New Threat on Formal Verification for Neural Networks: Example and Fault Tolerance

Augustin Viot, Benjamin Lussier, Walter Schön, Armando Tacchella, Stéphane Geronimi

New threat on formal verification for neural networks: example and fault tolerance

Augustin Viot * **Benjamin Lussier** * **Walter Schön** *
Armando Tacchella ** **Stéphane Géronimi** ***

* *CNRS, UMR 7253 Heudiasyc, Sorbonne Universités, université de technologie de Compiègne, Compiègne, France*
** *DIBRIS, Università Degli Studi di Genova, Genova, Italy*
*** *Stellantis*

Abstract: This article details a new threat to NN formal verification that is well known in the formal verification of classical systems: errors in the learned model of a NN could cause the NN to pass formal verification on a property while violating the same property in real life. The solution to this threat for classical systems (which is expert reviews) is inadequate for NN due to their lack of explainability. Here, we propose a detection and recovery mechanism to tolerate it. This mechanism is based on a mathematical diversification of the system's model and the online verification of the formal safety properties. It was successfully implemented and validated on an application example, which, to our knowledge, is one of the most concrete NN formal verification in the literature: the Adaptive Cruise Control function of an autonomous car.

Keywords: Formal verification, Neural network, Fault tolerance.

1. INTRODUCTION

With the explosion of big data to learn from and their ability to generalize and offer robust solutions to problems with a huge operating domain, neural networks (NN) offer more and more applications of interest, particularly for automated agents. The use of NN in safety critical application is however not recommended currently because of their lack of dependability, mainly due to their lack of explainability and the tremendous effort required to test their whole operating domain.

Formal verification (FV) offers answers to most of these problems by guaranteeing properties on entire domain of the NN. However, several locks keep this technique from increasing sufficiently the safety of NN. In particular, this article focuses on a problem that is well known for formal verification of classical systems but has never been studied for formal verification of NN: the fact that an erroneous model can lead to validating a property which will be violated in real life. This is a severe threat for formal verification of NN, as the solution used for classical systems (expert validation) is not usable due to their lack of explainability. The major contributions of this article are threefold. First, it presents an application example of formal verification for NN very close to industrialization: an ACC function used in real autonomous cars. Second, it details on this example how an erroneous network could pass formal verification and still cause failures in real life, using an erroneous network created through fault injection. Third, it proposes and validates a detection and recovery mechanism to tolerate this new threat to NN formal verification.

In the second section, we provide a quick overview on formal verification approaches for NN. In the third section, we detail the limits that appear when applying FV to an erroneous NN. In the fourth section we introduce our application case, the modelling of an ACC (Adaptive Cruise Control) by a NN, and the verification of the NN to validate its behavior. The validation of the NN is done by testing several scenarios in a simulator. The fifth section presents the FV we did on the nominal NN. The sixth section presents the erroneous NN that we developed for our experiment. We detail the fault we injected, the evaluation of the NN, and the how we made it pass the formal verification. In section seven, we introduce a detection mechanism, and a recovery mechanism based on diversification. In section eight, we present the application of these two mechanisms to our ACC example.

2. CONCEPTS AND RELATED WORK

In this section we present some references regarding the use of neural networks for critical applications, and then classifications of methods and tools for formal verification of NN.

Neural networks showed great capabilities for applications in profiling, image recognition, sound recognition, logic games, etc. Their ability to generalize well is particularly adequate for robust applications like autonomous vehicles Huang and Chen (2020). However, they still can not be used for safety critical applications because of their lack of dependability: their use is not recommended in railways

⋆ Sponsor and financial support acknowledgment goes here. Paper titles should be written in uppercase and lowercase letters, not all uppercase.

application [1] and in general in safety critical applications. Note that several standards are being developed to clarify their use in automotive applications [2, 3, 4].

Formal verification aims to guarantee that a component verifies a property on entire input domains. It appears to us as an interesting candidate to improve NN dependability because it answers several issues regarding the dependability of NN:

- guarantees over entire ranges of the domain: when verified, a property is valid on the entire domain specified in the conditions of the property. This answers the lock regarding open environments where it is impossible to describe all situations that may occur or use test coverage as a dependability tool.
- absence of oracle : because formal verification does not check the correctness of the output, it does not suffer from the oracle problem. Indeed, it focuses on guaranteeing safety properties designed by safety experts.

