UNIVERSITY OF GENOVA

MASTER THESIS IN COMPUTER ENGINEERING

Verification of neural networks

by

Andrea Gimelli

Thesis submitted for the degree of *Master degree* (35° cycle)

May 2023

Armando Tacchella — Supervisor
Stefano Demarchi — Relator

Dibris

Department of Informatics, Bioengineering, Robotics and Systems Engineering

Acknowledgements

The work presented in this Thesis would have never been possible without the support and the teaching of my supervisor Armando Tacchella, I am forever grateful for his utmost dedication and infinite patience. I also wish to thank all my colleagues in the lab: Marco Menapace introduced me to the world of elevators and helped me learn a ton about software development in practice, along with Giuseppe Cicala; Dario Guidotti on the other hand helped me understand the new context of neural networks and his work clearly and patiently. Frosina, Francesco, Massimo: our exchanges, fruitful and trivial, relieved the stress and fatigue. A special thanks to Luca Pulina and Danilo Calzetta, who supported me while preparing the admission.

I extend my gratitude to all the researchers I met during my last year, when it was finally possible to travel again after the COVID-19 pandemic: Prof. Dritan Nace who welcomed me at Universitè de Technologie de Compiègne, Guillaume Joubert who proposed new ideas, Joelle Al Hage with the valuable discussions we held and Stephane, Philippe, Maxime, Remy, Antoine, Thibault, Lyes, Sana, Ling, Soundouss and everyone I shared coffee and ideas with.

Last, but not least, I thank my family and friends who were always there during this journey, supporting and helping me. Erica, my beloved, who endured as much as me the weight of this work and still is crazy enough to try and start to be a researcher too. The whole San Giorgio Fight Team, a family before a sport team. All of you are very near and dear to me.

Abstract

This thesis focuses on the application of Constraint Satisfaction and Optimization techniques in two Artificial Intelligence (AI) domains: automated design of elevator systems and verification of Neural Networks (NNs). The three main areas of interest for my work are (i) the languages for defining the constraints for the systems, (ii) the algorithms and encodings that enable solving the problems considered and (iii) the tools that implement such algorithms.

Given the expressivity of the domain description languages and the availability of effective tools, several problems in diverse application fields have been solved successfully using constraint satisfaction techniques. The two case studies herewith presented are no exception, even if they entail different challenges in the adoption of such techniques. Automated design of elevator systems not only requires encoding of feasibility (hard) constraints, but should also take into account design preferences, which can be expressed in terms of cost functions whose optimal or near-optimal value characterizes "good" design choices versus "poor" ones. Verification of NNs (and other machine-learned implements) requires solving large-scale constraint problems which may become the main bottlenecks in the overall verification procedure.

This thesis proposes some ideas for tackling such challenges, including encoding techniques for automated design problems and new algorithms for handling the optimization problems arising from verification of NNs. The proposed algorithms and techniques are evaluated experimentally by developing tools that are made available to the research community for further evaluation and improvement.

Chapter 1

Introduction

1.1 Context and motivation

1.1.1 Neural networks verification

Adoption and successful application of deep neural networks (DNNs) in various domains have made them one of the most popular machine-learned models to date — see, e.g., **?** on image classification, **?** on speech recognition, and **?** for the general principles and a catalog of success stories. Despite the impressive progress that the learning community has made with the adoption of DNNs, it is well known that their application in safety- or security-sensitive contexts is not yet hassle-free. From their well-known sensitivity to *adversarial perturbations* **??**, i.e., minimal changes to correctly classified input data that cause a network to respond in unexpected and incorrect ways, to other less-investigated, but possibly significant properties — see, e.g., **?** for a catalog — the need for tools to analyze and possibly repair DNNs is strong.

As witnessed by an extensive survey **?** of more than 200 recent papers, the response from the scientific community has been equally strong. As a result, many algorithms have been proposed for the verification of neural networks and tools implementing them have been made available. Some examples of well-known and fairly mature verification tools are Marabou **?**, a satisfiability modulo theories (SMT)-based tool that answers queries regarding the properties of a DNN by transforming the queries into constraint satisfiability problems; ERAN **?**, a robustness analyzer based on abstract interpretation and MIPVerify **?**, another robustness analyzer based on mixed integer programming (MIP). Other widely-known verification tools are Neurify **?**, a robustness analyzer based on symbolic interval analysis and linear relaxation, NNV **?**, a tool implementing different methods for reachability analysis, Sherlock **?**, an output range analysis tool and NSVerify **?**, also for reachability analysis. A number of verification methodologies — without a corresponding tool — is also available like **?**, a game-based methodology for evaluating pointwise robustness of neural networks in safety-critical applications. Most of the above-mentioned tools and methodologies work only for feedforward fully-connected neural networks with ReLU activation functions, with some of them featuring verification algorithms for convolutional neural networks with different kinds of activation function. To the best of our knowledge, current state-of-the-art tools are restricted to verification/analysis tasks, in some cases they are limited to specific network architectures and they might prove difficult to use for practitioners and, in general, those who are not familiar with the complete background.

Our tool NEVER2 finds itself at the intersection of the issues explained above, and aims to bridge the gap between learning and verification of DNNs. NEVER2 borrows its design

philosophy from NEVER **?**, the first tool for automated learning, analysis and repair of neural networks. NEVER was designed to deal with multilayer perceptrons (MLPs) and its core was an abstraction-refinement mechanism described in **??**. As a system, one peculiar aspect of NEVER was that it included learning capabilities through the SHARK **?** library. Concerning the verification part, NEVER could utilize any solver integrating Boolean reasoning and linear arithmetic constraint solving — HYSAT **?** at the time. A further peculiarity of the approach was that NEVER could leverage abstract counterexamples to (try to) repair the MLP, i.e., retrain it to eliminate the causes of misbehaviour. NEVER2 relies on the PYNEVER API **?** and a first description of the system is available in **?**, where the verification capabilities were provided by external tools like Marabou, ERAN and MIPVerify. This Thesis describes the new abstraction-refinement procedure implemented by NEVER2 and formalizes all the theorems involved. The version of NEVER2 corresponding to this work is available online **?** under the Commons Clause (GNU GPL v3.0) license.

Chapter 2

Background

2.1 Neural Networks

Basic notation and definitions. We denote n-dimensional *vectors* of real numbers $x \in \mathbb{R}^n$ — also *points* or *samples* — with lowercase letters like x, y, z. We write $x = (x_1, x_2, \ldots, x_n)$ to denote a vector with its *components* along the n coordinates. We denote $x \cdot y$ the *scalar product* of two vectors $x, y \in \mathbb{R}^n$ defined as $x \cdot y = \sum_{i=1}^{n} x_i y_i$. The *norm* $\|x\|$ of a vector is defined as $\|x\| = \sqrt{x \cdot x}$. We denote sets of vectors $X \subseteq \mathbb{R}^n$ with uppercase letters like X, Y, Z. A set of vectors X is *bounded* if there exists $r \in \mathbb{R}, r > 0$ such that $\forall x, y \in X$ we have $d(x,y) < r$ where d is the *Euclidean norm* $d(x,y) = \|x - y\|$. A set X is *open* if for every point $x \in X$ there exists a positive real number ε_x such that a point $y \in \mathbb{R}^n$ belongs to X as long as $d(x,y) < \varepsilon_x$. The complement of an open set is a *closed* set — intuitively, one that includes its boundary, whereas open sets do not; closed and bounded sets are *compact*. A set X is *convex* if for any two points $x, y \in X$ we have that also $z \in X \ \forall z = (1 - \lambda)x + \lambda y$ with $\lambda \in [0, 1]$, i.e., all the points falling on the line passing through x and y are also in X. Notice that the intersection of any family, either finite or infinite, of convex sets is convex, whereas the union, in general, is not. Given any non-empty set X, the smallest convex set $\mathscr{C}(X)$ containing X is the *convex hull of* X and it is defined as the intersection of all convex sets containing X. A *hyperplane* $H \subseteq \mathbb{R}^n$ can be defined as the set of points

$$H = \{x \in \mathbb{R}^n \mid a_1 x_1 + a_2 x_2 + \ldots + a_n x_n = b\}$$

where $a \in \mathbb{R}^n$, $b \in \mathbb{R}$ and at least one component of a is non-zero. Let $f(x) = a_1 x_1 + a_2 x_2 + \ldots + a_n x_n - b$ be the affine form defining H. The *closed half-spaces associated with* H are defined as

$$H_+(f) = \{x \in X \mid f(x) \geq 0\} \qquad H_-(f) = \{x \in X \mid f(x) \leq 0\}$$

Notice that both $H_+(f)$ and $H_-(f)$ are convex. A *polyhedron* in $P \subseteq \mathbb{R}^n$ is a set of points defined as $P = \bigcap_{i=1}^{p} C_i$ where $p \in \mathbb{N}$ is a finite number of closed half-spaces C_i. A bounded polyhedron is a *polytope*: from the definition, it follows that polytopes are convex and compact in $\mathbb{R}^n$.

Neural networks. Given a finite number p of functions $f_1 : \mathbb{R}^n \to \mathbb{R}^{n_1}, \ldots, f_p : \mathbb{R}^{n_{p-1}} \to \mathbb{R}^m$ — also called *layers* — we define a *feed forward neural network* **?** as a function $\nu : \mathbb{R}^n \to \mathbb{R}^m$ obtained through the compositions of the layers, i.e., $\nu(x) = f_p(f_{p-1}(\ldots f_1(x)\ldots))$. The

layer f_1 is called *input layer*, the layer f_p is called *output layer*, and the remaining layers are called *hidden*. For $x \in \mathbb{R}^n$, we consider only two types of layers:

- $f(x) = Ax + b$ with $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$ is an *affine layer* implementing the linear mapping $f : \mathbb{R}^n \to \mathbb{R}^m$;
- $f(x) = (\sigma_1(x_1), \ldots, \sigma_n(x_n))$ is a *functional layer* $f : \mathbb{R}^n \to \mathbb{R}^n$ consisting of n *activation functions* — also called *neurons*; usually $\sigma_i = \sigma$ for all $i \in [1,n]$, i.e., the function σ is applied componentwise to the vector x.

