Counter-Example Guided Abstract Refinement for Verification of Neural Networks

Stefano Demarchi[1,*], Dario Guidotti[2]

[1]DIBRIS, Università degli Studi di Genova, Viale Causa 13, 16145, Genova, IT
[2]Dipartimento di Scienze Umanistiche e Sociali, Università degli Studi di Sassari, Via Roma 151, 07100, Sassari, IT

Abstract

In the last few decades, the employment of machine learning (ML) models has been increasingly common in the Artificial Intelligence community, with a particular focus on neural networks (NNs). However, even though they are widely adopted, the lack of formal guarantees on their behavior still restrain their use in safety-critical applications, such as avionics and self-driving vehicles. Formal Verification has been proposed to tackle the reliability issues of NNs, but its complexity and the sheer size of the models of interest have been proven to be hard challenges. In this paper we present an enhancement of our verification algorithm based on counter-example guided abstraction refinement (CEGAR) and show how it performs with respect to other approximate star-based methods.

Keywords

Safety and Reliability, Neural Networks, Formal Methods

1. Introduction

Adoption and successful application of deep neural networks (DNNs) in various domains across computer science have made them one of the most popular machine-learned models to date — see, e.g., [1] on image classification, [2] on speech recognition, and [3] 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 applications is not yet hassle-free. From their well-known sensitivity to *adversarial perturbations* [4, 5], 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., [6] for a catalog — the need for tools to analyze and possibly repair DNNs is strong.

As witnessed by an extensive survey [7] 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, as well as tools implementing them [8, 9, 10, 11, 12, 13, 14]. 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

CPSWS 2022: Cyber-Physical Systems Workshop, September 19, 2022, Pula, IT
*Corresponding author.
stefano.demarchi@edu.unige.it (S. Demarchi); dguidotti@uniss.it (D. Guidotti)
0000-0003-4532-4933 (S. Demarchi); 0000-0001-8284-5266 (D. Guidotti)

difficult to use for the non-initiated. Most of them work only for feed-forward fully-connected neural networks with ReLU activation functions, with some of them featuring verification algorithms for convolutional neural networks with different kinds of activation functions. Aside from verification tools, a great deal of research involves the analysis on how to modify networks to make them compliant to specifications, i.e., *repair* [15, 16, 17, 18] and how to learn networks which respect specific constraints on their behavior, i.e., *safe learning* [19, 20, 21, 22, 23].

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 [24], the first tool for automated learning, analysis and repair of neural networks. NeVer2 relies on the pyNeVer API [25] and a first description of the system is available in [26], where the verification capabilities were provided by external tools like Marabou [11], ERAN [12] and MIPVerify [13]. The version of NeVer2 corresponding to this work is available online [27] under the Commons Clause (GNU GPL v3.0) license.

The remainder of the paper is structured as follows: after an overview of the main concepts in Section 2, we introduce the abstraction definitions and algorithms in Section 3. Our contribution, i.e., the refinement step for the CEGAR [28] algorithm is presented in Section 4 and we discuss its applicability in Sections 5 and 6. While in our experimental evaluation the CEGAR algorithm did not show increased performances with respect to the original one, we believe that our approach may be further enhanced leveraging the insight obtained by the experimental evaluation.

2. Preliminaries

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 the most common activation function $\sigma : \mathbb{R} \to \mathbb{R}$: the *ReLU* function defined as $\sigma(r) = max(0, r)$. 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 ReLU layers offer universal approximation capabilities [29].

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 $I \subset \mathbb{R}^n$, i.e., 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* $d(x, y) = ||x - y||$. 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 and (ii) ReLU is a piecewise affine transformation of its input. 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

$$\forall x \in \Pi \rightarrow \nu(x) \in \Sigma. \tag{1}$$

While this query cannot express some problems regarding neural networks, e.g., invertibility or equivalence [6], it can easily express, for example, the general problem of testing robustness against *adversarial perturbations* [4]. For example, given a network $\nu : I \rightarrow O$ 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 (1) by letting $\Pi = \{\hat{X}_j\}$ and $\Sigma = \{Y_j\}$ with $Y_j = \{y \in O \mid y_j \geq y_i + \epsilon, \forall i \in [1, n] \wedge i \neq j, \epsilon > 0\}$. Analogously, in a regression task we may ask that points that are sufficiently 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 $\hat{Y} = \{\hat{y} \in O \mid \exists x.(x \in \hat{X} \wedge d(\hat{y}, \nu(x)) \leq \epsilon)\}$ to obtain $\Pi = \{\hat{X}\}$ and $\Sigma = \{\hat{Y}\}$.

