Soft Computing (2024) 28:11647–11665
https://doi.org/10.1007/s00500-024-09907-5

NEURAL NETWORKS

NeVer2: learning and verification of neural networks

Stefano Demarchi[1] · Dario Guidotti[2] · Luca Pulina[2] · Armando Tacchella[1]

Accepted: 23 May 2024 / Published online: 31 July 2024

Abstract

NeVer2 is an open-source, cross-platform tool aimed at designing, training, and verifying neural networks. It seamlessly integrates popular learning libraries with our verification backend, offering their functionalities via a graphical interface. Users can design the structure of a neural network by intuitively arranging blocks on a canvas. Subsequently, network training involves specifying dataset sources and hyperparameters through dialog boxes. After training, the verification process entails two steps: (i) incorporating input preconditions and output postconditions via dedicated blocks, and (ii) initiating verification with a simple "push-button" action. To our knowledge, there is currently no other publicly available tool that encompasses all these features. In this paper, we present a comprehensive description of NeVer2, illustrating its complete integration of design, training, and verification through examples. Additionally, we conduct experimental analyses on various verification benchmarks to illustrate the trade-off between completeness and computability using different algorithms. We also include a comparison with state-of-the-art tools such as α,β- CROWN and NNV for reference.

Keywords Neural networks · Formal methods · Computer-aided verification · Trustworthy artificial intelligence

1 Introduction

Deep Neural Networks (DNNs) and Machine Learning (ML) models are increasingly being utilised across various applications due to their exceptional performance and versatility across different domains. Their application spans diverse fields, including financial analysis (Chen et al 2022; Sako et al 2022), stock price prediction (Yang et al 2023), analysis and control of industrial processes (Zhang et al 2023; Pavithra et al 2023; Jahanbakhti et al 2023; Guidotti et al 2023b, a), medical analysis (Dash et al 2023; Gul et al 2023), and recently, interpretation of MRI images within the context of the COVID-19 pandemic (Al-Waisy et al 2023; Dansana et al 2023). However, their application in safety- or security-sensitive environments (Katz et al 2017; Demarchi et al 2022) raises significant concerns. Since seminal works by Goodfellow et al (2015) and Szegedy et al (2014), it has been demonstrated over the past decade that DNNs may exhibit vulnerabilities in terms of their robustness. In particular, minimal alterations to correctly classified input data may lead to unexpected and incorrect responses from the network. Consequently, verification has emerged as a powerful approach to provide formal assurances regarding the behaviour of neural networks (Ferrari et al 2022; Demarchi et al 2022; Wang et al 2021; Katz et al 2019; Bak et al 2020; Kouvaros et al 2021; Singh et al 2019a; Tran et al 2020; Guidotti et al 2021; Henriksen and Lomuscio 2020; Guidotti et al 2019b; Henriksen and Lomuscio 2021; Guidotti et al 2020; Eramo et al 2022; Guidotti 2022; Leofante et al 2023; Guidotti et al 2023c, e, d). Furthermore, significant research efforts have been directed towards modifying networks to conform to specified criteria (Guidotti et al 2019b; Kouvaros et al 2021; Sotoudeh and Thakur 2021; Henriksen et al 2022; Guidotti et al 2019a), as well as exploring training methods that adhere to specific behavioural constraints (Cohen et al 2019; Hu et al 2016; Eaton-Rosen et al 2018; Giunchiglia and Lukasiewicz 2021; Giunchiglia et al 2022).

✉ Stefano Demarchi
stefano.demarchi@edu.unige.it

Dario Guidotti
dguidotti@uniss.it

Luca Pulina
lpulina@uniss.it

Armando Tacchella
armando.tacchella@unige.it

[1] DIBRIS, Università degli Studi di Genova, Viale Causa, 13, 16145 Genova, Italy

[2] DUMAS, Università degli Studi di Sassari, Via Roma, 151, 07100 Sassari, Italy

In this paper, we introduce NEVER2, a comprehensive tool that seamlessly integrates design, training, and verification functionalities for DNNs. At present, NEVER2 offers an environment where:

- Users can visually design the structure of a neural network as a block diagram using a graphical interface.
- Network parameters can be trained using a wrapper to the PYTORCH library. Both dataset sources and hyperparameters can be supplied via dialog boxes, effectively concealing the intricacies of PYTORCH from the user.
- Properties regarding the network can be associated with it visually, through the addition of special blocks in the graphical interface. The network can then be verified using our backend, PYNEVER (Guidotti et al 2021), in a straightforward "push-button" manner. Additionally, the user can configure PYNEVER through dialog boxes designed to simplify interaction.
- Networks in the ONNX and PYTORCH file formats, as well as existing properties in the VNN- LIB (Demarchi et al 2023) standard, are supported. The VNN- LIB standard is the input format in the annual competition of verification tools for neural networks (VNN-COMP) (Müller et al 2022). If a trained network is available in one of the formats accepted by NEVER2, it can be paired with properties and verified without the need for retraining.

To the best of our knowledge, no other tool combines the design, learning, and verification of DNNs within an open-source platform featuring a graphical interface. Furthermore, while most existing toolsets target users from the verification community, NEVER2 is tailored for domain experts with little to no experience in verification. Its aim is to empower them to design robust networks without being burdened by numerous technical details, which are often irrelevant given the scope of their tasks.

Finally, we present an experimental evaluation across various learning domains and verification tasks to compare the performance of NEVER2 against two prominent tools: α,β-CROWN, the winner of the last three VNN-COMPs, and NNV, another VNN-COMP contestant leveraging fundamental algorithms similar to NEVER2. Our findings reveal that while NEVER2 may exhibit slower performance compared to α,β- CROWN in most cases, it outperforms NNV in terms of speed and also demonstrates capability in handling certain verification tasks that currently exceed the capabilities of both tools.

