Verification of Neural Network through MILP and Constraint Programming

Karim Pedemonte

Department of Computer Science, Bioengineering, Robotics and System Engineering (DIBRIS) - Università di Genova

Heuristics and Diagnostics for Complex Systems (Heudiasyc) Lab - Université de Technologie de Compiègne

Supervisors

Stefano Demarchi
Antoine Jouglet
Dritan Nace
Armando Tacchella

Computer Engineering: Artificial Intelligence and Human-Centered Computing (UniGe)
and
Machine Learning and Optimization of Complex Systems (AOS) (UTC)

March 27th, 2024

Acknowledgements

I want to express my gratitude also to my supervisors Prof. Armando Tacchella, Post-Doc Researcher Stefano Demarchi and Prof. Antoine Jouglet, for their guidance and their advice during this research, which proved to be invaluable. A special thanks to Prof. Dritan Nace, responsible for the ISC Master and EMECIS Head of Master, for his support during this entire academic year and his supervision during the internship.

Abstract

The current document describes the research conducted during the internship at the Heudiasyc laboratory at the University of Technology of Compiègne (UTC) from the 27th of February 2023 to the 31th of July 2023, and then at Università di Genova until today. During the internship the problem of neural network verification through Mixed Integer Linear Programming and Constraint Programming and the topic of neural network falsification through sampling have been explored. The report is organized as follows. At first, the research context of the problem is detailed. Then, a formulation is proposed for each method. Finally, some results are gathered and discussed.

Contents

Chapter 1

Scientific Background

Summary

In this chapter the problem of the verification of neural networks is introduced. This chapter aims at explaining the reasons behind the research, providing a background on the current state of the art neural network verification tools.

1.1 Introduction

In the last few years, neural networks have found a wide range of applications across various industries and domains, also due to advancements in hardware, software frameworks, and the availability of large datasets, and, as a result they have become one of the most popular machine learning models. However, due to their "black box" nature, it is not easy to understand what they do exactly and it has been discovered that sometimes they are very sensitive to slight changes in the input data. This has serious implications on the use of neural networks, as they are now used in safety critical domains, such as autonomous driving systems.

1.1.1 Neural Network Verification

Neural network verification is a process aimed at ensuring the correctness and reliability of neural network models. It uses mathematical and formal methods to analyze the behavior of a neural network and verify that it adheres to certain safety and correctness properties, which usually involve checking for the robustness of the considered model to changes in the input. The goal of this process is to provide guarantees that the model will behave as expected under certain defined conditions and inputs. To this end, bounds on the input and output are defined and the verification process results in either the satisfiability or unsatisfiability of the conditions, allowing to know whether a particular region of the input space is safe or unsafe. Usually, conditions on the output are given in such a way that they define the unsafe region, while conditions on the input are bounds to an area whose safety we want to deternime. Conditions on the input are called pre-conditions, while conditions on the output are called post-conditions.

However, it is a challenging task due to the complexity of neural network architectures, the high number of parameters, and the non-linear behavior introduced by activation functions. As a result, this field is still an active area of research, and the methods and tools for neural network verification are always trying to improve its efficiency.

1.1.2 Background and related works

The problem has been tackled using several methods, such as SMT solving [Katz et al., 2017], approximation methods [Weng et al., 2018], abstract interpretation [Singh et al., 2019] and mixed integer linear programming [Tjeng et al., 2019].

Each of these methods has its limitations, either in terms of scalability, that is to say, its efficiency declines rapidly as the network dimension increases, or in terms of precision. Each method is affected differently by these limitations, as reported in Singh et al. [2019].

In general, neural network verification methods can be divided into complete and incomplete methods [Tjeng et al., 2019]. The difference is that incomplete methods employ several techniques which enable them to be more efficient, at the cost of not being able to verify with certainty that all points in the input region are correctly classified. Such techniques include, for example, discretization of the search space and layer-by-layer approximations of the bounds.

Mixed Integer Linear Programming is one of the most explored complete methods among all, and consequently, it has been treated quite in detail in the literature. The general idea of this method is to encode the network into a mixed integer linear problem together with the constraints on the input and output. Challenges of this method include the encoding of the non linear activation functions which are present in every neuron of the network. However, the ReLU function, the most popular activation function, defined as $ReLU(x) = \max(x, 0)$, is piecewise linear, thus admitting a MILP encoding, although it requires the introduction of binary variables and a big M, as shown in Tjeng et al. [2019] for example.

Every state of the art network verification tool using a MILP encoding has always tried to bring further optimization in the efficiency of the method, in various forms. [Tjeng et al., 2019] define a presolve step, consisting in finding the bounds of the input of ReLU functions in the network. This step is carried out through two methods, one involving interval arithmetic, the other linear programming. [Botoeva et al., 2020] developed a dependency analysis procedure, which is run in conjunction with the MILP solver, and is aimed at introducing additional constraints between outputs of neurons. Neurons are classified as active, inactive or neither, depending on their output: a neuron with input x is said to be active if $ReLU(x) = x$ always and is said to be inactive instead if $ReLU(x) = 0$. A dependency between two neurons means that if one is in a certain state, active or inactive, then the other is always found in another. Dependencies can be found between layers of the network (inter-layer dependencies) or in the same layer (intra-layer dependencies).

