CAMUS: A Framework to Build Formal Specifications for Deep Perception Systems Using Simulators

Julien Girard-Satabin, Guillaume Charpiat, Zakaria Chihani, Marc Schoenauer

CAMUS: A Framework to Build Formal Specifications for Deep Perception Systems Using Simulators

Julien Girard-Satabin
julien.girard2@cea.fr

Guillaume Charpiat
guillaume.charpiat@inria.fr

Zakaria Hichem Chihani
zakaria.chihani@cea.fr

Marc Schoenauer
marc.schoenauer@inria.fr

Abstract

The topic of provable deep neural network robustness has raised considerable interest in recent years. Most research has focused on adversarial robustness, which studies the robustness of perceptive models in the neighbourhood of particular samples. However, other works have proved global properties of smaller neural networks. Yet, formally verifying perception remains uncharted. This is due notably to the lack of relevant properties to verify, as the distribution of possible inputs cannot be formally specified. We propose to take advantage of the simulators often used either to train machine learning models or to check them with statistical tests, a growing trend in industry. Our formulation allows us to formally express and verify safety properties on perception units, covering all cases that could ever be generated by the simulator, to the difference of statistical tests which cover only seen examples. Along with this theoretical formulation, we provide a tool to translate deep learning models into standard logical formulae. As a proof of concept, we train a toy example mimicking an autonomous car perceptive unit, and we formally verify that it will never fail to capture the relevant information in the provided inputs.

1 Introduction

Recent years have shown a considerable interest in designing "more robust" deep learning models. In classical software safety, asserting the robustness of a program usually consists in checking if the program respects a given specification. Various techniques can output a sound answer whether the specification is respected or not, provided it is sufficiently formally formulated. However, the deep learning field is different, since the subject of verification (the deep learning model) is actually obtained through a learning algorithm, which is not tailored to satisfy a specification by construction. In this paper, we will denote as "program" the deep learning model that is the result of a learning procedure. Such a program aims to perform a certain task, such as image classification, using statistical inference. For this, it is trained through a learning algorithm, usually involving a loss minimization by gradient descent over its parameters. This way, in most deep learning applications, there exists no formal specification

of what the program should achieve at the end of the learning phase: instead, the current dominant paradigm in statistical learning consists in learning an estimator that approximates a probability distribution, about which little is known. Failures of learning procedures, such as overfitting, are hard to quantify and describe in the form of a specification.

A particular flaw of deep learning, namely adversarial examples, has been the subject of intensive research [37, 10, 30, 21]. Recently, the quest for provable adversarial robustness has been bringing together the machine learning and formal methods communities. New tools are written, inspired by decades of work in software safety, opening new perspectives on formal verification for deep learning. However, the bulk of these works has focused on the specific issue of adversarial robustness. Apart from well-defined environments where strong prior information exists on the input space (see [23]), little work has been made on formulating and certifying specific properties of deep neural networks.

The goal of this work is to propose a framework for the general problem of deep learning verification that will allow the formulation of new properties to be checked, while still benefiting from the efforts of the formal methods community towards more efficient verification tools. We aim to leverage the techniques developed for adversarial robustness and extend the scope of deep learning verification to working on global properties. Specifically, we focus on a still unexplored avenue: models trained on simulated data, commonplace in the automotive industry. Our contribution is twofold: we first propose a formalism to express formal properties on deep perception units trained on simulated data; secondly we present an open source tool that directly translates machine learning models into a logical formula that can be used to soundly verify these properties, hence ensuring some formal guarantees. Recent work proposed to analyse programs trained on simulators [15]. Although their motivations are similar to ours, they work on abstract feature spaces without directly considering the perception unit, and they rely on sampling techniques while we aim to use sound, exhaustive techniques. Their aim is to exhibit faulty behaviour in some type of neural network controllers, while we can formally verify any type of perception unit.

The paper is structured as follows: We first describe our formulation of the problem of verification of machine learning models trained on simulated data. We then describe the translator tools, and detail its main features. Finally, we present as a first use case a synthetic toy 'autonomous vehicle' problem. We conclude by presenting the next issues to tackle.

2 Related work

2.1 Adversarial robustness: a local property

Adversarial perturbations are small variations of a given example that have been crafted so that the network misclassifies the resulting noisy example, called an *adversarial example*. More formally, given a sample x_0 in a set $\mathcal{X}$, a classification function $C : x \in \mathcal{X} \to \mathbb{R}^d$, a distortion amplitude $\varepsilon \geqslant 0$ and a distance metric $\|.\|_p$, a neural network is *locally ε-robust* if for all perturbations δ s.t. $\|\delta\|_p \leqslant \varepsilon$, $C(x_0) = C(x_0+\delta)$. To provably assert adversarial robustness of a network, the goal is then to find the *exact* minimal distortion ε. Note that this

property is local, tied to sample x_0. A global adversarial robustness property could be phrased as: A deep neural network is *globally* ε-robust if for any pair of samples $(x_1, x_2) \in \mathcal{X}^2$ s.t. $\|x_1 - x_2\|_p \leqslant \varepsilon$, $C(x_1) = C(x_2)$. Verifying this global property is intractable, thus all the work has focused on local adversarial robustness.

Since their initial discovery in [37], adversarial examples became a widely researched topic. New ways to generate adversarial examples were proposed [10, 25, 41], as well as defenses [1, 26]. Other works focus on studying the theory behind adversarial examples. While the initial work [18] suggests that adversarial examples are a result of a default in the training procedure, "bugs", recent investigations ([17, 21, 34]) suggest that (at least part of the) adversarial examples may be inherently linked to the design principles of deep learning and to their resulting effects on programs: using any input features available to decrease the loss function, including "non robust" features that are exploited by adversarial examples generation algorithms. It is important to note that their very existence may be tied with the fact that we employ deep neural network on highly-dimensional perceptual spaces such as images, where we witness counter-intuitive behaviours. In any case, their imperceptibility for humans and their capacity to transfer between networks and datasets [30] make them a potentially dangerous phenomenon regarding safety and security. For example, an autonomous car sensor unit could be fooled by a malicious agent to output false direction in order to cause accidents.

