Erroneous models in neural networks and their threats for formal verification

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

Erroneous models in neural networks and their threats for formal verification
Erreurs de modèles dans les réseaux de neurones et leurs menaces pour la vérification formelle

Augustin Viot
CNRS, UMR 7253 Heudiasyc
Sorbonne Universités,
université de technologie de Compiègne
Compiègne, France
augustin.viot@hds.utc.fr

Benjamin Lussier
CNRS, UMR 7253 Heudiasyc
Sorbonne Universités,
université de technologie de Compiègne
Compiègne, France
benjamin.lussier@hds.utc.fr

Walter Schön
CNRS, UMR 7253 Heudiasyc
Sorbonne Universités,
université de technologie de Compiègne
Compiègne, France
walter.schon@hds.utc.fr

Stéphane Geronimi
Automotive Research Innovation and Advanced Engineering
Groupe PSA
Vélizy Villacoublay, France
stephane.geronimi@mpsa.com

Armando Tacchella
DIBRIS
Università Degli Studi di Genova
Genova, Italy
armando.tacchella@unige.it

Keywords: Neural networks dependability, formal verification

***Abstract*—EN: This article explains why current dependability techniques are not suitable for neural networks (NN). It also shows with an experiment that we need to justifiably trust neural networks modeling before formal verification can be used for critical applications.**

FR: Cet article montre les limites de l'application des techniques de sureté de fonctionnement actuelles aux réseaux de neurones. Il montre également à l'aide d'une expérience, le besoin d'apporter une confiance justifiée dans les modèles des réseaux de neurones pour permettre l'utilisation de la vérification formelle.

I. Introduction

Neural networks have shown great capabilities in many domains (images and sound recognitions, logic games, natural language processing, etc.). They are also successfully applied in some non-safety critical domains such as digital marketing, automated translation, or photo and videos manipulation. Their performances in tasks necessary for some safety critical applications also raise interest, particularly for functions such as situation recognition and decision making, where traditional systems are often unable to deal with the multiplicity of possible situations. However, their lack of dependability still prevents their use in these safety critical applications.

Indeed, current software dependability methods are effective for traditional imperative systems but are yet not suitable for declarative mechanisms such as neural networks. In particular, the complexity and unintuitive knowledge representation of neural networks make expert analysis difficult, while the vast operational context of the intended applications makes exhaustive testing impossible.

Formal verification methods appear as interesting techniques to validate safety properties on neural network for any inputs, although they are still constrained by the size of the targeted neural network.

This article presents three main contributions for the dependability of neural networks. First, we propose a decomposition of neural networks in sub-components that can facilitate the study of their faults and failures. Second, we try to categorize software development faults and the resulting failures consistently with errors studied in the machine learning community. Finally, we explain how formal verification on neural networks can be dangerous as the proven property may be guaranteed on an erroneous NN but still unsatisfied in reality. This point applies on traditional systems, but model design faults in neural networks are particularly difficult to detect and eliminate for reasons detailed in this article. A theoretical experiment using formal verification and fault injection proves our point.

This article consists in three sections. After this introduction, we will present the main concepts related to neural networks and their dependability, particularly why classical dependability methods are not effective on such systems. Then, we will details faults, errors and failures affecting neural networks, and particularly the model that they learn. Finally we will present an experiment to prove the third given contribution: that we need techniques and methods to justifiably trust a NN model for formal verification to be dependable.

II. Concepts and related work

In this section, we will introduce neural networks, the current situation regarding their use in safety critical applications, and the research that is currently done to improve their applications. Note that we focus particularly on neural networks for supervised learning in this article, as the other existing learning (reinforcement and unsupervised) are even harder to use in critical systems. Reinforcement learning can be used for training in both development and operation, or only in development. When used in development only, it is similar to supervised learning in the sense that its behavior is already fixed and what we are presenting in this article will apply. However, its use in operation implies a modification of the behavior of the system, which means its dependability must be reassessed each time it learns (a feat generally impossible in practice). With unsupervised learning, we usually do not know what the correct behavior of the system is, which means we have no ground truth for testing nor oracle to detect errors.

A. Neural networks

Neural networks (or NN) are networks of interconnected artificial neurons, usually organized in layers, each neuron giving an output as the result of an activation function on its weighted inputs and a bias. Figure 1 presents an artificial neuron with f being the activation function, x_1 to x_k the inputs from other neurons each affected by a weight w_k, b the bias of the neuron (typically added to the sum of the weighted inputs of the neuron) and y the output of the activation function applied to the inputs and the bias: $y = f(x_1, x_2, ..., x_k, w_1, w_2, ..., w_k, b)$. The connections between neurons are generally organized in layers, typically with an input layer containing the inputs x, an output layer containing the output y, and one or more hidden layers containing each several neurons. Figure 2 presents a simple neural network with two hidden layers. The connections between the neurons of the hidden layers can be designed following different properties, the most common being feedforward NN, convoluted NN, recurrent NN and deep NN. A neural network has often more than one of these properties although some combinations are in opposition (like feedforward and recurrent).