Linear programming has also been used as part of other methods which do not encode the entire problem with a MILP formulation, but only steps of the verification algorithm. An example of this is found in Henriksen and Lomuscio [2021], where the authors present an algorithm which is based on relaxing the ReLU function and Linear Programming is used in the search phase of their algorithm.

On the other hand, constraint programming has never been tried as a method to verify neural networks. It provides some advantages when compared to MILP, such as the flexibility in the choice of the activation function in the network, as it is not bound by the choice of a piecewise linear function.

Chapter 2

Model Encoding

Summary

In this chapter some formal definitions of the neural network verification problem are given and two models are proposed.

2.1 Definitions

2.1.1 Neural Networks

In the following report, only feedforward fully connected neural networks are considered, that is to say the output of a layer only affects its successors and not the input of that same layer, and they are only composed by fully connected and functional layers. The two types of layers are defined as follows:

- Fully connected layer: it is a linear mapping $f(\cdot; W, b) : \mathbb{R}^m \to \mathbb{R}^n$ such that, if $x \in \mathbb{R}^m$ is its input, $f(x) = Wx + b$, where $W \in \mathbb{R}^{n \times m}$ is called the *weight matrix* of the layer, and $b \in \mathbb{R}^n$ is the *bias vector* of the layer.

- Functional layer: it is a mapping $f : \mathbb{R}^n \to \mathbb{R}^n$ such that, if $x \in \mathbb{R}^n$ is its input, $f(x) = (\sigma(x_1), \dots, \sigma(x_n))$, where $\sigma : \mathbb{R} \to \mathbb{R}$ is a nonlinear function, called *activation function.*

Two of the most widespread activation functions are the *logistic function*, defined as $\sigma(r) = \frac{1}{1+e^{-r}}$ and the *ReLU* (Rectified Linear Unit) function, defined as $ReLU(r) = \max(r, 0)$. The ReLU function is the most popular, but the logistic function is also sometimes used.

Thus fully connected neural networks can be defined as follows: they are a mapping $f : \mathbb{R}^m \to \mathbb{R}^n$, m the size of the input and n that of the output, such that, if $x \in \mathbb{R}^m$ is its input, $f(x) = f_p(f_{p-1}(\dots f_1(x)\dots))$, where $p \in \mathbb{N}$ is the number of layers of the network, and every $f_i, i \in [1, p]$ is one of the two mappings defined above.

Two tasks for which neural networks are trained are *classification* and *regression.* Given a network $f : \mathbb{R}^m \to \mathbb{R}^n$, classification of an input x is assigning to it one of n labels. An input is classified in class $l \in [1, n]$ if $f_l(x) > f_i(x),\ i \in [1, n]\backslash l$. The task of regression instead concerns the approximation of a functional mapping from $\mathbb{R}^m$ to $\mathbb{R}^n$.

2.1.2 Verification

Given a neural network $f : \mathbb{R}^m \to \mathbb{R}^n$ the task of *verification* is that of algorithmically verifying whether, given some conditions on the input, other conditions on the output are satisfied. The input domain of the network f is assumed to be bounded, and consequently the output is too [Demarchi and Guidotti, 2022].

Our verification problem is thus defined as follows: given $p \in \mathbb{N}$ bounded sets $X_1, \dots, X_p$ and $s \in \mathbb{N}$ bounded sets $Y_1, \dots, Y_p$ we want to check whether the following

$$\forall x \in \bigcup_{i=1}^{p} X_i . f(x) \in \bigcup_{i=1}^{s} Y_i$$

is true. All models detailed further will consider $p = 1$.

2.2 Mixed Integer Linear Programming

In the following sections, a method to express the neural network verification problem as solving a Mixed Integer Linear Programming (MILP) problem will be detailed, based on Tjeng et al. [2019]. In order to provide a fairer comparison, the MILP model is not optimized in any way.

2.2.1 First formulation

Given a neural network $f : \mathbb{R}^m \to \mathbb{R}^n$, the requirement for modeling the verification problem with MILP is that f must be the composition of piecewise linear functions. That is the reason this formulation is limited to networks using the ReLU activation function. As stated before, we consider only a feedforward fully connected neural network, so every layer in the network is either a ReLU layer, which is piecewise linear, or a fully connected layer which is a linear transformation.

The model proposed by Tjeng et al. [2019] is tailored to networks trained for classification, so this first base model proposition is too. However, it will become apparent that this is not sufficiently general.

We define X as the region in the input domain corresponding to all allowable perturbations of a certain input x^*. X must be expressed as a union of polyhedra for the problem to be expressed through MILP.

$\mu \in [1, n]$ represents all the possible labels of an input.

$\lambda(x^*)$ is the true label of x^* and takes integer values in the interval $[1, n]$.

