call a *"failure"* of the Controller is 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 a Safety Monitor, the System would inevitably go to a failure state. For this component, 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:

- True Positive
  - The system went in an alert state due to a Controller's failure, correctly detected and prevented by the monitor
- True Negative
  - The system is in a safe state and the Monitor doesn't raise any alarm
- False Positive

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

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

These measures are commonly used to see how good a model is in classification problems. However, for real time critical system it's not easy, if not possible 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 issue makes very hard to measure the amount of true negatives, since some times, this kind of situations may be seen only when a human operator analyse the specific run.

Fortunately this can be omitted since a true negative does not change the behaviour of the system, therefore it was not considered during the Controller re-training phase.

- True Positive Rate
    - Rate di pericoli scampati
- False Negatives Rate
    - Rate di pericoli non scampati
- False Positives Rate
    - Rate di errori

Prima Fase:
- Controller test
- Monitor test

Seconda fase:
- addestramento

Strategie di training:
- fare piu' pressione sulle reward
- istruire con un monitor
- monitor + reward
- rimisurazione

4

METHOD IMPLEMENTATION AND RESULTS

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

A Neural Network was trained to drive in an urban environment. Checkpoints of the network's state during the training were recorded for comparison. These stages 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 Barcellona, 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 doubts essential to the functionality of the server: not only the environment must be simulated (inlcuding movements of pedestrians/vehicles, weather simulation. . .), but also all the data needed from the sensors attached to the system.

==========
IMMAGINE CARLA
==========

CARLA is currently at version 0.9.7 and huge improvements are done at every release, gaining more attention from the experts for its realisticity. 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 processng 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.

The 4 sensors available in CARLA 0.8 were used during the experiments. These 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 1 per side). These cameras, combined with the *"depth map"* camera allows not only the car to see, but also to perceive 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. At the same time a misclassification can be considered as an error of the control system: the safety monitor, combining data from all the available sensors, will not "correct" the misclassification but it must react fast and safely to avoid the potential consequences of it.

++++==== quindi pensiamo che non importi poi molto?

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...

In the simulations, due to the high hw 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 FPSs. Actors-related measurements include for example vehicles' speed, intensity of collisions (if any) and the 3D acceleration vector.

4.1.2 *Self-Driving Network*

The next step was to have an algorithm to train a neural network to drive in CARLA. The software needed to have the following charachteristics:

1 Training code must be available

2 No known/critical issues in the codebase

3 Provide an environment for interfacing the network with CARLA

After analyzing all the machine-learning related projects for CARLA, our choice was the reinforcement-learning framework *Coach*.[8]

4.1.3 *Safety-Monitor Implementation*

PCL Implementazione (struttura client-server, object """detection""")

4.2 METHOD IMPLEMENTATION

LEGGERE GROUND-TRUTH PERMETTE DI IGNORARE ERRORI DI MISCLASSIFICATION DEGLI OGGETTI
DISCUTERE LE STRATEGIE DI TRAINING

4.3 RESULTS

METTI QUI I NUMERINI CHE TI SONO VENUTI

5

CONCLUSIONS

BIBLIOGRAFIA

[1] Peter Popov, Lorenzo Strigini - *Assessing Asymmetric fault-tolerant Software* (Cited on page 17.)

[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 14.)

[3] Arizona 2018 Uber Incident - *https:en.wikipedia.orgwikiDeath_of_Elaine_Herzberg* (Cited on page 13.)

[4] Elon Musk declarations - *https:techcrunch.com20200218elonmusksays-alladvancedaidevelopmentshouldberegulatedincludingattesla* (Cited on page 15.)

[5] Uber incident Preliminary Report - *https://www.ntsb.gov/investigations/AccidentReports/Reports/HWY18MH010-prelim.pdf* (Cited on page 13.)

[6] Philip Koopman - *http:safeautonomy.blogspot.com* (Cited on page 13.)

[7] CARLA - *http:carla.org* (Cited on page 29.)

[8] Nervana Systems - Coach - *https:github.comNervanaSystemscoach* (Cited on page 32.)

[9] Xingyu Zhao, Valentin Robu, David Flynn, Kizito Salako, Lorenzo Strigini - *Assessing the Safety and Reliability of Autonomous Vehicles from Road Testing* (Cited on pages 14 and 23.)

[10] Bev Littlewood, Lorenzo Strigini - *Validation of Ultra-High Dependability for Software-based Systems* (Cited on page 14.)

[11] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, Piotr DollÃ¡r - *Focal Loss for Dense Object Detection* (Cited on page 14.)

[12] DeepMind - https:deepmind.comresearchcase-studiesalphagothestorysofar (Cited on page 14.)

[13] Google AI defeats human Go champion - *https:www.bbc.co.uknewstechnology40042581* (Cited on page 14.)

[14] Anh Nguyen, Jason Yosinski, Jeff Clune - *Deep Neural Networks are Easily Fooled: High Confidence Predictions for Unrecognizable Images* (Cited on page 15.)

[15] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, Rob Fergus - *Intriguing properties of neural networks* (Cited on page 15.)