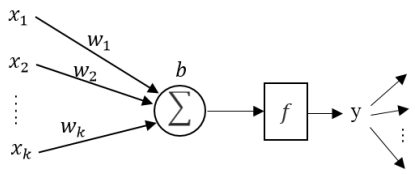


Fig. 1: Artificial neuron example

1) Neural networks as functions: All neural networks are in practice a mathematical function, assigning an output y (y being a m dimensional vector) to an input x (x being a n dimensional vector). We define in this article the computed output as $F_{nn}(x) = y$, with F_{nn} being the neural network function. The goal of a neural network is to approximate a desired function (or some similar mathematical object) defined in this article as $F(x)$ which describes the correct behavior of the system in all possible situations. Note that $F(x)$ may not be completely specified when developing the system, primarily because of the vast operational context of the application. Moreover, $F(x)$ may be impossible to approximate correctly within the functional space set of neural network functions $\mathcal{F}$ defined by a given architecture (that is, using the same layers and connections).

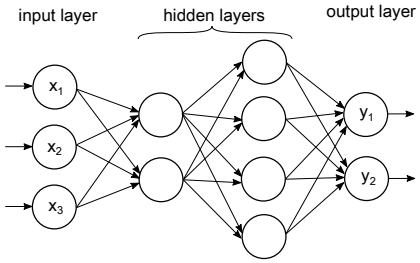


Fig. 2: Neural network example

We can distinguish two types of functions, based on two problems neural networks can be used for: classification (in which case $F_{nn}(x)$ is a piecewise-defined function whose outputs are the possible class labels) and regression (in which case $F_{nn}(x)$ is a continuous or semi-continuous function). The function F_{nn} can be computed and studied by taking into account every activation functions of every neurons, with their weights, biases and interconnections.

2) Neural networks components: In this article we propose a classification of the components of a neural network that are sufficient and necessary to completely determine its function $F_{nn}(x)$. There are two types of representation, depending whether we consider an operational fully implemented neural network in its operational phase (after the learning process has finished) or the neural network in its development phase (before the learning process has finished).

a) In the operational phase: A neural network that has finished learning can be completely determined by the description of its inputs, outputs and connections between neurons, by its activation functions, and by the values of all its weights and biases. We will call this information the NN's implementation.

b) In the development phase: We decompose the elements that completely determine a neural network during its development phase in three sub-components:

- *structure*: We define a neural network's structure as the enumeration of its *inputs and outputs*, the *description of its connections*, and its *activation functions*. The properties of a NN (convoluted, feedforward, etc.) are not necessary as they can be deduced from the description of its connections but are useful to mention as they imply specific behavior of the NN. Note that the difference between a NN's implementation and a NN's structure is that the weights and biases of the NN are fully determined in its implementation while they have no definite value in its structure.

- *training data*: The training data is a dataset of (x, y) examples used for learning. In machine learning, including NN learning, the training data is usually divided as followed: *training set*, *validation set*, *test set*. The training set contains the training data used for the 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 (in neural networks case, modifying the structure and the processes) 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*, and *weights and biases initialization*. The learning algorithm is the algorithm in charge of tuning the weight and biases values during learning by taking into account the errors between the outputs of the NN and the desired outputs. Each loop iteration of the learning algorithm over all the training data is called an epoch, the training data can be split into subsets called batches. The number of epochs, and 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. Regularization is the set of methods that modify the learning algorithm to avoid or decrease the overfitting problem described in section III-B2. Examples of regularization techniques include L_1 or L_2 regularization. Data preparation include all operations done on training data before the training start. For example, normalization and shuffle are common data preparation techniques. Weights and biases initialization corresponds to the determination of the initial values of the weights and biases, i.e. before the training starts.

Note that the functional space $F^*_{nn}(x)$, defined in section II-A1, is determined by the NN's structure, and some of its processes (typically the regularization and part of the learning algorithm). Also note that although a NN's implementation is sufficient to determine its function, the knowledge of its structure, processes and learning data can be useful to predict some properties on its behavior.

B. Neural networks use in safety critical applications and its limits

In this section, we will present the current situation regarding the use of neural networks in safety critical applications, and the limits of using traditional software dependability to verify neural networks.

1) Current use of neural networks in safety critical applications: Despite their ability to achieve complex tasks, the use of neural networks is still not recommended in railway applications [3] or more generally in safety critical applications [15]. In the case of automotive applications, different norms are under development to clarify the conditions and framework for the use of neural networks and machine learning techniques [1], [2], [4]. These precautions regarding neural networks use are due to the inability for neural networks to provide the sufficient and justified trust that these applications require. Many factors explain this lack of dependability:

- vast operational domain: neural networks are often used for applications with operational domains, where the state space of the system is large, and potentially infinite (such as autonomous cars, UAV, etc.). This usually lead to both incomplete specifications since all possible situations cannot be enumerated, and the impossibility to test all critical situations.
- 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.
- oracle problem: 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 approximations of $F(x)$, 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.
- 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.