The neural network is robust to perturbations on x^* if the predicted probability, which is represented by the output of the network for a certain input, of the true label $\lambda(x^*)$ exceeds that of every other label for all perturbations. Equivalently, the network is robust to perturbations on x^* if the following equation is unfeasible

$$x \in X \wedge \left(f_{\lambda(x^*)}(x) < \max_{\mu \in [1,n] \setminus \{\lambda(x^*)\}} f_\mu(x) \right)$$

A distance metric $d(x, x^*)$ measuring perceptual similarity between inputs could also be defined and introduced as a cost function. However, this is reasonable only for networks trained for classification, since for those trained for regression we are not interested in defining a point and its allowable perturbation region, rather we define directly a possible input region; moreover, it is not part of our definition of the verification problem.

Thus the problem formulation is the following feasibility problem:

$$\begin{aligned} & \min 0 \\ \textit{subject to} \quad & f_{\lambda(x^*)}(x) \leq \max_{\mu \in [1,n] \setminus \{\lambda(x^*)\}} f_\mu(x) \\ & x \in X \end{aligned}$$

In other words, what is found is whether an input in the allowable perturbations is classified incorrectly.
The current general formulation is not linear, but everything can be linearized by introducing new variables and/or constraints.
The constraint enforcing the incorrectness of the label is not necessarily written with a max function in the formulation: it can be written as a disjunction of inequalities, as it still means that an input is classified incorrectly.

$$\bigvee_{\mu \in [1,n] \setminus \{\lambda(x^*)\}} f_{\lambda(x^*)}(x) \leq f_\mu(x)$$

They can be linearized by introducing $n-1$ binary variables and n big M constraints as shown here

$$f_{\lambda(x^*)}(x) \leq f_\mu(x) + M(1-\delta_\mu), \quad \forall \mu \in [1,n] \setminus \{\lambda(x^*)\}$$

$$\sum_{\mu \in [1,n] \setminus \{\lambda(x^*)\}} \delta_\mu \geq 1$$

$$\delta_\mu \in \{0,1\}$$

The function f will be composed by ReLUs, which can be linearized by introducing a binary variable a, which is equal to 0 if the ReLU will be inactive and 1 otherwise, and the following set of constraints, where y is the value representing the output of the ReLU and x its input:

$$y \leq x + M(1-a)$$

$$y \geq x$$

$$y \leq Ma$$

$$y \geq 0$$

It can be seen that if $a = 0$, then $y = 0$, and if $a = 1$, then $y = x$.

2.2.2 Generalized formulation

As stated before, the previous formulation only works if the network is trained for a classification task. Sometimes networks are trained for other purposes: for example, controlling a system. In such cases, the architecture of a network does not change, but the interpretation of the output layer does: it is not the indication of confidence in a certain class anymore, it is now something else, for example, the suggestion of an action to take among others to refer to the previous control example. As a consequence, we may not be interested only in what is the maximum value of the output, but we may be interested in verifying whether the maximum is not a certain output or a set of outputs, or even whether the outputs respect some other property, on a problem per problem basis. Thus, the previously proposed optimisation problem must be generalized and becomes a feasibility problem, since we don't intend to find any distance. The new problem will still work for classification networks, provided the right property is given. The property on the output is usually specified as an unsafe property, which means that if the problem admits a solution, the property is not verified.

The mathematical formulation of the problem is the following:

$$\begin{aligned} & \min 0 \\ \textit{subject to} \quad & \bigvee_{i \in [1,r]} (f(x) \in Y_i) \\ & x \in X \end{aligned}$$

where $x \in \mathbb{R}^m$ is the input of the network and $f(x) \in \mathbb{R}^n$ the output, X is a polytope and every Y_i is a convex subset of $\mathbb{R}^n$ defined by the intersection of closed half-spaces and r is the number of these subsets.

X is the region of the input for which we want to verify a property on the output. In general, the property on the output is expressed as a disjunction of various constraints, that can be linearized by introducing a binary variable for every Y_i, which we will call δ_i. We also define h_i as the number of half-spaces defining Y_i.

$$\sum_{j=1}^{n} c_{kj} f_j(x) \leq b_k + M(1 - \delta_i), \quad \forall k \in [1, h_i], \forall i \in [1, n]$$

$$\sum_{i=1}^{r} \delta_i \geq 1$$

$$\delta_i \in \{0, 1\}$$

The nonlinearity introduced by the ReLU function is tackled as described before.

2.3 Constraint Programming

The following section is aimed at giving an understanding of constraint programming and at explaining its application to the problem of neural network verification.

2.3.1 Basic concepts

From [Apt, 2003] and [Bessière, 2006] here are some basic definitions.
We consider a sequence of variables $X := x_1, \ldots, x_k \quad k > 0$ and their domains $D_1, \ldots, D_k$. Domains can be sets of integer or real numbers, boolean values or even symbolic values.

A *constraint* on X is defined as a subset of the Cartesian product of the domains of the variables in the sequence $x_1, \ldots, x_k$.

A *domain expression* is a construct in the form $x \in D$, where x is a variable and D its domain.

A *constraint satisfaction problem (CSP)* is a sequence of variables X together with a set of constraints. We write it as $\langle \mathcal{C}; \mathcal{DE} \rangle$, where $\mathcal{C}$ is the set of constraints and $\mathcal{DE}$ is the set of all the domain expressions of the variables.