2.2 Proving global properties in non-perceptual space

It is possible to express formal properties in simpler settings than adversarial robustness. By simpler, we mean two main differences: (i) the dimensionality of the input is much lower than in typical perception cases, where most of adversarial examples occur, and (ii) the problem the program aims to solve provides an explicit description of the meaning of the inputs and outputs, making a formulation of safety property much simpler. Rephrased otherwise, the program is working on inputs whose semantics is (at least partially) defined. Since deep neural networks use simple programming concepts (*e.g.,* no loops), it is quite easy to translate them directly to a standard verification format, such as SMT-LIB [5]. Provided the inputs are sufficiently well defined, it is then possible to encode safety properties as relationships between inputs and outputs, such as inequality constraints on real values.

An example of such setting can be seen in the Anti Collision Avoidance System for Unmanned aircrafts (ACAS-Xu) [27]. Inputs correspond to aircraft sensors, and outputs to airplane commands. In such case, specifications can be directly encoded as a set of constraints on the inputs and outputs. In [23], the authors proposed an implementation of ACAS-Xu as a deep neural network, and they were able to formally prove that their program respected various safety properties.

It is important to note that the inputs of the program are here high-level information (existence of an intruder together with its position), which completely bypasses the problem of perception (as airplanes have direct access to this information, in a low-level form, through their sensors, and through communications with ground operators).

2.3 Tools for provable deep learning robustness

Critical systems perform operations whose failure may cause physical harm or great economical loss. A self-driving car is a critical system: failure of embedded software may cause harm, as seen in accidents such as [19]. In the automobile industry, one expects the airbag to resist to a given pressure, the tires to last for a lower-bounded duration, etc. As software is more and more ubiquitous in vehicles, it is natural to have high expectations for software safety as well. An attempt to meet these expectations makes use of formal methods. This general term describes a variety of techniques that aim to provide mathematically sound guarantees with respect to a given specification. In less than a decade, an impressive amount of research was undertaken to bring formal verification knowledge and tools to the field of adversarial robustness. Deep learning verification has developed tools coming from broadly two different sets of techniques; this taxonomy is borrowed from [9].

The first set is the family of exact verification methods, such as Satisfiability Modulo Theory [6]. SMT solvers perform automated reasoning on logical formulae, following a certain set of rules (a logic) on specific entities (integers, reals, arrays, etc.) described by a theory. An SMT problem consists in deciding whether, for a given formula, there exists an instantiation of the variables that makes the formula true. Programs properties and control flow are encoded as logical formulae, that specialized SMT solvers try to solve. It is possible to express precise properties, but since most SMT solvers try to be exhaustive over the search space, a careful formulation of the constraints and control flow is necessary to keep the problem tractable. It was formally proven in [23] that solving a verification problem composed of conjunction of clauses by explicit enumeration for a feedforward network is NP-hard. However, the NP-hardness of a problem does not prevent us from designing solving schemes. In this same work, the authors introduced ReLuPlex, a modified solver and simplex algorithm, that lazily evaluates ReLUs, reducing the need to branch on non-linearities. Their follow-up work [24] improves and extends the tool to support more complex networks and network-level reasoning. Others [9] rephrase the problem of adversarial robustness verification as a branch-and-bound problem and provide a solid benchmark to compare current and future algorithms on piece-wise linear networks. Other exact techniques are based on mixed integer linear programming (MILP). The verification of adversarial robustness properties on piece-wise linear networks can indeed be formulated as a MILP problem [38], and a preconditioning technique drastically reduces the number of necessary calculations. Adversarial robustness properties were thus checked on ResNets (a very deep architecture) with l_∞-bounded perturbations on CIFAR-10.

The second set of techniques in formal methods is based on overapproximating the program's behaviour. Indeed, since solving the exact verification problem is hard, some authors worked on computing a lower bound of ε, using techniques building overapproximations of the program, on which it is easier to verify properties. Abstract interpretation (first introduced in [12]) is an example of such technique. It is a mathematical framework aiming to prove sound properties on *abstracted semantics* of program. In this framework, a program's concrete executions are abstracted onto less precise, but more computationally tractable abstract executions, using numerical domains. Finding numerical domains that balance expressiveness, accuracy and calculation footprint is one of

the key challenges of abstract interpretation.

The first instance of specifically-tailored deep learning verification described how to refine non-linear sigmoid activation function to help verification [32]. [40] proposes an outer convex envelop for ReLu classifiers with linear constraints, expressing the robustness problem as a Linear Programming (LP) problem. [29] and [35] propose a framework for building abstract interpretations of neural networks, which they use to derive a tight upper bound on robustness for various architectures and for regularization. On MNIST, both works displayed a robustness of 97% bounded by a $l_\infty = 0.1$ perturbation. On CIFAR-10, they achieved a 50% robustness for a similar net with a $l_\infty = 0.006$ perturbation. Symbolic calculus on neural networks is performed in [39], allowing symbolic analysis and outperforming previous methods. A verification framework based on bounding ReLu networks with linear functions is proposed in [8].

The boundary between these two families of techniques can be blurry, and both techniques can be combined. For instance, [36] combines overapproximation and MILP techniques to provide tighter bounds on exact methods. Competing with complete methods, they verify a hard property on ACAS-Xu faster and provide precise bounds faster than other methods.