2) Applying traditional verification methods to neural networks: Current dependability methods include a wide variety of tools to evaluate and verify the safety of traditional software, but most of these methods can not be applied to neural networks. For fault tolerance techniques, the oracle problem makes it difficult to detect errors since we often do not know what the correct output is and several correct approximations may exist. Traditional system recovery mechanisms are also unadapted because declarative mechanisms are often the only techniques that can be applied to the considered problem, and thus we might not have more trust in the redundant mechanisms than in the original one.

Fault prevention techniques have been developed for traditional systems as design processes and good practices, but the neural network domain is not mature enough for such practices to exist yet. Moreover, the vast operational domain, the complexity and the lack of explainability of neural networks make such tasks extremely challenging.

Fault elimination techniques also suffer from same problems. We cite in the following the broad categories of existing fault elimination techniques and their limits with neural networks:

- Static analysis: manual static analysis (code review or inspections) can not currently be used on neural networks models because their complexity and lack of explainability make them impossible to analyze.
- Mathematical proof: this field of techniques correspond to formal verification methods. Current formal verification methods for neural network are limited to small neural networks with specific architectures due to the complexity of large NN. Also note that, to the best of our knowledge, no industrial formal verification tool exists yet.
- Behavior analysis: this consists in transforming a neural network into a model that could be studied easily. This technique is limited due to neural networks vast operational context and complexity.
- Symbolic execution: this technique consists in propagating symbolic inputs through the neural network to obtain symbolic expressions for the outputs. This technique is limited by the complexity of neural networks.
- Tests: in most applications exhaustive test is impossible due to the vast operational context. Also, the oracle problem makes it difficult to automatically generate tests.

C. Neural networks dependability methods classification

As explained in the previous section, traditional dependability methods are not adapted to declarative mechanisms and neural networks. In this section, we will present works that have been done to develop new dependability techniques specifically for neural networks.

The different subsections correspond to each dependability means.

1) Fault prevention: Fault prevention approaches aim to prevent the occurrence or introduction of faults. We mostly identified formal verification approaches like [7] that works on proving properties through abstract interpretation, [12] that performs a tree search and uses a simplex algorithm to resolve a linear programming problem representing the neural network and the property, and [5] that uses a tool with a Boolean satisfiability solver (SAT solver) to perform tree search. We also identified some approaches based on symbolic execution like [9] that aims to apply symbolic execution to neural networks to identify important pixels in image classification problems. These methods have in common their limitation to small neural networks. Formal verification can also be used to prove properties, like local robustness. Note that some properties (including local robustness) require the definition of a distance, which might be hard or even impossible to determine depending on the inputs. For example, in image applications, there are still no meaningful way to significantly quantify the closeness between two images. In the absence of meaningful distance, the *Minkowski distance* (or L_p norm) is generally used but remains unable to characterize some very similar inputs (for example, two images where the second one have the same pixels that the first, but with an x offset of a few pixels).

2) Fault tolerance: We identified two fault tolerance approaches in the literature:

- diversification: this aims to use different systems to compute and compare the output given by a neural network in order to increase the level of confidence towards the neural network decision. The use of many neural networks trained differently to perform the same function is one of the considered approaches [20], but this applies mainly to a subset of problems (such as classification), as several acceptable approximations of F with different outputs can exist and make impossible comparisons and voting (as explained in the oracle problem of section II-B1).
- recognition of unknown situations: in the same work [20], techniques based on diversification are proposed to recognize situations that the neural network was not trained for. However recovery mechanisms are not proposed and once again this work mainly focuses on classification problems.

3) Fault removal: We identified testing as the main method developed among possible fault removal approaches. There are however two categories of testing methods.

- testing to evaluate performances: this corresponds to the use ot test sets in the machine learning community. The aim is to see how the neural network behaves on a set of examples to evaluate what will be its performance in operation. This verification is not enough for safety critical systems.
- testing for safety: this method aims to search for test sets that are supposed to expose criticalities in the neural network. For example, [19] uses a white box approach to generate new test sets, and [21] determines the coverage of neurons activated during the tests. These test sets could then be used to improve the neural network. However, approaches aiming to generate new test sets automatically are challenged by the oracle problem.

4) Fault forecasting: We did not find any method corresponding to this dependability means. Note that very few fault forecasting methods can be used on software components such as neural networks. SEEA (Software Errors and Effects Analysis) is currently impossible to use as it requires fault representativity studies that have not been done for NN.

D. Existing neural network verification methods

In this section, we will explain what formal verification on NN entails, and present the possible classifications of these methods.

1) Definition of formal verification for neural networks: Neural network formal verification can be seen as formally proving a property establishing a relation between inputs and outputs of a neural network on specific input domains. [17] express mathematically the most common properties in current

formal verification methods, which we will reformulate for clarity as:

$$\forall x, (x \in X) \Rightarrow (F_{nn}(x) \in Y) \quad (1)$$

with D_x the domain of F_{nn}, D_y the codomain of F_{nn}, $X \subseteq D_x$ and $Y \subseteq D_y$.

