PhD Thesis

Dependability for declarative mechanisms: neural networks in autonomous vehicles decision making.

by

Augustin VIOT

Corso di dottorato

XXXIV CICLO - INFORMATICA E INGEGNERIA DEI SISTEMI/ COMPUTER SCIENCE AND SYSTEMS ENGINEERING - Ingegneria dei sistemi

Strutture dell'Ateneo

Dipartimento di Informatica, bioingegneria, robotica e ingegneria dei sistemi

Settori scientifico-disciplinari della tesi

Settore ING-INF/05 - Sistemi di Elaborazione delle Informazioni

Tutor

TACCHELLA, ARMANDO

Tutor esterni

Walter SCHÖN, Benjamin LUSSIER, Stéphane GÉRONIMI

UNIVERSITÀ DEGLI STUDI DI GENOVA

ALLIANCE SORBONNE UNIVERSITÉ

utc

ANRT ASSOCIATION NATIONALE RECHERCHE TECHNOLOGIE

UNIVERSITÉ FRANCO ITALIENNE

UNIVERSITÀ ITALO FRANCESE

STELLANTIS

Dibris

heudiasyc

PhD Thesis

Dependability for declarative mechanisms: neural networks in autonomous vehicles decision making.

Augustin Viot

December 2022

Contents

Glossary **7**

1 Introduction **9**

2 Basic concepts in dependability and autonomous systems **13**
2.1 Dependability . . . 13
2.1.1 General . . . 13
2.1.2 Fault tolerance . . . 17
2.1.3 Formal verification . . . 19
2.2 Autonomous vehicles . . . 20
2.2.1 Autonomous vehicle definition . . . 20
2.2.2 Standards . . . 25
2.2.3 Autonomous driving functional architecture. . . . 26
2.3 Conclusion . . . 38

3 Neural networks: definitions and formal verification **39**
3.1 Neural Networks . . . 39
3.1.1 Definition . . . 40
3.1.2 Learning the desired function $f_{desired}$ . . . 43
3.1.3 Datasets . . . 48
3.1.4 Types of neural network architecture . . . 49
3.1.5 Neural networks for autonomous vehicles . . . 52
3.1.6 Safety for NN . . . 55
3.2 Formal verification for neural networks . . . 60
3.2.1 Definition of formal verification for NN . . . 61
3.2.2 Classifications of formal verification methods for neural networks . . . 62
3.2.3 Existing applications of formal verification for neural networks . . . 65
3.2.4 International Verification of Neural Networks Competition (VNN-COMP) . . . 73
3.3 Conclusion . . . 74

4 Dependability of neural networks and limits of formal verification 77
4.1 Characterization of neural networks . . . 77
4.1.1 Characterization from the machine learning community . 78
4.1.2 Characterization from a software development point of view 78
4.2 Fault, error and failure of neural networks . . . 79
4.2.1 Faults . . . 80
4.2.2 Errors . . . 81
4.2.3 Failures . . . 82
4.3 Formal verification limits and problems . . . 82
4.4 Consequences of an erroneous model . . . 85
4.4.1 Different models in different communities . . . 85
4.4.2 Formal verification of neural networks with erroneous model 87
4.5 Theoretical example of formal verification on an erroneous model 87
4.5.1 Experimental neural network and injected faults . . . 89
4.5.2 Formal verification . . . 92
4.6 Conclusion . . . 97

5 Online neural network model verification 99
5.1 Detection mechanism . . . 99
5.1.1 Description . . . 99
5.1.2 Limits of the detection mechanism . . . 101
5.2 Recovery mechanism . . . 103
5.2.1 Description . . . 103
5.2.2 Limits . . . 104
5.3 Conclusion . . . 105

6 Autonomous vehicle application: nominal conditions 107
6.1 Adaptive Cruise Control . . . 107
6.1.1 Choice of the ACC as an autonomous function . . . 107
6.1.2 Adaptive Cruise Control function . . . 108
6.2 Development of the neural network . . . 112
6.3 Neural network . . . 114
6.3.1 Architecture . . . 114
6.3.2 Training data . . . 114
6.3.3 Processes . . . 116
6.4 Framework of the verification via testing . . . 119
6.4.1 Simulator . . . 119
6.4.2 Scenarios for evaluation . . . 124
6.5 Formal verification of the nominal neural network . . . 136
6.5.1 Safety property . . . 141
6.5.2 Modification of the NN . . . 144
6.5.3 Marabou tool use . . . 144
6.5.4 Limit due to formal verification tool . . . 146
6.5.5 Formal verification results . . . 147
6.5.6 Need for precision . . . 148

6.6 Conclusion . . . 154

7 Autonomous vehicle application: erroneous neural network 155
7.1 Fault injection . . . 155
7.1.1 Requirements . . . 155
7.1.2 Proposed injected fault . . . 156
7.1.3 Other candidates for fault injection . . . 156
7.2 Training the erroneous NN . . . 157
7.3 Behavior of the erroneous neural network . . . 160
7.3.1 Measures on all scenarios . . . 160
7.3.2 Focus on erroneous NN's behavior in scenario B.1.2 . . . 165
7.4 Conclusion . . . 166

8 Autonomous vehicle application: Detection and recovery of an error in a neural network's model 169
8.1 Detection of an erroneous model based on a diversified mathematical model . . . 170
8.1.1 Architecture . . . 170
8.1.2 Detection . . . 171
8.1.3 Recovery . . . 173
8.1.4 Focus on scenario B.1.2 with detection based on equation 175
8.2 Detection of an erroneous model based on online evaluation of the safety property . . . 181
8.2.1 Architecture . . . 181
8.2.2 Detection with online evaluation of the safety property . . . 181
8.2.3 Recovery with online evaluation of the safety property . . . 182
8.3 Conclusion . . . 186

9 Neural network problems for formal verification 189
9.1 Imprecision due to model . . . 189
9.1.1 Importance of the architecture . . . 189
9.1.2 Number of inputs and outputs . . . 190
9.2 Imprecision due to variance . . . 191
9.2.1 Generating uniform test sets to have several counter examples . . . 191
9.2.2 Conservative function . . . 193
9.2.3 Restriction on the domain of the formal verification . . . 196
9.3 Importance of choosing a formal verification tool . . . 196
9.4 Erroneous models in other machine learning techniques . . . 197
9.5 Conclusion . . . 198

10 Conclusion 199

Annexes 203

Bibliography 209

Glossary

ACC: Adaptive Cruise Control

ADS: Automated Driving System

AV: Autonomous Vehicle

CNN: Convolutional Neural Network

DDT: Dynamic Driving Task

DNN: Deep Neural Network

GNSS: Global Navigation Satellite System

HMI: Human Machine Interface

LSTM: Long Short-Term Memory

MLP: Multi Layer Perceptron

MRC: Minimal Risk Condition

MRM: Minimal Risk Maneuver

NN: Neural Network

ODD: Operational Design Domain

OEDR: Object and Event Detection and Response

PFMEA: Process Failure Modes and Effects Analysis

R-CNN: Region based CNN

TH: Time Headway, the time gap between two vehicles

TSF-FCN: two-stream fusion fully convolutional network

V2X: Vehicle to X

FV: Formal verification

Chapter 1

Introduction

Despite being introduced in 1958 [126], neural networks appeared in numerous applications of different fields in the last decade. This change was possible thanks to the reduced costs of computing power required for deep neural networks, and increasing available data that provide examples for training sets. The 2012 ImageNet image classification competition [73] is often used as a example to describe how neural networks became at this time good candidates for applications: during this competition a neural network based solution [68] won for the first time. In the following editions, all winning solutions were based on neural networks. Since then, neural networks have shown great results in several non critical applications (image recognition, sound recognition, text analysis, etc...). There is a growing interest to use them in critical applications as their ability to generalize makes them good candidates for applications such as autonomous vehicles, but standards do not allow that yet.

Autonomous driving functions are currently researched by the industry with the final objective of producing in the near future fully autonomous vehicles, as defined by the fifth level of the SAE international (Society of Automotive Engineers) classification. Autonomous driving process is usually decomposed into four different parts: the *perception* where sensors get information from the environment, the *fusion* where the data from the different sensors is merged into one representation of the environment, the *decision* that uses the representation of the environment to decide what should be the vehicles behavior and the commands to send to the actuators and finally the *action* part that implements these commands. In this thesis, following the interest of the company Stellantis, we will focus on the decision part of this process, considering neural network based solutions.

Automotive being a safety critical application, it is required to implement and ensure the dependability of the systems, and this is why neural networks use is not allowed at the moment: their lack of safety forbid their use in such applications. Dependability methods for classical software systems are well known, but neural networks do not have yet similar dependable mechanisms to guarantee

their trust. This problem is due to several reasons, among them the difficulty to test applications with a quasi-infinite operational domain and whose functions are hard to define exhaustively in the specifications. Here we can find the motivation of this thesis: how can we ensure the dependability of neural networks in the context of decision for autonomous vehicles?

Research is now being conducted on the topic of dependability and safety of neural networks with several approaches being considered and our research is motivated by the great potential in safety critical applications mentioned above. In this thesis, we will focus on one category of method that seems to be a good candidate to ensure the dependability of neural networks by solving some of the problems of testing: the formal verification for neural networks. These methods aim to prove that a neural network respects a safety property on an entire range of its input and output domains. Formal verification is already used in other domains and is seen as a trusted method to give confidence in a system, but it remains for the moment a research topic for neural networks with currently no industrial applications.

The main contributions of this thesis are the following:

- A proposal of a characterization of neural network from a software development perspective, and a corresponding classification of their faults, errors and failures.
- The identification of a potential threat to the use of formal verification. This threat is the erroneous neural network model problem, that may lead to trust a formally validated safety property that does not hold in real life.
- The realization of an experiment that implements a formal verification for neural networks in an autonomous driving application that is to the best of our knowledge the closest to industrial use. For this application, we chose to work with an ACC (Adaptive Cruise Control) function, which is an autonomous driving function that performs the longitudinal control of a vehicle. The experiment is conducted with the use of a simulator and a neural network formal verification tool.
- A theoretical example of the erroneous neural network model problem and a practical example in our autonomous driving experiment.
- A proposal of detection and recovery mechanisms as a solution to the erroneous model problem mentioned above.
- An implementation of these detection and recovery mechanisms in our autonomous driving experiment.
- A discussion about difficulties and possible processes for the implementation of formal verification for neural networks that we developed during our experiments.

To present these different contributions, this thesis is organized as follows:

- Chapter 2 presents the necessary knowledge on the fields of dependability and autonomous vehicles to understand our work.
- Chapter 3 defines neural networks. It particularly presents the potential use of neural networks in autonomous driving applications, and how the safety of NN is currently considered in the industry and research communities, focusing on formal verification.
- Chapter 4 proposes a characterization of dependability's threats for neural networks. It also describes the limits of formal verification for neural networks, and presents the erroneous model problem with a theoretical example.
- Chapter 5 presents a theoretical description of the detection and recovery mechanisms that we propose to tackle the erroneous model problem.
- Chapter 6 a realistic formal verification application. It presents the function that we want to model, the development of the neural network, the simulator that was used to test it, the formal verification process and the results of the experiment.
- Chapter 7 presents an application example of the erroneous model problem in our autonomous driving experiment. Particularly, we present the fault injection that was performed to obtain the erroneous neural network, its training process and the results of the experiment.
- Chapter 8 presents the implementation of our detection and recovery mechanisms in our autonomous driving experiment and how they are able to tolerate the erroneous model problem.
- Chapter 9 presents some problems, challenges, good practices and methodology regarding the implementation of formal verification for neural network that we faced during our experiments.

Finally, we conclude this thesis with an overview of the thesis outcomes and a presentation of the potential perspectives.

Chapter 2

Basic concepts in dependability and autonomous systems

Neural networks show great potential for use in several safety critical applications. One necessary step before using them in such domains, is to evaluate and ensure their dependability. Actually, at the moment their use in safety critical applications is prohibited by standards because of their lack of dependability. In this chapter we introduce the context and the state of the art related to dependability and autonomous systems. Firstly, we will present key concepts of dependability, focusing particularly on fault tolerance and formal verification. Secondly, we will give a definition of autonomous vehicles, which is one of the safety critical applications where neural networks have great potential. We will based the definition on documents from the SAE (international Society of Automotive Engineers) and present several autonomous vehicles architectures from real applications.

2.1 Dependability

In this section, we will present the definitions related to dependability and other associated concepts, particularly its attributes, threats and means as defined in [6].

2.1.1 General

The dependability of a system has two congruent definitions:

- the ability to deliver a service that can justifiably be trusted

- the ability to avoid service failures that are more frequent and more severe than is acceptable.

This notion of dependability usually encompasses five attributes: availability, reliability, safety, integrity and maintainability. To reach a dependable system, we can implement methods or techniques that can be categorized into four means: fault prevention, fault tolerance, fault removal, fault forecasting. Finally, three threats to the dependability of systems are identified by the community: faults, errors, and failures. We detail these attributes, means and threats in the following paragraphs.

2.1.1.1 Attributes

The attributes are what characterize the dependability of a system. A system can have a correct reliability, but an insufficient availability. Each concept is defined as followed:

- availability: readiness for correct service. In the case of neural networks, the availability will for example depend on the time required to calculate the outputs from the inputs: as long as the outputs are not resolved, new inputs can not be treated.

- reliability: continuity of correct service. In the case of neural networks, the reliability depends on several factors, such as the presence of incorrect inputs, hardware faults in the system, incorrect learning of the NN, etc.

- safety: absence of catastrophic consequences on the user(s) and the environment. Systems that may cause catastrophic consequences are called critical systems. In the context of decision for autonomous road vehicles, a neural network would cause catastrophic consequences if the decision computed by the neural network would result in deaths, injuries or structural damage due to car crashes or collisions.

- integrity: absence of improper system alteration. In the context of neural networks application, this attribute is not often considered, although system alteration can classically be caused by hardware and external faults such as components deterioration or electromagnetic pulses.

- maintainability: ability for a process to undergo modifications and repairs. In the context of neural networks, maintainability would require modification of the NN (architecture, weights, biases, etc).

Note that to this five attributes of dependability, we also often add *confidentiality* defined as the absence of unauthorized disclosure of information. It is one of the three attributes of the security of a system with the *availability* and *integrity* attributes described previously. All the attributes can be presented in the tree of dependability and security shown on figure 2.1.

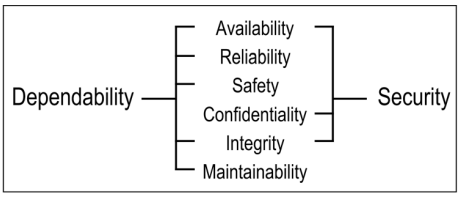


Figure 2.1: Tree of dependability [6]

2.1.1.2 Means

The means of dependability are four categories encompassing the strategies that can be implemented to improve the dependability of a system:

- fault prevention: it aims to prevent the occurrence or introduction of faults. Fault prevention is a part of general engineering. It can rely on software development methodologies (eg. modularization, use of strongly-typed languages), and hardware development methodologies (eg. design requirements). Another approach is to study the introduction of faults, and aim at eliminating their causes to reduce the induced faults via process modifications.

- fault tolerance: it aims to avoid service failures in the presence of faults. The two main techniques of fault tolerance are error detection and system recovery and usually require some form of redundancy. We will detail this particular mean in section 2.1.2.

- faults removal: it aims to reduce the number and severity of faults. Fault removal may happen during the system's development, or its operation. During the development phase, fault removal consists of three steps: verification, diagnosis and correction. Fault removal while in operation is usually done through preventive or corrective maintenance. Preventing maintenance aims to remove faults when the system is already in operation before they cause errors, and corrective maintenance removes faults that already triggered errors and have been reported.

- fault forecasting: it aims to estimate the present number, the future incidence, and the likely consequences of faults. Fault forecasting is done through an evaluation of the system behavior with respect to fault occurrence or activation. The evaluation can take two forms. The first one is the qualitative or ordinal evaluation, which aims to identify and classify the failure modes or event combinations that lead to failures. The second one is quantitative or probabilistic evaluation which evaluates in terms of probabilities the extent to which some of the attributes are satisfied, ultimately giving some sort of measures.

2.1.1.3 Threats

We differentiate three kinds of threats to dependability: failures, errors, and faults. These threats will endanger the correct service delivered by the system, which is the deliverance of the system's intended functions. We detail now the different threats:

- failure: it is an event that occurs when the delivered service deviates from correct service. For example, the failure of a car's braking system could be the inability to slow down the car. A service fails either because it does not comply with the functional specification, or because this specification did not adequately describe the system function. A system's failure may result in an external fault to other systems with which it is connected. Failures can be characterized based on four viewpoints: the failure domain, the detectability of the failure, the consistency of the failure, and the consequences on the environment. The failure domain differentiates failures on the content when the information provided is incorrect and failures on the timing when the service provided is not provided in time (too early or too late). Note that a failure can have both incorrect content and incorrect timing. When the service is halted (the external state becomes constant, or the system activity, if there is any, is no longer perceptible to the users), we describe it has a halt failure (or simply halt). When the service is delivered, but is erratic, we describe it as an erratic failure. The detectability of failures distinguishes signaled failures and unsignaled failures. The consistency of failures determines if the incorrect service is perceived identically by all users (consistent failures), or if some or all users perceive the incorrect service differently (inconsistent failures). Finally, the consequences of the failures determine the failures severities. The severities are related to the application of the system, and to the security attributes (availability, reliability, safety, integrity, maintainability) considered for the application. Some criteria for the determination of the severities could be the outage duration (related to the availability attribute), or the possibility of human lives being endangered (related to the safety attribute). In general, the lowest severity categorizes a failure as minor when the harmful consequences are of similar cost to the benefits provided by the correct service. Similarly, the highest level categorizes the failure as catastrophic when the cost of harmful consequences is orders of magnitude, or even incommensurably, higher than the benefit provided by correct service delivery.

- error: it is a deviation of the state of the system from the correct state. This deviation may cause a failure, either directly or through error propagation. In our example, an error causing the failure of the braking system may be "the throttle is up when it should be down".

- fault: it is the adjudged or hypothesized cause of an error. Faults can be internal or external to a system. There exist several categorizations

that classify possible faults. One of the categorization divides faults into three groups: development faults, physical faults (affecting hardware) and interaction faults. Faults can also be characterized by 8 basic view points that lead to elementary faults class:

- the phase of creation or occurrence, which differentiates whether the fault appeared during the development or in operation.
- the system boundaries, which differentiates internal and external faults.
- the phenomenological cause, which differentiates the cause is natural or human made.
- the dimension, that differentiates hardware and software faults.
- the objective (only applied to human made faults), which differentiates between malicious and non malicious faults.
- the intent (only applied to human made faults), which differentiates whether the introduction was deliberated or not. Note that when the objective is malicious, the fault is necessary deliberated. However, when the fault is non malicious, the intent can be *non deliberated* (for example, if an operator is not aware of the fault introduction because of some distraction), or *deliberated* (for example when an operator is doing consciously an action that he thinks to be correct, but is actually a fault).
- the capability, which differentiates whether the fault appeared inadvertently, or because of incompetence.
- the persistence of the fault, which differentiates whether the fault is permanent or transient.

In our example a possible fault resulting in the brake throttle not moving could be gripped mechanics due to age or a lack of maintenance.

To summarize, faults, when activated, trigger errors, which ultimately (possibly through error propagation) can cause a failure. A fault that is not activated is said to be *dormant*. Failures can propagate to other systems. A failure of a component A is an external fault to the component B. We sum up this propagation in figure 2.2.

2.1.2 Fault tolerance

Fault tolerance is the mean of dependability that seeks to maintain a system's correct service despite the presence of faults. Fault tolerance often requires some form of redundancy. Fault tolerance is usually carried out by error detection, error handling, and fault handling. Error handling and fault handling form the *system recovery*. Thus, the three strategies for fault tolerance can be summed up as *error detection* and *system recovery*.

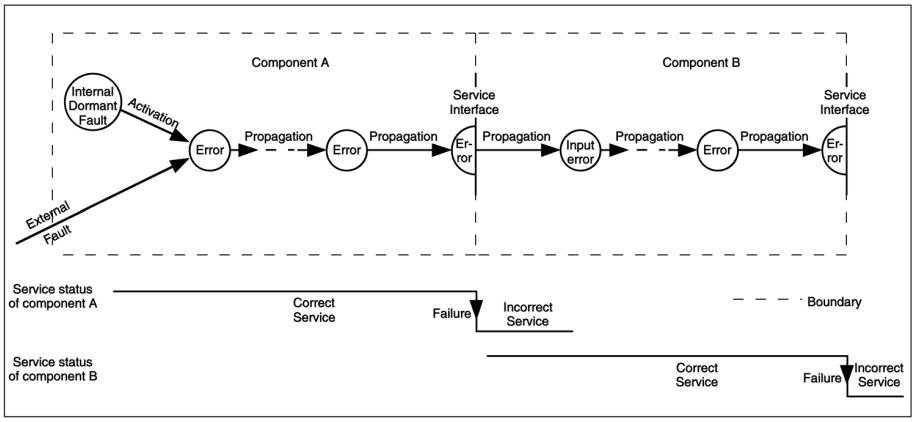


Figure 2.2: Fault, errors and failure propagation in systems ([6])

The error detection and system recovery can be concurrent with execution which means that the detection takes place during normal service delivery (we will refer to this as *online* detection). The detection can also be preemptive, when it takes place while normal service delivery is suspended (we will refer to this as *offline* detection).

The system's recovery aims to transform a system state that contains one or more errors and possibly faults into a state that is without detected errors and faults and can be activated again. We will now describe the system recovery techniques.

Error handling can be achieved through:

- *Rollback*, when the system is brought back to a safe saved state.
- *Rollforward*, when the system is brought to a new state exempt of faults or errors.
- *Compensation*, when redundancy is used to mask the error.

Fault handling is achieved through four successive techniques:

- Diagnosis that identifies and records the causes of errors (in terms of location and types).
- Isolation that performs a physical or logical exclusion of the faulty components from further participation into service delivery, i.e makes the fault dormant.
- Reconfiguration that either switches spare components or reassigns tasks among non-failed components.
- Re-initialization that checks, updates and records the new configuration and updates system tables and records.

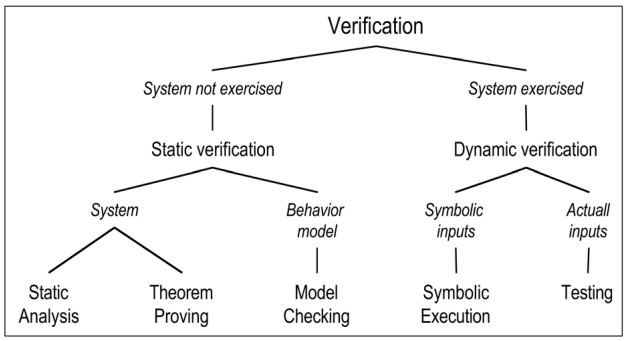


Figure 2.3: Verification approaches

2.1.3 Formal verification

Before describing the concept of formal verification, note that *verification* has to be differentiated from *validation*, although these two concepts are closely related. According to [6] *validation* usually refers to checking the specifications. The standard ISO PAS 21118 [125] (presented in section 2.2.2.2) defines validation as the "set of activities gaining confidence that an item is able to accomplish its expected functionalities and missions". On the other hand, as defined in [6], *verification* is the process of checking whether the system adheres to given properties, termed the verification conditions. There are several types of verification techniques that can be classified based on whether or not they are realized during the operation of the system. When the system is not in operation the verification is said to be static. In this case, two kinds of verification can be realized: analysis on the system itself (through static analysis or theorem proving) and analysis on a model of the system behavior through model checking. In this later case, the model is usually a state-transition model (Petri net, finite or infinite state automate, etc.). When the system is in operation, the verification is said to be dynamic. In this case, the inputs can be symbolic (for symbolic execution), or actual inputs. In the case of actual inputs, this is the very common approach of testing. The figure 2.3 sums up the different verification approaches.

In this classification, we can see that formal verification falls into the theorem proving approaches, but we can also classify it as a fault prevention technique. The main advantages of formal verification are the high level of trust it creates, and the coverage of the method on the system input domain. Indeed in [132], formal verification is presented as a method that explores the complete space of possible design behaviors, unlike testing, simulation, and emulation techniques, which can be used to find and eliminate errors. This makes formal verification a good candidate for safety critical application, such as autonomous systems as mentioned in [135] and [33]. For these reasons formal verification is one of the method that is recommended for the development of system in the automotive systems standards. In the standard ISO-26262 [122], formal verification

is defined as a method used to prove the correctness of an item or an element against the specification of its function or properties in formal notation. Note that in this case formal notation is defined as a description technique that has both its syntax and semantics completely defined. In this standard, formal verification is recommended for software unit verification, the verification of safety requirements and the verification of the software architectural design.

Among formal verification limitations we can find the computing time (there is no guarantee of a timely solution), and the fact that it cannot demonstrate the absence of some design flaws. For example, formal verification may not guarantee that among the design that produce a correct behavior the most efficient one is chosen, or that a formally verified system models correctly the environment. Therefore, formal verification is not a suitable technique to ensure that a system has an efficient design, or has a correct model of its environment. In section 4.4.2, we will see more precisely the consequences of formal verification on a neural network with an incorrect environment model.

2.2 Autonomous vehicles

In this section, we will define what is an autonomous vehicle, what standards are used to define it, and present several architectures used in real autonomous vehicles. We will base our definitions on the ones provided by the SAE (international Society of Automotive Engineers) in [52].

2.2.1 Autonomous vehicle definition

The SAE describes an autonomous vehicle as a vehicle equipped with a *vehicle driving automation system or technology* that performs part or all of the *dynamic driving task* (DDT) on a sustained basis. Note that sustained basis usually characterizes an activity that is realized in a continuous and significant duration, which in our context means an automated driving such that the operator does not have to take over for prolonged periods of time. The capabilities of the *vehicle driving automation systems* can be characterized using 6 levels of *driving automation*. The levels range from *no driving automation* to *full driving automation*:

- Level 0: No Driving Automation
- Level 1: Driver Assistance
- Level 2: Partial Driving Automation
- Level 3: Conditional Driving Automation
- Level 4: High Driving Automation
- Level 5: Full Driving Automation

To determine the level of automation of a driving automation system, the SAE considers the following criteria :

- Whether the driving automation system performs either the longitudinal or the lateral vehicle motion control subtask of the DDT.
- Whether the driving automation system performs both the longitudinal and the lateral vehicle motion control subtasks of the DDT simultaneously.
- Whether the driving automation system also performs the OEDR (Object and Event Detection and Response) subtask of the DDT. The OEDR being the subtasks of the DDT that include monitoring the driving environment (detecting, recognizing, and classifying objects and events and preparing to respond as needed) and executing an appropriate response to such objects and events (i.e., as needed to complete the DDT and/or DDT fallback).
- Whether the driving automation system also performs DDT fallback.
- Whether the driving automation system is limited by an ODD (Operational Design Domain).

Based on these criteria, the levels are determined as follows:

- If the driving automation system performs the sustained longitudinal or lateral vehicle motion control subtasks of the DDT (or both subtasks), the driver does not do so, although he or she is expected to complete the tasks that the DDT does not perform. This division of roles corresponds to Levels 1 (when only one of the subtasks is performed) and 2 (when the two subtasks are performed simultaneously).
- If the driving automation system performs the entire DDT, the user does not do so. However, if a DDT fallback-ready user (see definition above) is expected to take over the DDT when a DDT performance-relevant system failure occurs or when the driving automation system is about to leave its operational design domain (ODD), then that user is expected to be receptive and able to resume DDT performance when alerted to the need to do so. This division of roles corresponds to Level 3.
- Lastly, if a driving automation system can perform the entire DDT and DDT fallback either within a prescribed ODD (Level 4) or in all driver-manageable on-road operating situations (Level 5) then any users present in the vehicle while the ADS (Automated Driving System) is engaged are passengers.

The detailed characterization of each level is given in tables 2.1, 2.2, 2.3, 2.4, 2.5 and 2.6.

Another relevant concept of autonomous driving is the *minimal risk condition*. It is a stable, stopped condition to which a user (aka the driver) or an

Level 0	
Name	No Driving Automation
Sustained lateral and longitudinal vehicle motion control	Driver
OEDR	Driver
DDT fallback	Driver
ODD	n/A
Narrative definition	Performance by the driver of the entire DDT, even when enhanced by active safety systems.

Table 2.1: Level 0 of automation by SAE

Level 1	
Name	Driver Assistance
Sustained lateral and longitudinal vehicle motion control	Driver and System
OEDR	Driver
DDT fallback	Driver
ODD	Limited
Narrative definition	Sustained and ODD-specific execution by a driving automation system of either the lateral or the longitudinal vehicle motion control subtask of the DDT (but not both simultaneously) with the expectation that the driver performs the remainder of the DDT.

Table 2.2: Level 1 of automation by SAE

Level 2	
Name	Partial Driving Automation
Sustained lateral and longitudinal vehicle motion control	System
OEDR	Driver
DDT fallback	Driver
ODD	Limited
Narrative definition	Sustained and ODD-specific execution by a driving automation system of both the lateral and longitudinal vehicle motion control subtasks of the DDT with the expectation that the driver completes the OEDR subtask and supervises the driving automation system.

Table 2.3: Level 2 of automation by SAE

Level 3	
Name	Conditional Driving Automation
Sustained lateral and longitudinal vehicle motion control	System
OEDR	System
DDT fallback	Fallback-ready user (becomes the driver during fallback)
ODD	Limited
Narrative definition	Sustained and ODD-specific performance by an ADS of the entire DDT with the expectation that the DDT fallback-ready user is receptive to ADS-issued requests to intervene, as well as to DDT performance-relevant system failures in other vehicle systems, and will respond appropriately.

Table 2.4: Level 3 of automation by SAE

Level 4	
Name	High Driving Automation
Sustained lateral and longitudinal vehicle motion control	System
OEDR	System
DDT fallback	System
ODD	Limited
Narrative definition	Sustained and ODD-specific performance by an ADS of the entire DDT and DDT fallback without any expectation that a user will respond to a request to intervene.

Table 2.5: Level 4 of automation by SAE

Level 5	
Name	Full Driving Automation
Sustained lateral and longitudinal vehicle motion control	System
OEDR	System
DDT fallback	System
ODD	Unlimited
Narrative definition	Sustained and unconditional (i.e., not ODD-specific) performance by an ADS of the entire DDT and DDT fallback without any expectation that a user will respond to a request to intervene.

Table 2.6: Level 5 of automation by SAE

ADS may bring a vehicle after performing the DDT fallback in order to reduce the risk of a crash when a given trip cannot or should not be continued.

A *fallback-ready user* is the user (aka driver) of a vehicle equipped with an engaged Level 3 ADS feature who is properly qualified and able to operate the vehicle and is receptive to react when an ADS-issued request is sent or when there is some evident DDT system failures in the vehicle. A Level 3 ADS assumes that a fallback-ready user is available to perform the DDT as required. There is no such assumption at Levels 4 and 5.

2.2.2 Standards

Several standards are being developed to ensure the safety of autonomous vehicles. We present here the two main standards in the industry: ISO 26262 [122] that was developed before autonomous driving was taken into account, and ISO PAS 21448 [125] that fills the gap on autonomous driving left by the ISO 26262. A technical report [123] on Safety and Security for Automated Driving system is also available to guide the development of AV. This technical report is currently being used to develop a technical specification (TS) document currently under development, the ISO/AWI TS 5083 [124]. This technical specification covers the "Road vehicles — Safety for automated driving systems — Design, verification and validation".

2.2.2.1 ISO 26262

The standards ISO-26262 [122] is the state of the art standard in the automotive industry for system dependability. It is an adaptation to the automotive applications of the international standard IEC61508. It recommends the use of the V cycle for the development as a reference for the automotive safety life cycle. It suggests to perform a risk analysis like the Hazard Analysis and Risk Assessment (HARA), to identify potential dangers, and then design safety requirements and safety goals to prevent these dangers. It also provides an automotive-specific risk-based approach to determine Automotive Safety Integrity Levels (ASILs), ranking from A to D, with D being the highest. It describes for each of these ASIL levels, what is the applicable requirement to avoid unreasonable residual risk. Finally, it provides requirements for functional safety management, design, implementation, verification, validation and confirmation measures and for relations between customers and suppliers.

2.2.2.2 ISO PAS 21448

The standard ISO PAS 21448 [125] also known as Safety Of The Intended Functionality (SOTIF) is a complement to the ISO 26262 standard. It tackles the limitations of ISO 26262 regarding autonomous vehicles, notably hazardous behavior caused by the intended functionality or performance limitations of a system. The standard provides two examples of these limitations: the inability of the function to correctly comprehend the situation and operate safely (this

also includes functions using machine learning algorithms), and insufficient robustness of the function with respect to sensor input variations or diverse environmental conditions. The term SOTIF is defined in the standard ISO PAS 21448 as the absence of unreasonable risk due to hazards resulting from functional insufficiency of the intended functionality or from reasonably foreseeable misuse by persons. The ISO PAS 21448 also contains an informative annex (annex G) that discusses the verification and validation of off-line training such as those used for machine learning. This annex underlines the potential hazards in the data collection and learning system used in machine learning techniques. It states that machine learning training has the potential to introduce systematic faults without mentioning the potential causes. It presents an example of machine learning process flow from the making of the training set (collection and annotation), to the training, test (here called self verification) and cross verification (with different data). We can note here that for the different training sets, they consider the use of data from three sources on road data collection, structured test and simulation. The authors also underline the importance of a suitable criterion to accept a new trained system (for example, a neural network with new bias and weights values). Finally, the authors recommend to analyze and eliminate possible sources of bias and limitation during the training by using a PFMEA (Process Failure Modes and Effects Analysis). They also mention the importance of considering the coverage and diversity for the data collection and annotation.

2.2.3 Autonomous driving functional architecture.

In this section, we will present several types of architecture for autonomous vehicles. We will start by the Sense-Plan-Act paradigm, one of the first architecture for autonomous systems and which was the basis for many others, but is not used anymore. We will then present the ISO 4804 and detail the components and functional architecture recommended in this standard. We will also present several architectures from academic works: the Stanley vehicle architecture from the DARPA Grand Challenge, the Boss vehicle architecture from the DARPA Urban Challenge, the A1 vehicle architecture from the Korean Autonomous Vehicle Competition and the Ulm university vehicle architecture. Finally, we will present a simplified generic architecture of the car manufacturer Stellantis.

2.2.3.1 Sense-Plan-Act paradigm

The paradigm Sense-Plan-Act is from the robotic and automation literature. This design is almost abandoned in recent architectures, but it inspired several works. The paradigm encompasses three components: a *Sense* component that transforms data from the sensors into a representation of the environment, a *Plan* component that computes the actions to take based on the environment representation, and an *Act* component that sends the commands to the system's actuators. Figure 2.4 presents this model.

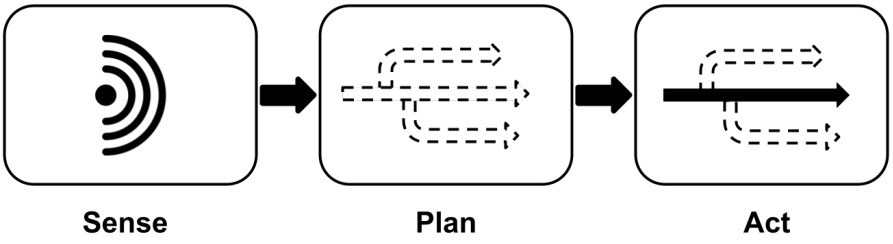


Figure 2.4: Sense-Plan-Act paradigm

2.2.3.2 ISO 4804 architecture

The ISO 4804:2020 [123] is a technical report that presents a possible design for automated driving systems. It considers the complementary and overlapping aspects of the standards ISO 26262 [122], ISO/SAE DIS 21424 [121], ISO/PAS 21448 [125]. The ISO 4804:2020 aims to provide guidelines and recommendations for the development and validation of automated driving systems based on basic safety principles derived from worldwide applicable publications. This standard covers several topics such as verification and validation (V&V) development cycle, safety-by-design, cybersecurity-by-design, taxonomy, etc. The document uses the concept of *capabilities* to describe functions. A capability is defined as the "ability of a product to deliver a function, feature or service". The description of architectures in the document is based on capabilities interactions. We will focus here on the functional architecture presented in it. The architecture is meant to be an illustration but represents all the elements implemented in level 3 and 4 automation functions. We will introduce first the components of the architecture (referred as *elements*). Then we will detail its *capabilities*. Figure 2.5 sums up how the different capabilities interact.

2.2.3.2.1 Components The ISO 4804:2020 [123] provides an extensive list of potential components used in autonomous vehicles. Here we present the categories of these components:

- Environment perception sensors are sensors present on the vehicle that capture data from the real world to contribute to the generation of a representation of the environment. These encompass Cameras, LIDAR, RADAR, Ultrasonic sensors and Microphones.
- A-priori sensors are sensors that provide a kind of data for the environment that is already stored before operation. This includes HD map with Global navigation satellite system (GNSS) references.
- V2X components (to understand as "vehicle to x") encompass tools that exchange information between the ego vehicle and any other objects (other vehicles, traffic lights, infrastructure, etc...). These components may expand the range of the ego vehicle sensing, and provide useful information

about traffic lights, other vehicles, weather conditions, crashes or construction on the road...

- Sensor fusion encompass processes that aim at combining multiple sources of data (sensors data and databases data) into a more detailed and complete representation. Sensor fusion can also incorporate a cross-referencing mechanism to evaluate individual sensor limitations that are not detectable by the sensor itself.

- Interpretation and prediction components aim to anticipate other road users behavior and trajectories. This is typically done using physical models, traffic rules, traffic congestion status, etc.

- Localization components include GNSS based localization, but may be completed by detection of real-world features (natural or artificial landmarks) or detection of the vehicle by environment perception sensors.

- ADS (automated driving system) mode manager is the module in charge of changing between the manual and the different automated driving modes. This tasks is done by considering whether the ego vehicle is currently inside or outside the ODD.

- Egomotion encompasses the components in charge of determining the current state of the ego vehicle in terms of yaw rate, longitudinal acceleration, lateral acceleration, speed, slip angle and more. These values can be measured by sensors (for example inertial measurement unit or wheel ticks) or derived from other sensors such as cameras.

- Drive planning components aim at establishing the driving policy. This task is typically achieved through rule-based decision making, cost-based decision making, game theory or Markov-decision-process-based decision making. The policy is based on two aspects: the explicit traffic rules (signs, lines, trafic lines, ...) and the implicit traffic rules (maintaining a safe distance, potential break of traffic rules for necessary safety-assured maneuvers, ...).

- Traffic rules components provide information on national and regional specificity.

- Motion control components compute the lateral and longitudinal commands based on a given desired trajectory.

- Motion actuators control the steering system (lateral control), braking system (longitudinal control) and the powertrain (longitudinal control).

- Secondary actuators encompass actuators that do not influence directly the egomotion of the vehicle. Example of secondary actuators are indicator lights, headlights and the windscreen wiper motor.

- Human-machine interaction components gather all means of interaction between the vehicle and the user. These components typically use visual, audible and haptic stimulus to interact with the user.
- User state determination components aim to determine occupant attributes such as driver distraction or mode confusion. This is typically done by using sensors directed to the user like camera, seat position sensor, hands-on steering wheel sensor, etc.
- Vehicle state provides information on several of the vehicle's attributes such as the fuel or the electric battery's state of charge, the tire pressures, the oil temperature and level or the doors state.
- Monitoring components evaluate the state and behavior of system elements.
- Processing units run the necessary software and computation for the vehicle to operate.
- Power supply provides the electricity required for the availability of the different items.
- Communication network ensures the communication between the different items to maintain the availability of these.

2.2.3.2.2 Capabilities and summary figure Using a paradigm close to the Sense-Plan-Act but with more reactivity, the architecture defines how the capabilities interact with each other. The capabilities are divided into 2 categories: fail-safe capabilities (FS) that can bring the system to achieve a minimal risk condition, and fail-degraded capabilities (FD) that implement reduced functionalities in the presence of a fault.

The document identifies 13 capabilities among which 7 fail safe capabilities and 6 fail degraded capabilities. We will detail below each of these capabilities:

- *FS_1: Determine location.* This capability aims to determine the location of the vehicle, in particular in regards with the ODD (if the vehicle is inside or outside of the ODD). This capability may rely on environment perception sensor and egomotion.
- *FS_2: Perceive relevant static and dynamic objects.* This capability aims to perceive all relevant entities for the automated driving system to operate. This typically includes the perception of dynamic objects such as other road users and their movement, but also static objects such as road boundaries, traffic guidance, communication signals, or obstacles. This capability typically relies on the fusion of several sensors data and other inputs to build present world model. Inputs used for the fusion contain environment perception sensors, traffic rules, egomotion, localization and V2X.

- *FS_3: Predict the future behavior of relevant objects.* This capability aims to extend the perception of the relevant entities by predicting their future motion. The aim is to estimate the future state and have a forecast of the environment. This capability typically uses the same sensor fusion as in FS_2, and the traffic rules to interpret relevant object movements and make predictions.

- *FS_4: Create a collision-free and lawful driving plan.* Based on the present world model, the prediction of relevant instances, the traffic rules, and the egomotion (used to also to determine the physical limitations), the vehicle can plan the vehicle trajectory. The planning is also done by considering the ADS mode manager. This task takes into account possible inputs from the user (activation or deactivation request), or from monitoring system (ADS no longer available), the state of the vehicle, the state of the user (the seatbelt status or vehicle operator attention), etc. The ADS mode manager may decide to perform a Minimal Risk Maneuver (MRM) to reach a Minimal Risk Condition (MRC).

- *FS_5: Correctly execute and actuate the driving plan.* This capability aims to implement the driving plan made by the capability FS_4. It uses the egomotion and a motion controller to generate lateral and longitudinal commands. These commands are then sent to the vehicle actuators.

- *FS_6: Communicate and interact with other road users.* This capability aims to implement the signaling of the car's behavior and planning to other road users. The signaling can be made through secondary actuators (turn indicator activation), or by direct communication through V2X.

- *FS_7: Determine if specified nominal performance is not achieved.* This capability aims to monitor and detect nominal performance limits. The monitoring can be done at the element or system level but must permit sufficient time to ensure a safe reaction (from the driver or from another system). Its evaluation is used by the ADS mode manager.

- *FD_1: Ensure controllability for the driver.* This capability aims to ensure that the driver is able to take over the ADS when needed. In the case of a SAE level 3 function, the user must be ready within a reasonable time (usually around 10 to 15 seconds). In this case, a constant monitoring of the user should be implemented to check for distraction, mode confusion (when the user does not understand the system current behavior), absence, unreadiness for takeover or other threat to his or her availability.

  In the case of SAE level 4 or 5 functions, the user does not have the constant responsibility to take over the ADS on request. In the case of the SAE level 4, the user may only need to perform an emergency stop when he or she recognizes a hazard or realizes that the vehicle is exiting the ODD.

- *FD_2: Detect when degradation is not available.* This capability aims to monitor if the other fail-degraded capabilities are available. It communicate its results to the ADS mode manager since it may impact the ADS availability.

- *FD_3: Ensure safe mode transitions and awareness.* This capability aims to perform switching between the several automatic and manual modes. This task is mostly achieved by the ADS mode manager, and may impact the drive planning (in case a MRM is needed), and the HMI (Human Machine Interface) in case the user needs to gain back control.

- *FD_4: React to insufficient nominal performance and other failures via degradation.* This capability is also performed by the ADS mode manager. The ADS mode manager uses monitors to detect possible failures or insufficient performances, and then must decide what degradation mode to choose.

- *FD_5: Reduce system performance in the presence of failure for the fail-degraded mode.* This capability aims to implement the response decided by the ADS mode manager, when it asks for a MRM (possibly with degraded constraints like reduced perception range). Following the ADS mode driver decision, the drive planning is established and the information is sent to the operator (for example take over request).

- *FD_6: Perform ODD functional adaptation within reduced system constraints.* This capability aims to determine the new ODD of the system considering its failed components and reduced performances.

We present in figure 2.5 the generic architecture proposed in the document that would implement these functionalities.

2.2.3.3 Stanley vehicle architecture from the DARPA Grand Challenge

The Stanley vehicle that won the 2005 DARPA Grand Challenge [146] used an architecture based on modularity with the use of several layers. All these layers are represented in figure 2.6 and described below:

- a *sensor interface layer* (composed of lasers, camera, GPS, radar, and inertial measurement unit),

- a *perception layer* that uses the sensors input to build several internal models of the environment (maps, obstacle list, vision map, road border for lane positioning) and of the vehicle state (coordinates orientation and velocity, and safe vehicle speeds),

- a *planning and control layer* that contains a path planner that determines the trajectory and then computes the acceleration, braking and steering values,

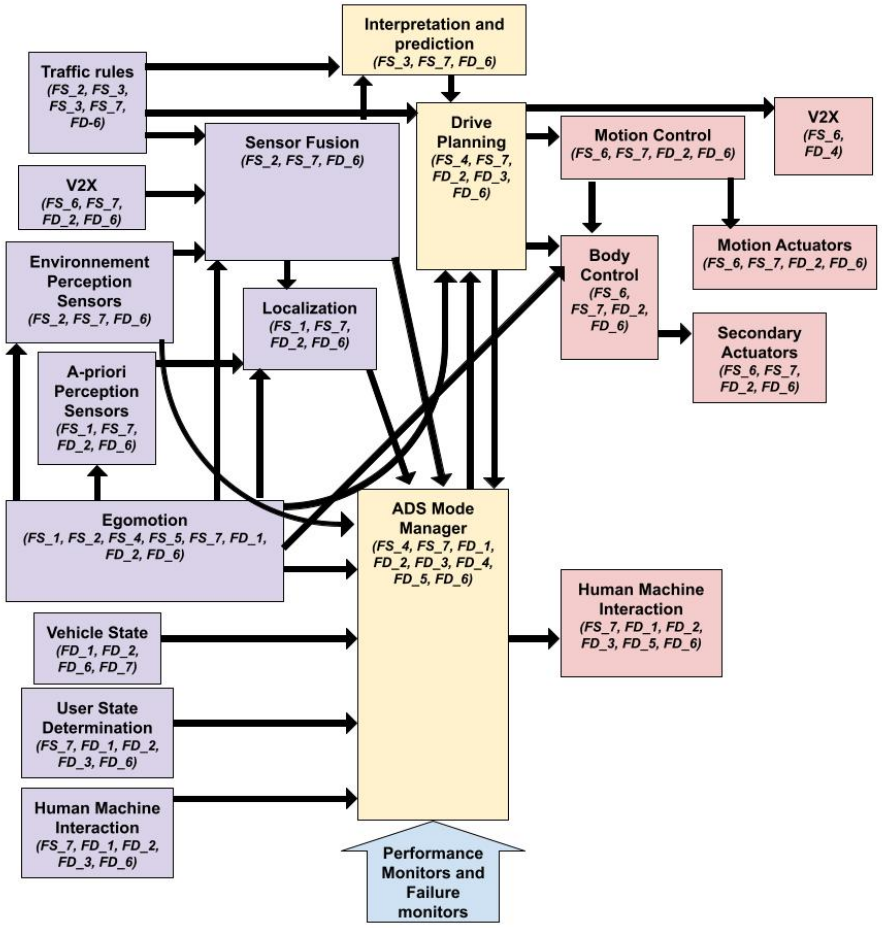


Figure 2.5: Autonomous vehicle generic architecture proposed in [123]

- a *user interface layer* that has a remote stop controller and the controller to start the vehicle software,

- a *vehicle interface layer* that connects the commands computed by the planning and control layer to the vehicle,

- a *global services layer* that performs several tasks like communication, database maintenance, physical power maintenance, health and restart of all systems, clock synchronization, and data logging.

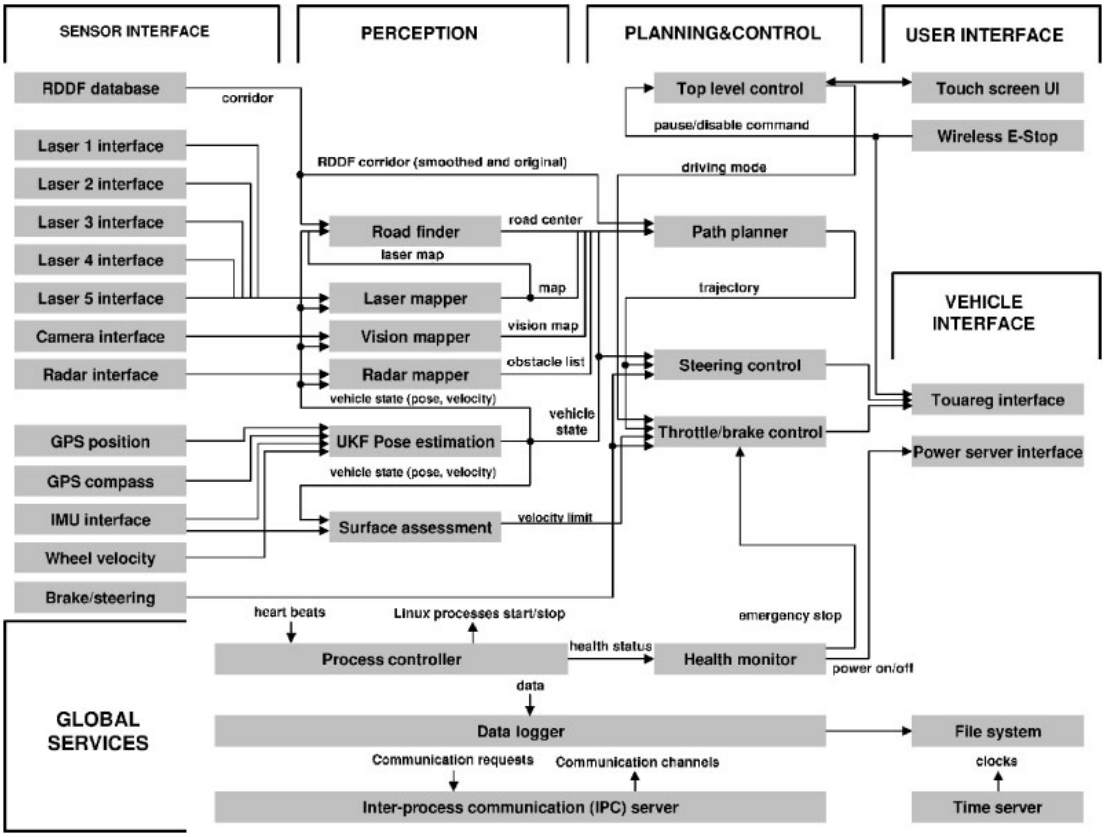


Figure 2.6: Flowchart of Stanley vehicle software system, winner of the DARPA Grand Challenge in 2005. This figure is presented in [146].

2.2.3.4 Boss vehicle architecture from the DARPA Urban Challenge

The Boss vehicle, winner of the 2009 DARPA Urban Challenge (presented in [151] and [150]), used an architecture relying on four subsystems: perception, mission planning, behavioral executive and motion planning. The perception subsystem takes care of making a world model for the other subsystems based on the vehicle sensors inputs. This built model is composed of a static obstacle map, a list of moving vehicles, and the position of the ego vehicle. The mission planning subsystem objective is to compute the cost of all possible routes to reach the desired location. The behavioral subsystem uses the inputs from the mission planning subsystem to provide a problem definition to the motion planning subsystem. The motion planning subsystem is responsible for executing the current motion goal issued by the behavioral subsystem. In figure 2.7, we present a summary of this architecture.

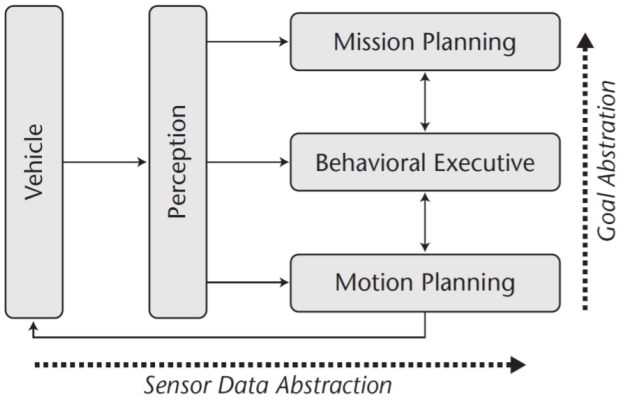


Figure 2.7: Architecture of Boss vehicle software system, winner of the DARPA Urban Challenge. This figure is presented in [150].

2.2.3.5 A1 from the Korean Autonomous Vehicle Competition

The architecture presented in [54] was implemented on the winning vehicle A1 of the Korean Autonomous Vehicle Competition in 2012. It is composed of a layer of sensors (a GPS, an inertial measurement unit, on-board sensors such as electronic stability control system, 2 cameras, 2 multilayer laser scanners, 4 single layer laser scanners, a Real-Time Kinematics GPS and a Differential GPS), a fusion layer, and five basic functions which are localization, perception, planning, vehicle control and system management. The perception function identifies objects using a machine learning algorithm and offers information on surrounding environments. The localization function determines the location of the vehicle using the two GPS systems, the inertial measurement unit and the on-board sensors. The outputs of these two functions are fed to the fusion layer that provides information to the planning function and vehicle control function. The planning function determines the trajectory of the vehicle and provides it to the vehicle control function. The vehicle control system is divided into a lateral controller (that acts on the steering controllers) and a longitudinal controller (that acts on the acceleration and braking controllers). Finally, the system management function supervises the entire autonomous driving system. The architecture also contains a human–machine interface (HMI), a driving mode management system, and a fault management system. We present a summary of this architecture in figure 2.8.