A constraint C on the variables $X(C) = (x_{i_1}, \ldots, x_{i_k})$ is said to be *satisfied* by a tuple $(d_1, \ldots, d_n) \in D_1 \times \ldots \times D_n$, given a CSP $\langle \mathcal{C}; \mathcal{DE} \rangle$ where $\mathcal{DE} = \{x_1 \in D_1, \ldots, x_n \in D_n\}$, when $(d_{i_1}, \ldots, d_{i_n}) \in C$.

A constraint satisfaction problem $\langle \mathcal{C}; \mathcal{DE} \rangle$ is said to *have a solution* if a tuple $(d_1, \ldots, d_n) \in D_1 \times \ldots \times D_n$ such that it satisfies all constraints in $\mathcal{C}$ exists. The tuple is the *solution* to the problem. A CSP is called *consistent* if it has a solution, otherwise it is called *inconsistent*.

Given a sequence of variables $X = x_1, \ldots, x_n$ with their corresponding domains $D_1, \ldots, D_n$ and a subsequence $Y = (x_{i_1}, \ldots, x_{i_k})$ the *projection* of a tuple $d = (d_1, \ldots, d_n) \in D_1 \times \ldots \times D_n$ is the sequence $(d_{i_1}, \ldots, d_{i_n})$, and is denoted by $d[Y]$.

Given a CSP $\langle \mathcal{C}; \mathcal{DE} \rangle$ with variables $x_1, \ldots, x_n$, an *instantiation* I on a subset $Y = \{x_1, \ldots, x_k\}$ of the variables is an assignment of the values $v_1, \ldots, v_k$ to them. An instantiation I on Y is valid if $I[x_i] \in D_i, \forall x_i \in Y$. An instantiation I on Y is *locally consistent* if it is valid and $I[X(C)]$ satisfies all $C \in \mathcal{C} \quad s.t. \quad X(C) \subseteq Y$. A locally consistent instantiation on all the variables of a CSP is one of its solutions.

2.3.2 Constraint Propagation

In order to solve a CSP, a constraint solver employs a search algorithm, for example a backtracking search, to assign a value to all the variables, until an instantiation is found; they may do so by trying to extend a partial instantiation to a global one. However, exploring the entire search space is a very expensive task, so the concept of constraint propagation is introduced.

Constraint propagation is a fundamental technique used to narrow down the domains of variables by exploiting the relationships and restrictions defined by constraints. It's a way to deduce new information about variable assignments from the constraints, thus reducing the search space and finding solutions more efficiently.

Constraint propagation algorithms generally aim at achieving local consistency. A definition of local consistency is reported verbatim from Bessière [2006]: "a local consistency is a property that characterizes some necessary conditions on values (or instantiations) to belong to solutions. A local consistency property (denoted by P) is defined regardless of the domains or constraints that will be present in the network. A network is P-consistent if and only if it satisfies the property P". One such local consistency is arc consistency. As defined in Apt [2003], let us consider a binary constraint C on the variables x and y, with domains respectively D_x and D_y. $C \subseteq D_x \times D_y$ is arc consistent if:

- $\forall a \in D_x, \exists b \in D_y \quad s.t. \quad (a, b) \in C.$
- $\forall b \in D_y, \exists a \in D_x \quad s.t. \quad (a, b) \in C.$

A CSP is arc consistent if all its binary constraint are arc consistent. If a CSP is reduced to an equivalent arc consistent one, that is to say an arc consitent CSP with the same set of solutions, the search space can be greatly reduced, and for certain classes of problems, the CSP is also consistent.

Now we introduce a framework [Apt, 2003] which will allow us to define domain reduction rules in order to achieve arc consistency. We consider two CSPs, $\phi = \langle \mathcal{C}; \mathcal{DE} \rangle$ and $\psi = \langle \mathcal{C}'; \mathcal{DE}' \rangle$, where $\mathcal{DE} = x_1 \in D_1, \ldots, x_n \in D_n$ and $\mathcal{DE}' = x'_1 \in D'_1, \ldots, x'_n \in D'_n$. We introduce proof rules representing the transformation of a CSP in the form

$$\frac{\phi}{\psi}$$

. The aim is reducing a CSP to another equivalent CSP. A domain reduction rule is a proof rule $\frac{\phi}{\psi}$ where $D'_i \subseteq D_i,\ \forall i \in [1, n]$. A rule R applied on ϕ which results in ψ is said to be a *relevant application* of R to ϕ if ψ is different from ϕ. If a rule R cannot be applied on ϕ or a relevant application of R on ϕ does not exist, ϕ is said to be *closed under the applications of* R.

Arc consistency can be expressed in terms of domain reduction rules [Apt, 2003]. Given the following rules and a constraint C on the variables x and y:

$$R1$$

$$\frac{\langle C; x \in D_x, y \in D_y \rangle}{\langle C; x \in D'_x, y \in D_y \rangle}$$

$$R2$$

$$\frac{\langle C; x \in D_x, y \in D_y \rangle}{\langle C; x \in D_x, y \in D'_y \rangle}$$

where $D'_x = \{a \in D_x,\ \exists b \in D_y\ s.t.\ (a, b) \in C\}$ and $D'_y = \{b \in D_y,\ \exists a \in D_x\ s.t.\ (a, b) \in C\}$. A CSP is arc consistent if it closed under the application of $R1$ and $R2$. Since these rules preserve equivalence, by repeatedly applying these rules to any problem, any problem can be transformed into an equivalent arc consistent one [Apt, 2003].