All these techniques are employed either for proving local properties (local adversarial robustness), or on simpler, non perceptual input spaces. On the opposite, our work proposes a framework to prove global properties on perceptual inputs.

3 CAMUS: a new formalism to specify and verify machine learning models

3.1 Motivation

In most deep learning application domains, such as image classification [20], object detection [11], control learning [7], speech recognition [33], or style transfer [22], there exists no formal definition of the input. Let us consider the software of an autonomous car as an example. A desirable property would be not to run over pedestrians. This property can be split in i) all pedestrians are detected, and ii) all detected pedestrians are avoided. For a formal certification, the property should be expressed in the form "For any image containing pedestrians, whatever the weather conditions or camera angle could be, all pedestrians present in that image are detected and avoided". Such a formulation supposes one is able to describe the set of all possible images containing pedestrians (together with their location). However, there exists no exact characterization of what a pedestrian is or looks like, and certainly not one that takes into account weather condition, camera angle, input type or light conditions. Any handmade characterization or model would be very tiresome to build, and still incomplete.

On the upside, machine learning has demonstrated its ability to make use of data that cannot be formally specified, yielding impressive results in all abovementioned application domains, among others; on the downside, it has also been demonstrated that ML models can easily fail dramatically, for instance when attacked with adversarial examples. Thus, manufacturers of critical systems need to provide elements that allow regulators, contractors and end-users to trust the systems in which they embed their software.

Usually, car manufacturers rely on test procedures to measure their system's performances and safety properties. But testing can, at best, yield statistical bounds on the absence of failures: The efficiency of a system against a particular situation is not assessed before this situation is actually met during a real-world experiment. As the space of possible situations is enormous (possibly infinite) and incidents are rare events, one cannot assess that an autonomous vehicle will be safe in every situation by relying on physical tests alone.

A current remedy is to use artificial data, and to augment the actual data with data generated by a simulation software, with several benefits: Removing the need to collect data with expensive and time consuming tests in the real world; Making it possible to generate potentially hazardous scenarios precisely, e.g., starting with the most common crash cases. Examples of such simulators are Carla [14] and the NVIDIA Drive Constellation system. However, even if it is possible to artificially generate corner cases more easily, the space of possible scenarios is still enormous, and some accidents remain completely unpredictable *a priori* by human test designers. For instance, in a recent car accident involving partially self-driving technology, the manufacturer admitted that the camera failed to distinguish a white truck against a bright sky [19], causing the death of the driver. Such a test case is difficult to come up with for a human, because it is the conjunction of specific environmental conditions and specific driving conditions.

Our motivation is to bring an additional layer of trust, not relying on statistical arguments, but rather on formal guarantees. Our long term objective is to be able to formalize a specification and to provide guarantees on every possible scenario, automatically finding violations of the specification. Because practitioners are now relying more and more on simulators, we propose as a first step to study such simulated setting, and to formalize it. The idea is to rephrase the verification problem in order to include both the deep learning model *and* the simulator software within the verification problem. As said earlier, a simulator offers more control on the learning data by providing explicit parameters (for instance: number and positions of pedestrians on the image).

3.2 Problem formulation and notations

Let $f : \mathcal{X} \to \mathcal{Y}$ be an algorithm taking a perceptual input $x \in \mathcal{X}$ and yielding a decision $y \in \mathcal{Y}$. The perceptual space $\mathcal{X}$ will typically be of the form $\mathbb{R}^d$ or $[0,1]^d$. In the general framework of this work, f is a program trained with a learning procedure on a finite subset of $\mathcal{X}$ to perform a specific task (e.g., drive the passengers safely home). In our example, the task would be to output a command from an image, in which case, for a given image x, $f(x)$ would be the driving action taken when in environment x.

Let us denote by $g : \mathcal{S} \to \mathcal{X}$ the simulator, that is, a function taking as input a configuration $s \in \mathcal{S}$ of parameters, and returning the result of the simulation associated to these parameter values. A configuration s of parameters contains all information needed by the simulator to generate a perceptual input; each parameter may be a discrete or continuous variable. Let us take as running example a simulator of autonomous car images: s would contain the road characteristics, the number of pedestrians and their positions, the weather conditions..., that is, potentially, thousands or millions of variables, depending on the simulator realism.

The problem to solve here is the following: *For a model f trained on data belonging to $\mathcal{X}$ generated by g to perform a certain task, how can we formulate and formally verify practical safety properties for all possible $x \in \mathcal{X}$, including samples never seen during training?*

3.3 Including the simulator in the verification

In standard settings, such as the ones schematized in Figure 1, specifications express relationships from $\mathcal{X}$ to $\mathcal{Y}$ using a formulation of f. But $\mathcal{X}$ is such a huge space that formulating properties that are non trivial, let alone verify these, is prohibitively difficult, especially in the case of perceptive systems where the domain of x cannot be specified: all matrices in $\left([0,255]^3\right)^{\#\text{pixels}}$ are images, technically, but few of them make sense, and one cannot describe which ones. Moreover, given an image x, the property to check might be difficult to express, as, to state that all pedestrians were detected and avoided, one needs to know whether there are pedestrians in x and where, which we do not know formally from just the image x. And if one had a way to retrieve such information from x (number and location of pedestrians) without any mistake, one would have already solved the initial problem, i.e., safe self-driving car.