3. Abstract methods

To enable algorithmic verification of neural networks, we consider a subclass of *generalized star sets*, introduced in [30] and defined as follows — the notation is adapted from [31].

Definition 1. *(Generalized star set)* Given a *basis matrix* $V \in \mathbb{R}^{n \times m}$ obtained arranging a set of m *basis vectors* $\{v_1, \ldots v_m\}$ in columns, a point $c \in \mathbb{R}^n$ called *center* and a *predicate* $R : \mathbb{R}^m \rightarrow \{\top, \bot\}$, a generalized star set is a tuple $\Theta = (c, V, R)$. The set of points represented by the generalized star set is given by

$$[\![\Theta]\!] \equiv \{z \in \mathbb{R}^n \mid z = Vx + c \text{ such that } R(x_1, \ldots, x_m) = \top\} \tag{2}$$

In the following we denote $[\![\Theta]\!]$ also as Θ. We consider only star sets 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; we further require that the set $Y = \{y \in \mathbb{R}^m \mid Cy \leq d\}$ is bounded. In order to ensure the convexity, we consider input sets which are either already convex or approximated by one. In this case such sets are polytopes in $\mathbb{R}^n$ whose set we represent as $\langle \mathbb{R}^n \rangle$.

[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.

Algorithm 1 Abstraction of the ReLU activation function.

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

function COMPUTE_RELU(*input* = $[\Gamma_1, \ldots, \Gamma_M]$, j, *level*, n)
  $output$ = [ ]
  for $k = 1 : M$ **do**
    (lb_j, ub_j) = GET_BOUNDS(*input*$[k]$, j)
    $M = [e_1 \ldots e_{j-1}\ 0\ e_{j+1} \ldots e_n]$
    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 $output$

Definition 2. *(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.

Algorithm 1 defines the abstract mapping of a functional layer with n ReLU activation functions. 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

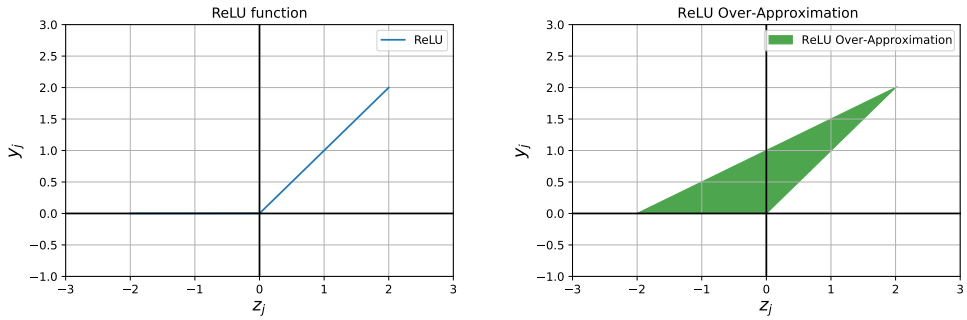

Figure 1: Graphical representation of the ReLU function (left) and its over-approximation (right) considering a single variable. The bounds for the triangle approximation are the bounds of the star along the variable considered.

output of function COMPUTE_LAYER is 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 Rn arranged in columns, with the exception of the j-th dimension which is zeroed.

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. Notice that, 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.

If the refinement level is 0, then the ReLU is abstracted using the over-approximation proposed in [31] and depicted in Figure 1. This approach is much less conservative than others, i.e., based on zonotopes or abstract domains, and provides a tighter abstraction. The computation of the resulting star is carried out from line 21 to line 25. Intuitively, given the predicates of the input star $Cx \leq d$, the matrix C and the vector d are modified to constrain the output star within the points inside the triangle defining the abstraction, given the points of the input star. 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.

In [25] we propose 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: using an heuristic detailed in [25], NeVer2 tries to concretize the least number of stars that enable proving the property without blowing the computation time. We call *mixed* the analysis of NeVer2 in this case.