We consider two kinds of activation functions $\sigma : \mathbb{R} \to \mathbb{R}$ that find widespread adoption: the *ReLU* function defined as $\sigma(r) = max(0,r)$, and the *logistic* function defined as $\sigma(r) = \frac{1}{1+e^{-r}}$. Although we do not consider them here, affine mappings can also represent convolutional layers with one or more filters **?**. For a neural network $\nu : \mathbb{R}^n \to \mathbb{R}^m$, the task of *classification* is about assigning to every input vector $x \in \mathbb{R}^n$ one out of m labels: an input x is assigned to a class k when $\nu(x)_k > \nu(x)_j$ for all $j \in [1,m]$ and $j \neq k$; the task of *regression* is about approximating a functional mapping from $\mathbb{R}^n$ to $\mathbb{R}^m$. In this regard, neural networks consisting of affine layers coupled with either ReLUs or logistic layers offer universal approximation capabilities **?**.

Verification task. Given a neural network $\nu : \mathbb{R}^n \to \mathbb{R}^m$ we wish to verify algorithmically that it complies to stated *post-conditions* on the output as long as it satisfies *pre-conditions* on the input. Without loss of generality[1], we assume that the input domain of ν is a bounded set $I \subset \mathbb{R}^n$. Therefore, the corresponding output domain is also a bounded set $O \subset \mathbb{R}^m$ because (i) affine transformations of bounded sets are still bounded sets, (ii) ReLU is a piecewise affine transformation of its input, (iii) the output of logistic functions is always bounded in the set $[0,1]$, and the composition of bounded functions is still bounded. We require that the logic formulas defining pre- and post-conditions are interpretable as finite unions of bounded sets in the input and output domains. Formally, given p bounded sets $X_1, \ldots, X_p$ in I such that $\Pi = \bigcup_{i=1}^{p} X_i$ and s bounded sets $Y_1, \ldots, Y_s$ in O such that $\Sigma = \bigcup_{i=1}^{s} Y_i$, we wish to prove that

$$\forall x \in \Pi . \nu(x) \in \Sigma. \tag{2.1}$$

While this query cannot express some problems regarding neural networks, e.g., invertibility or equivalence **?**, it captures the general problem of testing robustness against *adversarial*

[1] Input domains must be bounded to enable implementation of neural networks on digital hardware; therefore, also data from physical processes, which are potentially ubounded, are normalized within small ranges in practical applications.

perturbations **?**. For example, given a network $\nu : I \to O$ with $I \subset \mathbb{R}^n$ and $O \subset \mathbb{R}^m$ performing a classification task, we have that separate regions of the input are assigned to one out of m labels by ν. Let us assume that region $X_j \in I$ is classified in the j-th class by ν. We define an *adversarial region* as a set $\hat{X}_j$ such that for all $\hat{x} \in \hat{X}$ there exists at least one $x \in X$ such that $d(x,\hat{x}) \leq \delta$ for some positive constant δ. The network ν is *robust* with respect to $\hat{X}_j \subseteq I$ if, for all $\hat{x} \in \hat{X}_j$, it is still the case that $\nu(x)_j > \nu(x)_i$ for all $i \in [1,m]$ with $i \neq j$. This can be stated in the notation of condition (2.1) by letting $\Pi = \{\hat{X}_j\}$ and $\Sigma = \{Y_j\}$ with $Y_j = \{y \in O \mid y_j \geq y_i + \varepsilon, \forall i \in [1,n] \wedge i \neq j, \varepsilon > 0\}$. Analogously, in a regression task we may ask that points that are sufficently close to any input vector in a set $X \subseteq I$ are also sufficiently close to the corresponding output vectors. To do this, given the positive constants δ and ε, we let $\hat{X} = \{\hat{x} \in I \mid \exists x.(x \in X \wedge d(\hat{x},x) \leq \delta)\}$ and $\hat{Y} = \{\hat{y} \in O \mid \exists x.(x \in \hat{X} \wedge d(\hat{y}, \nu(x)) \leq \varepsilon)\}$ to obtain $\Pi = \{\hat{X}\}$ and $\Sigma = \{\hat{Y}\}$. Notice that, given our definition, we consider adversarial regions and output images that may not be convex.

Part I

Verification of Neural Networks

Chapter 3

Abstraction algorithms

In this chapter we show the algorithms and definitions that compose our abstraction model for the verification of neural networks by means of reachability analysis and robustness certification. We give the general definitions for abstracting domains, and afterwards we focus on how to propagate this abstraction throughout the activation layers.

3.1 Basic abstraction definitions

To enable algorithmic verification of neural networks, we consider the abstract domain $\langle \mathbb{R}^n \rangle \subset 2^{\mathbb{R}^n}$ of polytopes defined in $\mathbb{R}^n$ to abstract (families of) bounded sets into (families of) polytopes. We provide corresponding abstractions for affine and functional layers to perform abstract computations and we prove that their composition provides a consistent overapproximation of concrete networks.

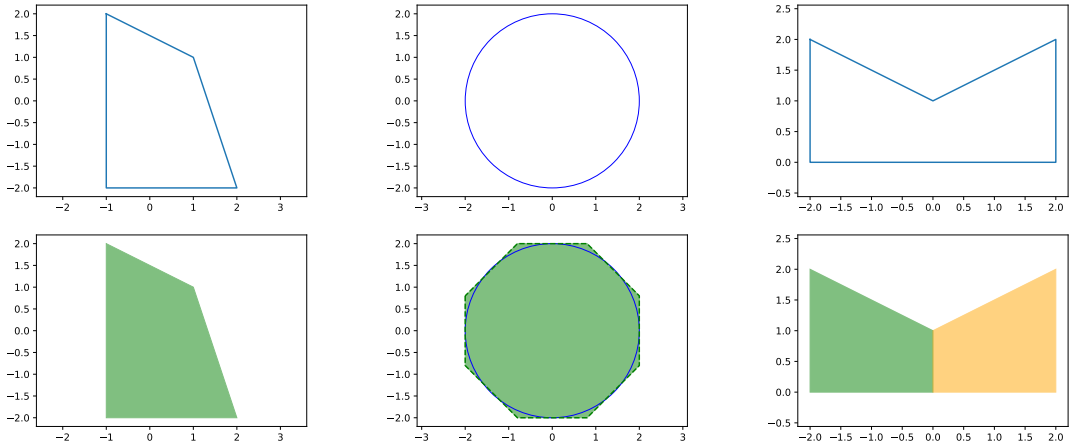
Definition 1 *(Abstraction)* Given a bounded set $X \subset \mathbb{R}^n$, an abstraction is defined as a function $\alpha : 2^{\mathbb{R}^n} \to \langle \mathbb{R}^n \rangle$ that maps X to a polytope P such that $\mathscr{C}(X) \subseteq P$.

Intutively, the function α maps a bounded set X to a corresponding polytope in the abstract space such that the polytope always contains the convex hull of X. Depending on X, the enclosing polytope may not be unique — see Figure 3.1 for different examples. However, given the convex hull of any bounded set, it is always possible to find an enclosing polytope. As shown in **?**, one could always start with an axis-aligned regular n simplex consisting of $n+1$ facets — e.g., the triangle in $\mathbb{R}^2$ and the tethraedron in $\mathbb{R}^3$ — and then refine the abstraction as needed by adding facets, i.e., adding half-spaces to make the abstraction more precise.

Definition 2 *(Concretization)* Given a polytope $P \in \langle \mathbb{R}^n \rangle$ a concretization is a function $\gamma : \langle \mathbb{R}^n \rangle \to 2^{\mathbb{R}^n}$ that maps P to the set of points cointained in it, i.e., $\gamma(P) = \{x \in \mathbb{R}^n \mid x \in P\}$.

Intutively, the function γ simply maps a polytope P to the corresponding (convex and compact) set in $\mathbb{R}^n$ comprising all the points contained in the polytope. As opposed to abstraction, the result of concretization is uniquely determined. We extend abstraction and concretization to finite families of sets and polytopes, respectively, as follows. Given a family of p bounded sets $\Pi = \{X_1, \ldots, X_p\}$, the abstraction of Π is a set of polytopes $\Sigma = \{P_1, \ldots, P_s\}$ such that $\alpha(X_i) \subseteq \bigcup_{i=1}^{s} P_i$ for all $i \in [1, p]$; when no ambiguity arises, we abuse notation and write $\alpha(\Pi)$ to denote the abstraction corresponding to the family Π. Given a family of s polytopes $\Sigma = \{P_1, \ldots, P_s\}$, the concretization of Σ is the union of the

Figure 3.1 Three possible abstractions of a set: the first row depicts the bounded set X, and the second the enclosing polytope P. Starting from the left, the first set is a convex set whose polytope matches perfectly. The second is not linear, and it is approximated with an octagon. The third is linear but non convex, therefore is split into two convex polytopes.



concretizations of its elements, i.e., $\bigcup_{i=1}^{s} \gamma(P_i)$; also in this case, we abuse notation and write $\gamma(\Sigma)$ to denote the concretization of a family of polytopes Σ.

Given our choice of abstract domain and a concrete network $\nu : I \to O$ with $I \subset \mathbb{R}^n$ and $O \subset \mathbb{R}^m$, we need to show how to obtain an *abstract neural network* $\tilde{\nu} : \langle I \rangle \to \langle O \rangle$ that provides a sound overapproximation of ν. To frame this concept, we introduce the notion of consistent abstraction.