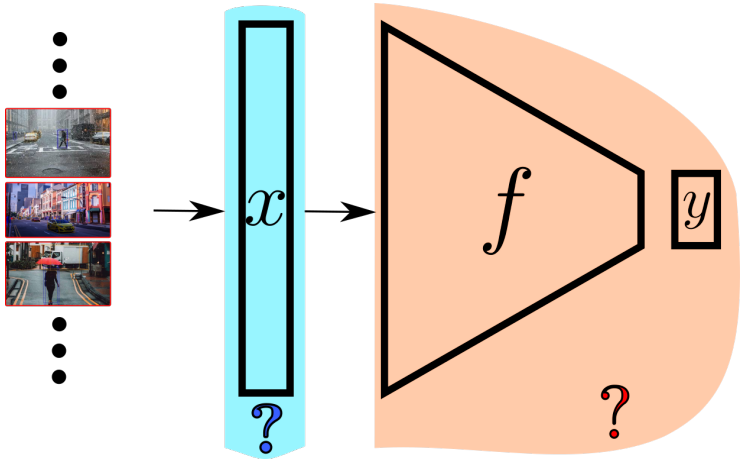

Figure 1: Natural inputs with huge perceptual space: no characterization of the input nor property can be formulated.

To summarize, in this setting, it is impossible to express a relevant space for x and a property to verify Φ:

$$\forall x \in ?, \quad \Phi?\left(f(x)\right)$$

In the setting of simulated inputs, though it remains difficult to formulate properties on the perceptual space $\mathcal{X}$, we know that this space is produced by g applied to parameters in $\mathcal{S}$. On the contrary to $\mathcal{X}$, $\mathcal{S}$ is a space where there exists an abstract, albeit simplistic characterization of entities. Indeed, setting parameters for a pedestrian in the simulated input yields a specification of what a pedestrian is in $\mathcal{X}$ according to the inner workings of g. The procedure g transforms elements $s \in \mathcal{S}$, that represent abstracted entities, into elements $x \in \mathcal{X}$ that describe these entities in the rich perceptual space. To output values

in $\mathcal{Y}$, f has to capture the inner semantics contained in $\mathcal{X}$, that is to say, to abstract back a part of $\mathcal{S}$ from $\mathcal{X}$.

The above remark is the key to the proposed framework: If we include $\mathcal{S}$ and g alongside f, $\mathcal{X}$ and $\mathcal{Y}$ in the verification problem, then all meaningful elements of $\mathcal{S}$ are de facto included. It then becomes possible to formulate interesting properties, such as "given a simulator that defines pedestrians as a certain pattern of pixels, does a model trained on the images generated by this simulator avoid all pedestrians correctly?". Formally, to ensure that the output $y = p \circ g(s)$ satisfies a property Φ for all examples $x = g(s)$ that can ever be generated by the simulator, the formula to check is of the form:

$$\forall s \in \mathcal{S}, \quad \Phi\big(s,\ p \circ g(s)\big)$$

The property Φ may depend on s indeed, as, in our running example, s explicitly contains the information about the number of pedestrians to be avoided as well as their locations.

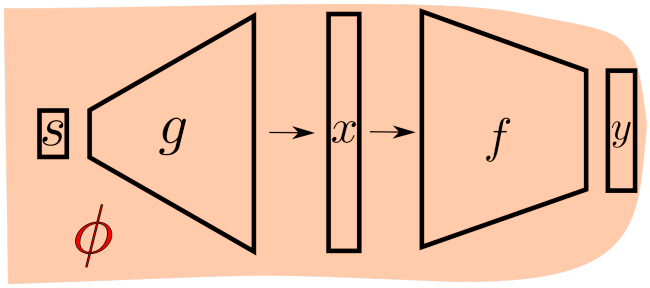

Figure 2: Generated inputs with integration of the generation procedure in the verification problem. There are now new properties to check since we have a formal characterization of the perceptual elements.

Including $\mathcal{S}$ and g in a formal property to check requires to formulate at least partially the multiple functions that compose g. Describing precisely these procedures is a key problem that we plan to address later.

As our framework relies on including the simulator in the verification problem, we call it Certifying Autonomous deep Models Using Simulators (CAMUS).

3.4 Separating perception and reasoning

Before the rise of deep learning, the perception function (which, *e.g.,* recognizes a certain pattern of pixels as a pedestrian) and the control, or reasoning function (which, *e.g.,* analyzes the location of a pedestrian and proposes a decision accordingly) in vehicles were designed and optimized separately. However, work such as [7] showed that end-to-end learning can in general be a much more efficient alternative; there exist many incentives to adopt this end-to-end architecture, mixing and training jointly the perception and control functions. However, combining perception and reasoning into one model makes the formulation of safety properties more difficult.

Thus in our description (see Fig. 3), we choose to separate the perception and the reasoning functions, respectively in the components p and r. The perception part p is in charge of capturing all relevant information contained in the image,

while the reasoning part r will make use of this relevant information to output directives accordingly to a specification.

One way to make sure that p retrieves *all* relevant information is to require it to retrieve *all* information available, that is, to reconstruct the full simulator parameter configuration s. In this setting, the output s' of the perception module p lies in the same space as the parameter configuration space $\mathcal{S}$, and the property we would like to be satisfy can be written as $p \circ g = \mathrm{Id}$, which can be rewritten as:

$$\forall s \in \mathcal{S}, \quad p \circ g(s) = s \tag{1}$$

This way, we ensure that the perception module p correctly perceives *all* samples that could ever be generated by the simulator. In the case some parameters are known not to be relevant (image noise, decoration details, etc.), one can choose not to require to find them back, therefore asking to retrieve only the other ones. For the sake of notation simplicity, we will here consider the case where we ask to reconstruct all parameters.

This separation between perception p and further reasoning r brings modularity as an additional benefit: even when dealing with different traffic regulations, it is only necessary to prove p once; the verification of compliance towards local legislations and specifications by r can be done separately. It allows to reuse the complex perception unit with different reasoning modules r without needing to re-prove it. Note also that r does not need to be as complex as p, since it will work on much smaller spaces; multiple verifications of r are then easier.