4. Counter-example guided abstraction refinement

Our algorithm can be used to compute the complete or over-approximate reachable set of the neural network of interest. Once the reachable set has been computed, the property of interest can be verified by computing the intersection between the negation of such property and the reachable set (which we call *reachable counter set*). If such intersection is the empty set, then the network is compliant with the property of interest; otherwise, if the reachable set is complete, we have shown that the network is unsafe. However, if the reachable set is over-approximated, the concrete network may satisfy the property, and the over-approximation may be too coarse. In both cases in which the reachable counter set is not the empty set, we are interested in extracting concrete input points corresponding to the output contained in the reachable counter set. In particular, when we have a complete counter reachable set we can leverage the following theorem:

Theorem 1. *Let ν be a feed-forward neural network, $\Theta = (c, V, R)$ be a star input set, $\nu(\Theta) = \bigcup_{i=1}^{k} \Theta_i$, $\Theta_i = (c_i, V_i, R_i)$ be the reachable set of the neural network and* S *be a safety specification. Denote $\overline{\Theta}_i = \Theta_i \cap \neg \mathbf{S} = (c_i, V_i, \overline{P}_i)$, $i = 1, ..., k$. The neural network is safe if and only if $\overline{P}_i = 0$ for all i. If the neural network violates its safety property then the complete counter input set containing all possible inputs in the input set that lead the neural network to unsafe states is $\mathbf{C} = \bigcup_{i=1}^{k} (c, V, \overline{P}_i), \overline{P}_i \neq 0$.*

For the proof of Theorem 1 we refer to [32]. Using Theorem 1 we can easily compute the complete counter input set, so the problem of extracting concrete input points becomes the problem of extracting points from a star-set which in itself can be considered as extracting points from a single star. To do this, we consider the problem of extracting points from the predicate of the star, which, under our pre-conditions, is always a polytope. We will then apply to the points of the predicate (α) the affine transformation $x = c + V\alpha$ to obtain a corresponding point of the star of interest. To extract the point from the polytope defined by the predicate, we leverage the hit and run sampler [33]. It should be noted that while the hit and run algorithm produces an approximation of a uniform distribution for the α of the predicate, the application of the affine transformation needed for the transformation to the point of the star skews such distribution. A possible solution to this issue is to transform the predicate to its V-representation, apply the affine transformation directly to the polytope, return to the H-representation and apply the hit and run sampler. However, for our aims, the skew of the distribution is not that relevant.

Therefore, at least at this time, we do not need to transform between the two representations, which is computationally expensive.

The problem is different when we are working with the over-approximate reachable counter set: in this case, we do not have a way to compute the counter input set since the addition of the new variables needed for the over-approximation to the predicate of the star invalidates Theorem 1. Therefore an alternative solution is needed to compute inputs that allegedly are not compliant with the property of interest. We define the *abstract counter output set* (ACOS) as the intersection between the abstract reachable set and the negation of the property S. Our algorithm extracts a point from the ACOS using hit and run sampling and then searches for the corresponding input point. Formally the search problem of the corresponding input point can be defined as:

Definition 3. *Given a reference output point $\hat{y}$, a starting input point x and a feed forward neural network ν we can define the search problem for the point $\hat{x}$ which satisfies $\nu(\hat{x}) = \hat{y}$ as the following minimization problem:*

$$\hat{x} = \min_{x} ||\hat{y} - \nu(x)||_2$$

However, the non-convexity and non-linearity of the function make the minimization problem not easily solvable: the non-convexity and the presence of local minima make it extremely difficult to apply gradient descent. Consequently, we developed a simple search-by-sampling algorithm which, given a starting point in the input space, generates a “cloud" of points using a normal distribution with the starting point as center and a given variance. Such points are then compared, and the one whose corresponding output is nearest to the desired one is selected as the center for another step of the algorithm. The search terminates when the euclidean distance between the output found and the one we are searching for is less than a given threshold or when a given number of steps is exceeded. If the algorithm finds an input point in the concrete input set and whose corresponding output is in the ACOS, we have found a concrete counter-example, and the network is proven unsafe. Otherwise, the point found is a point whose corresponding output is reasonably close to the ACOS and can be leveraged for our refinement.

Once an adequate sample is found, we can use it to guide our refinement. The idea behind the refinement algorithm is to rank the approximation error for each neuron by computing the triangle areas of the approximate method — see, Section 3 — and enhance it with a measure of the relevance of the neurons with respect to the sample found. To compute the relevance, we leveraged the layer-wise relevance propagation algorithm [34] which, while traditionally used by the explainability community for classification models, can provide an adequate relevance measure even for regression tasks. It should be noted that our implementation of the algorithm support, at present, only fully-connected layers and ReLU activation functions. For more details on layer-wise relevance propagation we refer to [35].