Definition 3 *(Consistent abstraction)* Given a mapping $\nu : \mathbb{R}^n \to \mathbb{R}^m$, a mapping $\tilde{\nu} : \langle \mathbb{R}^n \rangle \to \langle \mathbb{R}^m \rangle$, abstraction function $\alpha : 2^{\mathbb{R}^n} \to \langle \mathbb{R}^m \rangle$ and concretization function $\gamma : \langle \mathbb{R}^m \rangle \to 2^{\mathbb{R}^m}$, the mapping $\tilde{\nu}$ is a consistent abstraction of ν over a set of inputs $X \subseteq I$ exactly when

$$\{\nu(x) \mid x \in X\} \subseteq \gamma(\tilde{\nu}(\alpha(X))) \tag{3.1}$$

The notion of consistent abstraction can be readily extended to families of sets as follows. The mapping $\tilde{\nu}$ is a consistent abstraction of ν over a family of sets of inputs $X_1 \dots X_p$ exactly when

$$\{\nu(x) \mid x \in \cup_{i=1}^{p} X_i\} \subseteq \gamma(\tilde{\nu}(\alpha(X_1, \dots, X_p))) \tag{3.2}$$

where we abuse notation and denote with $\tilde{\nu}(\cdot)$ the family $\{\tilde{\nu}(P_1), \dots, \tilde{\nu}(P_s)\}$ with $\{P_1, \dots, P_s\} = \alpha(X_1, \dots X_p)$

To represent polytopes and define the computations performed by abstract layers we resort to a specific subclass of *generalized star sets*, introduced in **?** and defined as follows — the notation is adapted from **?**.

Definition 4 *(Generalized star set)* Given a *basis matrix* $V \in \mathbb{R}^{n\times m}$ obtained arranging a set of m *basis vectors* $\{v_1, \dots v_m\}$ in columns , a point $c \in \mathbb{R}^n$ called *center* and a *predicate* $R : \mathbb{R}^m \to \{\top, \bot\}$, a generalized star set is a tuple $\Theta = (c, V, R)$. The set of points represented by the generalized star set is given by

$$[\![\Theta]\!] \equiv \{z \in \mathbb{R}^n \mid z = Vx + c \text{ such that } R(x_1, \dots, x_m) = \top\} \tag{3.3}$$

In the following we denote $[\![\Theta]\!]$ also as Θ. Depending on the choice of R, generalized star sets can represent different kinds of sets, but we consider only those such that $R(x) := Cx \leq d$, where $C \in \mathbb{R}^{p\times m}$ and $d \in \mathbb{R}^p$ for $p \geq 1$, i.e., R is a conjunction of p linear constraints as in **?**; we further require that the set $Y = \{y \in \mathbb{R}^m \mid Cy \leq d\}$ is bounded.

Proposition 1 *Given a generalized star set $\Theta = (c, V, R)$ such that $R(x) := Cx \leq d$ with $C \in \mathbb{R}^{p\times m}$ and $d \in \mathbb{R}^p$, if the set $Y = \{y \in \mathbb{R}^m \mid Cy \leq d\}$ is bounded, then the set of points represented by Θ is a polytope in $\mathbb{R}^n$, i.e., $\Theta \in \langle \mathbb{R}^n \rangle$.*

The proof of proposition (1) is straightforward, since the set Y is a polytope in $\mathbb{R}^m$, the mapping $Vx + c$ is an affine mapping from $\mathbb{R}^m$ to $\mathbb{R}^n$ and affine mappings of polytopes are still polytopes. From **?** we know that polytopes can be represented as generalized star sets, and thus our restricted form of star sets provides an equivalent representation of polytopes in $\mathbb{R}^n$; in the following, we refer to generalized star sets obeying our restrictions simply as *stars*.

The simplest abstract layer to obtain is the one abstracting affine transformations. As we have already mentioned, affine transformations of polytopes are still polytopes, so we just need to define how to apply an affine transformation to a star — the definition is adapted from **?**.

Definition 5 *(Abstract affine mapping)* Given a star set $\Theta = (c, V, R)$ and an affine mapping $f : R^n \to R^m$ with $f = Ax + b$, the abstract affine mapping $\tilde{f} : \langle R^n \rangle \to \langle R^m \rangle$ of f is defined as $\tilde{f}(\Theta) = (\hat{c}, \hat{V}, R)$ where

$$\hat{c} = Ac + b \qquad \hat{V} = AV$$

Intuitively, the center and the basis vectors of the input star Θ are affected by the transformation of f, while the predicates remain the same.

Proposition 2 *Given an affine mapping $f : \mathbb{R}^n \to \mathbb{R}^m$, the corresponding abstract mapping $\tilde{f} : \langle\mathbb{R}^n\rangle \to \langle\mathbb{R}^m\rangle$ provides a consistent abstraction over any bounded set $X \subset \mathbb{R}^n$, i.e., $\{f(x) \mid x \in X\} \subseteq \gamma(\tilde{f}(\alpha(X)))$ for all $X \subset \mathbb{R}^n$.*

To prove proposition (2), we observe that the set $\alpha(X)$ is any polytope P such that $P \supseteq \mathscr{C}(X)$ — equality holds only when X is already a polytope, and thus $X \equiv \mathscr{C}(X) \equiv P$. Let $\Theta_P = (c_P, V_P, R_P)$ be the star corresponding to P defined as

$$c_P = 0^n \qquad V_P = I^n \qquad R_P = C_P x + d_P \leq 0$$

where 0^n is the n-dimensional zero vector, and I^n is the $n \times n$ identity matrix — the columns of I^n correspond to the standard orthonormal basis $e_1, \ldots, e_n$ of $\mathbb{R}^n$, i.e., $\|e_i\| = 1$ and $e_i \cdot e_j = 0$ for all $i \neq j$ with $i, j \in [1, n]$; the matrix $C_P \in \mathbb{R}^{q \times n}$ and the vector $d_P \in \mathbb{R}^q$ collect the parameters defining q half-spaces whose intersection corresponds to P. Given our choice of c and V, it is thus obvious that $\Theta_P \equiv P$. Recall that $f = Ax + b$ with $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$; from definition (5) we have that $\tilde{f}(\Theta_P) = \hat{\Theta}_P$ with $\hat{\Theta}_P = (\hat{c}_P, \hat{V}_P, R_P)$ and

$$\hat{c}_P = A0^n + b = b \qquad \hat{V}_P = AI^n = A$$

The concretization of $\hat{\Theta}_P$ is just the set of points contained in $\hat{\Theta}_P$ defined as

$$\gamma(\hat{\Theta}_P) = \{z \in \mathbb{R}^m \mid z = Ax + b \text{ such that } C_p x \leq d_P\} \tag{3.4}$$

Now it remains to show that $\{f(x) \mid x \in X\} \subseteq \gamma(\hat{\Theta}_P)$. This follows from the fact that, for a generic $y \in \{f(x) \mid x \in X\}$ there must exists $x \in X$ such that $y = Ax + b$; since x satisfies $C_p x \leq d_P$ by construction of P, it is also the case that $y \in \gamma(\hat{\Theta}_P)$ by definition (3.4).

3.2 ReLU abstraction algorithms

Algorithm 1 **?** defines the abstract mapping of a functional layer with n ReLU activation functions and adapts the methodology proposed in **?**. The function COMPUTE_LAYER takes as input an indexed list of N stars $\Theta_1, \ldots, \Theta_N$ and an indexed list of n positive integers called *refinement levels*. For each neuron, the refinement level tunes the grain of the abstraction: level 0 corresponds to the coarsest abstraction that we consider — the greater the level, the finer the abstraction grain. In the case of ReLUs, all non-zero levels map to the same (precise) refinement, i.e., a piecewise affine mapping. The output of function COMPUTE_LAYER is

Algorithm 1 Abstraction of the ReLU activation function.

1: **function** COMPUTE_LAYER(*input* = $[\Theta_1,\ldots,\Theta_N]$, *refine* = $[r_1,\ldots,r_n]$)
2: *output* = [ ]
3: **for** $i = 1 : N$ **do**
4: *stars* = $[\Theta_i]$
5: **for** $j = 1 : n$ **do** *stars* = COMPUTE_RELU(*stars*, j, *refine*$[j]$, n)
6: APPEND(*output*, *stars*)
7: **return** *output*

8: **function** COMPUTE_RELU(*input* = $[\Gamma_1,\ldots,\Gamma_M]$, j, *level*, n)
9: *output* = [ ]
10: **for** $k = 1 : M$ **do**
11: (lb_j, ub_j) = GET_BOUNDS(*input*$[k]$, j)
12: $M = [e_1 \ldots e_{j-1}\ 0\ e_{j+1} \ldots e_n]$
13: **if** $lb_j \geq 0$ **then** S = *input*$[k]$
14: **else if** $ub_j \leq 0$ **then** $S = M$ * *input*$[k]$
15: **else**
16: **if** *level* > 0 **then**
17: Θ_{low} = *input*$[k] \wedge z[j] < 0$; Θ_{upp} = *input*$[k] \wedge z[j] \geq 0$
18: $S = [M * \Theta_{low}, \Theta_{upp}]$
19: **else**
20: $(c, V, Cx \leq d)$ = *input*$[j]$
21: $C_1 = [0\ 0\ \ldots\ -1] \in \mathbb{R}^{1\times m+1}$, $d_1 = 0$
22: $C_2 = [V[j,:]\ -1] \in \mathbb{R}^{1\times m+1}$, $d_2 = -c_k[j]$
23: $C_3 = [\frac{-ub_j}{ub_j - lb_j} \cdot V[j,:]\ -1] \in \mathbb{R}^{1\times m+1}$, $d_3 = \frac{ub_j}{ub_j - lb_j}(c[j] - lb_j)$
24: $C_0 = [C\ 0^{m\times 1}]$, $d_0 = d$
25: $\hat{C} = [C_0;\ C_1;\ C_2;\ C_3]$, $\hat{d} = [d_0;\ d_1;\ d_2;\ d_3]$
26: $\hat{V} = MV$, $\hat{V} = [\hat{V}\ e_j]$
27: $S = (Mc, \hat{V}, \hat{C}\hat{x} \leq \hat{d})$
28: APPEND(*output*, S)
29: **return** *output*

still an indexed list of stars, that can be obtained by independently processing the stars in the input list. For this reason, the **for** loop starting at line 3 can be parallelized to speed up actual implementations.