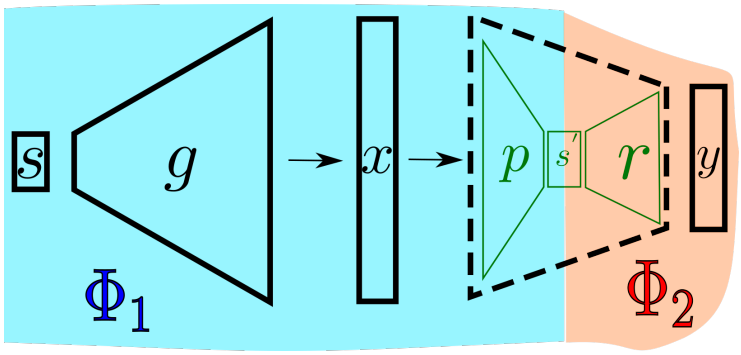

Figure 3: Integration of the generation procedure in the verification, with split between perception and reasoning: p learns to capture all the relevant parameters; r learns to respect the specification. Verifying ϕ_1 proves the perception unit once and for all; verifying ϕ_2 can be done when the specification changes (e.g., for different driving rules).

One could argue that this formulation makes the problem more complex, and it indeed may be the case. However, our proposition is aimed at safety, and in order to provide additional trust, it is sometimes necessary to formulate the problem differently. For instance, there are good practices to structure the code to provide some safety guarantees: bounded loops, correctly allocated and de-allocated references, ban of function references, ... are constructs that voluntarily restrain the expressive power of the programming language to ensure a safer behaviour. Hence, although this formulation might seem like a step back

w.r.t. the state-of-the-art, we argue that it provides a new way to formulate safety properties, and hence will be beneficial in the long run.

3.5 Properties Formulation

Considering jointly the simulator g and the ML model f, split in p and r, two families of properties are amenable to formal checking:

- Φ_1: perception unit p has captured sufficient knowledge from $\mathcal{X}$;
- Φ_2: reasoning unit r respects a specification property regarding $\mathcal{Y}$.

Type 2 properties have been addressed in the literature - see Section 2.2. The key point of the proposed approach is thus to obtain a representation space that reliably yields semantic meaning, which is the objective of Φ_1. Since the simulator is included in the verification problem, properties of family Φ_1 can be written as relationships between input parameter configurations $s \in \mathcal{S}$ and retrieved parameter configurations $s' \in \mathcal{S}$, outputs of the perception module p. Strict equality between s and s' may be difficult to achieve, and is actually not needed as long as the reasoning module r is able to deal with small estimation errors.

Expressed in the proposed formalism, the perception task is equivalent to finding (a good approximation of) $\mathcal{S}$. Thus, a relaxed version of property 1 to satisfy could be formalized as some tolerance $\varepsilon > 0$ on the reconstruction error $\|s' - s\|$ (for some metric $\|\cdot\|$):

$$\forall s \in \mathcal{S}, \quad \|s - p \circ g(s)\| \leqslant \varepsilon \tag{2}$$

3.6 Discussion

As stated earlier, it is not always necessary to retrieve all parameters of configuration s. For instance, one could seek to retrieve only the correct number of pedestrians and their locations, from any image generable by the simulator. In this case, the output of p would be just a few coefficients of s, and must consequently be characterized differently (*e.g.,* as belonging to a given subspace of $\mathcal{S}$). This would allow to express more flexible properties than simply reconstruct all parameters.

For the model f to correctly generalize, the simulated data must yield two characteristics:

1. they need to be sufficiently realistic (that is to say, they should look like real-world images); if not the network could overfit the simplistic representation provided by the simulator;
2. they must be representative of the various cases the model has to take into account, to cover sufficiently diverse situations.

Additional characterization of the simulator would be difficult. For instance, one could suggest to require the simulator g to be either surjective or injective, in order to cover all possible cases $x \in \mathcal{X}$, or for parameters to be uniquely retrievable. Yet, the largest part of the perceptual space $\mathcal{X}$ is usually made of nonsensical cases (think of random images in $\left([0, 255]^3\right)^{\#\text{pixels}}$ with each pixel

color picked independently: most are just noise), and the subspace of plausible perceptual inputs is generally not characterizable (without which the problem at hand would already be solved). Regarding injectivity, being one-to-one is actually not needed when dealing with properties such as 2.

Finally, let us consider the case where several simulators are available, and where, given a perceptive system p, we would like to assert properties of type Φ_1 for each of the simulators. At first glance, as the output of p consists of retrieved parameters, this would seem to require that all simulators are parameterized exactly identically (same $\mathcal{S}$). However, for real tasks, one does not need to retrieve all parameters but only the useful ones (*e.g.,* number of pedestrians and their locations), which necessarily appear in the configuration of all simulators. As it is straightforward to build for each simulator a mapping from its full list of parameters towards the few ones of interest, a shared space for retrieved parameters can be defined, a unique perception system p can be trained, and formal properties for all simulators can be expressed.

4 Translating neural networks into logical formulae

In previous section, we introduced two families of properties: Φ_1 involves the simulator g, its parameter space $\mathcal{S}$, the perceptual space $\mathcal{X}$ where simulated data lie, and the perception unit p; Φ_2 involves the representation space learned by p (which should be a copy of $\mathcal{S}$), the reasoning unit r and its output space $\mathcal{Y}$.

In order to be able to actually formulate properties of these families, we must first be able to represent all these elements as logical formulae. The goal of this section is to introduce ONNX2SMT, a tool to do so automatically.

