pyNeVer: a Framework for Learning and Verification of Neural Networks

Dario Guidotti[1], Luca Pulina[2], and Armando Tacchella[1,✉]

[1] University of Genoa, Italy
[2] University of Sassari, Italy
dario.guidotti@edu.unige.it,
lpulina@uniss.it, armando.tacchella@unige.it

1 Summary

Automated verification of neural networks (NNs) was first proposed in [1] and it is an established research topic with several contributions to date — see, e.g., [2]. The taxonomy proposed in [2] suggests a division among verification tools providing deterministic guarantees, e.g., Marabou [3], and those providing sound approximations, e.g., ERAN [4] and NNV [5]. pyNeVer borrows basic techniques from [5] and casts them into an abstraction approach inspired by [4]; like ERAN and NNV, it features complete verification methods, but it features a distinctive abstraction mechanism. Networks comprising layers of affine transformations and layers of activation functions such as Rectified Linear Units (ReLUs) and sigmoids are abstracted to mappings between polytopes represented as generalized star sets [6]; the main novelty is that the abstraction level of each layer can be controlled down to a single neuron to support various refinement policies. Additionally, pyNeVer can also load popular datasets and NN models in ONNX [7] and PyTorch [8] formats, and supports training of NNs carried out transparently through PyTorch. Additionally, NNs can be manipulated through network slimming and weight pruning to ease verification — see [9]. Here we focus on verification with pyNeVer and provide a brief experimental account. pyNeVer sources, documentation and examples are accessible at

`https://github.com/NeVerTools/pyNeVer`

In the remainder of this section, we briefly introduce some basic definitions and notation used in the paper.

Star sets. To represent polytopes and define abstract computations we consider a subclass of *generalized star sets*, introduced in [6] and defined as follows — the notation is adapted from [10]. 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* $R : \mathbb{R}^m \to \{\top, \bot\}$, a generalized star set is a tuple $\Theta = (c, V, R)$ yielding the set of points:

$$[\![\Theta]\!] \equiv \{z \in \mathbb{R}^n \mid z = Vx + c \text{ such that } R(x_1, \ldots, x_m) = \top\}. \quad (1)$$

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. We refer to generalized star sets obeying our restrictions simply as *stars*, and it is easy to show that such sets are polytopes in $\mathbb{R}^n$ whose set we represent as $\langle \mathbb{R}^n \rangle$. Given a star $\Theta = (c, V, R)$ and an affine mapping $f : R^n \rightarrow R^m$ with $f = Ax + b$, the affine mapping of the star is defined as $f(\Theta) = (\hat{c}, \hat{V}, R)$ where $\hat{c} = Ac + b$ and $\hat{V} = AV$. Notice that, if $\Theta \in \langle \mathbb{R}^n \rangle$ then also $f(\Theta) \in \langle \mathbb{R}^m \rangle$, i.e., the affine transformation of a polytope is still a polytope.

Neural networks. Given a finite number p of functions $f_1 : \mathbb{R}^n \rightarrow \mathbb{R}^{n_1}, \ldots, f_p : \mathbb{R}^{n_{p-1}} \rightarrow \mathbb{R}^m$ — also called *layers* — we define a *feed forward neural network* as a function $\nu : \mathbb{R}^n \rightarrow \mathbb{R}^m$ 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*. Given $x \in \mathbb{R}^n$, we consider two types of layers: the mapping $f(x) = Ax + b$ with $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$ is an *affine layer* implementing the linear mapping $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$; the mapping $f(x) = (\sigma_1(x_1), \ldots, \sigma_n(x_n))$ is a *functional layer* $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ 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 to each component of the vector x. We consider two kinds of activation functions $\sigma : \mathbb{R} \rightarrow \mathbb{R}$ that find widespread adoption: the *ReLU* function defined as $\sigma(r) = max(0, r)$, and the *logistic* function — of the family of *sigmoids* — defined as $\sigma(r) = \frac{1}{1+e^{-r}}$. For a neural network $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$, 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 $\mathbb{R}^m$.