Given a single input star $\Theta_i \in \langle R^n \rangle$, each of the n dimensions is processed in turn by the **for** loop starting at line 5 and involing the function COMPUTE_RELU. Notice that the stars obtained processing the j-th dimension are feeded again to COMPUTE_RELU in order to process the $j+1$-th dimension. For each star given as input, the function COMPUTE_RELU first computes the lower and upper bounds of the star along the j-th dimension by solving two linear-programming problems — function GET_BOUNDS at line 11. Independently from

the abstraction level, if $lb_j \geq 0$ then the ReLU acts as an identity function (line 13), whereas if $ub_j \leq 0$ then the j-th dimension is zeroed (line 14). The $*$ operator takes a matrix M, a star $\Gamma = (c, V, R)$ and returns the star (Mc, MV, R). In this case, M is composed of the standard orthonormal basis in $\mathbb{R}^n$ arranged in columns, with the exception of the j-th dimension which is zeroed.

3.2.1 Exact abstract propagation

When $lb_j < 0$ and $ub_j > 0$ we consider the refinement level. For any non-zero level, the input star is "split" into two new stars, one considering all the points $z < 0$ (Θ_{low}) and the other considering points $z \geq 0$ (Θ_{upp}) along dimension j. Both Θ_{low} and Θ_{upp} are obtained by adding to the input star *input[k]* the appropriate constraints. If the analysis at lines 17–18 is applied throughout the network, and the input abstraction is precise, then the abstract output range will also be precise, i.e., it will coincide with the concrete one: we call complete the analysis of NEVER2 in this case. The number of resulting stars is worst-case exponential, therefore the complete analysis may result computationally infeasible.

Proposition 3 *Given a ReLU mapping $f : \mathbb{R}^n \to \mathbb{R}^n$, the corresponding abstract mapping $\tilde{f} : \langle \mathbb{R}^n \rangle \to \langle \mathbb{R}^n \rangle$ defined in Algorithm 1 provides a consistent abstraction over any bounded set $X \subset \mathbb{R}^n$, i.e., $\{f(x) \mid x \in X\} \subseteq \gamma(\tilde{f}(\alpha(X)))$ for all $X \subset \mathbb{R}^n$.*

Proposition 4 *Given a concrete network $\nu : \mathbb{R}^n \to \mathbb{R}^m$ comprised of a finite number p of layers $f_1 : \mathbb{R}^n \to \mathbb{R}^{n_1}, \ldots, f_p : \mathbb{R}^{n_{p-1}} \to \mathbb{R}^m$ such that each f_i is either an affine or functional layer implementing ReLUs, the corresponding abstract network $\tilde{\nu} : \langle \mathbb{R}^n \rangle \to \langle \mathbb{R}^m \rangle$ comprised of the corresponding abstract layers $\tilde{f}_1 : \langle \mathbb{R}^n \rangle \to \langle \mathbb{R}^{n_1} \rangle, \ldots, \tilde{f}_p : \langle \mathbb{R}^{n_{p-1}} \rangle \to \langle \mathbb{R}^m \rangle$ provides a consistent abstraction over any bounded set $X \subset \mathbb{R}^n$, i.e., $\{\nu(x) \mid x \in X\} \subseteq \gamma(\tilde{\nu}(\alpha(X)))$ for all $X \subset \mathbb{R}^n$.*

Proposition 4 enforces that we can prove the (local) robustness of a neural network by propagating the abstraction of an input set representing the l_∞ ball around a given input with a small perturbation ε and check whether the output set is large enough to cause a misclassification.

Proposition 5 *The intersection of a star $\Theta = (c, V, R)$ and a half-space $\mathscr{H} = \{z | Hz \leq g\}$ is another star with the following characteristics: $\overline{\Theta} = \Theta \cap \mathscr{H} = (\overline{c}, \overline{V}, \overline{R})$ with $\overline{c} = c$, $\overline{V} = V$, $\overline{R} = R \wedge R'$ and $R'(x) = (HV)x \leq g - Hc$*

The proof of the proposition is straightforward since it is analogous to adding new constraints to the predicate of the star, as done for the ReLU abstract transformer.

Proposition 6 *Let $[\Theta_1,...,\Theta_n]$ be a star set obtained by applying Algorithm 1 to a network of interest ν and an input star set corresponding to the input component of the property of interest P. Moreover let $\hat{\mathscr{H}}$ be an half-space corresponding to the unsafe zone as defined by the property of interest. If $\overline{\Theta}_i = \Theta_i \wedge \hat{\mathscr{H}} = \emptyset$ for $i = 1,...,n$ then the neural network ν satisfy the property P.*

Proposition 7 *If in Algorithm 1 the stars were always refined for all neurons it is possible to compute the complete counter input set (i.e., the set containing all possible inputs that make the neural network unsafe) as $\mathscr{C}_\Theta = \bigcup_i (c,V,\overline{R}_i)$ where $\overline{R}_i$ are the predicates of the stars obtained by the intersection between the unsafe zone and the output star set, whereas c and V are the center and basis matrix of the input star.*

Proof 1 *If the complete version of Algorithm 1 is used then all the stars in the computation process are defined on the same predicate variables $\boldsymbol{x} = [x_1,...,x_m]$ which do not change during the computations since only the number of constraints on $\boldsymbol{x}$ is changed by the abstract transformers. As consequence the $\overline{R}_i$ contain values of $\boldsymbol{x}$ that make the network unsafe, moreover it also contains all the constraints of the base predicate R of the input star. Therefore the complete counter input set containing all possible inputs that make the neural network unsafe is $\mathscr{C}_\Theta = \bigcup_i (c,V,\overline{R}_i)$, $\overline{R}_i \neq \emptyset$.*

3.2.2 Over-approximate abstract propagation

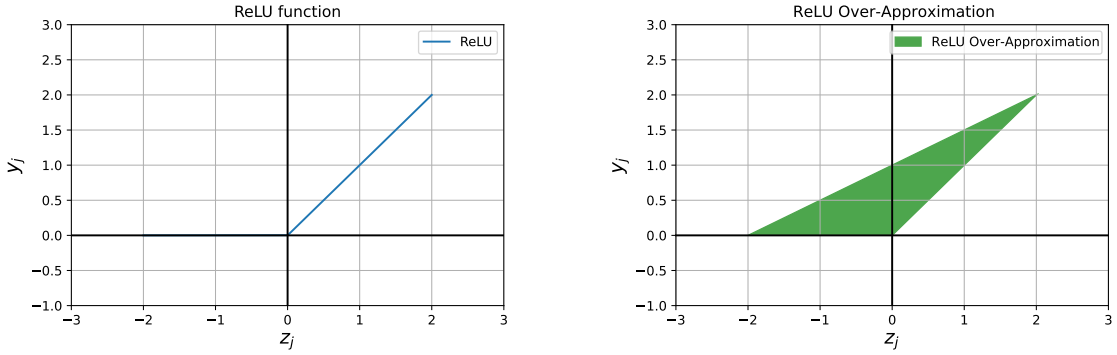
If the refinement level is 0, then the ReLU is abstracted using the over-approximation proposed in **?** and depicted in Figure 3.2. This approach is much less conservative than others, i.e., based on zonotopes or abstract domains, and provides a tighter abstraction.

As can be seen in Figure 3.2, three constraints are needed to construct the over-approximation:

$$
\begin{aligned}
& y_j \geq 0 \\
& y_j \geq z_j \\
& y_j \leq ub_j \frac{z_j - lb_j}{ub_j - lb_j}
\end{aligned}
$$

Such constraints must be added to the predicate matrix of the star, therefore we define an auxiliarly variable x_{m+1} and we modify the basis matrix so that $y_j = x_{m+1}$ (line 26 in

Figure 3.2 Graphical representation of the ReLU function (left) and the over-approximation considering a single variable (right) with $lb_j = -2$ and $ub_j = 2$.



Algorithm 1). By doing so we make it possible to express our constraints only in terms of the predicate variables. We remember that $z_j = V_j\mathbf{x} + c_j$, substituting it in the constraints we obtain:

$$
\begin{aligned}
x_{m+1} &\geq 0 \\
x_{m+1} &\geq V_j\mathbf{x} + c_j \\
x_{m+1} &\leq ub_j \cdot \frac{V_j\mathbf{x} + c_j - lb_j}{ub_j - lb_j}
\end{aligned}
$$

If we reorder these constraints we can bring them in the format $C\mathbf{x} \leq \mathbf{d}$:

$$
\begin{aligned}
-x_{m+1} &\leq 0 \\
V_j\mathbf{x} - x_{m+1} &\leq -c_j \\
-\frac{ub_j}{ub_j - lb_j}V_j\mathbf{x} + x_{m+1} &\leq \frac{ub_j}{ub_j - lb_j}(c_j - lb_j)
\end{aligned}
$$

From these constraints it is straightforward to identify the corresponding matrices in lines 21 to 23 of the algorithm.

If this analysis is carried out throughout the network, then the output star will be a (sound) over-approximation of the concrete output range: we call *over-approximate* the analysis of NEVER2 in this case. The number of star remains the same throughout the analysis, but at the cost of a new predicate variable for each neuron which, in turn, increases the complexity of the linear program required by GET_BOUNDS.