ONNX2SMT provides an interface to all machine learning models that use the Open Neural Network Exchange format [2], and translates them into the standard language SMT-LIB[5], allowing all state-of-the-art generalist SMT solvers and deep learning verification specialized tools to work on a direct transcription of state-of-the-art neural networks. ONNX2SMT will be open-sourced to further help the community effort towards safer deep learning software.

4.1 ONNX and SMT-LIB

The Open Neural Network eXchange format (ONNX)[1] is a community initiative kickstarted by Facebook and Microsoft, that aims to be an open format for representing neural networks, compatible across multiple frameworks. It represents neural networks as directed acyclic graphs, each node of the graph being a call to an operation. Common operations in machine learning and deep learning are tensor multiplications, convolutions, activations functions, reshaping, etc. [2]. Operations have predecessors and successors, describing the flow of information in the network. The network parameters are also stored in the ONNX graph. A wide variety of deep learning frameworks support ONNX, including Caffe2, PyTorch, Microsoft CNTK, MatLab, SciKit-Learn and TensorFlow. Examples

[1] https://onnx.ai/

[2] Full list of supported operators is available at:
https://github.com/onnx/onnx/blob/master/docs/Operators.md

of use cases presented in the official page[3] include benchmarking models coming from different frameworks, converting a model prototyped using PyTorch to Caffe2 and deploying it on embedded software.

SMT-LIB2 is a standard language used to describe logical formulae to be solved using SMT solvers. Most state-of-the-art solvers implement a SMT-LIB support, which facilitates benchmarks and comparisons between solvers. SMT-COMP [4] is a yearly competition using SMT-LIB as its format. This challenge is a unique opportunity to present different techniques used by solvers, to increase the global knowledge of the SMT community. SMT-LIB2 supports expressing formulae using bit vectors, Boolean operators, functional arrays, integers, floating points and real numbers, as well as linear and non-linear arithmetic. In this work, only the Quantifier-Free Non linear Real Arithmetic (QF_NRA) theory will be used. Since the language aims to be compatible with a wide variety of solvers, expressivity is limited compared to languages such as Python, used by most deep learning platforms. In particular, there is no built-in Tensor type, and it is hence necessary to adapt the semantics of tensors to SMT-LIB2. This adaptation is performed by ONNX2SMT.

4.2 Features

Features of ONNX2SMT include the support of the most common operations in modern neural networks, such as tensors addition and multiplication, max-pooling and convolution on 2D inputs. Support for a wider range of operators (such as reshaping or renormalization operators) is on-going work.

ONNX2SMT uses a Neural IntErmediate Representation (NIER) to perform modifications of the deep neural network structure, for instance by following rewriting rules. NIER is still at an early stage, but future work will integrate state of the art certified reasoning and pruning.

The conversion from ONNX to NIER is performed thanks to the reference protobuf description of ONNX, converted to OCaml types using the piqi[4] tool suite. ONNX2SMT provides straightforward conversion from ONNX to SMT-LIB, using NIER as an intermediate representation.

All the features described above allow us to encode machine learning models (p and r) as SMT formulae. $\mathcal{X}$ and $\mathcal{Y}$ can be expressed directly using QF_NRA existing primitives. Future work will provide an additional mechanism to encode g and $\mathcal{S}$.

4.3 Usage

ONNX2SMT workflow can be summarized as follows:
Input: an ONNX file created using an ML framework;
1. Convert the ONNX model to NIER (onnx parser);
2. Convert NIER to a SMT-LIB (smtifyer) string, written on disk;
3. Add the property to validate to the existing SMT-LIB file;
Output: An SMT-LIB file that can be solved to prove the property.

[3] https://github.com/onnx/tutorials/
[4] https://github.com/alavrik/piqi

5 Experiments

As a proof of concept for the proposed framework, it is applied it to a simple synthetic problem. All experiments are conducted using the PyTorch framework [31]. Neural networks are trained with PyTorch, then converted into ONNX using the built-in ONNX converter and finally converted into the intermediate representation in SMT-LIB format with ONNX2SMT. We use z3[13], CVC4[3], YICES[16] and COLIBRI[28] SMT solvers as standard verification tools.

Let us consider here the perception unit of an autonomous vehicle, whose goal is to output driving directives that result in safe driving behaviour. The perception unit is modeled as a deep neural network with one output node, taking as input an image. If an obstacle lies in a pre-defined "danger zone", the network should output a "change direction" directive. Otherwise, it should output a "no change" directive.

The "simulator" is here a Python script, taking as input the number and the locations on the image of the one-pixel wide obstacles and generating the corresponding black-and-white images.

The verification problem consists in the formulation of the network structure and constraints on the inputs, and in the following properties to check:

1. verify that an input with an obstacle (or several ones) in the danger zone will always lead to the "change direction" directive;

2. verify that an input without obstacle on the danger zone will never lead to the "change direction" directive.

If both properties are verified, our model is perfect for all the inputs that can be generated. If the first one is not verified, our verification system will provide examples of inputs where our model fails, which can be a useful insight on the model flaws. Such examples could be used for further more robust training, *i.e.,* integrated into a future training phase to correct the network misclassifications. Similarly, if the second property is not verified, the solver will provide false positives, that can help designers reduce erroneous alerts and make their tools more acceptable for the end-user.

Experimental setting

In this toy example, input data are $N \times N$ black-and-white images (see Fig. 4 for examples). The space of possible simulated data $g(\mathcal{S}) \subset \mathcal{X}$ can simply be described by the constraint that each pixel can only take two values (0 and 1). In real life, data are much more complex, possibly continuous; such data can also be handled in our framework, though experimenting with realistic simulators is the topic of future work. The neural network is fully-connected with two hidden layers. The number of neurons in the first and second hidden layers are respectively one half and one quarter of the flattened size of the input. All weights were initialized using Glorot optimization, with a gain of 1. The network was trained with Adam optimizer for 2000 epochs, with batch size of 100, using the binary cross entropy loss. The danger zone is defined as the bottom half part of the image. Any image with at least one white pixel in this zone should then yield a "change direction" directive.