2.2.3.6 Ulm university vehicle architecture

The university of Ulm proposed an architecture for an autonomous driving vehicle in 2015 in [70]. This architecture includes modules for perception, situation analysis, decision making, motion planning and control of actuators:

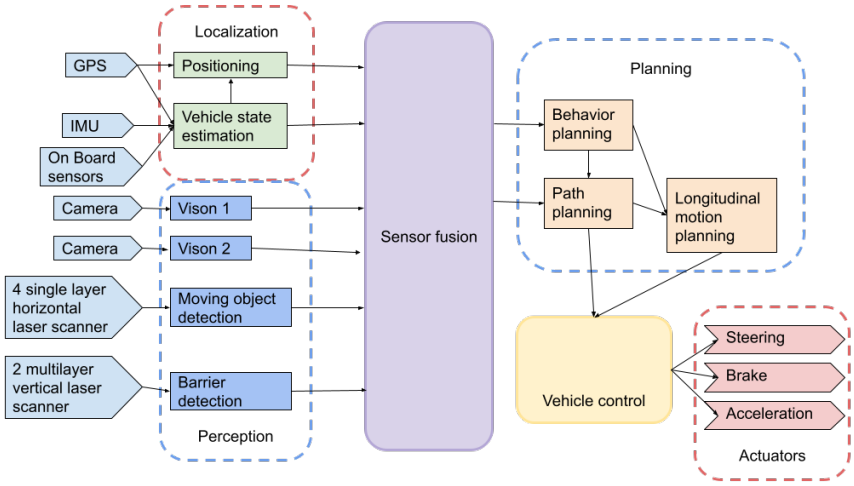


Figure 2.8: Architecture of the winning vehicle A1 of the Korean Autonomous Vehicle Competition in 2012

- Perception: The perception relies on several sensors, and is divided in 3 consecutive modules: Grid Mapping System, Localization System, Tracking System. The grid mapping system combines data from three laser scanners and offers the possibility to integrate new sensors. The result is a grid cell map of the vehicle environment with estimated occupancy probability for each grid cell. The localization system uses data from a high-precision real time kinematic positioning system, and extracts features from the grid map, and from images taken with a front-facing camera to identify landmarks. Other sensors (like radar) may also be added. The result of this process is a digital map with the ego position. Finally the tracking system aims to perceive the dynamic objects in the environment. Data from several sensors (radar, laser and video sensors) are preprocessed and used with inputs from the grid map and the digital map to provide an objects list in the environment. Finally, a combination module combines the grid map, the localization result, the digital map and the object list to a consistent environment model.

- Situation analysis: The situation analysis module interprets the environment model and creates long-term predictions for other vehicles.

- Decision making: The decision making module uses the predicted environment model and the itinerary to compute permitted maneuvers at certain passages.

- Motion planning: The motion planning provides a trajectory for the vehicle using as inputs the predicted environment model and the permitted

maneuvers. It is divided into two layers: a behavioral layer (that aims to mimic human driving), and a trajectory layer.

- Control of actuators: The trajectory with the lowest cost is given to the vehicle control and actuation module that takes cares of implementing the desired path.

The figure 2.9 presents a summary of this architecture.

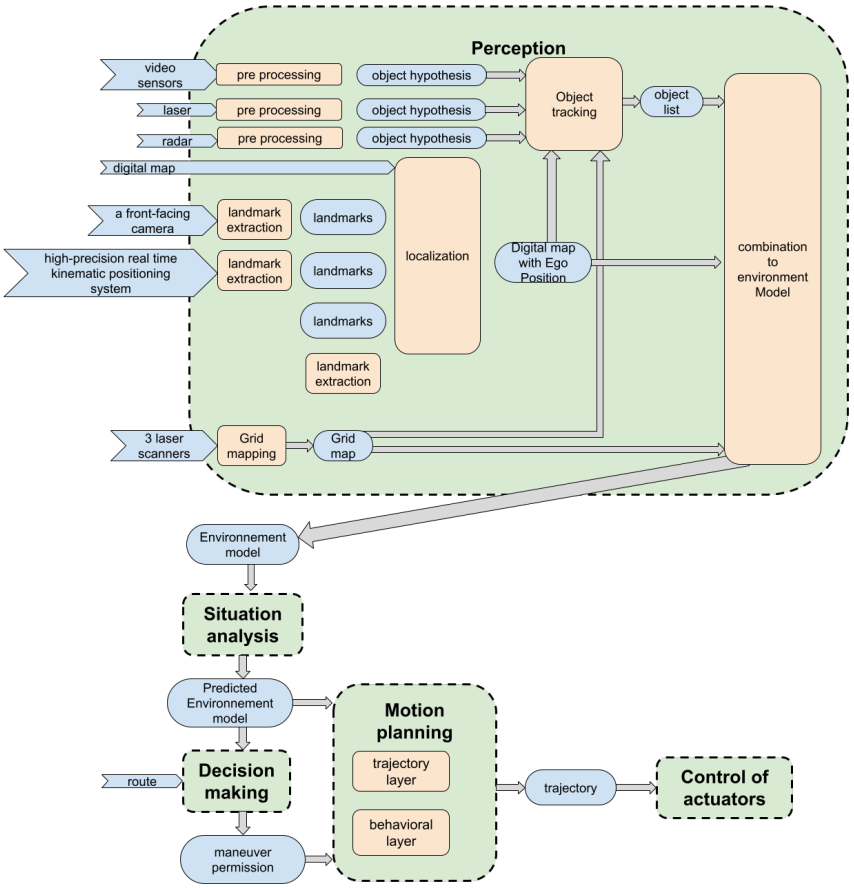


Figure 2.9: Architecture of the autonomous vehicle of Ulm university

2.2.3.7 Stellantis functional architecture for L2 to L5 automated functions

The car manufacturer Stellantis proposes a generic architecture [142] to implement L2 to L5 automated functions. This architecture can be decomposed into four modules: acquisition, analysis, decision, action.

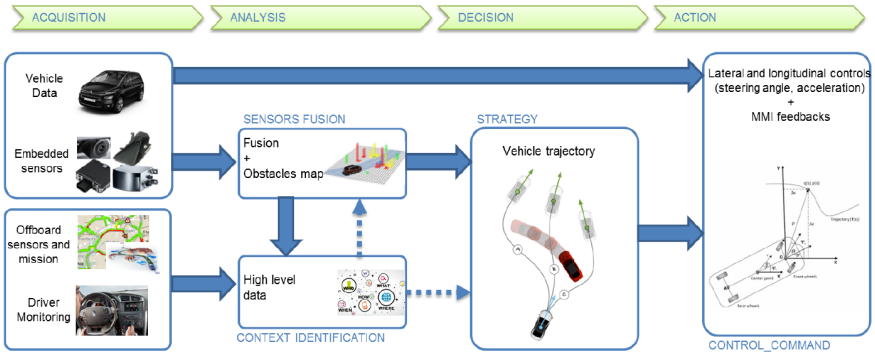


Figure 2.10: Stellantis functional architecture for L2 to L5 automated functions

- *Acquisition*: the acquisition phase aims to acquire data from the exteroceptive sensors fitted to the vehicles (sensors that give measures on the environment like radar, lidar, cameras, ...), from the proprioceptive sensors on the vehicles (sensors that gives measures on the system's state, like odometers, throttle sensors, etc.) but also from off board sources (data from other vehicles or the infrastructure) or from driver monitoring systems.

- *Analysis*: these modules aggregate data from the different inputs in order to build a representation of the environment (for example, a 3D map of the surrounding vehicles). They typically perform data fusion, classification and identification of the elements of the vehicle environment.

- *Decision*: the decision modules will determine the vehicle guiding strategy. This strategy includes trajectory planning, maneuvers, emergency maneuvers, choice of speed, wheel angle, etc. The outputs of these modules are commands to be sent to the vehicle actuators.

- *Action*: the action module is in charge of actually implementing the actuators commands decided in the previous module.

2.3 Conclusion

In this chapter we gave on overview of the concepts of dependability: we explained what are the attributes, the means and the threats of dependability. We also detailed the mean of fault tolerance, and on formal verification techniques. Then we presented the current state of autonomous vehicles in the research community and the industry. We introduced the automation levels classification, and the current standards used to develop autonomous driving functions. Finally we gave an overview of known architecture for autonomous vehicles. Among the important points for the rest of the thesis we can note that:

- formal verification is a dependability technique that contribute to fault prevention and fault elimination. It is based on the proof that a system respects one or several safety properties. It is a good candidate for ensuring the safety of critical applications, particularly machine learning techniques evolving in open environments, as they offer proofs on entire domain range and do not require to know the correct output of the system .
- fault tolerance is one of the dependability mean. It seeks to maintain a system's correct service despite the presence of faults, and the methods contributing to this mean rely on redundancy and diversification.
- autonomous driving is a complex task that will be implemented by developing successively more autonomous driving functions. These autonomous function can be classified following 5 levels of automation. Standards to ensure the dependability and software architecture are currently being developed for autonomous vehicles.
- each autonomous function is achieved through 4 different phases: perception, fusion, decision and action. In this thesis work, we will focus on the decision part. Its role is to decide what should be the behavior of the vehicle, and more precisely determine what should the actuators command based on a representation of the environment given by the perception and fusion phases.

Chapter 3

Neural networks: definitions and formal verification

Neural networks are an artificial intelligence technique classified as a machine learning approach. In the last decade, neural network have shown great potential for several applications and are successfully implemented in some (like face recognition, voice recognition, marketing, etc...). Their success is mostly explained by their ability to generalize well and operate in open or vast environments. Their use in other applications such as safety critical applications could permit the development of new technologies such as autonomous vehicles. However they did not reach yet a level of safety that can allow their safe use in such applications. In this chapter we introduce the context and the state of the art related to neural networks and formal verification for neural networks. First we will define what are neural networks and focus on their applications for autonomous vehicles and on works existing to improve their safety. Finally, we will present a state of the art on formal verification for neural networks.

3.1 Neural Networks

Neural networks (or Artificial Neural Networks) are one of the most popular machine learning technique. They aim to automatize complex tasks by learning from data. Their ability to learn a function and generalize make them common solutions in various domains such as image recognition, voice recognition, text analysis... The first neural network was introduced in 1958 as the perceptron model proposed in [126].

In this section, we first present a definition of neural networks based on the simple feedforward architecture, popular activation functions, and the tasks and applications where neural networks are commonly used. We will then present the

learning process of neural networks by presenting the generalization concept, the training process, the roles of backpropagation and of the optimization algorithm. We will also present the role and structure of the datasets used to train a neural network. Then we will present several popular neural network architectures, followed by their potential use in autonomous vehicles. We will finish this section with a state of the art and discussion about the safety of neural networks.

3.1.1 Definition

In this section, we will present a definition of neural networks and describe the simple feedfoward architecture. We will then present the common activation functions used in neural networks, and the tasks and applications a neural network can perform.

3.1.1.1 Architecture of NN

Neural networks are function approximators. They are a model based machine learning technique and can be used in supervised learning, unsupervised learning and reinforced learning. We will focus in our work on supervised learning, where the neural network aims to learn from a dataset the correct output for a given input. A neural network can in fact be represented as a mathematical function $f_{nn} = \mathcal{X} \rightarrow \mathcal{Y}$, with $\mathcal{X}$ being the input space, and $\mathcal{Y}$ the output space. The f_{nn} function associates an input vector X of size i to a computed output vector $\hat{Y}$ (often called the prediction of the neural network) such as $f_{nn}(X) = \hat{Y}$. We will note as $f_{desired}$, the function that the neural networks tries to approximate, which is often unknown. Neural networks are composed of layers. The first layer, also called input layer, corresponds to the input vector X and is thus of size i. The following layers are called the hidden layers and are composed of interconnected neurons. The last layer is the output layer, which corresponds to the output vector $\hat{Y}$. Figure 3.1 presents a fully connected neural network with an input layer of size 3, two hidden layers of size 2 and 3, and an output vector of size 2.

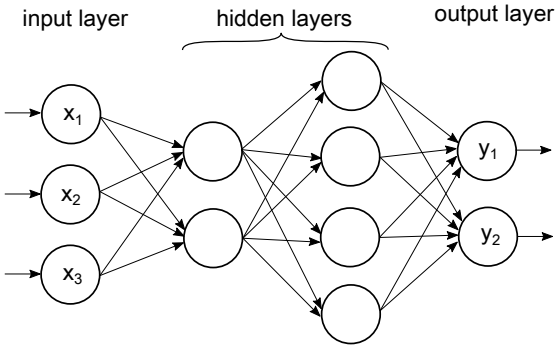


Figure 3.1: Neural network example

Several architectures exist for neural networks. We will focus first on fully connected neural networks but other architectures are presented in section 3.1.4. In a fully connected architecture, each neuron is connected to all neurons from the previous layers through arcs, each arc having an associated weight noted W_i. For each neuron, the vector of all weights is noted $W_{neuron} = (w_1, w_2, ..., w_k)$ with k being the size of the previous layer. The neurons have an input vector $X_{neuron} = (x_1, x_2, ..., x_k)$, whose values correspond to the values of every neurons in the previous layer. The neuron has an output value y_{neuron} computed as $\hat{Y}_{neuron} = f_{non-linear}(X^T_{neuron} W_{neuron} + b_{neuron})$, with $f_{non-linear}$ being the activation function and b_{neuron} a bias specific to each neuron. The figure 3.2 represents one such neuron. Note that we will present common activation functions in the next section.

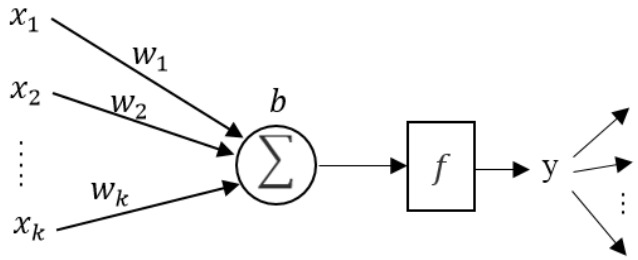


Figure 3.2: Artificial neuron example

3.1.1.2 Activation functions

Activation functions are the functions that determine each neuron's outputs from its inputs, weights and bias. They are non-linear function because non-linear patterns (that are present in the real life) can hardly be captured with a linear model. We present below the most commonly used activation functions in neural networks. We will use the following notation: for a given neuron j in the current layer, n is the number of neurons in the previous layer, m is the number of neurons in the current layer, $x = x_1, x_2, ..., x_n$ the outputs values of the previous layer, $w_{i,j}$ is the weight value on the arc connecting the neuron i of the previous layer to the neuron j, $z_j = \sum_{i=1}^{n} w_{i,j} * x_i$ and b_j the bias of the neuron j.

- the sigmoid function, which produces outputs values between 0 and 1: $S(z_j, b_j) = \frac{1}{1+e^{-(z_j+b_j)}}$
- the hyperbolic tangent function, which produces outputs values between -1 and 1: $f(z_j, b_j) = \frac{e^{z_j+b_j} - e^{-(z_j+b_j)}}{e^{z_j+b_j} + e^{-(z_j+b_j)}}$
- the ReLU (Rectified Linear Unit) function:

$$f(z_j + b_j) = \begin{cases} 0 & for\ z_j + b_j \leq 0 \\ z_j + b_j & for\ z_j + b_j > 0 \end{cases}$$

- the softmax function which produces outputs values between 0 and 1 whose total sum is equal to 1, akin to a probability function. The function is applied to all neurons of a given layer (usually the last one), we then denote here the function of the neuron j as $softmax_j$:

  $softmax_j(z_1, b_1, z_2, b_2, ..., z_m, b_m) = \frac{exp(z_j+b_j)}{\sum_{k=1}^{k=m} exp(z_k+b_k)}$ with exp the exponential.

More details on these activation functions and other ones are presented in [24].

3.1.1.3 Tasks

Neural networks in supervised learning can perform several tasks among which the following. Note that we do not aim here to provide an overview of all existing tasks, but just to give an idea of what is possible.

- classification tasks, when the output vector Y is a category (or class). For example classifying photos based on the presence or absence of a pedestrian.

- regression tasks, when the output vector is a value. For example determining the value of the acceleration command of a vehicle based on its current speed and the speed limit.

- transcription tasks, when the output of the neural network is a text structured data (for example, giving a textual description for pictures). A similar task is the translation task (from English to French for example).

- others tasks include denoising (when the objective is to remove noise from the input vector), synthesis and sampling (when the objective is to generate similar inputs to the ones in the dataset)...

3.1.1.4 Applications

The current and possible applications of neural networks are very diverse. In [45], the authors present a survey of several applications of deep learning neural networks. There are several domains covered in this survey such as:

- image and video applications like in [49] where the authors use a MLP (Multi Layer Perceptron) to reconstruct the quality of a compressed video frames. Another example is proposed in [141] where the author use a LSTM (Long Short-Term Memory) NN to perform analysis of facial expression responses of individuals watching advertisements.

- smart assistants like in [45] where audio processing is used for this domain.

- sentiment analysis in text like in [3] where the authors use deep learning to perform sentiment analysis through text analysis.
- autonomous systems and robotics ; we will particularly focus in our work on autonomous vehicles in section 3.1.5.
- medical and biological applications such as the image recognition application presented in [144], where a neural network is used to detect potential errors in radiation therapy delivery. Another example is presented in [154] where the authors use deep learning to classify lung tissues from images obtained by high resolution computed tomography scans. The field of computational biology also has neural networks application such as the work presented in [112] where the authors use a deep learning approach to predict DNA binding proteins solely from their primary sequences.
- physical sciences, like the work presented in [37] which explored the use of neural networks to detect gravitational waves produced by binary black hole mergers.
- architectural and algorithmic enhancement like in [63] that investigated the effects of batch size in DNN (Deep Neural Network) training via Stochastic Gradient Descent (SGD), namely that large batches converge to sharp minima and result in poor generalization.
- cyber security applications like in [174] where the authors present an Android malware detection tool called DeepFlow, based on deep learning.
- finance, economics and market analysis, like the work presented in [173] that describes a deep learning ensemble approach for crude oil price forecasting.

3.1.2 Learning the desired function $f_{desired}$

In this section, we will discuss the processes used to learn the desired function $f_{desired}$. We will first describe the generalization phenomena. Then we will present the role and inner working of the training process of a neural network which permits the neural network function f_{nn} to fit the desired function $f_{desired}$. We will present the different tools and mechanisms used in the learning process: the cost function, the backpropagation, and the optimization algorithm. Finally we will mention the role of the dataset in learning.

3.1.2.1 Generalization

One of the big advantage of neural networks is their capability to generalize, that is the ability of a model to perform well on previously unobserved inputs ([40]). Because of this generalization capability, neural networks are an attractive solution for problems with open environments such as autonomous vehicles. However, generalization is impaired by two opposed problems: underfitting and

overfittingThese phenomena can be measured by comparing the training error (error computed on the training set), and the testing error (error computed on the test set, which may also be called test error).We will give more details on the training set and the test set in section 3.1.3.

- underfitting occurs when the model did not reach its maximum capacity to fit the function to model, or in other words when the test error did not reach its minimal value. It typically occurs when the training is interrupted too early. Underfitting can be seen as f_{nn} being not precise enough on the training data.
- overfitting occurs wen the model fits too precisely the training set and actually lost some ability to generalize on other inputs, or in other words when the training error decreases while the testing error increases. Overfitting can be seen as f_{nn} being too specialized on the training data and then imprecise during operation on examples not encountered during the training.

We sum up in figure 3.3 the evolution of the two errors, and the separation between the two phenomena.

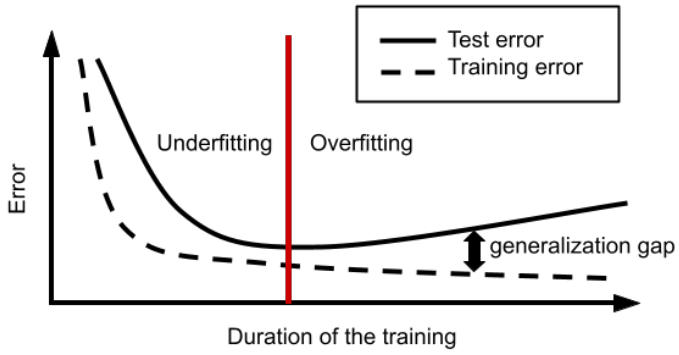


Figure 3.3: Training Error and test error for NN

3.1.2.2 Training

The weights and bias of each neuron are called the parameters of the neural network, and their values are determined through a process called training. The training uses a machine learning algorithm (learning algorithm, or training algorithm) and the training dataset to determine the parameters values. Training a neural network is an iterative process, where at each step, the neural network parameters (weight and biases) are updated. The update of the parameters is done using an optimizer (or optimization algorithm) and the back propagation. At each step, the value of all elements of an entire batch (or the entire training

set if there is one batch) is computed and compared to the ground truth using the cost function. Then the optimizer computes how the last layers values should be updated to get closer to the ground truth. These updates are then propagated to the previous layers using back-propagation. By updating the weights and biases, the training algorithm makes the neural network function f_{nn} improve its approximation of the desired function $f_{desired}$ at each step.

3.1.2.3 Cost function

The cost function (also called loss function) provides a measure that compares the computed output of a given input with its actual output. The objective of the algorithm is to minimize this difference, so the neural network output approaches the real value. Several cost functions exist and the choice is usually based on the type of task the neural network should perform (regression, binary classification, multi label classification, ...). Among the possible cost functions, we can identify the following:

- The Mean Square Error (MSE) is a cost function mostly used for regression tasks. It is the average squared error over all the elements of the training set. Its mathematical expression is $MSE = \frac{1}{2n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$ with n being the size of the dataset, y the real output given in the dataset pairs, and $\hat{y}$ the outputs computed by the neural network.

- The Binary cross entropy is a cost function used when the task to perform is a binary classification (classification with only two possible labels). In case of binary classification a common approach is to use normalized outputs. The neural network computes the probability of the element to be in one class or the other, the single output is then classified as label 0 or label 1 based on which one it is the closest to. The binary cross entropy loss compares the computed probability to the actual class with the following formula: $LogLoss = \frac{1}{n}\sum_{i=1}^{n} -(y_i * log(p_i) + (1 - y_i) * log(1 - p_i))$, with y_i being the actual class of the element i (either 0 or 1), p_i being the probability of class 1 (aka, the computed output of the neural network, for the element i) and $(1 - p_i)$ the probability of class 0. Note that binary cross entropy is a specific case (with two classes) of the multi-call case presented below. Furthermore the term *LogLoss* is often used to designate one or the other.

- The cross entropy for multi-class classification is designed to handle several possible labels. This technique uses a neural network with as many outputs as labels. Each output represents the probability of the element to be classified with its label. The cross entropy for multi-class classification works in a similar manner to the binary cross entropy cost function, but makes a sum over all the computed class probabilities. Its formula is the following:

  $LogLoss = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{m} y_{ij} log(p_{ij})$, with m being the number of possible classes, y_{ij} the actual probability of the i^{th} element of the dataset to

be of class j (the ground truth that we wish the NN learns). Note that y_{ij} has a value of either 0 or 1. Finally p_{ij} is the probability given by the neural network that the i^{th} element of the dataset is of class j.

3.1.2.4 Backpropagation

Popularized in 1986 in [131], back-propagation is the part of the learning algorithm that computes the Jacobian of the cost function with respect to the parameters noted as $\nabla_\theta J(\theta_{t-1})$. As reminder, the Jacobian is equivalent to the gradient of a function, but also covers functions that has several outputs, and the gradient is the equivalent of the derivative but for functions with several variables. It gives the direction of how to update the weights and biases of the neural network for a loss computed on one or several outputs. This process is done using the chain rule of calculus.

We present now the process for a MLP (Multi Lyaer Perceptron) composed of two layers of one neuron each. The neural network has three parameters: the weight w_1 of the arc connecting the input layer to the first hidden layer, the bias b_1 of the first hidden layer neuron, the weight of the arc connecting the first layer to the output layer w_2, the bias of the last neuron b_2. We also denote f_1 and f_2 the activation functions of the hidden and output neuron, $\hat{y}$ the output value of the NN, and y the ground truth. We show in figure 3.4 the architecture of this NN.

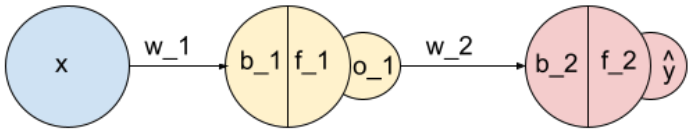


Figure 3.4: Architecture of the NN used for backpropagation example.

The output can be now expressed as a composition of several functions as follows: $\hat{y} = f_2(b_2 + w_2 * f_1(b_1 + w_1 * x))$. The objective of backpropagation is then to compute the gradient of the cost function with respect to the parameters. The cost function used in our example is the mean squared error that we will write as $L(b_1, w_1, b_2, w_2, X_{data\ for\ backprob}) = \frac{1}{2n}\sum_{i=1}^{n}(y_i - f_2(b_2 + w_2 * f_1(b_1 + w_1 * x_i))^2$, with X being the set of n elements (x, y) used for the computation of the loss. Finally, we compute all the gradients of the cost function with respect to the parameters using the chain rule: $\frac{\partial L}{\partial w_1}$, $\frac{\partial L}{\partial w_2}$, $\frac{\partial L}{\partial b_1}$ and $\frac{\partial L}{\partial b_2}$. These gradients are then used to update the parameters using the optimization algorithm. A basic example could be: $w_{1_t} = w_{1_{t-1}} - \eta * \frac{\partial L}{\partial w_1}$ with $w_{1_{t-1}}$ the previous value of the parameter, w_{1_t} the new value of the parameter, η a value that accelerates or slows down the change of parameter value.

3.1.2.5 Optimization algorithms

The role of the optimization algorithm is to determine how neural network parameters (weights and biases) should change following the Jacobian computed by the backpropagation. In the descriptions below, θ_t will designate the parameters updated values (at a new step) and θ_{t-1} the previous parameters values. The optimization algorithm also impacts how the backpropagation computes the Jacobian (which data is used). We describe here several optimization algorithms:

- *The batch gradient descent* is based on updating the parameters θ following equation 3.1 with $\nabla_\theta J(\theta_{t-1})$ being the Jacobian of the cost function computed at each step using the entire training set. The learning rate is η, it must be in $[0; 1]$ and it is usually in $[0.01; 0.1]$. This method is presented in the context of deep learning in [13] and is based on the gradient descent method originally presented in [5].

$$\theta_t = \theta_{t-1} - \eta * \nabla_\theta J(\theta_{t-1}) \tag{3.1}$$

- *The stochastic gradient descent* is similar to the previous method, but the Jacobian is computed on one element at each step. The new equation to compute the updated parameters is 3.2, with (x_i, y_i) being a random single element of the training set, and $\nabla_\theta J(\theta_{t-1}, x_i, y_i)$, the Jacobian at each step on the training element (x_i, y_i). This algorithm is detailed in [13].

$$\theta_t = \theta_{t-1} - \eta * \nabla_\theta J(\theta_{t-1}, x_i, y_i) \tag{3.2}$$

- *The mini-Batch Gradient Descent* is similar to Stochastic gradient descent, but this time the Jacobian is computed on several elements sampled from the training set (a batch). The new equation to compute the updated parameters is 3.3, with n being the size of the mini batch (number of elements randomly sampled form the training set to compute the Jacobian). This method is presented in [40] under the name "Stochastic gradient descent". We can note here that each batch $x_{i:i+n}$ and $y_{i:i+n}$ is composed of successive elements of the training set. In practice, the training set is shuffled so the order of elements in the training set and in the batches is random.

$$\theta_t = \theta_{t-1} - \eta * \nabla_\theta J(\theta_{t-1}, x_{i:i+n}, y_{i:i+n}) \tag{3.3}$$

- *The momentum technique* is based on an update of the parameters that accelerates when the Jacobians are small and consistent, noisy, or when the Jacobians points in the same direction. It permit to accelerate the learning. This method, introduced in [108] and presented in [40] uses equation 3.4 to update the parameters. The terms v_t and γ are respectively called the velocity, and the momentum term. The momentum term determines how quickly the contributions of previous Jacobians decay. Note that it

uses a gradient calculated on a batch, similarly to the mini-batch gradient descent method.

$$\begin{aligned} v_t &= \gamma * v_{t-1} + \eta * \nabla_\theta J(\theta_{t-1}, x_{i:i+n}, y_{i:i+n}) \\ \theta_t &= \theta_{t-1} - v_t \end{aligned} \tag{3.4}$$

- *The adagrad method* aims to give a rapid decrease of the learning rate to parameters with a large loss derivative, and a small decrease to parameters with a small change in derivative. Here, each parameter is noted as θ_j. The method computes the new values $\theta_{j,t}$ from the equations 3.5, 3.6 and 3.7. Note that ϵ is a small value that prevents division by zero, usually with a value of $1 * 10^{-8}$. This method is presented in [40] and was introduced in [25].

$$g_{j,t-1} = \nabla_{\theta_j} J(\theta_{(j,t-1)}, x_{i:i+n}, y_{i:i+n}) \tag{3.5}$$

$$r_{j,t} = r_{j,t-1} + (g_{j,t-1})^2 \tag{3.6}$$

$$\theta_{j,t} = \frac{\eta}{\sqrt{r_{j,t}} + \epsilon} * g_{j,t-1} \tag{3.7}$$

An overview of these algorithms and other is presented in [130] and in [40].

3.1.3 Datasets

Machine learning algorithms use data to learn to perform a task. In the case of neural networks, the datasets are used for the neural network to approximate the desired function. A dataset is a set of pairs of input/output (X, Y) usually split into 3 sub datasets: the training set, the validation set and the test set. The training set is used by the learning algorithm to determine the parameters of the model (the weights and bias of each neuron). When the learning is done, it is common to observe how the training can be improved by tuning some hyper parameters and repeat the learning. Hyper parameters for neural networks are typically the architecture of the neural network (number of layers, number of neurons per layers, architecture type,...) and parameters related to the training algorithm (optimization algorithm parameters, number of iterations, condition of ending, number of batch, cost function, bias and weight initialization technique ...). The validation set is used to compare neural networks obtained with different hyper parameters and choose the best ones. Finally, the test set is used to evaluate the final ability of the neural network to predict accurately the outputs.

To evaluate a model, the machine learning community uses the concepts of training error and test error. Training error is computed on the training set using the cost function. We can measure its evolution over the iterations of the training algorithm (or also over the training time) and it is used to improve the learning of the NN. The test error, that is also called generalization error, is computed on the test set and represents the correctness of the neural network predictions on data that was not used to train the NN.

There are other types of learning process than the one relying on 3 datasets (training, test and validation). For example, reinforcement learning is a set of methods where the neural network (or another machine learning model) can be trained while in operation. More information about these methods can be found in [90]. As previously stated, we will focus in this work on supervised learning but we may quote some references that use reinforcement learning as illustration of neural network applications.

3.1.4 Types of neural network architecture

In this section, we will present the most common architectures for neural networks.

3.1.4.1 Feedfoward Multi Layer Perceptron neural network

A feedfoward Multi Layer Perceptron neural network is a basic NN architecture only composed of fully connected layers. It corresponds to the architecture described in section 3.1.1. This architecture is scarcely used in applications since one of the other architectures are usually better adapted for it. It is however still used as a reference when a simple NN is sufficient, for example as test case or proof of concept. The Multi Layer Perceptron architecture is more detailed in ([74]).

3.1.4.2 Deep neural network

A neural network is usually considered as a deep neural network when it has a complex architecture and a significant number of layers. Today, most of state of the art neural network are deep neural networks, for example AlexNet [69], a CNN (Convolutional neural network) that won the ImageNet challenge ILSVRC-2012 in 2012 had 8 hidden layers.

3.1.4.3 Recursive NN

Recursive neural networks were introduced in [107] and are based on the idea of using the same set of weights recursively in the neural network. They are adequate for data that can be arranged in tree structures. They are used in natural language processing (NLP) applications ([139]), and computer vision applications ([140]). In figure 3.5, we show the architecture of a typical recursive NN presented in [40]. The inputs $x_{(1)}$, $x_{(2)}$, $x_{(3)}$, $x_{(4)}$ are typically elements from a sequence data, for example words of a sentence. U, V and W are the weight matrices that are used recursively in the NN, and o is the computed output. In this scheme, the ground truth y is compared to o through the cost function represented as L.

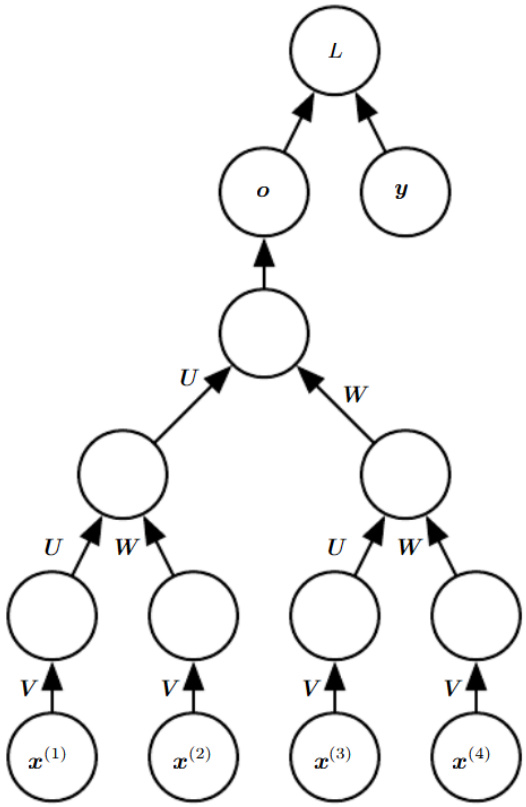


Figure 3.5: Architecture of a typical recursive NN presented in [40].

3.1.4.4 Recurrent NN

Recurrent NN (RNN) are designed to work with sequential data: they take as input a vector representing a sequence. In recurrent NN, neurons of each hidden layers are connected to neurons of the next layer, but also to adjacent neurons of the same layer. This *inner-layer* connection can be seen as an imitation of a short term memory, since at each cycle it will allow a a direct impact from the previous value of a neuron on its current value. Recurrent neural networks are mostly used for speech processing ([11]). In figure 3.6, we show an example of a recurrent NN structure presented in [40]. In this figure, the neural network is presented in a simplified version (on the left) and its equivalent unfold version on the right. The inputs $x_{(t-1)}$, $x_{(t)}$, $x_{(t+1)}$ are typically elements from a sequence data, for example words of a sentence. U, V and W are the weight matrices, $o_{(t-1)}$, $o_{(t)}$, $o_{(t+1)}$ the computed outputs. The ground truths $y_{(t-1)}$, $y_{(t)}$ and $y_{(t+1)}$ are compared to the computed outputs the loss represented as L.

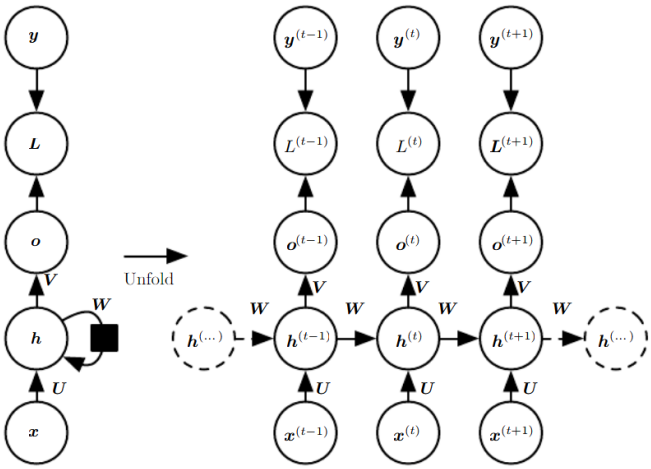


Figure 3.6: Architecture of a typical recurrent NN presented in [40].

3.1.4.5 Convolutional NN

Convolutional neural networks (CNN) are among the most used NN architectures. They are used in computer vision, face recognition and speech analysis. A CNN implements one or several convolutional layers. These layers typically perform a convolutional operation. The convolutional operation uses kernel features that slide step by step on the input image, and computes an activation map by calculating the dot product between the weights of the kernel filters and the value of the input image. This process is illustrated in figure 3.7 where the filter (in red) is set of a portion to the input image (in blue), and computes one value of the activation map (in green). The convolution operation of a CNN makes them adapted for two dimensional data input like images, because it detects features more easily in images by agglomerating the information of adjacent pixels.

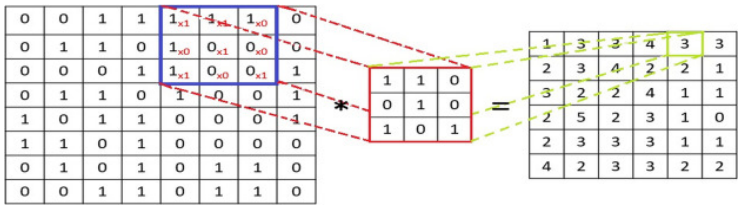


Figure 3.7: Illustration of a a typical convolution operation presented in [138].

A neural network can also be fully convolutional when it only contains convolutional layers.

3.1.5 Neural networks for autonomous vehicles

There exist an extensive research on the applications of neural networks for autonomous vehicles. Several survey papers exist among which:

- [48] that covers a wide range of possible applications related to AV driving tasks applications (perception, mapping, localization, planning and control) but also other AV applications such as behavior and trajectory prediction, simulation, V2X and safety.

- [71] that focuses on applications for the AV control, and sorts them into lateral control applications, longitudinal control applications, and applications that combine both.

- [42] that classifies the research work by applications: perception, localization, path planning, behavior arbitration (that determines the future actions of the vehicle) and end to end neural networks that cover approaches that try to use one neural network for the entire loop from perception to control. It also presents some challenges and approaches regarding the safety of neural networks for AV applications. Finally, it presents the state of available datasets for training, and possible hardware components.

- [66] that presents specifically the applications of neural networks for AV based on reinforcement learning such as controller optimization, path planning and trajectory optimization, motion planning and dynamic path planning, development of driving policies, scenario-based policy learning, prediction of traffic actors such as pedestrian and vehicles, and finally learning of policies that ensure safety and perform risk estimation.

We will now present in more details the main tasks where neural networks could be used for AV applications.

3.1.5.1 Image denoising

Image denoising aims to sharpen pictures taken by a camera sensor in order to facilitate image recognition by a human user or another NN. Several approaches exist to tackle this, but most of them are based on Convolutional Neural Networks (CNN) since their structure is adapted to use images as inputs. In [44], the authors present a review of methods based on the use of CNN to perform image denoising that includes 31 research papers. In [153], the authors use a CNN to perform image denoising and deraining (removal of rain from images) in the context of AV. More precisely, they designed an image deraining and denoising CNN structure. They trained the NN to produce an image without rain as output when receiving the same image with rain as input. They used a mix of synthetic and real work images for the training set, and made an experimental evaluation with real world images.

3.1.5.2 Object detection

Object detection can rely on data provided by camera (images), radar or lidar sensors. Among the image based existing techniques, we can cite [30] that uses a Faster R-CNN (a CNN specialized for object detection) for vehicle detection on the KITTI dataset (an image dataset for autonomous driving application). Another example is presented in [172] where the authors use a structure based on the Faster R-CNN to perform vehicle detection. This model was trained using the imageNet dataset [22] and uses format and evaluation algorithm from the VOC dataset [29]. In [56] more examples of deep learning methods for object detection in autonomous driving are provided.

3.1.5.3 Segmentation

Segmentation is the task that analyses input data (typically images or videos) to identify components (instance segmentation) and classify them with a label (semantic segmentation). More explicitly, it will separate the input data (typically images) into regions and give a label to each of these regions. The main outcome is an image with clear separation of the different elements composing it. Semantic segmentation typically attributes a category to each pixel (human, tree, car, building, etc) while instance segmentation detects and delimits each object of interest. The survey [87] presents an overview of the deep learning methods for segmentation. An example of segmentation by neural networks is presented in [82]. In this work, the neural networks used are fully convolutional networks, with an image as input and a segmentation map as output that is the same size as the input image. Note that the difference between segmentation and detection resides in defining where in the image (or other input data) the detected component is. More explicitly, segmentation will generally focus on finding the exact pixel limitation of objects where object detection will just focus on determining the presence or absence of the object, and optionally determining a bounding box to place around the object.

3.1.5.4 Depth Estimation

Depth estimation is a key task in autonomous driving for building a correct representation of the environment by determining the distance between an object and the system. Image based estimation can be monocular like in [28] where the authors use a network made of two component stacks that each contains several layers including convolutional and max pool layers. The first component is called Global Coarse-Scale Network and its role is to predict the overall depth map structure using a global view of the scene. The second component is called Local Fine-Scale Network whose role is to edit the coarse prediction produced by the other component and align it with local details such as objects and wall edges. The image based depth estimation can also be stereoscopic like in [62]. In this paper, the author focus in particular on pixel disparity estimation, one of the existing method to perform depth estimation. The authors propose a

deep stereo regression architecture called GC-Net (Geometry and Context Network) that has a stereo image (so two images of the same scene from slightly different angles) as input and produces a disparity estimation for each pixel of the input image. Their architecture is composed of 37 various convolutional layers. Depth estimation by neural networks can also be done on Lidar data like in [148] where the authors use a structure called Sparse Convolution network (SparseConvNet).

3.1.5.5 Fusion

Fusion is a task of the AV processes when data from at least two different sensors are used to complement each other to obtain more precise and/or complete data. A typical example is the data fusion of several exteroceptive sensors (lidar, radar, camera, etc) to build a more complete representation of the environment. As an example, [83] created and tested a new NN structure named two-stream fusion fully convolutional network (TSF-FCN). This is composed of two stream of fully convolutional network that meet in a fusion layer, later fed into four more layers. The two input streams are fed respectively with LIDAR point clouds and RGB images. In [166], the authors propose a structure called multi-stage residual fusion network to perform the fusion of LIDAR point clouds and camera image features. In their structure, a two-stream encoder collects the data from the two sources and send it to a module named multi-stage residual input representation fusion module. This module is connected to layers called decoder layers that produce the output (a road detection representation). In [20] the authors present a survey of several methods for image and point cloud fusion based on the use of deep learning in the context of autonomous driving.

3.1.5.6 Human and vehicle behavior prediction

Predicting the behavior of humans and human controlled vehicles is a necessary but difficult task for AV as it allows to anticipate the trajectories of other road users. Two survey paper exist for this task: [91] that focuses on existing deep learning techniques for AV applications, and [129] that also presents other applications and methods. As examples of vehicle behavior prediction, [75] bases its approach on CNN and [2] uses LSTM (Long short-term memory) neural networks, a type of neural network that has feedback connections and can process data formatted in sequences (video, sound, text, ...) just like RNN. In [120], the human and vehicles behavior predictions are made with CNN and RNN.

3.1.5.7 Lateral Control

Several deep learning techniques were studied to achieve the lateral control of an AV. One of the first approach, presented in [109], uses a feedfoward neural network, with images from a camera as input, and the outputs being possible discrete steering actions. Other approaches use CNN as in [92] where a 6-layer CNN is proposed using camera images as inputs and the steering angle as output.

3.1.5.8 Longitudinal Control

Longitudinal control covers functions that manage the acceleration and braking actuators of the vehicle like ACC system or emergency braking system. As an example, [15] uses reinforcement learning to train a feedforward network. It has the vehicle velocity and the pedestrians relative position for the 5 last time steps as inputs, and discrete deceleration actions as output. Another example is provided in [72] where the authors use a reinforcement learning technique to train a recurrent neural network. The system learns autonomous vehicle longitudinal control in an end-to-end fashion mapping ranging sensor readings and host vehicle states to pedal signals in continuous action space, maintaining a safe distance from the lead vehicle. The resulting network is then tested in IPG CarMaker, a car simulation environment for automotive development.

3.1.5.9 Mixed Lateral and Longitudinal Control

To be fully autonomous, the autonomous driving system must perform both lateral and longitudinal control. In [102], the authors use several neural networks specialized to perform the longitudinal and lateral control in different high-level situations (driving down a straight road and stopping at a stop sign, scenarios where another car is immediately ahead of the ego vehicle, a crowded road before and after the intersection, intersection with multiple other vehicles, change of lane). Each neural network is a multilayer perceptrons with a single hidden layer of 32 fully connected neurons, and has two outputs: steering angle rate and acceleration command. The inputs are information about the ego vehicle and its environment such as the ego vehicle states, a set of features for each nearby vehicle, vehicle position and priority in intersection.

3.1.5.10 End to end

End to end approaches aim to create neural networks that directly use the sensors as inputs and the actuators as outputs, thus being end to end components. An example of such application is [98] that has camera images as inputs, and steering and throttle commands as outputs. This work is usually based on CNN. The application introduced in [64] is also an end to end approach. It uses images as inputs and produces steering and vehicle speed commands as outputs. The system is based on several CNN and LSTM neural networks. We can also note that in the literature, the term end-to-end sometimes designates methods where NN are used for several functions at the same time, without completely covering processes from the sensors to the actuators. For example, in [62], the authors call their method *end-to-end learning* because they use raw data directly as input. However, the NN does not command the actuators.

3.1.6 Safety for NN

In this section we will present what current work exists for the safety of neural networks. We will first underline the challenges in using them in safety critical

applications such as autonomous road vehicles, and then give an overview of the approaches currently being developed to improve their safety.

3.1.6.1 A lack of safety in neural networks

Neural networks are already used in several non safety critical applications, but remains excluded from safety critical ones. For example in railways application the Cenelec EN 50128 standard [114] does not recommend their use for railways safety critical applications. This is because of their lack of dependability. In our opinion, the main constraints to the dependability of neural networks are:

- Their complexity. Due to their structure and the wideness of their input domains, neural networks are usually too complex to be fully understandable by humans, or analyzed by computers. This makes the use of automated tools or expert analysis to inspect neural networks very limited.
- The oracle problem. In the context of testing, the oracle problem is defined in [6] as being able to predict the correct results of a test to compare them to those of the system being tested. In other words, is there a reliable manner to decide if outputs produced by a test are correct or not. For neural networks as we often do not know the desired function (due both to its complexity and the vast operational context), it is difficult to generate automatically or not the desired output (also called oracle) for a specific problem. Moreover, as neural networks are only approximations of the desired function, several different values may be close enough to the desired behavior to be deemed correct, but different enough to cause classical diversification techniques (voting, error detection) to be not implementable. This is even more true for decision problems, where several different decisions could lead to a correct solution.
- Their lack of explainability. Because neural networks model their knowledge only through the weights and biases in their structure, their model representation is extremely difficult for humans to understand. Safety experts or analysts are thus unable to study their decision processes and understand their behaviors. This is a well known problem in the machine learning community.
- The size of operational domains. Neural networks are used for applications with vast operational domains (autonomous vehicles, image recognition, ...) where the operational context of the system is very vast, and possibly even infinite. This specificity causes usually incomplete specifications (all the possible situations can not be enumerated) and the impossibility for testing to cover all critical situations.

3.1.6.2 Neural Network robustness

In the dependability community, robustness is defined as the *dependability with respect to external faults, which characterizes a system reaction to a specific class*

of faults [6]. In the artificial intelligence and formal verification communities there exist several definitions of robustness for neural networks in the literature. Most of them encompass the idea of consistency of the outputs for very close input values, the closeness usually being defined through some distance measure. In practice, it is usually a category of distance measures between two points called L_p norm, also called Minkowski distance, that are being used as measure. The general definition of the Minkowski distance between two points is defined in equation 3.8.

$$d(X_1, X_2) = \left(\sum_{i=1}^{n} \|x_{1_i} - x_{2_i}\|^p \right)^{\frac{1}{p}} \tag{3.8}$$

with $X_1 = (x_{1_1}, x_{1_2}, ..., x_{1_n}) \in \mathbb{R}^n, X_2 = (x_{2_1}, x_{2_2}, ..., x_{2_n}) \in \mathbb{R}^n, p \in \mathbb{N}^+$

Among the most used distance measures, we find:

- the Manhattan distance for $p = 1$
- the Euclidian distance for $p = 2$
- the Chebyshev distance for $p = \infty$ the distance is then defined with $\lim_{p\to\infty} \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{1/p}$

Note that in the rest of the thesis, we may also refer to these distances as norms.

We present below examples of robustness definitions from [76]. Note that here, the robustness definition targets NN performing classification tasks, which means that the outputs for inputs that are close to x has to be the same class than for x.

- local robustness: A neural network represented as a function f_{nn}, is $\epsilon -$ *locally robust* at point x, with respect to norm $\|.\|$, if $\forall x_0, \|x - x_0\| \leq \epsilon \Rightarrow f_{nn}(x) = f_{nn}(x_0)$.
- global robustness: A neural network represented as a function f_{nn}, is $\epsilon -$ *globally robust*, with respect to norm $\|.\|$, if $\forall x_0, \forall x, \|x - x_0\| \leq \epsilon \Rightarrow f_{nn}(x) = f_{nn}(x_0)$.

3.1.6.3 The current approaches in trying to make NN safe

There exist several survey papers that list methods under development for NN safety. The authors of [171] present a systematic review of existing methods for testing and verification of neural networks that we are presenting here. Note that there are terminology differences between the dependable system community and the artificial intelligence community.

- *Assuring the robustness of NN*: in [171], robustness is defined as the ability to cope with erroneous inputs, erroneous inputs being adversarial examples (i.e., an input that adds small perturbation intentionally to mislead

classification of an NN), or benign but wrong input data. This definition, when put in perspective of the dependability community taxonomy [6], fits in the *reliability* attributes. Among methods aiming to achieve this robustness for NN, [171] identified 4 research axes:

  - *understanding the characteristics and impacts of adversarial examples.* This axis includes approaches such as [10] that proposes metrics to measure the frequency and severity of adversarial examples for a given NN and devise an algorithm to approximate these metrics. They test their metrics and algorithm on image based experiment on the MNIST [23] and CIFAR-10 [67] datasets. [18] that developed a layerwise regularization method for reducing the network's sensitivity to small perturbations. They test their method on three image based applications with the MNIST [23], CIFAR-10, CIFAR-100 [67] and Street View House Numbers (SVHN) [94] datasets. Note that [10] proposes a fault forecasting strategy since they aim at identifying fault characteristic (here the fault being the existence of adversarial example) while [18] focuses on a fault prevention strategy.
  - *methods to detect adversarial examples.* This topics encompasses approaches like [160] that uses a feature-guided black-box approach to detect adversarial examples on images. The detection is done as a two-player stochastic game and uses the Monte Carlo Tree Search to identify adversarial examples. This research axis corresponds to the dependability mean of fault tolerance, because it aims at detecting faults (in this case adversarial examples) during a system's operation.
  - *methods to mitigate impact of adversarial examples.* This research axis actually corresponds to fault elimination, since the methods listed in [171] aim at decreasing the number of adversarial examples. For example in [99], the authors train NN with defensive distillation to reduce the number of adversarial examples.
  - *increasing robustness of NN through using adversarial examples.* In [134], the authors aim to improve the robustness of NN by using adversarial training. This methods aims to generate adversarial examples and use them as training set data. This approach corresponds to the means of fault prevention.

- *Improving failure resilience of NN*: the methods classified in this category are mostly fault tolerance approaches. In [119], the authors use diversification in training strategies to obtain diverse NN as a fault tolerance approach. One of the diversification methods proposed by the authors is to change the desired function $f_{desired}$ the neural network should learn to another one performing an equivalent service. The initial desired function being a common classification task, they train diversified neural networks that perform slightly different tasks. Note that these classifications are all presented in the context of image classification:

- *Classification with additional class"unknown"*, where they simply add an extra class *unknown*.
- *Class reference classification* that uses as input the element to classify and another element called *class reference* (for which the class is already known), and as output two variables indicating if the two inputs are *similar* or *different*. A variation of this classification method is also proposed. This variation uses a refined class reference element, that would only contain the object corresponding to the class on a white background.
- *Tree classification* that requires all possible classes to be organized in a tree structure (with macro-classes and their sub-classes different as tree nodes, and the elements classes as leaves). The classification of an input element in one of the classes in the leaves is done by going through the tree and determining at each node the following path by a neural network specially trained for this node.
- *Set of binary classifiers* that would rely on several binary classifiers (one neural network per class, trained to predict if the input is from this class or not) that could be used to determine the class of a given input element (by comparing the outputs of all NN), or validate another classifier.
- *two out of three voter* that require 3 additional neural networks in addition to the one performing classification. Three sets of classes have to be created, each set containing two-third of all classes (with an obvious consequence that each third is shared in two of the set of classes). Each of the three neural networks is then trained on one of these sets. With this system, each element should in practice be recognized as a part of the set of classes by exactly two of the three neural networks (due to the shared third of classes). The goal is then to use this three neural networks as a voter, that would decide that a given input belongs to a specific class only if exactly two networks place it in that class.

The method developed in [115] aims to identify miss-classification of images by a NN. This falls into the fault tolerance approach of error detection. Other approaches on hardware faults are also mentioned in the survey.