2) Classifications of formal verification methods for neural networks: There exists a wide variety of neural network formal verification methods currently being developed. Each method is usually specialized for a specific situation, with very little possibility to adapt for other uses. Many classifications were proposed for these methods and we introduce here three classifications with their criteria.

- [16] builds its classification based on two criteria: the kind of proof the method brings and the neural network structures which it can be applied to (*e.g.* Binarized neural networks, neural networks with only *ReLu* activation functions, deep neural networks...). The different kinds of proof mentioned in the article are: *invariance* that aims to prove that a neural network respects a property (or that it does not), *invertibility* that aims to prove that for a given set of outputs, the set of inputs leading to these outputs can be computed, and *equivalence* that aims to prove that two neural networks will attribute the same outputs for the same inputs.
- [10] builds its classification based on three criteria: the guarantees and precision that the method will bring on the verification of the property, the proving method, and the type of property to verify. The guarantees and precision can be of the following types. *Exact deterministic* means that the method determines exactly if the property holds for the given neural network. *Approximated* means that the method uses either an over or an under approximation of the neural network structure or of the property. *Converging* means that the method uses both over and under approximation and then convergence to verify the property. *Statistical* means that the method aims to estimate the probability that the property holds. The type of property to verify can be of the following types. *Robustness* means that the property is a local robustness property. *Reachability* means that the property aims to compute the set of outputs for a given set of inputs. *Interval* means that the property aims to compute an over estimation of the set of outputs for a given set of inputs. *Lipschitzian* means that the property aims to validate the Lipschitz continuity of the NN function, which is similar to but more restrictive than the local robustness (and requires a distance too).
- [17] builds its classification based on the proving methods. The main proving strategies include *reachability* that aims to compute the reachable set for a given set of inputs, *optimization* that represents the NN as a set of constraints and then aims to falsify the property, *search* that aims to find a counterexample that does not respect the property using either *reachability* or *optimization*.

For some formal verification methods, a tool has been developed to automate formal verification on a neural network. We identified three tools with good levels of maturity and stability, and an available documentation with a correct level of details: Marabou [13], MIPVerify [22] and ERAN [6]. We focus in this article on the tool Marabou for reasons detailed in the section IV-B1. Marabou [13] would be classified by [16] as a method aiming to verify Invariance properties and that can be applied to neural networks using *ReLu* activation functions. In [10] Marabou would be classified as a method providing exact deterministic guarantees and aiming to validate and interval properties.

III. Modeling errors and their threats for formal verification

In this section, we will underline two activities realized by neural networks: modeling and decision making. We will then present specific threats in NN, particularly faults and errors, and their consequences for NN formal verification.

A. Neural networks model

A neural network aims to approximate as a complex mathematical function an unknown desired process , referred in this article as the desired function $F(x)$ although it might not be expressed as a mathematical function. As stated previously, this desired function might be impossible to approximate mathematically with a given accuracy. The desired process usually consists in making an informed decision from a set of inputs (such as in image recognition, car lane change decision or actuators commands). By model of a NN, we identify in this article its means during operation to:

- model the relation that exists in reality between its different inputs and infer other possible important relations and variables to have a correct and sufficient representation of the current problem,
- take a correct decision from this current problem and generate the corresponding outputs.

During operation, these two activities are completely defined by the NN's implementation. A major difference between neural networks and traditional imperative software is that the models here are not explicitly detailed by the programmer, but are implicitly learned during development and then contained in NN's connections and weights and biases values. As said before, this is a cause for the lack of explainability of NN.

Also note that even if this article focuses particularly on NN models, both modeling and decision making are intrinsically linked in the NN structure. Indeed, as explained in this section, the NN has to model both the evolution of the environment and the correct decision that the system should take in a unique mathematical function. Thus, we found it impossible to make a distinction between modeling errors and decision making errors and will consider them both together in this article.

Note finally that the term model applied to NN in the learning community has another meaning that the one we will use: it commonly includes both the structure and the processes of the neural network.

B. Faults, errors and failures in neural network

In this section, we will discuss the faults, errors and failures that can affect neural networks. In particular, we focus on software development faults and their impacts on errors and failures.

1) Faults: Following the decomposition of a neural network in three subsets (structure, processes and data) as seen in II-A2, faults can affect any of these subsets.

Faults related to the structure of neural networks are incorrect choices in designing the network structure. They encompass incorrect choices in the inputs and outputs of the NN, in its type (convoluted, deep NN, etc.), in its connections (number of layers and connections between them) and in its activation functions. Incorrect choices in the input usually consists in relevant inputs that are not taken into account as input of the NN, which will inevitably cause errors 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 imprecision errors. Wrong choices in a NN's connections or activation functions may cause the NN to not achieve an approximation as precise as otherwise possible. Note that wrong choices in the NN types (convoluted, recurrent, etc.) are included in the incorrect choices in its connections.