The refinement procedure is detailed in Algorithm 2. As the first thing, it needs to apply our verification methodology in its over-approximate form (line 2) to compute the over-approximate reachable counter set and the triangle areas. If the network is proven to be safe (line 3) then the verification algorithm terminates (line 4), otherwise we can search the counter-example as shown before (line 5 and 6). If we found a concrete counter-example then the network is proven to be unsafe and the procedure terminates (line 7 and 8), otherwise we use the spurious

Algorithm 2 CEGAR Algorithm.

1: **function** CEGAR_VERIFICATION(*input_set*, *unsafe_zone*, *network*)
2: *ref_levels* = $[0, ..., 0]$
3: $ACOS, areas, safe$ = STARSET_VER(*input_set*, *unsafe_zone*, *network*, *ref_levels*)
4: **if** IS_EMPTY(*ACOS*) **then**
5: **return** $ACOS, areas, True$

6: $output\_counter$ = GET_SAMPLE(*ACOS*)
7: $input\_counter$ = INPUT_SEARCH(*network*, *output_counter*)
8: **if** $input\_counter \in input\_set$ **then**
9: **return** $ACOS, areas, False$

10: $neuron\_relevances$ = COMPUTE_REL(*input_counter*, *network*)
11: ref_levels = COMPUTE_REF_LEVELS(*neuron_relevances*, *areas*)
12: **return** STARSET_VER(*input_set*, *unsafe_zone*, *network*, *ref_levels*)

counter-example to find the relevances of the neurons of the network (line 9). At this point, the relevances and the triangle areas can be used to evaluate the significance of each ReLU neuron of the network. Once a measure of the significance is computed for each neuron of each ReLU layer, we can choose a given number of neurons to refine for each layer (line 10), and we can change the refinement levels of Algorithm 1 as needed. Then our verification methodology is applied again using the new refinement levels (line 11).

5. Experimental evaluation

In this Section, we provide some empirical results about NeVer2[2]. Our experiments are focused on the verification task. For the comparison, we considered networks and properties from the ACAS Xu evaluation [36]. ACAS Xu is an airborne collision avoidance system based on DNNs whose purpose is to issue advisory commands to an autonomous vehicle (ownship) about evasive maneuvers to be performed if another vehicle (intruder) comes too close. In particular, we selected Property 3 and 4 since they could be easily expressed as a single verification query in our tool. In the words of [36], these safety properties "*deal with situations where the intruder is directly ahead of the ownship and state that the NN will never issue a COC (clear of conflict) advisory*". Considering the analysis in [36], each property can be assessed on 42 different networks depending on the choice of two parameters, i.e., the the previous advisory value and the time to loss of vertical separation. Among the networks available, we selected those for which our over-approximate analysis could not find a definitive answer, ending with a total of 9 networks. Notice that Property 3 and Property 4 are always satisfied in these networks.

In our experimental evaluation, we compare two different significance measures. *Product significance* (PS) computes, for each neuron, the value of the multiplication between its relevance

[2]All experiments ran on a laptop equipped with an Intel i7-8565 CPU (8 core at 1.8GHz) and 16 GB of memory with Ubuntu 20 operating system.

Table 1
Performances of NeVer2 on a subset of ACAS Xu networks. Columns PROPERTY and NETWORK report the property and the network considered, respectively. The other columns report the verification time (TIME) in seconds and result (VERIFIED) for MIXED, CEGAR-PS and CEGAR-mR analyses, respectively. Given the randomic nature of the counter-example generator, we report the average time and the number of results over 10 repetition of the experiment.

PROPERTY	NETWORK	MIXED		CEGAR-PS		CEGAR-mR	
		TIME	VERIFIED	TIME	VERIFIED	TIME	VERIFIED
# 3	1_1	13	T	10	3/10	9	9/10
	1_3	10	T	14	6/10	10	0/10
	2_3	7	T	10	9/10	7	6/10
	4_3	15	T	17	10/10	14	10/10
	5_1	6	T	11	10/10	9	10/10
# 4	1_1	11	T	10	0/10	9	0/10
	1_3	8	T	16	0/10	11	0/10
	3_2	12	T	12	10/10	12	10/10
	4_2	12	T	11	10/10	12	10/10

and the area of the triangle abstraction, whereas *mixed-R* (mR) uses the relevances as coefficients for the ranking used in the standard mixed methodology. The PS refinement (*CEGAR-PS*) selects six neurons in the whole network to refine, while the mR refinement (*CEGAR-mR*) refines one single neuron for each layer. In Table 1 we show the performance of the two versions of the refinement algorithm, and we compare them with our mixed abstraction methodology. Note that, by design, the number of neurons refined is the same for every methodology: six in the whole network. The difference between the three algorithms is which neurons are selected and how.