2.3.3 Neural Network Verification as a CSP

In this section we define two domain reduction rules for the ReLU function and formulate the problem of neural network verification as a CSP.

2.3.3.1 ReLU as a domain reduction rule

Let us consider the CSP $\langle ReLU(x, y); x \in [a, b], y \in [c, d] \rangle$, where $a, b, c, d \in \mathbb{R}$, $c \geq 0$ and $y = ReLU(x)$. The ReLU domain reduction rules are:

$$\frac{\langle ReLU(x, y); x \in [a, b], y \in [c, d] \rangle}{\langle ReLU(x, y); x \in [a, b], y \in [\max(a, c, 0), \min(b, d)] \rangle}$$

$$\frac{\langle ReLU(x, y); x \in [a, b], y \in [c, d] \rangle}{\langle ReLU(x, y); x \in [e, \min(b, d)], y \in [c, d] \rangle}, \quad e = \begin{cases} c, & \text{if } c > 0 \quad \text{and} \quad c > a \\ a, & \text{otherwise} \end{cases}$$

2.3.3.2 CSP formulation

Given a feedforward neural network $f : \mathbb{R}^m \to \mathbb{R}^n$ and a property to verify, which means we have bounds on the input x and the output y that may be written as $x \in X \subseteq \mathbb{R}^m$ and $y \in Y \subseteq \mathbb{R}^n$, we want to model the verification problem as a CSP. In order to be able to compare the two models, X and Y in practice will be assumed to be unions of polytopes, and the activation function of the layers of the network will be the ReLU function.
The variables of the problem are defined as follows:

- the m components of the input $x_i \in \mathbb{R},\ i \in [1, m]$.
- the n components of the output $y_i \in \mathbb{R},\ i \in [1, n]$.
- each component of a hidden layer which is the result of the vector multiplication of the fully connected layers $z_{ij} \in \mathbb{R},\ i \in [1, l],\ j \in [1, c_i]$, where l is

the number of fully connected layers in the network and c_i is the number of neurons in the layer i.

- each output of the activation function $t_{ij} \in [0, \infty],\ i \in [1, r],\ j \in [1, c_i]$, where r is the number of ReLU layers in the network and c_i is the number of neurons in the layer i.

A remark is that r does not always equal l, as often neural networks do not have a functional layer as the output layer; in such case $r = l - 1$.

The constraints of the problem are:

- the C_x and C_y representing the linear bounds on the input and output, respectively.
- the C_z tying together the output of the vector multiplication and its inputs; given an output z_{ij} and inputs $z_{kh},\ h \in [1, c_k]$, with $k = i - 1$; they are in the form $z_{ij} = (W_i z_k + b_i)_j$, where W_i is the weight matrix of the layer i and b_i is its bias. For $k = 0$, $z_k = x$.
- the C_t which are the binary ReLU constraints, tying input and output of the activation function.
- the C_o constraints, which tie together the output of the network y and the output of the last layer, z_l or t_r depending on the network; they are in the form $y_j = z_l$ or $y_j = t_r$ with $j \in [1, n]$.

2.4 Falsification

Neural Network falsification is a complementary approach to verification that aims to find scenarios where a neural network model fails to meet its intended specifications or makes incorrect predictions. While verification aims to prove the correctness of a neural network under certain conditions, falsification seeks to identify situations where the model behaves unexpectedly or incorrectly. The process of neural network falsification involves systematically generating test cases or input data that push the boundaries of the model's capabilities, potentially exposing weaknesses, vulnerabilities, or inaccuracies in its behavior. These test cases are designed to trigger failure modes or edge cases that may not have been adequately addressed during the model's development or testing phases.

To this end, we propose a method to determine whether a neural network satisfies the conditions on the output of a given property by finding at least an input of the network satisfying both the conditions on the input and the output. Such input, if it exists, is called a counterexample.

2.4.1 Sampling algorithm

In this section we will propose an algorithm to find a counterexample based on a uniform sampling on the input region defined by the conditions on the input of the property which we want to verify.

First, we generate samples to obtain a uniform distribution over our sampling space, then, we need to check whether a counterexample has been found. In order to arrive at this objective, we need to calculate the output of each sample by applying the neural network function and see if it belongs to the unsafe region defined by the conditions on the output of the property.

The algorithm, based on the Hit and Run sampler described in Smith [1996], will now be described. Our objective is obtaining a uniform distribution over the set $S \subseteq \mathbb{R}^n$, which represents the pre-conditions on the n-dimensional input of the neural network f, and verifying whether any of the outputs of the samples is in the unsafe region U.

The algorithm is composed of the following steps:

1. Generate a starting point $x_0 \in S$.

2. Generate a random direction d in $D \subseteq \mathbb{R}^n$.

3. Generate a random point x_1 in the set $L = S \cap \{x | x = x_0 + \lambda d, \lambda \in \mathbb{R}\}$.

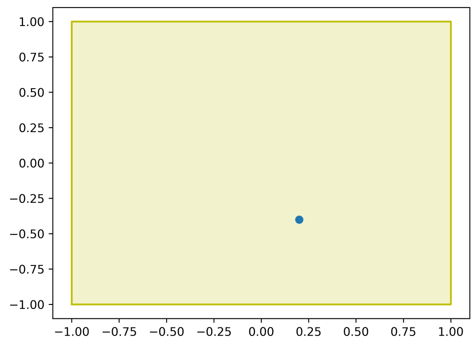
4. If the number of samples is the desired number, stop, otherwise repeat steps 2 and 3.