The remainder of the paper is organized as follows. In Sect. 2, we conduct an extensive survey of related works, encompassing tools for visualization, learning, and verification of DNNs. Section 3 introduces definitions and notation to be utilized throughout the paper. In Sect. 4, we elaborate on the verification algorithms implemented in NEVER2. Section 5 offers an overview of the system architecture, while Sect. 6 presents examples of its usage via the graphical interface. Section 7 presents the results of our experimental evaluation. Finally, we conclude the paper in Sect. 8 with some closing remarks about the capabilities of the tool.

2 Related work

2.1 Learning tools

Tools for configuring a neural network for a specific task typically offer interfaces for defining the network's layers and their arrangement, as well as algorithms for learning the hyperparameters. The most widely used toolkits include PYTORCH (Paszke et al 2019), TENSORFLOW (Abadi et al 2016), and KERAS (Joseph et al 2021), among others tailored for specific case studies or programming languages.

PYTORCH, initially developed by Meta AI in 2017, is a Python library that serves as a modernized version of the older *Torch* library for machine learning. It is freely available and open source, serving as the foundation for numerous popular deep learning applications, including Tesla Autopilot, Uber's Pyro, and PyTorch Lightning.

TENSORFLOW, created by Google Brain, is a comprehensive machine learning platform offering both high-level and low-level interfaces for various tasks. It is used as the backbone for numerous commercial AI-enabled products, such as voice recognition, search engines, and email services.

KERAS is a high-level neural network Python library that operates atop TENSORFLOW, providing an optimized interface for defining neural network models and facilitating their learning process. Additionally, it supports multiple libraries as back-ends, including Microsoft's CNTK (Seide and Agarwal 2016).

2.2 Verification tools

Following the first release of NEVER (Pulina and Tacchella 2010), numerous other automated verification tools have emerged. Since 2020, an international competition (Müller et al 2022) has been held to stimulate development and promote collaboration within the verification community. In this summary, we outline the main tools that have yielded successful results in the competition, offering a comprehensive overview of the techniques developed for verification of neural networks.

MARABOU (Katz et al 2019) is a user-friendly Neural Network Verification toolkit that addresses queries about a network's properties by encoding and solving them as constraint satisfaction problems. It offers both Python and C++ APIs, enabling users to load neural networks and define arbitrary linear properties over them.

ERAN (Singh et al 2019b) is a neural network verifier that leverages abstract interpretations to encode the pre- and post-conditions of a network as Linear Programming (LP) or Mixed Integer Linear Programming (MILP) problems. It offers support for both complete verification, where a yes/no answer can be provided, and incomplete verification, where the property is verified on an over-approximation of the network, potentially resulting in false negatives. ERAN is capable of handling fully-connected, convolutional, and residual network architectures, encompassing various non-linearities such as ReLU, Sigmoid, Tanh, and Maxpool.

MIPVerify (Tjeng et al 2019) is a tool designed to assess the robustness of neural networks using Mixed Integer Linear Programming (MILP). It transforms queries regarding a neural network's robustness for specific inputs into MILP problems. This approach employs efficient solvers facilitated by tight specifications of ReLU and maximum constraints. Additionally, it uses a progressive bounds tightening strategy, focusing on refining bounds only when it can enhance the problem formulation with additional information.

NNV (Tran et al 2020) is primarily implemented using Matlab and applies reachability analysis techniques for verifying neural networks, particularly focusing on closed-loop neural network control systems in autonomous cyber-physical systems. NNV employs geometric representations enabling a layer-by-layer computation of the exact reachable set for feed-forward DNNs.

Venus (Botoeva et al 2020) is a verification toolkit that incorporates the dependency analysis procedure and enhances it with symbolic interval arithmetic and domain splitting techniques. Domain splitting methods partition the input domain into sub-domains, thereby refining the bound intervals of the nodes. Symbolic interval arithmetic techniques efficiently and accurately approximate these refined intervals.

nnenum (Bak 2021) employs various levels of abstraction to achieve high-performance verification of ReLU networks while maintaining completeness. The analysis combines three types of zonotopes with star set over-approximations and utilizes efficient parallelized ReLU case splitting.

VeriNet (Henriksen and Lomuscio 2020, 2021) is a comprehensive verification toolkit based on Symbolic Interval Propagation (SIP) for feed-forward neural networks. The core algorithm employs SIP to construct a linear abstraction of the network, which is then utilized in an LP-encoding to address the verification problem. To ensure completeness, a refinement phase based on a branch and bound methodology is incorporated.

α,β- CROWN is an efficient neural network verifier based on the linear bound propagation framework. It builds upon prior research on bound-propagation-based verifiers such as CROWN (Zhang et al 2018), α-CROWN (Xu et al 2021), β-CROWN (Wang et al 2021), and GCP-CROWN (Zhang et al 2022). Their most recent work, GCP-CROWN, represents the most comprehensive formulation of the linear bound propagation framework for neural network verification currently available.

Debona (Brix and Noll 2020) is a verification toolkit developed based on the 2020 edition of VeriNet. In addition to the Error-based Symbolic Interval Propagation (ESIP), which utilizes parallel upper and lower bounds for each neuron, it also supports Reversed Symbolic Interval Propagation (RSIP), which employs independent lower and upper bounds.

CGDTest (Nagisetty 2021) is a testing algorithm for DNNs that aims to identify an input compliant with user-defined constraints. It functions akin to a gradient-descent optimization method: initially, CGDTest interprets user-defined constraints and transforms them into a differentiable constraint loss function. Subsequently, starting from a random input, it leverages gradient descent to adjust it until the termination criteria are satisfied.

MN- BaB (Ferrari et al 2022) is an open-source neural network verifier that uses precise multi-neuron constraints in conjunction with efficient GPU-enabled linear bound-propagation within a branch and bound framework. MN-BaB offers completeness for piece-wise linear activation functions and is capable of handling fully-connected, convolutional, and residual network architectures containing ReLU, Sigmoid, Tanh, and Maxpool non-linearities.