As can be seen, the performances of the two refinement algorithms are comparable; however, they seem to be less effective than our mixed methodology and CEGAR-PS seems to be slightly more accurate than CEGAR-mR at the cost of a small increase in the time needed to solve the query. We believe that the difference in performance is mainly attributable to the fact that, while the measurements of relevance we used are valid, they do not capture how the coarseness of the abstraction changes dynamically when a particular neuron is refined. On the contrary, the mixed methodology chooses in each layer the neuron to refine based on the values of the areas of the triangles given the previous layer output. As a consequence, the choice of which neurons to refine is guided by the coarseness of the abstraction *after* the refinement is already applied in the previous layers.

6. Conclusions

In this paper we tried to enhance our verification methodology, leveraging a novel counter-example guided abstraction refinement algorithm. The refinement algorithm computes the over-approximate reachable set, searches for a spurious counterexample and then uses it to

compute the relevance measures for all neurons in the network. Then, a specific heuristic is used to select which neurons must be refined in computing the new reachable set. Although our experimental results showed that our refinement methodology performances do not present an enhancement with respect to the ones of the algorithms presented in Section 3, we believe that with further investigations we will be able to further enhance this technique.

References

[1] Y. Taigman, M. Yang, M. Ranzato, L. Wolf, Deepface: Closing the gap to human-level performance in face verification, in: 2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2014, Columbus, OH, USA, June 23-28, 2014, 2014, pp. 1701–1708.

[2] D. Yu, G. E. Hinton, N. Morgan, J. Chien, S. Sagayama, Introduction to the special section on deep learning for speech and language processing, IEEE Trans. Audio, Speech & Language Processing 20 (2012) 4–6.

[3] Y. LeCun, Y. Bengio, G. E. Hinton, Deep learning, Nature 521 (2015) 436–444.

[4] I. J. Goodfellow, J. Shlens, C. Szegedy, Explaining and harnessing adversarial examples, in: 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings, 2015.

[5] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. J. Goodfellow, R. Fergus, Intriguing properties of neural networks, in: 2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings, 2014.

[6] F. Leofante, N. Narodytska, L. Pulina, A. Tacchella, Automated verification of neural networks: Advances, challenges and perspectives, CoRR abs/1805.09938 (2018).

[7] X. Huang, D. Kroening, M. Kwiatkowska, W. Ruan, Y. Sun, E. Thamo, M. Wu, X. Yi, Safety and trustworthiness of deep neural networks: A survey, arXiv preprint arXiv:1812.08342 (2018).

[8] H. Tran, X. Yang, D. M. Lopez, P. Musau, L. V. Nguyen, W. Xiang, S. Bak, T. T. Johnson, NNV: the neural network verification tool for deep neural networks and learning-enabled cyber-physical systems, CoRR abs/2004.05519 (2020).

[9] M. Akintunde, A. Lomuscio, L. Maganti, E. Pirovano, Reachability analysis for neural agent-environment systems, in: Principles of Knowledge Representation and Reasoning: Proceedings of the Sixteenth International Conference, KR 2018, Tempe, Arizona, 30 October - 2 November 2018, AAAI Press, 2018, pp. 184–193.

[10] S. Dutta, X. Chen, S. Jha, S. Sankaranarayanan, A. Tiwari, Sherlock - A tool for verification of neural network feedback systems: demo abstract, in: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2019, Montreal, QC, Canada, April 16-18, 2019, 2019, pp. 262–263.

[11] G. Katz, D. A. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljic, D. L. Dill, M. J. Kochenderfer, C. W. Barrett, The marabou framework for verification and analysis of deep neural networks, in: Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I, 2019, pp. 443–452.