5. For each sample x_i, check whether $f(x_i) \in U$.

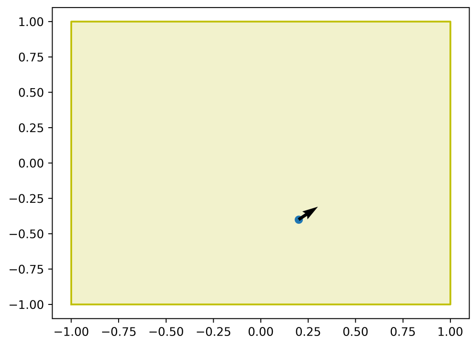
6. If $f(x_i) \in U$, then the property is unsafe and the algorithm stops.

If no output belongs to the unsafe region, we may not deduce that the property is safe, since we may simply lack a sample whose output belongs to it.

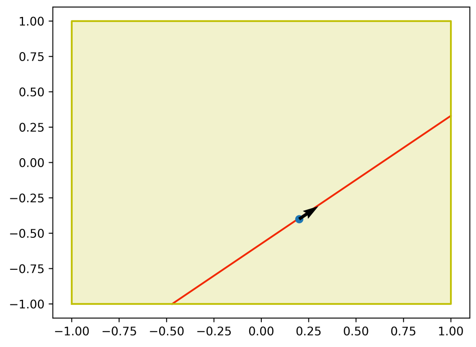
As shown in Smith [1996], this algorithms generates a distribution of samples which converges to a uniform distribution, which means that, with enough samples, it can be approximated to one. The implementation of the algorithm makes the assumption that the pre-conditions of the property can be represented by a hyperrectangle, which means that each input of the network is bounded by two real numbers. This however, is not a relevant restriction, as most properties are encoded in this way.



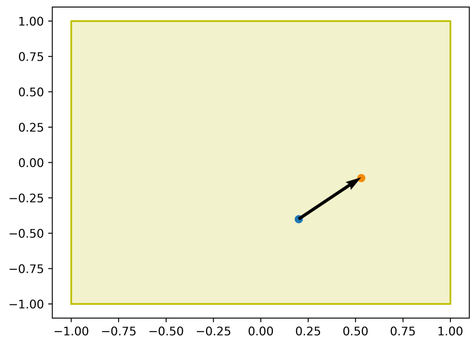
(a) Choice of the starting point



(b) Choice of the random direction



(c) Generation of the L set



(d) Choice of λ and the next point

Figure 2.1: Visual representation of the steps of the algorithm.

```
Data: The network $f$, the number $N$ of $n$-dimensional samples to
      generate and two vectors $lower$ and $upper$ representing their
      bounds, and the set $U$ of unsafe outputs.
Result: $True$ if the property is falsified, $False$ otherwise.
$samples \leftarrow new\ Vector[N]$;
$sampleoutputs \leftarrow new\ Vector[N]$;
$startingpoint \leftarrow new\ Vector[n]$;
$startingpoint \leftarrow$ RandomUniformVector$(lower, upper, n)$;
$samples[0] \leftarrow startingpoint$;
$sampleoutputs[0] \leftarrow f(startingpoint)$;
for $i \leftarrow 1$ to $N-1$ do
    $v \leftarrow$ RandomUniformVector$(lower, upper, n)$;
    $v \leftarrow \frac{v}{\texttt{VectorNorm}(v)}$;
    $\lambda_{max} \leftarrow new\ Vector[n]$;
    $\lambda_{min} \leftarrow new\ Vector[n]$;
    for $i \leftarrow 0$ to $n-1$ do
        $\lambda_{max}[i] \leftarrow \frac{upper[i] - startingpoint[i]}{v[i]}$;
        $\lambda_{min}[i] \leftarrow \frac{lower[i] - startingpoint[i]}{v[i]}$;
    end
    $\lambda \leftarrow$ RandomUniformNumber$(\max(\lambda_{min}), \min(\lambda_{max}))$;
    $startingpoint \leftarrow startingpoint + \lambda v$;
    $samples[i] \leftarrow startingpoint$;
    $sampleoutputs[i] \leftarrow f(startingpoint)$;
end
for $i \leftarrow 0$ to $N-1$ do
    if BelongsToSet$(sampleoutputs[i], U)$ then
        return $True$;
    end
end
return $False$;
```

Algorithm 1: Pseudocode of the sampling algorithm

Chapter 3

3.1 Implementation

The models previously presented have been implemented in the pyNeVer Python library, an open source project for the learning and verification of neural networks and other machine learning models, by the researchers working at the Department of Informatics, Bioengineering, Robotics and Systems Engineering (DIBRIS) of the University of Genoa. The library handled the parsing of the neural networks, which are provided as input to the programs in the ONNX format, and of the properties to verify, provide in the VNN-LIB format. The MILP model, it has been implemented using the IBM CPLEX Optimizer Python API, while the constraint programming one has been implemented in C++ using the IBM ILOG CP Optimizer; in this last case, the parsing of the network is handled by pyNeVer, an then the C++ function is called in the Python code by using a binding library, pybind11.