PyRAT (Girard-Satabin et al 2022) is an acronym for Python Reachability Assessment Tool and serves as a neural network verification tool based on abstract interpretation techniques. Due to its tailored approach designed specifically for neural networks and their high dimensionality, PyRAT effectively applies abstract interpretation techniques while fully leveraging tensor operations. It supports a broad spectrum of neural networks and layers, ranging from simple and small tabular problems and networks to complex architectures with convolutional layers and skip connections.

2.3 Visualization tools

Creating graphical visualizations of neural network architectures is often a laborious and challenging task when done manually. NN- SVG (LeNail 2019) and Netron (Roeder 2023) stand out as the most popular tools offering this kind of capability. NN- SVG serves as a parametric designer tailored for generating high-quality examples of feed-forward and convolutional neural network models, primarily catering to researchers. Conversely, Netron functions as a visualizer for various formats of neural network models, providing comprehensive insights into the architecture and parameters of all layers within a network. More recently, the tool AIFiddle (Chappat 2023) has emerged with a modern interface, enabling users to design neural networks with a three-dimensional representation of data. Additionally, it offers features such as training on pre-defined datasets and in-depth analysis of the training procedure.

3 Background

In this section, we provide the primary background and context regarding neural networks and their verification. We explain our notation, define the behavior of neural networks, and outline the task of neural network verification.

3.1 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 $\|x\| = x \cdot x$. 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 $d(x, y) < \epsilon_x$. 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 $\mathcal{C}(X)$ 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

$$H = \{x \in \mathbb{R}^n \mid a_1x_1 + a_2x_2 + \ldots + a_nx_n = b\}$$

where $a \in \mathbb{R}^n$, $b \in \mathbb{R}$ and at least one component of a is non-zero. Let $f(x) = a_1x_1 + a_2x_2 + \ldots + a_nx_n - b$ be the affine form defining H. The *closed half-spaces associated with* H are defined as

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

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 $\mathbb{R}^n$.

3.2 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 layers*. 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 $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$;
- $f(x) = (\sigma_1(x_1), \ldots, \sigma_n(x_n))$ 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 component-wise to the vector x.

In this work, we consider two widely adopted activation functions $\sigma : \mathbb{R} \rightarrow \mathbb{R}$: the Rectified Linear Unit (*ReLU*) function, defined as $\sigma(r) = max(0, r)$, and the logistic function, defined as $\sigma(r) = \frac{1}{1+e^{-r}}$. Although not discussed here, convolutional layers with one or more filters can be represented as affine mappings (Gehr et al 2018).

For a neural network $\nu : \mathbb{R}^n \rightarrow \mathbb{R}^m$, the task of *classification* involves assigning one out of m labels to every input vector $x \in \mathbb{R}^n$: an input x is assigned to class k if $\nu(x)_k > \nu(x)_j$ for all $j \in [1, m]$ and $j \neq k$. It is important to mention that in the majority of neural network applications, a single *SoftMax* neuron is typically appended after the m outputs to yield a single value corresponding to the chosen class. However, existing verification benchmarks do not require the presence of this neuron; instead, they impose conditions directly on the m outputs. The task of *regression* aims to approximate a functional mapping from $\mathbb{R}^n$ to $\mathbb{R}^m$. In this context, neural networks comprising affine layers coupled with either ReLUs or logistic layers offer universal approximation capabilities (Hornik et al 1989).

3.3 Verification task

Given a neural network $\nu : \mathbb{R}^n \rightarrow \mathbb{R}^m$, our objective is to algorithmically verify its adherence to specified *post-conditions* on the output, provided it satisfies certain *pre-conditions* on the input. To ensure practical implementation on digital hardware, input domains must be bounded. Hence, even data from potentially unbounded physical processes are typically normalized within small ranges in practical applications. Consequently, we can assume without loss of generalization that the input domain of ν is a bounded set $I \subset \mathbb{R}^n$. This assumption leads to the corresponding output domain being a bounded set $O \subset \mathbb{R}^m$ because: (i) affine transformations of bounded sets remain bounded sets, (ii) ReLU is a piece-wise affine transformation of its input, (iii) the output of logistic functions is always bounded in the set $[0, 1]$, and (iv) the composition of bounded functions remains bounded. We stipulate that the logic formulas defining pre- and post-conditions should be 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 represent certain properties related to neural networks, such as invertibility or equivalence, it is able to represent the general task of testing resilience against adversarial perturbations. For instance, considering a network $\nu : I \rightarrow O$, where $I \subset \mathbb{R}^n$ and $O \subset \mathbb{R}^m$, which performs a classification task, an input vector $\hat{x} \in I$, and a corresponding output $\hat{y} \in O$ correctly classified with label λ, the formal definition of the safety property is as follows:

$$\forall x.\forall y.(\|x - \hat{x}\|_\infty \leq \varepsilon \wedge y = \nu(x)) \implies \|y - \hat{y}\|_\infty < \delta \tag{2}$$

where ε and δ are, respectively, the maximum perturbation admitted on the input vector and the maximum acceptable variance on the output vector such as the output label is still λ, and $\|x(y) - \hat{x}(\hat{y})\|_\infty$ is the Chebyshev norm measuring the difference between the original and perturbed vectors. Here Π corresponds to the ℓ_∞-norm around a given point with radius ε and Σ to the ℓ_∞-norm around the corresponding output, with radius δ.

Due to the presence of universal quantifiers, proving Equation (2) is challenging. As consequence, as all verification tools and benchmarks participating in VNN-COMP (Müller et al 2022), we focus on verifying an *unsafety* property defined as follows:

$$\exists x.\exists y.(\|x - \hat{x}\|_\infty \leq \varepsilon \wedge y = \nu(x)) \implies \|y - \hat{y}\|_\infty \geq \delta \tag{3}$$

It should be clear that, if we are able to certify that a solution to Eq. (3) exists, then Equation (2) is falsified and therefore the network is proven to be unsafe. That is, the solution of Equation (3) serves as a *counterexample* for the given safety property.