Several classifications exist to classify the numerous FV methods (Urban and Miné (2021), Liu et al. (2019), Huang et al. (2020), Leofante et al. (2018)):

- Leofante et al. (2018) classifies the methods based on which architectures are supported by the tools and methods, and the type of property that the methods can prove (invariance, invertibility, equivalence).
- Huang et al. (2020) classifies the methods based on the kind 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).
- Liu et al. (2019) classifies the methods based on the kinds of proving methods (reachability, optimization, search for counterexample).
- Urban and Miné (2021) classifies the methods based on the complete and incomplete aspects of the method.

Among these several methods, we identified several tools that combine an available mature software and a precise documentation: Marabou Katz et al. (2019), MIPVerify Tjeng et al. (2019) and ERAN Gehr et al. (2018). As explained in section 5.3, we will use Marabou [5] in this article.

3. ERRONEOUS MODELING AND FORMAL VERIFICATION OF NN

In this section, we will study what could happen when a NN has incorrectly learned external relations between its inputs and outputs (due to dynamic properties or environmental evolution).

[1] EN 50128, Railway applications - communication, signalling and processing systems - software for railway control and protection systems.
[2] ISO/CD TR 4804, Road vehicles safety and security for automated driving systems design, verification and validation methods.
[3] ISO/CD 21448, Road vehicles safety of the intended functionality.
[4] ISO 26262, Road vehicles functional safety
[5] commit number: b0e29fb43b6722dfe9b5a90cc1353990aa732327

3.1 Modeling relations for FV in classical systems

In classical systems, the 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 a model of the system. This model is developed by the system's engineers and is supposed to describe the system's behavior in a scheme suitable for the chosen formal verification tool. Obviously, if the model does not correctly describe 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 this model is achieved through code reviews and functional tests.

3.2 Modeling relations for FV in NN

The model of a NN is intrinsically part of the weight and bias of the neurons of the NN, and usually impossible to understand by humans. In the same way as in classical systems, errors in the model of the NN can result 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 or even impossible to verify by experts, and testing is often limited due to the open environment that required AI techniques. To sum up our argument, formal verification for NN can not give much trust in a NN because it is partly based on a model learned by the NN, which is very hard to validate.

Verifying properties on a NN that models incorrectly the relations between its inputs, outputs, and/or other implicit state's variables may lead to verify properties that do not hold in real life. We present in the following section a more concrete example of this problem.

4. APPLICATION EXAMPLE: AN ACC FUNCTION

In this section, we will present the modelling of the autonomous driving Adaptive Cruise Control (ACC) function that is used as an experimental validation for our work.

4.1 Description of the ACC function

We chose an ACC function as an example of autonomous driving as it is still a relatively simple mathematical function, while demonstrating a lot of the locks of critical autonomous systems (such as an open environment, possible catastrophic consequences of failures, etc.). Note that an ACC function would not be implemented by a NN as it is simple enough to be written with classical control laws. We use it as an application example and the same process would apply similarly to other NN.

The 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 either to a speed limit entered by the user (*speed*

following mode), or to keep a safe distance from a car in front, which we call exo car (*distance following mode*, also called *car following mode*). The ACC function that we will consider in this article has 6 inputs:

- S_{ego}: the speed of the ego car, expressed in $m.s^{-1}$
- S_r: the relative speed of the exo car, in comparison to our car, expressed in $m.s^{-1}$. 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 150 m this input has the value 150.
- TH_{req}: the minimal time headway expressed in s. It is the minimal time gap between the exo car and the ego car, set by the user but usually dependent on road rules. For our experiment, this input will always have the value 1.5.
- 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 5.
- S_{ref}: the speed limit set by the user, expressed in $m.s^{-1}$

It has one output, the acceleration command, expressed in $m.s^{-2}$. In real vehicles this command is calculated by a function (whose algorithm is in appendix) using classical control laws. We will call this function ACC_{orig}, which is the function our NN will aim to model. In section 5.4, we will introduce ACC_{mod}, a more conservative version, that will be used for generating the training sets for all the NN of our experiment. Note that the function uses a variable called D_s that represents the safety distance the ego car must always keep with an exo car in front and is derived from the TH_{req}.

4.2 Nominal neural network

In this section, we will detail the NN that we used in our experiment to approximate the ACC function.