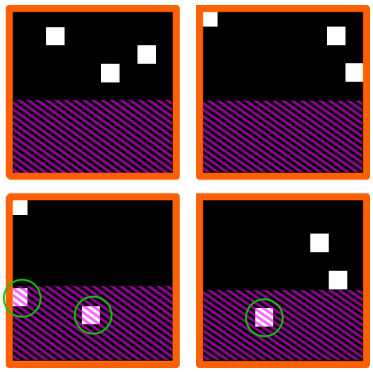

Figure 4: Example of inputs for the toy problem. White pixels represent obstacles. If they are in the top half of the image, no alert should fire (first two exemples), while an alert must fire if in the (dashed) bottom half of the image (last two examples). 9x9 picture is depicted here for clarity.

Here, constraints on inputs are encoded as statements on the SMT-LIB variables. A fragment of property to check is presented on Figure 5. On such a simple problem, the decomposition perception/reasoning is not needed, since there exists a formal characterization of what an obstacle is.

Experiments results will come on later versions of the paper, and will additionaly be available at https://www.lri.fr/ gcharpia/camus/.

6 Discussion and perspectives

We introduced CAMUS, a formalism describing how to formally express safety properties on functions taking simulated data as input. We also proposed ONNX2SMT, a tool soon to be open sourced, that leverages two standards used by the communities of formal methods and machine learning, to automatically write machine learning algorithms as logical formulae. We demonstrated the joint use of ONNX2SMT and CAMUS on a synthetic example mimicking a self-driving car perceptive unit, as a proof of concept of our framework. This toy example is of course still simplistic and much work on scalability is needed before real self-driving car simulators can be incorporated into formal proofs.

Among future work, ONNX2SMT will be released and gain support for more deep learning operations. While we provide a toolkit to translate neural networks directly in our framework, a way to easily represent a simulator is yet to include. It is not an easy task, since the simulator must be describable with sufficient granularity to allow the solver to use the simulator internal working to simplify the verification problem. A scene description language with a modelling language for simulators is a possible answer to these issues. Further theoretical characterization of the simulator procedure and its link with the perceptive unit will be undertaken, for instance to encompass stochastic processes. Besides, on more complex simulators, programs and examples, the problem to verify will remain computationally difficult. Various techniques to enhance solvers performances will be developed and integrated in CAMUS, taking advantage of the domain knowledge provided by the simulators parameters. Finally, our current framework checks properties for all possible inputs, including anomalous ones

```
;;;; Automatically generated part
;; Inputs declaration
(declare-fun |actual_input_0_0_0_8| () Real)
       [ommitted for brevity]
;; Weights declaration
(declare-fun |l_1.weight_31_4| () Real)
(assert (= |l_1.weight_31_4| (/ -5585077 33554432)))
       [ommitted for brevity]
;; An example of encoded calculation
(assert (= |8_0_0_0_39| (* |actual_input_0_0_0_8| (+ |7_80_39| (* |
    actual_input_0_0_0_7| (+ |7_79_39| (* |actual_input_0_0_0_6| (+
    |7_78_39| (* |actual_input_0_0_0_5|
       [ommitted for brevity]
;; Outputs declaration
(assert (= |actual_output_0_0_0_1| ( + |16_0_0_0_1| |l_3.bias_1| )))
       [ommitted for brevity]

;;;; Handmade annotations
;; Simulator description
;; Input space constraints: inputs between 0 and 1
(assert (or (= actual_input_0_0_0_8 0) (= actual_input_0_0_0_8 1.)))
       [ommitted for brevity]
;; Property to check
;; At least one input in the danger zone is white
(assert
  (or
    (or (= actual_input_0_0_0_5 1.)
        (= actual_input_0_0_0_6 1.))
    (or (= actual_input_0_0_0_7 1.)
        (= actual_input_0_0_0_8 1.))
        [ommitted for brevity]
;; Formulate constraint on outputs:
;; Output is always higher than a
;; confidence value
;; Negation: output can fire lower than a
;; confidence value
(assert (< actual_output_0_0_0_1 actual_output_0_0_0_0))
```

Figure 5: A SMTLIB2 file describing our problem. First part is a full description of the network, automatically produced by ONNX2SMT. Handmade annotations describe the property to check; the goal is to find a counterexample.

such as adversarial attacks. A possible extension would be to identify "safe" subspaces instead, where perception is guaranteed to be perfect, and "unsafe" subspaces where failures may happen.

References

[1] Alexandre Araujo, Rafael Pinot, Benjamin Negrevergne, Laurent Meunier, Yann Chevaleyre, Florian Yger, and Jamal Atif. Robust Neural Networks

using Randomized Adversarial Training. *arXiv:1903.10219 [cs, stat]*, March 2019. arXiv: 1903.10219.

[2] Junjie Bai, Fang Lu, Ke Zhang, et al. Onnx: Open neural network exchange. `https://github.com/onnx/onnx`, 2019.

[3] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi'c, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, July 2011. Snowbird, Utah.

[4] Clark Barrett, Leonardo De Moura, and Aaron Stump. Smt-comp: Satisfiability modulo theories competition. In *International Conference on Computer Aided Verification*, pages 20–23. Springer, 2005.

[5] Clark Barrett, Pascal Fontaine, and Aaron Stump. The SMT-LIB Standard. page 104.

[6] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018.