3.2.3 Mixed abstract propagation

In **?** it is proposed a new approach that adopts different levels of abstraction during the analysis: since each neuron features its own refinement level, algorithm 1 controls the abstraction down to the single neuron. This setting strikes a trade-off between complete and over-approximate settings. In order to reduce as much as possible the approximation error, we rank the neurons in each layer based on the area of the over-approximation triangle depicted in Figure 3.2: intuitively, the neuron with the widest bounds introduces a broader triangle and, by design, a bigger approximation.

We concretize the star along that neuron and propagate the approximate method along the others, such that each layer results in at most a single split. This reduces the computational cost significantly, as the growth becomes quadratic in the number of layers and the complexity increase by the approximation is contained. We call *mixed* the analysis of NEVER2 in this case.

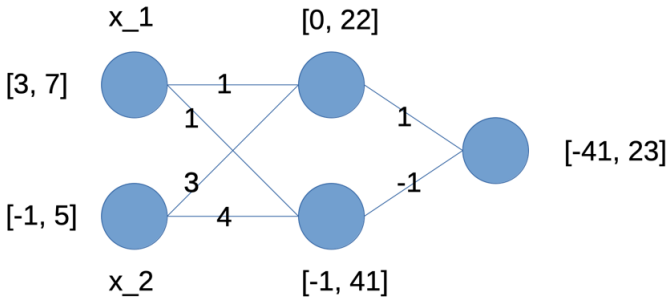
Part II

Bound Propagation

Chapter 4

Bound Propagation Introduction

Figure 4.1 Example of naive propagation



4.1 Bound propagation types

The main goal of this chapter is to explain how we can apply a bound propagation on a Neural Network, made up of only fully connected layers at first, then also ReLU layers. Assume we have a property that set up an upper and lower bound for each neuron of the input. Essentially, let $\mathbf{x}$ be the set of input neurons and x_i the i-th neuron of $\mathbf{x}$. We can apply different types of bound propagation. The first that will be dealt with is the **naive propagation**.

4.1.1 Naive Propagation

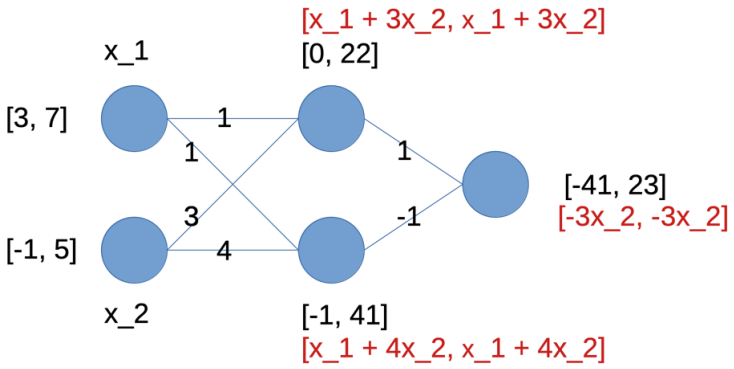
In Fig 4.1 represents a neural network consisting of two layers fully connected.
By performing the products and sums between the intervals and the fully connected weights along the layers we get an output interval [-41, 23].
Let the affine mapping (fc) be defined as $\mathbf{y} = W\mathbf{x} + \mathbf{b}$ and W^+ and W^- be the positive and negative entries of W.
We also define l_i^+ and l_i^- respectively as the vectors of the upper bounds and of the lower bounds.

For calculating the intervals of the i-th fully connected layer for all nodes:

$$\begin{aligned} lower\_bound\_i &= W^+ l_{i-1}^- + W^- l_{i-1}^+ + b \\ upper\_bound\_i &= W^+ l_{i-1}^+ + W^- l_{i-1}^- + b \end{aligned} \tag{4.1}$$

Now, note that the upper bound 23 of the output cannot be reached. It can appear only if the upper and lower nodes of the hidden layers are respectively 22 and -1 but this is impossible. This is because in order to have 22 in the upper node of the hidden layer it is necessary to have $x_1 = 7$ and $x_2 = 5$ as input values but to have -1 in the lower node

Figure 4.2 Example of bound propagation



of the hidden layer it is necessary to have $x_1 = 3$ and $x_2 = -1$ at the same time. These two conditions are contradictory and therefore can't be satisfied, this is due to the fact that this is an overestimation of the real bounds. It is also an example of the **dependency problem**. Naive interval analysis suffers from large overestimation errors as it ignores the input dependencies during interval propagation.

4.1.2 Symbolic Interval Propagation

In Fig 4.2 we use a different approach to overestimate the output interval. The main idea is to minimize overestimation by explicitly representing the intermediate computations of each neuron using symbolic intervals. These intervals encode the interdependency of the inputs and help in accurately estimating the output range of the neuron. In simpler terms, symbolic interval propagation is a technique that optimizes the calculation process within a neuron by using symbolic intervals to better understand the relationships between inputs and reduce the tendency to overestimate the output.
This approach on average, reduces significantly the over-approximation error with respect to the naive propagation.

In our example the intervals, made up respectively from a lower and an upper bound vector, associated to the nodes of the input layers $[x\_1, x\_2]$ are $l_0^- = [3, -1]$ and $l_0^+ = [7, 5]$. Instead $[x_1 + 3x_2, x_1 + 3x_2]$ and $[x_1 + 4x_2, x_1 + 4x_2]$ are the lower and upper symbolic bounds of the intermediate nodes.

Below are the procedure and formulas to compute the symbolic bounds of the i-th layer:

1. For the input layer instantiate two identity matrix whose size is the number of input neuron and two zeroes offset vectors. In our example, we would instantiate 2 2x2 matrices and two zero-vectors of 2 elements.

$$lower_0 = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix}$$

$$upper_0 = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix}$$

$$lower\_offset_0 = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix}$$

$$upper\_offset_0 = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix}$$

2. Let $upper_i$ and $lower_i$ respectively the matrix of the symbolic upper bound equations and lower bound equation of a fully connected layer i. A fully connected layer is characterised by a weight matrix W and a bias vector b.
   Below are the formulas to calculate the upper symbolic bound and the lower symbolic bounds after a fully connected layer:

$$\begin{aligned} lower_i &= W_i^+ lower_{i-1} + W_i^- upper_{i-1} \\ upper_i &= W_i^+ upper_{i-1} + W_i^- lower_{i-1} \\ lower\_offset_i &= W_i^+ lower\_offset_{i-1} + W_i^- upper\_offset_{i-1} \\ upper\_offset_i &= W_i^+ upper\_offset_{i-1} + W_i^- lower\_offset_{i-1} \end{aligned} \tag{4.2}$$

   where:
   $lower_{i-1}$ and $lower_{i-1}$ are the upper and lower symbolic bounds matrixes of the previ-

ous layer.

By iterating the above formula through all layers of a complete fully connected network, you get the symbolic upper and lower bounds equations for each fc layer.

Given the symbolic upper and lower bounds matrices $lower\_i$, $upper\_i$, $lower\_offset_i$ and $upper\_offset_i$ of layer i and a couple of vectors (the upper and lower bound vector l_0^+ l_0^-) on the input layer you can calculate the **numeric** lower and upper bounds of layer i. The formulas are below:

$$\begin{aligned} numeric\_lower_i &= lower_i^+ \cdot l_0^- + lower_i^- \cdot l_0^+ + offset_i \\ numeric\_upper_i &= upper_i^+ \cdot l_0^+ + lower_i^- \cdot l_0^- + offset_i \end{aligned} \tag{4.3}$$

where:
$numeric\_lower_i$ and $numeric\_upper_i$ are the lower and upper numeric bounds vectors of layer i
l_0^- and l_0^+ are the lower and upper bounds vectors of input layer

By iterating this process for all layers, we get the numeric and numeric bounds for all the layer of the network.
The main problem that arises is that we often need to deal with networks formed by FC and ReLU. The rectified linear activation function or ReLU for short is a piecewise linear function that will output the input directly if it is positive, otherwise, it will output zero. But we can't deal with it because it is not a linear and, therefore, we have to use a convex approximation to deal with it.

4.1.3 Iterative Refinement Propagation

In fig 4.3, the input interval on a node is divided into two adjacent sub-intervals of equal size, effectively cutting it in half. This division helps decrease the overestimation and allows us to narrow down the range of possible values for the output. As a matter of fact the output interval is tighter respect to the one got with the naive propagation. It's important to note that we can continue refining the output interval by repeatedly splitting the input intervals in order to further tighten the overapproximation. This process is easily parallelizable because each split sub-interval can be independently checked.

Figure 4.3 Example of bisection and refinement

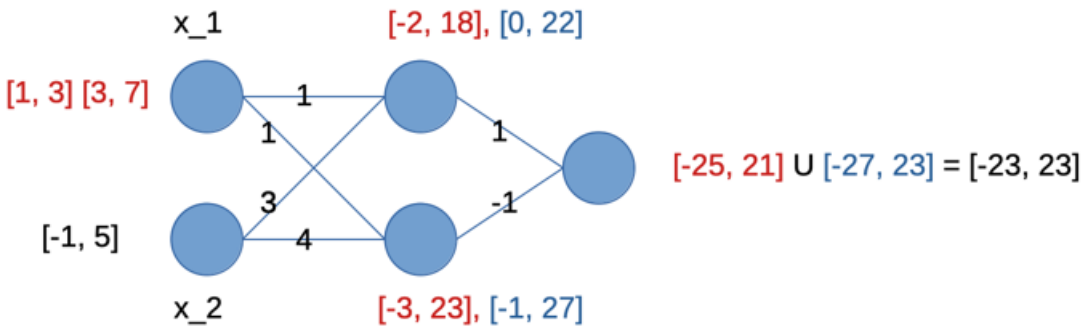
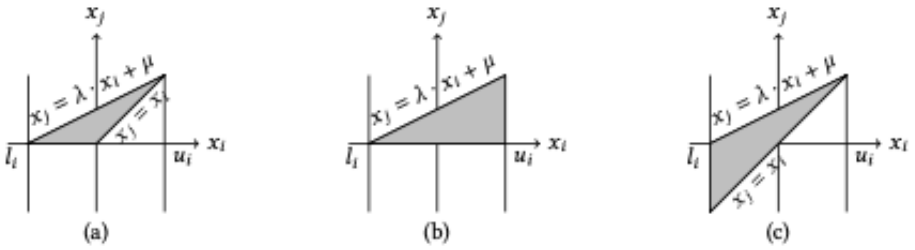