4 Verification methodology

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 abstract transformers for affine and functional layers and we prove that their composition provides a consistent over-approximation of corresponding concrete networks.

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

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. As shown in Zheng (2019), we can always start with an axis-aligned regular n simplex consisting of $n + 1$ facets—e.g., the triangle in $\mathbb{R}^2$ and the tetrahedron 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 $\gamma : \langle \mathbb{R}^n \rangle \rightarrow 2^{\mathbb{R}^n}$ that maps P to the set of points contained in it, i.e., $\gamma(P) = \{x \in \mathbb{R}^n \mid x \in P\}$.

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

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

$$\{\nu(x) \mid x \in X\} \subseteq \gamma(\tilde{\nu}(\alpha(X))) \tag{4}$$

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 $X_1 \ldots X_p$ exactly when

$$\{\nu(x) \mid x \in \cup_{i=1}^{p} X_i\} \subseteq \gamma(\tilde{\nu}(\alpha(X_1, \ldots, X_p))) \tag{5}$$

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

To represent polytopes and define the computations performed by abstract layers we resort to a specific subclass of *generalized star sets*, introduced in Bak and Duggirala (2017) and defined as follows—notation and proofs are derived from Tran et al (2019).

Definition 4 *(Generalized star set)* Given a *basis matrix* $V \in \mathbb{R}^{n \times m}$ obtained arranging a set of m *basis vectors* $\{v_1, \ldots v_m\}$ 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

$$[\![\Theta]\!] \equiv \{z \in \mathbb{R}^n \mid z = Vx + c \text{ s.t. } R(x_1, \ldots, x_m) = \top\} \tag{6}$$

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 Tran et al (2019); 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., $\Theta \in \langle \mathbb{R}^n \rangle$.*

This definition allows us to represent polytopes as generalized star sets. Henceforth, we will refer to generalized star sets adhering to these constraints simply as "stars". The most straightforward abstract layer to obtain is the one abstracting affine transformations. Since affine transformations of polytopes are still polytopes, we just need to define how to apply an affine transformation to a star—the definition is adapted from Tran et al (2019).

Definition 5 *(Abstract affine mapping)* Given a star set $\Theta = (c, V, R)$ and an affine mapping $f : R^n \rightarrow R^m$ with $f = Ax + b$, the abstract affine mapping $\tilde{f} : \langle R^n \rangle \rightarrow \langle R^m \rangle$ 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 $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, the corresponding abstract mapping $\tilde{f} : \langle \mathbb{R}^n \rangle \rightarrow \langle \mathbb{R}^m \rangle$ 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 $X \subset \mathbb{R}^n$.*

4.1 ReLU abstraction algorithms

The algorithm in Fig. 1 (Guidotti et al 2021) defines the abstract mapping of a functional layer with n ReLU activation functions and adapts the methodology proposed in Tran et al (2019). 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 piece-wise affine mapping. 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 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 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. 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.

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.