Architecture We use a feedforward NN architecture with ReLu activation functions. This choice was motivated by he requirements of our formal verification tool. The final architecture of our NN is composed of:

- 1 input layer with the same 6 inputs as the function ACC_{orig}: S_{ego}, S_r, D, TH_{req}, D_0, S_{ref}
- 4 hidden layers of size 64, 64, 32, 32
- 1 output layer with 3 outputs: *acceleration* the acceleration command, and the two outputs needed for formal verification (see section 5.1 for details) $futureSpeed$ and $futureTH$.

Training data To train our NN, we created a training set of 1 million elements from the function ACC_{mod}. Each element contains the 6 inputs and the corresponding 3 outputs. To improve the accuracy of the NN sufficiently to pass formal verification, we divided the training set into 8 datasets, with the data distribution described in table the appendix.

Training For the training, we shuffled then divided the training dataset into 100 batches on 1000 epochs. This process was done with the framework pytorch (version 1.15 for GPU). The optimizer used was the Adam algorithm Kingma and Ba (2015) available in pytorch, and the MSE loss function also in pytorch. We kept the default weight and biases initialization method offered in pytorch: an uniform law $U(-\sqrt{k}, \sqrt{k})$, with $k = \frac{1}{number\ of\ input\ features} = \frac{1}{6}$).

4.3 Validation framework

To validate our experiments, we used a driving simulator. The objective was to model enough situations that could be representative of the ACC functionalities. In this section, we will describe the methodology and tools we used for this validation.

Simulator We used an adapted version of the airsim simulator developed by Microsoft Shah et al. (2018). 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 used only one exo car at a time. For the track, we used a 1290m straight line, with cars staying on the left lane.

The initial condition of a scenario is defined by several state variables. For the ego car the state variables are: *initial position* this is where we start every experiments, *initial speed*, S_{ref}. For the exo car (when present in the scenario) the state variables are: *initial position* relative to the position of the ego car (set as 0), and noted D_{init}, *initial speed*, and $S_{ref-Exo}$ the speed that the exo car will reach and follow automatically.

The simulator uses a 33 ms cycle, which means that every 33 ms it updates the values of the state variables, and computes the new ego car behavior based on the ACC command sent and the new exo car state from its variables and its set behavior. The lateral control of the ego car is commanded automatically by the simulator and the lateral and longitudinal controls of the exo car are also commanded automatically by the simulator.

Scenarios for evaluation As previously stated, validating an autonomous vehicle is a tremendous task considering the quasi-infinite possible situations due to its open environment, particularly considering constrained time and resources. Thus we first decided to limit ourselves to scenarios on a straight road with one or no exo car as we focus on a longitudinal control function. We proposed 15 scenarios for this configuration, which we will partly present in this section. Obviously we do not consider them sufficient to exhaustively test the ACC function but we tried to use a systematic method to generate them. Our creation process follows the steps defined in Menzel et al. (2018). We begin by defining the scenario attributes and then refine the considered scenarios under three levels of abstraction: functional scenarios, logical scenarios and finally concrete scenarios.

The functional scenarios we used are separated in two kinds: scenarios A that have no exo car, scenarios B that have an exo car:

- Scenario A: no exo car detected is in front. The ego car is thus in **speed following mode**.

Table 1. Definition of the concrete scenarios A.1.1, A.2.2, B.1.2, B.2.1, B.3.3

	A.1	A.2	B.1	B.2	B.3
	A.1.1	A.2.2	B.1.2	B.2.1	B.3.3
S_{ego} (km/h)	27	90	130	7	50
S_{ref} (km/h)	7	120	130	70	110
initial exo speed (km/h)	X	X	7	7	50
$S_{ref-Exo}$ (km/h)	X	X	7	50	130
D (m)	X	X	150	13	32

  - Scenario A.1: the ego car drives at a 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 a lower speed than S_{ref}, and thus must accelerate before maintaining its speed at S_{ref}.
- Scenario B: an exo car is 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.
  - Scenario B.2: the ego car is following an exo car that accelerates no higher than S_{ref} and must thus follow this acceleration while staying 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 reaches S_{ref}. Then, the ego car will stay at this speed while the ego car drives away from the ego car.

From these functionnal scenarios, we created 15 concrete ones. In this article, due to the lack of place, we will only present the 5 concrete scenarios described in table 1.