- *Measuring and ensuring test completeness*: the authors explain that these approaches are about ensuring good coverage of the NN testing. The first set of methods aims to create test sets, and improve NN ability to pass the tests. In [97] the authors provide a scenario description language (SDL) to create driving scenarios and generate test cases with the aim of providing a good diversity of situations for the testing. The second set of methods aims to measure the coverage of a given test set. For example, the tool DeepXplore in [106] proposes the neuron coverage measure which

is defined as the ratio of the number of unique activated neurons for all test inputs of a given test set, and the total number of neurons in the DNN. Note that this measure does not provide guarantees about the coverage of the ODD. Because these approaches cover the testing methods, we can classify them as fault elimination approaches.

- *Assuring safety property of NN:* the approaches in this theme are divided into two axes:
  - *Proving safety properties on NN.* This corresponds to approaches based on formal verification such as ReLuPlex [59] that verifies safety properties on a NN. We will develop further these approaches in section 3.2. They fall into the fault elimination and fault prevention means.
  - *Making sure the system will not violate safety properties with runtime monitoring*: such approach can be assimilated to a safety bag, which is a fault tolerance technique. We did not find implementation of safety bags methods to ensure the dependability of NN.
- *Improving interpretability of NN:* the authors of the survey identified 3 categories for this research axis:
  - *Understanding how a specific decision is made.* This covers approaches like [7] that uses pixel-wise explanations to explain a NN decision, or [21] that provides real time explanation. These approaches aim to design NN which will make their intentions clear to human, which falls into the fault prevention mean.
  - *Facilitating understanding of the internal logic of NN* : This covers approaches like [35] that aims to transform NN into more explainable model: decision trees. Such approach falls again into fault prevention techniques.
  - *Visualizing internal layers of NN to help identifying errors in NN.* This covers approaches like [85]. In this paper, the authors aim to reconstruct the possible input images for given internal values of a neural network (these value corresponding to the neural network state when given one input image). This approach aims to give an idea of what is the representation of a given input image for the neural network. This falls into fault removal techniques.

The authors of another survey [46] enumerate existing approaches on the topics of verification, testing, explainability, and adversarial attack and defense of neural networks.

3.2 Formal verification for neural networks

As presented in section 2.1.3, formal verification aims to guarantee properties on a system or on the model of the system. It appears to us as an efficient method

for the verification of NN as it answers several issues that were mentioned in section 3.1.6.1 due to:

- its validity over entire ranges of the input domain: when verified, a property is valid on the entire domain specified in the conditions of the property. This quality is interesting for applications with an open environment where it is impossible to describe all situations that may occur and testing will thus be limited to a few situations among all the possibilities.
- the absence of oracle : because formal verification is not a method that applies on specific inputs/outputs, it does not require an oracle, but only the definition of safety properties.

Note that we will discuss the limits of formal verification for neural networks in section 4.3 when studying the dependability of neural networks.

3.2.1 Definition of formal verification for NN

Formal verification (FV) for neural networks aims to formally verify that a given NN respects a property. Given a set of conditions on the inputs and outputs of the NN, automated verification guarantees that either all conditions are satisfied or that there exists at least one input that violates at least one condition. To propose the following formal definition, we adapted different concepts presented in [77]. More precisely,we used the three following simplification :

- we did not use the concept of mapping, as it is not necessary when considering a NN as a function f_{nn},
- we did not detail the possible input and output domains but simply used the generic terms of $\mathcal{X}$ and $\mathcal{Y}$,
- we only focused on the property *invariance* defined in the article since it is most of the time what formal verification designates.

Definition 3.2.1 (Formal verification for neural networks)**.** Let $f_{nn} = \mathcal{X} \to \mathcal{Y}$ be a neural network, let $x \in \mathcal{X}$ be a generic input and $y \in \mathcal{Y}$ be a generic output of f_{nn}. We define a *property* P as a couple $P = \langle pre(x), post(y) \rangle$, where the precondition $pre(x)$ and the postcondition $post(y)$ are sorted first order logic (FOL) formulas. Given these definitions, *formal verification* that a network $f_{nn} = \mathcal{X} \to \mathcal{Y}$ respects a property $P = \langle pre(x), post(y) \rangle$ is to prove that the formula 3.9 is valid for the whole input domain $\mathcal{X}$ where "$\Rightarrow$" denotes Boolean implication and "$\wedge$" and "$\vee$" denote Boolean conjunctions.

$$\forall x \in \mathcal{X}. \forall y \in \mathcal{Y}. (pre(x) \wedge y = f_{nn}(x)) \Rightarrow post(y) \tag{3.9}$$

Note that 3.9 can also be written as 3.10, by changing the Boolean implication "$\Rightarrow$" to the equivalent contraposition as "$A \Rightarrow B$" is equivalent to "$\neg A \vee B$".

$$\forall x \in \mathcal{X}. \forall y \in \mathcal{Y}. \left(\neg(pre(x) \wedge y = f_{nn}(x))\right) \vee (post(y)) \tag{3.10}$$

Alternatively, proving that a neural network does not respect a formal property is equivalent to proving that the negation of equation 3.9 is true. The negation is presented in 3.12, and can be obtained with the development presented in 3.11. At the end, the negation of a formal property is equivalent to finding a counter example $x_{ce} \in \mathcal{X}$.

$$\begin{aligned} \neg\left(\forall x \in \mathcal{X}. \forall y \in \mathcal{Y}. \left(\neg(pre(x) \wedge y = f_{nn}(x))\right) \vee (post(y))\right) \\ \exists x_{ce} \in \mathcal{X}, \exists y \in \mathcal{Y}. \neg\left(\neg(pre(x) \wedge y = f_{nn}(x_{ce}))\right) \vee (post(y))\ ) \\ \exists x_{ce} \in \mathcal{X}, \exists y \in \mathcal{Y}. (pre(x_{ce}) \wedge y = f_{nn}(x_{ce})) \wedge \neg(post(y))\ ) \\ \exists x_{ce} \in \mathcal{X}, (pre(x_{ce}) \wedge \neg\, post(f_{nn}(x_{ce})) \end{aligned} \tag{3.11}$$

$$\exists x_{ce} \in \mathcal{X}, (pre(x_{ce}) \wedge \neg\, post(f_{nn}(x_{ce})) \tag{3.12}$$

In the general case, since validity in FOL is semi-decidable, any algorithm for automated verification of neural networks may possibly not finish the verification in bounded time when equation 3.9 is not valid. However, if f_{nn} contains only linear or piece-wise linear activation functions and other linear operations, decidability is not a hurdle anymore, but is still NP-hard at least.

3.2.2 Classifications of formal verification methods for neural networks

Several classifications exist to categorize the numerous FV methods ([149], [80], [46], [77]). We will enumerate here the different criteria used to classify the referenced methods.

- [77] classifies the methods based on which architecture are supported by the tools and methods, and the kind of property the methods can prove (invariance, invertibility, equivalence).
  - The *invariance* is defined as follows: for specific conditions pre and post, asserting an invariance property for a NN function f_{nn} amounts to state: $\forall x. \forall y. (pre(x) \wedge y = f_{nn}(x)) \Rightarrow post(y)$. We can note that this definition corresponds to the definition of formal verification that we give in equation 3.9.
  - The *invertibility* is defined as follows: for specific conditions pre and post, asserting an invertibility property for a NN function f_{nn} amounts to state: $\forall y. \exists x. (post(y) \wedge y = f_{nn}(x)) \Rightarrow pre(x)$.
  - The *equivalence* is defined as follows: given two NN functions f_{nn} and f'_{nn}, for specific conditions pre and post, it is defined as: $\forall x. \forall y. \forall w (pre(x) \wedge y = f_{nn}(x) \wedge post(y) \wedge w = f'_{nn}(x) \wedge post(w)) \Rightarrow y = w$.

This last notion can be understood as the following: if two NN functions f_{nn} and f'_{nn} respect the same formal property for all inputs on X, then the functions on X of the two neural networks are equivalent for this formal property.

- [46] classifies the methods based on the type of proving methods used, the associated precision (exact deterministic, approximated, converging, statistical) and the type of property that can be proven (robustness, reachability, interval, lipschitzian). We develop below what these four properties mean. Note that here we will note $\mathbb{R}^{S_1}$ as the input domain of dimension S_1 of f_{nn} and $\mathbb{R}^{S_2}$ as the output domain of dimension S_2 of f_{nn}.
  - The *robustness property* aims to prove that a given neural network is locally or globally robust.
  - Proving that a NN respects a *reachability property* means proving that a given set $\mathbb{Y}$ is equal to the output reachable set of the NN from a given input region η defined as $reach(f_{nn}, \eta) = \{f_{nn}(x)|x \in \eta\}$ with f_{nn} the function of the NN, and $\eta \subseteq \mathbb{R}^{S_1}$ an input region.
  - The *interval property* aims to show that a given set $\mathbb{Y}$ contains all outputs for a given input region η. More formally we can write it as $\mathbb{Y} \subseteq \{f_{nn}(x)|x \in \eta\}$ with f_{nn} the function of the NN.
  - The *Lipschitzian property* monitors the changes of the output with respect to small changes of the inputs. It is defined in the article as proving that for a given NN represented by a function $f_{nn} : \mathbb{R}^{S_1} \rightarrow \mathbb{R}^{S_2}$, for an input region $\eta \subseteq \mathbb{R}^{S_1}$, for a given L_p norm, for $x_1 \in \eta$ $x_2 \in \eta$ and $d \in \mathbb{R}$, the neural network will respect the property $sup\left(\frac{|f_{nn}(x_1)-f_{nn}(x_2)|}{\|x_1-x_2\|_p}\right) \leq d$.

  The article makes all methods fall into 4 categories: constraint solving, search, global optimization, and overapproximation.

- [80] classifies the methods based on the kinds of proving strategy: *reachability*, *optimization* and the combination of one of these two with search algorithms (which we will respectively call search and reachability, and search and optimization).
  - *Reachability* computes what is the given output set $\mathbb{R}(\mathcal{X}, f_{nn})$ for a given neural network function f_{nn}, and a given input set $\mathcal{X}$. Methods using reachability encompass [38] that computes an over-approximation of the reachable set (we can note here the definition of *reachability* in the classification proposed in [80] differs from the one proposed in [46] since [80] also includes over approximation), [164] that computes the exact reachable set of neural networks with ReLu activation functions and [163] that approximates the reachable set by dividing the input domain into small grid cells and computing approximation for grid cell.

  - *Optimization* will build an optimization problem where the objective is to try to falsifies constraints representing the NN and the property. The article differentiates primal optimization and dual optimization. A primal optimization method will encode the neural network and property as a set of constraints then have for objective to look for a counter example (is none is found, the property is respected). Dual optimization proposes a simpler encoding of the constraints, at the cost of a more complex objective and only returns a Boolean result describing if the property is respected or not without any counter example.

  - *Search and reachability* combines search algorithms to the reachability approaches. According to the categories proposed in the survey, this combination may improve both efficiency and accuracy of the reachability approaches. These methods usually search in the input or the domains covered by the hidden layers for a counter example.

  - *Search and optimization* combines search algorithms to the optimization approaches. According to categories proposed in the survey, these methods will focus on searching in the input domain, or search how does the succession of activation functions impact the property.

- [149] classifies the methods based on the complete and incomplete aspects of the method. The authors divide the complete methods into three groups: the methods based on satisfiability modulo theory (SMT) solving, the mixed integer linear programming (MILP) based formal methods, and the other complete formal methods. Incomplete formal methods are classified into the ones based on abstract interpretation, and other incomplete formal methods. The authors also present formal methods for other machine learning techniques.

  SMT based methods work by encoding a given neural network and the desired safety property as a set of constraints. The objective of the methods is then to try to prove that the set of constraint holds by failing to find a counterexample. Methods using this approach encompass [110], [27], [59], [61], [47].

  MILP based formal methods transform the safety verification problem into a mixed integer linear program. Then they try to find a counter example like the SMT based methods. Methods falling in this category are [16], [32], [14], [26], [147].

Among these, we identified several tools that combine an available mature software that permits experiments with reasonable adaptations and a correct documentation: Marabou [61], MIPVerify [147] and ERAN [38]. We will present Marabou in more detail in section 4.5.2.1.

3.2.3 Existing applications of formal verification for neural networks

In this section we present the applications where the formal verification for NN was used.

- Image recognition application: Image recognition applications for formal verification are usually based on proving robustness properties. An example of robustness property is the local robustness presented in section 3.1.6.2.

  Among the image recognition applications proposed as examples in the literature, we found the following:

  - CIFAR-10: A dataset classifying 60 000 32x32 color images in 10 different classes. This dataset is used in the following papers: [39] (where the property proven by the tool is a robustness property inspired from [106]), [57], [105], [158], [168], [167], [47], [160], [128], [159] (that presents the CLEVER tool), [10], [147] (MIPverify tool).
  - SVHN (Street View House Numbers): a dataset that contains real photos of housenumbers. This dataset contains 10 classes, 1 for each digit. It is used as an example in [88], with a method based on abstract interpretation.
  - Nexar traffic light dataset: In [160], the authors implement an example of traffic light images classification based on the Nexar traffic light challenge [95]. The dataset was composed of over eighteen thousand dashboard camera images publicly available, each image being labeled either green if the traffic light appearing in the image is green, red if the traffic light appearing in the image is red, or null if there is no traffic light appearing in the image. The authors evaluated the winning neural network of the challenge for their article.
  - End to end self driving car case study: In [156], the authors test their method named Neurify on an application for self driving car (DAVE) similar to the DAVE-2 CNN presented by NVIDIA in [12]. This CNN uses images as inputs, and the car steering wheel angle as outputs. The properties verified in this work are related to brightness and contrast perturbation, they use the L_1 and L_∞ norms presented in section 3.1.6.2. The experiment is conducted on 10 random images from the dataset. The properties verified in this application are not safety properties but actually robustness properties since they prove that the output remains the same for similar inputs.
  - ImageNet: ImageNet dataset from the large-scale visual recognition challenge (ILSVRC) [73]. The images are of size 224 x 224 and have three channels. In [47], the authors use this dataset as an example, for the VGG16 neural network [136], which has 16 layers and was used by the VGG team in the ILSVRC-2014 competition. It contains

138,357,544 real-valued parameters and includes convolutional layers, ReLU layers, zero-padding layers, dropout layers, max-pooling layers, fully-connected layers, and a softmax layer. The ImageNet dataset is also used as an example in [162], [47], [128] and [159].

- German Traffic Sign Recognition: In [47], the article presents an image classification application on neural networks trained on the ImageNet Dataset and the German Traffic Sign Recognition Benchmark (GTSRB) which has 43 classes.
- MNIST is a famous dataset of hand written digit images. This dataset is typically used as a benchmark in several machine learning application and research ([96]). The dataset consists of 60 000 gray-scale images with a resolution 28x28 pixels. The digits are white on a black background. The typical task for the neural network is to learn to predict correctly the number on the image. The property to be formally verified is usually related to the correctness of the neural network prediction or its robustness. Scientific articles using this as an application are [27], [93], [81], [16], [14], [39], [79], [161], [57], [157], [105], [158], [168], [167] (that presents the tool RecurJac), [47], [162], [160], [127] (that present the tool DeepGo), [128] (that present the tool DeepTRE), [159] (that present the tool CLEVER), [10], [41], [147] (that present the tool MIPVerify), [113], [156], [32].

  Some papers, while still using the MNIST dataset, propose as examples to prove theoretical properties with no real values rather than robustness or safety properties. For example in [14], the authors developed specific properties to verify on the MNIST dataset on a neural network aiming to differentiate odd and even numbers. Specifically, the image goes through a pre-processing and is turned into a vector of k values used as input for the NN. The NN has one output. This output gives the information that an input belongs to odd class or to the even class. The NN is trained to have an higher score in the output corresponding to the correct class. The property to verify states that there exists an input for which the score assigned to the odd class is greater than the score of the even class plus a large confidence. We can reformulate this property in English as *"Is there an element that is classified as odd with a large confidence?"*. The confidence margin can be adjusted to make the problem harder to pass the formal verification. The author repeat the experiments for several NN architectures (by changing the number of layers and number of neurons per layer). The authors do not provide the number of properties tested, but provide the proportion of verified properties for different tools with varying time limits to perform the verification. Each of the tested tools was able to prove the property in at least 20 percents of all the configurations, but none of the tool verified the property in more than 60 percents of all the configurations.

  The application developed in [57] is also done with properties derived

from the MNSIT dataset. This application called *digit sum task* aims at observing by how much the predicted sum of two MNIST digits used as inputs differs by the real sum (the ground truth). The property verified here is presented as an optimization problem equivalent to the question *"Can an attacker perturb each input image, such that the sum of the digits predicted by the classifier differs from the true sum of those digits by as large an amount as possible?"*. Since the focus is put on how perturbations modify the behavior of the NN, this property is close to a robustness property.

The tool DeepTRE presented in [128] targets 5 different objectives. Each of this objective is illustrated with an image recognition application:

* Convergence Analysis and Global Robustness Evaluation presents an application on the MNIST dataset,
* competitive L0 attacks has an application with the MNIST and CIFAR-10 datasets,
* Local Robustness Evaluation has an application for the ImageNet dataset,
* Guiding the Design of Robust DNN Architectures is done on the MNIST dataset,
* the Test Case Generation for DNN is done on the MNIST and CIFAR-10 datasets.

- Simple function and numerical based examples: Several formal verification examples are just illustrations that the method works, with no concrete application (industrial or other). We present here some of them:
  - analytical functions: In [26], the authors test their Sherlock tool and compare it with the ReLuPlex tool [59]. The comparison is done by proving properties on 4 analytical functions.
  - numerical example: In [164], the authors simply provide a unique numerical example to illustrate their method.
  - two-dimensional point classification: In [47], the authors demonstrate exhaustive verification of their tool on a two-dimensional point classification task. The neural network is trained for classifying points above and below a two-dimensional curve.
- gamified example: Several applications presented in research paper are game versions of real applications. Note that these simulations are very simple applications.
  - agent game : In [58], an agent game is proposed. It implements a NN with 9 sensors, each pointing into a different direction and returning the distances to three different objects (apple, poison or a wall), which amounts to the 27 inputs.

  - deeptraffic : Another application proposed in [58] is deeptraffic, a gamified simulation environment for highway traffic (proposed in [34]).
  - control theory examples: In [81], the tool NSVerify focuses on speeding up the Mixed integer linear programming approach. The applications are well know examples for classic control theory: Inverted pendulum, Acrobot , Pendulum and Mountain car, etc. ([143], [89], [36], [84]).

- Applications close to real life application, with relevant safety properties:

  - ACAS X :

    Several papers implement the example provided in [59] and [60] for the tool ReLuPlex. This application implements a NN that performs an Airborne Collision Avoidance System X (ACAS X). The function that the NN aims to model (that we will designate is as f_{ACASX}) aims to provide horizontal maneuver advisories based on input information about incoming aircrafts. The outputs of f_{ACASX} are the advisory maneuvers, which are represented in the NN as five outputs corresponding to the following names: Clear-of-Conflict, weak left, weak right, strong left and strong right. For a given input, the value of each of these outputs is a score, and the output with the lowest score is taken as the horizontal maneuver advice to give to the pilot. The ACAS function that the NN aims to model has 6 inputs:

    * ρ: Distance from ownship to intruder
    * θ: Angle to intruder relative to ownship heading direction
    * ψ: Heading angle of intruder relative to ownship heading direction
    * υ_{own}: Speed of ownship
    * υ_{int}: Speed of intruder
    * τ : Time until loss of vertical separation
    * a_{prev}: Previous advisory

    The figure 3.8 illustrates 5 of the 7 inputs.

    In their experiment, the authors trained 45 neural networks. These neural networks correspond to a decomposition of 2 of the inputs variables (τ and a_{prev}). The discrete values of τ are the following: 0, 1, 5, 10, 20, 40, 60, 80, 100. The discrete values of a_{prev} correspond to the five possible advisory maneuvers presented earlier. All the neural networks had the same architecture, but no detail is given about the training. The NN are fully connected, use ReLU activation functions, and have 6 hidden layers with a total of 300 ReLU nodes each. In total, 10 safety properties are used in the article, but not all of the 45 NN are verified on them. The authors use $N_{x,y}$ to denote the network trained for the $x-th$ value of a_{prev} and $y-th$ value of

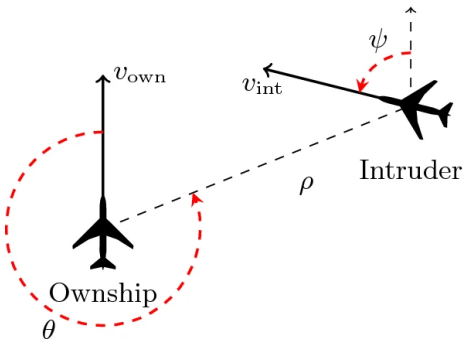


Figure 3.8: Illustration of the ReLuplex NN inputs (from [59]

Property 5	
Description	If the intruder is near and approaching from the left, the network advises "strong right".
Tested on	$N_{1,1}$
Input constraints	$250 \leq \rho \leq 400$, $0.2 \leq \theta \leq 0.4$, $-3.141592 \leq \psi \leq -3.141592 + 0.005$, $100 \leq v_{own} \leq 400$, $0 \leq v_{int} \leq 400$
Desired output property	the score for strong right is the minimal score.

Table 3.1: ReLuPlex property 5

τ. For this notation, the a_{prev} and τ values are ordered as follows: a_{prev}=[Clear-of-Conflict, weak left, weak right, strong left, strong right] and τ =[0, 1, 5, 10, 20, 40, 60, 80, 100]. For example, $N_{2,3}$ is the network trained for the case where $a_{prev} = weak\ left$ and $\tau = 5$. We present in this chapter the property 5 (in table 3.1) and 9 (in table 3.2). Both of these properties describe a critical situation where the airplane must perform the recommended maneuver or risk to crash. The other properties are presented in the annex (section 10) in tables 10.1, 10.2, 10.4, 3.1, 10.6, 10.7, 10.8 and 10.10.

The results of formal verification can either be UNSAT, when the property is verified, SAT when a counter example is found, or TIMEOUT when the answer is not found under a certain delay (unspecified). The authors also worked on proving robustness properties. For this, they selected 5 arbitrary reference points in the input domain. For each reference point, the objective is to verify that all inputs with a range have the same output as the reference point. Here the range is defined using the L_∞. The authors use the five different following L_∞ values to perform the formal verification: 0.1, 0.075, 0.05, 0.025 and 0.01. At the end, three of the five reference points were proven robust with all distance ranges, and two reference points were only

Property 9	
Description	Even if the previous advisory was "weak right", the presence of a nearby intruder will cause the network to output a "strong left" advisory instead.
Tested on	$N_{3,3}$
Input constraints	$2000 \leq \rho \leq 7000$, $-0.4 \leq \theta \leq -0.14$, $-3.141592 \leq \psi \leq -3.141592 + 0.01$, $100 \leq v_{own} \leq 150$, $0 \leq v_{int} \leq 150$
Desired output property	the score for "strong left" is minimal.

Table 3.2: ReLuPlex property 9

proven robust with the distance range of 0.025 and 0.01. In [157], the authors also use the ACAS X example as a case study for their method named ReLuVal. They worked on proving the original properties described in the paper [59], and five new properties that we describe in the annex chapter (tables 10.11, 10.12, 10.13, 10.14 and 10.15). The ACAS X example is also used as an application in [14] and for the Marabou tool [79].

– trajectories:

In [27], the Planet tool is tested on a collision detection case study. The NN used has tuples (x, y, s, d, c_1, c_2) describing trajectories as inputs, and must classify them into whether they represent a colliding or non-colliding case. In each tuples, the variables have the following definitions:

* the x and y components represent the relative distances of the vehicles in their workspace in the X- and Y-dimensions
* s is the speed of the second vehicle
* d is the starting direction of the second vehicle
* c_1 and c_2 are the rotation speed values of the two vehicles

The vehicles are represented as circles. The goal of the NN is to classify the tuples as collision or non collision cases. Figure 3.9 provides a graphic representation of a collision case and a non-collision case.

The neural network has 40 linear nodes in the first hidden layer, followed by a layer of MaxPool nodes, each having 4 input edges, followed by a layer of 19 ReLu nodes, and 2 ReLu nodes for the output layer. To train the neural network, 3000 tuples are generated (with equal number of collision and non collision examples). The property verified by the tool is the following: for a given reference input (x, y, s, d, c_1, c_2), all the inputs within a safety margin are have the same classification as the reference input. The safety margin is defined as $(x \pm \epsilon, y \pm \epsilon, s \pm \epsilon, d \pm \epsilon, c_1 \pm \epsilon, c_2 \pm \epsilon)$ with $\epsilon > 0$. This

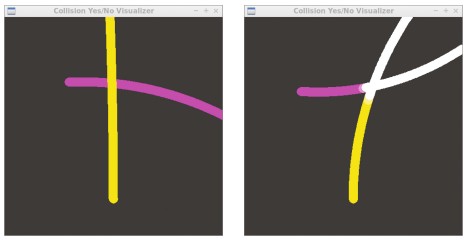


Figure 3.9: Illustration collisions simulation presented in [27]. The left image represent a situation where the two points do not collide. The left image present a collision between the two points.

property was verified on the first 100 examples of the training set. We can note here that the property verified in this example is actually a local robustness property. The authors also present an application on the MNIST dataset where the use formal verification to study local robustness. The collision detection example is also used as an application in [14].

– Robotic arm: In [163], the authors present an application on a robotic arm shown in figure 3.10. The robotic arm possible positions are divided into three zones: Normal Working Zone and Buffering Zone that the robotic arm is authorized to reach, and the Forbidden Zone that specifies the region that the robotic arm will never operate in due to physical constraints or safety considerations in design.

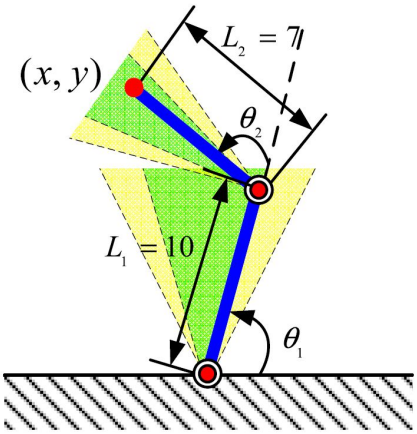


Figure 3.10: Illustration of the robotic arm presented in [163] with the normal working zone (in green) and the buffering zone (in yellow).

The verification method aims at computing an over approximation of the output space for a given input set. It encompasses the following

steps: discretizing the input space into a set of small areas call *lattices*, and compute the outputs of the small areas through the neural network, each computed area being called a reachtube. The union of all computed reachtubes is an over approximation of the output space. With this methods, we can check if a given input set will have all its output within a safe zone corresponding to the safety property we wish to verify. An important parameter of this method described in the article is δ, the radius of the lattices. The smaller δ is, the more precise the estimation will be, but the total procedure will be also computationally more costly. The neural network (a multi layer perception) has two input variables which are the angles (θ_1, θ_2) of the two robotic arm joints and two outputs (x, y) describing the future position of the arm. The formal verification aims to prove that for any initial position in the Normal Working Zone or Buffering Zone, the future position (computed by the neural network) will respect the following property: $(-14 \leq x \leq 3) \wedge (1 \leq y \leq 17)$. The property is verified with a discretization using lattices with a radius of $\delta = 0.02$.

These three applications (ACAS X, vehicle trajectories and robotic arm) are to our knowledge the applications of formal verification for neural network closest an actual implementation in an industrial context. We consider that this puts our own application presented in chapter 6 among the most advanced example of formal verification for neural networks.

- Other applications: We group here applications that are not close to industrial domains, but in other potential deep learning application domains.
  - github classifier stability: This application is presented in [57]. The predictions task consists of predicting whether a given github repository will reach more than 40 commits within 250 days given data observed until a certain day in this 250 days period. The two possible prediction values are then *Yes, the 40 commits will be reached within the 250 days period* and *No, the 40 commits will NOT be reached within the 250 days period.* The neural network used has two inputs: a variable that has a value between 0 and 1 indicating the percentage corresponding to number of days left before the end of the 250 day period, and a second variable that indicates the progress of commits towards the total of 40. The output provided by the neural network will then evolve over the 250 days, and should be more precise as we approach the end of the period. On this example, the formal verification (combined with dynamic programming) will provide a bound of how many time the NN prediction will switch between the two possible predictions over the 250 days period. We can note that here, the verification is about a succession of output values, which does not match our definition of formal verification presented in 3.2.1.
  - Android application malware detection: In [156], the author test their method named Neurify on the DERBIN neural networks for malware

detection presented in [4]. The neural network has as input a set of features describing the app (permissions, hardware components used, network addresses, etc...). The task of the NN is to classify an app as malware or not based on this list of features. The safety property being checked in this case is whether the addition of an app permissions to the list of feature, without changing any functionality, will not cause the NN to misclassify malware apps as benign. We can note that this property is not a safety property.

– Text classification: In [81], one of the example used is the text classification example with the Reuters-21578 dataset [117]. The dataset contains a collection of documents with news articles. The original corpus has 10,369 documents and a vocabulary of 29,930 words. The information provided by the author about the structure of the neural network used is as follows: the neural network used in the article is composed of a binary input layer followed by a hidden layer with a ReLU activation function followed by the output layer. The structure of the network is thus rather shallow, but contains a large number of input and hidden neurons.

We can note that most of the examples and experiments of formal verification for neural networks are for now more illustrative examples than practical ones with a real interest for the industry.

3.2.4 International Verification of Neural Networks Competition (VNN-COMP)

Since 2020, an international competition is organized for verification of neural network. The competition is at the moment at its third edition. The competition aims to gather researchers working on the tools and methods for the verification for neural networks. It has the advantage of offering a benchmark to compare verification tools. The competition is affiliated to the Conference on Computer Aided Verification (CAV) research conference [19].

We will describe the 2022 edition of this competition [1], but most of the competition format and participants remain similar to the 2021 edition, which provided a well documented report [9]. The participants can propose both tools and benchmarks. All the tools are tested on all the benchmarks. Each benchmark consists in proving that a neural network respect (or not) a property. The NN is provided in the ONNX format [8], an open format to encode neural network. The result of the verification can be *sat* (when a counter example is found and provided), *unsat* (when no counter example is found and the property is verified), *timeout* (when a time limit is reached), *error* and *unknown*. The property must respect the VNNLIB format presented in [43] which describes properties as a first order formula, a common form for safety properties. Tool participating to the competition earn 10 points when they manage to prove that a property holds (unsat result), or prove that it does not by giving a counter example (sat result). If a tool is the fastest to correctly provide one of these

two results, it receives two bonus points. The second fastest receives one bonus point. A tool looses 100 points if it returns one of the previous results (sat or unsat) and it is actually false. No points are awarded for timeout, unknown and error results. The tool are given equal access to computing power.

There were twelves different benchmarks used for the competition, with neural network sizes going from 64 neurons to 13.6 millions of neurons. Diverse neural network architectures were used from fully connected with ReLu activation functions, to CNN, and more complex ones. Since the report of the 2022 edition is not published at the moment, it is complex to gather an overview of all the benchmarks, but they are available on github[1]. For all of the benchmarks, there was at least one tool that managed to prove all of the proposed properties.

Eleven tools took part in the 2022 competition. In this competition, the tool Marabou scored 7th out of the 11 tools. The tool that is ranked first is α,β-CROWN which is a combination of works presented in [169], [165], [155] and [170]. We can note that based on the description provided on the Github repository of the tool[2], the tool could be adequate for the formal verification conducted in our experiments presented in chapters 6 and 7 because it can support our NN architecture and property format. However the sources and documentation are available only since June 2021, when most of the work presented in this thesis was already done. The tool that ranked second is the Multi-Neuron Guided Branch-and-Bound (MN-BaB) tool presented in [31]. This tool also seems to be adequate for our experiment, but was released in April 2022, also after our experiment.

3.3 Conclusion

In this chapter, we presented the concepts and current works related to neural networks and their formal verification. Especially, we can see that:

- *Safety of neural network is necessary but still insufficient.* We saw that, due to the ability of neural networks to perform complex tasks, there are a lot of potential applications where neural networks could be used, including in safety critical applications. The use of neural networks would require however the adequate trust, especially for applications where dependability is required. However, there is today no appropriate method that fully ensures this dependability for neural networks. This absence of dependability method is due to the complexity of their structure that makes human analysis impossible, the oracle problem that prevent the use of test generation for application where the function to model is unknown (which is often the case for NN), their lack of explainability, and the size of the operational domain.

- *Safety of NN for AV is currently being developed.* We saw that there are a lot of applications in Autonomous Vehicle for neural network. As we

[1] https://github.com/ChristopherBrix/vnncomp2022_benchmarks
[2] https://github.com/Verified-Intelligence/alpha-beta-CROWN

saw, a lot of the research on neural network safety takes AV application as examples, which shows the interest of the research communities for this topics. We also saw that the standard ISO PAS 21448 [125] being developed for autonomous vehicle has an annex dedicated to guidelines for the use of neural networks, which is a first step in having a development process for neural network that ensures the required dependability. However, none of the techniques currently proposed in the literature fully resolve their lack of dependability yet.

- *Formal Verification for neural network seems an interesting solution for neural networks dependability.* We saw that formal verification answers several of the dependability problems of neural networks: it covers entire input domains and make the test unnecessary. It does not require the use of an oracle (because the correct behavior is not defined by a correct function to follow, but by the limits imposed though the safety property to verify). However, it still has several limitations that we will discuss in the next chapter.

- *Formal verification for neural network is not yet an industrialized activity.* The current applications presented in research are still far from industrial applications.

Chapter 4

Dependability of neural networks and limits of formal verification

In the previous chapter, we presented what neural networks are, how they work and introduced their architecture. Then we gave a state of the art on their applications and the dependability methods used to improve trust in their correct behavior, particularly focusing on formal verification, which presents significant advantages to validate them. In this chapter, we begin to present our work by focusing on better characterizing the components of neural networks, which faults and errors can affect them, and the problems faced by formal verification. We start with the characterization of neural networks from a software development perspective by providing a decomposition of the neural network into three components: data, structure and processes. This decomposition is not limited to the operational neural network but is applicable from the development phase. We then present what are faults, errors and failures of neural networks, including the concepts of errors that are presented in the machine learning community. We then explore several limits that still prevent the use of formal verification as a solution to ensure the safety of neural networks for industrial and safety critical applications, particularly the risk of validating formal properties on a wrong model, which was never mentioned to our knowledge in the literature at the moment.

4.1 Characterization of neural networks

In this section we will describe the NN characterization in the machine learning domain and we propose a characterization from the software development perspective. This characterization is based on a decomposition into three key components of the neural network development: structure, data, and processes.

4.1.1 Characterization from the machine learning community

In the ML community, a NN is typically characterized as a function approximator. The data collection and training remain important research topics, but for the definition of a NN the focus is put on the structure of neural networks when the training is complete. The neural network is then completely characterized by the description of the neurons after training: the number of neurons, their connections, their activation functions, and their biases and weights. Note that this characterization does not allow to completely describe a NN before training. The machine learning community also generally refers to a *model* (or *machine learning model*) as the methods used to represent an approximation (neural network, linear regression, Bayesian network, etc.). The learning algorithm, is the set of processes used to define the model parameters. In the case of neural networks, the parameters are the weights and biases of each neuron.

4.1.2 Characterization from a software development point of view

We propose in this section a characterization of neural networks based on three software development components: structure, data and processes.

Structure We define a neural network's *structure* as the enumeration of its *inputs and outputs* variables, the *description of its connections*, and its *activation functions*. The type of architecture (convoluted, feedforward, etc.) is not necessary as it can be deduced from the description of its connections but is useful to mention as it implies specific behavior of the NN. Note that the weights and biases values are not part of this structure as they are only completely determined after the training.

Data The training data is a dataset of (x, y) examples used for learning, (x, y) being a couple of respectively one input and its corresponding output. As presented in section 3.1.3, the data is usually divided as followed: *training set*, *validation set*, *test set*. The training set contains the training data used for learning. The validation set is commonly used at the end of one training phase to evaluate the learning mechanism performances before possibly modifying the learning conditions (modifying the structure or the processes of the NN for example) and doing another training with these changes to improve the performances. The test set is typically used after the last training, to evaluate the definitive performances of the learning mechanism. Note that recurrent NN examples are formed from a temporal succession of vectors x_t rather than an unique vector x.

Processes Processes are algorithms or practices that define some necessary activities in NN development. To our knowledge, this includes *learning algorithm*, *regularization*, *data preparation*, *weights and biases initialization*, *hyper*

parameter tuning, and performance verification on the *test set*. The learning algorithm is the algorithm in charge of modifying the weights and biases values during learning by taking into account the errors between the outputs of the NN and the desired outputs. The learning algorithm will typically go over all the training data several times. Each repetition is called an epoch. The training data can be split into subsets called batches. The batches correspond to the number of elements that the learning algorithm will go through between each update of the model parameters. The number of epochs, the number of batches and their sizes are parameters of the learning algorithm. This algorithm typically uses back-propagation, an optimization technique, and a cost function that as we presented in section 3.1.2.3 provides a measure that compares the output computed by the NN for a given input with its actual output (ground truth). Regularization is an optional set of methods that modify the learning algorithm to take into account known problems in neural networks training. Examples of regularization techniques include L_1 or L_2 regularization to diminish the overfitting problem, or inputs normalization to ensure that inputs with high values have no more impact that inputs with small values on the NN's outputs. Data preparation includes all operations done on training data before the training starts. For example, shuffle is a common data preparation technique. Weights and biases initialization corresponds to the determination of the weights and biases initial values, i.e. before the training starts. The tuning of hyper parameters is the process of modifying the parameters of the processes (for example: number of batches, type of regularization, pace of the learning algorithm). This step is typically done by comparing different neural network implementations that were trained with different hyper parameters on the validation data. Finally, a performance verification of the neural network implementation is done on the test set. This verification is not meant to make further changes on the neural network implementation, since the objective is to evaluate the neural network definitive behavior on data that was not used to train the NN.

Note that to completely determine a NN from these components, one needs to give exactly the training set in the order that it will be used for training in every intended epochs, and the exact initial weight and biases values. In practice, the training data is usually shuffled before each epoch and the weights and biases are generated randomly or from an existing NN similar to the intended function. In these cases, several NN can be generated from the same components, due to the variance error described in section 4.2.2. It is however much easier to give this incomplete information than to detail the exact order used for the training data and this is how we will describe our NN in the rest of this work.

4.2 Fault, error and failure of neural networks

In this section, we will discuss the faults, errors and failures that can affect neural networks. First, we propose several faults according to the components presented in the previous section. Second, we present errors from the machine

learning community, which focuses on errors on the structure, processes and data selection. We add other errors resulting from incorrect data. Third, we present failures that can result from these errors.

4.2.1 Faults

Considering the decomposition of a neural network in three components (structure, processes and data) as seen in section 4.1.2, faults can affect any of these subsets.

Faults related to the structure of neural networks are incorrect choices or programming mistakes in designing the network *structure*. We will list here possible faults with concrete examples:

- *Incorrect choices or programming mistakes in the selection of the inputs:* these usually consist in relevant inputs that are not taken into account as input of the NN, which will inevitably cause irreducible errors (as we will see in 4.2.2) in the approximation of the desired function as their influence will not be considered. Irrelevant inputs present in the NN should a priori not impact the behavior of the NN, but may cause added complexity in both its structure and the learning processes, possibly causing bias errors.

- *Incorrect choices or programming mistakes in the design of the connections and activation functions:* wrong choices in a NN's connections or activation functions may cause the NN to not achieve an approximation as precise as otherwise possible, causing bias errors. They encompass an incorrect type of NN (feed-forward instead of recurrent for example), incorrect connections (number of layers and connections between them) or incorrect activation functions.

Faults related to the data include erroneous data (when some data in the training set is incorrect), incorrect distribution (when the possible situations considered in the training are not distributed appropriately for the neural network to learn the desired function), and lack of training data (when the training data for some or all situation is too small for the neural network to learn the desired function). Note that lack of data in only some part of the training set can be also considered as an incorrect distribution. A typical fault in the latter case is the over or under representation of specific situations in the training set. For example, the training data may contain not enough examples of situations where an exo vehicle is performing an overtaking that is not following the road regulation.

Faults related to the processes are wrong choices or programming mistakes about the parameters of the learning algorithm (cost function, optimizer and number of batches, etc.), in the parameters of the training data preparation (shuffling, training data distribution, etc.), in the neural network initialization (choice of initial value of weights and biases), and in the tuning of hyper parameters (wrong pace, too large number of batches, etc.).

4.2.2 Errors

In the machine learning community, the error is defined as the difference between the computed output of the trained system and the real output (ground truth). This community distinguishes several types of errors, including irreducible errors, reducible errors, estimation variance, estimation bias and model bias [53]. In equations 4.1 and 4.2 we present the two error decompositions that we will detail in this section.

$$Error = Reducible\ Error + Irreducible\ Error \tag{4.1}$$

$$Reducible\ Error = Estimation\ Variance + Estimation\ Bias + Model\ Bias \tag{4.2}$$

The objective of machine learning is to build a function that maps input variables to output variables in a way that best matches some acquired data. In this process, errors appear due to choices on the algorithm and the machine learning model used to implement the function. This machine learning model can be a neural network, but also a linear or polynomial regression, a Bayesian network or others.

We detail in the following list the different types of errors. We will consider the following definitions from [53]: let Y be the output to be predicted by the learning system, and $X = (X_1, X_2, X_3, ..., X_n)$ the vector of n inputs of the system (also called predictors). f is then a function that maps the inputs variables to the output variable as $Y = f(X) + \epsilon$, where ϵ is a random error term that represents the irreducible error. The objective of machine learning is then finding the best estimation f^* of f. For later notation, we will write as $\hat{f}$ the actual estimation of f in the trained system.

- *irreducible errors* (also called inherent errors or Bayes errors): they are errors that are independent from the input vector X. Thus they can not be reduced for a particular function by learning better but are due to unconsidered factors or a stochastic behavior of the desired function.
- *reducible errors*: as shown in equation 4.1 and 4.2 the reducible error can further be decomposed into model bias, estimation bias and estimation variance.
  - The model bias covers errors that are due to the limits of the chosen complete machine learning model to accurately estimate f. In other words, it means that f can not be expressed in the chosen complete machine learning model.
  - The estimation bias covers errors due to some restriction of the complete machine learning model space to the chosen restricted machine learning model. These restrictions can be for example the suppression of part of the input vector to reduce the complexity of f, or

some regularization technique (like the normalization of the inputs and outputs) used to diminish the variance errors. If the machine learning model space has not been restricted, the estimation bias is 0.

– The variance errors, or estimation variance, cover errors due to how much $\hat{f}$ could change if it had been trained in another way or with another training set. It characterizes how far $\hat{f}$ is from f^*. For neural networks, a high variance means that training the same structure twice with different training data (or possibly the same training data but just randomly shuffling it before each epoch) would lead to significantly different functions.

We present in figure 4.1 a scheme that illustrates the different errors described here.

For neural networks, the model bias is caused solely by structure faults. The estimation bias is caused partly by structure faults (for example when restricting the input vectors) and processes faults (for example when implementing regularization on the dataset). Finally, the variance errors are caused by processes faults, but also by incorrect distribution (both in size and distribution) of the data. Regarding the faults on data (incorrect values, missing data, etc...), they are not included in this representation. The quality of learning data is however a well studied topic in the machine learning community. Note that the overfitting and underfitting errors presented in section 3.1.2.1 are part of the variance errors.

4.2.3 Failures

As a reminder, a system's failure is the deviation from its correct service. For a neural network, that means that its output deviates from the desired function. In particular for classification tasks, a failure would be to assign the wrong class to an input. For regression tasks, a failure would occur when the NN's function for the considered input deviates significantly from the desired function's output. All this is consistent with the errors previously detailed, as they are all value errors from the desired function. Note however that the desired function is usually not known in machine learning, as otherwise it would be much simpler to implement it in an imperative language. The system's specifications may also be incomplete considering the vast execution context of most NN's applications.

4.3 Formal verification limits and problems

Formal verification for neural network is a promising method to achieve the trust required for safety critical applications. But despite the numerous advantages presented in section 3.2, it still suffers from problems when we try to implement it on a real application. In this section we will present these limits.

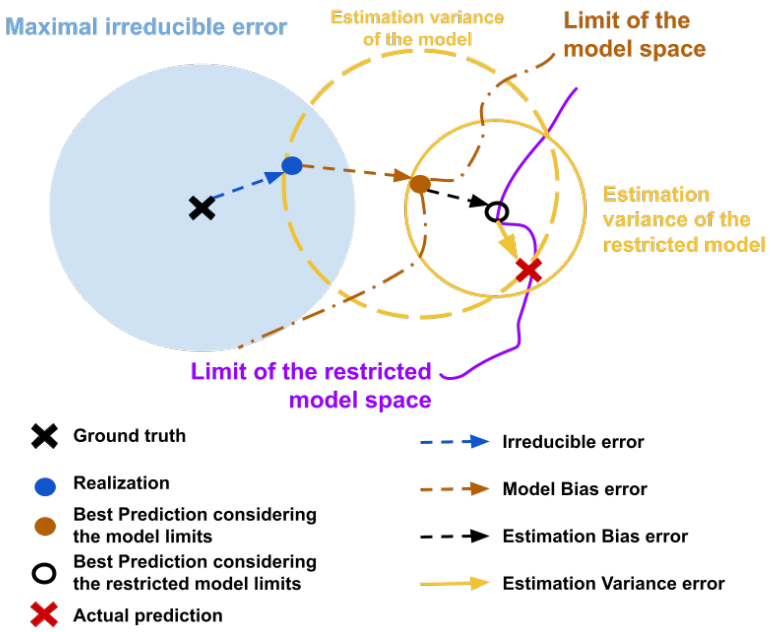


Figure 4.1: Illustration of the 4 errors impacting a machine learning model prediction: irreducible error, model bias error, estimation bias error, estimation variance error. The blue circle represent the possible values of the irreducible error ϵ. This first error, due to a stochastic behavior of the function or some unknown parameters, is added to the ground truth (the black cross) and creates the actual possible realization of the desired function. Then the model bias is added due to the model limits (for example, the number of nodes and layers for the neural network). The brown doted line represents the approximation of the realization in all possible choices of the model, and the best prediction is the result of the best model that could be chosen. The next error is the estimation bias, which leads to a restricted version of the model space (purple doted line). Finally, the estimation variance is caused by the variation between different trainings. The maximum estimation variance is displayed by the yellow circle and the actual prediction, represented as a red cross, could be anywhere in it. Note that the estimation variance of the non restricted model is bigger than the variance of the restricted model, as regularization aim to diminish this variance error at the cost of an added estimation bias.

Limits due to inputs and outputs To be formally verified, most formal properties must express restrictions and constraints on the NN's inputs and outputs. In real applications, this may lead to add outputs and inputs to the NN as some of the state variables expressed in the desired properties may not originally be part of the function's inputs and outputs.

For example, a NN that is learning to regulate a car speed will have as output the acceleration of the car. But acceleration itself is not sufficient to express safety and functional properties, as we might want to check that the system will respect other physical constraints, like not exceeding a speed limit or keeping safe distances with other vehicles. A solution is then to add the output $futureSpeed$ to the NN (and train it with appropriate data), so we can also check safety constraints on this variable. Note that we could derive the future speed from the acceleration output, but formal verification tools are not yet able to take such calculus into consideration when formally validating properties.

Regarding missing inputs, a possible example could be the absence of input information about the environment. For example, we may need to verify a stronger safety property when the weather conditions are bad. This may then require to add variables describing the weather or road adherence as input to the neural network.

Limits due to formal verification tools Since FV for NN is still an emerging domain, few tools are available. They are research tools, with diverse level of maturity, using different technologies and having different constraints. Most of them pose limitations on the structure of the NN that they can process (for example, the Marabou tool [61] at the moment of our experiment only supported fully connected and convolutional NN with piecewise-linear activation functions). In [14], the authors use only piecewise-Linear neural networks. The input domain can then be decomposed into a set of polyhedra C_i such that $C = U_i C_i$, and the restriction of the neural network function to C_i is a linear function for each i. We can then note that this excludes activation functions such as sigmoid or tanh and only allows neural networks with piecewise-linear functions like ReLu.

Another limit is the expression of the safety properties. Safety properties have to be written with specific rules, that may reduce the possible safety properties we wish to verify. For example, the tool Marabou only permits expressions that use multivariate polynomials that have a degree no higher than one. More explicitly, the properties must be either an equality as the equation 4.4 or an inequality like the equation 4.3. For both equations 4.4 and 4.3 we note $c_0, ..., c_{n+m}$ floats variables, and n and m being respectively the number of inputs and outputs of the neural network.

$$\begin{aligned} input_1 * c_1 + input_2 * c_2 + ... + input_n * c_n \\ +output_1 * c_{n+1} + output_2 * c_{n+2} + ... + output_m * c_{n+m} + c_0 < 0 \end{aligned} \tag{4.3}$$

$$\begin{aligned} input_1 * c_1 + input_2 * c_2 + ... + input_n * c_n \\ +output_1 * c_{n+1} + output_2 * c_{n+2} + ... + output_m * c_{n+m} + c_0 = 0 \end{aligned} \tag{4.4}$$

Another example of limits on the property format is presented in [57], where the authors assume that the property to verify is expressed in a finite set of linear

constraints on the values of final layer, considering that the possible inputs are in a nominal input domain.

Depending on the tool, the answer provided will not necessarily be complete. For example, in the survey presented in [149], methods based on abstract interpretation like AI2 [38] or DeepPoly [137] are classified as incomplete and might give false positive counter examples.

Finally, this technology is limited by the NN size, although significant progresses have been made about this limit in recent years. Depending on the NN task (image based application, control command, etc...) and the type of property to verify (robustness limited to several points and their surrounding like local robustness, or property that cover the entire input domain) the maximum size of a NN to be verified can range from a few thousands to a few millions of neurons in most tools.

The need for precision to achieve formal verification Because formal verification properties usually apply on entire ranges of the input/output domain, the NN must be close enough to the intended function $f_{desired}(x)+\epsilon$ to not contain any counter examples of these properties as one unique counter example would prevent a property from passing the verification. As neural networks are function approximations and subjects to the errors presented in section 4.2.2, and as their learning can be trapped in local minima, numerous tweaking and trainings may bee needed to obtain neural networks satisfying this precision, as we will see in chapter 6.

4.4 Consequences of an erroneous model

In this section, we will first define models in traditional systems, in the machine learning community, and what we will consider as a neural network model in this thesis. We will then present what are the consequences of having an erroneous neural network model on formal verification and illustrate them with two theoretical examples.

4.4.1 Different models in different communities

In this section, we compare the definition of models in the dependability community and the machine learning community. We will then present the definition of a neural network model that we will use in this thesis.

4.4.1.1 Models in dependability community

In classical systems, a model is usually a simplified representation of the system, describing its behavior either in physical equations or through a formal language (like a statechart diagram or the B language). It usually includes state's variables that together describe the state of the system and possibly of its environment. This model is developed by the system's engineers and in the

case of formal verification it is supposed to describe the system's behavior in a scheme suitable for the chosen formal verification tool. Formal properties usually describe constraints on some of the system's state variables under specific conditions. The formal verification is then done through some mathematical proof on the system's model. Obviously, if the model does not correctly detail the system's behavior, the formal verification will have been done on a description of the system differing from the real one and thus would guarantee nothing on the real system. Correctness of the model is then fundamental for the formal verification process. To our experience in the industry's practices, and particularly in railway applications, trust in the system's model is achieved through code reviews and functional tests. However, such methods are very hard (or even impossible) to use with a neural network because of its lack of explainability and the fact that both the model and the decision mechanism of neural networks are merged into the biases and weights of each neuron.

4.4.1.2 Machine learning models

As seen in section 4.2.2, a model in the machine learning community is both the type of methods used to approximate the function (neural network, linear regression, Bayesian network, etc.) and the specific chosen form or structure of this method (the structure of a NN, the structure of a Bayesian network, simple or multiple linear regression, etc.). Moreover, a machine learning model can be further restricted through the selection of part of the input vector or regularization techniques. All approximators $\hat{f}$ obtained from the same specific restricted model will have the same bias error, which is as previously stated the error between the best possible approximator f^* using the chosen model (which can be restricted or not) and the intended function $f_{desired}$.

4.4.1.3 Neural networks model

In this thesis, we define by model of a neural network the outcome of two modeling tasks that the neural network must perform:

- The modeling of the system and environment behaviors and variables relations (which we will call system model)
- The modeling of the decision to take (that is obtainment of a correct value of y for a specific x, which we will call decision model)

In practice, the system model and the decision model are intrinsically merged in the values of the weights and the biases of the neural network and are impossible to differentiate. Also note that the system model probably does not exist explicitly, but only implicitly in some parts of how the neural network achieves a specific output y for a given input x.

In the context of autonomous driving, the system model includes real world physics, other users behavior or the ego car physical capabilities while the decision model decides how much to accelerate and which direction to go.

4.4.2 Formal verification of neural networks with erroneous model

The model of a NN is intrinsically part of the weights and biases of its neurons, and is very hard or even impossible to understand by humans. In the same way as in classical systems, errors in the model of the NN can result in a satisfied formal verification that might not hold in the real world. But whereas the model in a classical system can be reviewed by experts, the model in the NN is as stated extremely hard to verify by experts, and testing is often limited due to the open environment that often requires AI techniques to implement the function. To sum up our argument, formal verification for NN can not give complete trust in a NN because it is partly based on a model learned by the NN, which is very hard to validate. From the ML community perspective, one may argue that it is excepted that the formal verification can not be trusted if the neural network did not learn correctly. However, there are currently no method to ensure that the neural network's learning is correct enough to trust the formal verification (if such method existed, formal verification would even be useless since we could trust the neural network). Therefore, our point that the formal verification can not be trusted is still an issue to consider for applications such as safety critical applications where the trust in its result is needed.