2 Abstraction algorithms

In Algorithm 1 we detail the abstract mapping of a ReLU node — abstraction of sigmoid nodes and affine transformations are also implemented. Let us assume that the concrete functional layer contains n activation functions. The function COMPUTE_LAYER takes as input an indexed list of N stars $\Theta_1, \ldots, \Theta_N$ representing an abstraction of the input 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 and the greater the level, the finer the abstraction grain becomes. In the case of ReLUs, all non-zero levels map to the same (precise) refinement, i.e., a piece-wise affine mapping. Notice that, since each neuron features its own refinement level, Algorithm 1 controls abstraction down to the single neuron as expected, enabling the computation of layers with mixed degrees of abstraction. The 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 is parallelized in the actual implementation. Given a single input star $\Theta_i \in \langle \mathbb{R}^n \rangle$, each of the n dimensions is

Algorithm 1 Abstraction of the ReLU activation function.

```
 1: function COMPUTE_LAYER($input = [\Theta_1, \dots, \Theta_N]$, $refine = [r_1, \dots, r_n]$)
 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($input = [\Gamma_1, \dots, \Gamma_K]$, $j$, $level$, $n$)
 9:     $output = [\,]$
10:     for $k = 1 : K$ do
11:         $(lb, ub)$ = GET_BOUNDS($input[k]$, $j$)
12:         $M = [e_1 \dots e_{j-1}\ 0^n\ e_{j+1} \dots e_n]$
13:         if $lb \geq 0$ then $S = input[k]$
14:         else if $ub \leq 0$ then $S = M * input[k]$
15:         else
16:             if $level > 0$ then
17:                 $\Theta_{low} = input[k] \wedge z_j < 0$;  $\Theta_{upp} = input[k] \wedge z_j \geq 0$
18:                 $S = [M * \Theta_{low}, \Theta_{upp}]$
19:             else
20:                 $(c, V, Cx \leq d) = input[k]$
21:                 $C_1 = [0\ 0\ \dots\ -1] \in \mathbb{R}^{1 \times m+1}$, $d_1 = 0$
22:                 $C_2 = [V_j\ -1] \in \mathbb{R}^{1 \times m+1}$, $d_2 = -c_j$
23:                 $C_3 = [\frac{-ub}{ub-lb} V_j\ -1] \in \mathbb{R}^{1 \times m+1}$, $d_3 = \frac{ub}{ub-lb}(c_j - lb)$
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 $output$
```

processed in turn by the **for** loop starting at line 5 and involving the function COMPUTE_RELU. Notice that the stars obtained processing the j-th dimension are fed again to COMPUTE_RELU in order to process the $j+1$-th dimension. The function APPEND(p_1, p_2) (line 6) takes an indexed list p_1 and either an element or another indexed list p_2 and appends it to p_1. 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 a linear-programming problem — function GET_BOUNDS at line 11. Independently from the abstraction level, if $lb \geq 0$ then the ReLU acts as an identity function (line 13), whereas if $ub \leq 0$ then the j-th dimension is zeroed (line 14). The "asterisk" 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. When $lb < 0$ and $ub > 0$ we must consider the refinement level. For any non-zero level, the input star is "split" into two 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 analy-

PROPERTY	NET	COMPLETE		MIXED		OVERAPPROX	
		TIME	VERIFIED	TIME	VERIFIED	TIME	VERIFIED
# 3	1_1	460	T	25	T	2	F
	1_3	83	T	11	T	3	F
	2_3	33	T	9	T	2	F
	4_3	319	T	31	T	3	F
	5_1	44	T	10	T	2	F
# 4	1_1	143	T	11	F	3	F
	1_3	96	T	16	F	3	F
	3_2	67	T	20	T	3	F
	4_2	177	T	15	T	3	F