Faults related to the training data include erroneous data (when some data in the training set is incorrect), and erroneous distribution (when the situations considered in the training set are not representative of the distribution and the importance of the real situations). The absence of data for a possible situation is a specific case of erroneous distribution.

Faults related to the processes are wrong choices in the parameters of the learning algorithm (chosen cost function, optimizer and number of batches, for example), in the parameters of the training data preparation (choice of the normalization method, of training data repartition, etc.), and in the neural network initialization (choice of initial value of weights and biases).

2) Errors: The statistical learning community distinguishes the following types of errors concerning an approximation function (such as a NN) [11]:

- *irreducible errors*, which are approximation errors that can not be reduced for a particular function by learning better but are dependent of unconsidered factors or an indeterministic behavior of the desired function. In our case, some structural faults, mainly wrong choices in the inputs and outputs of the system (and particularly the absence of relevant inputs), can cause such errors.
- *reducible errors*, which are further decomposed in [11] into bias errors and variance errors.
  - Bias errors are errors introduced due to the fact that the NN function is a simplification of the (possibly impossible to achieve) desired function. In our case, structural faults, particularly wrong choices in the NN's type and architecture as they limit the possible expression of its function, and some process faults cause such errors.
  - Variance errors correspond to errors that are due to the learning. In our case, both process faults and training data faults can cause such errors.

  The learning community also adds to these errors a final category of reducible errors called approximation errors, which signifies that the NN could have learned better using the same training data, structure and learning algorithm. They are typically due to being trapped in a local minimum during the learning process, or stopping the learning process before the difference between the NN's outputs and the training data outputs converges to a minimum. Process faults (such as the initialization of weights and biases) can cause approximation errors.

Additionally to previously mentioned errors, the machine learning community distinguishes two phenomena that can be associated to errors: overfitting and underfitting. As defined in [8] they are phenomena that appear when there is a significant difference between the ground truth and the computed output on the test set for overfitting and on the training set for underfitting. Overfitting can be seen as F_{nn} being too specialized on the training data and then imprecise during opration on examples unseen during the training. Underfitting can be seen as F_{nn} being not precise enough on the training data.

3) Failures: As a reminder, failure in a system is the deviation from 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 all can be expressed as imprecision errors from the desired function. Note however that the desired function is usually not known for learning mechanisms, 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 contest of most NN's applications. These facts are the cause of the oracle problem which makes NN validation and the use of recovery mechanisms difficult as said in section II-B2.

C. Formal verification of neural networks with erroneous model

Formal verification is a tempting method for NN's validation because it can guarantee properties over ranges of the input domain. This would answer part of the problems in testing due to possibly infinite execution contexts. However, we believe that faults in the NN's model could lead to a neural network's guaranteeing a property in its model (that is, on its approximated function) while the property is in fact not guaranteed in real life (on the desired function).

For example, let us consider the following desired function: state A leads to state D and state B leads to state E. Let us

consider an erroneous NN model where the NN's function wrongly assigns state D to state B. We will then be able to prove the following property on the NN: *state E is never reached*. However, the property is false in reality, that is on the desired function. In fact, the system will believe that it is in state D after B, while reality forces E after B.

This is equivalent in traditional systems design to using formal verification on an incorrect model. However, in traditional systems we usually validate the model with expert reviews and other such analyses. Because of the lack of explainability and the complexity of NN such a verification is difficult or even impossible in our case. How can we trust a formal verification's result when we can not trust the model on which it is applied? For formal verification to be applied trustfully to NN we need means to justifiably trust in the NN's model. This is currently not the case and dependability techniques for NN modeling are thus necessary to develop before formal verification can be applied for critical applications.

IV. Erroneous modeling experiment

In this section, we detail an experiment using formal verification and fault injection, to prove that a neural network with an erroneous model can guarantee a propriety that is false in reality, as theoretically stated in section III-C. We first present the desired function $F_{exp}(x)$ and a property that $F_{exp}(x)$ does not satisfy. We then explain how we designed our neural networks and present a developed nominal neural network nNN that closely approximates F_{exp}. Then, we introduce neural networks with injected faults, specifically faults on the training data, namely erroneous distribution and absence of data. Finally we present the formal verification of the property on the nominal neural network and the erroneous neural networks using the Marabou tool.

A. Activity, experimental neural network and injected faults

In this section, we describe first the nominal behavior of the neural network (the desired function $F_{exp}(x)$), second how we implemented our neural networks, and third which faults we injected in these neural networks.