As an example, let us say that we want to model a NN that handles a car's decision at a crossroad. Let us consider more specifically the situation presented in figure 4.2, where the ego vehicle (yellow car) wish to cross the intersection by going left. A neural network is in charge of the car controller, and must take a decision based on a representation of the environment sent by a perception layer (aka, a representation of the interaction and all users on it).

Let us consider that in the training set used to train the neural network, for all situations with an exo car (represented in green in figures 4.3 and 4.4) coming from the right road of the intersection, the exo car always turns right like in figure 4.4. In other words, there are no data showing the exo car turning left or going ahead (like in figure 4.3).

The NN will then learn a representation of the environment that considers that cars coming from the right road always turn right.

On such a neural network, we may then be able to formally verify the following property *"Turning left can never result in collision"* but the property obviously does not hold in real life as the exo car on the right (green car) can also go straight or turn left. In the next section we develop a similar theoretical example on a real neural network. Note that a real application example will be presented in chapters 6 and 7.

4.5 Theoretical example of formal verification on an erroneous model

In this section, we will detail a theoretical example of formal verification on a neural network with an erroneous model. The experience in decomposed in

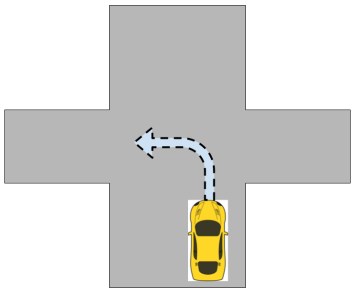


Figure 4.2: Illustrative example of car decision at an intersection

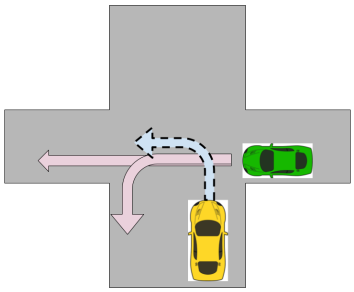


Figure 4.3: Illustrative example of car decision at an intersection, situations where a car on the right is turning left or going ahead

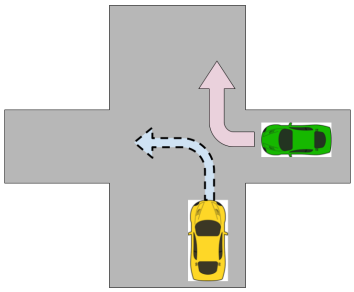


Figure 4.4: Illustrative example of car decision at an intersection, situation where a car on the right is turning right

several parts:

- definition of the function to be learned

- training of the nominal neural network
- formal verification of a property on the nominal neural network
- definition of the faults to inject to obtain the erroneous neural network
- training of the erroneous neural network
- formal verification of the property on the erroneous neural network

At the end, we obtained an erroneous neural network that formally verifies a property that does not hold on the desired function. This experiment was presented in [152].

4.5.1 Experimental neural network and injected faults

We introduce here the desired function that the neural network should learn in our experiment. We present then the training of our experiment (structure, processes and data). We also detail the property that we will formally verify on the neural network. Finally, we present the faults that we use to create an erroneous neural network.

4.5.1.1 Desired function and property to verify

Let us consider the following desired function $F_{exp}(x)$, representing the ground truth that the neural network must model:

$$F_{exp}(x_0, x_1) = \begin{cases} x_0^2 \; if \; x_1 = 0 \\ x_0 \; if \; x_1 = 1 \end{cases} \tag{4.5}$$

with $x_0 \in [-10; 10]$ and $x_1 \in \{0, 1\}$.

Note that we chose this function because for $x_0 \in [-10; 0]$, its behavior strongly differs depending on the value of x_1: namely, it is negative if $x_1 = 1$ and positive if $x_1 = 0$. We are thus interested in the following property on a function $g(x)$:

$$\begin{aligned} \forall x_0, \forall x_1, (x_0 \in [-10; 10], x_1 \in \{0, 1\}) \\ \Rightarrow (g(x_0, x_1) \in [0; +\infty[) \end{aligned} \tag{4.6}$$

In other words, $g(x)$ is always positive on the specified input domain, which corresponds to the input domain of $F_{exp}(x)$. However, this property is obviously false for $F_{exp}(x)$ as, for any negative x_0, $F_{exp}(x)$ is negative when $x_1 = 1$.

4.5.1.2 Experimental neural networks

We will here explain how we developed our neural networks by detailing the three components presented in section 4.1.2.

- Structure: All neural networks in this chapter use the same structure. It is a feedforward architecture with 4 layers: the input layer, 2 hidden layers and the output layer, respectively of sizes 2, 15, 15, 1. The input layer corresponds to the two input variables x_0 and x_1. The size of the 2 hidden layers were chosen as a compromise between a correct approximation of the function $F_{exp}(x)$ and a short time for the learning and formal verification processes. The last layer has a size of 1 corresponding to the unique output of the function $F_{exp}(x)$. All activation functions in our neural networks are *ReLu* functions.

- Training data: The training set used for the nominal network has 10000 examples. We generated the x_0 values of these examples randomly using a uniform law between -12 and 12. We decided to have a training set with values outside of the operational domain of the $F_{exp}(x)$ function in order to have a better approximation on the extreme values of the x_0 domain (-10 and 10). Note that this can be considered a good practice, although we have not studied it enough to pretend that it is necessary in our case. 50% of the 10000 examples were given a x_1 value equal to 0, with the other 50% were given a x_1 value equal to 1. For each example, we computed the y value using the $F_{exp}(x)$ function. The test set was generated using the same algorithm as the training set for the nominal situation (50% of examples with $x_1 = 1$), but with x_0 values between -10 and 10. We did not use a validation set for these neural networks.

- Processes: The training was done using the pytorch[1] tool described in [100]. To perform the training, we first shuffled our training data, then we divided our training set in 100 batches, and performed 200 epochs. The cost function used was the MSE (Mean Squared Error), and the optimization algorithm was the Adam algorithm (proposed by [65]) implemented in pytorch. The weights and biases initialization was done automatically by pytorch following this pattern: the values of each weights and biases were generated using a random uniform law $U(-\sqrt{k}, \sqrt{k})$, with $k = \frac{1}{number of input features} = \frac{1}{2}$. The default implementation of the Adam algorithm used by pytorch does not use regularization techniques.

Note that different trainings with these parameters would produce different neural networks as the weights and biases are generated randomly and each batch of data is shuffled. To evaluate the quality of the approximation done by such neural networks, we implemented a nominal neural network called nNN, and its mathematical function $F_{nNN}(x)$. We plotted its outputs in Figure 4.5 and 4.6. We can see that the approximation is very close to the ground truth. On our test set, the mean of the absolute values of its errors (the difference between its output and $F_{exp}(x)$) is 0.254.

[1] version 1.6.0 for GPU

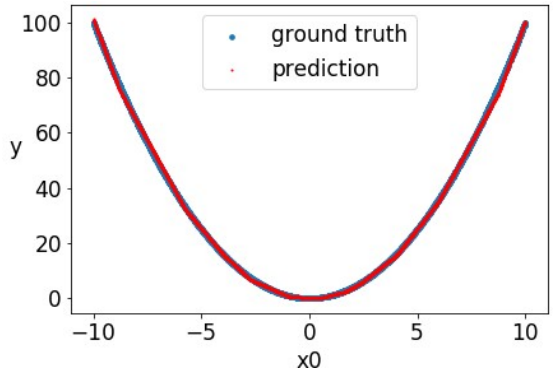


Figure 4.5: Ground truth and output of a nominal neural network when $x_1 = 0$

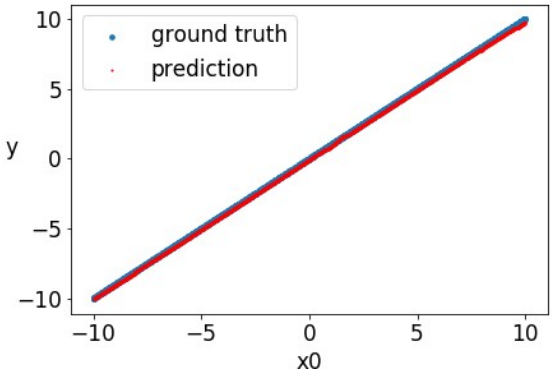


Figure 4.6: Ground truth and output of a nominal neural network when $x_1 = 1$

4.5.1.3 Erroneous neural networks

The erroneous neural networks were generated using the same structure and processes as the nominal neural network. However, we injected faults in the training data by modifying the quantity of examples where $x_1 = 1$ (erroneous distribution and absence of situation). We present hereafter the results of two erroneous neural networks, called eNN_0 and $eNN_{0.05}$ respectively obtained with 0% and 0.05% of examples where $x_1 = 1$. We name the mathematical functions of these neural networks F_{eNN_0} and $F_{eNN_{0.05}}$.

Figures 4.9 and 4.10 show the outputs of eNN_0 compared to the ground truth, and figures 4.7 and 4.8 show the same for $eNN_{0.05}$. We can see in both cases that they approximate correctly $F_{exp}(x)$ when $x_1 = 0$. However, when $x_1 = 1$ eNN_0 presents the same outputs as when $x_1 = 0$. This is understandable as we did not give it examples to learn from when $x_1 = 1$. Meanwhile, $eNN_{0.05}$ has a better but still wrong approximation of $F_{exp}(x_0, 1)$ impacted by the very few examples where $x_1 = 1$ in the training set.

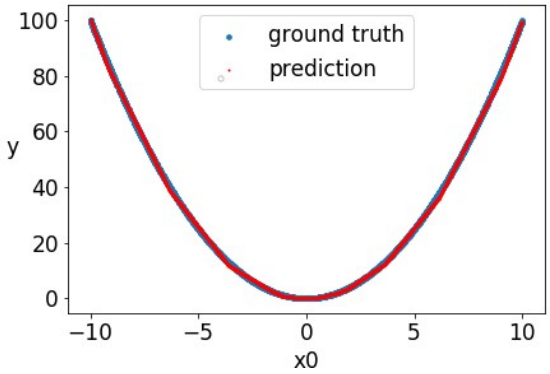


Figure 4.7: Ground truth and output of an erroneous neural network for $x_1 = 0$, with 0.05 percent of training data with $x_1 = 1$

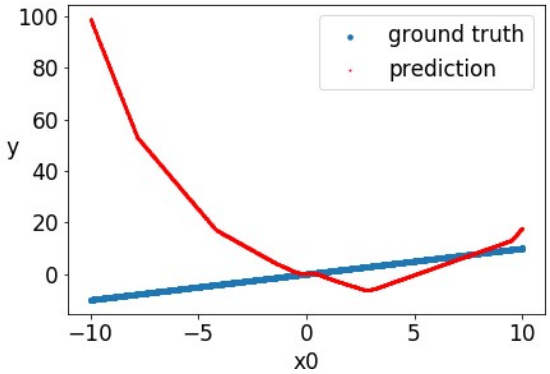


Figure 4.8: Ground truth and output of an erroneous neural network for $x_1 = 1$, with 0.05 percent of training data with $x_1 = 1$

4.5.2 Formal verification

In this section, we present the formal verification of the property 4.6 on the three developed neural networks: nNN, eNN_0 and $eNN_{0.05}$.

4.5.2.1 Choice of the Marabou tool

Among the variety of formal verification techniques presented in the classifications of section 3.2, Marabou [61], MIPVerify [147] and ERAN [39] were potential candidates for our verification. We chose to use Marabou[2] for its maturity, including a better documentation and support, and the convenience that it offers when working on regression tasks like in our experiment.

The Marabou tool is based in the ReLuPlex tool presented in [59]. The Marabou tool can handle fully connected neural networks (that we will use in

[2]Version released on github the 3/02/2020 (commit fc985bf).

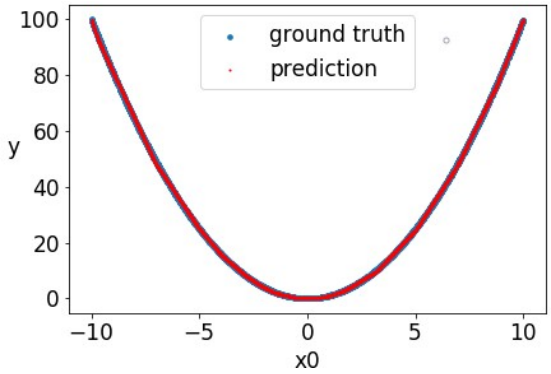


Figure 4.9: Ground truth and output of an erroneous neural network for $x_1 = 0$, with 0 percent of training data with $x_1 = 1$

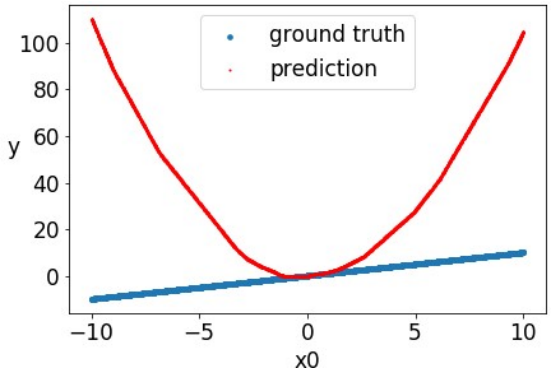


Figure 4.10: Ground truth and output of an erroneous neural network for $x_1 = 1$, with 0 percent of training data with $x_1 = 1$

our experiment) and convolutional neural networks (more used in image applications). It only processes piecewise-linear activation functions (like ReLu for example).

Marabou can only verify properties that can be expressed as linear constraints on the inputs and outputs variables. ore explicitly, as presented in section 4.3, the constraints have to follow the format of equation 4.3 or 4.4. It can process properties provided as textual format or as a protocol buffer file containing a TensorFlow model and verifies neural networks encoded under the NNet format [55].

The Marabou tool works by encoding the neural network and the safety property as a set of linear constraints, and tries to find a counter example that does not respect the set of constraints, aka a counter example that would prove that the neural network does not respect the safety property. The verification procedure of the Marabou tool is said to be sound and complete. Sound means

that, when a counter example is provided by Marabou, it is actually a counter example. Complete means that when there exist a counter example, it will always be found. The output of the verification procedure is either a SAT result, when a counter example is found (the counter example is then provided), or an UNSAT result when no counter example is found (which means that the neural network respects the safety property). The procedure that looks for a counter example is based on the simplex algorithm [17]. The simplex based solver is included in the Marabou tool contrary to the ReLuPlex tool which uses an external solver.

4.5.2.2 Adaptation of the properties to fit Marabou

To perform the verification with Marabou, we had to put property (4.6) under a form for which Marabou could find a counterexample (property (4.8)). To do that, we go through the following steps:

We first obtain the negation of (4.6) which is the property (4.7):

$$\exists x_0, \exists x_1, (x_0 \in [-10; 10], x_1 \in \{0, 1\}) \wedge \neg (g(x_0, x_1) \in [0; +\infty[) \tag{4.7}$$

Note that (4.7) can be reformulated as (4.8), which is very close to what we will express in the Marabou tool:

$$\exists x_0, \exists x_1, (x_0 \in [-10; 10], x_1 \in \{0, 1\}) \wedge (g(x_0, x_1) \in]-\infty; 0[) \tag{4.8}$$

Proving that (4.8) is unsatisfied will be equivalent to proving that (4.6) is satisfied, and the former is a task that Marabou can do. Thus when asking Marabou to find an example that verifies (4.8), getting an UNSAT answer will mean that the property (4.6) is true, while getting a SAT answer will mean that the property (4.6) is not satisfied.

However, we had to limit the $g(x_0, x_1) \in]-\infty; 0[$ constraint to $g(x_0, x_1) \in]-10000; -0.3[$ for two practical reasons. The left side of the interval is changed because formal verification tools usually do not have representations for infinite values. We thus chose instead a negative number (-10000) with a significant value compared to the theoretical lower bound of our desired function (-10). This is an example of the limits of FV due to validation tools as discussed in section 4.3. The right side of the interval is changed because our developed neural networks (nNN, eNN_0 and $eNN_{0.05}$) are only approximations of the desired function, and will probably still have negative outputs even for the square function around the point $(x_0, x_1) = (0, 0)$. This is an example of limits of FV due to the imprecision of NN as discussed in section 4.3. Note that the absolute value of this chosen number (0.3) is very close to the mean absolute values of nNN's errors (0.254). The property that we actually verify can be then written as (4.9) and what we will ask Marabou as (4.10).

$$\forall x_0, \forall x_1, (x_0 \in [-10; 10], x_1 \in \{0; 1\}) \Rightarrow (g(x_0, x_1) \in]-\infty; -10000[\cup]-0.3; \infty[) \tag{4.9}$$

$$\exists x_0, \exists x_1, (x_0 \in [-10; 10], x_1 \in \{0, 1\}) \wedge (g(x_0, x_1) \in [-10000; -0.3]) \quad (4.10)$$

Finally, to set x_1 to either 0 or 1 during the verification, we split (4.10) into two properties: one for $x_1 = 0$ and another $x_1 = 1$.

Here is the code that we use to verify property (4.10) for $x_1 = 1$ with Marabou:

```
1 # Set input bounds
2 network.setLowerBound(X0,-10.0)
3 network.setUpperBound(X0, 10.0)
4 network.setLowerBound(X1, 1)
5 network.setUpperBound(X1, 1)
6 # Set output bounds
7 network.setLowerBound(Y, -10000)
8 network.setUpperBound(Y, -0.3)
```

As previously said, an UNSAT answer would mean that there is no (x_0, x_1, y) example that satisfies the given constraints.

Note that when analyzing the results, we will also mention property (4.11) which is similar to (4.6) but with the constraint $g(x_0, x_1) \in]-6.5; +\infty[$. This property is obviously less constraining than (4.6).

In a similar manner, we actually transform (4.11) into (4.12) and split the latter into two properties: one for $x_1 = 0$ and another $x_1 = 1$.

$$\forall x_0, \forall x_1, (x_0 \in [-10; 10], x_1 \in \{0; 1\}) \Rightarrow (g(x_0, x_1) \in [-6.5; \infty[) \quad (4.11)$$

$$\exists x_0, \exists x_1, (x_0 \in [-10; 10], x_1 \in \{0, 1\}) \wedge (g(x_0, x_1) \in [-10000; -6.5]) \quad (4.12)$$

The code that we use to verify property (4.12) for $x_1 = 1$ is the same as before, but with this change on line 8:

```
8 network.setUpperBound(Y, -6.5)
```

An UNSAT answer would mean that there exists no (x_0, x_1, y) example that satisfies the given constraints.

4.5.2.3 Results of the formal verification

In this section, we present the results of the verification done with Marabou on each property.

4.5.2.3.1 Results for nNN

For nNN the results are presented in table 4.1.

Due to the two examples found, we can see that nNN does not respect the properties (4.9) and (4.11) for $x_1 = 1$ (respectively equivalent to (4.10) and (4.12)), and thus nNN has negative outputs (strictly, outputs below -0.3). These results are consistent with figures 4.5 and 4.6.

property	result
(4.10) for $x_1 = 0$	UNSAT
(4.10) for $x_1 = 1$	SAT input 0 = -8.585003190816256 input 1 = 1.0 output 0 = -8.698448463114229
(4.12) for $x_1 = 0$	UNSAT
(4.12) for $x_1 = 1$	SAT input 0 = -9.44927891732047 input 1 = 1.0 output 0 = -9.524702312172995

Table 4.1: Formal verification for nNN

4.5.2.3.2 Results for eNN_0

For eNN_0 trained with an absence of situation fault for $x_1 = 1$, Marabou gives the results presented in table 4.2

property	result
(4.10) for $x_1 = 0$	UNSAT
(4.10) for $x_1 = 1$	UNSAT
(4.12) for $x_1 = 0$	UNSAT
(4.12) for $x_1 = 1$	UNSAT

Table 4.2: Formal verification for eNN_0

We can see that in every cases, no example was found. We can conclude that properties (4.11) and (4.9) hold on this NN. This confirms what we proposed in section 4.4.2: a property has been formally verified on a neural network that has learned an incorrect model, while the property is incorrect in the desired function (the ground truth of our experiment). Thus, formal verification can not hold if we do not have arguments for justifiably trusting how the neural network modeled the problem that the desired function represents. These results are consistent with figures 4.9 and 4.10.

4.5.2.3.3 Results for $eNN_{0.05}$

For $eNN_{0.05}$ trained with an erroneous situation representativity fault, lesser than the previous absence of situation fault, Marabou gives the results presented in table 4.3.

We can see that although property (4.9) does not hold, property (4.11) does. Which means that although we have negative numbers in the output, we have no negative numbers below -6.5, a property that once again is incorrect in the reality. These results are consistent with figures 4.7 and 4.8.

property	result
(4.10) for $x_1 = 0$	UNSAT
(4.10) for $x_1 = 1$	SAT input 0 = 0.6011074525473619 input 1 = 1.0 output 0 = -0.3
(4.12) for $x_1 = 0$	UNSAT
(4.12) for $x_1 = 1$	UNSAT

Table 4.3: Formal verification for $eNN_{0.05}$

4.6 Conclusion

In this chapter, we presented a characterization of dependability threats in neural networks based on a software development approach. We also made the comparison with the characterization proposed in the ML community. We presented the faults, errors and failures of a neural network. More importantly, we presented the limits of the use of formal verification to ensure the safety of a neural network. Among these limits, the erroneous model we found stands out because it threatens the trust that formal verification can bring. Whereas the other limits (limits due to the outputs and inputs, limits due to the formal verification tool, and limits due to the precision of the NN to achieve) are just usage limitations that could be resolved in the future. We presented theoretical examples illustrating the erroneous model limitation. Note that the simple example of formal verification on a neural network with an erroneous model published in [152]. In chapter 7, we will see a an application of this threat on an autonomous vehicle function.

Chapter 5

Online neural network model verification

In this chapter, we will present a proposal to answer the erroneous model of neural networks. This solution is based on a detection mechanism that performs online verification of the model correctness. We also present a recovery mechanism based on diversification of the neural network used to perform the function. This chapter will focus on a generic description of the two components, and a validation application using the autonomous car simulator from the previous chapter will be presented in chapter 8.

5.1 Detection mechanism

As detailed in section 4.4.2, having a neural network with an erroneous model can lead to the formal verification of properties that do not hold in real life, as the model may stray too far from the reality. Our proposed approach aims to estimate online if a neural network model is erroneous while in operation. When this occurs, the formal verification of the safety property can not be trusted in the current situation and we have to use some recovery mechanism as the neural network could currently cause a catastrophic failure. In this section, we first present a generic description of our detection mechanism, and then list the limits we identified to our approach.

5.1.1 Description

The online evaluation is done on two levels: a model evaluation that would allow to detect discrepancies in the neural network's model as soon as possible but which might not detect all errors (for various reasons that we will discuss in section 5.1.2) and a safety property evaluation that is always able to detect when the safety properties fail. To perform the model level evaluation, we

suppose that part of the modeling function done by the neural network can be modeled by experts (for example by dynamics or physics equations). This will happen in most control systems as their environment obeys to physical laws that are well known and well studied. Note however that the decision process of the neural network is supposed to be unknown as otherwise we could just use imperative programming to implement. More concretely, we suppose that at least one output of the neural network can be diversified by experts. The model evaluation relies on the observation of potential difference between the outputs computed by the neural network, and the diversified ones generated by expert knowledge. We illustrate this in figure 5.1 with a neural network that has 4 inputs variables and 2 outputs variables. Among these outputs, one can be computed directly using diversified expert knowledge, while the first one is presumably too complex to resolve in imperative programming. By comparing the neural network output with the diversified one, we are able to detect when the neural network model fails as its output value diverges significantly from the diversified one. To be effective, a threshold must be found to trigger the error detection where the difference between a correct output and the diversified one stays below the threshold for all the situations in the operational domain, while the difference between an erroneous output leading to failure and the diversified ones goes over the threshold.

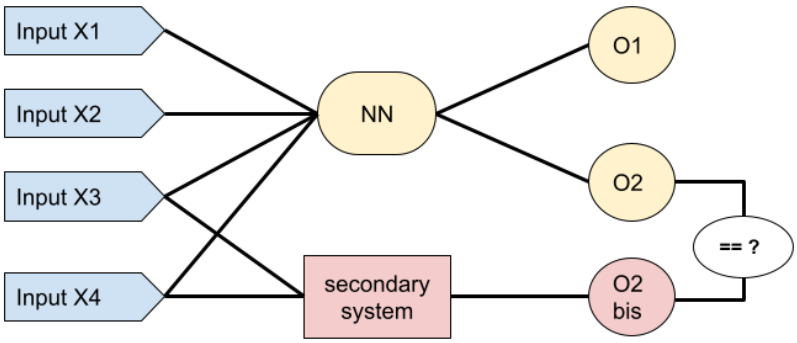


Figure 5.1: Example illustrating the diversified model on a neural network.

Secondary system The secondary system has at least one input in common with the NN and at least one output in common with the NN. Because we use it to detect erroneous system model as presented in section 4.4.2, it is important that the secondary system represents some of the system and environment behaviors and variables relations. This can be achieved through different ways:

- equations describing environment relations. These kind of equations would be known equations that describe the environment and the system with which the NN interact. Because it interacts with it, the inner working of this environment and system have to be a part of the NN system model. Using equation describing part of this inner working would permit to check that part of the system model is correct. However, there are no guarantees that the other outputs will be correct: it simply allows us to detect situations where the neural network model has been learned incorrectly on the compared outputs (and thus possibly on all outputs).

- the safety properties verified on the NN. In this approach, we rely on the idea that if a safety property is violated, then the formal verification should not have succeeded and the neural network model is erroneous. In this case, the architecture of the detection is slightly different (as presented in figure 5.2), since it does not rely on comparison of the NN output with another diversified value. Note that in this case the detection happens at the same time as the error is activated, which gives less time to bring back the system in a safe state and rely on a possible safety margin to recover before the failure. This solution however guarantees that the detection will always happen when the neural network commands cause the safety property to be violated. This architecture is similar to the concept of safety bag which has been used in several safety critical applications such as railways [145], robotics [111] and autonomous vehicles [101].

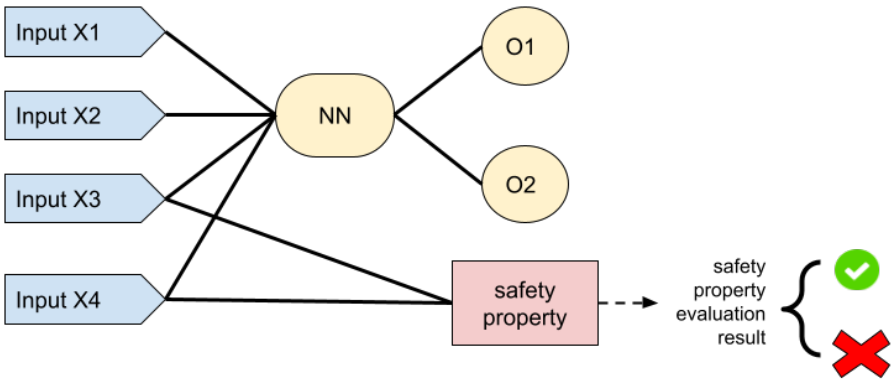


Figure 5.2: Example illustrating the diversified model on a neural network, using a safety property. Here the safety property used as a detection mechanism uses the input variables *input* $X3$ and *input* $X4$.

5.1.2 Limits of the detection mechanism

This detection mechanism still has some limits. We present in this section its common limits, then the limits of the model verification used alone and

finally the limits of the safety property verification used alone. Note that a complementary use of the model verification and safety verification is a good way to mitigate or even resolve their specific limitations.

5.1.2.1 Common limits

We identified one common limit which is the determination of a threshold:

- *determination of the threshold:* as usual, thresholding is a key aspect of error detection, since a threshold too low can lead to false positives (detection of an error when none is present), and a too high value may lead to false negatives (not detecting an existing error). Note that the thresholding problem is present for any situation where a NN is performing a regression task since the output will be some real value. However in most cases we will not encounter this problem for classification tasks.

5.1.2.2 Model evaluation limit

Regarding the limits that only apply to the model evaluation, we identified the three following ones:

- *false negative:* as expert knowledge will only be able to model part of the problem faced by the neural network, we will never be able to have diversified comparison for all its outputs. There is thus no guarantee when using model evaluation that a non-diversified output is correct or not: we can only detect situations where the diversified outputs are incorrect. Therefore, we might have situations where the detection does not signal an error when there is actually one.

- *false positive:* we can detect an error on one diversified output, while the overall behavior of the neural network will stay correct (for example if the erroneous output is not a command). Note however that this still means that the neural network has learned the current situation wrongly and that it is justified to not trust it anymore.

- *difficulty to find an adequate secondary system:* It may not always be possible for experts to find a secondary system that describes part of the modeling done by the NN, especially because NN are commonly used in applications where we do not have traditional systems or equations to perform the function. Even when some expert knowledge exist, it might not (or only partially) be directly linked to the outputs of the NN.

5.1.2.3 Safety property limit

For the safety property limits, we identify one limit presented here:

- *limited time for anticipation:* as previously stated, detecting an error when a safety propriety is violated (unlike the model evaluation which can detect

errors as soon as they appear) will give less time for the system to recover before a failure might occur. Increasing the safety margin on the safety property would mitigate this problem, but it would come at the cost of possible false positive and some loss of the system's availability.

5.2 Recovery mechanism

After an error has been detected, the recovery mechanism aims to stop a failure from occurring by placing the system in a safe state exempt of the detected fault. Keeping the system safe can be achieved through different degraded modes or by switching to a redundant functional system. In this section we propose a recovery mechanism that can be implemented to recover from an error detected by the error detection mechanism presented in section 5.1. We first present a generic description of our recovery mechanism, then identify its current limits.

5.2.1 Description

Recovery mechanisms rely on redundancy and diversification. In our case, this diversification can be achieved through several complementary approaches that can be combined. All these approaches follow the forward recovery strategy: the erroneous neural network will be replaced by another diversified component which supposedly will not suffer from the same faults.

- A first approach relies on redundant neural networks. It requires several NN performing the same function, but diversified enough to decrease the probability of having the same erroneous model errors in several NN. The passive neural networks need to be checked for model errors in the same way than the active NN to know whether or not they can be used as a recovery component. Note that since NN are not much dependable, we advise to use at least another approach than NN to be used ultimately in the case where no NN are found exempt of errors. We propose below several ways in which we can force diversification of the NN. They were heavily inspired from the work described in [119] that was presented in section 3.1.6.3.
  - the first possible diversification is to train several neural networks through the same process, and rely on the variance error presented in section 4.2.2. This variance makes the training non repeatable and is caused by several factors such as the random initialization of weights and biases, the shuffling of the datasets, or changing the number of epochs.
  - A second possibility relies on the bias error. This can typical be done by changing the regularization techniques, or not using them at all.
  - A third possibility is to rely on the model error, typically by changing the structure of the NN. Changes in the *structure* can be made

by changing the number of layers, neurons in the NN, but also by changing the architecture type of NN (convolutional, recurrent, ...).

  - A fourth possibility, diversification through changes in the *data* can be done simply by using different datasets but also by increasing or decreasing the size of the datasets. Another possibility for diversification through the *data* is to use different data distribution (for example with more or less elements in specific parts of the input domain).
  - A fifth diversification is to learn a different function than the desired function $f_{desired}$, as proposed in [118]. The new function could be simply using different inputs and outputs, for example adding previously unused state variables as new inputs of the NN, or in a more complex way reformulating the problem differently, using completely different state variables or representations.

- Another approach could be to use a different AI technology than neural networks. Note that the proposed error detection and recovery mechanisms should still work in this case.

- As another approach for diversification, a recovery mechanism of a simpler function as a degraded mode could be implemented by a simpler NN (easier to validate or at least significantly diversified from the original function), other AI methods, or even in some cases by imperative programming if the degraded function is simple enough. In this last case, such redundancy should be verifiable via known and established dependability methods. For example, in the case of autonomous vehicles, a recovery system could perform a minimal risk maneuver or an emergency brake. Such method could ensure the safety in case of failure of the detection or recovery mechanism, at the acceptable cost of not performing the autonomous driving function the NN is able to perform.

5.2.2 Limits

For the recovery mechanism using redundant NN, we identified the following limits:

- *Relying on neural networks that may also have an erroneous model:* when the detection mechanism detects an error in one NN, we have no guarantee that the other NN does not also have the same error. The probability that several NN have the same error can be significant if they learned from the same dataset and have the same structure. This phenomenon can be limited by the diversification mentioned in section 5.2.1.

- *Limits of number of NN used for the redundancy*: if all NN used for the recovery mechanism have errors detected by the detection mechanism, then we face a situation where no solution is provided by our system to ensure the safety of the system. We strongly suggest to have as an ultimate

recovery a simple imperative function that can be more easily validated by classic dependability methods.

5.3 Conclusion

In this chapter, we presented a generic approach for detection and recovery of the erroneous model problem in NN. In particular, the proposed mechanisms aim to tackle the issue of neural networks which passed formal verification while having a model that represents incorrectly the real world and thus can cause a critical failure of the system. For the detection mechanism, we propose two solutions:

- a model evaluation through equations describing environment's relations. This system relies on expert knowledge to diversify some of the NN outputs and compare the values with the NN actual outputs. When the difference reaches a threshold, the model is considered erroneous and the formal verification can then not be trusted. This solution is limited by the possible presence of false positive and false negative, and the difficulty to find equations.
- online verification of the safety property. This approach consists in verifying online if the safety property verified by the formal verification holds while the system is in operation. The main advantage of this method is that it is always possible to use it. The limit of this technique is that when we detect that the property does not hold, we have to rely on safety margin and other recovery systems to maintain the system in a safe state.

Note that a common limit for both solution is the determination of the threshold. Regarding the recovery methods, we identified three methods that could be used:

- use of redundant NN. To be efficient, the redundant NN needs to be diversified. We presented some ideas to bring diversity to the NN by changing the NN processes, using the bias error, changing the structure of the NN, changing the data, or the $f_{desired}$ function. For this method, we identified two limits. Firstly, the redundant NN may also have erroneous models. Secondly it is possible that the detection mechanism detects a model error in all redundant NN, thus it is necessary to have redundant mechanisms that do not rely on NN.
- use of another AI technology. For example the use of other machine learning technology like decision trees could permit to provide a diversified redundancy that could be more explainable than NN.
- use a mechanism based on simpler NN or AI method, or even imperative programming. As the dependability of simpler AI mechanisms, simpler NN, and particularly imperative programming, is easier to ensure, we could switch to a degraded mode using such mechanisms with higher trust but a simpler function as an alternative to the erroneous NN.

We will present in chapter 8 a practical application for autonomous vehicle of these methods.

Chapter 6

Autonomous vehicle application: nominal conditions

In this chapter we will present an application that implements an autonomous vehicle function using neural networks. The neural network is trained using data generated by an automatic control law regulating the longitudinal control of an ACC (adaptive cruise control) function. The neural network is then evaluated in a driving simulator, by observing its behavior at performing a vehicle's longitudinal control in different scenarios and comparing it with the original ACC function. This process is a verification of the neural network via testing on different scenarios. We finally perform a formal verification of a safety property on this neural network. We will first present the desired ACC function, second the neural network and its development, third the verification of the neural network on a simulator, and finally formal verification of a safety property on the neural network.

6.1 Adaptive Cruise Control

In this section, we present the autonomous driving function that we chose to model in our experience. We present the reasons of this choice, and then describe the function extensively (algorithm, ODD, specifications).

6.1.1 Choice of the ACC as an autonomous function

Among several autonomous vehicle functions, we chose to work with an ACC (Adaptive Cruise Control). This function has the advantage of being well-known by the industry, making it easier to find models and produce data to train the neural network. The ACC function is already implemented in several

vehicles, although using imperative programming with an automatic control law. Compared to other autonomous functions (lateral control, full control, parking assist, urban chauffeur, lane keeping assist) which are usually complex enough to require AI mechanisms, it allows us to better understand and analyze the intended behavior of the function. Note that a neural network would not be usually needed to implement this function, as control laws are enough and have the benefit of being easier to validate. However we do not consider that a problem in our study as we intend it as an application example and a well defined intended function will allow us to better analyze and control it. We will nonetheless consider in our safety property and our fault tolerance mechanisms that the intended function is impossible to predict, as is generally the case when using neural networks.

6.1.2 Adaptive Cruise Control function

In this section we focus on describing the ACC function. We present the function, its role in autonomous driving, the algorithm we chose to work with, and the corresponding ODD and specifications.

6.1.2.1 Description of the ACC

The adaptive cruise control (ACC) is an autonomous driving function of level 1: it implements a longitudinal control of the vehicle, which we will call ego car. It has two main functionalities: adapt the speed to a speed limit entered by the user (*speed following mode*), or keep a safe distance from a car in front, which we will call exo car (*distance following mode*, also called *car following mode*). The standards [51], [50] present in detail the function.

6.1.2.2 Inputs and outputs of the ACC

The ACC function that we consider in our work has 6 inputs:

- S_{ego}: the speed of the ego car, expressed in $m.s^{-1}$
- S_{exo}: the relative speed of the exo car, in comparison to our car, expressed in $m.s^{-1}$. When the exo car is faster than the ego car, this input is positive, and when the exo car is slower than the ego car its value is negative. Note that when there is no exo car, this input has the value 0.
- D: the distance between the ego car and the exo car, expressed in m. When there is no exo car or when the exo car is further than $150m$ this input has the value 150.
- TH_{req}: required time headway (gap time between the exo car and the ego car), set by the user but usually dependant on road rules. This input is expressed in s. For our experiments, this input will always have the value $1.5s$.

- D_0: A safety margin to add to the minimal safety distance between the ego and exo cars, expressed in m. For our experiment, this input will always have the value $5m$.
- S_{ref}: the speed limit set by the user, expressed in $m.s^{-1}$

The ACC function has one output, the acceleration command, expressed in $m.s^{-2}$.

6.1.2.3 ACC algorithm

The ACC function that we use was provided by the company Stellantis. The code of this function is presented below in listing 6.1.

```
1  float ACC(float S_ego,float S_exo, float D, float TH_req, float D0,
       float S_ref) {
2
3      // Here we define some of the gains
4      float Kv1 = 0.36;
5      float K = 1.4;
6      float Kd = 1.5;
7      float Kv2 = 1;
8
9      // Here are the several acceleration variables:
10     // Acons_RVV is based only on the ego speed S_ego and the speed
         limit S_ref. It corresponds to the speed following mode
11     float Acons_RVV;
12
13     // Acons_S_exo is an acceleration variable that is based on the
         exo speed S_exo and the distance to the car in front
14     float Acons_S_exo;
15
16     // Acons_D is an acceleration variable based on the distance to
         the vehicle in front
17     float Acons_D;
18
19     //float Acons_RDD is based on the Acon_S_exo and Acons_D
       variables. Is i the acceleration corresponding to the car
       following mode
20     float Acons_RDD;
21
22     // The final one is the Acons variable. It takes the minimum
       acceleration of the two modes: Acons_RDD and Acons_RVV
23     float Acons;
24
25     // if D=150 and S_exo=0 => No Exo car => D=10 km
26     if ((D==150) && (S_exo==0)) {
27         D = 10000;
28     }
29
30     // Here we compute some variables based on the gains defined
31     // earlier. These variables adjust the behavior of the ACC.
32     // We can see the first one is the safe distance Ds that uses
33     // the TH_req value, the ego speed and the safe distance D0
34     float Ds   = TH_req*S_ego + D0;
35     float KDs  = K* Ds;
```

```
36      float K1Ds = (K-1) * Ds;
37      float K2Ds = (2*(K+0)-1) * Ds;
38
39      // In the following block of code, we define the value of the
        S_exo variable based on the exo car relative speed S_exo, the
        distance and the gains.
40      if (S_exo>=0){
41          //In this case, the exo vehicle goes faster than the ego
        vehicle. We compute a positive acceleration Acons_S_exo.
42          Acons_S_exo = Kv1*S_exo;
43      } else if (D>KDs) {
44          // In this case, the exo car is slower than the ego car,
        but the distance if over KDs (with KDs = 1.4 * (TH_req*S_ego +
        D0) ). The computed acceleration is negative, we want the car
        to slow down.
45          Acons_S_exo = (-Kv2*(S_exo*S_exo)/(2*(D-Ds)));
46      } else {
47          // in this case, we are close to the safe distance Ds (
        because D < KDs ). We compute a strong deceleration. The closer
         we are to the car in front, the stronger the deceleration is.
48          Acons_S_exo = (((-Kv2*(S_exo*S_exo))/(2*K1Ds))*((K2Ds-D)/
        K1Ds));
49      }
50
51      // Here we compute the Acons_D used for the car following mode.
         This is based on the distance D and how far from a safe
        distance the vehicle is
52      Acons_D = Kd * (D-1.15*Ds)/(1.15*Ds);
53      // Here we add a min and max value to the Acons_D variable to
        make it fit in the interval [-1.5 ; 0.3]
54      Acons_D = min(0.3,max(-1.5,Acons_D));
55
56      //Here we compute the Acons_RDD variable based on Acons_S_exo
        and Acons_D that corresponds to the car following mode
57      Acons_RDD = min(1,max(-3,Acons_S_exo + Acons_D));
58      // Here we compute the Acons_RVV variable that corresponds to
        the Speed following mode
59      Acons_RVV = min(1,max(-1, Kv1*(S_ref - S_ego)));
60      // Here we select the lowest acceleration between Acons_RVV (
        car following mode) and Acons_RVV (speed following mode) to be
        used
61      Acons = min(Acons_RDD,Acons_RVV);
62
63      return Acons;
64  }
```

Listing 6.1: ACC in C

Note that this function uses gains values ($Kv1$, K, Kd, $Kv2$). For the nominal behavior of the ACC, we use the values described in the table 6.1. Also, this function uses the variable D_s that represents the safety distance that must be kept between the ego car and the exo car in front. It is computed as $D_s = TH_{req} * S_{ego} + D_0$, $TH_{req} * S_{ego}$ being the distance traveled during a time equals to the minimal headway time allowed with a speed equals to the current ego car speed, and D_0 being a safety margin as previously stated. Another used variable is KDs which is Ds multiplied by the gain value $K = 1.4$. KDs

is used to determine when to switch between the two modes of the ACC. The two modes of the ACC (*distance following* and *speed following*) are respectively controlled by $Acons_{RDD}$ and $Acons_{RVV}$, and the smaller acceleration between the two is chosen to command the vehicle.

- *Acons_RVV*: this acceleration corresponds to the *speed following mode* (when there is no exo car in the 150m in front of the ego vehicle). This value is computed based on the ego speed S_{ego} and on the speed limit set by the user S_{ref} (line 56). This command has a maximal acceleration of $1m/s^2$ and a maximal deceleration of $-1m/s^2$.
- *Acons_RDD*: this acceleration corresponds to the *distance following mode* (when the ego car must adapt its speed to an exo car in front). The value is computed using a complex balance between the distance to the exo car D, the exo car relative speed S_{exo} and the ACC gains described in table 6.1. The details about this calculation can be found in the code (lines 37 to 46).

We can also note that when there is no exo vehicle (represented as $D = 150$ and $S_{exo} = 0m.s^{-1}$), the distance is set to 10000m (10km) to represent a car far away and model a behavior where the exo car has no impact on the acceleration command.

$Kv1$	0.36
K	1.4
Kd	1.5
$Kv2$	1

Table 6.1: Nominal ACC gains

6.1.2.4 Operational Design Domain of our Adaptive Cruise Control function

The Operational Design Domain (ODD) of our ACC function was inspired by industry standards defined in [51] and [50]. These standards have an ACC ODD defined by a minimal S_{ref} of 7 km/h. These standards also mention that for the ACC to compute a positive acceleration, the ego car must already have a speed of 5 m/s, which means that the standards do not consider a vehicle that is stopped as an initial state. The standards also specify that the acceleration output should be over $-3.5m.s^{-2}$. This minimum value is however an average over 2 seconds, which means that an ego vehicle could decelerate stronger than $-3.5m.s^{-2}$ if it stays over this value on average during two seconds. The maximum acceleration is defined as $2.0m.s^2$. For our experiment, the ODD is defined as follows:

- the S_{ref} range is [1.94 m/s (approximately 7 km/h) ; 41,67 m/s (approximately 150 km/h)]. This range is bigger than the one proposed in

standards like [51], but we find useful that it encompasses several speed values that correspond to the three main driving situations: city, country road, highway.

- the S_{ego} range is [1.94 m/s (approximately 7 km/h) ; 41,67 m/s (approximately 150 km/h)] as it has to be at least equals the S_{ref} range for the system to reach the desired speed.

- the S_{exo} range is [-41.67 m/s ; 41,67 m/s]. These values permit to include the two extreme situations: an ego car arriving at full speed (150km/h) on a stopped exo car, and an exo car going at full speed detected by a stopped ego car.

- the function should be activated when the distance to the exo car D is at least equal to D_s, otherwise we can not expect a safe behavior of the system. Nonetheless, we consider a range of [0,150] for D, but note that we will submit in our scenarios that the initial distance D_{init} will always be greater or equal than D_s.

We also limit the ODD to situations where the exo car goes forward. We make this choice because these are very abnormal and improbable situations and we want to make the modeling simpler.

6.1.2.5 Specifications

The specifications are inspired from the industry standards defined in [51] and [50]. Here we present them:

- S_{ego} must never exceed S_{ref}. This correspond to the *speed following mode*.

- TH must be over $TH_{req} = 1.5s$. This correspond to the *car following mode*. The current TH is computed with the equation 6.1.

$$current_{TH} = \frac{D}{S_{ego}} \tag{6.1}$$

- the acceleration command should be in the following range: $[-3.5m.s^2; 1.5m.s^2]$. This is inspired by the industry standard, the reason for this bounds is more for the comfort of the car's passengers than for their safety.

6.2 Development of the neural network

We describe in the figure 6.1 the development of the neural network (training, verification via testing and formal verification). We represent in the purple boxes the inputs related to the ACC function definition: the specifications and the safety property to validate. Based on this ACC function, we create a modified

version ACC_{mod} that will be used to train the NN. The reason why we need this modified version is detailed in section 6.5.6.1, but the general idea is to have a more conservative ACC function to compensate the variance of the NN accuracy and pass the formal verification more easily. The dataset is created with data generated with the ACC_{mod} function, and according to the subdataset distribution presented in section 6.3.2. The three components of a NN dataset necessary to perform the NN training are the subdatasets (presented in section 6.3.2), NN architecture (presented in section 6.3.1) and NN processes (presented in section 6.3.3). The outcome of the training is the trained NN, which will then go through the formal verification of the safety property, and the verification via testing through simulation. The verification via testing of the NN is based on simulations using a set of scenarios that we present in section 6.4.2, and a simulator presented in section 6.4.1. We run simulations of a car being controlled by the NN in all scenarios, and we obtain the scenarios recordings. From these scenarios recordings, we compute measures (presented in 6.4.2.7), which indicate if the NN passes the verification via testing or not. For the formal verification, we use the tool Marabou. The tool verifies that the trained NN respects the safety property presented in section 6.5.1.2.

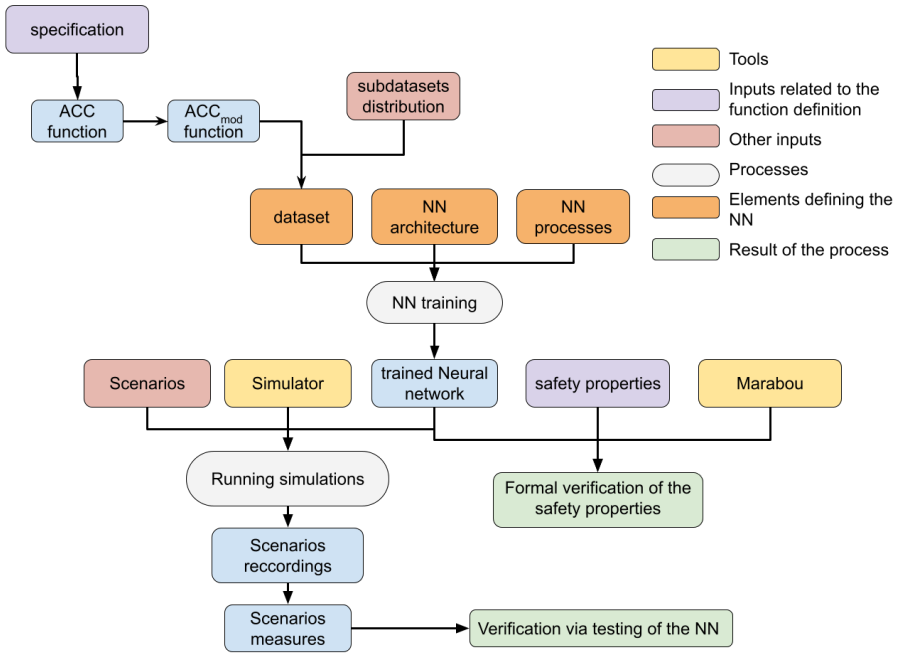


Figure 6.1: Summary of the development (training, verification via testing and formal verification) of the neural network

6.3 Neural network

In this section, we present the neural network used for the nominal ACC. We make a description based on the characterization presented in section 4.1.2: the architecture, the training data, and the processes.

6.3.1 Architecture

For our experiment, we use a feedforward NN architecture with ReLu activation functions. This choice was motivated by the requirements of the formal verification tools we could use, as most of them require ReLu activation functions, notably our chosen tool Marabou. The final architecture of our NN is composed of:

- 1 input layer with the 6 ACC inputs: S_{ego}, S_{exo}, D, TH_{req}, D_0, S_{ref}.
- 4 hidden layers of respective size 64, 64, 32, 32.
- 1 output layer with 3 outputs: "$acceleration$" the acceleration command given by the ACC function, and the two additional outputs needed for formal verification (as we will explain in section 6.5.2) "$futureSpeed$" and "$futureTH$".

To obtain this architecture, we experimented different combinations of number and size of hidden layers while keeping a fully connected network with ReLu activation functions. We list here some of the architectures we tried (we indicate the size of the hidden layers): (15;15), (15;15;15), (64;64;64), (64;64;32), (64;64;64;64), (32;32;32;32), and (64;64;64;64). We will also display some of the architectures we tried in section 9.1.1.

The final choice was guided by two methods:

- firstly by comparing the average mean squared error (MSE) of the three NN outputs on a test set (that has the same distribution as the training set), as well as the median MSE on this test set.
- secondly by comparing the results of the different NN in the simulator to those of the original ACC function, using the measures presented in 6.4.2.7

6.3.2 Training data

To train our NN, we created a training set of 1 million elements from the original ACC function. Each element contains the 6 inputs and the corresponding 3 outputs. For our experiments, a neural network needs to pass the formal verification later described in section 2.1.3 and at the same time pass the verification via testing of the neural network ACC function through the scenario measurements on the simulator described in section 6.4.2.7. We used two strategies to obtain the training data for developing such NN, that we will detail in section 6.5.6:

- the first strategy aims to generate the training data from a more conservative function to obtain acceleration commands slightly lower than the original ACC function, being thus safer and able to validate more easily the safety property, but less available and going slightly lower than the desired speed compared to the original ACC function. The details of this more conservative function already introduced in section 6.2 ACC_{mod}, are given in section 6.5.6.1.

- the second strategy aims to iteratively tune the data distribution in the training set to focus on the parts of the ODD that caused the formal verification to fail.

Regarding the dataset distribution, the final distribution is spread over eight subdatasets and presented in table 6.2. The eight subdatasets are as follows:

- no exo: This dataset (detailed in table 6.3) describes situations where there is no exo car. The absence of an exo car is characterized by $D = 150$ and $S_{exo} = 0$. We generated all S_{ego} inputs following a uniform law on the entire ODD range, and S_{ref} inputs are generated so they are close to S_{ego} (to gain precision when the ego speed approaches S_{ref}). As we will see in section 6.4.2.9, a consequence of this choice is that the ego car may have an incorrect behavior when S_{ego} has values far away from S_{ref}.

- exo uniform: This dataset (detailed in table 6.4) aims to cover all situations with an exo car. The inputs are all generated following a uniform law on the entire range of the ODD with an exo car. Note that the distance only includes data that are in a safe situations.

- exo car following v1: This dataset (described in table 6.5) covers the entire ODD of S_{ego}, but restricts the S_{exo} input to situations where the exo car is no more faster than 4 km/h than the ego car, in order to simulate a following at similar speed. It also only encompasses D values that guarantee a safe situation.

- exo car following v2: This dataset (described in table 6.6) is the same as the previous one, but limits the S_{ego} range to $[30; 40]$.

- exo car following v3: This dataset (detailed in table 6.7) aims to focus on situations that are dangerous, particularly where the exo car is slower than the ego car ($S_{exo} < 0$), and with a distance D slightly higher than D_s. The S_{ego} range is $[20; 41]$.