Fig. 1 Abstraction of the ReLU activation function

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

function COMPUTE_RELU($input = [\Gamma_1, \dots, \Gamma_M]$, $j$, $level$, $n$)
    $output = [\,]$
    for $k = 1 : M$ do
        $(lb_j, ub_j)$ = GET_BOUNDS($input[k]$, $j$)
        $M = [e_1 \dots e_{j-1}\ 0\ e_{j+1} \dots e_n]$
        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
                $\Theta_{low} = input[k] \wedge z[j] < 0$;   $\Theta_{upp} = input[k] \wedge z[j] \geq 0$
                $S = [M * \Theta_{low}, \Theta_{upp}]$
            else
                $(c, V, Cx \leq d) = input[j]$
                $C_1 = [0\ 0 \dots\ -1] \in \mathbb{R}^{1 \times m+1}$, $d_1 = 0$
                $C_2 = [V[j,:]\ \ -1] \in \mathbb{R}^{1 \times m+1}$, $d_2 = -c_k[j]$
                $C_3 = [\frac{-ub_j}{ub_j - lb_j} \cdot V[j,:]\ \ -1] \in \mathbb{R}^{1 \times m+1}$, $d_3 = \frac{ub_j}{ub_j - lb_j}(c[j] - lb_j)$
                $C_0 = [C\ 0^{m \times 1}]$, $d_0 = d$
                $\hat{C} = [C_0;\ \ C_1;\ \ C_2;\ \ C_3]$, $\hat{d} = [d_0;\ \ d_1;\ \ d_2;\ \ d_3]$
                $\hat{V} = MV$, $\hat{V} = [\hat{V}\ e_j]$
                $S = (Mc, \hat{V}, \hat{C}\hat{x} \leq \hat{d})$
            end if
        end if
        APPEND($output$, S)
    end for
    return $output$
end function
```

The linear-programming problem we need to solve in the GET_BOUNDS solver can be formalized as the following:

$$\begin{aligned} &(min/max)\, z_j = \mathbf{V}[j,:]\mathbf{x} + c[j] \\ &with \quad \mathbf{Cx} \leq \mathbf{d} \end{aligned}$$

The problem must be solved as minimization and maximization to provide the lower bound and the upper bound respectively. It should be noted that the size of the problem increases with the number of variables of the predicate of the star of interest. As a consequence the GET_BOUNDS function runtime increases as more over-approximation steps are applied to the stars of interest, whereas for "splitted" stars the size of the problem remains the same. Given a ReLU mapping $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$, the corresponding abstract mapping $\tilde{f} : \langle \mathbb{R}^n \rangle \rightarrow \langle \mathbb{R}^n \rangle$ defined in Fig. 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 $X \subset \mathbb{R}^n$.

5 Verification backend

pyNeVer (Guidotti et al 2021) serves as the backend for NeVer2. It is conceptualized as a modular API for managing DNNs, encompassing tasks from their building and training to verification. It comprises seven packages, each dedicated to either providing models of neural networks or implementing strategies for their conversion, training, and verification. To ensure a precise semantics of the internal model and to abstract away implementation details, we structured our own internal network representation as a graph, where nodes correspond to distinct layers. Additionally, we provide a

Table 1 List of the concrete layer classes available in PYNEVER, grouped by functionality

Layer type	Class name	Layer	Notes
Linear layers	`FullyConnectedNode`	Fully connected	General matrix multiplication
Convolution layers	`ConvNode`	Convolutional	Multi-dimensional
Activation layers	`ReLUNode`	ReLU	
	`ELUNode`	ELU	
	`CELUNode`	CELU	
	`LeakyReLUNode`	Leaky ReLU	
	`SigmoidNode`	Sigmoid	
	`TanhNode`	Tanh	
	`SoftMaxNode`	SoftMax	
Pooling layers	`AveragePoolNode`	Average pooling	
	`MaxPoolNode`	Max pooling	
Normalization layers	`BatchNormNode`	Batch normalization	Multi-dimensional
	`LRNNode`	Local response norm	
Dropout layers	`DropoutNode`	Dropout	
Utility layers	`FlattenNode`	Flatten	
	`ReshapeNode`	Reshape	
	`UnsqueezeNode`	Unsqueeze	

This set of operators is sufficient to represent most of the available benchmarks for sequential networks

representation for the datasets used in the learning phase. The system's primary capabilities revolve around abstraction, training, and verification. These functionalities are predominantly organized using *Strategy* patterns, defining general interfaces for network operations, with specialized subclasses offering support for these operations. Furthermore, to harness the capabilities of contemporary learning frameworks, we devised a set of conversion strategies to and from our internal representation and the representations accepted by the learning frameworks.

5.1 Representation