Figure 4.4 Main used ReLU's convex approximation in the literaturet



4.2 ReLU's Over-approximation

The Rectified Linear Unit (ReLU) activation function is a commonly used activation function in neural networks. It is defined as follows:

$$ReLU(x) = max(0,x)$$

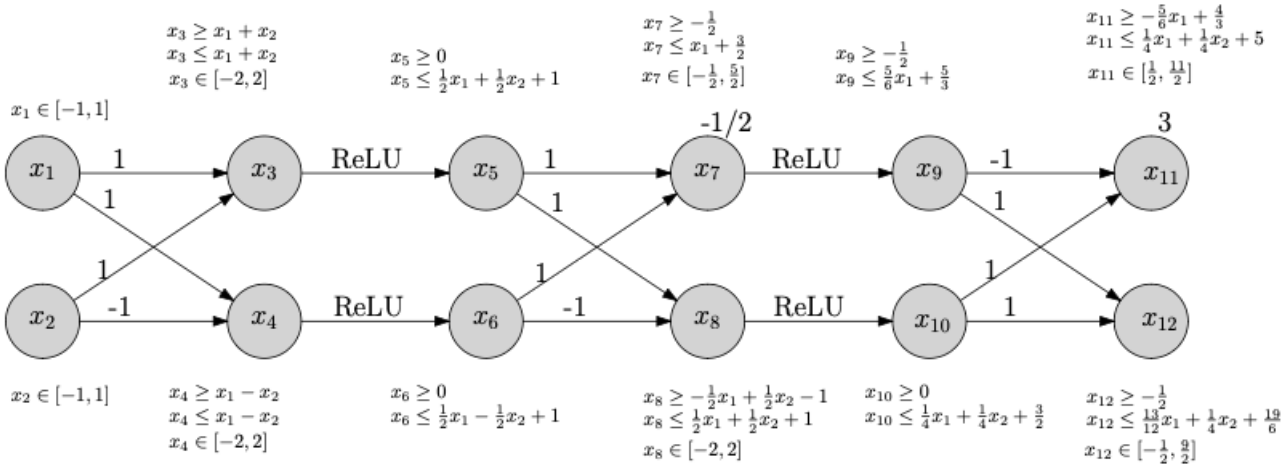
In other words, if the input to the ReLU function (x) is greater than zero, the output will be equal to the input. If the input is less than or equal to zero, the output will be zero.
The ReLU function has a simple and computationally efficient implementation, which contributes to its popularity. It introduces non-linearity into the network, allowing neural networks to learn complex relationships between inputs and outputs. ReLU is particularly useful in deep neural networks as it helps alleviate the vanishing gradient problem, where gradients become very small during backpropagation, by allowing the gradient to flow more easily for positive inputs.

With regard to symbolic interval bound propagation the main problem that arises is that we often need to deal with networks formed by FC and ReLU. But we can't deal with it because it is not a linear and, therefore, we have to use a convex over-approximation. Below the main used convex over-approximation:

1. The approximation of Fig. 4 (a) minimizes the area in the x_i , x_j plane, and would add the following relational constraints and concrete bounds for x_j:

$$\begin{aligned} &x_i \leq x_j, 0 \leq x_j, \\ &x_j \leq u_i \left(x_i - l_i\right) / \left(u_i - l_i\right). \\ &l_j = 0, u_j = u_i. \end{aligned} \tag{4.4}$$

Figure 4.5 Simple neural network with FC and ReLU



This ReLU approximation is the best possible in term of area but it has the disadvantage that uses 2 lower constraints. For this reason it is not well suitable for a further implementation in a bound propagation alghoritm.

2. The approximation from Fig. 4 (b) adds the following constraints and bounds for x_j :

$$\begin{aligned} &0 \leq x_j \leq u_i \left(x_i - l_i\right) / \left(u_i - l_i\right), \\ &l_j = 0, u_j = u_i. \end{aligned} \tag{4.5}$$

3. The approximation from Fig. 4 (c) adds the following constraints and bounds:

$$\begin{aligned} &x_i \leq x_j \leq u_i \left(x_i - l_i\right) / \left(u_i - l_i\right), \\ &l_j = l_i, u_j = u_i. \end{aligned} \tag{4.6}$$

4.2.1 Application of ReLU approximation to bound propagation

The networks used for verification purposes are typically small and typically consist of fully connected layers and ReLU activation layers. Earlier, we explained the process of bound propagation for networks composed solely of fully connected layers, referred to as Symbolic Interval propagation. Now, let's discuss how to modify the bound propagation method to make it compatible with ReLU layers.

Consider a network composed of three fully connected (FC) layers with dimensions 2x2, along with two ReLU layers, as shown in Figure 5 of the paper titled "Optimized Symbolic Interval Propagation for Neural Network Verification."

To incorporate ReLU layers into the bound propagation process, we need to account for the behavior of the ReLU activation function. Below are the steps to modify bound propagation for networks containing ReLU layers:

1. Symbolic Interval Propagation (SIP) with Fully Connected (FC) Layers:

    - Initially, we start with an input layer, where the input bounds are represented symbolically.
    - For each FC layer, we calculate the affine transformation symbolic bounds as seen previously.
    - The numeric output bounds of the FC layer are obtained through the concretization operation of the symbolic bounds and input bounds

2. Incorporating ReLU layers:

    - After each FC layer, introduce a ReLU layer, which applies the ReLU activation function element-wise to the output of the FC layer.

3. Bound Propagation through ReLU Layers

    - For each ReLU layer, we compute the lower and upper bounds of the output based on the input bounds.
    - If the lower bound of the input is greater than zero, the lower bound of the output remains the same as the lower bound of the input. This is intuitive because if the lower bound is greater that zero then we can ignore the ReLU second dial of the Carthesian plan and treat it as a simple bisector of the first quadrant.
    - If the upper bound of the input is less than or equal to zero, the upper bound of the output is zero. This is intuitive because if the upper bound il less that zero than we can ignore the ReLU first dial of the Carthesian plan and treat it as a simple zero function.
    - If the lower bound is less than zero and the upper bound is greater than 0 then it is needed to apply an over-approximation of the ReLU symbolic bounds.
      In the example in the Fig 4.5 all the neurons are in the third case and it is used the following metric for the linearization:
      If $u < |l|$ is applied the ReLU over-approximation in Fig 4.4 b
      Else If $u > |l|$ is applied the ReLU over-approximationin in Fig 4.4 c

This diversification in the over-approximation's choice is due to the fact that if $u < |l|$ the over-approximation in Fig 4.4 b introduces a smaller error than the one in Fig 4.4 c and vice-versa. The over-approximation of the ReLU introduces a discrepancy between the symbolically propagated limits and the actual value limits, causing a potential over-estimation errors in limit propagation which increases as the number of ReLu layers increases and consequently as the depth of the network increases.

4. Iterative process:

   - Apply the modified bound propagation process iteratively across all FC and ReLU layers in the network.
   - The output bounds obtained from the last FC layer represent the output bounds of the network.

4.2.2 Detailed Study of Symbolic Linear Relaxation of ReLU

The key insight of symbolic linear relaxation is to find the tightest linear bounds for the ReLU function, minimizing the overestimation error when approximating network outputs. However, it is important to note that the approximation errors may vary for overestimated nodes in different layers, depending on the symbolic intervals assigned as inputs.

For layers preceding the n_0-th layer, which is the first layer with overestimated nodes, all nodes have the same symbolic lower and upper bounds as you can notice in the first ReLU layer of the example.

However, in deeper layers where overestimated nodes become more frequent, the expressions for the lower and upper symbolic bounds can be more complex. To address this challenge, we provide a detailed discussion and illustration of how symbolic linear relaxation works. The linearization below in not present in section 2, but it is the tightest linearization possible and it is the one that will be used in the bound propagation. This is due to the fact that it introduces the smallest error possible respect to the linearizations seen before.

Let's consider an arbitrary overestimated node A, with the equation given by $z = Relu(Eq)$, where Eq represents its input kept as a symbolic interval $[Eq_{low}, Eq_{up}]$. Additionally, let n_0 represent the first layer where overestimated nodes occur, n_A denote the layer in which node A appears, and (l_{low}, u_{low}) and (l_{up}, u_{up}) represent the concrete lower and upper bounds of A's symbolic bounds Eq_{low} and Eq_{up}.

We can consider several cases for the symbolic linear relaxations applied to node A, depending on its position relative to the n_0 layer.