- exo car following v4: This dataset (described in table 6.8) also focuses on dangerous situations similarly to the subdataset exo car following v3, but the distance is focused on higher values (higher than 50 m) and includes values up to a TH of two seconds, which means that D is up to $2 * S_{ego}$. The S_{ego} range is $[30; 41]$.

name of the dataset	proportion in the training set
no exo	18%
exo uniform	13%
exo car following v1	16%
exo car following v2	8%
exo car following v3	15%
exo car following v4	15%
exo car following v5	3%
car following specific	12 %

Table 6.2: Training datasets distribution

Dataset No Exo	
S_{ego}	$[0; 41.6]$
S_{exo}	$[0; 0]$
D	$[150; 150]$
TH_{req}	$[1.5; 1.5]$
D_0	$[5; 5]$
S_{ref}	$[S_{ego} - 1; S_{ego} + 1]$

Table 6.3: Dataset No exo

- exo car following v5: This dataset (detailed in 6.9) is a very small one (only 3 % as described in table 6.2). It is focused on high ego speed (S_{ego} values in $[40; 12]$), with low S_{exo} values.
- car following specific: this dataset (detailed in table6.10) focuses on small S_{exo} values. The distances D are all defined close to the safety distance. The S_{ego} values are on the entire S_{ego} ODD range.

6.3.3 Processes

We describe here the processes we used to train the neural network. We shuffled then divided the training dataset into 100 batches on 1000 epochs. This process was done with the framework pytorch (version 1.6.0 for GPU). The optimization algorithm was the Adam algorithm ([65]) available in pytorch, and the cost function was the Mean Square Error (MSE) also implemented in pytorch. We kept the default weight and biases initialization method offered in pytorch: a uniform law $U(-\sqrt{k}, \sqrt{k})$, with $k = \frac{1}{number\ of\ input\ features} = \frac{1}{6}$). The cost function and optimization algorithm were chosen following expert recommendations from Stellantis, and the batch sizes and epochs were defined empirically after several trials. Note that we do not use regularization techniques on our NN: we found after some trials that using output regularization generated slightly

Dataset exo uniform	
S_{ego}	$[0; 41.6]$
S_{exo}	$[-S_{ego}; 41.6 - S_{ego}]$
D	$[max(max(0, 0.75 * D_s), -2.5 + 1.5 * S_{ego}); 150]$
TH_{req}	$[1.5; 1.5]$
D_0	$[5; 5]$
S_{ref}	$[-41.6; 41.6]$
Note: $D_s = TH_{req} * S_{ego} + D_0$	

Table 6.4: Dataset exo uniform

Dataset exo car following v1	
S_{ego}	$[0; 41.6]$
S_{exo}	$[-S_{ego}; min(4/3.6, 41.6 - S_{ego})]$
D	$[max(max(max(0, 0.75 * D_s), -2.5 + 1.5 * S_{ego}), 0.9 * D_s); 150]$
TH_{req}	$[1.5; 1.5]$
D_0	$[5; 5]$
S_{ref}	$[-41.6; 41.6]$
Note: $D_s = TH_{req} * S_{ego} + D_0$ and $KD_s = 1.4 * D_s$	

Table 6.5: Dataset car following v1

Dataset exo car following v2	
S_{ego}	$[30; 40]$
S_{exo}	$[-S_{ego}; min(4/3.6, 41.6 - S_{ego})]$
D	$[max(max(max(0, 0.75 * D_s), -2.5 + 1.5 * S_{ego}), 0.9 * D_s); 150]$
TH_{req}	$[1.5; 1.5]$
D_0	$[5; 5]$
S_{ref}	$[-41.6; 41.6]$
Note: $D_s = TH_{req} * S_{ego} + D_0$ and $KD_s = 1.4 * D_s$	

Table 6.6: Dataset car following v2

Dataset exo car following v3	
S_{ego}	$[20; 41]$
S_{exo}	$[-S_{ego}; -15]$
D	$[KD_s * 1.1; KD_s * 1.3]$
TH_{req}	$[1.5; 1.5]$
D_0	$[5; 5]$
S_{ref}	$[-41.6; 41.6]$
Note: $D_s = TH_{req} * S_{ego} + D_0$ and $KD_s = 1.4 * D_s$	

Table 6.7: Dataset car following v3

Dataset exo car following v4	
S_{ego}	$[30; 41]$
S_{exo}	$[-S_{ego}; -15]$
D	$[min(50, D_s); S_{ego} * 2]$
TH_{req}	$[1.5; 1.5]$
D_0	$[5; 5]$
S_{ref}	$[-41.6; 41.6]$
Note: $Ds = TH_{req} * S_{ego} + D_0$ and $KD_s = 1.4 * D_s$	

Table 6.8: Dataset car following v4

Dataset exo car following v5	
S_{ego}	$[40; 42]$
S_{exo}	$[-S_{ego}; -15]$
D	$[min(50, D_s); S_{ego} * 2]$
TH_{req}	$[1.5; 1.5]$
D_0	$[5; 5]$
S_{ref}	$[-41.6; 41.6]$
Note: $D_s = TH_{req} * S_{ego} + D_0$ and $KD_s = 1.4 * D_s$	

Table 6.9: Dataset car following v5

Dataset car following specific	
S_{ego}	$[0; 41.6]$
S_{exo}	$[-S_{ego}/3; min(4/3.6, 41.6 - S_{ego})]$
D	$[max(max(max(0, 0.75 * D_s), -2.5 + 1.5 * S_{ego}), D_s); D_s * 1.05]$
TH_{req}	$[1.5; 1.5]$
D_0	$[5; 5]$
S_{ref}	$[-41.6; 41.6]$
Note: $D_s = TH_{req} * S_{ego} + D_0$ and $KD_s = 1.4 * D_s$	

Table 6.10: Dataset car following specific

worse NN but we did not investigate this problem much as it is not particularly relevant to our work. We also tried to normalize the inputs of the NN, but with no significant improvements. We did not investigate this problem longer as it is also not particularly relevant to our work.

6.4 Framework of the verification via testing

For the verification via testing we used a driving simulator proposed by Stellantis. The objective was to model enough situations that can be representative of the ACC functionalities, although it is obviously impossible to test every possible situation. In this section, we will describe the methodology and tools we used to define the scenarios and measures for our testing.

6.4.1 Simulator

We used an adapted version of the airsim simulator developed by Microsoft [133]. Our version differs from the original mostly on the tracks, the available vehicles, and the autonomous control of cars. We used Windows 10 Entreprise for our experiments. The simulator can integrate one ego car (that can be controlled by an API), and several exo cars (set on a predefined behavior). In our experiments, we will use only one exo car at a time. For our simulations, we used a 1290m straight line track, with the ego car, and possibly an exo car ahead of it, both staying on the left lane. In the photo 6.2 we can see a picture of the simulator with the ego car and an exo car. Note that we only considered a straight track because first our ACC function is only responsible for the longitudinal control of the vehicle and second we had technical difficulties defining S_{ref} inputs the ACC function appropriate to the curves.

Figure 6.2: Simulator

6.4.1.1 Initial state variables

The initial condition of a scenario is set by the following state variables:

- Ego car:
  - initial position: we start each experiment at the position -1950 in the circuit. We will consider in the rest of this thesis this initial position as being 0.
  - $S_{ego_{init}}$: the initial speed of the ego vehicle.
  - S_{ref}: the speed limit set by the user (in our case, sent by the API).
- Exo car (when present in the scenario):
  - initial position: this value is relative to the position of the ego car (set as 0), and noted D_{init}.
  - $S_{exo_{init}}$: initial speed of the exo vehicle.
  - S_{refExo}: the speed that the exo car will reach and follow automatically. This value is a constant in our scenarios.

6.4.1.2 State variable and cycles

The state variables of the simulator describe completely the simulation's state at any point during the simulation. Here is the list of these state variables:

- Environment:
  - t: time since the beginning of the current simulation.
- Ego car:
  - p_{ego}: position of the ego car.
  - S_{ego}: speed of the ego car.
  - acc_{ego}: current acceleration of the ego car.
  - simulator longitudinal control: to perform the longitudinal control of the ego car, we transform the acceleration value computed by the ACC into two throttle values. We will give more details in the description of the ego control in the following section.
    - $acc_{throttle}$: current acceleration throttle value.
    - acc_{brake}: current brake command value.
- Exo car (when there is one):
  - D: distance to the ego car.
  - $S_{exo\ absolute}$: speed of the exo car.
  - acc_{exo}: current acceleration of the exo car.

  - S_{refExo}: the speed that the exo car will reach and follow automatically. This value is a constant in our scenarios.

The simulator uses a 33ms cycle, which means that every 33ms it updates the values of the state variables, and computes the new ego car state based on the ACC command sent and the new exo car state from its variables and its set behavior.

6.4.1.3 Ego car control

The lateral control of the ego car is managed automatically by the simulator. As our scenarios follow a straight line, it is of no relevance to our study. To perform the longitudinal control, we first compute a *desiredAcceleration* value with the ACC function or the NN. Then, based on the current acceleration of the ego car and on the acceleration command of the ACC (the control law or the NN), an acceleration throttle command and a brake command are computed and then used by the simulator to calculate the new state of the ego car. This is done through the *accelToPedal* function presented in listing 6.2. To compute the acceleration throttle value and brake command value, the function computes an error residue of the difference between the current acceleration and the desired one (line 13). The error residue is used to adjust the two variables, however the adjustment is tempered by a factor that changes based on the situation:

- if the brake command is currently not pressed, and the current acceleration is below the one we wish to reach, the factor is equal to 0.05 (line 21).
- if the brake command is currently not pressed, then it will depend on how far away from an exo car we are:
  - if we are close from an exo car (we used the Ds variable to estimate the distance), then the factor is equal to 0.00001 (line 31).
  - if we are away from an exo car (or no exo car at all), then the factor is equal to 0.05 (line 34).
- if the brake command is currently not pressed, but our acceleration is stronger than the one we want, then we completely release the acceleration throttle (we set the value to 0). This corresponds to the line 39.
- when the brake command is at 0, we first check if our current desired acceleration is negative (if we want to slow down) and if at the same time we did not reach yet this desired deceleration. If it is the case, we bring a stronger correction to the brake with a factor of 0.8 (line 48).
- if the desired acceleration is positive or if the current acceleration is already below the desired one (we brake stronger than what we want), then the factor is equal to 0.01 (line 51), which corresponds to a slow release of the brake command.

- finally, if the current throttle is higher than 0, we completely release the brake (line 54).

```
1  // We define a structure used as a return value for our function
2  struct ThrottleAndBrake {
3      // the throttle variable represents the pressure applied to the
        acceleration throttle. A value of 1 indicates maximum pressure
       . A value of 0 indicates no pressure.
4      float throttle;
5      // the brake variable represents the pressure applied to the
       brake throttle. A value of 1 indicates maximum pressure. A
       value of 0 indicates no pressure.
6      float brake;
7  };
8
9  struct ThrottleAndBrake accelToPedal(float currentAccel, float
       desiredAccel, float currentBrake, float currentThrottle, float
       Ds=150, float D=150) {
10     // here we define the variables that will later correspond to
       the variables of the ThrottleAndBrake structure;
11     float throttle, brake;
12     // Here we compute the error residue of the acceleration. In
       other words, how far are we from the correct acceleration
13     float error = desiredAccel - currentAccel;
14
15     // Here we are in the case where the car is currently not
       braking, so we will compute is the throttle should reach 0 or
       increase
16     if (currentBrake == 0)
17     {
18         if (desiredAccel < currentAccel)
19         {
20             // if the current acceleration is stronger than what we
        want, we diminish the throttle proportionally to the error
       residue.
21             throttle = currentThrottle + 0.05*error;
22         } else {
23             // Here we know that we are below the desired
       acceleration, which means we will want to increase the throttle
        variable
24
25             // In this case, we had to split into two cases:
26             // when the ego car is very closed to an exo car (below
        the distance 1.4 * Ds)
27             // when the ego car is further away (D > Ds)
28             if (D < (1.4 * Ds))
29             {
30                 // When we are below the safe distance, we want to
       have a very slight increase of the acceleration value. We do
       that by correcting the throttle command wth a very small amount
       .
31                 throttle = currentThrottle + 0.00001*error;
32             } else {
33                 // when we are far away from the vehicle, we know
       we can accelerate as much as the ACC wants to, so we perform an
        adjustment proportional to the error residue.
34                 throttle = currentThrottle + 0.05*error;
```

```
35              }
36          }
37      } else {
38          // if we are over the desired acceleration: we completely
        release the acceleration throttle (we set the value to 0)
39          throttle = 0;
40      }
41
42      // Here the throttle is at 0, which mean we can potentially
        brake. We will evaluate is the braking is necessary (and
        increase it or decrease it).
43      if (currentThrottle == 0)
44      {
45          if (desiredAccel < currentAccel and desiredAccel < 0)
46          {
47              // if our desired acceleration is negative (aka, we
        want to decelerate) and we currently do not reach this
        deceleration, we will strongly correct the current brake. The
        brake command will increase.
48              brake = currentBrake - 0.8*error;
49          } else {
50              // If we want to accelerate, we release the brake pedal
         (either because the desired aceleration is positive, or
        because the deceleration is stronger than the one we desire).
        To do that we slightly adjust the proportionally to the error
        residue.
51              brake = currentBrake - 0.01*error;
52          }
53      } else {
54          brake = 0;
55      }
56
57      // Here we make sure both variables saty in the [0,1] interval
58      throttle = min(0, max(1, throttle));
59      brake = min(0, max(1, brake));
60
61      // Here we return the two variables through the dedicated
        structure
62      ThrottleAndBrake throttleAndBrake = {throttle, brake};
63      return throttleAndBrake;
64  }
```

Listing 6.2: accelToPedal in C

6.4.1.4 Exo car control

The lateral and longitudinal controls of the exo car are managed automatically by the simulator. In our scenarios, the longitudinal control is set to adapt to the required speed (S_{refExo}) and then keep this speed. When launching a scenario, the car will thus brake or accelerate until it reaches this speed.

6.4.1.5 Need for several runs

Because the simulator is not deterministic, the same scenario (that is using identical initial settings) could lead to slight differences in the cars behaviors.

To counter this variance, we repeat the same scenario several times to compute an average of the measures described in the section 6.4.2.7. For the measures used in our experiments, we took a number of 10 repetitions (also called number of runs) for each scenario. An important number of repetitions means that the time to realize the experiment increases. A small number of repetitions does not compensate the variance due to the undeterministic nature of the simulator. The choice of 10 repetitions for each scenario was determined empirically, as a balance between these two phenomena.

6.4.2 Scenarios for evaluation

Verification via testing of an autonomous driving function is a tremendous task due to the quasi-infinite possible situations of its open environment, particularly considering constrained time and resources. We decided to limit ourselves to scenarios on a straight road with one or no exo car as we focus on the longitudinal control function. We proposed 15 scenarios for this configuration, which we will detail in this section. Obviously we do not consider them close to an exhaustive testing of the ACC function but we tried to use a systematic method to generate them. Our creation process follows the steps defined in [86]. We begin by defining the scenario attributes and then refined the considered scenarios under three levels of abstraction: *functional scenarios*, *logical scenarios* and finally *concrete scenarios*. According to the authors of [86], these levels of abstraction have the following definition:

- "Functional scenarios include operating scenarios on a semantic level. The entities of the domain and the relations of those entities are described via a linguistic scenario notation. The scenarios are consistent. The vocabulary used for the description of functional scenarios is specific for the use case and the domain and can feature different levels of detail.". We can see in this definition that the functional scenarios are text based and must introduce the different objects characterizing a scenario, as well as a description of the behavior of these objects.

- "Logical scenarios represent the entities and the relations of those entities with the help of parameter ranges in the state space. The parameter ranges can optionally be specified with probability distributions. Additionally, the relations of the parameter ranges can optionally be specified with the help of correlations or numeric conditions. A logical scenario includes a formal notation of the scenario." With this definition, we can see that this level of abstraction permits to have a precise definition of the limits of each scenario by defining the ranges formally.

- "Concrete scenarios distinctly depict operating scenarios on a state space level. Concrete scenarios represent entities and the relations of those entities with the help of concrete values for each parameter in the state space." With this level of abstraction, the description describes precisely all variables characterizing a scenario.

6.4.2.1 Scenario attributes

The scenario attributes are the state variables that define a scenario. They are based on the initial state variables mentionned in 6.4.1.1. Of course, the attributes specifying the exo car information only have a value when an exo car is present in the scenario.

- $S_{ego_{init}}$: initial speed of the ego car
- S_{ref}: speed limit of the ego car set by the user
- $S_{exo_{init}}$: initial speed of the exo car
- S_{refExo}: the speed the exo car will reach and keep automatically
- D_{init}: initial distance between the ego and the exo car

Additionally, we use the variable KDs already introduced in 6.1.2.3. As a reminder, this KDs is equal to the safe distance Ds multiplied by the gain $K = 1.4$, and it is used to determine when to switch between the two modes of the ACC (*car following* and *speed following*). Here we use it to define some of the initial values of the scenarios attributes (D_{init} in particular).

6.4.2.2 Functional scenarios

We identified two main functional scenarios for our ACC, one for each of its main functionalities. We consider that a scenario which would combine these two functionalities could always be decomposed in a succession of scenarios of only one functionality.

- Scenario A: the ego car drives without any exo car detected in front. The ego car is then in *speed following mode*.
- Scenario B: the ego car drives with an exo car in front of the ego car. The ego car will then be in *car following mode* if the ego car is slower than S_{ref}, or in *speed following mode* if the exo car is faster than S_{ref}.

We make a new subdivision of these scenarios based on the expected behavior of the ego car by choosing to consider the following situations:

- Scenario A: when there is no exo car detected in front.
  - Scenario A.1: the ego car drives at higher speed than S_{ref}, and thus must lower its speed before maintaining it below S_{ref}.
  - Scenario A.2: the ego car drives at lower speed than S_{ref}, and thus must accelerate before maintaining its speed at S_{ref}.
- Scenario B: when there is an exo car detected in front of the ego car.
  - Scenario B.1: the ego car detects at maximal range (150m) a slower exo car and thus must decelerate and keep following it at a safe distance.

A.1	
$S_{ego_{init}}$	[S_{ref} ; 150 km/h]
S_{ref}	[7 km/h ; 130 km/h]

Table 6.11: Logical scenario A.1

A.2	
$S_{ego_{init}}$	[7km/h ; S_{ref}]
S_{ref}	[7 km/h ; 130 km/h]

Table 6.12: Logical scenario A.2

- Scenario B.2: the ego car is following an exo car that accelerates no higher than S_{ref} and must thus follow it at a safe distance.
- Scenario B.3: the ego car is following an exo car that will accelerate higher than S_{ref} and must thus follow it until its speed goes higher than S_{ref}. Then, the ego car will stay at this speed while the ego car drives away from the ego car.

6.4.2.3 Logical scenarios

A logical scenario is refined from a functional scenario by determining the range of possible values for each attributes. These values can be either an interval or values relative to another attribute. The tables 6.11, 6.12, 6.13, 6.14 and 6.15 present the logical scenarios from the previous functional scenarios. Note that this representation remains exhaustive compared to the choice of situations made at the functional level. We can note that at this stage, each attribute still has a range of possible values: we will select precise values in the next phase (when building concrete scenarios). The ranges of the logical scenarios are built according to the following explanations:

- A.1 (table 6.11): for the S_{ref} value, we encompassed all values from the ODD with a maximum speed of 130 km/h (the maximum speed on highways). For the $S_{ego_{init}}$ range, we encompassed all values over S_{ref} within the limits of the ODD (max value of 150 km/h).
- A.2 (table 6.12): for the S_{ref} value, as for the scenario A.1, we encompassed all values from the ODD with a maximum speed of 130 km/h (the maximum speed on highways). For the $S_{ego_{init}}$ range, we encompassed all values below S_{ref} within the limits of the ODD (min value of 7 km/h).
- B.1 (table 6.13): for the S_{refExo} values, we selected the entire range of the ACC ODD ego speed with a limitation of a maximum speed of 130km/h corresponding to the highway maximum speed. We then fixed the initial speed of the exo car $S_{exo_{init}}$ to the same value as the S_{refExo}, which

B.1	
$S_{ego_{init}}$	[S_{refExo} ; 150 km/h]
S_{ref}	[max(S_{refExo} , S_{ego}) ; 150 km/h]
initial speed exo	equal to S_{refExo}
S_{refExo}	[7 km/h ; 130 km/h]
D_{init}	150m

Table 6.13: Logical scenario B.1

means the exo car is already at its targeted speed at the beginning of the scenario and will just keep this speed for the entire scenario. For the $S_{ego_{init}}$ range, we encompassed all values over S_{refExo} within the limits of the ODD (max value of 150 km/h). This means we considered all cases where the ego car initial speed is higher than the exo car. Following this, we set the S_{ref} to a value higher than both the S_{refExo} and the initial ego speed $S_{ego_{init}}$, which means the ego car will have to adapt its speed to the exo vehicle when it meets it. Finally, we set the initial distance D_{init} to $150m$, which is the distance where the ACC ODD starts, which means the ACC will start considering the exo car to compute the acceleration command right at the beginning of the scenario.

- B.2 (table 6.14): for the S_{refExo} values, we selected the entire range of the ACC ODD ego speed with a limitation of a maximum speed of 130km/h corresponding to the highway maximum speed. We then fixed the initial speed of the exo car $S_{exo_{init}}$ to any value between the minimum ego speed of the ODD (7 km/h) and the exo car speed limit S_{refExo}, which means the exo car drives slower than its targeted speed at the beginning of the scenario and will accelerate until it reaches it, then keep this speed for the rest of the scenario. For the $S_{ego_{init}}$, we set it to be equal to the initial exo car speed $S_{exo_{init}}$. However, the S_{ref} is set to be any value higher than the exo car speed limit S_{refExo} within the maximum value of 150 km/h defined by the ODD. This means that the ego car could reach a higher speed than the exo car, but it will actually be limited by the speed of the exo car. The initial distance D_{init} is set to KDs, which corresponds to the distance in the ACC algorithm where the ACC switches to computing an acceleration command based on the *car following mode*.

- B.3 (table 6.15): for the S_{refExo} values, we selected the entire range of the ACC ODD ego speed with a limitation of a maximum speed of 130km/h corresponding to the highway maximum speed. Then, we set the ego car speed limit S_{ref} to any value below this exo speed limit (with a minimal value of 7 km/h to be within the ODD). We then set both initial speed of the vehicles ($S_{exo_{init}}$ and $S_{ego_{init}}$) to be below S_{ref} and thus also below S_{refExo}). The initial distance D_{init} is set to be between Ds and KDs, which corresponds to the range of the ACC algorithm where the accel-

B.2	
$S_{ego_{init}}$	equal to initial speed exo
S_{ref}	[S_{refExo} ; 150 km/h]
initial speed exo	[7km/h ; S_{refExo}]
S_{refExo}	[7 km/h ; 130 km/h]
D_{init}	KDs (distance where the ACC switches to *car following mode*)

Table 6.14: Logical scenario B.2

B.3	
$S_{ego_{init}}$	equal to intial speed exo
S_{ref}	[7km/h ; S_{refExo}]
initial speed exo	[7 km/h ; S_{ref}]
S_{refExo}	[7 km/h ; 130 km/h]
D_{init}	[Ds ; KDs]

Table 6.15: Logical scenario B.3

eration command is based on the *car following mode*. This determines the behavior described in the functional scenario, since both vehicles will accelerate to reach their speed limit, but the exo speed limit being higher than the ego one, the exo car will reach a higher speed and drive away from the ego vehicle.

6.4.2.4 Concrete scenarios

Finally we chose some concrete values for each scenario attributes among the possible values of the logical scenarios. In the tables 6.16, 6.17, 6.18, 6.19 and 6.20, we sum up the concrete scenarios we chose. To guide us in the process of choosing the values of the scenario attributes, we followed the following principle to ensure a good coverage: pick values from different relevant domains, and particularly, at the limits of these domains. For example we considered that a speed attribute has three domains: [7;50 km/h] for city driving, [50;90km/h] for country roads, [90;130km/h] for highway roads. In the case of the distance, these domains are: $[D_s; KD_s]$ if the ego car is in *car following mode*, $D > KD_s$ if the ego car is in *speed following mode*. Note that we decided to create more scenarios of type B than scenarios of type A, because we considered the scenarios of type B to be more critical due to the presence of the exo car.Another constraint was the duration of all the scenarios. The process of running an important number of scenarios can be long (especially when several runs of the same one are necessary to create an average), therefore we limited the total number of scenarios to approximately 15. We present now in details each concrete scenarios:

- A.1: for this scenario, we created two concrete scenarios presented in table

A.1		
	A.1.1	A.1.2
$S_{ego_{init}}$ (km/h)	27	110
S_{ref} (km/h)	7	90

Table 6.16: Concrete scenarios A.1

A.2		
	A.2.1	A.2.2
$S_{ego_{init}}$ (km/h)	7	90
S_{ref} (km/h)	50	120

Table 6.17: Concrete scenarios A.2

6.16.

  - The scenario A.1.1 corresponds to a car slowing down in the context of an urban environment. We chose an initial speed $S_{ego_{init}}$ of 27 km/h as a common speed for driving in a city, and a speed limit of 7 km/h so the car has to reach the minimal speed of the ODD. As a reminder, S_{ref} is the speed limit of the ego car.
  - The scenario A.1.2 corresponds to a car slowing down in a context of country roads. We chose the initial speed of the ego car S_{ref} to be 110 km/h which is one of the possible speed limits on country roads with at least two lanes per direction, and a speed limit of 90 km/h which is the maximum speed on country roads [116].

- A.2: for this scenario, we created two concrete scenarios presented in table 6.17.

  - The scenario A.2.1 corresponds to a car accelerating in the context of entering a country road. We set the initial ego speed to the minimal ODD ego speed of 7km/h. For the speed limit S_{ref}, we set the value to 90 km/h, which represent one of the possible speed limits on country roads with at least two lanes per direction [116].
  - The scenario A.2.2 corresponds to a car accelerating to reach a highway speed. We set the initial ego speed to 90 km/h which represents one of the possible speed limits on country roads with at least two lanes per direction [116]. We set the speed limit to 120 km/h, a bit below the highway speed limit of 130 km/h.

- B.1: for this scenario, we create four different concrete scenarios presented in table 6.18. For all these scenarios, the initial distance D_{init} is set to 150m as required by the logical scenario.

B.1				
	B.1.1	B.1.2	B.1.3	B.1.4
$S_{ego_{init}}$ (km/h)	30	130	90	130
S_{ref} (km/h)	30	130	90	130
initial speed exo (km/h)	7	7	50	90
S_{refExo} (km/h)	7	7	50	90
D_{init} (m)	150	150	150	150

Table 6.18: Concrete scenarios B.1

- The scenario B.1.1 corresponds to an ego car meeting an exo car in urban conditions. We set the ego speed limit and ego initial speed to 30km/h, a common speed in cities. The exo car initial speed and exo car speed limit are set to 7 km/h, the lower limit of the ACC ODD.
- The scenario B.1.2 corresponds to an ego car driving at highway speed meeting an ego vehicle at low speed. This situation could typically occur at a highway toll or in case of a traffic jam. We set the ego car initial speed and speed limit to 130 km/h which is the maximum speed on the highway. We set the exo car initial speed and speed limit to 7 km/h which is the lower bound of the ego car speed in the ACC ODD.
- The scenario B.1.3 corresponds to a situation where an ego car driving at 90 km/h (the lowest speed recommended on highways) meets an exo car that drives at 50 km/h (the most common city speed limit). To represent this scenario, we set both initial speed and speed limit of the ego car to 90 km/h, and both the exo car initial speed and speed limit to 50 km/h.
- The scenario B.1.4 corresponds to a situation where an ego car driving at 130 km/h (the maximum speed on highways) meets an exo car that drives at 90 km/h (one of the possible speed limits on country roads with at least two lanes per direction [116]). To represent this scenario, we set both initial speed and speed limit of the ego car to 130 km/h, and both the exo car initial speed and speed limit to 90 km/h.

- B.2: for this scenario, we create four different concrete scenarios presented in table 6.19.
  - The scenario B.2.1 describes a situation where both ego and exo vehicles start at a speed of 7km/h, but have different speed limits. The exo vehicle has a speed limit of 50 km/h corresponding to a city speed. The ego vehicle has a speed limit significantly higher (20 km/h faster), to be confident that it is the car following mode that will be

tested in this scenario. The initial distance D_{init} has a value of 13 m which is a bit higher than the initial KDs distance (approximately 11.0 m) where the ego car switches to car following mode.

- The scenario B.2.2 describes a situation where both ego and exo vehicles start at a speed of 30km/h (equivalent to an urban speed), but have different speed limits. The exo vehicle has a speed limit of 50 km/h corresponding to a city speed. The ego vehicle has a speed limit significantly higher (20 km/h faster), as in the previous scenario. The initial distance D_{init} has a value of 25 m which is approximately equal to the initial KDs distance (24.5 m) where the ego car switches to car following mode. Note that the KDs is different because it is based on the S_{ego} which in this case is the initial ego speed $S_{ego_{init}}$.
- The scenario B.2.3 describes a situation where both ego and exo vehicles start at a speed of 50km/h (equivalent to the maximum city speed), but have different speed limits. The exo vehicle has a speed limit of 90 km/h corresponding to a country road speed. The ego vehicle has a speed limit significantly higher: 130 km/h which corresponds to the maximum speed on highways. This difference is significant to be confident that it is the car following mode that will be tested in this scenario. The initial distance D_{init} has a value of 60 m which is significantly higher than the initial KDs distance (approximately 36.1 m) where the ego car switches to car following mode.
- The scenario B.2.4 describes a situation where both ego and exo vehicles start at a speed of 110km/h (equivalent to a highway speed), but have different speed limits. The exo vehicle has a speed limit of 130 km/h corresponding to the maximum highway speed. The ego vehicle has a speed limit of 150 km/h corresponding to the maximal value permitted by the ACC ODD. It is significantly higher than the exo car speed limit (20 km/h faster), as in the previous scenarios. The initial distance D_{init} has a value of 72 m which is approximately equal to the initial KDs distance (71.1 m) where the ego car switches to car following mode.

- B.3: for this scenario, we created three different concrete scenarios presented in table 6.20.
  - The scenario B.3.1 describes a situation where both ego and exo vehicles start at a speed of 10km/h (equivalent to an urban speed), but have different speed limits. The exo vehicle has a speed limit of 50 km/h corresponding to the maximum city speed. The ego vehicle has a speed limit of 30 km/h corresponding to a common city speed limit. The exo car speed limit is significantly higher than the ego car speed limit (20 km/h faster). The initial distance D_{init} has a value of 12 m which is between the Ds value (approximately 9 m) and the KDs value (approximately 12.8 m).

B.2				
	B.2.1	B.2.2	B.2.3	B.2.4
$S_{ego_{init}}$ (km/h)	7	30	50	110
S_{ref} (km/h)	70	70	130	150
initial speed exo (km/h)	7	30	50	110
S_{refExo} (km/h)	50	50	90	130
D_{init} (m)	13	25	60	72

Table 6.19: Concrete scenarios B.2

- The scenario B.3.2 describes a situation where both ego and exo vehicles start at a speed of 50km/h (equivalent to the maximum city speed), but have different speed limits. The exo vehicle has a speed limit of 90 km/h corresponding to the maximum country road speed. The ego vehicle has a speed limit of 70 km/h corresponding to a common country road speed. The exo car speed limit is significantly higher than the ego car speed limit (20 km/h faster). The initial distance D_{init} has a value of 37 m which is approximately equal to the initial KDs distance (36.1 m) where the ego car switches to car following mode.
- The scenario B.3.3 describes a situation where both ego and exo vehicles start at a speed of 50km/h (equivalent to the maximum city speed), but have different speed limits. The exo vehicle has a speed limit of 130 km/h corresponding to the maximum highway speed. The ego vehicle has a speed limit of 110 km/h corresponding to a possible speed on a highway. The exo car speed limit is significantly higher than the ego car speed limit (20 km/h faster), to be confident that the scenario will display the switch between the car following mode and the speed following mode (since the exo car will drive far away from the ego car). The initial distance D_{init} has a value of 32 m which is between the initial safe distance Ds (approximately 25.8 m) and the initial KDs distance (approximately 36.1 m), this is to display the switch between both car following and speed following mode.

6.4.2.5 Duration of the scenarios

We set two limits for the duration of each scenario. One time limitation: the maximum duration of a scenario is 60 seconds. This is to shorten the total time of the experiment. Note that usually the scenarios reach a stable state before the end of this duration, as the ego car enters either a stable *car following mode* or a stable *speed following mode*. We also set a condition based on the distance covered by the vehicles: if the ego car or the exo car reaches the end of the straight line track, we would also interrupt the scenario. This is to prevent the

B.3			
	B.3.1	B.3.2	B.3.3
$S_{ego_{init}}$ (km/h)	10	50	50
S_{ref} (km/h)	30	70	110
initial speed exo (km/h)	10	50	50
S_{refExo} (km/h)	50	90	130
D_{init} (m)	12	37	32

Table 6.20: Concrete scenarios B.3

uncontrolled behavior that happens outside of the straight lines from disrupting the experiment's measures.

6.4.2.6 Results of simulation runs

For each run, we collected several information on the simulation and the cars behavior. In particular, we record at each cycle of the simulator the following variables:

- timeSinceStart: the time since the begining of the run
- ego car state variables:
  - egoPosX: position of the ego car on the road (curvilinear abscissa in regards to the road)
  - egoPosY: position of the ego car regarding the center of the lane.
  - S_{ego}: speed of the ego car
  - acc_{ego}: current acceleration in the simulator
  - $gear_{ego}$: current gear of the ego car in the simulator
  - S_{ref}: speed limit set by the user
  - $acc_{throttle}$: value of the throttle pedal (between 0 and 1)
  - acc_{brake}: value of the brake pedal (between 0 and 1)
- exo car state variables:
  - exoPosX: position of the exo car on the road (curvilinear abscissa in regards to the road). Note that here, we do not collect the position regarding the center of the lane as this information is not useful for our function. Furthermore, we have no control over it since the scenario parameters only influence the exoPosX variable through the initial distance. This position towards the center of the lane of the exo car is actually completely determined by the lateral control of the autopilot for exo cars provided by the simulator.
  - exoSpeed: speed of the exo car

- other variables:
  - S_{exo}: relative speed of the exo car compared to the ego car speed
  - D: distance between the two cars
  - TH_{req}: the TH set by the user (in our case, always $1.5s$)
  - D_0: the safety distance (in our case, always 5 m)
  - *desiredAcceleration*: the acceleration computed by the ACC function (or the NN modeling it), used as the acceleration command for the ego vehicle.
  - $desiredAccelerationGroundTruth$: we also collected the ground truth of the desired acceleration (in case the control is done with a NN). This value is the one computed by the ACC function.
  - the outputs of the NN used for our mathematical model, with the associate ground truth (note that these ground truths are computed using the *desiredAcceleration* value):
    - $futureSpeed$ (the NN output) and $futureSpeedGroundTruth$ (computed using equation 6.2).

$$futureSpeedGroundTruth = S_{ego} + desiredAcceleration * delta_t \quad (6.2)$$

    - $futureTH$ (the NN output) and $futureTHGroundTruth$ (computed using the equation 6.3). Note that for this calculation, we set a minimal value of 0 and a maximal value of 60 seconds.

$$futureTHGroundTruth = max\left(0, min\left(60, \frac{future\ distance}{futureSpeedGroundTruth}\right)\right)$$
$$futureTHGroundTruth = max\left(0, min\left(60, \frac{D + S_{exo} * delta_t}{S_{ego} + desiredAcceleration * delta_t}\right)\right) \quad (6.3)$$

For each scenario, we executed 10 runs to compute an average to compensate the simulator instability as explained in 6.4.1.5

6.4.2.7 Measures for the scenarios

Defining measures is of particular importance to compare the different systems that we use. We propose the following measures for each scenario.

- M_{TH}: time (in s) with $TH < TH_{req}$. This measure is related to the safety property that we intend to formally verify on the neural network. A value over 0 for one specific scenario indicates that the safety property was invalidated at some point in one or several of the runs of this scenario.

- $M_{D<D_s}$: time (in s) with $D < Ds$. This measure indicates when the car is getting closer to invalidate the property. Having a value over 0 does not necessarily means that the system is in an unsafe state, since the distance Ds when expressed in seconds is higher than the TH_{req} value.

- $M_{S_{ego}>S_{ref}}$: time (in s) with $S_{ego} > S_{ref}$. Note that this property may be invalidated without the system's behavior being incorrect, like in scenario A.1 where we start the scenario with $S_{ego} > S_{ref}$. In order to know whether or not a positive measure is normal, we recommend to compare it to the similar measure of the original ACC command.

We also added two measures that estimate the time that the output *desiredAcceleration* spent outside of the specification boundaries.

- $M_{high\ accel}$: time spent with $1.5 < desiredAcceleration$

- $M_{low\ accel}$: time spent with $desiredAcceleration < -3.5$

6.4.2.8 Neural network evaluation

In addition to the measures described in section 6.4.2.7, we developed other measures specifically to evaluate our trained NN, and compare it with the ACC function that it should learn.

- $M_{S_{ego}avg}$: the average ego speed. This measure computes the average speed of the ego vehicle on each scenario.

- $M_{D_{avg}}$: the average distance: this measure computes the average distance between the ego car and the exo car in each scenario.

- We also implemented measures to evaluate how close the ego car was driving to the speed limit S_{ref}. From an availability point of view, when there is no exo car slowing the ego car down, the ego car should be as close to S_{ref} as possible without going over it.

  - $M_{S_{ref}*0.9}$ time spent with $S_{ego} < 0.9 * S_{ref}$
  - $M_{S_{ref}*0.85}$ time spent with $S_{ego} < 0.85 * S_{ref}$
  - $M_{S_{ref}*0.8}$ time spent with $S_{ego} < 0.8 * S_{ref}$
  - $M_{S_{ref}*0.75}$ time spent with $S_{ego} < 0.75 * S_{ref}$

6.4.2.9 Measures of the nominal neural network tests

Tables 6.21, 6.22, 6.23 and 6.24 present the results of our measures for the original ACC command (noted ACC in the tables) and a nominal NN that we generated to emulate it (noted NN_n in the tables).

First, considering the safety property, we can see that M_{TH} is 0 for both systems in all the scenarios, which means that the safety property stayed true in all scenarios. Note however that $M_D < D_s$ is 0 except for the scenario B.3.1, where it has the same value of 0.06 for both systems. This is because in this scenario, the simulator has a small malfunction making the initial distance shorter than in the scenario description and making the actual initial distance below the safe distance Ds. This is corrected quickly with both systems.

Second, considering the functional properties, we can see that $M_{S_{ego}} > S_{ref}$ is greater than 0 for both systems only in scenario A.1.1 and A.1.2 (with very close values). This is because we began these scenarios with a speed higher than S_{ref}. As soon as the vehicle's speed goes below S_{ref}, it stays there. $M_{low\ accel}$ and $M_{high\ accel}$ are equal to 0 in all scenarios, as required.

Regarding the other measures used to compare the behavior of the NN to the original ACC command, we can see that most of them are very similar, except in certain cases. Those cases are due to two reasons:

- for scenarios A1.1, B3.1, B3.2 and B3.3, we can see than the four $M_{S_{ref}}$ measures are most of the times significantly higher for the NN than for the original ACC command. This is due to the more conservative behavior learned by the NN to ensure its safe behavior. As we explained previously in section 6.3.2, focusing on safety has a negative impact on availability.

- for scenarios B2.1, B2.2 and B2.3, the problem is due to bad learning from the NN in situations when the ego-car is close to 0 speed and can have a high acceleration. Due to training errors, the acceleration value stays low in such situations. In particular, this is due to us concentrating on S_{ego} values close to S_{ref} in the no exo dataset presented in section 6.3.2: as there was very few data of S_{ego} being far from S_{ref}, the NN didn't learn these situations well. We have corrected this problem on some erroneous NN that we will present in section 7.3.1.

6.5 Formal verification of the nominal neural network

The formal verification presented in this section aims to prove that a neural network can validate a safety property on his ODD. In this section we present how we performed the formal verification by detailing the safety property, the tool we used, and the modifications that we had to make to the neural network. We also present two complementary strategies that we implemented to pass the formal verification, and the results of the formal verification.

		scenarios			
		A.1.1	A.1.2	A.2.1	A.2.2
M_{TH}	ACC	0.0	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{D<D_s}$	ACC	0.0	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{S_{ego}>S_{ref}}$	ACC	0.332	0.333	0.0	0.0
	NN_n	0.327	0.333	0.0	0.0
$M_{low\ accel}$	ACC	0.0	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{high\ accel}$	ACC	0.0	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{S_{ego}avg}$	ACC	1.82	24.15	10.38	29.72
	NN_n	1.6	22.64	10.37	29.18
$M_{D_{avg}}$	ACC	150.0	150.0	150.0	150.0
	NN_n	150.0	150.0	150.0	150.0
$M_{S_{ref}*0.9}$	ACC	4.7	0.56	18.73	9.83
	NN_n	39.56	2.43	19.34	10.27
$M_{S_{ref}*0.85}$	ACC	3.9	0.0	17.54	7.03
	NN_n	39.55	0.0	14.99	4.3
$M_{S_{ref}*0.8}$	ACC	3.37	0.0	16.33	4.23
	NN_n	4.18	0.0	13.87	2.57
$M_{S_{ref}*0.75}$	ACC	3.0	0.0	15.13	1.37
	NN_n	3.42	0.0	12.77	0.9

Table 6.21: All measures on nominal NN vs ACC function on scenarios A

		scenarios			
		B.1.1	B.1.2	B.1.3	B.1.4
M_{TH}	ACC	0.0	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{D<D_s}$	ACC	0.0	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{S_{ego}>S_{ref}}$	ACC	0.0	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{low\ accel}$	ACC	0.0	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{high\ accel}$	ACC	0.0	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{S_{ego}avg}$	ACC	3.63	3.63	15.24	26.84
	NN_n	3.61	3.61	15.24	26.84
$M_{D_{avg}}$	ACC	30.2	17.2	44.6	68.5
	NN_n	43.2	18.1	48.5	74.4
$M_{S_{ref}*0.9}$	ACC	32.71	39.74	37.52	20.07
	NN_n	39.87	39.74	39.8	22.64
$M_{S_{ref}*0.85}$	ACC	30.86	39.61	35.81	17.92
	NN_n	39.87	39.61	39.7	22.5
$M_{S_{ref}*0.8}$	ACC	30.12	39.51	34.61	16.92
	NN_n	37.1	39.51	37.32	16.37
$M_{S_{ref}*0.75}$	ACC	29.55	39.41	33.62	16.18
	NN_n	32.8	39.41	32.79	14.63

Table 6.22: All measures on nominal NN vs ACC function on scenarios B.1

		scenarios			
		B.2.1	B.2.2	B.2.3	B.2.4
M_{TH}	ACC	0.0	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{D<D_s}$	ACC	0.0	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{S_{ego}>S_{ref}}$	ACC	0.0	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{low\ accel}$	ACC	0.0	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{high\ accel}$	ACC	0.0	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{S_{ego}avg}$	ACC	12.76	13.52	23.96	35.02
	NN_n	3.29	13.0	20.45	33.83
$M_{D_{avg}}$	ACC	33.1	36.3	66.5	78.2
	NN_n	133.9	119.3	133.0	103.6
$M_{S_{ref}*0.9}$	ACC	39.94	39.94	25.6	16.17
	NN_n	39.94	35.03	25.6	17.27
$M_{S_{ref}*0.85}$	ACC	39.94	39.94	25.6	6.4
	NN_n	39.94	23.43	24.31	8.85
$M_{S_{ref}*0.8}$	ACC	37.0	39.94	25.6	3.6
	NN_n	39.94	22.01	22.46	5.13
$M_{S_{ref}*0.75}$	ACC	30.75	36.63	22.11	2.53
	NN_n	39.94	20.73	20.66	3.4

Table 6.23: All measures on nominal NN vs ACC function on scenarios B.2

		scenarios		
		B.3.1	B.3.2	B.3.3
M_{TH}	ACC	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0
$M_{D<D_s}$	ACC	0.067	0.0	0.0
	NN_n	0.066	0.0	0.0
$M_{S_{ego}>S_{ref}}$	ACC	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0
$M_{low\ accel}$	ACC	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0
$M_{high\ accel}$	ACC	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0
$M_{S_{ego}avg}$	ACC	7.38	18.19	23.95
	NN_n	3.22	15.87	19.48
$M_{D_{avg}}$	ACC	127.9	123.9	121.8
	NN_n	134.9	128.0	123.4
$M_{S_{ref}*0.9}$	ACC	6.13	3.55	11.43
	NN_n	39.94	17.54	20.34
$M_{S_{ref}*0.85}$	ACC	5.33	2.97	8.83
	NN_n	28.83	12.23	18.06
$M_{S_{ref}*0.8}$	ACC	4.85	2.4	7.84
	NN_n	26.81	9.51	16.41
$M_{S_{ref}*0.75}$	ACC	4.46	1.9	7.07
	NN_n	25.98	7.21	14.87

Table 6.24: All measures on nominal NN vs ACC function on scenarios B.3

6.5.1 Safety property

In this section, we describe the safety property that we will verify on the NN. We also describe the constraints on our operational domain that we will use as preconditions for all formal properties.

6.5.1.1 Restrictions on the operational domain

Restraining the input space for the formal verification is necessary as the properties to be validated may not hold for some inputs that are not part of the operational domain. Moreover it will allow to limit the search during the formal verification and finish it faster. The restrictions on the inputs of the NN due to our operational domain are the following:

- First, we implemented the limitations on S_{ego}, S_{exo}, D and S_{ref}, that correspond to the one on our ACC ODD described in 6.1.2.4:

$$\begin{cases} restriction\ 1: & 1.94 < S_{ego} < 41.67 \\ restriction\ 2: & -41.67 < S_{exo} < 41.67 \\ restriction\ 3: & 0 < D < 150 \\ restriction\ 4: & 1.94 < S_{ref} < 41.67 \end{cases} \tag{6.4}$$

- We also set constant values for TH and D_0 for our experiment:

$$\begin{cases} restriction\ 5: & TH_{req} = 1.5 \\ restriction\ 6: & D_0 = 5 \end{cases} \tag{6.5}$$

- Finally, we added restrictions on the speed of the exo car to limit the search state by excluding situations that we consider outside of the situations that the NN is expected to deal with. These constraints are set knowing that the absolute exo speed is defined as $V_{exo} = S_{ego} + S_{exo}$. We separate this constraint into two restrictions: one that excludes situations where the ego car goes backward (restriction 7) to be aligned with the ODD presented in section 6.1.2.4, and one that excludes situations with extremely high exo speed (more than 150 km/h) corresponding to the restriction 8.

$$\begin{cases} restriction\ 7: & S_{ego} + S_{exo} > 0, \\ restriction\ 8: & S_{ego} + S_{exo} < 41.67, \end{cases} \tag{6.6}$$

The complete precondition that represents the ODD limitation can thus be formally written as follow: $((1.94 < S_{ego} < 41.67)$ AND $(-41.67 < S_{exo} < 41.67)$ AND $(0 < D < 150)$ AND $(TH_{req} = 1.5)$ AND $(D_0 = 5)$ AND $(1.94 < S_{ref} < 41.67)$ AND $(\ 0 < S_{ego} + S_{exo} < 150))$. We will call these preconditions in the following *common preconditions*.

6.5.1.2 Safety property

We looked for several properties, among them we identified one safety requirement that the ACC function should respect during operation: *The ego car should not crash into the car in front.* We reformulate this requirement as the following property: if the TH between the ego car and the exo car is above TH_{req} (1.5s), then it should stay above TH_{req} in the future. This can be expressed formally as: IF $TH > TH_{req}$ THEN $futureTH > TH_{req}$, $futureTH$ being the TH in 0.5s. This formal property should guarantee that the ego car stays at more than 1.5s from the exo car at any time, thus respecting the industrial standard to theoretically avoid possible crashes. We also included a second precondition that excluded situations outside of the physical capabilities of the function: we only consider situations where the distance will stay safe when the car brakes at the maximal deceleration ($minAcc = -3$) allowed by the ACC function, or as formally demonstrated below IF $D > 1.5 * S_{ego} + 0.5 * S_{exo} - 3$. Indeed, we can not ask our system to do something that is physically impossible. We detail below how we obtained this constraint:

$$
\begin{aligned}
futureTH &> TH_{req} \\
\Leftrightarrow \frac{futureD}{futureSpeed} &> TH_{req} \\
\Leftrightarrow \frac{D + (S_{exo} - delta_t * minAcc) * delta_t}{S_{ego} + minAcc * delta_t} &> TH_{req}
\end{aligned}
$$

$$
\begin{aligned}
\Leftrightarrow D + (S_{exo} - delta_t * minAcc) * delta_t &> \\
TH_{req} * (S_{ego} + minAcc * delta_t) & \\
\Leftrightarrow D > TH_{req} * (S_{ego} + minAcc * delta_t) & \\
-(S_{exo} - delta_t * minAcc) * delta_t & \\
\Leftrightarrow D > TH_{req} * S_{ego} + TH_{req} * minAcc * delta_t & \\
-(S_{exo} * delta_t) + delta_t * minAcc * delta_t &
\end{aligned}
$$

with :

$$
\begin{aligned}
TH_{req} &= 1.5 \\
delta_t &= 0.5 \\
minAcc &= -3
\end{aligned}
$$

we thus get:

$$
\begin{aligned}
D > 1.5 * S_{ego} + 1.5 * (-3) * 0.5 - (S_{exo} * 0.5) + 0.5 * (-3) * 0.5 & \qquad (6.7) \\
\Leftrightarrow D > 1.5 * S_{ego} + 0.5 * S_{exo} - 3 &
\end{aligned}
$$

The final formal safety property is then the one presented in equation 6.8. We can note that the measure M_{TH} can be used to perform an online evaluation of the safety property. According to tables 6.21, 6.22, 6.23 and 6.24 we can then see that the script ACC validates the safety property in all scenarios (since the measure is always equal to zero).

$$\begin{aligned} &IF\ (common\ preconditions) \\ &\quad AND\ (TH > TH_{req}) \\ &\quad AND\ (D > 1.5 * S_{ego} + 0.5 * S_{exo} - 3) \\ &\quad THEN\ futureTH > TH_{req} \end{aligned} \tag{6.8}$$

6.5.1.3 Other possible verification properties

We also considered other properties but did not formally verified them on the neural network because of the cost of such verifications. We based these other properties on industry requirements that we could transform into formal properties. Two are based on guaranteeing comfort for the user (by not having a too rough driving) and three are based on the specification of the desired function of the system.

The two properties based on the user's comfort are:

- The system should not accelerate too strongly. We reformulate this requirement as the following property: the system should not accelerate above 1.5 $m.s^{-2}$, or formally as: IF (*common preconditions*) THEN $acceleration < 1.5$.

- The system should not brake too strongly. We reformulate this as the following property: the system should not decelerate below $-3.5 m.s^{-2}$, or formally as:

$$IF\ (common\ preconditions)\ THEN\ acceleration > -3.5 \tag{6.9}$$

The following properties are based on specifications of the ACC:

- When the system speed is below the speed set by the user, it should stay below it. We reformulate this requirement as the following property: if the current speed of the ego car is below the speed limit, its future speed should also stay below it, or formally as: IF ($S_{ego} < S_{ref}$) AND (*common preconditions*) THEN $futureSpeed < S_{ref}$, with $futureSpeed$ being the speed of the ego car in 0.5 seconds. Note that we considered this property as a functional property and not a safety property as the speed of the car follows the speed set by the user, without indication whether this speed is safe or not.

- When there is no exo car, and the ego car's speed is not at its maximal speed, it should accelerate. We reformulate this requirement as the following property: if there is no exo car and the ego speed is significantly below S_{ref}, the car should accelerate, or formally as: IF (*common preconditions*) AND $S_{exo} = 0$ AND $D = 150$ AND $egoSpeed < S_{ref} * 0,9$ THEN $acceleration > 0$, with $S_{exo} = 0$ AND $D = 150$ being the description of an absence of exo car.

- When there is no exo car and the system is close to its maximal speed, it should stay close to its maximal speed. We reformulate this requirement as the following property: if there is no exo car and the ego speed is close to S_{ref}, the future speed should stay close to S_{ref} as well, or formally as: IF (*common preconditions*) AND $S_{exo} = 0$ AND $D = 150$ AND $S_{ref} * 0,9 < egoSpeed$ THEN $S_{ref} * 0,9 < futureSpeed$.

6.5.2 Modification of the NN

Because our verification properties use state variables that are not in the initial ACC inputs and outputs, we need to add these missing state variables as extra inputs or outputs of the NN in order to perform the formal verification. This is an example of the problem with formal verification for NN mentioned in section 4.3. In our case, we added two outputs in addition to the acceleration command: $futureTH$ and $futureSpeed$. Note that the outputs $futureSpeed$ and $futureTH$ were added as outputs of the NN rather than inputs because they are consequences of the acceleration output decided by the NN.