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:

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

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

We consider that this ground truth will happen whatever the output of the neural network is: the neural network objective is only to approximate the function (similarly to perception and situation recognition functionalities). 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 $g(x)$:

$$\begin{aligned} \forall x_0, \forall x_1, (x_0 \in [-10; 10], x_1 \in \{0, 1\}) \\ \Rightarrow (g(x_0, x_1) \in [0; +\infty[) \end{aligned} \tag{3}$$

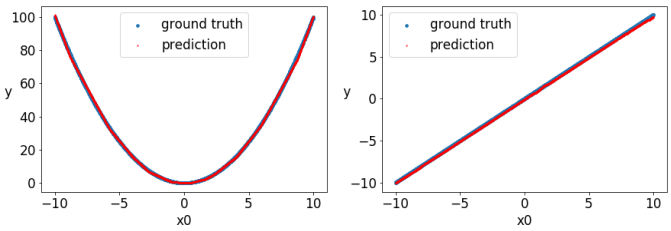
In other words, $g(x)$ is always positive on the specified input domain, which corresponds to the input domain of $F_{exp}(x)$. However, this property is obviously false for $F_{exp}(x)$ as, for any negative x_0, $F_{exp}(x)$ is negative when $x_1 = 1$.

2) Experimental neural networks: We will here explain how we developed our neural networks by detailing the three components presented in section II-A2.

- Structure: All neural networks in our study 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 yet to pretend that it is necessary in our case. 50% of the 10000 examples were given a x_1 value equal to 0, 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 $x1 = 1$), 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 [18]. To perform the training, we first shuffled our training data, then we divided our training set in 100 batches, and performed 200 epochs. The loss function used was the MSE (Mean Squared Error), and the optimization algorithm was the Adam algorithm (proposed by [14]) implemented in pytorch. The weights and biases initialization is done automatically by pytorch following this pattern: the values of each weights and biases are generated using a random uniform law $U(-\sqrt{k}, \sqrt{k})$, with $k = \frac{1}{numberofinputfeatures} = \frac{1}{2}$. The default implementation of the Adam algorithm used by pytorch does not use regularization techniques.

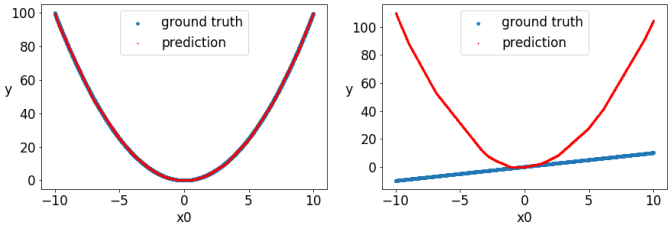
To evaluate the quality of the approximation done by neural networks developed as proposed, we implemented a nominal neural network we call nNN, and its mathematical function $F_{nNN}(x)$. We plotted its outputs in Figure 3a and 3b. We can see that the approximation is very close to the ground truth (in fact, the two curves cannot even be distinguished on the

[1] version 1.15 for gpu



(a) for $x_1 = 0$ (b) for $x_1 = 1$

Fig. 3: Ground truth and output of a nominal neural network developed as in IV-B2



(a) for $x_1 = 0$ (b) for $x_1 = 1$

Fig. 4: Ground truth and output of a erroneous neural network eNN_0 developed as defined in IV-B2

figures). 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.

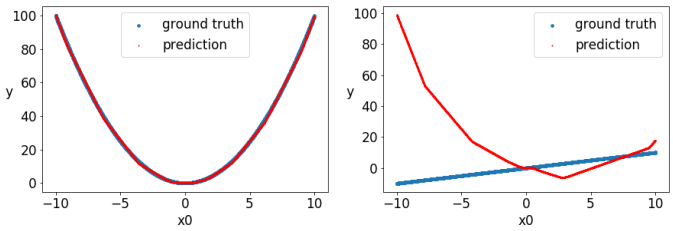
3) Erroneous neural networks: The erroneous neural networks were generated using the same structure and processes as the nominal neural network. However, we injected faults in the training data by modifying the quantity of examples where $x_1 = 1$ (erroneous distribution and absence of situation). We present in this article 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 4a and 4b show the outputs of eNN_0 compared to the ground truth, and figures 5a and 5b show the same for $eNN_{0.05}$. We can see in both cases that they approximate correctly the outputs 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 $F_{nNN}(x_0, 1)$ impacted by the very few examples where $x_1 = 1$ in the training set.

B. Formal verification

In this section, we present the formal verification of the property defined in IV-B1 on the three developed neural networks: nNN, eNN_0 and $eNN_{0.05}$.

1) Choice of the Marabou tool: Among the variety of formal verification techniques presented in the classifications of section II-D2, Marabou [13], MIPVerify [22] and ERAN [6] were potential candidates for our verification. We chose to



(a) for $x_1 = 0$ (b) for $x_1 = 1$

Fig. 5: Ground truth and output of a erroneous neural network $eNN_{0.05}$ developed as defined in IV-B2

use Marabou[2] for the convenience that it offers when working on regression tasks like in our experiment.

To perform the verification, we used the ability of Marabou to determine if there is at least one example that satisfies given constraints (lower and upper bounds) on the different input and output variables. When no example following these constraints exists, Marabou will return the answer UNSAT. When at least one example satisfying these constraints exists, Marabou will return the answer SAT with the example that it found.