The internal representation of a neural network is managed through two abstract base classes: `NeuralNetwork` and `LayerNode`. In essence, `NeuralNetwork` serves as a container for `LayerNode` objects organized within it as a graph. For internal purposes, a list of `AlternativeRepresentation` objects is maintained—refer to subsection 5.2 for more details. In the current implementation, the only concrete subclass of `NeuralNetwork` is `SequentialNetwork`, representing networks where the corresponding graph is a list, meaning each layer is connected only to the next one. More intricate network topologies can be easily incorporated by creating other concrete subclasses of the abstract `NeuralNetwork` class. The concrete subclasses of `LayerNode` correspond to the network layers currently supported. Presently, based on the VNN- LIB (Demarchi et al 2023) specifications, the available layers are detailed in Table 1, which suffice to encode sequential DNNs with the most commonly used activation functions and layers. Notably, our representation is not "executable", that is, it lacks the capability to compute the output of a DNN given the input. As a consequence, our nodes contain only sufficient information to generate corresponding executable representations in different learning frameworks or to support encoding for verification purposes.

5.2 Conversion

The design of a model, aimed at generalizing those utilized in various learning frameworks, is based on the *Adapter* design pattern. We have introduced the abstract class `AlternativeRepresentation`, which is then specialized by `ONNXNetwork` and `PyTorchNetwork` to encode ONNX and PYTORCH models, respectively. These concrete subclasses encapsulate the actual network model within the corresponding learning framework and facilitate the interchangeability of their formats, such as in the case of ONNX.

The capabilities for conversion between our internal representation and the concrete subclasses of `AlternativeRepresentation` are provided by the subclasses of `ConversionStrategy`. This can also be regarded as an implementation of the *Builder* pattern. `ConversionStrategy` defines an interface comprising two functions: one for converting from our internal representation to another specific representation, and the other for performing the inverse operation. The concrete subclasses of

ConversionStrategy implement these functions for the corresponding concrete subclasses of AlternativeRepresentation. As new types of learning frameworks or architectures are integrated into pyNeVer, additional concrete subclasses of AlternativeRepresentation and ConversionStrategy will be introduced to support conversion.

5.3 Datasets

Datasets are managed through a versatile Dataset class, facilitating the direct encoding of Torch datasets for MNIST and fMNIST into the concrete classes MNISTDataset and FMNISTDataset. Additionally, the GenericFileDataset concrete class may be used to load any user-defined dataset, requiring specifications such as the dataset's separator character, data type, and target index, which indicates the index distinguishing inputs from outputs within each row. A dataset *Transform*, comprising a series of functions to be applied to dataset elements, such as normalization or flattening, is an optional parameter, allowing for extensive customization using this specific dataset class.

5.4 Abstraction

pyNeVer is a verifier which leverages abstract interpretation using star sets (Tran et al, 2019; Demarchi et al, 2022). To establish the abstract representation of a neural network, we use the same conceptual framework as the concrete network. This is achieved through the class AbstNeuralNetwork, which acts as a container for AbsLayerNode objects. Concrete subclasses such as AbsFullyConnectedNode, AbsReLUNode, AbsSigmoidNode, and AbsTanhNode define the algorithms for propagating bounded sets through the layers. For representing star sets, we utilize the Star class, which serves as the primary component for representing an abstract domain through a set of inequalities and an affine transformation. Additionally, the StarSet class is employed to create a collection of star elements for propagation throughout the network. The propagation algorithms, as detailed in Demarchi et al (2022), currently support only Fully Connected layers and ReLU, Sigmoid, or Tanh activation functions through the *forward* method of AbsLayerNode instances.

5.5 Training