6.5.3 Marabou tool use

Among the several FV for NN tools available, we chose to use Marabou [61] as discussed in section 4.5. We already presented Marabou in section 4.5.2.1 and we used the same version[1] for this experiment. In listing 6.3 we present the code used to perform the formal verification of the safety property. In this listing, we start by importing the numpy and Marabou libraries (line 2 and 3). Then we specify the name of the file containing the neural network (line 6) and load the neural network (line 9). We define the input and output variables of the neural network in lines 12 and 13. We have in the code the following corresponding variables:

- S_{ego} is $inputVars[0]$
- S_{exo} is $inputVars[1]$
- D is $inputVars[2]$
- TH_{req} is $inputVars[3]$

[1] version from the 4 September 2021, commit number: b0e29fb43b6722dfe9b5a90cc1353990aa732327

- $D0$ is $inputVars[4]$
- S_{ref} is $inputVars[5]$
- $desiredAcceleration$ is $outputVars[0]$
- $futureSpeed$ is $outputVars[1]$
- $futureTH$ is $outputVars[2]$

The max and min values for these variables are defined on lines 16 to 24. The values are defined by the constraints related to the ODD limits. From line 27 to 38, we define the upper and lower bounds of each input variables using the min and max values previously defined. On line 42 we add an inequality corresponding to the restriction 8 from equation 6.6. On line 45 we add an inequality corresponding to the restriction 7 from equation 6.6. On lines 48, 49 and 50 we define variables used for the precondition presented in equation 6.7. This precondition is then added on line 53. On line 56, we encode the precondition $TH > TH_{req}$, and on line 59 the precondition $S_{exo} < 0$. Finally the property $futureTH < 1.5$ is encoded on line 63. Note that because we look for counter examples of the property $futureTH >= 1.5$, we encode the opposite of the property. On line 66, we call the Marabou solver that will compute if the neural network respect the set of constraints and thus determine if it validates the safety property, and store the results in a log file.

```
1  # first we import the libraries
2  import numpy as np
3  from maraboupy import Marabou
4
5  # here we identify the file containing the neural network
6  filename = "name_of_the_NN_file.pb"
7
8  # here we load the neural network
9  network = Marabou.read_tf(filename)
10
11 # here we identify the input and output variables
12 inputVars = network.inputVars[0][0]
13 outputVars = network.outputVars[0]
14
15 # min and max values of each input
16 minS_ego, maxS_ego = 1.94 , 41.67
17 minS_exo, maxS_exo = -41.67 , 41.67
18 minD, maxD = 0, 150
19 minTH, maxTH = 1.5 , 1.5
20 minD0, maxD0 = 5,5
21 minS_ref, maxS_ref =  1.94 , 41.67
22
23 minacons, maxacons = -3, 1
24 epsilon_on_output = 0.5
25
26 # Set input bounds
27 network.setLowerBound(inputVars[0],minS_ego)
28 network.setUpperBound(inputVars[0], maxS_ego)
29 network.setLowerBound(inputVars[1],minS_exo)
```

```
network.setUpperBound(inputVars[1], maxS_exo)
network.setLowerBound(inputVars[2],minD)
network.setUpperBound(inputVars[2], maxD)
network.setLowerBound(inputVars[3],minTH)
network.setUpperBound(inputVars[3], maxTH)
network.setLowerBound(inputVars[4],minD0)
network.setUpperBound(inputVars[4], maxD0)
network.setLowerBound(inputVars[5],minS_ref)
network.setUpperBound(inputVars[5], maxS_ref)


# S_ego+ S_exo< 41.67
network.addInequality([inputVars[0], inputVars[1]], [1,1], 41.67)

# S_ego+ S_exo > 0 km/h
network.addInequality([inputVars[0], inputVars[1]], [-1,-1], 0 )

# Specific pre-conditions:
minFutureAcc = -3
delta_t = 0.5
THreq = 1.5

# IF D > TH_{req} * S_{ego} + TH_{req} * minAcc * delta\_t - (S_{
    exo} * delta \_t) + delta\_t * minAcc * delta\_t
network.addInequality([inputVars[0], inputVars[1], inputVars[2] ],
    [THreq , -delta_t , -1], - THreq * minFutureAcc * delta_t  -
    delta_t * minFutureAcc * delta_t )

# IF TH > THreq
network.addInequality([inputVars[0], inputVars[2] ], [THreq , -1],
    0)

# IF S_exo < 0
network.addInequality([inputVars[1] ], [1], 0)

# PROPERTY
# let's look for inputs where futureTH < requiredTH
network.addInequality([outputVars[2] ], [1], 1.5 )

# Call to C++ Marabou solver
vals, stats = network.solve("marabou.log")
```

Listing 6.3: Code used for the formal verification of the safety property

6.5.4 Limit due to formal verification tool

As mentioned in section 4.3, one of the issues with FV is the limits imposed by the tool. In our case, a recurrent neural network architecture would have been more adequate to the problem since our system model includes a time dimension (we have to anticipate the behavior of the car and its environment in the future). Also, other activation functions that ReLu might be possible, but we did not focus on this aspect. An important limitation is also the format of the properties that can be verified by Marabou as already mentioned earlier in section 4.3. For example, an interesting experiment would be to have the possibility to change

the TH_{req} value (which means not considering it as a constant), and verify a property that checks that the future distance computed through equation 6.10 is always over the future safe distance computed through the equation 6.11.

$$futureD = D + S_{exo} * delta_t \tag{6.10}$$

$$futureDs = TH_{req} * (S_{ego} + desiredAcceleration * delta_t) + D0 \tag{6.11}$$

However, this would require to multiply several inputs and outputs of the neural network, which is not permitted by the property format.

6.5.5 Formal verification results

As explained in section 4.3, Marabou is based on the search for a counter example. Let us consider a safety property expressed under the following format, similar to our safety property 6.8:

$$\begin{gathered} IF\ ((ODD\ constraints)\ AND\ (specific\ preconditions)) \\ THEN\ (\nexists\ (inputs, outputs), |(inputs, outputs)\ satisfy\ (\overline{post\ condition})) \end{gathered} \tag{6.12}$$

This property can be rewritten as follows

$$\begin{gathered} IF\ (common\ preconditions)\ AND\ (specific\ preconditions) \\ THEN\ (\forall\ (inputs, outputs), (inputs, outputs)\ satisfy\ (post\ condition)) \end{gathered} \tag{6.13}$$

As Marabou will search for counter examples, we have to express the negation of this property, which is:

$$\begin{gathered} IF((common\ preconditions AND\ specific\ preconditions)) \\ THEN exists(inputs, outputs)|(inputs, outputs) satisfy(\overline{post\ condition}) \end{gathered} \tag{6.14}$$

Finally, considering our safety property in equation 6.8 we obtain the following formal property to be verified in Marabou:

$$
\begin{aligned}
IF\ (\ & (1.94 < S_{ego} < 41.67) \\
& AND(-41.67 < S_{exo} < 41.67) \\
& AND(0 < D < 150) \\
& AND(1.94 < S_{ref} < 41.67) \\
& AND(TH_{req} = 1.5) \\
& AND(D_0 = 5) \\
& AND(S_{ego} + S_{exo} > 0) \\
& AND(S_{ego} + S_{exo} < 41.67) \\
& AND\ (TH > TH_{req}) \\
& AND\ (D > 1.5 * S_{ego} + 0.5 * S_{exo} - 3)\ ) \\
THEN\ (\ & \\
& \exists(S_{ego}, S_{exo}, D, TH_{req}, D_0, S_{ref}, acceleration, futureTH, futureSpeed) \\
& \mid\ not(futureTH > TH_{req})\ )
\end{aligned}
\tag{6.15}
$$

After tuning the training as described in section 6.5.6, we obtained two neural networks that would pass the formal verification of the safety property with Marabou and give satisfying results compared to the original ACC command from the simulator (as presented in section 6.4). The second aspect is important to take into account, as for example a Neural Network that would brake as much as possible in every situations would pass the formal verification, but obviously would not deliver the desired function.

6.5.6 Need for precision

During the preparation of this application, we quickly obtained neural networks that gave satisfying results in the simulator. However, the neural networks were often not precise enough to provide the formal verification of the safety property. Indeed, our first neural networks would systematically invalidate the safety property on some parts of the ODD. As the Marabou tool would only give the first counter example it found, we quickly began to first test the neural networks on a uniformly generated random set of approximately 500 000 inputs from the ODD where we checked the validity of the safety property before doing the time costly formal verification on Marabou. We call this the *safety property testing*. Note that contrary to the formal verification this testing is not exhaustive, but as it is much faster to do (around 15 minutes instead of the 3 to 8 hours for the formal verification), it is first very time effective to prune out the NN that have too much variance errors, and second it allows us to better see which parts of the ODD raise problems than the unique counter example of Marabou. This testing showed us that significant parts of the ODD caused problems in all the trained NN. To obtain a successful formal verification, we developed two strategies to obtain neural networks more prone to satisfy the

property: first we developed the NN from a more conservative function ACC_{mod} as described in 6.3.2, second we iteratively modified the training set to provide more data at each step on the part of the ODD that would not pass the formal verification. In this section we will present these two strategies in more details.

6.5.6.1 Conservative ACC function

Our first strategy consisted in training the neural network to learn a function slightly more conservative than the ACC function described in 6.1. The idea is that if we learn a function whose results give more time between the exo car and ego car, we would have less outliers from the neural network's training that violate the safety property, as they would require bigger errors to do so. We thus developed a modified version of the ACC function that we call ACC_{mod}. This modified version mostly differs from the original in the calculation of the acceleration command *Acons_S_exo* corresponding to the *distance following mode*, as this is obviously the mode where the headway time TH can become low. In the modified version, we increased the impact of the distance on the deceleration of the ACC, and we also added thresholds on the ego car speed and on the exo car speed. These thresholds are separation between different decelerating behaviors. We present below in more details when deceleration changes occur:

- if the distance is above KDs, the behavior is similar to the original ACC
- if $1.04 * Ds < D < KDs$, the deceleration of *Acons_S_exo* is the same as the original ACC when $D < KDs$ (reminder: $KDs = 1.4 * Ds$).
- if $1.02 * Ds < D < 1.04 * Ds$, as the ego car gets closer to the exo car, deceleration of *Acons_S_exo* is 1.5 stronger than the original ACC.
- if $Ds < D < 1.02 * Ds$, as the ego car is very close to the safety distance, the deceleration command will also take into account the relative speed of the exo vehicle S_{exo}:
  - if $S_{exo} < -5$ (if the exo car is significantly slower than the ego car), then the deceleration of *Acons_S_exo* is 2.5 stronger than the original ACC.
  - if $S_{exo} > -5$, then the value value of *Acons_S_exo* will depend on the final command of the function (that also takes into account *Acons_D*). We multiply all values of *Acons_S_exo* by 6 when the final acceleration would fall between $-0.5m/s^2$ and $0.5m/s^2$, and by 4 otherwise.
- if $0.95 * Ds < D < Ds$, which means we should soon (or already have) violated the safety property, the deceleration of *Acons_S_exo* is 2.5 stronger than the original ACC.
- if $D < 0.95 * Ds$, as we risk an imminent failure, the deceleration of *Acons_S_exo* will also depend on the speed on the ego car speed S_{ego} and the exo car speed S_{exo}:

- if $S_{exo} > -10$ and $S_{ego} < 30$ (the ego car is slower than 108 km/h) then the deceleration of *Acons_S_exo* is multiplied by 3.
- if $S_{exo} > -10$ and $S_{ego} > 30$ (the ego car speed is equal or higher than 108 km/h), then the acceleration of *Acons_S_exo* is multiplied by 5.
- if $S_{exo} < -10$ (if the exo car is significantly slower than the ego car), then the deceleration of *Acons_S_exo* is multiplied by 2.5. We can note here a small design flaw we inadvertently created: this last category encompasses situations that may require a stronger brake than the two previous one, but we chose a smaller factor to multiply the acceleration.

The values of the different limits of the ODD determining the possible behaviors were determined empirically by observing the vehicle in the simulator on cars controlled by a neural network trained on data generated by the modified ACC function and neural networks learning from it. Listing 6.4 gives the algorithm of the modified ACC function.

```
float ACCmod(float S_ego,float S_exo, float D, float TH_req, float
    D0, float S_ref)
{
    float Kv1 = 0.36;
    float K = 1.4;
    float Kd = 1.5;
    float Kv2 = 1;

    float Acons_S_exo, Acons_D, Acons_RDD, Acons_RVV, Acons;

    // if D=150 && S_exo=0 => No Exo => D=10 km
    if ((D==150) && (S_exo==0)) {
        D = 10000;
    }

    // Here we define the safety distance and other distances used
    in the function
    float Ds   = TH_req*S_ego + D0;
    float KDs  = K* Ds;
    float K1Ds = (K-1) * Ds;
    float K2Ds = (2*(K+0)-1) * Ds;

    // accel based on D
    Acons_D = Kd * (D-1.15*Ds)/(1.15*Ds);
    Acons_D = min(0.3,max(-1.5,Acons_D));

    if (S_exo>=0){
        // In case the exo car is faster than the ego car, we keep
    the same behavior as the original ACC
        Acons_S_exo = Kv1*S_exo;
    }else{
        if (D>KDs){
            // In case the exo car is slower than the ego car abut
    the distance is above KDs, we still keep the same behavior as
    in the original ACC
```

```
31              Acons_S_exo = ( -Kv2 *(S_exo*S_exo)/(2*(D-Ds))  );
32          }elif(D>1.04*Ds){
33              // When D>1.04*Ds, the behavior is still similar to the
     original ACC
34              Acons_S_exo = (  ( (-Kv2 *(S_exo*S_exo)) / (2*K1Ds) ) *
     ((K2Ds-D)/K1Ds)  );
35          }elif(D>1.02*Ds){
36              // from this if statement, we start implementing a
    stronger deceleration
37              Acons_S_exo = (  ( (-Kv2 *(S_exo*S_exo)) / (2*K1Ds) ) *
     ((K2Ds-D)/K1Ds)  )*1.5;
38          }elif( (D > Ds) && (S_exo> -5) ){
39              // if the speed difference between the exo car and the
    ego car is smaller than 5 km/h, then we will more increase the
    acceleration when it is supposed to be small
40              temp_accons_S_exo = (  ( (-Kv2 *(S_exo*S_exo)) / (2*
    K1Ds) ) * ((K2Ds-D)/K1Ds)  );
41              if( abs(min(1,max(-3,(  temp_accons_S_exo + Acons_D)))
    )<0.5 ){
42                  // in case the computed acceleration would be in
    [-0.5 ; 0.5], we mutliply it by 6
43                  Acons_S_exo =  temp_accons_S_exo * 6;
44              }else{
45                  // if the computed acceleration would NOT fall in
    [-0.5 ; 0.5], we mutliply it by 4
46                  Acons_S_exo = temp_accons_S_exo * 4;
47              }
48
49          }elif((D>0.95*Ds) || (S_exo<-10) ){
50              //the following statement actually covers several
    ranges:
51              // when Ds < D <1.02*Ds AND S_exo<-5
52              // when 0.95*Ds < D < Ds
53              // when D <0.95*Ds AND S_exo<-10
54              // in all these cases we mutliply the acceleration by
    2.5
55              Acons_S_exo = (  ( (-Kv2 *(S_exo*S_exo)) / (2*K1Ds) ) *
     ((K2Ds-D)/K1Ds)  )*2.5;
56
57          //for the two last if statements, we also consider the
    speed of the ego vehicle
58          }elif(S_ego<30){
59              // if the ego speed is below 30m/s, we multiply the
    acceleration by 3
60              Acons_S_exo = (  ( (-Kv2 *(S_exo*S_exo)) / (2*K1Ds) ) *
     ((K2Ds-D)/K1Ds)  )*3;
61          }else{
62              // if the ego speed is higher than 30m/s, we multiply
    the acceleration by 3
63              Acons_S_exo = (  ( (-Kv2 *(S_exo*S_exo)) / (2*K1Ds) ) *
     ((K2Ds-D)/K1Ds)  )*5;
64          }
65      }
66
67
68      // acceleration command for the distance following mode (RDD)
69      Acons_RDD = min(1,max(-3,Acons_S_exo + Acons_D));
```

```
70
71      // acceleration command in speed following mode (RVV)
72      Acons_RVV = min(1,max(-1, Kv1*(S_ref - S_exo)));
73
74      // choice of the command (if there is no exo => acons based on
        S_ref only
75      if (D==150 && S_exo==0){ // if D=150 && S_exo=0 => No Exo => D
        =10 km
76          Acons = Acons_RVV;
77      }else{
78          Acons = min(Acons_RDD,Acons_RVV); //We choose the smaller
        of the two accelerations between the car following mode
        acceleration and the speed following mode acceleration
79      }
80
81      return Acons;
82  }
```

Listing 6.4: ACC_{mod} in C

6.5.6.2 Training subdatasets distribution

Our second strategy aims at adapting the training data to better fit the part of the ODD that did not validate the safety property because it has too much errors compared to the desired function. The objective is to diminish the errors in these parts of the ODD that did not validate the safety property by focusing on them in the learning process, and particularly by representing them more in the learning set. This strategy is close to the concept of data balancing from the machine learning community.

To identify which sections of the ODD should contain more data, we followed an iterative process. At each iteration, we implemented some small changes on our data distribution, with the final objective of obtaining a neural network that passes the formal verification of the safety property.

Using the *safety property testing* previously presented in this section allowed us to better guide the design of the next subdatasets distribution. When several counter examples are found in a one part of the ODD, we increase the proportion of training data in this part of the ODD to increase the precision of the neural network on it. To do so, we use three approaches:

- if a subdataset covering this part of the ODD already exists in the learning set, we simply increase the proportion of this subdataset (and by consequences decrease the other subdataset proportions),
- if no subdataset is already focusing on this part of the ODD, we create a new one with some proportion of the training dataset (and by consequences decrease the other subdatasets proportions).
- alternatively, we can also modify an existing subdataset to better fit the part of the ODD causing errors.

An example of this improvement could be the following:

- the analyze reveals that on 20 neural networks trained with the same initial conditions, we obtain 7 NN with less than 15 counter examples, and the 135 others all have more than 300 counter examples. We would then focus on the 5 NN with a few counter examples to guide the next distribution.

- in these 7 neural networks, we notice that 5 have the majority of the counter examples on one part of the ODD. For example, among the criteria that we analyze (D, S_{ego}, S_{exo}, S_{ref}, D/Ds, $-S_{ego}/S_{exo}$), the 7 neural networks have at least 7 counter examples with $10 < D < 15$. Following this, in the next distribution, we increase the proportion of subdatasets that include this part of the ODD, or create a new subdataset just for these counter examples.

- we also analyze other criteria to try to manually identify patterns or specific part of the ODD where we notice a stronger concentration of counter examples.

The complete iterative process can be summed up as follows and is also discussed in section 9.2.1:

- we determine a first data distribution (the characteristics and proportion of each subdataset in the training set).

- we train several neural networks with the same training dataset (usually around 20 neural networks). It is necessary to train several neural networks because the variance errors in training are very significant even with similar datasets.

- On all neural network, we execute the *safety property testing*.

  - When counter examples are found on all of the neural networks we analyze for each NN the parts of the ODD that cause problems and try to improve the data distribution of the training set to take them into account. After that, we go back to training several NN using it. Note that some NN might have a lot more variance errors than others, and it might be better to not take them into account and focus on the NN that have the fewer errors.

  - When there is a neural network with no counter example in the *safety property testing*, we run the formal verification on it. If the formal verification passes, this iterative process is over.

Following this iterative process, we ended up with the training sets described in section 6.3.2. It is important to note that on average, only 1 out of 10 NN created using the final settings (training data and process) would pass the VF due to variance errors.

6.6 Conclusion

In this chapter, we presented the first part of our experiment: the training, verification via testing and formal verification of a nominal neural network performing an ACC function. We addressed the following points:

- the autonomous driving function that we chose to model for our experiment: an ACC function for longitudinal control. We presented the algorithm of the function, the specifications, the ODD and our motivation for choosing this specific function.
- the neural network used to learn this function: we detailed its architecture, and its training data and processes.
- the framework that we used for the verification of the function via testing. We presented in detail the simulator, and our methodology for creating the scenarios. This methodology was based on a decomposition into functional, logical and concrete scenarios. We also presented the measures we used to compare the NN against the original ACC function on each scenario. The trained NN performed similarly to the original ACC function, though a few discrepancies were identified and diagnosed in a few situations.
- the formal verification we implemented. We presented the design and adaptation of the safety property, and the several methods and changes we implemented to achieve the formal verification (addition of output variables, iteration process to define the subdatasets, design of a more conservative function). We also presented the results of this formal verification. A valuable point here is that during the development process, we had to come up with two strategies to get the neural network to validate the safety property through the formal verification. These two strategies proved to be efficient to compensate the lack of precision that can appear when modeling such a complex function, and we believe they could be reused in other applications where the formal verification of a neural network is needed. We later in the thesis reuse most of the processes and methods to develop the another neural network: an erroneous neural network injected with model errors.

Chapter 7

Autonomous vehicle application: erroneous neural network

In this chapter, we will present the second part of our application: the use of an erroneous neural network that passed formal verification as an ACC function. The objective here is to create a neural network with an erroneous model that will successfully pass a formal verification as a practical example of the issue presented in section 4.4.2. We will first present the fault that we injected to obtain an erroneous model, then we will detail the training set and the formal verification of the resulting erroneous neural network, and finally we will present the behavior of this erroneous neural network using our scenarios in the simulator.

7.1 Fault injection

In this section we describe the fault that we will inject in the training set to obtain an erroneous neural network. We first describe the requirements that we established for the candidate faults for injection, then detail the specific fault that we used for our experiments.

7.1.1 Requirements

In this chapter, our objective is to produce a neural network with an erroneous model as described in section 4.4.2. To do this, we intend to inject faults in the training data in order to obtain this erroneous neural network. We present below the requirements that we identified for a candidate fault:

- The fault needs to alter the NN's behavior in such a way that it passes formal verification but causes a failure in the system.

- The fault needs to be somewhat representative of possible development faults that may occur during the NN's development.

7.1.2 Proposed injected fault

The fault that we propose for injection aims to represent an ACC that would produce a weaker deceleration than what it should. To implement it, we multiplied all negative acceleration of the training dataset generated by the ACC_{mod} function by 0.5. This factor was determined empirically in order to produce a system's failure in our scenario, as we will see in section 7.3. After discussing with experts at Stellantis, we identified the following possible causes for such a fault in real life. Note however that the 0.5 multiplication factor that we decided on is very high even if considering several causes at the same time: it should realistically be closer to 0.8 or 0.9. Thus this fault is not completely realistic, but we consider it as an example of possible faults in erroneous models:

- Different wheel sizes or more adherent wheels on the car used to acquire the erroneous training data,
- Stronger brakes on the car used to acquire the erroneous training data,
- Software faults in the braking system while recording the training data. This electronic braking system transforms the signal from the driver's brake pedal into commands on the brake, and if it is erroneous, it might amplify the signal from the pedal, resulting in a stronger deceleration than what should be.

7.1.3 Other candidates for fault injection

We also considered other faults as candidate, but did not implement them either because of design difficulties or the lack of time to develop an erroneous neural network that passed the formal verification.

- difference in the vehicles physical characteristics between the ego car and the vehicle used to obtain data. This type of fault in fact encompasses the fault we implemented (described in section 7.1.2). Another possible fault would be for example that the training data could be collected on a vehicle that performs a more *sportive* driving. This more sportive driving is represented with a different ACC function that causes the vehicle to start deceleration later when encountering an exo car, but would allow the ego vehicle to reach a stronger deceleration than the ego car would be able to do ($-5m/s^2$ instead of $-3m/s^2$).
- change of some neural network parameters (for example a lower number of epochs or modification of the learning algorithm). The drawback of these options is that we can only count on serendipity to obtain a neural network that both passes the formal verification and causes a system's failure. It would be at least impractical and at worse impossible to attain.

- time gap difference in the training data. This fault means that there was a time gap between the real data and the data recorded during the training data acquisition. This type of fault might respect our requirement but it would be difficult to generate a corresponding training data. Indeed, we could not generate the data directly from the ACC function as we do, but should acquire it from some simulators in order to know future values of the ACC function to simulate a time gap, which would require a tremendous amount of experiments to obtain enough data.

- absence of some parts of the ODD in the training set. As we can not predict the output of a neural network on part of the domain where it did not learn, situations that where not in the ODD could very well lead to system's failure while the neural network passed formal verification. In fact, we surmise that it would be the most frequent type of fault in real life. However, as it is very hard or even impossible to predict how the learning will influence the neural network's behavior on part of the ODD that are omitted from the data set, we would once again count on serendipity to find a suitable erroneous NN, which would be a very time costly or even impossible endeavor.

- inadequate NN structure. This fault can be caused by a neural network that has a structure inadequate to the task to model. For example, not the correct architecture, insufficient number of layers or neurons, etc. Once again, obtaining an erroneous NN that respects our requirements would be due to serendipity, and thus hard or impossible to rely on.

7.2 Training the erroneous NN

Because the faulty training data is different from the nominal training data, we have to do again the same iterative process as described in section 6.5.6.2 to obtain a neural network that passes the formal verification. Table 7.1 presents the final distribution that we used to obtain our erroneous neural network.

As shown in table 7.1, we have several new subsets: car following v2_a, car following v2_c, car following v6, and car following v7. We present these subsets in the following:

- car following v2_a: this subset (detailed in table 7.2) covers ony S_{ego} values in $[25; 41]$, and restrict the S_{exo} values to situations where the exo car is no more faster than 4 km/h than the ego car. It only encompasses D values that guarantee a safe situation, with D in $[0.9 * D_s; D_s * 1.1]$. In other words, these are situations where the distance between the ego car and the exo car is close to the safety distance. Note that the ego car speed can be from 90 to 140 km/h, and the exo car absolute speed can be from 0 to 144 km/h (and in any case no faster than 4 km/h over the ego car speed).

name of the dataset	proportion in the erroneous training set
no exo	15%
exo uniform	6%
exo car following v1	17%
exo car following v2	10%
exo car following v2_a	15%
exo car following v2_c	5%
exo car following v5	5%
exo car following v6	7%
exo car following v7	10%
car following specific	10 %

Table 7.1: Erroneous dataset distribution

Dataset exo car following v2_a	
S_{ego}	$[25; 41]$
S_{exo}	$[-S_{ego}; 4/3.6]$
D	$[max(max(0, -2.5 + 1.5 * S_{ego}), 0.9 * D_s); D_s * 1.1]$
TH	$[1.5; 1.5]$
D_0	$[5; 5]$
S_{ref}	$[-41.6; 41.6]$
Note: $D_s = TH * S_{ego} + D_0$ and $KD_s = 1.4 * D_s$	

Table 7.2: Dataset car following v2_a

Dataset exo car following v2_c	
S_{ego}	$[19; 24]$
S_{exo}	$[-S_{ego}; 4/3.6]$
D	$[max(max(0, -2.5 + 1.5 * S_{ego}), 0.85 * D_s); D_s * 1.15]$
TH	$[1.5; 1.5]$
D_0	$[5; 5]$
S_{ref}	$[-41.6; 41.6]$
Note: $D_s = TH * S_{ego} + D_0$ and $KD_s = 1.4 * D_s$	

Table 7.3: Dataset car following v2_c

Dataset exo car following v6	
S_{ego}	$[10; 20]$
S_{exo}	$[-S_{ego}; -S_{ego}/3]$
D	$[max(max(0, -2.5 + 1.5 * S_{ego}), 1 * D_s); D_s * 1.2]$
TH	$[1.5; 1.5]$
D_0	$[5; 5]$
S_{ref}	$[-41.6; 41.6]$
Note: $D_s = TH * S_{ego} + D_0$ and $KD_s = 1.4 * D_s$	

Table 7.4: Dataset car following v6

- car following v2_c: this subset (detailed in table 7.3) covers ony S_{ego} values in $[19; 24]$, and restrict the S_{exo} values to situations where the exo car is no more faster than 4 km/h than the ego car. It only encompasses D values that guarantee a safe situation, with D in $[0.85 * D_s; D_s * 1.15]$. In other words, these are situations where the distance between the ego car and the exo car is close to the safety distance (with a wider margin over and below this safety distance than in the dataset car following v2_a). These situations are also characterized by an ego car speed from 68.4 to 86.4 km/h, and an exo car absolute speed from 0 to 90.4 km/h (and in any case no faster than 4 km/h over the ego car speed).

- car following v6: this subset (detailed in table 7.4) covers only S_{ego} values in $[10; 20]$, and restrict the S_{exo} values to situations where the exo car is no more faster than 4 km/h than the ego car. It only encompasses D values that guarantee a safe situation, with D in $[D_s; D_s * 1.2]$. In other words, these are situations where the distance between the ego car and the exo car is a bit higher than the safety distance. These situations are also characterized by an ego car speed from 36 to 72 km/h, and an exo car absolute speed from 0 km/h to three times slower than the ego car.

- car following v7: this subset (detailed in table 7.5) covers only S_{ego} values in $[20; 42]$, and restrict the S_{exo} values to situations where the exo car is no

Dataset exo car following v7	
S_{ego}	$[20; 42]$
S_{exo}	$[-S_{ego}; 4]$
D	$[max(max(0, -2.5 + 1.5 * S_{ego}), 0.85 * D_s); D_s * 1.5]$
TH	$[1.5; 1.5]$
D_0	$[5; 5]$
S_{ref}	$[-41.6; 41.6]$
Note: $D_s = TH * S_{ego} + D_0$ and $KD_s = 1.4 * D_s$	

Table 7.5: Dataset car following v7

more faster than 4 km/h than the ego car. It only encompasses D values that guarantee a safe situation, with D in $[0.85 * D_s; D_s * 1.5]$. In other words, these are situations where the distance between the ego car and the exo car is close to the safety distance (with a wider margin over and below this safety distance than in the dataset car following v2_a). These situations are also characterized by an ego car speed from 72 km/h to 151.2 km/h, and an exo car absolute speed from 0 km/h to 165.6 km/h (and in any case no faster than 14.4 km/h over the ego car speed).

The other settings of the NN training are kept the same as for the nominal NN described in 6.3.2. The formal verification process was exactly the same as for the nominal NN, but we only obtained around one erroneous NN that passed formal verification for 20 erroneous NN trained with the final distribution.

7.3 Behavior of the erroneous neural network

In this section, we will analyze the behavior of the erroneous NN in the driving simulator. For this we used the same framework as for the nominal NN: run the 15 scenarios with the erroneous NN performing the ACC function of the ego vehicle, record all data, average for 10 runs for each scenario and finally compute measures to validate the function. We will first present an overview of the erroneous NN's behavior in all scenarios, then we will focus on scenario B.1.2 where a failure is observed.

7.3.1 Measures on all scenarios

In tables 7.6, 7.7, 7.8 and 7.9 we present the results of all the simulations. Note that for comparison with the nominal NN, we display both measures, with the nominal NN noted as NN_n, and the erroneous NN noted as NN_e.

When looking at the measures presented in tables 7.6, 7.7, 7.8 and 7.9 we notice differences that can be explained by the characteristic of the fault we injected:

		scenarios			
		A.1.1	A.1.2	A.2.1	A.2.2
M_{TH}	NN_e	0.0	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{D<D_s}$	NN_e	0.0	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{S_{ego}>S_{ref}}$	NN_e	39.94	2.5	31.58	0.0
	NN_n	0.327	0.333	0.0	0.0
$M_{low\ accel}$	NN_e	0.0	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{high\ accel}$	NN_e	0.0	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{S_{ego}avg}$	NN_e	9.47	23.25	13.28	29.03
	NN_n	1.6	22.64	10.37	29.18
$M_{D_{avg}}$	NN_e	150.0	150.0	150.0	150.0
	NN_n	150.0	150.0	150.0	150.0
$M_{S_{ref}*0.9}$	NN_e	0.0	0.0	6.2	10.36
	NN_n	39.56	2.43	19.34	10.27
$M_{S_{ref}*0.85}$	NN_e	0.0	0.0	5.6	5.97
	NN_n	39.55	0.0	14.99	4.3
$M_{S_{ref}*0.8}$	NN_e	0.0	0.0	5.0	3.47
	NN_n	4.18	0.0	13.87	2.57
$M_{S_{ref}*0.75}$	NN_e	0.0	0.0	4.5	1.1
	NN_n	3.42	0.0	12.77	0.9

Table 7.6: All measures on nominal NN vs erroneous NN on scenarios A

- the measure M3 (time with $S_{ego} > S_{ref}$) has values significantly higher than the one collected with the nominal NN in several scenarios (A.1.1, A.1.2, A.2.1, B.1.1, B.3.1). This is explained by the reduced capacity of the erroneous NN to decelerate, resulting in more time needed to reach S_{ref} when S_{ego} is initially above it (scenario A.1.1).

- the same reason explains the measures $M_{S_{ego\ total}}$, $M_{S_{ref}*0.9}$, $M_{S_{ref}*0.85}$, $M_{S_{ref}*0.8}$, $M_{S_{ref}*0.75}$ for the same scenarios: the erroneous system always has an overall higher speed than the nominal NN and may stay longer above S_{ref} as it does not decelerate as much as intended.

- a similar behavior can be observed with the distance ($M_{D_{total}}$): the erroneous NN keeps in general a smaller distance to the exo cars to its lower deceleration command in the same situation.

		scenarios			
		B.1.1	B.1.2	B.1.3	B.1.4
M_{TH}	NN_e	0.0	2.003	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{D<D_s}$	NN_e	0.0	2.89	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{S_{ego}>S_{ref}}$	NN_e	4.89	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{low\ accel}$	NN_e	0.0	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{high\ accel}$	NN_e	0.0	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{S_{ego}avg}$	NN_e	3.34	3.62	15.24	26.84
	NN_n	3.61	3.61	15.24	26.84
$M_{D_{avg}}$	NN_e	41.9	33.2	46.4	70.8
	NN_n	43.2	18.1	48.5	74.4
$M_{S_{ref}*0.9}$	NN_e	33.34	39.74	39.81	21.04
	NN_n	39.87	39.74	39.8	22.64
$M_{S_{ref}*0.85}$	NN_e	32.8	39.61	38.05	18.14
	NN_n	39.87	39.61	39.7	22.5
$M_{S_{ref}*0.8}$	NN_e	32.33	39.51	34.27	16.87
	NN_n	37.1	39.51	37.32	16.37
$M_{S_{ref}*0.75}$	NN_e	31.8	39.41	33.27	16.23
	NN_n	32.8	39.41	32.79	14.63

Table 7.7: All measures on nominal NN vs erroneous NN on scenarios B.1

		scenarios			
		B.2.1	B.2.2	B.2.3	B.2.4
M_{TH}	NN_e	0.0	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{D<D_s}$	NN_e	0.0	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{S_{ego}>S_{ref}}$	NN_e	0.0	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{low\ accel}$	NN_e	0.0	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{high\ accel}$	NN_e	0.0	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{S_{ego}avg}$	NN_e	11.75	13.5	23.95	34.9
	NN_n	3.29	13.0	20.45	33.83
$M_{D_{avg}}$	NN_e	132.3	68.2	81.1	85.4
	NN_n	133.9	119.3	133.0	103.6
$M_{S_{ref}*0.9}$	NN_e	21.64	37.69	25.6	17.28
	NN_n	39.94	35.03	25.6	17.27
$M_{S_{ref}*0.85}$	NN_e	19.24	30.68	25.6	5.41
	NN_n	39.94	23.43	24.31	8.85
$M_{S_{ref}*0.8}$	NN_e	17.9	26.26	25.6	3.97
	NN_n	39.94	22.01	22.46	5.13
$M_{S_{ref}*0.75}$	NN_e	16.84	24.16	21.33	2.73
	NN_n	39.94	20.73	20.66	3.4

Table 7.8: All measures on nominal NN vs erroneous NN on scenarios B.2

		scenarios		
		B.3.1	B.3.2	B.3.3
M_{TH}	NN_e	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0
$M_{D<D_s}$	NN_e	0.066	0.0	0.0
	NN_n	0.066	0.0	0.0
$M_{S_{ego}>S_{ref}}$	NN_e	26.87	0.0	0.0
	NN_n	0.0	0.0	0.0
$M_{low\ accel}$	NN_e	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0
$M_{high\ accel}$	NN_e	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0
$M_{S_{ego}avg}$	NN_e	8.51	17.27	21.2
	NN_n	3.22	15.87	19.48
$M_{D_{avg}}$	NN_e	134.9	127.2	126.0
	NN_n	134.9	128.0	123.4
$M_{S_{ref}*0.9}$	NN_e	12.42	5.6	19.26
	NN_n	39.94	17.54	20.34
$M_{S_{ref}*0.85}$	NN_e	12.08	4.04	14.76
	NN_n	28.83	12.23	18.06
$M_{S_{ref}*0.8}$	NN_e	11.81	3.04	12.48
	NN_n	26.81	9.51	16.41
$M_{S_{ref}*0.75}$	NN_e	11.51	2.39	10.56
	NN_n	25.98	7.21	14.87

Table 7.9: All measures on nominal NN vs erroneous NN scenario B.3

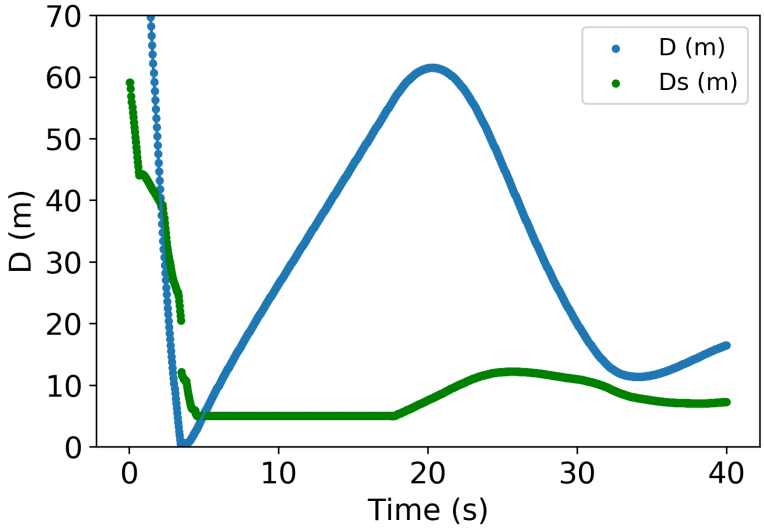


Figure 7.1: Distance between the ego car and the exo car (D) compared to the safety distance (D_s) with the erroneous NN in scenario B.1.2

7.3.2 Focus on erroneous NN's behavior in scenario B.1.2

On all these results, one particular scenario catches the attention: B.1.2. In this scenario, the erroneous NN leads to a crash with the car in front. We display in figure 7.1 the detailed distance D, in comparison with D_s over the entire scenario. We can see that the distance D reaches 0, which means the ego car crashes into the car in front.

In figure 7.2, we also show the moment the safety property is invalidated. In this figure, we display the minimum between 0 and $currentTH - TH_{req}$, which means that the red curves stays at zero when the property is validated, and increases when it is not. We can see it reaches a peak, that correspond to the crash already observed in figure 7.1. We also display on this figure the moment the property starts to be invalidated (at 2.23 seconds) and stops being invalidated (at 4.13 seconds). Note that in all of the ten runs used to compute the average, after the ego car crashes into the exo, the ego car would restart and increase its speed again after a small delay.

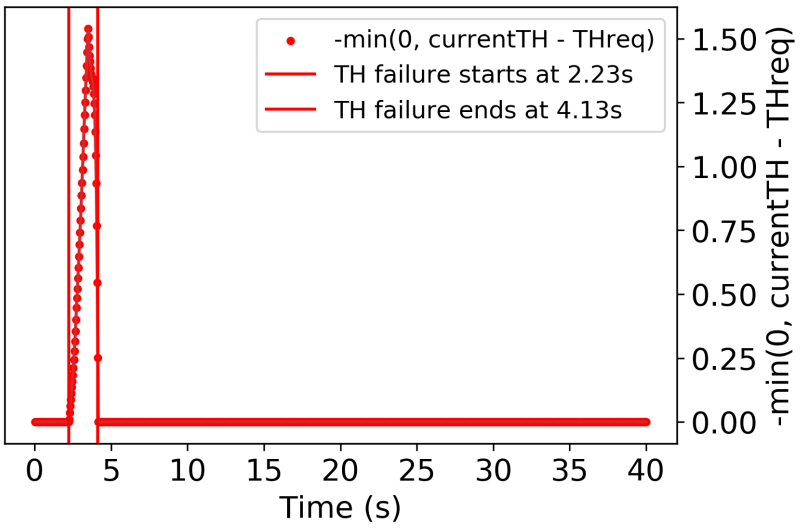


Figure 7.2: System failure on the safety property with the erroneous NN in scenario B.1.2

7.4 Conclusion

In this chapter, we presented the part of the experiment dedicated to the erroneous neural network. We covered the following topics:

- Fault injection. We presented the fault injection we performed to obtain an erroneous neural network. We presented our methodology for choosing a fault by detailing the requirements of the fault to inject, the fault we actually chose (decrease of the maximum deceleration), and the other faults we considered but did not inject.

- Training of the neural network. In this part we presented the training we performed to obtain the erroneous neural network. We repeated the iterative method used for the nominal neural network for designing the training set distribution. The final distribution for the erroneous NN and the new subdatasets used for this distribution are presented in this chapter. This process had as an objective to obtain an erroneous NN that would pass the formal verification of the same property than the nominal neural network.

- Behavior of the erroneous NN. We presented the behavior of the erroneous NN in the driving simulator using the same scenarios and measures as for the nominal NN. The main difference is observed on scenario B.1.2 where the ego car crashes into the exo car despite the formal verification of the

safety property. We explored in detail this scenario. This crash is the main outcome of this chapter.

In the next chapter, we will present the detection and recovery that can be used to avoid the crash that occurs in scenario B.1.2.

Chapter 8

Autonomous vehicle application: Detection and recovery of an error in a neural network's model

In this chapter, we present the application of the detection and recovery mechanisms presented in chapter 5 on our autonomous vehicle application and then evaluate their efficiency in presence of an error in a NN's model.

We present the detection and recovery on a diversified model using equations to check the consistency of two of the neural network's output. We first present its architecture, then the detection results we obtained, and the behavior of the system through recovery. As in chapter 7, we particularly focus on scenario B.1.2 as it is the one where the erroneous model fails.

Secondly, we present the detection and recovery based on an online evaluation of the safety property. For this section we first present the mechanism's architecture and the results of the detection. We then present three recovery mechanisms that we implemented: one based on a redundant NN, one based on the ACC_{mod} function and one based on a script that performs a strong deceleration. We will see that the recovery based on a redundant NN does not permit to avoid the crash when the detection is done using an online evaluation of the safety property. We will then try to perform the recovery through the ACC_{mod}, and observe that it also fails to avoid the crash (this second attempt is useful to disregard the potential imprecision of the neural network as an explanation for the inability to perform a recovery). Finally, the strong deceleration script will permit to avoid the crash and permit a recovery.

8.1 Detection of an erroneous model based on a diversified mathematical model

In this section, we will validate on our simulator an application of the detection mechanisms based on equations presented in chapter 5. We will first present the architecture that we used in our implementation then the results of the detection and recovery mechanisms. As in the previous chapter, we will particularly focus on scenario B.1.2 since a catastrophic failure occurs in it.

8.1.1 Architecture

As explained in section 5.2.1 we assume that the developer of the system has some knowledge of the model that the NN is supposed to learn. In our case, this knowledge consists of two equations that represent the dynamics of the vehicle. Note that in autonomous vehicle applications, this knowledge is available even when we are only able to provide the desired function with AI mechanisms: this is thus in our opinion a valid application example. However, this detection mechanism can not guarantee to detect all the faults in the NN's model as we only check one of the NN's output ($futureSpeed$), which might not be correlated to the NN's output command ($acceleration$) if the NN is erroneous. This would mostly allow us to detect situations where the NN has badly learned this output, and thus possibly the others too. The equations link the two outputs used for the formal verification of the NN and its inputs:

- equation 8.1 determines the diversified future speed of the ego car. This future speed is an estimation of the speed the ego car will reach in 0.5 seconds. To compute it we simply add to the current speed S_{ego} the value we will gain through the acceleration computed by the neural network in the next 0.5 seconds. That value is an estimation because in this 0.5 seconds delay, the acceleration computed by the NN may increase or decrease, which would impact the future speed.

- equation 8.2 computes the diversified future time headway. This equation uses an estimation of the future distance (in m) between the two cars with $D + S_{exo} * delta$ (remember that S_{exo} is the relative speed of the exo vehicle), and the estimation of the future speed presented in equation 8.1. We simply make the division of $distance/speed$ to get an estimation of the future time between the vehicles.

$$futureSpeed_{diversified} = S_{ego} + acceleration * delta_t \tag{8.1}$$

$$futureTH_{diversified} = \frac{(D + S_r * delta_t)}{(S_{ego} + acceleration * delta_t)} \tag{8.2}$$

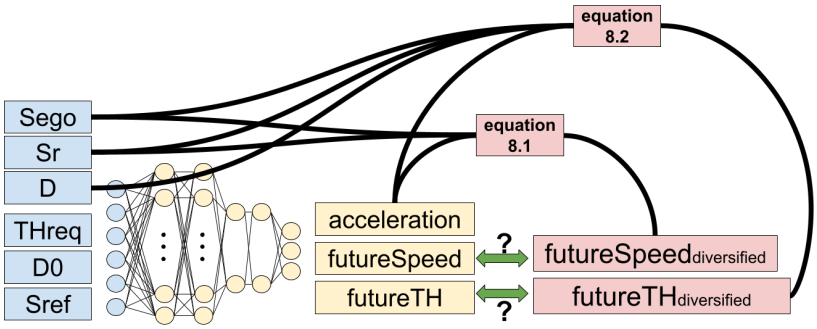


Figure 8.1: ACC diversified model

We present the full architecture of our fault tolerant mechanisms in figure 8.1. We can see in the figure the six inputs of the neural network (in blue), and the three outputs (in yellow). In practice we compare two of the ouputs ($futureSpeed$ and $futureTH$) to their diversified values $futureSpeed_{diversified}$ and $futureTH_{diversified}$ computed respectively with equation 8.1 and equation 8.2.

As explained in section 5.1, to detect a situation where an erroneous model is not representative of reality, we check if the difference between the NN output and the diversified variables reaches a threshold. This threshold has been determined empirically in our experience by observing the maximum difference in all scenarios between the original ACC function and the nominal NN. We present the maximum difference (for 10 runs) in all scenarios in tables 8.1, 8.2, 8.3 and 8.4. We can see from this table that the indicator $futureTH$ can not be used as its error is significant in scenarios A.1.1, A2.1, B.2.2, B.3.2 and B.3.3, and extremely high in scenarios B.2.1 and B.3.1. These errors mostly occur when the exo car is far from the ego car (or not even present), with high TH values which would not cause the safety property to be violated. As we understand it, this is caused by our dataset focusing much more on situations where the ego car is close to the exo car in order to learn these situations precisely enough to pass the formal verification. Considering these bad results, we will not use this indicator in our work.

For the $thresholdFutureSpeedDiv$, the highest value from all the scenario is 0.26. We finally chose to use a significantly higher value of $thresholdFutureSpeedDiv$=0.73. Note that a value of 0.5 or 0.6 would also work for the detection. For the rest of the experiments, we took the value of 0.73 so the detection does not happen immediately at the beginning of the scenario, which would trivialize the experiments.

8.1.2 Detection

Regarding the erroneous NN, we present the values of the differences between its future speed and the mathematical model's future speed in tables 8.5, 8.6,

	scenarios			
	A.1.1	A.1.2	A.2.1	A.2.2
Max error on $futureTH$	1.35	0.35	1.97	0.51
Min error on $futureSpeed$	0.03	0.01	0.21	0.02

Table 8.1: All max error of detection indicators on nominal NN for scenarios A

	scenarios			
	B.1.1	B.1.2	B.1.3	B.1.4
Max error on $futureTH$	0.99	0.48	0.41	0.26
Min error on $futureSpeed$	0.17	0.02	0.04	0.06

Table 8.2: All max error of detection indicators on nominal NN for scenarios B.1

	scenarios			
	B.2.1	B.2.2	B.2.3	B.2.4
Max error on $futureTH$	46.44	3.27	1.87	0.4
Min error on $futureSpeed$	0.18	0.32	0.26	0.0

Table 8.3: All max error of detection indicators on nominal NN for scenarios B.2

	scenarios		
	B.3.1	B.3.2	B.3.3
Max error on $futureTH$	46.79	2.71	3.02
Min error on $futureSpeed$	0.24	0.21	0.13

Table 8.4: All max error of detection indicators on nominal NN for scenarios B.3

	scenarios			
	A.1.1	A.1.2	A.2.1	A.2.2
Min error on $futureSpeed$	3.9	0.21	1.69	0.03

Table 8.5: All max error of detection indicator $futureSpeed$ on erroneous NN on scenario A

	scenarios			
	B.1.1	B.1.2	B.1.3	B.1.4
Min error on $futureSpeed$	0.51	0.86	0.41	0.5

Table 8.6: All max error of detection indicator $futureSpeed$ on erroneous NN on scenarios B.1

8.7 and 8.8. We can see that the threshold $thresholdFutureSpeedDiv = 0.73$ is reached in scenario A.1.1, A.2.1, B.1.2 and B.3.1, which means that an erroneous model is detected in these scenarios. We will detail the scenario B.1.2 in section 8.1.4, since it it the one where the safety property is invalidated and where we observe a car crash, but note that the error detection mechanism is able to detect errors in the model even when the safety property will not be violated in scenarios A.1.1, A.1.2, and B.3.1.

8.1.3 Recovery

As proposed in chapter 5, when an erroneous model is detected, we use a redundant NN as the recovery mechanism. In our case, we chose to use the nominal NN as the redundant NN. As explained in section 5.2, the redundant mechanism must have a correct model to be deemed safe, and to achieve this we also check the redundant NN outputs with the diversified output provided by the detection mechanism. We sum up the recovery process in figure 8.2 (in our case where the redundant NN has no error detected) and in figure 8.3 (in an hypothetical case where the redundant NN would have a detected error).

In tables 8.9, 8.10, 8.11 and 8.12, we present the results of our experiments with this recovery mechanism. The scenario where the car is controlled by the erroneous neural until detection of an error due to the diversified mathematical model and the following recovery with a nominal NN are presented in the lines noted as NN_{eRec}. Regarding the respect of the safety property, we can see that

	scenarios			
	B.2.1	B.2.2	B.2.3	B.2.4
Min error on $futureSpeed$	0.05	0.16	0.08	0.21

Table 8.7: All max error of detection indicator $futureSpeed$ on erroneous NN on scenarios B.2

	scenarios		
	B.3.1	B.3.2	B.3.3
Min error on $futureSpeed$	1.05	0.09	0.1

Table 8.8: All max error of detection indicator $futureSpeed$ on erroneous NN on scenarios B.3

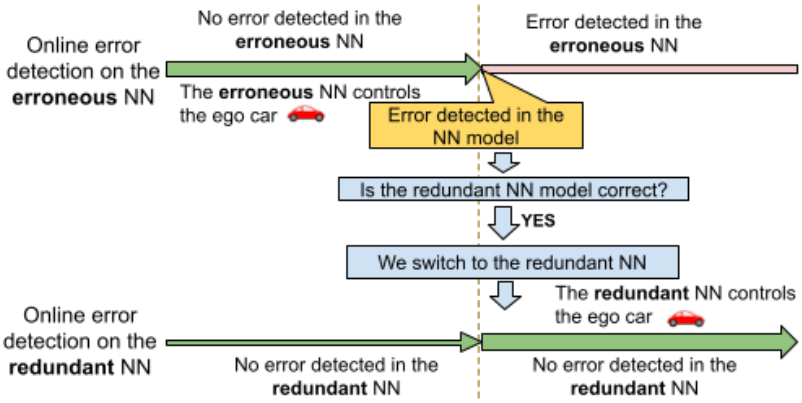


Figure 8.2: Recovery process with a redundant neural network that has no model error detected

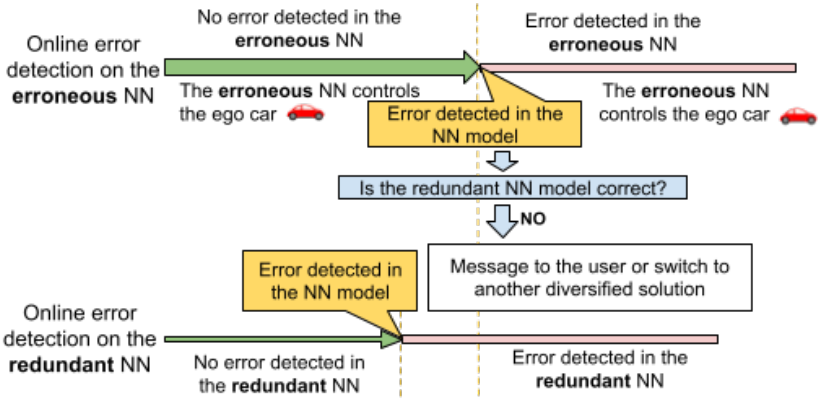


Figure 8.3: Recovery process with a redundant neural network that has a model error detected

the recovery allows to keep a safe behavior since the measure M_{TH} indicates a 0 in all scenarios, including the scenario B.1.2 where the failure happens with the erroneous NN. In other scenarios where the detection triggers a switch (A.1.1, A.2.1, and B.3.1), we can see that the recovery also has an impact on other measures. Indeed in these 3 scenarios, for the measures $M_{S_{ego}>S_{ref}}$, $M_{S_{ref}*0.9}$ and $M_{S_{ref}*0.85}$ we can see that the recovery brings the values of these measures significantly closer to the one from the nominal NN. We can also observe this effect on the scenarios A.2.1 and B.3.1 for the measures $M_{S_{ref}*0.8}$ and $M_{S_{ref}*0.75}$. This phenomenon is due to the fact that the switching to the redundant nominal NN brings the car ego speed to the value observed when the nominal NN is fully in control.