[7] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[8] Akhilan Boopathy, Tsui-Wei Weng, Pin-Yu Chen, Sijia Liu, and Luca Daniel. CNN-Cert: An Efficient Framework for Certifying Robustness of Convolutional Neural Networks. *arXiv:1811.12395 [cs, stat]*, November 2018. arXiv: 1811.12395.

[9] Rudy Bunel, Ilker Turkaslan, Philip H. S. Torr, Pushmeet Kohli, and M. Pawan Kumar. A Unified View of Piecewise Linear Neural Network Verification. *arXiv:1711.00455 [cs]*, November 2017. arXiv: 1711.00455.

[10] Nicholas Carlini and David Wagner. Towards Evaluating the Robustness of Neural Networks. *arXiv:1608.04644 [cs]*, August 2016. arXiv: 1608.04644.

[11] Florian Chabot, Mohamed Chaouch, Jaonary Rabarisoa, Céline Teulière, and Thierry Chateau. Deep manta: A coarse-to-fine many-task network for joint 2d and 3d vehicle analysis from monocular image. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2040–2049, 2017.

[12] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[13] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[14] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An open urban driving simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*, pages 1–16, 2017.

[15] Tommaso Dreossi, Daniel J Fremont, Shromona Ghosh, Edward Kim, Hadi Ravanbakhsh, Marcell Vazquez-Chanlatte, and Sanjit A Seshia. Verifai: A toolkit for the formal design and analysis of artificial intelligence-based systems. In *International Conference on Computer Aided Verification*, pages 432–442. Springer, 2019.

[16] Bruno Dutertre and Leonardo De Moura. The yices smt solver. Technical report, 2006.

[17] Nic Ford, Justin Gilmer, Nicolas Carlini, and Dogus Cubuk. Adversarial Examples Are a Natural Consequence of Test Error in Noise. *arXiv:1901.10513 [cs, stat]*, January 2019. arXiv: 1901.10513.

[18] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and Harnessing Adversarial Examples. *arXiv:1412.6572 [cs, stat]*, December 2014. arXiv: 1412.6572.

[19] Andrew J. Hawkins. Tesla didnt fix an autopilot problem for three years, and now another person is dead. *The Verge*, May 2019.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *arXiv:1512.03385 [cs]*, December 2015. arXiv: 1512.03385.

[21] Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Logan Engstrom, Brandon Tran, and Aleksander Madry. Adversarial Examples Are Not Bugs, They Are Features. *arXiv:1905.02175 [cs, stat]*, May 2019. arXiv: 1905.02175.

[22] Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4401–4410, 2019.

[23] Guy Katz, Clark Barrett, David Dill, Kyle Julian, and Mykel Kochenderfer. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. *arXiv:1702.01135 [cs]*, February 2017. arXiv: 1702.01135.

[24] Guy Katz, Derek A. Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zelji, 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*, volume 11561, pages 443–452. Springer International Publishing, Cham, 2019.

[25] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial examples in the physical world. *arXiv:1607.02533 [cs, stat]*, July 2016. arXiv: 1607.02533.

[26] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards Deep Learning Models Resistant to Adversarial Attacks. *arXiv:1706.06083 [cs, stat]*, June 2017. arXiv: 1706.06083.

[27] Guido Manfredi and Yannick Jestin. An introduction to acas xu and the challenges ahead. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, pages 1–9. IEEE, 2016.

[28] Bruno Marre, François Bobot, and Zakaria Chihani. Real behavior of floating point numbers. 2017.

[29] Matthew Mirman, Timon Gehr, and Martin Vechev. Differentiable Abstract Interpretation for Provably Robust Neural Networks. page 9.

[30] Nicolas Papernot, Patrick McDaniel, and Ian Goodfellow. Transferability in Machine Learning: from Phenomena to Black-Box Attacks using Adversarial Samples. *arXiv:1605.07277 [cs]*, May 2016. arXiv: 1605.07277.

[31] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017.

[32] Luca Pulina and Armando Tacchella. An Abstraction-Refinement Approach to Verification of Artificial Neural Networks. In *CAV*, 2010.

[33] Mirco Ravanelli, Titouan Parcollet, and Yoshua Bengio. The PyTorch-Kaldi Speech Recognition Toolkit. *arXiv:1811.07453 [cs, eess]*, November 2018. arXiv: 1811.07453.

[34] Carl-Johann Simon-Gabriel, Yann Ollivier, Lon Bottou, Bernhard Schlkopf, and David Lopez-Paz. First-order Adversarial Vulnerability of Neural Networks and Input Dimension. *arXiv:1802.01421 [cs, stat]*, June 2019. arXiv: 1802.01421.

[35] Gagandeep Singh, Timon Gehr, Markus Pschel, and Martin Vechev. An Abstract Domain for Certifying Neural Networks. 3:30.

[36] Gagandeep Singh, Timon Gehr, Markus Pschel, and Martin Vechev. Robustness Certification with Refinement. September 2018.

[37] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv:1312.6199 [cs]*, December 2013. arXiv: 1312.6199.

[38] Vincent Tjeng, Kai Xiao, and Russ Tedrake. Evaluating Robustness of Neural Networks with Mixed Integer Programming. *arXiv:1711.07356 [cs]*, November 2017. arXiv: 1711.07356.

[39] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Formal Security Analysis of Neural Networks using Symbolic Intervals. *arXiv:1804.10829 [cs]*, April 2018. arXiv: 1804.10829.

[40] Eric Wong and J. Zico Kolter. Provable defenses against adversarial examples via the convex outer adversarial polytope. November 2017.

[41] Zhewei Yao, Amir Gholami, Peng Xu, Kurt Keutzer, and Michael Mahoney. Trust Region Based Adversarial Attack on Neural Networks. *arXiv:1812.06371 [cs, stat]*, December 2018. arXiv: 1812.06371.