2) Adaptation of the properties to fit Marabou: To perform the verification with Marabou, we had to put property (3) under a form on which Marabou could find a counterexample (property (5)). To do that, we go through the following steps:

We first obtain the negation of (3) which is the property (4):

$$\begin{aligned} \exists x_0, \exists x_1, (x_0 \in [-10; 10], x_1 \in \{0, 1\}) \\ \wedge \neg (g(x_0, x_1) \in [0; +\infty[) \end{aligned} \tag{4}$$

Note that (4) can be reformulated as (5), which is very close to what we will express in the Marabou tool:

$$\begin{aligned} \exists x_0, \exists x_1, (x_0 \in [-10; 10], x_1 \in \{0, 1\}) \\ \wedge (g(x_0, x_1) \in]-\infty; 0[) \end{aligned} \tag{5}$$

Proving that (5) is unsatisfied will be equivalent to proving that (3) is satisfied, and the former is a task that Marabou can do. Thus when asking Marabou to find an example that verifies (5), getting an UNSAT answer will mean that the property (3) is true, while getting a SAT answer will mean that the property (3) 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). 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 still have negative outputs even for the square function around the point $(x_0, x_1) = (0, 0)$. Note that the absolute value of this chosen number (0.3) is very close to the mean absolute

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

values of nNN's errors (0.254). The property we actually verify can be then written as (6) and what we will ask Marabou as (7).

$$\begin{aligned} &\forall x_0, \forall x_1, (x_0 \in [-10; 10], x_1 \in \{0; 1\}) \\ &\Rightarrow (g(x_0, x_1) \in]-\infty; -10000[\cup]-0.3; \infty[) \end{aligned} \tag{6}$$

$$\begin{aligned} &\exists x_0, \exists x_1, (x_0 \in [-10; 10], x_1 \in \{0, 1\}) \\ &\wedge (g(x_0, x_1) \in [-10000; -0.3]) \end{aligned} \tag{7}$$

Finally, to set x_1 to either 0 or 1 during the verification, we split (7) into two properties: one for $x_1 = 0$ and another $x_1 = 1$.

Here is the code that we use to verify property (7) 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 exists no (x_0, x_1, y) example that satisfies the given constraints, and that (6) is true for $x_1 = 1$.

Note that when analyzing the results, we will also mention property (8) which is similar to (3) but with the constraint $g(x_0, x_1) \in]-6.5; +\infty[$. This property is obviously less constraining than (3).

In a similar manner, we actually transform (8) into (9) and split the latter into two properties: one for $x_1 = 0$ and another $x_1 = 1$.

$$\begin{aligned} &\forall x_0, \forall x_1, (x_0 \in [-10; 10], x_1 \in \{0; 1\}) \\ &\Rightarrow (g(x_0, x_1) \in [-6.5; \infty[) \end{aligned} \tag{8}$$

$$\begin{aligned} &\forall x_0, \forall x_1, (x_0 \in [-10; 10], x_1 \in \{0; 1\}) \\ &\Rightarrow (g(x_0, x_1) \in]-\infty; -10000[\cup]-6.5; \infty[) \end{aligned} \tag{9}$$

The code that we use to verify property (9) 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, and that (9) is true for $x_1 = 1$.

3) Results of the formal verification: In this section, we present the results of the verification done with marabou on each property, and give a conclusion of these results.

a) For nNN: Marabou gives the following results:

Due to the two examples found, we can see that nNN does not respect the properties (6) and (8) for $x_1 = 1$ (respectively equivalent to (7) and (9)), and thus nNN has negative outputs (strictly, outputs below -0.3). These results are consistent with figures 3a and 3b.

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

TABLE I: Formal verification for nNN

property	result
(7) for $x_1 = 0$	UNSAT
(7) for $x_1 = 1$	UNSAT
(9) for $x_1 = 0$	UNSAT
(9) for $x_1 = 1$	UNSAT

TABLE II: Formal verification for eNN_0

b) For eNN_0: injected with an absence of situation fault for $x_1 = 1$, Marabou gives the following results:

We can see that in every cases, no example has been found. We can conclude that properties (8) and (6) hold on this NN. This confirms what we proposed in Section III-C: a property has been formally verified on an (erroneous) NN, while it 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 NN modeled the problem that the desired function represents. These results are consistent with figures 4a and 4b.

c) For $eNN_{0.05}$: injected with an erroneous situation representativity fault, lesser than the previous absence of situation fault, Marabou gives the following results:

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

TABLE III: Formal verification for $eNN_{0.05}$

We can see that although property (6) does not hold, property (8) 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 5a and 5b.