Results and measures For each simulator cycle of each run, we collect all the state variables of the simulator. For each scenario, we make 10 runs to aggregate their results and obtain representative measures as the simulator is not deterministic, and the same scenario could lead to slightly difference runs.

We present a summary of our experiments in table 2 at the end of the article. The measures presented in the table are:

- M_{TH}: average time (in s) for all experiments with $TH < TH_{req}$
- M_D: average time (in s) for all experiments with $D < Ds$
- $M_{S_{ego}}$: average time (in s) for all experiments with $S_{ego} > S_{ref}$

Observing the nominal NN measures presented in table 2 we can draw two conclusions: the nominal NN has a mostly similar behavior to the ACC_{orig} function, and it maintains a safe behavior. We consider that the NN_n learned correctly the ACC function because most measures on the NN runs have a very similar value to the ones with the ACC function.

We can also say that the NN behavior is safe because in all our scenarios, the measure M_{TH} (time with $TH < TH_{req}$) and the measure M_D (time with $D < D_s$) have a 0 value. This shows that we spend no time in unsafe situation. There are however some specific situations where the measures shows singular values. As in scenarios A.1.1, the values of measures M_{TH} and M_D are also slight above 0 for both the ACC and the nominal NN (all with values around 0.33 s). This is explained because these scenarios actually start with an ego speed S_{ego} higher than the speed limit S_{ref} as the goal of the scenario is to evaluate the function's capacity to decrease its ego speed S_{ego} until it reaches the speed limit S_{ref}.

We show below graphs describing part of the behavior of the system for the scenario B.1.2 (figure 1 and 2).

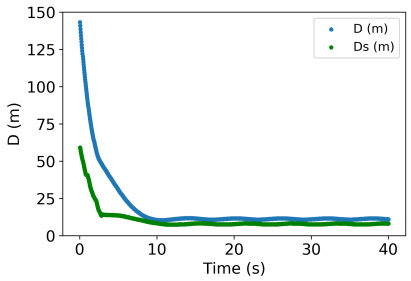

Fig. 1. Distance between the ego car and the exo car (D) compared to the safety distance (D_s) with the nominal NN for scenario B.1.2

5. FV OF OUR NN

In this section, we will detail the methods and tools we used to perform the formal verification on our NN.

5.1 Verification properties

In this subsection, we describe the property that we want to verify on the NN. We formally verified two types of properties: one safety property and four functional properties, but we only present results regarding the safety property in this article. Note that the functional properties required the addition of the $futureSpeed$ output that is present in our NN.

Restrictions on the operational domain Restraining the input space for the formal verification is necessary to be able to validate the properties as they 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 as it is a long and computationally intense process.

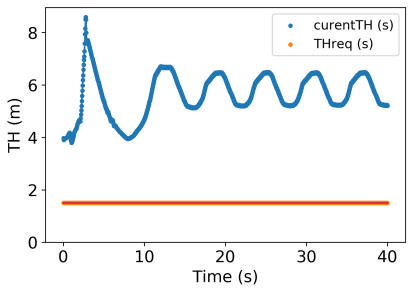

Fig. 2. Current time headway (currentTH) and required time headway (TH_{req}) with the nominal NN for scenario B.1.2

First, we implement the limitation on S_{ego}, S_r, D, S_{ref}, that correspond to the ACC ODD (Operational Design Domain): $S_{ego} \in [1.94; 41.67]$, $S_r \in [-41.67; 41.67]$, $D \in [0; 150]$. For the same reason, we set the following constant values $TH = 1.5$ and $D_0 = 5$. Finally, we add to these ODD limitations, restrictions on S_{ego} and S_r to limit the search state by excluding situations that we consider outside of the nominal situations. These constraints are $S_{exo} = S_{ego} + S_r > 0$, to exclude situations where the exo car goes backward and $S_{exo} = S_{ego} + Sr < 41.67$, to exclude situations with an exo car speed superior to 150 km/h.

Safety property We focused on one safety requirement that the ACC function should respect in 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 $futureTH > TH_{req}$ with $futureTH$ being the TH betwwen the ego car and the exo car in $delta_t = 0.5$ seconds. This $delta_t$ duration was chosen arbitrarily as a time long enough for the ego car's behavior to be impacted by the new command, but small enough for the behavior of the exo car to not change much. We also include a second precondition that excludes 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 allowed by the ACC function, (or formally $D > TH_{req} * S_{ego} + minFutureAcc - S_r * delta_t$). Indeed, we can not ask our system to do something that is physically impossible. The final formal safety property is then the following: IF (*common preconditions*) AND ($TH > TH_{req}$) AND ($D > TH_{req} * S_{ego} + minFutureAcc - S_r * delta_t$) THEN $futureTH > TH_{req}$. By using the value $TH_{req} = 1.5$, $delta_t = 0.5$, and $minAcc = -3$ we finally obtain the safety property to be validated.