Table 1. Performances of pyNeVer on a subset of ACAS XU networks. Columns property and net report the property and the network considered, respectively. The other columns report the verification time (time) and the result of verification (verified) for complete, mixed and over-approximate analyses, respectively.

sis of pyNeVer in this case. Currently, pyNeVer does not attempt to merge stars. Therefore, in the complete analysis, the number of stars is worst-case exponential — see [5]. If the refinement level is 0, then the ReLU is abstracPted using the tightest polyhedral abstraction available, i.e. a triangle with vertices in $(lb, 0)$, $(0, 0)$ and (ub, ub). 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 the analysis at lines 21-25 is carried out throughout the network, assuming that the input star contains all potential input points, then the output star will be a (sound) over-approximation of the concrete output range: we call *over-approximate* the analysis of pyNeVer in this case. As we mentioned before, we can mix different levels of abstraction, down to the single neuron: we call *mixed* an analysis that adopts different levels of abstraction.

3 Experimental evaluation

In this section, we provide some empirical results about pyNeVer[3]. Our experiments are focused on the verification task, i.e., 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. In the first experiment, we compare the three different verification methodologies available in pyNeVer, namely complete, mixed and over-approximate analysis. In this experiment, the mixed strategy is implemented by refining a fixed amount of neurons in each layer. The results that we present are obtained refining at most a single neuron for each layer. Clearly, different refinement heuristics may yield different results, but a thorough experimentation of such heuristics is beyond the scope of this paper. Here, we just wish to show how combining concrete and over-approximate analysis, even with a very straightforward approach,

[3] 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

P	NET	ERAN$_{cp}$		ERAN$_{cz}$		ERAN$_{op}$		ERAN$_{oz}$		MARABOU		NNV$_c$		NNV$_o$	
		TIME	VER	TIME	VER	TIME	VER	TIME	VER	TIME	VER	TIME	VER	TIME	VER
# 3	1_1	139	T	73	T	105	F	65	F	7073	T	329	T	1	F
	1_3	9	T	–	–	9	T	36	F	3451	T	37	T	<1	F
	2_3	4	T	3	T	4	T	2	T	966	T	17	T	<1	F
	4_3	4	T	4	T	4	T	5	T	1452	T	112	T	<1	F
	5_1	7	T	3	T	8	T	4	T	763	T	17	T	<1	F
# 4	1_1	11	T	6	T	11	T	7	T	2401	T	141	T	1	F
	1_3	8	T	3	T	8	T	3	T	756	T	41	T	<1	F
	3_2	4	T	3	T	5	T	2	T	63	T	21	T	<1	F
	4_2	4	T	2	T	4	T	2	T	44	T	59	T	<1	F

Table 2. Performances of a pool of state-of-the-art tools on a subset of ACAS XU networks. The table is organized similarly to Table 1. In the results, "<1" indicates that the CPU time spent was less than 1 second, while a dash ("–") denotes that the tool exhausted the available memory.

may yield improvements in the overall verification time. For the comparison, we consider networks and properties from the ACAS Xu evaluation [11]. ACAS Xu is an airborne collision avoidance system based on NNs whose purpose is to issue advisory commands to an autonomous vehicle (ownship) about evasive maneuvers to be be performed in case another vehicle (intruder) comes too close. In particular, we selected Property 3 and 4 since they can be easily expressed as a single verification query in our tool. In the words of [11], 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 [11], each property can be assessed on 42 different networks depending on the choice of two parameters. Among the 84 networks available, we selected those for which our over-approximate analysis was not able to find a definitive answer, ending with a total of 9 networks. Notice that Property 3 and Property 4 are always satisfied in these networks. Table 1 shows the results of this experiment. Looking at the table, we can see that the complete analysis of pyNeVer is able to answer all the queries, whereas the over-approximate analysis does not succeed on any of them. Considering the results of the mixed analysis, we see that pyNeVer is able to answer all but two queries and the total amount of CPU time spent is noticeably less than the complete analysis and uniformly closer to the one reported for the over-approximate one. Arguably, the mixed methodology provides a good trade-off between precision and speed.