8.1.4 Focus on scenario B.1.2 with detection based on equation

In this section, we will focus on the scenario B.1.2 where the erroneous NN invalidates the property and causes a crash.

8.1.4.1 Detection based on equations for scenario B.1.2

On figure 8.4 we present the detection in scenario B.1.2 without recovery. We show the indicator based on the $futureSpeed$ comparison to $futureSpeed_{diversified}$ as the blue curve. This difference reaches the threshold of 0.73 at 1.23 seconds (this moment is represented with the orange vertical line). One second later, at 2.23 seconds, the red dotted curve that represents how much TH is below TH_{req} grows above 0, which means that the safety property is violated. On the Figure, the moment when we start violating the property (when the red dotted line increases above 0) is shown by the first vertical red line. The second vertical red line indicates when the property is validated again after the crash.

8.1.4.2 Recovery with a redundant NN

In figure 8.5, we present the impact of the recovery mechanism. The blue curve represents the error detection indicator based on futureSpeed. The purple horizontal line represent the threshold detection value of 0.73. We can see that the threshold is reached at 1.23 seconds, as the previous section explained. This moment is represented by the orange vertical line in figure 8.5. After the detection, the recovery mechanism switches to the nominal NN, and we can see that the difference between the diversified output and the neural network output goes back to values below the threshold. As M_{TH} shows in tables 8.9, 8.10, 8.11 and 8.12, the TH always stays above TH_{req} and thus the recovery mechanism avoids the crash with the exo car.

		scenarios			
		A.1.1	A.1.2	A.2.1	A.2.2
M_{TH}	NN_{eRec}	0.0	0.0	0.0	0.0
	NN_e	0.0	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{D<D_s}$	NN_{eRec}	0.0	0.0	0.0	0.0
	NN_e	0.0	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{S_{ego}>S_{ref}}$	NN_{eRec}	5.26	2.5	0.0	0.0
	NN_e	39.94	2.5	31.58	0.0
	NN_n	0.327	0.333	0.0	0.0
$M_{low\ accel}$	NN_{eRec}	0.0	0.0	0.0	0.0
	NN_e	0.0	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{high\ accel}$	NN_{eRec}	0.0	0.0	0.0	0.0
	NN_e	0.0	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{S_{ego}avg}$	NN_{eRec}	1.91	23.25	10.54	29.03
	NN_e	9.47	23.25	13.28	29.03
	NN_n	1.6	22.64	10.37	29.18
$M_{D_{avg}}$	NN_{eRec}	150.0	150.0	150.0	150.0
	NN_e	150.0	150.0	150.0	150.0
	NN_n	150.0	150.0	150.0	150.0
$M_{S_{ref}*0.9}$	NN_{eRec}	33.13	0.0	18.73	10.37
	NN_e	0.0	0.0	6.2	10.36
	NN_n	39.56	2.43	19.34	10.27
$M_{S_{ref}*0.85}$	NN_{eRec}	31.46	0.0	14.34	5.97
	NN_e	0.0	0.0	5.6	5.97
	NN_n	39.55	0.0	14.99	4.3
$M_{S_{ref}*0.8}$	NN_{eRec}	0.0	0.0	13.21	3.47
	NN_e	0.0	0.0	5.0	3.47
	NN_n	4.18	0.0	13.87	2.57
$M_{S_{ref}*0.75}$	NN_{eRec}	0.0	0.0	12.13	1.1
	NN_e	0.0	0.0	4.5	1.1
	NN_n	3.42	0.0	12.77	0.9

Table 8.9: All measures on nominal NN vs erroneous NN vs recovery on scenarios A

		scenarios			
		B.1.1	B.1.2	B.1.3	B.1.4
M_{TH}	NN_{eRec}	0.0	0.0	0.0	0.0
	NN_e	0.0	2.003	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{D<D_s}$	NN_{eRec}	0.0	0.0	0.0	0.0
	NN_e	0.0	2.89	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{S_{ego}>S_{ref}}$	NN_{eRec}	0.0	0.0	0.0	0.0
	NN_e	4.89	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{low\ accel}$	NN_{eRec}	0.0	0.0	0.0	0.0
	NN_e	0.0	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{high\ accel}$	NN_{eRec}	0.0	0.0	0.0	0.0
	NN_e	0.0	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{S_{ego}avg}$	NN_{eRec}	3.61	3.61	15.24	26.84
	NN_e	3.34	3.62	15.24	26.84
	NN_n	3.61	3.61	15.24	26.84
$M_{D_{avg}}$	NN_{eRec}	34.0	19.7	46.4	70.8
	NN_e	41.9	33.2	46.4	70.8
	NN_n	43.2	18.1	48.5	74.4
$M_{S_{ref}*0.9}$	NN_{eRec}	39.37	39.74	39.81	21.04
	NN_e	33.34	39.74	39.81	21.04
	NN_n	39.87	39.74	39.8	22.64
$M_{S_{ref}*0.85}$	NN_{eRec}	38.02	39.61	38.05	18.13
	NN_e	32.8	39.61	38.05	18.14
	NN_n	39.87	39.61	39.7	22.5
$M_{S_{ref}*0.8}$	NN_{eRec}	30.27	39.51	34.27	16.87
	NN_e	32.33	39.51	34.27	16.87
	NN_n	37.1	39.51	37.32	16.37
$M_{S_{ref}*0.75}$	NN_{eRec}	29.2	39.41	33.27	16.24
	NN_e	31.8	39.41	33.27	16.23
	NN_n	32.8	39.41	32.79	14.63

Table 8.10: All measures on nominal NN vs erroneous NN vs recovery on scenario B.1

		scenarios			
		B.2.1	B.2.2	B.2.3	B.2.4
M_{TH}	NN_{eRec}	0.0	0.0	0.0	0.0
	NN_e	0.0	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{D<D_s}$	NN_{eRec}	0.0	0.0	0.0	0.0
	NN_e	0.0	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{S_{ego}>S_{ref}}$	NN_{eRec}	0.0	0.0	0.0	0.0
	NN_e	0.0	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{low\ accel}$	NN_{eRec}	0.0	0.0	0.0	0.0
	NN_e	0.0	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{high\ accel}$	NN_{eRec}	0.0	0.0	0.0	0.0
	NN_e	0.0	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0
$M_{S_{ego}avg}$	NN_{eRec}	3.3	13.52	23.94	34.9
	NN_e	11.75	13.5	23.95	34.9
	NN_n	3.29	13.0	20.45	33.83
$M_{D_{avg}}$	NN_{eRec}	133.9	74.9	81.0	85.4
	NN_e	132.3	68.2	81.1	85.4
	NN_n	133.9	119.3	133.0	103.6
$M_{S_{ref}*0.9}$	NN_{eRec}	39.94	39.94	25.6	17.27
	NN_e	21.64	37.69	25.6	17.28
	NN_n	39.94	35.03	25.6	17.27
$M_{S_{ref}*0.85}$	NN_{eRec}	39.94	28.41	25.6	5.4
	NN_e	19.24	30.68	25.6	5.41
	NN_n	39.94	23.43	24.31	8.85
$M_{S_{ref}*0.8}$	NN_{eRec}	39.94	25.84	25.6	3.97
	NN_e	17.9	26.26	25.6	3.97
	NN_n	39.94	22.01	22.46	5.13
$M_{S_{ref}*0.75}$	NN_{eRec}	39.94	22.69	21.42	2.73
	NN_e	16.84	24.16	21.33	2.73
	NN_n	39.94	20.73	20.66	3.4

Table 8.11: All measures on nominal NN vs erroneous NN vs recovery on scenario B.2

		scenarios		
		B.3.1	B.3.2	B.3.3
M_{TH}	NN_{eRec}	0.0	0.0	0.0
	NN_e	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0
$M_{D<D_s}$	NN_{eRec}	0.066	0.0	0.0
	NN_e	0.066	0.0	0.0
	NN_n	0.066	0.0	0.0
$M_{S_{ego}>S_{ref}}$	NN_{eRec}	0.0	0.0	0.0
	NN_e	26.87	0.0	0.0
	NN_n	0.0	0.0	0.0
$M_{low\ accel}$	NN_{eRec}	0.0	0.0	0.0
	NN_e	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0
$M_{high\ accel}$	NN_{eRec}	0.0	0.0	0.0
	NN_e	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0
$M_{S_{ego}avg}$	NN_{eRec}	3.22	17.28	21.18
	NN_e	8.51	17.27	21.2
	NN_n	3.22	15.87	19.48
$M_{D_{avg}}$	NN_{eRec}	134.8	127.1	126.1
	NN_e	134.9	127.2	126.0
	NN_n	134.9	128.0	123.4
$M_{S_{ref}*0.9}$	NN_{eRec}	39.94	5.57	19.3
	NN_e	12.42	5.6	19.26
	NN_n	39.94	17.54	20.34
$M_{S_{ref}*0.85}$	NN_{eRec}	28.84	4.02	14.81
	NN_e	12.08	4.04	14.76
	NN_n	28.83	12.23	18.06
$M_{S_{ref}*0.8}$	NN_{eRec}	26.81	3.01	12.53
	NN_e	11.81	3.04	12.48
	NN_n	26.81	9.51	16.41
$M_{S_{ref}*0.75}$	NN_{eRec}	25.99	2.37	10.62
	NN_e	11.51	2.39	10.56
	NN_n	25.98	7.21	14.87

Table 8.12: All measures on nominal NN vs erroneous NN vs recovery on scenario B.3

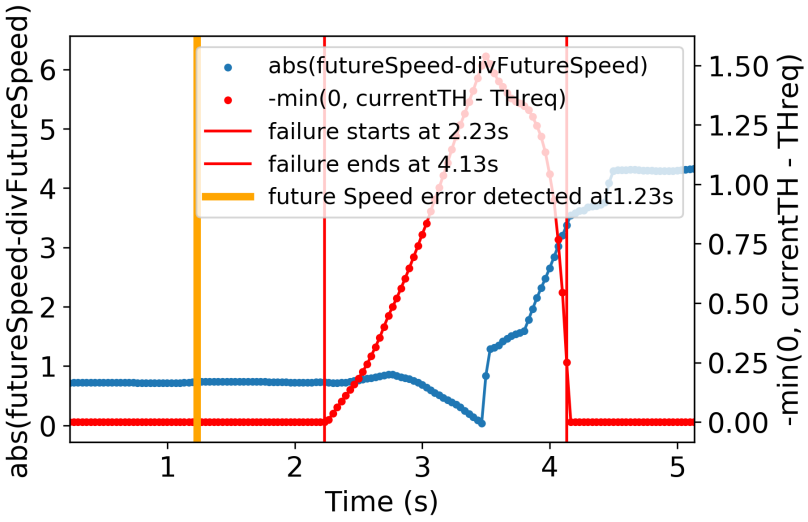


Figure 8.4: futureSpeed detection indicator and system's failure

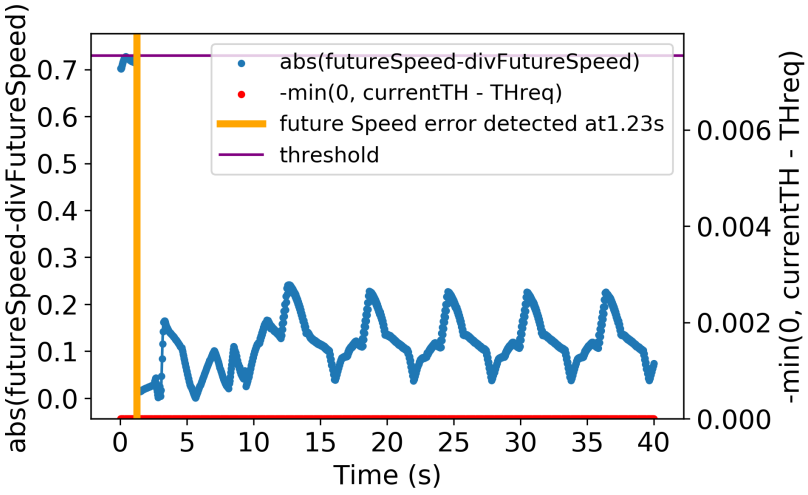


Figure 8.5: FutureSpeed detection and system's failure on the erroneous NN in scenario B.1.2

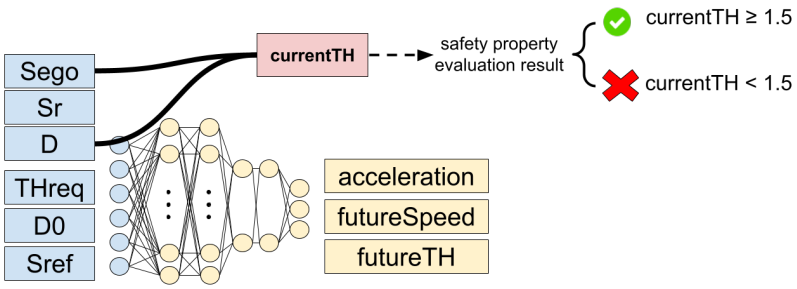


Figure 8.6: Architecture of the detection through online property check

8.2 Detection of an erroneous model based on online evaluation of the safety property

In this section, we will present the validation of our proposed error detection and recovery mechanisms using an online safety property evaluation. We will first present the specific architecture for these mechanisms, and then the implementation of the detection and recovery mechanisms on our autonomous vehicle application. As in previous sections, we will focus on scenario B.1.2 where the erroneous NN causes the ego car to crash into the exo car.

8.2.1 Architecture

In this experimental validation, we detect an erroneous NN by checking online if the safety property is respected. To achieve that, we compute the value of $currentTH$ online using equation 6.1 presented in section 6.1.2.5 and recalled hereafter in equation 8.3. We then check if this value is over or below the safety margin of $TH_{req} = 1.5$ seconds. These verifications do not require more information than the definition of the formal safety property and can thus be applied for every applications using formal verification. We present the architecture in figure 8.6.

$$current_{TH} = \frac{D}{S_{ego}} \tag{8.3}$$

8.2.2 Detection with online evaluation of the safety property

Since the detection is based on the online evaluation of the safety property, it is in our case equivalent to observing whether or not the measure M_{TH} becomes greater than 0. As we can see in tables 8.9, 8.10, 8.11 and 8.12, this measure is over 0 only in scenario B.1.2. The error is detected on average at 2.23 seconds

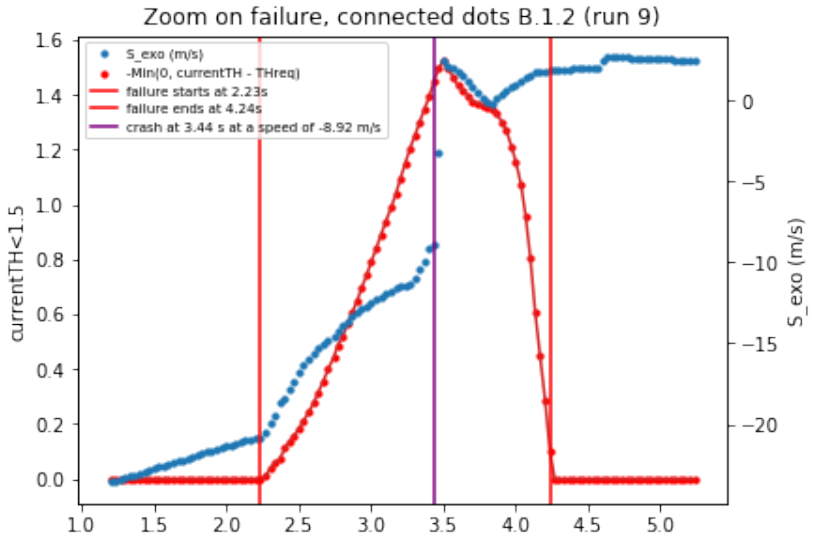


Figure 8.7: Erroneous neural network detection through online evaluation of the safety property and no recovery

and the crash happens on average at 3.42 seconds. The error is detected one second later compared to the detection using the mathematical diversified model on $futureSpeed$, which detected the error at 1.23 seconds. This will have a significant impact on the recovery, as we will see in the next section. However this method will be able to detect any errors in the NN, unlike the diversified mathematical model which will detect errors only on some of the outputs. We thus believe that these two methods are better used complementary. We present the recording of one experimental run using the online evaluation of the safety property without recovery in figure 8.7. On this figure, the red curve shows how much the TH goes below TH_{req}, invalidating the safety property. The first red vertical line represents the moment the error is detected, that is when the TH goes below TH_{req}, and the purple vertical line represents the crash. The second line shows when the safety property is validated again, which occurs after the crash. The blue curve represent the relative speed of the exo car. We also indicate in the legend the relative speed of the exo vehicle at the moment of the crash. On average this speed is -8.96 m/s when no recovery is applied, that is -32.256 km/h.

8.2.3 Recovery with online evaluation of the safety property

We will present in this section three recovery mechanisms that we implemented and tested on scenario B.1.2, which is the only one to cause a system failure with the erroneous NN. First we present the recovery based on nominal neural

	NN_e	$NN_{e-SPD-NN_nRec}$	$NN_{eSPD-ACC-script-Rec}$	$NN_{e-SPD-SBRec}$
$M_{t-crash}$	3.42	3.64	3.66	X
$M_{S-crash}$	-8.96	-3.66	-3.05	X
M_{TH}	2.098	2.041	1.848	1.547
$M_{D<D_s}$	2.942	3.522	3.133	2.344
$M_{S_{ego}>S_{ref}}$	0.0	0.0	0.0	0.0
$M_{low\ accel}$	0.0	0.0	0.0	1.513
$M_{high\ accel}$	0.14	0.0	0.0	0.0
$M_{S_{ego}avg}$	5.23	2.43	3.62	3.62
$M_{D_{avg}}$	46.8	64.7	15.6	14.3
$M_{S_{ref}*0.9}$	39.75	39.74	39.75	39.75
$M_{S_{ref}*0.85}$	39.62	39.61	39.61	39.62
$M_{S_{ref}*0.8}$	39.52	39.51	39.51	39.52
$M_{S_{ref}*0.75}$	39.28	39.41	39.41	39.42

Table 8.13: Measures on scenario B.1.2

network, similarly to section 8.1.4.2, then a recovery based on the ACC_{mod} function, and finally a recovery based on an emergency braking command followed by the ACC_{mod} function.

To describe more precisely the vehicle's behavior when a crash occurs, we will present in this section two more measures :

- $M_{T-crash}$: the time when the crash occurs, expressed in seconds.
- $M_{S-crash}$: the exo car relative speed when the crash occurs, expressed in m/s. This measures gives an estimation of how strong the crash is. Note that this value can only be negative, and that the higher the speed, the stronger the crash.

All the measures on scenario B.1.2 are summed up in table 8.13. We show results for the following NN:

- NN_e is the erroneous neural network with no recovery mechanism. A more detailed analysis and comparison with the nominal neural network is presented in section 7.3.
- $NN_{e-SPD-NN_nRec}$ is the erroneous neural network, with a detection based on the online safety property evaluation, and a recovery through the use of a nominal NN. We present its results in section 8.2.3.1.
- $NN_{eSPD-ACC-script-Rec}$ is the erroneous neural network, with a detection based on the online safety property evaluation, and a recovery based on the script of the ACC_{mod} function. We present its results in section 8.2.3.2.
- $NN_{e-SPD-SBRec}$ is the erroneous neural network, with a detection based on the online safety property evaluation, and a recovery based on an emergency braking command and the script of the ACC_{mod} function. We present its results in section 8.2.3.3.

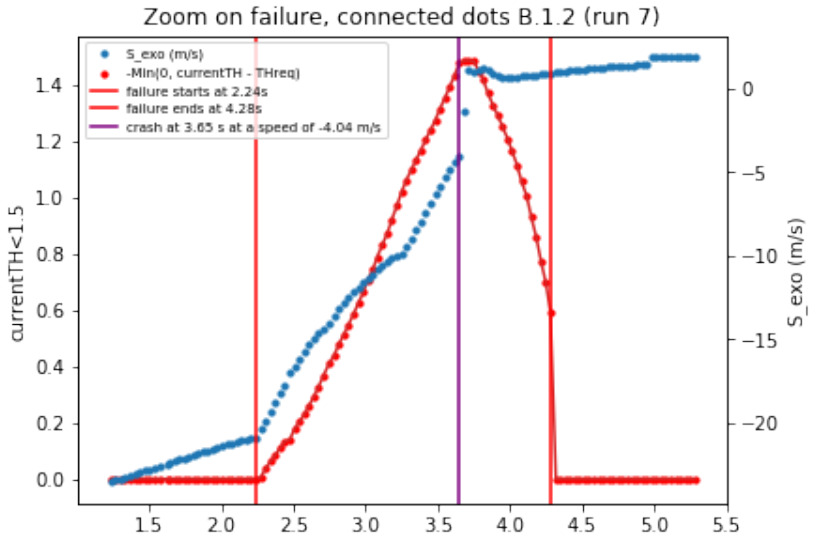


Figure 8.8: Erroneous neural network detection through online evaluation of the safety property and recovery through a nominal neural network.

8.2.3.1 Recovery based on redundant neural network

We perform a recovery with a redundant NN, using a nominal NN as redundancy, similarly as in section 8.1.4.2. However, we can see in table 8.13 that this recovery does not manage to avoid the crash, contrary to the previous one. This is due to the slower error detection of the safety property evaluation (2.3s instead of 1.3s), that only detects failures after they occur. One possible way to alleviate this problem would be to use a bigger safety margin (a bigger TH_{req} for example, or an additional safety margin used only for the error detection mechanism) but that would also consequently diminish the availability of the system. The average time of crash ($M_{t-crash}$) is 3.64 seconds, with a relative exo speed of -3.66 m/s or -13.176 km/h, which is much lower in absolute value than the speed without the recovery mechanism (-8.95 m/s, or -32.22 km/h). Thus, even if it fails to totally avoid the crash, the recovery mechanism still significantly alleviate it. Figure 8.8 presents an experimental run of this scenario. This figure is similar to the one without recovery mechanisms (figure 8.7), except that the crash is postponed and happens at a lower speed.

8.2.3.2 Recovery based on the ACC script

In these experiments, the recovery mechanism used after the error detection is to switch to the ACC_{mod} function. Note that we obtain similar results when using the original ACC script function.

The behavior of the system with this recovery mechanism is similar to the

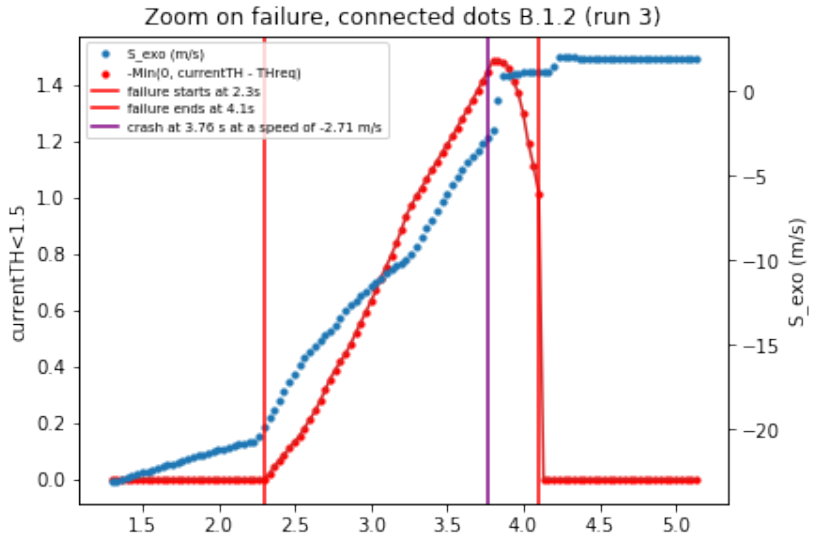


Figure 8.9: Erroneous neural network detection through online evaluation of the safety property and recovery through the ACC script.

previous one, with an average time of crash of 3.66 seconds, and a relative exo speed of -3.05 m/s or -10.98 km/h. The difference in value between this recovery and the previous one is due to variance errors in the nominal NN. In figure 8.9 we present a run of this scenario with the recovery based on the ACC, which is similar to figure 8.8.

8.2.3.3 Recovery based on an emergency braking followed by the ACC

As a third recovery mechanism, we present in this section an emergency brake command that produces a deceleration of $-4.5m/s^2$ when the safety property is invalidated, and switches back to the ACC_{mod} function when the safety property is respected again. Note that here again, we obtain similar results when the function used after the $-4.5m/s^2$ deceleration is the original ACC function.

As we can see in table 8.13, there is no crash when the recovery is done by this method: the ego car manages to stop before the crash. In figure 8.10 we present an experimental run of this scenario. We can see in this figure that the red curve does not reach the 1.5 seconds which means that we successfully avoided to into the exo car. Note that $-4.5m/s^2$ is a very strong deceleration, which would be felt strongly by the ego car's passenger, but a smaller deceleration of $-4m/s^2$ was still not enough to avoid the crash after the error detection.

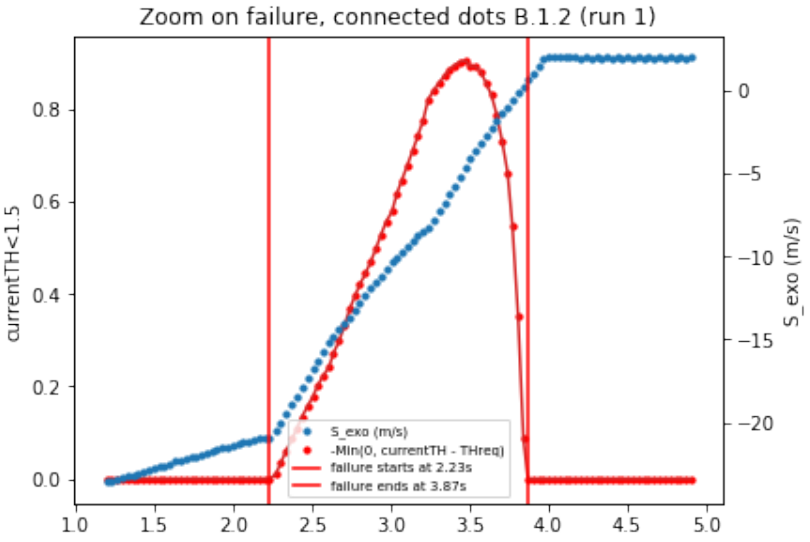


Figure 8.10: Erroneous neural network detection through online evaluation of the safety property and recovery based on an emergency braking.

8.3 Conclusion

In this chapter, we presented implementations of the detection and recovery mechanisms that we proposed in chapter 5 on an ACC application. First, we presented experiments with an error detection based on the use of equations as a diversified mathematical model to the NN's learned model:

- Detection: we chose to work with equations 8.1 and 8.2 that model respectively the future speed and the future time headway. We used these equations to compute two of the neural network outputs used in formal verification, and compare them with the value given by the NN. Only one of the equations was used at the end (the one based on $futureSpeed$), since the nominal NN output of the other one ($futureTH$) was too imprecisely modeled to allow us to determine an empirical threshold. Using the $futureSpeed$ comparison, we were able to predict more than one second in advance an invalidation of the safety property in the scenario that causes a system failure. Note however that this detection mechanism can not guarantee to detect all the faults in the NN's model. Indeed, we only check one of the NN's output ($futureSpeed$), which might not be correlated to the NN's output command ($acceleration_{desired}$) if the NN is erroneous. However, it allows us to detect a problem before a system's failure.

- Recovery: we use our nominal neural network as the redundancy in our recovery mechanism. When the detection mechanism detects an error,

we switch this neural network to replace the erroneous one. With this method, we manage to avoid both the car crash and the violation of the safety property in scenario B.1.2.

Secondly, we presented experiments where the error detection is based on an online evaluation of the safety property.

- Detection: The detection checks online if the TH becomes lower than the TH_{req} value of 1.5s. This is in fact equivalent to a M_{TH} measure higher than zero. The main inconvenient of this detection is that we detect the failure only when the system violates the safety property, giving less time to recover than the previous method, but it allows us to detect all errors leading to this failure.

- Recovery: considering that we had less time to recover with this error detection method than with the previous one, we tried three different recovery methods.

  - *Recovery based on redundant NN.* As in the previous application experiments, we used the nominal NN as a redundancy mechanism. This mechanism does not permit to avoid the crash but it succeeds in reducing from -8.96 m/s to -3.66 m/s the relative speed of the exo vehicle when the crash occurs, significantly mitigating it.
  - *Recovery based on the ACC_{mod} function.* Secondly, we used the ACC_{mod} function as a recovery mechanism. As the nominal NN was modeled after this ACC command, the later gave similar results to the former, with slight differences due to variance errors in the NN learning. In our case, it gave a slightly better relative exo speed at the impact of -3.05 but was still unable to avoid the crash.
  - *Recovery based on a strong deceleration and the ACC_{mod} function.* Third, we used an emergency braking command of $-4.5m/s^2$ after the error detection, followed by a switch back to the ACC command when the safety property is reestablished. This recovery mechanism totally avoids the crash even with the later detection, although obviously the safety property is violated for some time.

In this chapter, we successfully implemented the model error detection methods described in section 5.1.1. We also observed some of the theoretical limits of these mechanisms, and proposed solutions to some of them.

- *determination of the threshold* that we observed in the detection mechanism based on diversified equations. The solution we implemented for this limit was to evaluate empirically a threshold value that would not trigger a detection when the behavior of the car is correct (when it is driven by the nominal NN).

- *false negative.* We did not observe such limit in the implementation.

- *false positive.* We observed this limit for one of the potential indicator ($futureTH$) which was not precise enough to use. The other indicator did not suffer from this problem as we determined the threshold value empirically from the nominal NN's behavior.

- *difficulty to find an adequate secondary system.* In our case, we had no problem finding an adequate secondary system because of the experimental context of our application (we know already the function to model).

- *limited time for anticipation.* We observed this problem in the detection mechanism based on online safety evaluation in section 8.2.3.1, when the redundancy based on a neural network (or based on the ACC function) could not avoid the crash. Our solution to this limit was to create a recovery script with a stronger deceleration.

Regarding the limits related to the recovery through redundant NN (*Relying on neural networks that may also have an erroneous model* and *limits of number of NN used for the redundancy*), we did not encounter them since our redundant NN was the nominal one. To conclude, we can observe that it would be interesting to use both methods at the same time to benefit from their complementarity and improve the dependability of the system.

Chapter 9

Neural network problems for formal verification

In this chapter, we present different phenomena and comments we gathered during the realizations of our formal verifications. These observations are only qualitative since study in details quantitative aspects was not the focus of the thesis work. We still consider them as valuable experience feedback that could improve other formal verification work for neural networks. We will first discuss imprecisions that are due to model errors in neural networks. Then we will present imprecisions due to the variance errors and what approaches we implemented to decrease them. Then we will describe how the choice of the formal verification tool impacts the formal verification. Finally we will argue that the problem and the solutions presented in this work can be generalized to most machine learning approaches.

9.1 Imprecision due to model

As detailed in section 6.5.6, a lack of precision was one of the main factors that would prevent a neural network from passing the formal verification. In this section, we will present the factors related to the NN model that impact the precision of a neural network, particularly its architecture and its number of inputs and outputs.

9.1.1 Importance of the architecture

For our experiments, we implemented several techniques to improve the precision during the training, but tuning the architecture of the neural networks also plays an important role. The architecture tuning was at first necessary for the experiment presented in section 4.5.1 then in the ACC experiment presented in chapter 6. We witnessed that the tuning of the architecture was an impor-

NN architecture	64-32	64-32-32	64-64-32-32
average MSE on $futureTH$	18.07	12.75	18.38
average MSE on $futureSpeed$	0.21	0.10	0.19
average MSE on $acceleration$	0.21	0.09	0.13

Table 9.1: Average MSE computed on a 100 000 elements test set for different architectures.

tant factor in obtaining a neural network precise enough. The architecture of a neural network is first defined by the type of neural network (CNN, recurrent, MLP, etc...). This choice type must be guided not only by the task to achieve (regression, classification, format of the data, use of the neural network, etc.), but also by the restrictions imposed by the formal verification tool. In our case for example, we could only use classical feed-forward neural networks with ReLu activation functions. After this decision, the tuning of the number of layers and neurons per layers can be done by observing the impact on the precision on a verification test. We noticed in our experiments that bigger neural networks are not necessarily the most precise ones, probably because even if they're more expressive they require much more work to be trained well and are possibly more susceptible to local minimum or maximum. However, note that small NN may be not expressive enough to allow good results. In our case, a NN with only one hidden layer carried big errors. As another example, when trying several architectures (given in section 6.3.1) for the NN used in our ACC application, we observed that some architectures with more neurons were less precise than others with less. More precisely, when comparing the average MSE on a test set, we observed that the MSE of the 64-64-32-32 architecture was higher than the MSE of the 64-32-32 architecture for the three NN outputs. However the 64-32 architecture was still the less precise of the three. We present these results in table 9.1. Note that at the end, we still selected the 64-64-32-32 architecture based on the measures to evaluate the scenarios presented in section 6.4.2.7.

9.1.2 Number of inputs and outputs

When designing a neural network, one must consider the needed number of inputs and outputs. This depends on several parameters such as the format of the available data for the training (how many variables, how many dimensions, etc...), the task to achieve by the neural network (classification, regression, etc...), the available computing capabilities and time for the training, etc. The number of inputs and outputs will also impact the precision of the neural network prediction.

Regarding the number of inputs, the problem of selecting which inputs to use is known as *feature selection*. The choice of the input variables may impact the training (duration, speed to reach the same result and difficulty to tune the hyper parameters), the precision of the neural network after the training, the amount of data necessary to reach a given precision, etc. According to [78],

feature selection techniques for neural networks typically require a criterion to compare different sets of features (and then choose the best set of features), a search procedure to explore the possible combinations of features, and a stop criteria (when to stop the search procedure). There exist a great variety of methods for these tasks. Among the most used ones we can find the Pearson coefficient [103] that measures the linear correlation between two sets of data, and the Chi-squared test [104] that is used to measure correlation for categorical data. The choice of the feature selection technique is guided by the type of task the neural network must perform (regression, classification, etc...) and the format of the inputs and output values (integer variables, floating point variables, categorical variables, Boolean variables, etc.).

Similarly the number of outputs impact the precision of each output individually. We observed as a general rule that more outputs make it more complex to obtain a satisfying precision. However, as we presented in chapter 5 and observed in chapter 8, adding relevant outputs is useful to estimate the correctness of the neural network model, or even necessary to express then validate some safety properties.

9.2 Imprecision due to variance

In chapter 6, we presented the processes implemented to pass the formal verification on our neural network. In our experiments, the aim of our training was not only to have a neural network that would model accurately the function on the entire ODD, but we needed it to be precise enough in the situations where the safety property is still barely validated (for example, when the ego car is close to the exo car). During our experiments we noticed that for similar initial conditions, the same training parameters could provide a great variety of different neural networks, with some more precise than others. To improve our chances of getting a neural network that would pass the formal verification, we first started to train several neural networks with similar training conditions. With this approach we obtained a better sense of how to improve the training parameters. We first started to train 10 neural networks for the same training parameters, but later we trained 20 neural networks with the same training parameters at the same time. In the following subsections, we present five other techniques we implemented.

9.2.1 Generating uniform test sets to have several counter examples

When running the tool Marabou to see if a neural network would respect or not the formal property, the result of the formal verification would be the absence of counter example, which means that the neural network respects the safety property, or one counter example to the safety property. In this second case, we would need to re-train the neural network, ideally with some changes

in the training parameters and subdatasets (as in the process described in section 6.5.6.2) to increase our chances of getting a neural network that would pass the formal verification. Having only one counter example to guide the tuning makes the process very difficult, because at each step, we have little information to change the training set. This is why we developed the *safety property testing* process introduced in section 6.5.6.2. With this process, we test 20 neural networks produced with similar initial condition and trained on a uniform data set that covers the domain of the safety property. The dataset, which we will call formal verification dataset (or FV dataset), was created by generating all 6 inputs uniformly on the domain covered by the safety property. The *safety property testing* for one neural network then happens as follows: for each element of this dataset, we would compute the output through the neural network, and check if it respects the safety property. We would then collect all elements that do not respect the safety property as counter examples similar to the one provided by Marabou. The differences between the formal verification and the safety property testing is that passing the *safety property testing* does not guarantee that the safety property holds on the whole domain (only that it holds on the tried dataset), but it also gives several counter examples if it fails instead of one. For example, for a dataset of approximately 500 000 elements, we generally obtained between 10 and 600 counter examples depending on the precision of the neural network and its advancement on the iterative process to tune the training set. As previously said, we run the *safety property testing* on 20 neural networks from the same learning set at the same time, to decide how to better tune the learning set for the next set of 20 neural networks. To analyze the counter examples of one neural network, we first look at the exact number of counter examples and how far from the ground truth they are. We display an illustration of this process for one neural network in figure 9.1. In this figure, we can see that the counter examples have a $future_{TH}$ output value below 1.5 seconds, when they should have a value over 1.5 second (as we see in their ground truth). Note that very small errors (from 0.1 to 0.01 in figure 9.1) can be sufficient to invalidate the safety property.

To improve our training set, we observe what inputs produce these counter examples. A real example of inputs values of the counter examples are given in figures 9.2a, 9.2b, 9.2c and 9.2d. We also observe the values of D_S/D (as in figure 9.2e) and $-S_{ego}/S_{exo}$ (as in figure 9.2f), as we found empirically that these coefficients are sometimes a good characterization of situations that cause a failure of the safety property testing. Based on these observations, we try to identify which parts of the ODD produce counter examples, and we add to the dataset more entries that fit these situations (possibly adding a whole new type of entries in the dataset). Following the counter examples displayed in figures 9.2a to 9.2f, a typical modification of the training set would be to dedicate some small part of the training data only for data that cover values of S_{ego} in $[38; 42]$, S_{exo} in $[-2.0; -0.5]$ and D in $[57; 63]$ (S_{ref} covering the entire ODD). This modification is based on observing where the counter examples are located.

When a neural network does not have a counter example during the *safety property testing*, this is not a guarantee that it will pass the formal verification

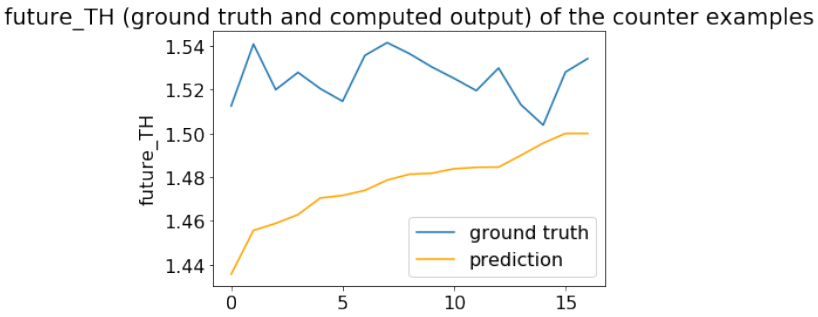


Figure 9.1: $future_{TH}$ (ground truth and computed outputs) of counter examples. Note that the X axis does not have any unit, it is simply used to sort the counter examples based on the $future_{TH}$ of their ground truth in ascending order. We can see in this figure that there are 16 counter examples.

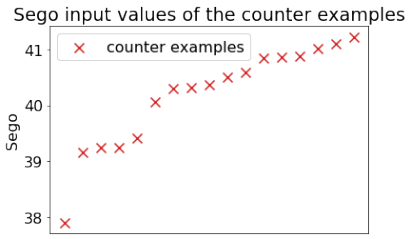
as previously stated. In our process, we then run another *safety property testing* on this neural network, but on a bigger dataset (approximately 5 millions of elements). This second phase would again be used to guide the subdataset tuning if counter examples were found, and avoid running the formal verification (this second larger dataset would still only be 10 to 30 minutes instead of the hours for the formal verification). Finally, when a neural network does not have a counter examples on this 5 millions elements dataset, we would run the formal verification on it. In general between 30 percent and 60 percent of the neural network reaching this phase would verify the formal property and have no counter example. We sum up the process of *safety property testing* in figure 9.3.

We observed during our experiments that this process was very useful to speed up (and even necessary for) the development of a neural network that passes the formal verification. The improvement can be explained by two reasons:

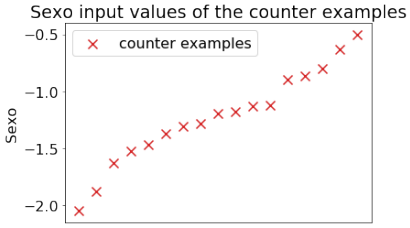
- Gain of time, because instead of running the formal verification for each neural network, we run the *safety property testing* which is significantly shorter (approximately 5 to 15 minutes against several hours).
- Gain of precision for guiding the changes to make for the next training (new subdatasets mostly), because we have dozens or hundreds of counter examples per neural network, instead of one per neural network.

9.2.2 Conservative function

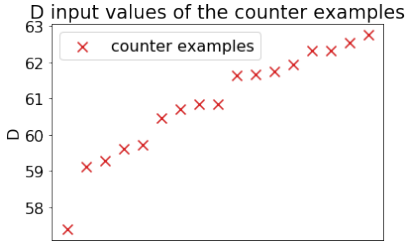
As presented in section 6.5.6.1, another method we used was to train the neural network with data provided with a more conservative function than the one we aim to model. This gives an error margin for the imprecision of the neural network with regards to the safety property to verify, since the training data



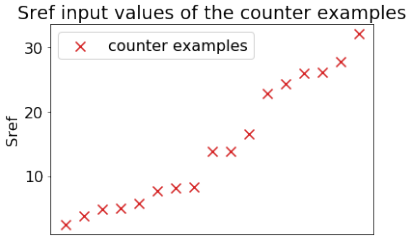
(a) S_{ego} of counter examples



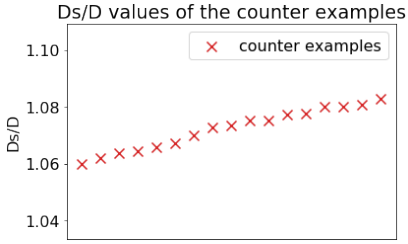
(b) S_{exo} of counter examples



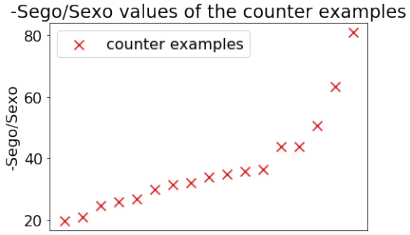
(c) D of counter examples



(d) S_{ref} of counter examples



(e) D_S/D of counter examples



(f) $-S_{ego}/S_{exo}$ of counter examples

Figure 9.2: Inputs of the counter examples. Note that for all these figures, the X axis does not have any unit, it is simply used to sort the 16 counter examples based on the unit of the Y axis in ascending order.

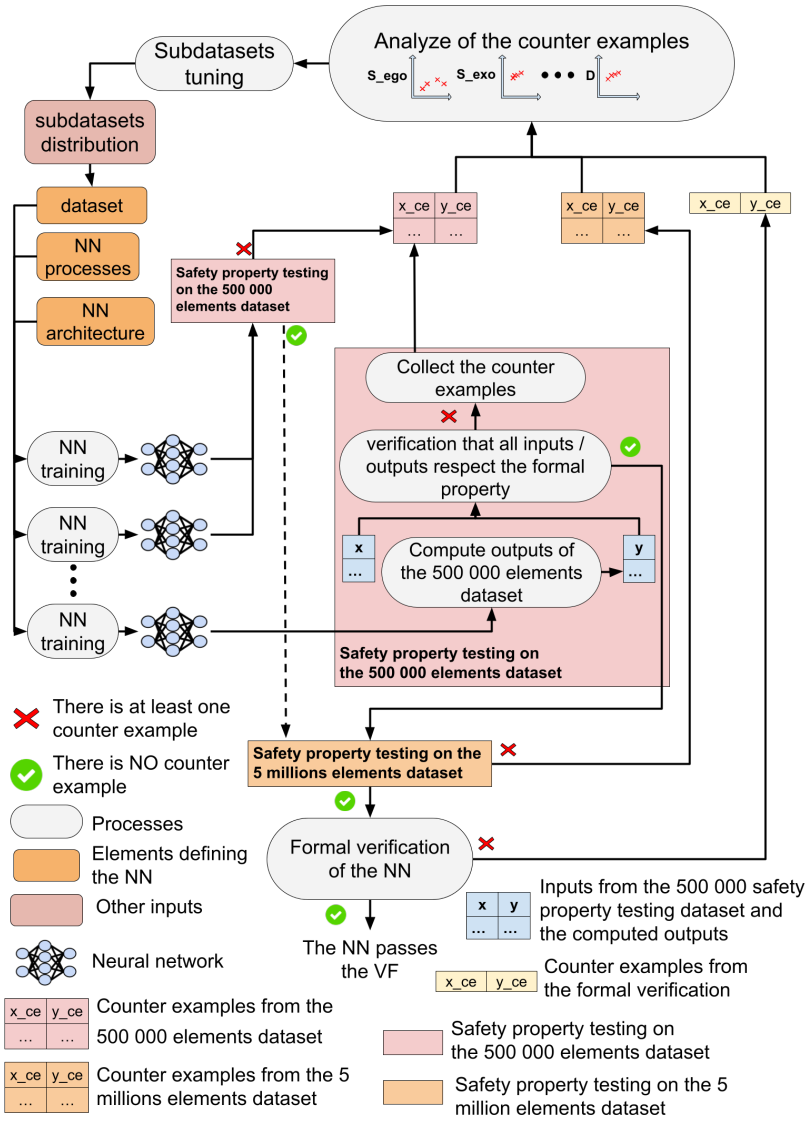


Figure 9.3: Erroneous neural network detection through online evaluation of the safety property and recovery through a nominal neural network. In this figure, we only detail the *safety property testing* process on the 500 000 elements dataset for the last NN. The process is the same for the 5 millions elements dataset. The other neural networks also go through these two processes and their counter examples are also used for the analysis.

make the neural network behavior safer (keeping a larger safety distance). We can note that when changing the original function that generates the data to a more conservative one, we also negatively impact the reliability of the neural network because the system will accelerate less and generally keep a bigger distance with the exo car than the original one. We can also note that the use of the conservative function alone did not allow us to pass the formal verification, but we needed also to tune the training set into subsets discribed in section 6.5.6.2.

9.2.3 Restriction on the domain of the formal verification

Another way to increase the chances of passing the formal verification is to decrease the domain where the constraints on the neural network outputs are applied. In other words, make the *pre* conditions more strict so the domain to explore and verify is smaller. In our experiment, we used this method in two different ways:

- Limit the safety property to what is physically achievable. As presented in section 6.5.1.2, we added a constraint that limits the safety property to situations where the maximal braking of $-3m.s^{-2}$ is sufficient to prevent the ego car from getting closer to the exo car than the safe distance TH_{req} permit. This is justified from a safety point of view as we can not ask the system to do better than the limits stated in the specifications.

- Remove situations that are outside of the ODD. This was done in our experiment by excluding situations where the exo speed goes backward and where the exo car reaches a very high speed (greater than 150km/h), which means that it would go faster than the ego car. As a reminder, we explained in section 6.1.2.4 that the situations where the exo car goes backward were also excluded because they were a very specific and abnormal situation.

Note that adding these constraints does not necessary make the verification shorter for Marabou (we observed that the verification was still hours long), but it prevents the appearance of counter examples that are not relevant for the safety problem that we want to consider. It is also important to check that these new restrictions do not diminish the intended fault prevention effect. In other words we want the final formal property to still guarantee as much safety as the original in the considered ODD.

9.3 Importance of choosing a formal verification tool

During our experiment, we had to choose a neural network formal verification tool. This led us to realize that when performing formal verification for a neural

network, the choice of the tool will impact what kind of formal verification can be done. It is specifically important to consider:

- the task of the neural network (classification, regression, etc..). We observed that most of the tools are designed for neural networks performing classification.

- the architecture supported by the tool. Different tools cover different architectures (CNN, RNN, MLP, etc...) and different activation functions.

- the format of the property supported by the tool. We saw that several tools are designed especially for robustness properties. Another example is the limitation of the format in Marabou, where input and output variables can not be multiplied with other inputs or outputs and have to be presented as in equations 4.3 or 4.4.

- the level of documentation and complexity to adapt the tool to a new formal verification problem. This factor is especially important since all available tools currently are research projects, with not much configuration options.

- the format of the neural network file. To perform the formal verification, the tool must receive a complete description of the neural network (architecture, weights, bias etc...). This is usually done by loading a file containing the neural network description. The formats can be linked to machine learning frameworks. For example in TensorFlow, neural networks can be saved as protocol buffer files, with a .pb file extension. In the case of Marabou, the format accepted is the NNet format [55].

9.4 Erroneous models in other machine learning techniques

Formal verification can also be applied in other machine learning methods than neural networks. For example in the review presented in [149] the authors list formal methods for two other machine learning methods: Support Vector Machines and Decision Tree Ensemble. Although we focused in this work on neural networks, we believe that the problem of erroneous model presented in section 4.4.2 can be encountered in any machine learning methods as all formal verification in machine learning is done on the models learned, which can contain errors and are usually not easily understandable by humans. The methods we propose in this thesis to tackle the erroneous model problem (detection and recovery mechanism respectively presented in section 5.1.1 and 5.2.1) should also be applicable to other machine learning techniques, since they do not rely on any NN specificities.

9.5 Conclusion

In this chapter, we detailed some lessons that we learned during our experiments about formal verification for neural network. A first part of these lessons are related to the precision of the neural network. We witnessed that one of the main factor preventing a neural network from being formally verified was the lack of precision leading to the presence of counter examples. Among the causes of model bias errors, we identified:

- the neural network architecture (type of NN, number of layers and neurons),
- the number of inputs and outputs.

We also presented the approaches we used to reduce the variance errors, which are significant. Among these techniques there are:

- the *safety property testing* where we generated a test set that respect the property, and look for counter examples to guide the learning set tuning,
- the use of more conservative data to train the NN, to have a bigger error margin for the variance errors,
- restrictions of the domain where the formal verification is applied, which cause the decrease of domain where counter examples can arise. These restrictions are based on the removal of situations that do not pose a safety threat, or situations that are outside of physical constraints (in our case, due to the vehicle limitations).

We also presented the importance of the formal verification tool choice and the criteria to consider when choosing a formal verification tool. Finally we expressed that in our opinion, the erroneous model issues presented in this thesis in section 4.4.2 can also be present in other formal verification methods for machine learning techniques, on which our proposed solutions could be applied.

Chapter 10

Conclusion

In this thesis, we aimed to improve the dependability of neural networks for safety critical applications, with a focus on neural network dedicated to the decision part of the autonomous driving process. Among the existing research approaches that try to tackle this subject, we chose to work on formal verification for neural networks.

Among the main lessons of this thesis, we identify the following ones:

- Formal verification seems to us a good candidate to ensure the dependability of neural networks in safety critical applications. We identified two reasons for this:
  - Formal verification for neural networks permits to ensure that a safety property is valid over entire ranges of the input and output domains. This is very useful for open environments where testing is insufficient to guarantee the safety.
  - Formal verification for neural networks does not require to provide an oracle: we do not need to know what the correct output should be for a given input, we just need to define the safety property that determines what behavior is safe or not.
- Several problems still need to be resolved to allow the use of formal verification for neural networks as a method to ensure their dependability in safety critical applications. Among them we found:
  - Limits due to the inputs and outputs. This is due to the fact that for some cases, we may face a situation where the safety property to verify may use variables that are not input or output of the NN. It is then necessary to add these output and output variables to the neural network.
  - Limits due to the formal verification tools. For now few tools are available, and they have restrictions on the type of neural network

that they can verify (specific structure, activation functions, or size limitation) and the format of property accepted. For example, at the moment, we did not encountered a tool that can handle non linear properties (aka, a property that contains a multiplication between inputs or outputs).

- Limits related to the level of precision necessary to achieve a formal verification of a property. Since the formal verification can only be achieved when there are no counter example to a safety property, this requires a sufficient precision of the NN on entire ranges of the input and output domains.
- Although already known for classical systems, the erroneous model problem is a particular threat to the use of formal verification. It can be summed up as follows: since that there are currently no method that can guarantee that the model of a NN is correct, we currently have no guarantee that a safety property formally verified on a neural network will be respected in real life. In classical systems, faults in the model were eliminated by experts reviews but the lack of explainability of neural networks knowledge makes that near impossible.

- As a proposal to answer the erroneous model problem, we developed and implemented detection and recovery mechanisms. We sum up here both of them.
  - The objective of the detection mechanism is to detect if a neural network has an erroneous model, and then prevent potential failure (since as we saw, when a NN model is erroneous, the guarantees of the FV do not hold). This can be done by two means:
    * by using equations describing environment relations. In this case, we use known equations that describe part of the inner working of the NN. The equation must link at least one input to at least one output. We compare the output produced by the NN to the output computed by the equations, the observation of important differences between them indicating that the model of the NN is erroneous. This solution has no guarantee against false negative (we have no guarantee that we will detect all errors in the model) and against false positive (when an error is detected, the overall behavior of the NN might still be correct). Another limit is that it might be difficult, or even impossible to find adequate equations for this system.
    * by using an online evaluation of the safety property. For this method, we simply evaluate online if the output produced by the NN respects the safety property formally verified. If the safety property is violated even one time, then the formal verification should not have succeeded and the NN model is erroneous. This method offers little margin between the detection and the potential failure since the detection happens when the safety property

is already violated. However, it will always be possible to implement it, since it only uses the safety property formally verified and all safety property's violation will be detected.