1. if $n_0 == n_A$: if A is an overstimed node in n_0 than we see that A's input equation Eq satisfies $Eq_{low} = Eq_{up} = Eq$. Due to the fact that $u > 0$ and $l < 0$ the output can be approximated with:

$$\text{Relu}([Eq, Eq]) \mapsto \left[\frac{u}{u-l}Eq, \frac{u}{u-l}(Eq-l)\right] \tag{4.7}$$

2. $n_A > n_0$: If A is an overestimated node after n_0-th layer, possibly its symbolic lower bound equation Eq_{low} is no longer the same as its upper bound equation Eq_{up} before relaxation. Though we can still approximate the it as $\text{Relu}\left([Eq_{low}, Eq_{up}]\right) \mapsto \left[\frac{u_{up}}{u_{up}-l_{low}}Eq_{low}, \frac{u_{up}}{u_{up}-l_{low}}\left(Eq_{up} - l_{\text{low}}\right)\right]$, this is not the tightest possible bound. Therefore, we consider bounds on Eq_{low} and Eq_{up} independently to achieve tighter approximations.

$$\text{Relu}\left([Eq_{\text{low}}, Eq_{up}]\right) \mapsto \begin{cases} \left[0, \frac{u_{up}}{u_{up}-l_{up}}\left(Eq_{up} - l_{up}\right)\right] & (l_{\text{low}} \leq 0, l_{up} \leq 0, u_{low} \leq 0, u_{up} > 0) \\ [0, Eq_{up}] & (l_{\text{low}} \leq 0, l_{up} \leq 0, u_{low} > 0, u_{up} > 0) \\ \left[\frac{u_{low}}{u_{low}-l_{low}}Eq_{low}, \frac{u_{up}}{u_{up}-l_{up}}\left(Eq_{up} - l_{up}\right)\right] & (l_{\text{low}} \leq 0, l_{up} > 0, u_{low} \leq 0, u_{up} > 0) \\ \left[\frac{u_{\text{low}}}{u_{\text{low}}-l_{low}}Eq_{low}, Eq_{up}\right] & (l_{\text{low}} \leq 0, l_{up} \leq 0, u_{low} > 0, u_{up} > 0) \end{cases} \tag{4.8}$$

Chapter 5

PyNeVer BP Application

5.1 Bound Propagation applied to PyNeVer

The PyNeVer algorithm allows for property verification using different levels of refinement. The higher the level of refinement, the more efficient the algorithm becomes. However, as the level of refinement increases, the resulting polytope from the network will be larger than the real one, leading to decreased accuracy in the verification process.
Nevertheless, to achieve an exact computation of the output polytope for a given network and property, the minimum level of over-approximation must be used.
The objective of this thesis is to leverage bound propagation to improve the efficiency of the PyNeVer algorithm and achieve better time performance. The idea is to utilise bound propagation to determine the node bounds in advance, thereby reducing the number of calculations required during property verification. This will result in a more efficient solution by avoiding redundant computations and focusing only on the regions of interest.

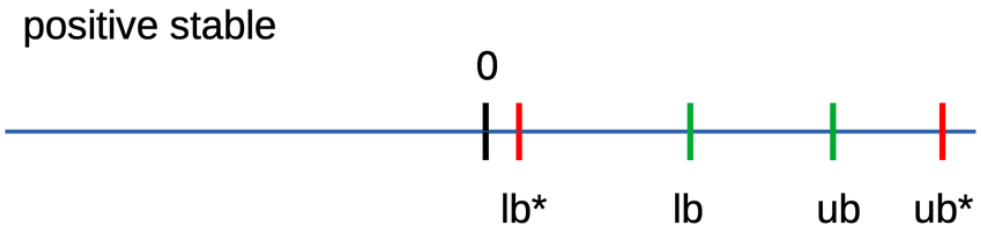
5.1.1 Real bounds and extimated bounds with bound propagation: all scenarios

The bound propagation performs a calculation that return the numeric bound for all the fully connected layers on the network. Through the upper and lower bound of a neuron we can know if it is stable or unstable. The problem is that the bound propagation implemented must deal with ReLU functions which are not linear and hence introduces an over-approximation. This implies:

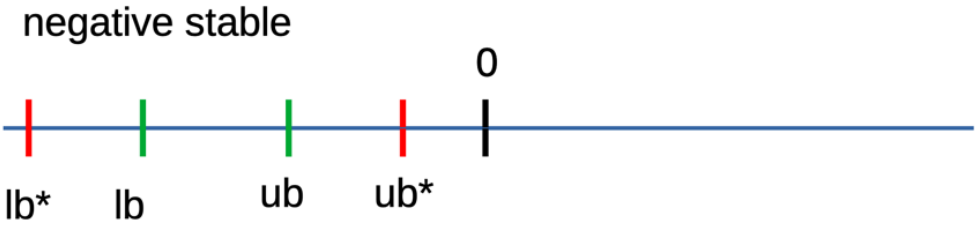
$$\begin{aligned} lb^* &< lb \\ ub^* &> ub \end{aligned} \tag{5.1}$$

where lb and ub are the real lower and upper bounds of a node, while $lb*$ and $ub*$ are the overestimated ones.
And this over-extimation grows up with the number of ReLU layers.
This over-extimation leads to different scenarios:

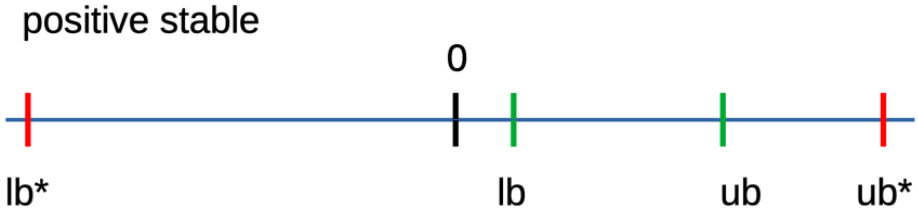
- if the over-extimated lower bound of a node $lb* > 0$, knowing that $lb > lb*$ then the neuron is for sure positive stable



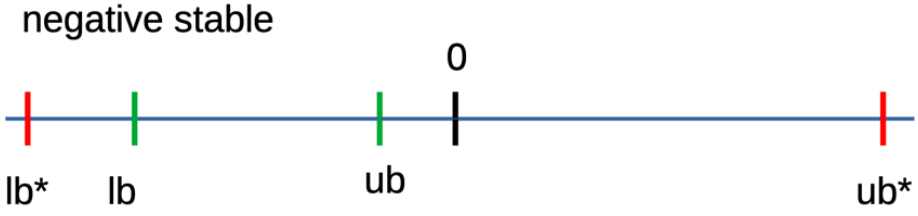
- if the over-extimated upper bound of a node $ub* < 0$, knowing that $ub < ub*$ then the neuron for sure negative stable



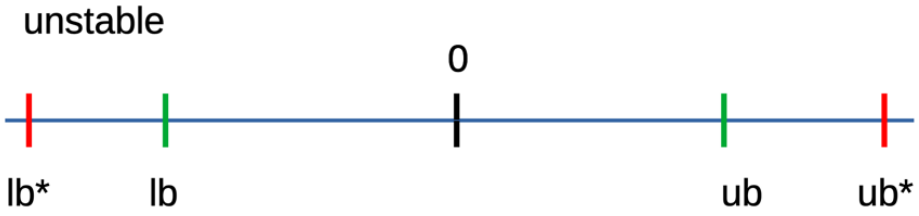
- if $lb* < 0$ and $ub* > 0$ knowing that $lb > lb*$ and $ub < ub*$ we cannot state if the neuron is stable or not. We can in turn distinguish three different cases:
  - When $lb* < 0$ and $lb > 0$, in this case the bound propagation suggest that the neuron is unstable while in fact is positive stable



  - When $ub* > 0$ and $ub < 0$, in this case the bound propagation suggests that the neuron is unstable while in fact is negative stable



– When $lb* < 0, ub* > 0$ and $lb < 0, ub > 0$, in this case the neuron is unstable as suggested by the bound propagation



This means that the bound propagation states correctly if a neuron is positive or negative stable. Differently if it states that a neuron is unstable, it isn't possible to get useful information from it.

5.1.2 Optimized version of PyNeVer with Bound Propagation

As discussed earlier, the goal of this thesis is to apply bound propagation to optimize the code of pyNever. As discussed in SubSection 5.1.1, through the precomputation of bound propagation, given a network and a property, it is possible to identify stable nodes without having to repeatedly call the GET_BOUNDS function of pyNever. Below, an improved version of pyNever will be presented, which utilizes precalculated bounds to avoid using the GET_BOUNDS function whenever possible. This enhances the efficiency of the program.

In Section 5.1.1, we thoroughly examined the concept of bound propagation through precomputation. During this phase, bounds are computed for each node in the network, considering the specified property. The resulting bounds are then stored for later use during program execution.

The enhanced version of pyNever we will present leverages these precalculated bounds to avoid unnecessary calls to the GET_BOUNDS function. Instead of invoking the function for each star and for each evaluated ReLU node, the program checks if the pre-computated bound states that the node is stable. If so, it is possible to avoid to call the GET_BOUNDS function. If, differently, the precomputated bound states that the node is unstalbe then it will be necessary to calculate the exact bounds. This optimization results in a significant improvement in overall performance.

The improved implementation of pyNever introduces an initial phase of bound precomputation, which takes time at program startup but greatly reduces execution time during the evaluation of subsequent nodes. Thanks to this strategy, a part of stable nodes can be efficiently identified without repeating costly computations for each evaluation.

The new version of pyNever, incorporating this improved implementation, offers a substantial advantage in terms of efficiency and execution speed. By minimizing calls to the GET_BOUNDS function through the use of precalculated bounds, the program can handle large and complex networks more efficiently.

In conclusion, applying bound propagation with precomputed bounds in pyNever represents a significant advancement in code optimization. By utilizing precalculated bounds, the program can rapidly and efficiently identify stable nodes, avoiding redundant calculations and reducing overall execution time. This implementation offers a considerable advantage in the efficiency and usability of pyNever in various application contexts.

5.1.3 Pynever Complete Verification Example

The Fig 5.2 represents a NN composed by:s

- Input: the NN has a two dimensional input
- FC1: a fully connected layer 2x4
- A ReLU activation function is applied to the FC1
- FC2: a fully connected layer 4x4
- A ReLU activation function is applied to the FC2
- A FC3 4x2. The output is bidimensional

In Fig 5.2 it is represented a NN without a pre-applied bound propagation. The grey nodes which belongs to the FC layers are nodes whose stability or instability is not knows. Differently, in Fig 5.1, it is pre-applied a bound propagation. It is known a priori if green nodes are positive or negative stable, instead of grey nodes that might be stable or unstable. Below is explained which are the differences and the advantages between the PyNeVer algorithm with bound propagation applied and the "normal" one:

- bp-case: PyNeVer algorithm with bound propagation applied
- non-bp-case: PyNeVer algorithm without bound propagation applied.