3.1.1 MILP and CP comparison

To compare the two models, they were tested on 20 different neural network trained on a dataset concerning the arm of a humanoid robot named James. As described in Pulina and Tacchella [2012], James is a torso composed by a head, a

left arm and a hand, each with respectively 7, 7 and 8 degrees of freedom. James uses a neural network to estimate internal forces in the arm in order to estimate measure external forces, which require the subtraction of the internal forces from sensor readings. Internal forces in James's arm are estimated considering angular positions and velocities of two shoulder and two elbow joints; these are the 8 inputs of the network. The outputs are the corresponding values of internal forces and torques, 6 values.
The 20 network each have a different number of neurons (16, 32, 64 and 128) and hidden layers (from 1 ro 5). The tests were conducted on 36 local robustness properties.

MILP	1	2	3	4	5
16	10.53762	10.77175	11.03901	11.20042	11.94238
32	11.368274	11.29198	14.42265	85.48617	16.19434
64	12.10045	18.10646	25.42662	132.79223	172.07932
128	14.00589	43.85542	216.17484	235.42857	265.00034

Table 3.1: Average time to verify each network using MILP. Unit: 10^{-3} seconds.

CP	1	2	3	4	5
16	13.92459	13.57841	15.69175	16.21007	4831.13050
32	13.35287	15.82646	20.53093	23.34165	24.94311
64	15.98238	8465.96407	32.19246	15167.81926	23567.52109
128	14190.54722	54.35848	3426.19323	46697.54147	99522.97973

Table 3.2: Average time to verify each network using CP. Unit: 10^{-3} seconds.

In 3.1 we are able to observe that, when using MILP, as the networks get deeper, the time required to solve them increases substantially. This is apparent when comparing the 128x2 and the 64x4 network, which have the same number of binary variables. This is due to the bounds of the polytope constructed by the program, whose bounds get more complex as the network deepens.

The data in 3.2 are not averaged on 36 properties, but only on 18: this is because, in the case the property was not verified, meaning there was no solution to be found, the program timed out (after 20 minutes). This is not surprising, as the search phase of constraint programming solvers is slow.

Also, it can be noticed that the Constraint Programming solver scales much worse than the MILP with the dimension of the network.

The MILP problem, while obtaining better results in terms of time, is subject to a problem: the choice of a correct big M, especially in deeper networks. Due to the technical limitation of the finite representation of floating point numbers, the big M cannot be chosen to be too big as it will cause the multiplication of itself and a binary variable approximated to 0 to yield a result different from 0. However, due to the nature of neural networks, the choice is not obvious, since sometimes even a big M of 3 or 4 orders of magnitude greater than the data is not enough.

3.1.2 Sampling Results

Sample Points	Accuracy	Average Time
100	60.0%	0.01465
1000	60.0%	0.03581
5000	80.0%	0.16805
10000	80.0%	0.33724
100000	100.0%	3.33776

Table 3.3: Time and accuracy of the sampling algorithm on the Cartpole network. Unit: seconds.

The tests were conducted on three reinforcement learning networks, called Cartpole, LunarLander and Dubins Rejoin.

Tables 3.3, 3.4 and 3.5 report the results of running the sampling algorithm on multiple unsafe properties for each of the networks. They are the median results of the experiment run over 50 different random seeds.

Sample Points	Accuracy	Average Time
100	98.75%	0.00743
1000	98.75%	0.03741
5000	100.0%	0.17313
10000	100.0%	0.34424
100000	100.0%	3.45316

Table 3.4: Time and accuracy of the sampling algorithm on the Lunar Lander network. Unit: seconds.

Sample Points	Accuracy	Average Time
100	11.11%	0.01021
1000	33.33%	0.04469
5000	38.89%	0.20327
10000	38.89%	0.40673
100000	55.56%	4.19081

Table 3.5: Time and accuracy of the sampling algorithm on the DubinsRejoin network. Unit: seconds.

The columns represent, respectively, the number of sample inputs, the accuracy, defined as the percentage of correctly identified unsafe properties, and the time spent running the algorithm averaged on the number of properties. We can observe that as the number of samples increases, the accuracy increases accordingly, as there is less of a chance that a sample is not in the unsafe region. A visual representation of this phenomenon can be observed in Figures 3.1 and 3.2. This dependency of accuracy on the size of the sample space is also seen in the difference in accuracy between the networks, as we can see the Dubins Rejoin network performs the worst, since its sample space is much larger than that of the other two networks.