Because our verification of properties uses 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. In our case, we add two outputs in addition to the acceleration command: $futureTH$ and $futureSpeed$.

5.2 Marabou tool limitations

Among the several FV for NN tools available, we chose to use Marabou, because this tool works on NNs that perform regression tasks and appeared to us as the most mature and documented alternative. It limits however the possible structure of the NN, as only with connected feed-forward and convolutional architectures with ReLu activation functions are currently permitted. In our case, a recurrent neural network architecture would have been more adequate 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 than ReLu could be more relevant in our problem. In addition, Marabou works by the searching for counter examples of the properties to be validated in the whole given input domain. A property must thus be designed in a way where the absence of counter examples guarantees the property. Finally, Marabou does not allow multiplication or division between inputs in the properties, which can restrict the formulation of some properties and thus require additional inputs or outputs in the NN.

5.3 Safety property formulation for Marabou

As explained previously, formal verification with Marabou is based on the research of a counter example. We designed our properties to match the format *if (common preconditions) AND (preconditions) then (postcondition).* The Marabou post condition of the safety property is formally described in equation 1.

$$\exists (S_{ego}, S_r, D, TH_{req}, D_0, S_{ref}, \\ acceleration, futureTH, futureSpeed) \\ such\ as, (\overline{TH_{future} > TH_{req}}) \quad (1)$$

5.4 Need for precision

During the preparation of the experiment, we quickly obtained NNs that gave satisfying functional results in our simulator but the NNs were often not precise enough to provide the formal verification of the safety property. Indeed, in some cases at the limit of the ODD, for example when $currentTH$ (the value of TH computed at the cuurent time) is very close to TH_{req}, a very small error in the NN could cause $futureTH$ to be lower than TH_{req}. To pass formal verification, we proposed and implemented two complementary design methods: first to learn a more conservative behavior for small errors to stay in the conservative margin (we call this more conservative function ACC_{mod}), second to analyze the part of the ODD that gives the most counter examples and adapt the training set distribution to better fit the formal properties. After a few iterations, we obtained in average one of of ten nominal NN that pass the formal verification. This rate is still due to the randomness and non reproducibility of the NN training as well as the presence of local minima.

6. ERRONEOUS NN FOR ACC

In this section, we will present the erroneous NN that we developed for our experiment, and how it performed with the FV and in the simulator.

6.1 Fault injection

In this section, we present the fault we injected in the nominal NN to obtain an erroneous NN.

Requirements We defined several requirements for the fault that we want to inject:

- The fault needs to alter the NN in a way that allows it to pass the formal verification but causes a failure in the system.
- The fault need to be somewhat representative of possible development faults that may occur in the development of a NN.

Fault The injected fault that we propose in this article consist in multiplying all negative accelerations from the ACC_{mod} function by 0.5. This means that the NN will send deceleration commands twice as weak as the intended ones. We consider that this fault would be representative of training the NN with data from a car that required less decelerating commands than the car that the NN will be controlling during operation. Such a fault could be caused by a multitude of causes, such as a different size or adherence of wheels, stronger brakes, or some software faults in the braking system commanding the brakes for the training data. Note that these causes would not usually generate such an important modification on the acceleration command, but we consider it satisfying because:

- measures show that the system still behaves correctly in almost all scenarios,
- this fault is proof enough that other possible faults could lead to similar consequences (causing system's failures).

Training the erroneous NN Because the erroneous training data is different from the nominal training data, we have to design a new data distribution to be precise enough to pass the formal verification. Similarly to the nominal neural network, we determined the distribution and subsets after several iterations. The final distribution is available in the appendix. The other settings of the NN training are kept the same as for the nominal NN described in 4.2.2.