We suggest to use both methods complementarily, as they limit each others disadvantages. However a common problem to both mechanisms is the determination of a threshold when neural network performs a regression task (which is usual in decision problem). Note that the threshold determination limit can also be present for certain classification tasks (for example in the case of robustness property, to determine a minimum distance).

- The recovery mechanism that we implemented relies on the use of diversified neural network as a redundancy. We also suggested that others recovery mechanisms should be used, for example based on a different AI technology, a simpler NN, or degraded function implemented using imperative programming.

- The formal verification of a neural network is a complex procedure where one must pay attention to the following points:
  - The choice of the formal verification tool. This will determine the type of NN architecture that can be verified and the format of the property that can be verified. When choosing a formal verification tool for neural network, one must also pay attention to the level of documentation of the tool, the task of the NN supported by the tool (regression, classification) and the format of the NN file.
  - The safety property to verify, and in particular the input domain. In our case, by removing situations that are outside of the ODD and limiting the safety property to what is physically achievable, we reduced a bit the part of the input domain where the property to verify is applied to diminish the chances of encountering counter examples.
  - The precision of the NN. As we observed, this is the main source of failure to pass the formal verification. The precision of the neural network is influenced by the following factors:
    * The structure of the NN.
    * The number of inputs and outputs.
    * The function used to generate the data. In our case, we implemented a conservative function to generate the data. This gives more margin for the precision of the NN with regard to the safety property.
    * The distribution of the training dataset. In our case, we used an iterative procedure to tune the dataset that we detail precisely and that was of tremendous help to obtain NN that passed the formal verification.

- Future work could be conducted to investigate further the topics mentioned in this thesis. We list here possible experiments and research axis:
  - Regarding the experiment we conducted we should:
    - Propose and inject more faults in our experimental application to further illustrate the erroneous model problem and show that our proposed solutions also tolerates them successfully. In particular, we could implement the faults proposed in section 7.1.3.
    - Use another tool than Marabou to perform the formal verification. This would bring more guarantee that the erroneous model problem holds for all formal verifications on neural networks. The best tools mentioned in the VNN competition [1] could be a basis to look for additional tools.
    - Consider additional properties to formally verify. With more time, the ones presented in section 6.5.1.3, despite not being safety properties, could be used in our experiment.
  - Another experiment could also be conducted on a different autonomous driving function than the ACC. For example, a lateral control function (including overtaking another car), which could not be easily implemented with classical control laws, could be considered, or functions in the perception domain.
  - Finally, since we are confident that the erroneous model issue is present in most machine learning approaches, some experiments on other ML methods like linear regression or generalized additive model (GAM) could also be realized.

Annexes

Properties of the ReluPlex tool

Here we present the properties of the ReluPlex tool detailed in [59] and [60]. The properties are presented in tables 10.1, 10.2, 10.3, 10.4, 10.5, 10.6, 10.7, 10.8, 10.9 and 10.10. We also present five additional properties (tables 10.11, 10.12, 10.13, 10.14 and 10.15) that are introduced in [157].

Property 1	
Description	If the intruder is distant and is significantly slower than the ownship, the score of a COC advisory will always be below a certain fixed threshold.
Tested on	all 45 networks
Input constraints	$55947.691 \leq \rho$, $1145 \leq v_{own}$, $v_{int} \leq 60$.
Desired output property	the score for COC is at most 1500

Table 10.1: ReLuPlex property 1

Property 2	
Description	If the intruder is distant and is significantly slower than the ownship, the score of a COC advisory will never be maximal.
Tested on	$N_{x,y}$ for all $x \geq 2$ for all y
Input constraints	$55947.691 \leq \rho$, $1145 \leq v_{own}$, $v_{int} \leq 60$.
Desired output property	the score for COC is not the maximal score

Table 10.2: ReLuPlex property 2

Property 3	
Description	If the intruder is directly ahead and is moving towards the ownship, the score for COC will not be minimal.
Tested on	all networks except$N_{1,7}$, $N_{1,8}$, and $N_{1,9}$
Input constraints	$1500 \leq \rho \leq 1800$, $-0.06 \leq \theta \leq 0.06$, $3.10 \leq \psi$, $980 \leq v_{own}$, $960 \leq v_{int}$.
Desired output property	the score for COC is not the minimal score

Table 10.3: ReLuPlex property 3

Property 4	
Description	If the intruder is directly ahead and is moving away from the ownship but at a lower speed than that of the ownship, the score for COC will not be minimal.
Tested on	all networks except$N_{1,7}$, $N_{1,8}$, and $N_{1,9}$
Input constraints	$1500 \leq \rho \leq 1800$, $-0.06 \leq \theta \leq 0.06$, $\psi = 0$, $1000 \leq v_{own}$, $700 \leq v_{int} \leq 800$.
Desired output property	the score for COC is not the minimal score

Table 10.4: ReLuPlex property 4

Property 5	
Description	If the intruder is near and approaching from the left, the network advises "strong right".
Tested on	$N_{1,1}$
Input constraints	$250 \leq \rho \leq 400$, $0.2 \leq \theta \leq 0.4$, $-3.141592 \leq \psi \leq -3.141592 + 0.005$, $100 \leq v_{own} \leq 400$, $0 \leq v_{int} \leq 400$
Desired output property	the score for strong right is the minimal score.

Table 10.5: ReLuPlex property 5

Property 6	
Description	If the intruder is sufficiently far away, the network advises COC.
Tested on	$N_{1,1}$
Input constraints	$12000 \leq \rho \leq 62000$, $(0.7 \leq \theta \leq 3.141592) \vee (-3.141592 \leq \theta \leq -0.7)$, $-3.141592 \leq \psi \leq -3.141592 + 0.005$, $100 \leq v_{own} \leq 1200$, $0 \leq v_{int} \leq 1200$
Desired output property	the score for COC is the minimal score.

Table 10.6: ReLuPlex property 6

Property 7	
Description	If vertical separation is large, the network will never advise a strong turn.
Tested on	$N_{1,9}$
Input constraints	$0 \leq \rho \leq 60760$, $-3.141592 \leq \theta \leq 3.141592$, $-3.141592 \leq \psi \leq 3.141592$, $100 \leq v_{own} \leq 1200$, $0 \leq v_{int} \leq 1200$
Desired output property	the scores for "strong right" and "strong left" are never the minimal scores.

Table 10.7: ReLuPlex property 7

Property 8	
Description	For a large vertical separation and a previous "weak left" advisory, the network will either output COC or continue advising "weak left".
Tested on	$N_{2,9}$
Input constraints	$0 \leq \rho \leq 60760$, $-3.141592 \leq \theta \leq -0.75 * 3.141592$, $-0.1 \leq \psi \leq 0.1$, $600 \leq v_{own} \leq 1200$, $600 \leq v_{int} \leq 1200$
Desired output property	the score for "weak left" is minimal or the score for COC is minimal.

Table 10.8: ReLuPlex property 8

Property 9	
Description	Even if the previous advisory was "weak right", the presence of a nearby intruder will cause the network to output a "strong left" advisory instead.
Tested on	$N_{3,3}$
Input constraints	$2000 \leq \rho \leq 7000$, $-0.4 \leq \theta \leq -0.14$, $-3.141592 \leq \psi \leq -3.141592 + 0.01$, $100 \leq v_{own} \leq 150$, $0 \leq v_{int} \leq 150$
Desired output property	the score for "strong left" is minimal.

Table 10.9: ReLuPlex property 9

Property 10	
Description	For a far away intruder, the network advises COC.
Tested on	$N_{4,5}$
Input constraints	$36000 \leq \rho \leq 60760$, $0.7 \leq \theta \leq 3.141592$, $-3.141592 \leq \psi \leq -3.141592 + 0.01$, $900 \leq v_{own} \leq 1200$, $600 \leq v_{int} \leq 1200$
Desired output property	the score for COC is minimal.

Table 10.10: ReLuPlex property 10

Property 11	
Description	If the intruder is near and approaching from the left but the vertical separation is comparably large, the network still tend to advise "strong right" more than COC.
Tested on	$N_{1,1}$
Input constraints	$250 \leq \rho \leq 400$, $0.2 \leq \theta \leq 0.4$, $-3.141592 \leq \psi \leq -3.141592 + 0.005$, $100 \leq v_{own} \leq 400$, $0 \leq v_{int} \leq 400$
Desired output property	the score for "strong right" is always smaller than COC.

Table 10.11: ReLuPlex property 11

Property 12	
Description	If the intruder is distant and is significantly slower than the ownship, the score of a COC advisory will be the minimal.
Tested on	$N_{3,3}$
Input constraints	$55947.691 \leq \rho$, $1145 \leq v_{own}$, $v_{int} \leq 60$
Desired output property	the score for COC is the minimal score.

Table 10.12: ReLuPlex property 12

Property 13	
Description	For a far away intruder but the vertical distance are small, the network always advises COC no matter the directions are.
Tested on	$N_{1,1}$
Input constraints	$60000 \leq \rho \leq 60760$, $-3.141592 \leq \theta \leq 3.141592$, $-3.141592 \leq \psi \leq 3.141592$, $0 \leq v_{own} \leq 360$, $0 \leq v_{int} \leq 360$
Desired output property	the score for COC is the minimal.

Table 10.13: ReLuPlex property 13

Property 14	
Description	If the intruder is near and approaching from the left and vertical distance is small, the network always advises strong right no matter previous action is strong right or strong left.
Tested on	$N_{4,1}$, $N_{5,1}$
Input constraints	$250 \leq \rho \leq 400$, $0.2 \leq \theta \leq 0.4$, $-3.141592 \leq \psi \leq -3.141592 + 0.005$, $100 \leq v_{own} \leq 400$, $0 \leq v_{int} \leq 400$
Desired output property	the score for "strong right" is always the minimal.

Table 10.14: ReLuPlex property 14

Property 15	
Description	If the intruder is near and approaching from the right and vertical distance is small, the network always advises strong left no matter previous action is strong right or strong left.
Tested on	$N_{4,1}$, $N_{5,1}$
Input constraints	$250 \leq \rho \leq 400$, $-0.4 \leq \theta \leq -0.2$, $-3.141592 \leq \psi \leq -3.141592 + 0.005$, $100 \leq v_{own} \leq 400$, $0 \leq v_{int} \leq 400$
Desired output property	the score for "strong left" is always the minimal.

Table 10.15: ReLuPlex property 15

Bibliography

[1] *3rd International Verification of Neural Networks Competition (VNN-COMP'22)*. 2022. URL: https://sites.google.com/view/vnn2022.

[2] Florent Altché and Arnaud de La Fortelle. "An LSTM network for highway trajectory prediction". In: *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*. 2017, pp. 353–359. DOI: 10.1109/ITSC.2017.8317913.

[3] Oscar Araque et al. "Enhancing deep learning sentiment analysis with ensemble techniques in social applications". In: *Expert Systems with Applications* 77 (2017), pp. 236–246. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2017.02.002. URL: https://www.sciencedirect.com/science/article/pii/S0957417417300751.

[4] Daniel Arp et al. "DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket". In: Feb. 2014. DOI: 10.14722/ndss.2014.23247.

[5] Cauchy Augustin. "Méthode générale pour la résolution de systèmes d'équations simultanées." In: *Compte rendu des séances de l'académie des sciences*. 1847, pp. 536–538.

[6] Algirdas Avizienis et al. "Basic Concepts and Taxonomy of Dependable and Secure Computing". In: *IEEE Trans. Dependable Secur. Comput.* 1.1 (Jan. 2004), pp. 11–33. ISSN: 1545-5971. DOI: 10.1109/TDSC.2004.2. URL: http://dx.doi.org/10.1109/TDSC.2004.2.

[7] Sebastian Bach et al. "On Pixel-Wise Explanations for Non-Linear Classifier Decisions by Layer-Wise Relevance Propagation". In: *PLOS ONE* 10.7 (July 2015), pp. 1–46. DOI: 10.1371/journal.pone.0130140. URL: https://doi.org/10.1371/journal.pone.0130140.

[8] Junjie Bai, Fang Lu, Ke Zhang, et al. *ONNX: Open Neural Network Exchange*. https://github.com/onnx/onnx. 2019.

[9] Stanley Bak, Changliu Liu, and Taylor T. Johnson. "The Second International Verification of Neural Networks Competition (VNN-COMP 2021): Summary and Results". In: *CoRR* abs/2109.00498 (2021). arXiv: 2109.00498. URL: https://arxiv.org/abs/2109.00498.

[10] Osbert Bastani et al. “Measuring Neural Net Robustness with Constraints”. In: *Advances in Neural Information Processing Systems*. Ed. by D. Lee et al. Vol. 29. Curran Associates, Inc., 2016. URL: https://proceedings.neurips.cc/paper/2016/file/980ecd059122ce2e50136bda65c25e07-Paper.pdf.

[11] Özlem BATUR DİNLER and Nizamettin AYDIN. “An Optimal Feature Parameter Set Based on Gated Recurrent Unit Recurrent Neural Networks for Speech Segment Detection”. In: *Applied Sciences* 10.4 (2020). ISSN: 2076-3417. DOI: 10.3390/app10041273. URL: https://www.mdpi.com/2076-3417/10/4/1273.

[12] Mariusz Bojarski et al. “End to End Learning for Self-Driving Cars”. In: *CoRR* abs/1604.07316 (2016). arXiv: 1604.07316. URL: http://arxiv.org/abs/1604.07316.

[13] Léon Bottou. “Stochastic Gradient Descent Tricks”. In: *Neural Networks: Tricks of the Trade: Second Edition*. Ed. by Grégoire Montavon, Geneviève B. Orr, and Klaus-Robert Müller. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 421–436. ISBN: 978-3-642-35289-8. DOI: 10.1007/978-3-642-35289-8_25. URL: https://doi.org/10.1007/978-3-642-35289-8_25.

[14] Rudy Bunel et al. “A Unified View of Piecewise Linear Neural Network Verification”. In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. NIPS’18. Montréal, Canada: Curran Associates Inc., 2018, pp. 4795–4804.

[15] Hyunmin Chae et al. “Autonomous braking system via deep reinforcement learning”. In: *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*. 2017, pp. 1–6. DOI: 10.1109/ITSC.2017.8317839.

[16] Chih-Hong Cheng, Georg Nührenberg, and Harald Ruess. “Maximum Resilience of Artificial Neural Networks”. In: *Automated Technology for Verification and Analysis*. Ed. by Deepak D’Souza and K. Narayan Kumar. Cham: Springer International Publishing, 2017, pp. 251–268. ISBN: 978-3-319-68167-2.

[17] Vasek Chvátal. *Linear Programming*. W. H. Freeman Company, 1983.

[18] Moustapha Cisse et al. “Parseval Networks: Improving Robustness to Adversarial Examples”. In: *Proceedings of the 34th International Conference on Machine Learning - Volume 70*. ICML’17. Sydney, NSW, Australia: JMLR.org, 2017, pp. 854–863.

[19] *Computer Aided Verification - 34th International Conference, CAV*. Haifa, Israel, Aug. 2022.

[20] Yaodong Cui et al. “Deep Learning for Image and Point Cloud Fusion in Autonomous Driving: A Review”. In: *IEEE Transactions on Intelligent Transportation Systems* 23.2 (2022), pp. 722–739. DOI: 10.1109/TITS.2020.3023541.

[21] Piotr Dabkowski and Yarin Gal. "Real Time Image Saliency for Black Box Classifiers". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper/2017/file/0060ef47b12160b9198302ebdb144dcf-Paper.pdf.

[22] Jia Deng et al. "ImageNet: A large-scale hierarchical image database". In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848.

[23] Li Deng. "The mnist database of handwritten digit images for machine learning research". In: *IEEE Signal Processing Magazine* 29.6 (2012), pp. 141–142.

[24] Bin Ding, Huimin Qian, and Jun Zhou. "Activation functions and their characteristics in deep neural networks". In: *2018 Chinese Control And Decision Conference (CCDC)*. 2018, pp. 1836–1841. DOI: 10.1109/CCDC.2018.8407425.

[25] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In: *Journal of Machine Learning Research* 12.61 (2011), pp. 2121–2159. URL: http://jmlr.org/papers/v12/duchi11a.html.

[26] Souradeep Dutta et al. "Output Range Analysis for Deep Feedforward Neural Networks". In: *NASA Formal Methods*. Ed. by Aaron Dutle, César Muñoz, and Anthony Narkawicz. Cham: Springer International Publishing, 2018, pp. 121–138. ISBN: 978-3-319-77935-5.

[27] Rüdiger Ehlers. "Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks". In: *Automated Technology for Verification and Analysis*. Ed. by Deepak D'Souza and K. Narayan Kumar. Cham: Springer International Publishing, 2017, pp. 269–286. ISBN: 978-3-319-68167-2.

[28] David Eigen, Christian Puhrsch, and Rob Fergus. "Depth Map Prediction from a Single Image Using a Multi-Scale Deep Network". In: *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*. NIPS'14. Montreal, Canada: MIT Press, 2014, pp. 2366–2374.

[29] M. Everingham et al. *The PASCAL Visual Object Classes Challenge 2007 (VOC2007) Results*. http://www.pascal-network.org/challenges/VOC/voc2007/workshop/index.html.

[30] Quanfu Fan, Lisa Brown, and John Smith. "A closer look at Faster R-CNN for vehicle detection". In: *2016 IEEE Intelligent Vehicles Symposium (IV)*. 2016, pp. 124–129. DOI: 10.1109/IVS.2016.7535375.

[31] Claudio Ferrari et al. "Complete Verification via Multi-Neuron Relaxation Guided Branch-and-Bound". In: *International Conference on Learning Representations*. 2022. URL: https://openreview.net/forum?id=l_amHf1oaK.

[32] Matteo Fischetti and Jason Jo. "Deep neural networks and mixed integer linear optimization". In: *Constraints* 23.3 (July 2018), pp. 296–309. ISSN: 1572-9354. DOI: 10.1007/s10601-018-9285-6. URL: https://doi.org/10.1007/s10601-018-9285-6.

[33] Michael Fisher, Louise Dennis, and Matt Webster. "Verifying Autonomous Systems". In: *Commun. ACM* 56.9 (Sept. 2013), pp. 84–93. ISSN: 0001-0782. DOI: 10.1145/2494558. URL: https://doi.org/10.1145/2494558.

[34] Lex Fridman, Jack Terwilliger, and Benedikt Jenik. *DeepTraffic: Crowdsourced Hyperparameter Tuning of Deep Reinforcement Learning Systems for Multi-Agent Dense Traffic Navigation*. 2019. arXiv: 1801.02805 [cs.NE].

[35] Nicholas Frosst and Geoffrey E. Hinton. "Distilling a Neural Network Into a Soft Decision Tree". In: *Proceedings of the First International Workshop on Comprehensibility and Explanation in AI and ML 2017 co-located with 16th International Conference of the Italian Association for Artificial Intelligence (AI*IA 2017), Bari, Italy, November 16th and 17th, 2017*. Ed. by Tarek R. Besold and Oliver Kutz. Vol. 2071. CEUR Workshop Proceedings. CEUR-WS.org, 2017. URL: http://ceur-ws.org/Vol-2071/CExAIIA%5C_2017%5C_paper%5C_3.pdf.

[36] K. Furuta, M. Yamakita, and S. Kobayashi. "Swing up control of inverted pendulum". In: *Proceedings IECON '91: 1991 International Conference on Industrial Electronics, Control and Instrumentation*. 1991, 2193–2198 vol.3. DOI: 10.1109/IECON.1991.239008.

[37] Hunter Gabbard et al. "Matching Matched Filtering with Deep Networks for Gravitational-Wave Astronomy". In: *Phys. Rev. Lett.* 120 (14 Apr. 2018), p. 141103. DOI: 10.1103/PhysRevLett.120.141103. URL: https://link.aps.org/doi/10.1103/PhysRevLett.120.141103.

[38] Timon Gehr et al. "AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation". In: *2018 IEEE Symposium on Security and Privacy (SP)* (2018), pp. 3–18.

[39] Timon Gehr et al. "AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation". In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018, pp. 3–18. DOI: 10.1109/SP.2018.00058.

[40] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. Book in preparation for MIT Press. MIT Press, 2016. URL: http://www.deeplearningbook.org.

[41] Divya Gopinath et al. "DeepSafe: A Data-Driven Approach for Assessing Robustness of Neural Networks". In: *Automated Technology for Verification and Analysis*. Ed. by Shuvendu K. Lahiri and Chao Wang. Cham: Springer International Publishing, 2018, pp. 3–19. ISBN: 978-3-030-01090-4.

[42] Sorin Grigorescu et al. “A survey of deep learning techniques for autonomous driving”. In: *Journal of Field Robotics* 37 (Nov. 2019). DOI: 10.1002/rob.21918.

[43] Dario Guidotti et al. *The Verification of Neural Networks Library (VNN-LIB)*. 2022. URL: https://www.vnnlib.org/.

[44] Abhishek Gupta et al. “Deep learning for object detection and scene perception in self-driving cars: Survey, challenges, and open issues”. In: *Array* 10 (2021), p. 100057. ISSN: 2590-0056. DOI: https://doi.org/10.1016/j.array.2021.100057. URL: https://www.sciencedirect.com/science/article/pii/S2590005621000059.

[45] William Grant Hatcher and Wei Yu. “A Survey of Deep Learning: Platforms, Applications and Emerging Research Trends”. In: *IEEE Access* 6 (2018), pp. 24411–24432. DOI: 10.1109/ACCESS.2018.2830661.

[46] Xiaowei Huang et al. “A survey of safety and trustworthiness of deep neural networks: Verification, testing, adversarial attack and defence, and interpretability”. In: *Computer Science Review* 37 (2020), p. 100270. ISSN: 1574-0137.

[47] Xiaowei Huang et al. “Safety Verification of Deep Neural Networks”. In: *Computer Aided Verification*. Ed. by Rupak Majumdar and Viktor Kunčak. Cham: Springer International Publishing, 2017, pp. 3–29. ISBN: 978-3-319-63387-9.

[48] Yu Huang and Yue Chen. *Autonomous Driving with Deep Learning: A Survey of State-of-Art Technologies*. 2020. arXiv: 2006.06091 [cs.CV].

[49] Michael Iliadis, Leonidas Spinoulas, and Aggelos K. Katsaggelos. “Deep fully-connected networks for video compressive sensing”. In: *Digital Signal Processing* 72 (2018), pp. 9–18. ISSN: 1051-2004. DOI: https://doi.org/10.1016/j.dsp.2017.09.010. URL: https://www.sciencedirect.com/science/article/pii/S1051200417302130.

[50] *Intelligent transport systems — Adaptive cruise control systems — Performance requirements and test procedures*. standard ISO 15622:2018. International Organization for Standardization. URL: https://www.iso.org/standard/71515.html.

[51] *Intelligent transport systems — Full speed range adaptive cruise control (FSRA) systems — Performance requirements and test procedures*. standard ISO 22179. International Organization for Standardization. URL: https://www.iso.org/standard/40753.html.

[52] SAE International. *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*. Apr. 2021.

[53] Gareth James et al. *An introduction to statistical learning*. Vol. 112. Springer, 2013.

[54] Kichun Jo et al. "Development of Autonomous Car—Part II: A Case Study on the Implementation of an Autonomous Driving System Based on Distributed Architecture". In: *IEEE Transactions on Industrial Electronics* 62.8 (2015), pp. 5119–5132. DOI: 10.1109/TIE.2015.2410258.

[55] Kyle Julian. *NNet Format*. 2018. URL: https://github.com/sisl/NNet.

[56] Amit Juyal, Sachin Sharma, and Priya Matta. "Deep Learning Methods for Object Detection in Autonomous Vehicles". In: *2021 5th International Conference on Trends in Electronics and Informatics (ICOEI)*. 2021, pp. 751–755. DOI: 10.1109/ICOEI51242.2021.9452932.

[57] Dvijotham K. et al. "A dual approach to scalable verification of deep networks". In: vol. 2. Cited by: 69. 2018, pp. 550–559. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85059387384&partnerID=40&md5=6a8e12bfcb98a1ae9a844e5b113f6560.

[58] Andrej Karpathy. *ConvNetJS Deep Q Learning Demo*. https://cs.stanford.edu/people/karpathy/convnetjs/demo/rldemo.html. Accessed: 2022-03-01.

[59] Guy Katz et al. "Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks". In: *Computer Aided Verification*. Ed. by Rupak Majumdar and Viktor Kunčak. Cham: Springer International Publishing, 2017, pp. 97–117. ISBN: 978-3-319-63387-9.

[60] Guy Katz et al. *Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks*. 2017. DOI: 10.48550/ARXIV.1702.01135. URL: https://arxiv.org/abs/1702.01135.

[61] Guy Katz et al. "The Marabou Framework for Verification and Analysis of Deep Neural Networks". In: *Computer Aided Verification*. Springer International Publishing, 2019, pp. 443–452. ISBN: 978-3-030-25540-4.

[62] Alex Kendall et al. "End-To-End Learning of Geometry and Context for Deep Stereo Regression". In: *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. Oct. 2017.

[63] Nitish Shirish Keskar et al. "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima". In: *International Conference on Learning Representations*. 2017. URL: https://openreview.net/forum?id=H1oyRlYgg.

[64] Cheol-jin Kim et al. "End-to-end deep learning-based autonomous driving control for high-speed environment". In: *The Journal of Supercomputing* 78.2 (Feb. 2022), pp. 1961–1982. ISSN: 1573-0484. DOI: 10.1007/s11227-021-03929-8. URL: https://doi.org/10.1007/s11227-021-03929-8.

[65] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *arXiv e-prints*, arXiv:1412.6980 (Dec. 2014), arXiv:1412.6980. arXiv: 1412.6980 [cs.LG].

[66] B Ravi Kiran et al. “Deep Reinforcement Learning for Autonomous Driving: A Survey”. In: *IEEE Transactions on Intelligent Transportation Systems* (2021), pp. 1–18. DOI: 10.1109/TITS.2021.3054625.

[67] Alex Krizhevsky. “Learning Multiple Layers of Features from Tiny Images”. In: 2009.

[68] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[69] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Commun. ACM* 60.6 (May 2017), pp. 84–90. ISSN: 0001-0782. DOI: 10.1145/3065386. URL: https://doi.org/10.1145/3065386.

[70] Felix Kunz et al. “Autonomous driving at Ulm University: A modular, robust, and sensor-independent fusion approach”. In: *2015 IEEE Intelligent Vehicles Symposium (IV)*. 2015, pp. 666–673. DOI: 10.1109/IVS.2015.7225761.

[71] Sampo Kuutti et al. “A Survey of Deep Learning Applications to Autonomous Vehicle Control”. In: *IEEE Transactions on Intelligent Transportation Systems* 22.2 (2021), pp. 712–733. DOI: 10.1109/TITS.2019.2962338.

[72] Sampo Kuutti et al. “End-to-end Reinforcement Learning for Autonomous Longitudinal Control Using Advantage Actor Critic with Temporal Context”. In: *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*. 2019, pp. 2456–2462. DOI: 10.1109/ITSC.2019.8917387.

[73] *Large scale visual recognition challenge*. URL: http://www.image-net.org/challenges/LSVRC/.

[74] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep learning”. In: *Nature* 521.7553 (May 2015), pp. 436–444. ISSN: 1476-4687. DOI: 10.1038/nature14539. URL: https://doi.org/10.1038/nature14539.

[75] Donghan Lee et al. “Convolution neural network-based lane change intention prediction of surrounding vehicles for ACC”. In: *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*. 2017, pp. 1–6. DOI: 10.1109/ITSC.2017.8317874.

[76] Klas Leino, Zifan Wang, and Matt Fredrikson. “Globally-Robust Neural Networks”. In: *Proceedings of the 38th International Conference on Machine Learning*. Ed. by Marina Meila and Tong Zhang. Vol. 139. Proceedings of Machine Learning Research. PMLR, 18–24 Jul 2021, pp. 6212–6222. URL: https://proceedings.mlr.press/v139/leino21a.html.

[77] Francesco Leofante et al. "Automated Verification of Neural Networks: Advances, Challenges and Perspectives". In: *arXiv e-prints*, arXiv:1805.09938 (May 2018), arXiv:1805.09938. arXiv: 1805.09938 [cs.AI].

[78] Philippe Leray and Patrick Gallinari. *Feature Selection with Neural Networks*. Research Report lip6.1998.012. LIP6, Mar. 1998. URL: https://hal.archives-ouvertes.fr/hal-02547719.

[79] Jianlin Li et al. "Analyzing Deep Neural Networks with Symbolic Propagation: Towards Higher Precision and Faster Verification". In: *Static Analysis*. Ed. by Bor-Yuh Evan Chang. Cham: Springer International Publishing, 2019, pp. 296–319. ISBN: 978-3-030-32304-2.

[80] Changliu Liu et al. "Algorithms for Verifying Deep Neural Networks". In: *Foundations and Trends® in Optimization* 4.3-4 (2021), pp. 244–404. ISSN: 2167-3888. DOI: 10.1561/2400000035. URL: http://dx.doi.org/10.1561/2400000035.

[81] Alessio Lomuscio and Lalit Maganti. "An approach to reachability analysis for feed-forward ReLU neural networks". In: *CoRR* abs/1706.07351 (2017). arXiv: 1706.07351. URL: http://arxiv.org/abs/1706.07351.

[82] Jonathan Long, Evan Shelhamer, and Trevor Darrell. "Fully convolutional networks for semantic segmentation". In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015, pp. 3431–3440. DOI: 10.1109/CVPR.2015.7298965.

[83] Xin Lv et al. "A Novel Approach for Detecting Road Based on Two-Stream Fusion Fully Convolutional Network". In: *2018 IEEE Intelligent Vehicles Symposium (IV)*. 2018, pp. 1464–1469. DOI: 10.1109/IVS.2018.8500551.

[84] Mohamed Magdy, Abdallah El Marhomy, and Mahmoud A. Attia. "Modeling of inverted pendulum system with gravitational search algorithm optimized controller". In: *Ain Shams Engineering Journal* 10.1 (2019), pp. 129–149. ISSN: 2090-4479. DOI: https://doi.org/10.1016/j.asej.2018.11.001. URL: https://www.sciencedirect.com/science/article/pii/S209044791830073X.

[85] Aravindh Mahendran and Andrea Vedaldi. "Understanding deep image representations by inverting them". In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015, pp. 5188–5196. DOI: 10.1109/CVPR.2015.7299155.

[86] Till Menzel, Gerrit Bagschik, and Markus Maurer. "Scenarios for Development, Test and Validation of Automated Vehicles". In: *2018 IEEE Intelligent Vehicles Symposium (IV)*. 2018, pp. 1821–1827.

[87] Shervin Minaee et al. "Image Segmentation Using Deep Learning: A Survey". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2021), pp. 1–1. DOI: 10.1109/TPAMI.2021.3059968.

[88] Matthew Mirman, Timon Gehr, and Martin Vechev. "Differentiable Abstract Interpretation for Provably Robust Neural Networks". In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, Oct. 2018, pp. 3578–3586. URL: https://proceedings.mlr.press/v80/mirman18b.html.

[89] Andrew Moore. *Efficient Memory-based Learning for Robot Control*. Tech. rep. Pittsburgh, PA: Carnegie Mellon University, Nov. 1990.

[90] Seyed Sajad Mousavi, Michael Schukat, and Enda Howley. "Deep Reinforcement Learning: An Overview". In: *Proceedings of SAI Intelligent Systems Conference (IntelliSys) 2016*. Ed. by Yaxin Bi, Supriya Kapoor, and Rahul Bhatia. Cham: Springer International Publishing, 2018, pp. 426–440. ISBN: 978-3-319-56991-8.

[91] Sajjad Mozaffari et al. "Deep Learning-Based Vehicle Behavior Prediction for Autonomous Driving Applications: A Review". In: *IEEE Transactions on Intelligent Transportation Systems* 23.1 (2022), pp. 33–47. DOI: 10.1109/TITS.2020.3012034.

[92] Urs Muller et al. "Off-Road Obstacle Avoidance through End-to-End Learning". In: *Advances in Neural Information Processing Systems*. Ed. by Y. Weiss, B. Schölkopf, and J. Platt. Vol. 18. MIT Press, 2006. URL: https://proceedings.neurips.cc/paper/2005/file/fdf1bc5669e8ff5ba45d02fded729feb-Paper.pdf.

[93] Nina Narodytska et al. "Verifying Properties of Binarized Deep Neural Networks". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 32.1 (Apr. 2018). URL: https://ojs.aaai.org/index.php/AAAI/article/view/12206.

[94] Yuval Netzer et al. "Reading Digits in Natural Images with Unsupervised Feature Learning". In: *NIPS Workshop on Deep Learning and Unsupervised Feature Learning 2011*. 2011. URL: http://ufldl.stanford.edu/housenumbers/nips2011_housenumbers.pdf.

[95] Nexar. *Challenge: Using deep learning for traffic light recognition*. 2016. URL: https://challenge.getnexar.com/challenge-1/.

[96] David A. Noever and Samantha E. Miller Noever. "Overhead MNIST: A Benchmark Satellite Dataset". In: *CoRR* abs/2102.04266 (2021). arXiv: 2102.04266. URL: https://arxiv.org/abs/2102.04266.

[97] Matthew O'Kelly, Houssam Abbas, and Rahul Mangharam. "Computer-aided design for safe autonomous vehicles". In: *2017 Resilience Week (RWS)*. 2017, pp. 90–96. DOI: 10.1109/RWEEK.2017.8088654.

[98] Yunpeng Pan et al. "Imitation learning for agile autonomous driving". In: *The International Journal of Robotics Research* 39.2-3 (2020), pp. 286–302. DOI: 10.1177/0278364919880273.

[99] Nicolas Papernot et al. “Distillation as a Defense to Adversarial Perturbations Against Deep Neural Networks”. In: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016, pp. 582–597. DOI: 10.1109/SP.2016.41.

[100] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[101] Crubille Paul et al. “Validation of Safety Necessities for a Safety-Bag Component in Experimental Autonomous Vehicles”. In: *2018 14th European Dependable Computing Conference (EDCC)*. 2018, pp. 33–40. DOI: 10.1109/EDCC.2018.00017.

[102] Chris Paxton et al. “Combining neural networks and tree search for task and motion planning in challenging environments”. In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2017, pp. 6059–6066. DOI: 10.1109/IROS.2017.8206505.

[103] Karl Pearson. “Note on regression and inheritance in the case of two parents”. In: *Proceedings of the Royal Society of London* 58 (1895), pp. 240–242.

[104] Karl Pearson. “X. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling”. In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 50.302 (July 1900), pp. 157–175. DOI: 10.1080/14786440009463897. URL: https://doi.org/10.1080/14786440009463897.

[105] Jonathan Peck et al. “Lower bounds on the robustness to adversarial perturbations”. In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper/2017/file/298f95e1bf9136124592c8d4825a06fc-Paper.pdf.

[106] Kexin Pei et al. “DeepXplore: Automated Whitebox Testing of Deep Learning Systems”. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP ’17. Shanghai, China: ACM, 2017, pp. 1–18. ISBN: 978-1-4503-5085-3. DOI: 10.1145/3132747.3132785. URL: http://doi.acm.org/10.1145/3132747.3132785.

[107] Jordan B. Pollack. “Recursive distributed representations”. In: *Artificial Intelligence* 46.1 (1990), pp. 77–105. ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(90)90005-K.

[108] B.T. Polyak. "Some methods of speeding up the convergence of iteration methods". In: *USSR Computational Mathematics and Mathematical Physics* 4.5 (1964), pp. 1–17. ISSN: 0041-5553. DOI: https://doi.org/10.1016/0041-5553(64)90137-5. URL: https://www.sciencedirect.com/science/article/pii/0041555364901375.

[109] Dean A. Pomerleau. "ALVINN: An Autonomous Land Vehicle in a Neural Network". In: *Advances in Neural Information Processing Systems*. Ed. by D. Touretzky. Vol. 1. Morgan-Kaufmann, 1989. URL: https://proceedings.neurips.cc/paper/1988/file/812b4ba287f5ee0bc9d43bbf5bbe87fb-Paper.pdf.

[110] Luca Pulina and Armando Tacchella. "An Abstraction-Refinement Approach to Verification of Artificial Neural Networks". In: *Computer Aided Verification*. Ed. by Tayssir Touili, Byron Cook, and Paul Jackson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 243–257. ISBN: 978-3-642-14295-6.

[111] F. Py and F. Ingrand. "Dependable execution control for autonomous robots". In: *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*. Vol. 2. 2004, 1136–1141 vol.2. DOI: 10.1109/IROS.2004.1389549.

[112] Yu-Hui Qu et al. "On the prediction of DNA-binding proteins only from primary sequences: A deep learning approach". In: *PLOS ONE* 12.12 (Dec. 2017), pp. 1–18. DOI: 10.1371/journal.pone.0188129. URL: https://doi.org/10.1371/journal.pone.0188129.

[113] Aditi Raghunathan, Jacob Steinhardt, and Percy Liang. "Certified Defenses against Adversarial Examples". In: *International Conference on Learning Representations*. 2018. URL: https://openreview.net/forum?id=Bys4ob-Rb.

[114] *Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems*. standard EN 50128. European Committee for Electrotechnical Standardization, 2011.

[115] Manikandasriram Srinivasan Ramanagopal et al. "Failing to Learn: Autonomously Identifying Perception Failures for Self-Driving Cars". In: *IEEE Robotics and Automation Letters* 3.4 (2018), pp. 3860–3867. DOI: 10.1109/LRA.2018.2857402.

[116] *Réglementation des limitations de vitesse en france*. 2022. URL: https://www.securite-routiere.gouv.fr/reglementation-liee-aux-risques/reglementation-de-la-vitesse-au-volant.

[117] Reuters. *Reuters-21578 dataset*. 1987. URL: http://kdd.ics.uci.edu/databases/reuters21578/reuters21578.html.

[118] Kaoutar Rhazali et al. “Fault Tolerant Deep Neural Networks for Detection of Unrecognizable Situations”. In: *10th IFAC Symposium on Fault Detection, Supervision and Safety for Technical Processes (SAFEPROCESS 2018)*. Vol. 51. 24. Warsaw, Poland, Aug. 2018, pp. 31–37. URL: https://hal.archives-ouvertes.fr/hal-01996374.

[119] Kaoutar Rhazali et al. “Tolérance aux fautes pour détecter les comportements indésirables des réseaux de neurones”. In: *12ème Congrès International Pluridisciplinaire en Qualité, Sûreté de fonctionnement et Développement durable (Qualita2017)*. Bourges, France, Aug. 2017. URL: https://hal.archives-ouvertes.fr/hal-01997210.

[120] Nicholas Rhinehart et al. “PRECOG: PREdiction Conditioned on Goals in Visual Multi-Agent Settings”. In: *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. 2019, pp. 2821–2830. DOI: 10.1109/ICCV.2019.00291.

[121] *Road vehicles — Cybersecurity engineering*. standard ISO/SAE 21434:2021. International Organization for Standardization, 2021. URL: https://www.iso.org/standard/70918.html.

[122] *Road vehicles — Functional safety*. standard ISO 26262. International Organization for Standardization, 2018. URL: https://www.iso.org/standard/68383.html.

[123] *Road vehicles — Safety and cybersecurity for automated driving systems — Design, verification and validation methods*. standard ISO/CD TR 4804. International Organization for Standardization. URL: https://www.iso.org/standard/80363.html.

[124] *Road vehicles — Safety for automated driving systems — Design, verification and validation*. standard ISO/AWI TS 5083. International Organization for Standardization. URL: https://www.iso.org/standard/81920.html.

[125] *Road vehicles — Safety of the intended functionality*. standard ISO/CD 21448. International Organization for Standardization. URL: https://www.iso.org/standard/77490.html.

[126] Frank Rosenblatt. “The perceptron: a probabilistic model for information storage and organization in the brain.” In: *Psychological review* 65 6 (1958), pp. 386–408.

[127] Wenjie Ruan, Xiaowei Huang, and Marta Kwiatkowska. “Reachability Analysis of Deep Neural Networks with Provable Guarantees”. In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*. International Joint Conferences on Artificial Intelligence Organization, July 2018, pp. 2651–2659. DOI: 10.24963/ijcai.2018/368. URL: https://doi.org/10.24963/ijcai.2018/368.

[128] Wenjie Ruan et al. "Global Robustness Evaluation of Deep Neural Networks with Provable Guarantees for the Hamming Distance". In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*. International Joint Conferences on Artificial Intelligence Organization, July 2019, pp. 5944–5952. DOI: 10.24963/ijcai.2019/824. URL: https://doi.org/10.24963/ijcai.2019/824.

[129] Andrey Rudenko et al. "Human motion trajectory prediction: a survey". In: *The International Journal of Robotics Research* 39 (June 2020), p. 027836492091744. DOI: 10.1177/0278364920917446.

[130] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. 2016. DOI: 10.48550/ARXIV.1609.04747. URL: https://arxiv.org/abs/1609.04747.

[131] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning Representations by Back-propagating Errors". In: *Nature* 323.6088 (1986), pp. 533–536. DOI: 10.1038/323533a0. URL: http://www.nature.com/articles/323533a0.

[132] Erik Seligman, Tom Schubert, and M V Achutha Kiran Kumar. "Chapter 2 - Basic formal verification algorithms". In: *Formal Verification*. Ed. by Erik Seligman, Tom Schubert, and M V Achutha Kiran Kumar. Boston: Morgan Kaufmann, 2015, pp. 23–47. ISBN: 978-0-12-800727-3. DOI: https://doi.org/10.1016/B978-0-12-800727-3.00002-2. URL: https://www.sciencedirect.com/science/article/pii/B9780128007273000022.

[133] Shital Shah et al. "AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles". In: *Field and Service Robotics*. 2017. eprint: arXiv:1705.05065. URL: https://arxiv.org/abs/1705.05065.

[134] Uri Shaham, Yutaro Yamada, and Sahand Negahban. "Understanding adversarial training: Increasing local stability of supervised models through robust optimization". In: *Neurocomputing* 307 (2018), pp. 195–204. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2018.04.027. URL: https://www.sciencedirect.com/science/article/pii/S0925231218304557.

[135] R. Simmons, C. Pecheur, and G. Srinivasan. "Towards automatic verification of autonomous systems". In: *Proceedings. 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2000) (Cat. No.00CH37113)*. Vol. 2. 2000, 1410–1415 vol.2. DOI: 10.1109/IROS.2000.893218.

[136] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014).

[137] Gagandeep Singh et al. "An Abstract Domain for Certifying Neural Networks". In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019). DOI: 10.1145/3290354. URL: https://doi.org/10.1145/3290354.

[138] Sinam Ajitkumar Singh, Takhellambam Gautam Meitei, and Swanirbhar Majumder. "6 - Short PCG classification based on deep learning". In: *Deep Learning Techniques for Biomedical and Health Informatics*. Ed. by Basant Agarwal et al. Academic Press, 2020, pp. 141–164. ISBN: 978-0-12-819061-6. DOI: https://doi.org/10.1016/B978-0-12-819061-6.00006-9.

[139] Richard Socher et al. "Dynamic Pooling and Unfolding Recursive Autoencoders for Paraphrase Detection". In: *Advances in Neural Information Processing Systems*. Ed. by J. Shawe-Taylor et al. Vol. 24. Curran Associates, Inc., 2011. URL: https://proceedings.neurips.cc/paper/2011/file/3335881e06d4d23091389226225e17c7-Paper.pdf.

[140] Richard Socher et al. "Parsing Natural Scenes and Natural Language with Recursive Neural Networks". In: *Proceedings of the 28th International Conference on International Conference on Machine Learning*. ICML'11. Bellevue, Washington, USA: Omnipress, 2011, pp. 129–136. ISBN: 9781450306195.

[141] K.G. Srinivasa et al. "Analysis of Facial Expressiveness Captured in Reaction to Videos". In: *2017 IEEE 7th International Advance Computing Conference (IACC)*. 2017, pp. 664–670. DOI: 10.1109/IACC.2017.0140.

[142] Stellantis. *Internal presentation on autonomous systems and dependability*. 2018.

[143] Richard S Sutton. "Generalization in Reinforcement Learning: Successful Examples Using Sparse Coarse Coding". In: *Advances in Neural Information Processing Systems*. Ed. by D. Touretzky, M. C. Mozer, and M. Hasselmo. Vol. 8. MIT Press, 1995. URL: https://proceedings.neurips.cc/paper/1995/file/8f1d43620bc6bb580df6e80b0dc05c48-Paper.pdf.

[144] Phawis Thammasorn et al. "Deep convolutional Triplet network for quantitative medical image analysis with comparative case study of gamma image classification". In: *2017 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. 2017, pp. 1119–1122. DOI: 10.1109/BIBM.2017.8217813.

[145] N. Theuretzbacher. ""ELEKTRA": A System Architecture that Applies New Principles to Electronic Interlocking". In: *IFAC Proceedings Volumes* 20.3 (1987). 5th IFAC/IFIP/IFORS Conference on Control in Transportation Systems (1986), Vienna, Austria, 8-11 July, 1986, pp. 329–336. ISSN: 1474-6670. DOI: https://doi.org/10.1016/S1474-6670(17)55918-2.

[146] Sebastian Thrun et al. "Stanley: The Robot That Won the DARPA Grand Challenge". In: *The 2005 DARPA Grand Challenge: The Great Robot Race*. Ed. by Martin Buehler, Karl Iagnemma, and Sanjiv Singh. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 1–43. ISBN: 978-3-540-73429-1. DOI: 10.1007/978-3-540-73429-1_1. URL: https://doi.org/10.1007/978-3-540-73429-1_1.

[147] Vincent Tjeng, Kai Y. Xiao, and Russ Tedrake. "Evaluating Robustness of Neural Networks with Mixed Integer Programming". In: *Seventh International Conference on Learning Representations - ICLR19*. 2019.

[148] Jonas Uhrig et al. "Sparsity Invariant CNNs". In: *2017 International Conference on 3D Vision (3DV)*. 2017, pp. 11–20. DOI: 10.1109/3DV.2017.00012.

[149] Caterina Urban and Antoine Miné. "A Review of Formal Methods applied to Machine Learning". working paper or preprint. Apr. 2021. URL: https://hal.sorbonne-universite.fr/hal-03192255.

[150] Chris Urmson et al. "Autonomous Driving in Traffic: Boss and the Urban Challenge". In: *AI Magazine* 30.2 (June 2009), p. 17. DOI: 10.1609/aimag.v30i2.2238. URL: https://ojs.aaai.org/index.php/aimagazine/article/view/2238.

[151] Chris Urmson et al. "Autonomous Driving in Urban Environments: Boss and the Urban Challenge". In: *The DARPA Urban Challenge: Autonomous Vehicles in City Traffic*. Ed. by Martin Buehler, Karl Iagnemma, and Sanjiv Singh. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 1–59. ISBN: 978-3-642-03991-1. DOI: 10.1007/978-3-642-03991-1_1. URL: https://doi.org/10.1007/978-3-642-03991-1_1.

[152] Augustin Viot et al. "Erroneous models in neural networks and their threats for formal verification". In: *Congrès Lambda Mu 22, " Les risques au coeur des transisitions "*. Congrès Lambda Mu 22, " Les risques au cœur des transitions ", actes complets. Le Havre, France, Oct. 2020, p. 596. URL: https://hal.archives-ouvertes.fr/hal-03144913.

[153] Kaige Wang et al. "Image Deraining and Denoising Convolutional Neural Network For Autonomous Driving". In: *2021 International Conference on High Performance Big Data and Intelligent Systems (HPBD&IS)*. 2021, pp. 241–245. DOI: 10.1109/HPBDIS53214.2021.9658464.

[154] Qiangchang Wang et al. "Multiscale Rotation-Invariant Convolutional Neural Networks for Lung Texture Classification". In: *IEEE Journal of Biomedical and Health Informatics* 22.1 (2018), pp. 184–195. DOI: 10.1109/JBHI.2017.2685586.

[155] Shiqi Wang et al. "Beta-CROWN: Efficient bound propagation with per-neuron split constraints for complete and incomplete neural network verification". In: *Advances in Neural Information Processing Systems* 34 (2021).

[156] Shiqi Wang et al. "Efficient Formal Safety Analysis of Neural Networks". In: *Advances in Neural Information Processing Systems*. Ed. by S. Bengio et al. Vol. 31. Curran Associates, Inc., 2018. URL: https://proceedings.neurips.cc/paper/2018/file/2ecd2bd94734e5dd392d8678bc64cdab-Paper.pdf.

[157] Shiqi Wang et al. "Formal Security Analysis of Neural Networks using Symbolic Intervals". In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 1599–1614. ISBN: 978-1-939133-04-5. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/wang-shiqi.

[158] Lily Weng et al. "Towards Fast Computation of Certified Robustness for ReLU Networks". In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, Oct. 2018, pp. 5276–5285. URL: https://proceedings.mlr.press/v80/weng18a.html.

[159] Tsui-Wei Weng et al. "Evaluating the Robustness of Neural Networks: An Extreme Value Theory Approach". In: *International Conference on Learning Representations*. 2018. URL: https://openreview.net/forum?id=BkUHlMZ0b.

[160] Matthew Wicker, Xiaowei Huang, and Marta Kwiatkowska. "Feature-Guided Black-Box Safety Testing of Deep Neural Networks". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Dirk Beyer and Marieke Huisman. Cham: Springer International Publishing, 2018, pp. 408–426. ISBN: 978-3-319-89960-2.

[161] Eric Wong and Zico Kolter. "Provable Defenses against Adversarial Examples via the Convex Outer Adversarial Polytope". In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, Oct. 2018, pp. 5286–5295. URL: https://proceedings.mlr.press/v80/wong18a.html.

[162] Min Wu et al. "A game-based approximate verification of deep neural networks with provable guarantees". In: *Theoretical Computer Science* 807 (2020). In memory of Maurice Nivat, a founding father of Theoretical Computer Science - Part II, pp. 298–329. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2019.05.046. URL: https://www.sciencedirect.com/science/article/pii/S0304397519304426.

[163] Weiming Xiang, Hoang-Dung Tran, and Taylor T. Johnson. "Output Reachable Set Estimation and Verification for Multilayer Neural Networks". In: *IEEE Transactions on Neural Networks and Learning Systems* 29.11 (2018), pp. 5777–5783. DOI: 10.1109/TNNLS.2018.2808470.

[164] Weiming Xiang et al. "Reachable Set Estimation and Safety Verification for Piecewise Linear Systems with Neural Network Controllers". In: *2018 Annual American Control Conference (ACC)*. 2018, pp. 1574–1579. DOI: 10.23919/ACC.2018.8431048.

[165] Kaidi Xu et al. "Fast and Complete: Enabling Complete Neural Network Verification with Rapid and Massively Parallel Incomplete Verifiers". In: *International Conference on Learning Representations*. 2021. URL: https://openreview.net/forum?id=nVZtXBI6LNn.

[166] Dameng Yu et al. "Multi-Stage Residual Fusion Network for LIDAR-Camera Road Detection". In: *2019 IEEE Intelligent Vehicles Symposium (IV)*. 2019, pp. 2323–2328. DOI: 10.1109/IVS.2019.8813983.

[167] Huan Zhang, Pengchuan Zhang, and Cho-Jui Hsieh. "RecurJac: An Efficient Recursive Algorithm for Bounding Jacobian Matrix of Neural Networks and Its Applications". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 33.01 (July 2019), pp. 5757–5764. DOI: 10.1609/aaai.v33i01.33015757. URL: https://ojs.aaai.org/index.php/AAAI/article/view/4522.

[168] Huan Zhang et al. "Efficient Neural Network Robustness Certification with General Activation Functions". In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. NIPS'18. Montréal, Canada: Curran Associates Inc., 2018, pp. 4944–4953.

[169] Huan Zhang et al. "Efficient Neural Network Robustness Certification with General Activation Functions". In: *Advances in Neural Information Processing Systems* 31 (2018), pp. 4939–4948. URL: https://arxiv.org/pdf/1811.00866.pdf.

[170] Huan Zhang et al. "General Cutting Planes for Bound-Propagation-Based Neural Network Verification". In: *Advances in Neural Information Processing Systems* (2022).

[171] Jin Zhang and Jingyue Li. "Testing and verification of neural-network-based safety-critical control software: A systematic literature review". In: *Information and Software Technology* 123 (2020), p. 106296. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2020.106296. URL: https://www.sciencedirect.com/science/article/pii/S0950584920300471.

[172] Qinghui Zhang, Chenxia Wan, and Shanfeng Bian. "Research on Vehicle Object Detection Method Based on Convolutional Neural Network". In: *2018 11th International Symposium on Computational Intelligence and Design (ISCID)*. Vol. 01. 2018, pp. 271–274. DOI: 10.1109/ISCID.2018.00068.

[173] Yang Zhao, Jianping Li, and Lean Yu. "A deep learning ensemble approach for crude oil price forecasting". In: *Energy Economics* 66 (2017), pp. 9–16. ISSN: 0140-9883. DOI: https://doi.org/10.1016/j.eneco.2017.05.023. URL: https://www.sciencedirect.com/science/article/pii/S0140988317301871.

[174] Dali Zhu et al. "DeepFlow: Deep learning-based malware detection by mining Android application for abnormal usage of sensitive data". In: *2017 IEEE Symposium on Computers and Communications (ISCC)*. 2017, pp. 438–443. DOI: 10.1109/ISCC.2017.8024568.