The PyNeVer algorithm is sequential respect to the layers of the NN, this means that the verification process iterates from the first layer till the last one.

Figure 5.1 non-bp-case

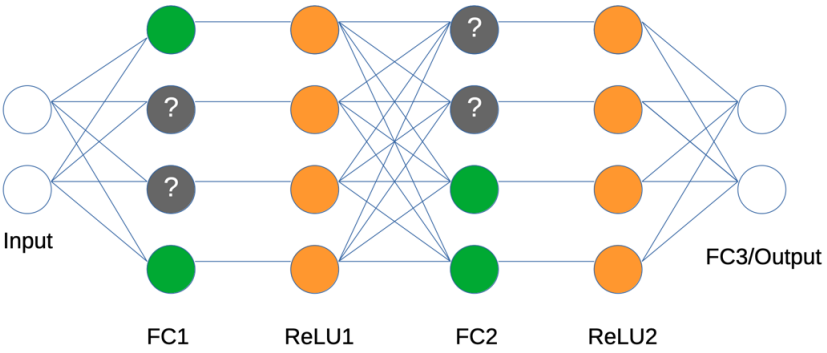


Figure 5.2 bp-case

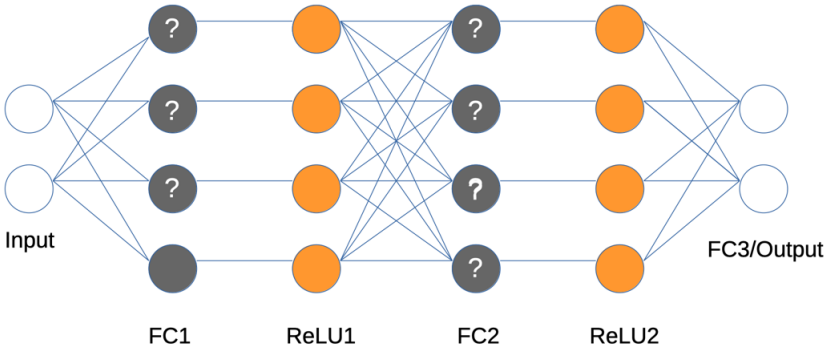


Figure 5.3 Neural network composed by three FC and two ReLU layers

1. FC1: the hyper-rectangle star defined by the input property is propagated through the FC1. For PyNeVer algorithm the number of stars in input in a fully connected layer undergo a transformation (translation or rotation) but don't change in number. In our example we have one star in input in the FC1 layer and so we'll have one star in output. **The two algorithm versions behave in the same way for FC layers.**

2. ReLU1: for computing this layer PyNeVer calls COMPUTE_LAYER function that, in our example, receives in input 1 star. COMPUTE_LAYER function calls the function COMPUTE_RELU one time for each dimension (the $i-th$ ReLU node corresponds to the $i-th$ dimension).
   For both the two algorithms, the COMPUTE_RELU bounds is called 4 times. The difference between the two algorithms is that the COMPUTE_RELU in the non-bp-case must call the GET_BOUNDS function for all stars along the j-th dimension while in the bp-case the GET_BOUNDS is called only for the nodes we don't know if they are stable or unstable (the grey ones in fig 5.2). In this case the the bp-case the GET_BOUNDS function is called 2 times while in the nn-bp-case it is called 4 times. The max number of output stars is 2^{n_1} where n_1 is the number of nodes in a generic ReLU layer for each star in input.

3. FC2: all the input stars undergo a transformation, but the number remains invaried.

4. ReLU2: the COMPUTER_LAYER function is called and receives a set of stars as input . For each star is called the COMPUTE_RELU and for each call are returned at max 2_2^n stars. So in the end, the max number of stars returned to COMPUTE_LAYER function is $2^{n_1} * 2^{n_2}$. As for the ReLU1 layer, in the bp-case the GET_BOUNDS function is not called for stars processed along the dimensions that correspond to the green nodes.

5. FC3: is the output layer and behaves as the previous fully connected layers.

The GET_BOUNDS function is necessary to understand if a star at which is applied a ReLU function along the j-th dimension must be zeroes along that dimension if it has $ub_j <= 0$, or directly propagated if $lb_j >= 0$ or must be splitted along the j-th dim if $ub_j >= 0$ and $lw_j <= 0$.
The GET_BOUNDS function is function that solves an optimization problem and it is quite expensive. The idea to make more efficient the algorithm is to apply a bound propagation algorithm before running the verification process. In this way it is possible to know a priori which neurons are stable .

5.1.4 Bound Propagation Pseudocode

Algorithm 2 Abstraction of the ReLU activation function with Bound Propagation.

function COMPUTE_LAYER(*input* = $[\Theta_1, \ldots, \Theta_N]$, *refine* = $[r_1, \ldots, r_n]$, ***lower_bounds: list***, ***upper_bounds: list***)
  output = [ ]
  for $i = 1 : N$ **do**
    stars = $[\Theta_i]$
    for $j = 1 : n$ **do**
      stars = COMPUTE_RELU(*stars*, j, *refine*$[j]$, n, ***lower_bounds[i]***, ***upper_bounds[i]***)
    APPEND(*output*, *stars*)
  return *output*
function COMPUTE_RELU(*input* = $[\Gamma_1, \ldots, \Gamma_M]$, j, *level*, n, ***lower_bound***, ***upper_bound***)
  output = [ ]
  for $k = 1 : M$ **do**
    is_positive_stable = False
    is_negative_stable = False
    if *lower_bound* ≥ 0 **then**
      is_positive_stable = True
    else if *upper_bound* ≤ 0 **then**
      is_negative_stable = True
    else
      (lb_j, ub_j) = GET_BOUNDS(*input*$[k]$, j)
    $M = [e_1 \ \ldots \ e_{j-1} \ 0 \ e_{j+1} \ \ldots \ e_n]$
    if *is_positive_stable* **then**
      S = *input*$[k]$
    else if *is_negative_stable* **then**
      $S = M$ * *input*$[k]$
    else if $lb_j \geq 0$ **then**
      S = *input*$[k]$
    else if $ub_j \leq 0$ **then**
      $S = M$ * *input*$[k]$
    else
      if *level* > 0 **then**
        Θ_{low} = *input*$[k] \wedge z[j] < 0$; Θ_{upp} = *input*$[k] \wedge z[j] \geq 0$
        $S = [M * \Theta_{low}, \Theta_{upp}]$
      else
        $(c, V, Cx \leq d)$ = *input*$[j]$
        $C_1 = [0 \ 0 \ \ldots \ -1] \in \mathbb{R}^{1 \times m+1}$, $d_1 = 0$
        $C_2 = [V[j,:] \ -1] \in \mathbb{R}^{1 \times m+1}$, $d_2 = -c_k[j]$
        $C_3 = [\frac{-ub_j}{ub_j - lb_j} \cdot V[j,:] \ -1] \in \mathbb{R}^{1 \times m+1}$, $d_3 = \frac{ub_j}{ub_j - lb_j}(c[j] - lb_j)$
        $C_0 = [C \ 0^{m \times 1}]$, $d_0 = d$
        $\hat{C} = [C_0; \ C_1; \ C_2; \ C_3]$, $\hat{d} = [d_0; \ d_1; \ d_2; \ d_3]$
        $\hat{V} = MV$, $\hat{V} = [\hat{V} \ e_j]$
        $S = (Mc, \hat{V}, \hat{C}\hat{x} \leq \hat{d})$
    APPEND(*output*, S)
  return *output*

Algorithm 3 Bound Propagation Algorithm

```
1: function ComputeBounds(network, property)
2:     ▷ the PropertyFormatConverter takes in input a PyNeVer property and return and return an HyperRectangleBounds
3:     property_converter = PropertyFormatConverter(property)
4:     input_hyper_rect = property_converter.get_vectors()
5:     layers = get_abstract_network(network)
6:     input_size = input_hyper_rect.get_size()
7:     lower = LinearFunctions(np.identity(input_size), np.zeros(input_size))
8:     upper = LinearFunctions(np.identity(input_size), np.zeros(input_size))
9:     input_bounds = SymbolicLinearBounds(lower, upper)
10:    numeric_preactivation_bounds = dict()
11:    numeric_postactivation_bounds = OrderedDict()
12:    symbolic_bounds = dict()
13:    current_input_bounds = input_bounds
14:    for i in 0 to len(layers) − 1 do
15:        if layers[i] is instance of ReLU then
16:            symbolic_activation_output_bounds = compute_relu_output_bounds(symbolic_dense_output_boun
17:            postactivation_bounds = HyperRectangleBounds(np.maximum(preactivation_bounds.get_lower(),
18:        else if layers[i] is instance of FullyConnected then
19:            symbolic_dense_output_bounds = compute_dense_output_bounds(layers[i], current_input_bounds)
20:            preactivation_bounds = symbolic_dense_output_bounds.to_hyper_rectangle_bounds(input_hyper_
21:            symbolic_activation_output_bounds = symbolic_dense_output_bounds
22:            postactivation_bounds = HyperRectangleBounds(preactivation_bounds.get_lower(), preactivation_
23:        else
24:            raise Exception
25:        symbolic_bounds[layers[i].identifier] = (symbolic_dense_output_bounds, symbolic_activation_output_b
26:        numeric_preactivation_bounds[layers[i].identifier] = preactivation_bounds
27:        numeric_postactivation_bounds[layers[i].identifier] = postactivation_bounds
28:        current_input_bounds = symbolic_activation_output_bounds
29:    return numeric_postactivation_bounds
```

Algorithm 4 Class HyperRectangleBounds

```
1: procedure HyperRectangleBounds
2:     public lower : list
3:     public upper : list
4: procedure Constructor
5:     lower = initialize
6:     upper = initialize
```