6.2 FV of erroneous NN

Around one out of thirty erroneous networks passed the formal verification of the safety property. To achieve this, we used the Marabou tool in the same way than for the nominal NN described in 5.3. Note that this proportion could be increased with more iterations on the definition of the training set, and that the erroneous NN should theoretically be as easy to validate as the nominal NN, as the outputs regarding the safety property ($futureTH$ and $futureSpeed$) nor the ODD have not been modified.

6.3 Behavior of erroneous NN

In this section, we analyze the behavior of the erroneous NN (NN_e), particularly compared to the nominal NN (NN_n) using the same scenarios than in section 4.3.3. The behavior of the ego car controlled by the erroneous NN stays safe in every scenario except the scenario B.1.2. In this scenario, the erroneous NN leads to a crash with the exocar in front. We display in figure 3 the detailed distance D, in comparison with D_s and D_0 over the entire scenario. For this scenario, we also display on graph 4 the evolution of $-min(0, TH - TH_{req})$, which helps us to track if and when the safety property (corresponding to measure M_{TH}) is violated. Note that we display only one of the ten executions realized for this scenario, but the other executions were very similar. We clearly see that the property is violated around 2 seconds after the beginning of the scenario.

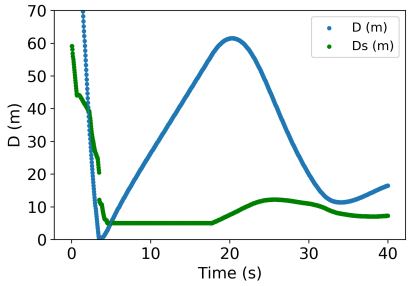

Fig. 3. Distance between the ego car and the exo car (D) compared to the safety distance (D_s) with the erroneous NN in scenario B.1.2

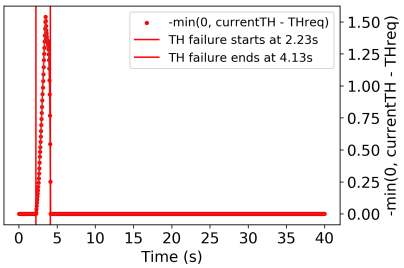

Fig. 4. System failure on the safety property with the erroneous NN in scenario B.1.2

7. DIVERSIFIED ERROR DETECTION AND RECOVERY

As proved in the previous section, formally validating a safety property on a NN does not guarantee the safety property in real life as long as the NN's model can be faulty. To solve this issue, we propose in this section a method using a mathematical model to detect inconsistencies in the NN's model online. In this way, we improve the trust in the validity of the formal validation as long as the NN's model stay consistent with the diversified mathematical model. After an error detection, we propose to recover the system by switching to a diversified NN or another control mechanism as the NN can no longer be trusted. First, we present our detection mechanism based on a diversified mathematical model (DMM). Then we discuss possible recovery techniques.

7.1 Diversified Mathematical Model

In this section, we present the theoretical explanation of our diversified mathematical model (DMM), and its limitations.

Proposed architecture Our architecture aims to ensure online (while in operation) that the NN's model on which formal verification was based on (or at least part of it) is correct. As previously discussed, formal verification will use some of the NN's inputs and outputs to verify a property, but the property could be verified while the relationships between the inputs and outputs might have been wrongly learned by the NN. In this case, the formal validation would not guarantee that the property holds in real life, as the learned model is incorrect and thus not representative of real life situations.

As explained in section 3, contrary to classical system, NN's models can not be easily verified by experts as they lack explainability and they are deeply buried in the weights and biases of every neurons.

In order to improve the trust in the formal validation, we propose to use the same knowledge of the system (which would be used to verify the model in classical systems) to check the consistency of the NN's outputs which were used during the formal verification online. This way, we can detect when we are confronted to situations where the NN has wrongly learned its model, and thus where the formal verification will not hold. Typically, this knowledge takes the form of mathematical equations of physical phenomena that link states variables of the system, some of which will be inputs and outputs of the NN.

Our detection mechanism will thus be integrated as in in the system as follows: at each cycle, the diversified mathematical model mechanism will use the NN's inputs and its decision outputs to calculate through diversified mathematical equations some state variables which are parts of the outputs of the NN. Then, the outputs of the NN and the corresponding outputs of the DMM will be compared. If the NN outputs have deviations from the DMM ones above a certain threshold, we can conclude that the NN model has been learned wrongly and thus that the formal verification can not be trusted in the current situation. We can then recover the system in a way proposed in section 7.2. Note that the properties validated during the formal verification are also good candidates as mathematical equations to be verified by our DMM detection mechanism.