To facilitate network training, we devised a training strategy that requires a *NeuralNetwork* and a *Dataset* instance. The result of the application of this strategy, presenting a singular function called train, yields a trained, concrete *NeuralNetwork* object. Presently, we have implemented a single procedure, PyTorchTraining, which relies on pyTorch as the training backend and is tailored to our internal representation. To support diverse training procedures and backends, it is possible to implement new subclasses of the abstract class TrainingStrategy and tailor them accordingly.

5.6 Verification

The ultimate objective of pyNeVer is to leverage abstract interpretation for verifying a specified property. Built upon the VNN- LIB standard (Demarchi et al 2023), our verification framework features the abstract class VerificationStrategy, which represents a generic interface comprising a single function. This function requires a NeuralNetwork instance alongside a Property, and returns a Boolean value indicating whether the property is verified or not, along with a counterexample (if available). The abstract class Property presents two subclasses: NeVerProperty and LocalRobustnessProperty. NeVerProperty embodies a general property conforming to the VNN- LIB standard and, as consequence, can be also parsed through reading an SMT-LIB (Barrett et al 2010) file. It comprises the input bounds and the output unsafe region(s). Conversely, LocalRobustnessProperty serves as a "template" property, encoding the search for an adversarial example corresponding to a specific data sample. Concrete subclasses of VerificationStrategy include NeVerVerification, which constitutes our principal contribution detailed in Guidotti et al (2021) and Demarchi et al (2022), along with a refinement-based variant named NeVerVerificationRef.

6 System overview

From a user's perspective, the primary feature of NeVer2 is to offer a graphical interface for interacting with the functionalities of both pyTorch and pyNeVer. The graphical user interface (GUI) of NeVer2 is constructed using the PyQt6 graphical library, with its main architecture outlined in Fig. 2. The application is structured as a canvas where nodes representing the network layers can be displayed and organized. A sidebar on the left provides buttons for drawing the supported layers, while another sidebar on the right can be opened on demand to display detailed information on specific layers. The training and verification windows can be accessed via the menu bar, which also includes placeholders for future features. In the following, we provide a detailed description of the environment and the resources available. It is worth noting that the graphical interface of NeVer2 is shared with another tool of ours, CoCoNet (Demarchi et al 2023), which is used for visualizing and converting DNNs in various formats.

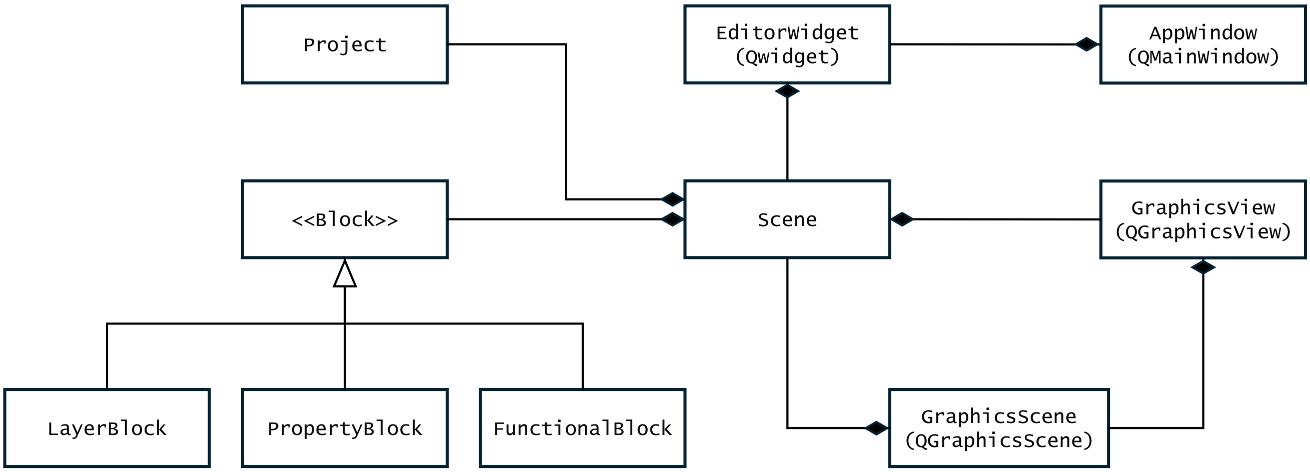


Fig. 2 UML Class Diagram representing the main software components of NEVER2. Using the PyQt API we leverage the `QGraphicsView` and `QGraphicsScene` interfaces to build a workspace in the `QMainWindow`. On the other hand, the class `Scene` serves as a controller for the creation and display of graphics blocks and as an interface to PYNEVER components

6.1 Building

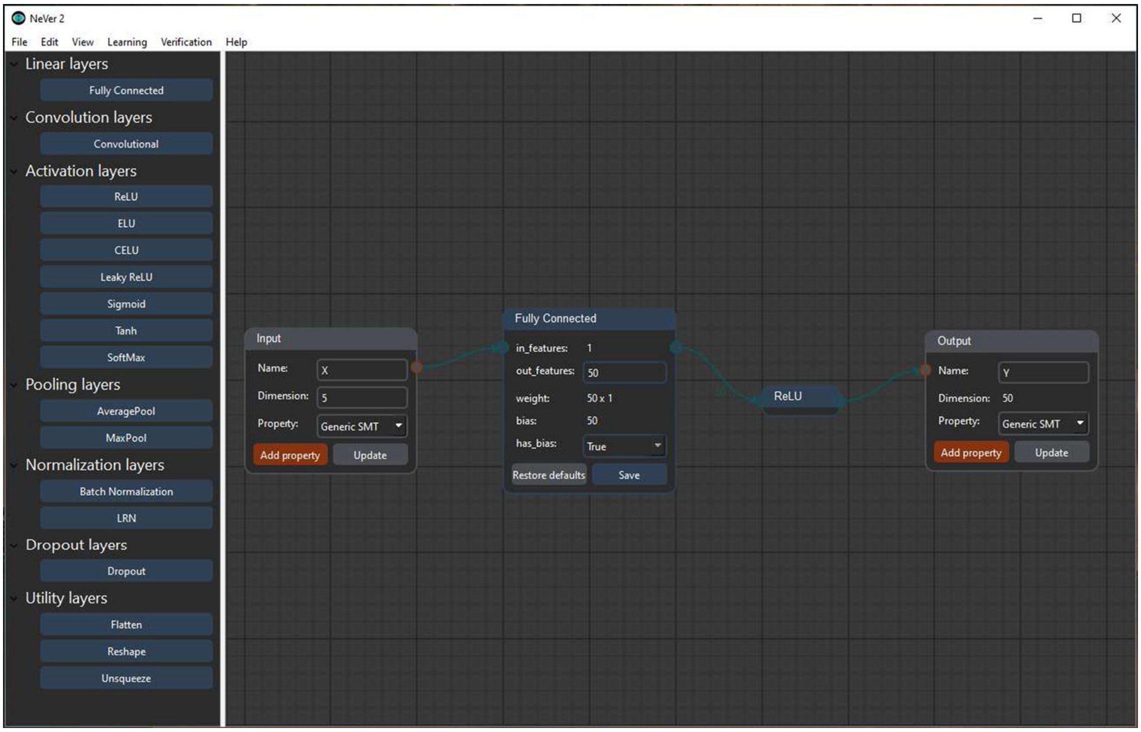
In Fig. 3, a screenshot of NEVER2's graphical interface is displayed. The interface shows input and output blocks for defining the network input and corresponding labels (first and last blocks). Additionally, it shows the definition of a fully connected layer with 50 neurons (second block), which is automatically added sequentially to the network, followed by a ReLU activation layer (third block). To update the layer's parameters before adding a new one, including the input block for specifying the input dimension, it is necessary to click on the block's "Save" button. The "Restore defaults" option resets the values to their default settings without overwriting. At this stage, the neural network designed graphically is supported by the internal representation outlined in Sect. 5.1. This representation performs necessary checks to ensure a correct representation. For instance, attempting to add incompatible layers, such as convolutional layers with a single-dimension shape, results in error messages explaining why the connection is not possible. Once the network is finalized, it can be saved in the ONNX or PYTORCH file formats by navigating to the "File... → Save/Save as..." menu.

6.2 Training