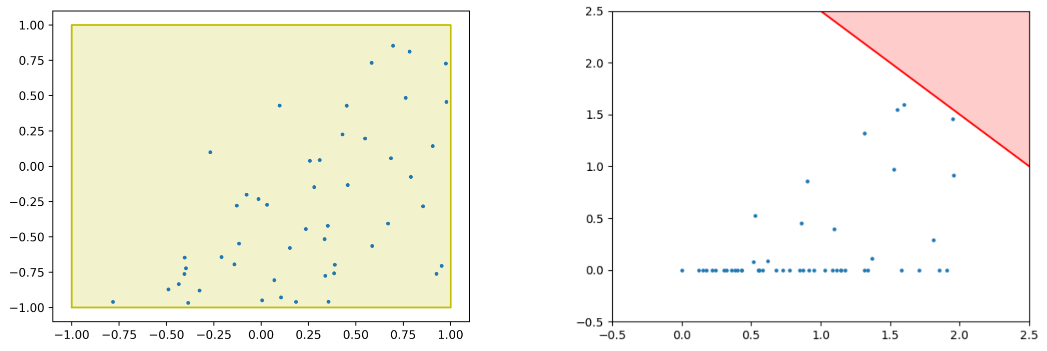


Figure 3.1: The yellow region represents the region defined by the pre-conditions, while the red one is the unsafe region defined by the post-conditions. If we were to run the algorithm with this number of samples, the property would not be falsified.

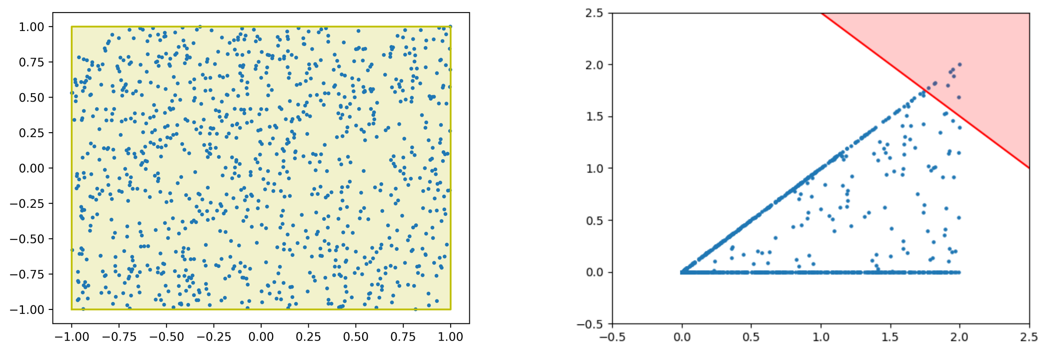


Figure 3.2: We can see that with enough samples the property will be declared unsafe.

3.2 Conclusion and future work

The experiments show that Constraint Programming, unfortunately, despite being able to tackle a non piece-wise linear activation function and not being subject to the choice of the big M, yields a worse performance than Mixed Integer Linear Programming as it is.

However, maybe a global constraint between the layers of the network may be introduced, in order to enhance the performance of the constraint propagation. In case the constraint propagation reaches fast enough time, it may be used as a starting point for another solving method instead of a generic search phase, as it will probably yield better results.

Concerning sampling, while on its own lacks the capacity to confirm the safety of a network, it may still be used to enhance other verification methods. We may, for instance, consider the closest points to a bound and perform a search from there or maybe consider the convex hull of said points and perform a verification procedure on this newfound region. Also, we may be able to optimize the sampling procedure by developing an algorithm which better approximates a uniform distribution of input samples by needing less samples, or also by only sampling on the boundary of the hyperrectangle defined by the pre-conditions of the property to be verified.

References

Apt, K. R. (2003). *Principles of Constraint Programming.* Cambridge University Press, Cambridge. 11, 13, 14

Bessière, C. (2006). Constraint propagation. *Foundations of Artificial Intelligence*, 2. 11, 13

Botoeva, E., Kouvaros, P., Kronqvist, J., Lomuscio, A., and Misener, R. (2020). Efficient verification of ReLU-based neural networks via dependency analysis. 34(04):3291–3299. 3

Demarchi, S. and Guidotti, D. (2022). Counter-example guided abstract refinement for verification of neural networks. 6

Henriksen, P. and Lomuscio, A. (2021). DEEPSPLIT: An efficient splitting method for neural network verification via indirect effect analysis. In Zhou, Z.-H., editor, *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, pages 2549–2555. International Joint Conferences on Artificial Intelligence Organization. Main Track. 4

Katz, G., Barrett, C., Dill, D., Julian, K., and Kochenderfer, M. (2017). Reluplex: An efficient smt solver for verifying deep neural networks. 2

Pulina, L. and Tacchella, A. (2012). Challenging SMT solvers to verify neural networks. *AI Communications*, 25(2):117–135. 21

Singh, G., Gehr, T., Püschel, M., and Vechev, M. (2019). An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages*, 3:1–30. 2, 3

Smith, R. L. (November 1996). The hit-and-run sampler: a globally reaching markov chain sampler for generating arbitrary multivariate distributions. In *Proceedings of the 1996 Winter Simulation Conference*, pages 260–264. 17, 18

Tjeng, V., Xiao, K., and Tedrake, R. (2019). Evaluating robustness of neural networks with mixed integer programming. *arXiv.org*. 2, 3, 7

Weng, T.-W., Zhang, H., Chen, H., Song, Z., Hsieh, C.-J., Boning, D., Dhillon, I., and Daniel, L. (2018). Towards fast computation of certified robustness for relu networks. 2