V. Conclusion and perspectives

In this article, we introduced basic concepts of neural networks, and proposed a decomposition in sub-components of a neural network during its development phase that are necessary and sufficient to determine its future behavior. We then presented several reasons why traditional dependability techniques are not well suited for declarative mechanisms and neural networks in particular. We also detailed how these reasons impede each dependability means. We then introduced

a fault classification based on our proposed decomposition in sub-components and explained how they are related to errors known in the machine learning community. Finally, we explained that formal verification on neural networks can be dangerous as the proven property may be unsatisfied in reality but still guaranteed on an erroneous NN suffering from any of the faults previously presented. As these faults are still very difficult to detect and eliminate, we believe that this is an important lock for formal verification in neural network. We detailed in the last section of this article a theoretical experiment based on formal verification and fault injection to prove this assertion.

This article underlines the importance of justifiably trusting a NN's model in order to be able to formally verify safety properties on it. As such, the main perspectives of this article are to develop techniques to improve the confidence in a neural network ability to correctly learn and represent the real world model. Moreover, the experiment presented in section IV could be realized on a less theoretical example. We could use an autonomous vehicle application, such as an ACC (automated cruise control) which typically use the vehicle current speed and current distance to a preceding vehicle as inputs to generate an acceleration command. We could then show that the relation between distance, speed an acceleration is erroneously modeled in a NN. This would be done by proving that formal verification guarantees under some conditions that the vehicle will not hit the previous car, while in reality (i. e. according to a precise and correct dynamical model) it could.

ACKNOWLEDGEMENT

The authors would like to thank Yves Grandvalet from Heudiasyc and Patrick Boutard from Groupe PSA for their helpful comments and inputs.

REFERENCES

[1] Road vehicles safety and security for automated driving systems design, verification and validation methods. standard ISO/CD TR 4804, International Organization for Standardization.

[2] Road vehicles safety of the intended functionality. standard ISO/CD 21448, International Organization for Standardization.

[3] Railway applications - communication, signalling and processing systems - software for railway control and protection systems. standard EN 50128, European Committee for Electrotechnical Standardization, 2011.

[4] Road vehicles functional safety. standard ISO 26262, International Organization for Standardization, 2018.

[5] Ruediger Ehlers. Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks. *arXiv e-prints*, page arXiv:1705.01320, May 2017.

[6] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. Ai 2: Safety and robustness certification of neural networks with abstract interpretation. In *Security and Privacy (SP), 2018 IEEE Symposium on*, 2018.

[7] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin T. Vechev. Ai2: Safety and robustness certification of neural networks with abstract interpretation. *2018 IEEE Symposium on Security and Privacy (SP)*, pages 3–18, 2018.

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. Book in preparation for MIT Press.

[9] Divya Gopinath, Kaiyuan Wang, Mengshi Zhang, Corina S. Pasareanu, and Sarfraz Khurshid. Symbolic Execution for Deep Neural Networks. *arXiv e-prints*, page arXiv:1807.10439, July 2018.

[10] Xiaowei Huang, Daniel Kroening, Wenjie Ruan, James Sharp, Youcheng Sun, Emese Thamo, Min Wu, and Xinping Yi. A Survey of Safety and Trustworthiness of Deep Neural Networks. *arXiv e-prints*, page arXiv:1812.08342, Dec 2018.

[11] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*, volume 112. Springer, 2013.

[12] Guy Katz, Clark Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks. In Rupak Majumdar and Viktor Kunčak, editors, *Computer Aided Verification*, pages 97–117, Cham, 2017. Springer International Publishing.

[13] Guy Katz, Derek A. Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljić, David L. Dill, Mykel J. Kochenderfer, and Clark Barrett. The marabou framework for verification and analysis of deep neural networks. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 443–452, Cham, 2019. Springer International Publishing.

[14] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv e-prints*, page arXiv:1412.6980, December 2014.

[15] Zesha Kurd, Tim Kelly, and Jim Austin. Safety criteria and safety lifecycle for artificial neural networks. In *Proc. of Eunite*, 2003.

[16] Francesco Leofante, Nina Narodytska, Luca Pulina, and Armando Tacchella. Automated Verification of Neural Networks: Advances, Challenges and Perspectives. *arXiv e-prints*, page arXiv:1805.09938, May 2018.

[17] Changliu Liu, Tomer Arnon, Christopher Lazarus, Clark Barrett, and Mykel J. Kochenderfer. Algorithms for Verifying Deep Neural Networks. *arXiv e-prints*, page arXiv:1903.06758, Mar 2019.

[18] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[19] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 1–18, New York, NY, USA, 2017. ACM.

[20] Kaoutar Rhazali, Benjamin Lussier, Walter Schön, and Stéphane Géronimi. 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)*, volume 51, pages 31–37, Warsaw, Poland, August 2018.

[21] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. DeepTest: Automated Testing of Deep-Neural-Network-driven Autonomous Cars. *arXiv e-prints*, page arXiv:1708.08559, August 2017.

[22] Vincent Tjeng, Kai Xiao, and Russ Tedrake. Evaluating Robustness of Neural Networks with Mixed Integer Programming. *arXiv e-prints*, page arXiv:1711.07356, November 2017.