[12] G. Singh, T. Gehr, M. Püschel, M. T. Vechev, Boosting robustness certification of neural networks, in: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019, 2019.
[13] V. Tjeng, K. Y. Xiao, R. Tedrake, Evaluating robustness of neural networks with mixed integer programming, in: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019, 2019.
[14] S. Wang, K. Pei, J. Whitehouse, J. Yang, S. Jana, Efficient formal safety analysis of neural networks, in: Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada, 2018, pp. 6369–6379.
[15] D. Guidotti, F. Leofante, L. Pulina, A. Tacchella, Verification and repair of neural networks: a progress report on convolutional models, in: International Conference of the Italian Association for Artificial Intelligence, Springer, 2019, pp. 405–417.
[16] P. Kouvaros, T. Kyono, F. Leofante, A. Lomuscio, D. Margineantu, D. Osipychev, Y. Zheng, Formal analysis of neural network-based systems in the aircraft domain, in: International Symposium on Formal Methods, Springer, 2021, pp. 730–740.
[17] M. Sotoudeh, A. V. Thakur, Provable repair of deep neural networks, in: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, 2021, pp. 588–603.
[18] P. Henriksen, F. Leofante, A. Lomuscio, Repairing misclassifications in neural networks using limited data, in: Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing, 2022, pp. 1031–1038.
[19] J. Cohen, E. Rosenfeld, Z. Kolter, Certified adversarial robustness via randomized smoothing, in: International Conference on Machine Learning, PMLR, 2019, pp. 1310–1320.
[20] Z. Hu, X. Ma, Z. Liu, E. Hovy, E. Xing, Harnessing deep neural networks with logic rules, arXiv preprint arXiv:1603.06318 (2016).
[21] Z. Eaton-Rosen, F. Bragman, S. Bisdas, S. Ourselin, M. J. Cardoso, Towards safe deep learning: accurately quantifying biomarker uncertainty in neural network predictions, in: International Conference on Medical Image Computing and Computer-Assisted Intervention, Springer, 2018, pp. 691–699.
[22] E. Giunchiglia, T. Lukasiewicz, Multi-label classification neural networks with hard logical constraints, Journal of Artificial Intelligence Research 72 (2021) 759–818.
[23] E. Giunchiglia, M. C. Stoian, T. Lukasiewicz, Deep learning with logical constraints, arXiv preprint arXiv:2205.00523 (2022).
[24] L. Pulina, A. Tacchella, Never: a tool for artificial neural networks verification, Annals of Mathematics and Artificial Intelligence 62 (2011) 403–425.
[25] D. Guidotti, L. Pulina, A. Tacchella, pyNeVer: A framework for learning and verification of neural networks, in: International Symposium on Automated Technology for Verification and Analysis, Springer, 2021, pp. 357–363.
[26] D. Guidotti, L. Pulina, A. Tacchella, Never 2.0: Learning, verification and repair of deep neural networks, arXiv preprint arXiv:2011.09933 (2020).
[27] D. Guidotti, A. Tacchella, L. Pulina, S. Demarchi, NeVer 2.0, 2022. URL: https://github.com/NeVerTools/NeVer2.
[28] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Counterexample-guided abstraction

refinement, in: Proceedings of CAV 2000, volume 1855 of *Lecture Notes in Computer Science*, 2000, pp. 154–169.

[29] K. Hornik, M. Stinchcombe, H. White, Multilayer feedforward networks are universal approximators., Neural networks 2 (1989) 359–366.

[30] S. Bak, P. S. Duggirala, Simulation-equivalent reachability of large linear systems with inputs, in: International Conference on Computer Aided Verification, Springer, 2017, pp. 401–420.

[31] H.-D. Tran, D. M. Lopez, P. Musau, X. Yang, L. V. Nguyen, W. Xiang, T. T. Johnson, Star-based reachability analysis of deep neural networks, in: International Symposium on Formal Methods, Springer, 2019, pp. 670–686.

[32] D. Tran, Verification of Learning-enabled Cyber-Physical Systems, Ph.D. thesis, Vanderbilt University, 2020.

[33] R. L. Smith, The hit-and-run sampler: A globally reaching markov chain sampler for generating arbitrary multivariate distributions, in: J. M. Charnes, D. J. Morrice, D. T. Brunner, J. J. Swain (Eds.), Proceedings of the 28th conference on Winter simulation, WSC 1996, Coronado, CA, USA, December 8-11, 1996, IEEE Computer Society, 1996, pp. 260–264.

[34] W. Samek, G. Montavon, A. Binder, S. Lapuschkin, K. Müller, Interpreting the predictions of complex ML models by layer-wise relevance propagation, CoRR abs/1611.08191 (2016).

[35] G. Montavon, A. Binder, S. Lapuschkin, W. Samek, K. Müller, Layer-wise relevance propagation: An overview, in: Explainable AI: Interpreting, Explaining and Visualizing Deep Learning, volume 11700 of *Lecture Notes in Computer Science*, Springer, 2019, pp. 193–209.

[36] G. Katz, C. W. Barrett, D. L. Dill, K. Julian, M. J. Kochenderfer, Reluplex: An efficient SMT solver for verifying deep neural networks, in: Proceedings of CAV 2017, volume 10426 of *Lecture Notes in Computer Science*, 2017, pp. 97–117.