Our second experiment aims to compare pyNeVer to a pool of state-of-the-art tools. In particular, we consider four versions of ERAN [12, 13], i.e. the ones resulting from the combination of complete (c) and over-approximate (o) methodologies, using either polytopes (p) or zonotopes (z); we consider also Marabou [3], and two versions of NNV [10] featuring both complete and over-approximate methodologies (NNV$_c$ and NNV$_o$, respectively). We report the results in Table 2, where we denote ERAN versions with ERAN$_{xy}$, where $x \in \{c, o\}$ indicates the analysis, while $y \in \{p, z\}$ denotes the polyhedron type. Focusing on complete analyses, i.e., the results of ERAN$_{cp}$, ERAN$_{cz}$, Marabou and NNV$_c$, and comparing them with the related results of pyNeVer reported in Table 1, we can see that the complete analysis of pyNeVer is in the same ballpark as all but one of the other tools — ERAN$_{cz}$ exhausts available memory in one query. The same comparison, but focusing on over-approximation techniques, yields a different result: ERAN seems to strike a better balance between speed and precision since it is able to verify the properties even when using over-approximation. On the other hand, the performances of pyNeVer are on the same page with the ones reported for NNV$_o$. Finally, looking at Table 1 and focusing on the results related to the mixed analysis, we can see that it outperforms NNV$_o$ and it is close to ERAN$_{op}$ and ERAN$_{oz}$ in terms of verified properties.

References

1. L. Pulina and A. Tacchella. An abstraction-refinement approach to verification of artificial neural networks. In *Proc. of CAV'10*, pages 243–257, 2010.
2. X. Huang, D. Kroening, M. Kwiatkowska, W. Ruan, Y. Sun, E. Thamo, M. Wu, and X. Yi. Safety and trustworthiness of deep neural networks: A survey. *arXiv preprint arXiv:1812.08342*, 2018.
3. 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, and C. W. Barrett. The marabou framework for verification and analysis of deep neural networks. In *Proc. of CAV'19*, pages 443–452, 2019.
4. G. Singh, T. Gehr, M. Püschel, and M. T. Vechev. Boosting robustness certification of neural networks. In *Proc. of ICLR'19*, 2019.
5. H. Tran, X. Yang, D. M. Lopez, P. Musau, L. V. Nguyen, W. Xiang, S. Bak, and T. T. Johnson. NNV: the neural network verification tool for deep neural networks and learning-enabled cyber-physical systems. *CoRR*, abs/2004.05519, 2020.
6. S. Bak and P. S. Duggirala. Simulation-equivalent reachability of large linear systems with inputs. In *Proc. of CAV'17*, pages 401–420. Springer, 2017.
7. Open Neural Network Exchange the open standard for machine learning interoperability. `https://onnx.ai/`.
8. A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Proc. of NIPS'19*, pages 8024–8035, 2019.
9. D. Guidotti, F. Leofante, L. Pulina, and A. Tacchella. Verification of neural networks: Enhancing scalability through pruning. In *Proc. of ECAI'20*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, pages 2505–2512. IOS Press, 2020.
10. H. Tran, D. M. Lopez, P. Musau, X. Yang, L. V. Nguyen, W. Xiang, and T. T. Johnson. Star-based reachability analysis of deep neural networks. In *Proc. of FM'19*, volume 11800, pages 670–686. Springer, 2019.
11. G. Katz, C. W. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In *Proc. of CAV'17*, volume 10426 of *Lecture Notes in Computer Science*, pages 97–117, 2017.
12. G. Singh, T. Gehr, M. Mirman, M. Püschel, and M. T. Vechev. Fast and effective robustness certification. In *Proc. of NIPS'18*, pages 10825–10836, 2018.
13. G. Singh, T. Gehr, M. Püschel, and M. T. Vechev. An abstract domain for certifying neural networks. *Proc. ACM Program. Lang.*, 3(POPL):41:1–41:30, 2019.