Limits of the DMM The system requires the use of a system knowledge to express through mathematical expressions the relationships between the inputs and the outputs of the NN. In some cases, part of the relationships, or even all of them, might not be known. Also note that when we are unable to express mathematical equations for all the non-decisional NN outputs, the DMM detection mechanism will not be able to detect every possible NN model errors, and we could thus still have undetected inconsistencies that lead to violating verified formal properties. Thus, what we propose will improve the confidence in the formal validation, but not guarantee it completely. Other complementary mechanisms might then be needed to improve sufficiently the trust in the NN. Another limit is that the mathematical expressions might not cover the entire ODD of the NN. Another issue might be that the detection detects the error too late for the system to recover.

7.2 Recovery mechanism

A possible recovery mechanism is the use of a diversified NN trained to approximate the same function, possibly with different architecture, processes and/or data. This diversified NN would also need to have passed formal verification, and would have been checked online by the DMM detection mechanism during operation (even if its outputs would not have been used) to check whether or not errors in its model are detected. When a model error has been detected in the original NN, and if no errors were detected until now on the diversified NN, we can then recover by switching to this diversified NN. Note that when the first error detection is made, be it on the original NN or the diversified one, an alarm must be raised to inform the operator that the controlling mechanism has no more redundancies.

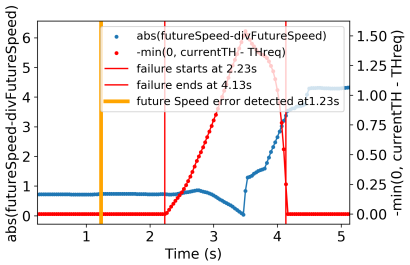

Fig. 5. futureSpeed detection indicator and system's failure

8. APPLICATION EXAMPLE OF ERROR DETECTION AND SYSTEM RECOVERY

In this section, we present an application of the error detection and recovery system presented in section 7 on the erroneous NN introduced in section 6.

8.1 Error detection in our experiment

As explained in section 7 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 physical dynamic of the vehicle. These equations link the three outputs of the NN, determining the outputs used in the formal verification from the decision taken by the NN and its inputs: equation 2 determines the diversified future speed while equation 3 computes the diversified future time headway. Finally, we also check online the property formally validated on the neural network using the diversification $futureTH_{diversified}$ and comparing it to $TH_{req} = 1.5$.

$$futureSpeed_{diversified} = S_{ego} + acceleration * delta_t - S_r \quad (2)$$

$$futureTH_{diversified} = (D + S_r * delta_t)/(S_{ego} + acceleration * delta_t) \quad (3)$$

As explained in section 7, to detect an erroneous NN's model, we check if the difference between the NN output and the diversified variables is over a certain threshold. This threshold is determined empirically by observing the maximum difference on the nominal NN to avoid false detection: $thresholdFutureSpeedDiv$=0.73 and $thresholdFutureTHDiv = 0.5$.

On scenario B.1.2, for the nominal NN no indicator exceed its threshold and the property stays validated during the experiment. On the same scenario with the erroneous NN, we can see on figure 5 that the diversified speed indicator detects the error one second before the failure.

8.2 Recovery in our experiment

In this section, we present a recovery mechanism using a diversified neural network following the first proposal of

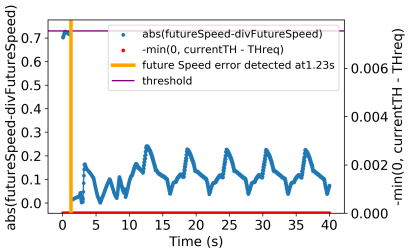

Fig. 6. future speed error indicator and system's failure indicator on fault tolerant NN in scenario B.1.2

section 7.2. When an error is detected by any of the three detectors described in 8.1, we switch from the erroneous NN to the diversified NN. We present in table 2 the results of this recovery.

We can first see that the redundancy permits to get all scenarios to respect the safety property since all measures M_{TH} are equal to 0. Thus, the error detection mechanism and the redundancy permit to avoid the failure in scenario B.1.2 that we observed in section 6.3.

This recovery can been seen in graph 6 where the future speed indicator reaches its threshold, and triggers the change of NN in control of the car. This recovery is successful in keeping the system in a safe state and avoiding the failure.

With a further analysis on the erroneous NN's behavior in table 2 we can also note that the $M_{S_{ego}}$ measure significantly improved with the recovery mechanism compared to the erroneous NN.

Table 2. Measures on five concrete scenarios for the ACC function ACC_{orig} (section 4.3.3), the nominal NN NN_n (section4.3.3), the erroneous NN NN_e (section 6.3) and the fault tolerant mechanism NN_{e_r} (section 8).

		scenarios				
		A.1.1	A.2.2	B.1.2	B.2.2	B.3.3
M_{TH}	ACC_{orig}	0.0	0.0	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0	0.0
	NN_e	0.0	0.0	2.003	0.0	0.0
	NN_{e_r}	0.0	0.0	0.0	0.0	0.0
M_D	ACC_{orig}	0.0	0.0	0.0	0.0	0.0
	NN_n	0.0	0.0	0.0	0.0	0.0
	NN_e	0.0	0.0	2.89	0.0	0.0
	NN_{e_r}	0.0	0.0	0.0	0.0	0.0
$M_{S_{ego}}$	ACC_{orig}	0.332	0.0	0.0	0.0	0.0
	NN_n	0.327	0.0	0.0	0.0	0.0
	NN_e	39.94	0.0	0.0	0.0	0.0
	NN_{e_r}	5.27	0.0	0.0	0.0	0.0

9. CONCLUSION AND PERSPECTIVES

In this article we developed a concrete example of a new threat to NN formal verification. The presented application is also currently to our knowledge the closest NN formal verification to an industrialized application in the literature: it focuses on an ACC function for autonomous vehicles on which we formally validate a safety property. We then presented an erroneous NN designed through fault injection that passes formal verification while causing a severe failure. Finally we proposed fault tolerance mechanisms that are able to detect and recover from this error. These mechanisms can be applied to most NN applications to give more trust in their formal verification. In future works, we intend to generate more erroneous networks passing formal verification and causing failures (on this example and others) in order to better study the generalized application of the proposed mechanisms. We also intend to implement other possible methods for recovery that we proposed but did not implement in this article.

REFERENCES

Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., and Vechev, M.T. (2018). Ai2: Safety and robustness certification of neural networks with abstract interpretation. *2018 IEEE Symposium on Security and Privacy (SP)*, 3–18.

Huang, X., Kroening, D., Ruan, W., Sharp, J., Sun, Y., Thamo, E., Wu, M., and Yi, X. (2020). A survey of safety and trustworthiness of deep neural networks: Verification, testing, adversarial attack and defence, and interpretability. *Computer Science Review*, 37, 100270.

Huang, Y. and Chen, Y. (2020). Survey of state-of-art autonomous driving technologies with deep learning. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 221–228.

Katz, G., Huang, D.A., Ibeling, D., Julian, K., Lazarus, C., Lim, R., Shah, P., Thakoor, S., Wu, H., Zeljić, A., Dill, D.L., Kochenderfer, M.J., and Barrett, C. (2019). The marabou framework for verification and analysis of deep neural networks. In *Computer Aided Verification*, 443–452. Springer International Publishing.

Kingma, D.P. and Ba, J. (2015). Adam: A method for stochastic optimization. In Y. Bengio and Y. LeCun (eds.), *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Leofante, F., Narodytska, N., Pulina, L., and Tacchella, A. (2018). Automated Verification of Neural Networks: Advances, Challenges and Perspectives. *arXiv e-prints*, arXiv:1805.09938.

Liu, C., Arnon, T., Lazarus, C., Barrett, C., and Kochenderfer, M.J. (2019). Algorithms for Verifying Deep Neural Networks. *arXiv e-prints*, arXiv:1903.06758.

Menzel, T., Bagschik, G., and Maurer, M. (2018). Scenarios for development, test and validation of automated vehicles. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, 1821–1827.

Shah, S., Dey, D., Lovett, C., and Kapoor, A. (2018). Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In M. Hutter and R. Siegwart (eds.), *Field and Service Robotics*, 621–635. Springer International Publishing.

Tjeng, V., Xiao, K.Y., and Tedrake, R. (2019). Evaluating robustness of neural networks with mixed integer programming. In *Seventh International Conference on Learning Representations - ICLR19*.

Urban, C. and Miné, A. (2021). A Review of Formal Methods applied to Machine Learning. *arXiv e-prints*, arXiv:2104.02466.