Suppose we wish to train a neural network on the ACAS XU dataset (Katz et al 2017). In Fig. 4, we have clicked on the menu "Learning... → Train..." and can see the window for setting up the procedure. Initially, we select the dataset as a "Custom data source". As mentioned in Sect. 5.3, we directly provide access to the MNIST and fMNIST datasets. However, any dataset can be loaded through our interface as long as it is formatted as a text file. When importing a custom data source, the user is prompted to enter the expected data type, the delimiter character, and the target index. The default values for the data type and delimiter character, matching those used in ACAS XU, are "float" and the comma character, respectively.

While some networks may include pre-processing layers for, e.g., normalization, in VNN- LIB we strongly discourage this behavior because the properties and verification algorithms are defined on already normalized networks. For this reason, it is possible to apply a *transform* to the dataset following PYTORCH's style. In Fig. 5, we demonstrate how a custom transform can be applied by combining one or more functions, such as normalization and flattening for the ACAS XU dataset. It is important to note that the normalization parameters are required from the user when selecting the corresponding transform function. We also provide two transforms whose parameters are already specifically tailored for convolutional and fully connected networks for the MNIST and fMNIST datasets.

Finally, we can initiate the training procedure by specifying the remaining parameters in the window. The learning algorithm utilizes the *Adam* optimizer (Kingma and Ba 2015) for PYTORCH networks, which means that the model is internally converted to PYTORCH before training. The window provides selectors for the *Optimizer* and the *Scheduler*, although there is only one option available at this time. Both the optimizer and the scheduler have additional parameters accessible on the right side of the dialog. In Fig. 6, we can see the parameters related to the **Adam** optimizer. Next, we can choose the *Loss function* (either **Cross Entropy** or **MSE loss**) and the *Precision metric* (either **Inaccuracy** or **MSE loss**). Then, we specify the number of training *Epochs*, the

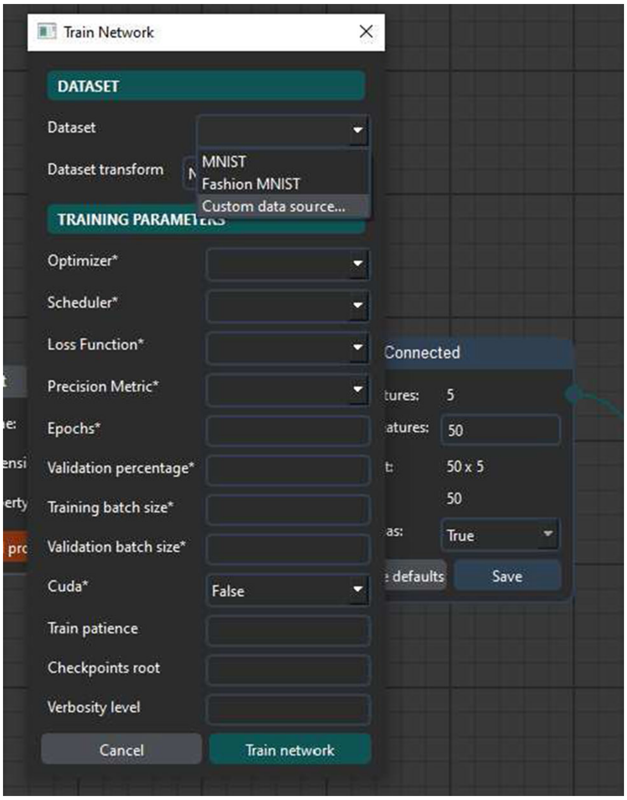


Input
Name: X
Dimension: 5
Property: Generic SMT

Fully Connected
in_features: 1
out_features: 50
weight: 50 x 1
bias: 50
has_bias: True

Output
Name: Y
Dimension: 50
Property: Generic SMT

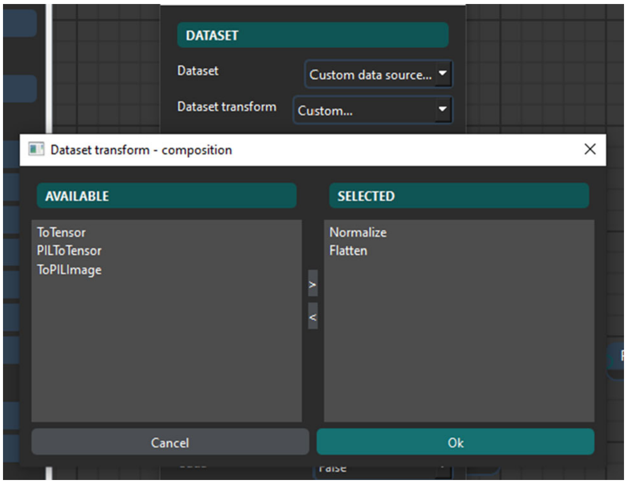
Fig. 3 Screenshot of the NeVer2 interface. The main component is the canvas, where layers are depicted as blocks. On the left side, there is a list of available layers. In this screenshot, we have added a fully connected layer followed by a ReLU layer



Dataset
Dataset transform
Optimizer*
Scheduler*
Loss Function*
Precision Metric*
Epochs*
Validation percentage*
Training batch size*
Validation batch size*
Cuda* False
Train patience
Checkpoints root
Verbosity level

tures: 5
atures: 50
t: 50 x 5
50
as: True

Fig. 4 Screenshot of the training window in NeVer2. Here we see the required parameters and the selection of the dataset as a custom data source



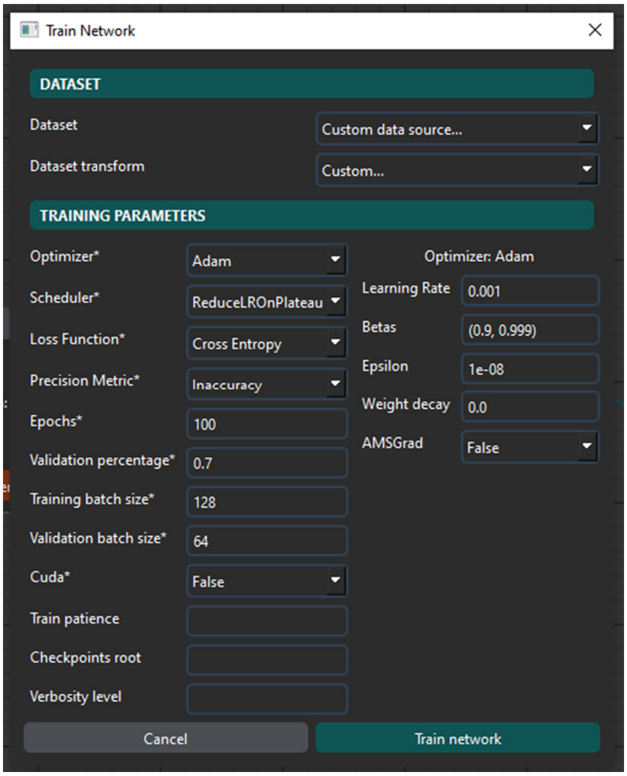
Dataset: Custom data source...
Dataset transform: Custom...

ToTensor
PILToTensor
ToPILImage

Normalize
Flatten

Fig. 5 Screenshot of the dataset transform builder in NeVer2. The left list shows the available transforms, which can be composed in the right list using the two keys in the middle

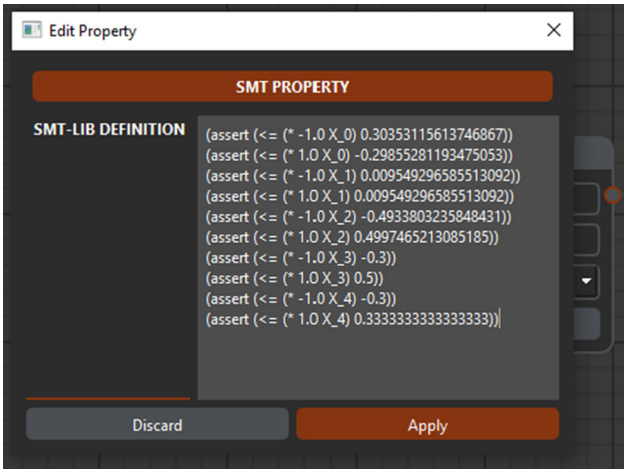
portion of the dataset to be used as the *Validation* set, and the *Training batch* and *Validation batch* sizes. Additionally, we can utilize the *CUDA* cores of the GPU, if supported, set an early stopping criterion using the *Train patience*, adjust the directory for storing *Checkpoints*, and control the *Verbosity level*.



Dataset: Custom data source...
Dataset transform: Custom...
Optimizer*: Adam
Scheduler*: ReduceLROnPlateau
Loss Function*: Cross Entropy
Precision Metric*: Inaccuracy
Epochs*: 100
Validation percentage*: 0.7
Training batch size*: 128
Validation batch size*: 64
Cuda*: False
Train patience:
Checkpoints root:
Verbosity level:

Optimizer: Adam
Learning Rate: 0.001
Betas: (0.9, 0.999)
Epsilon: 1e-08
Weight decay: 0.0
AMSGrad: False

Fig. 6 Screenshot of the filled training window in NeVer2



```
(assert (<= (* -1.0 X_0) 0.30353115613746867))
(assert (<= (* 1.0 X_0) -0.29855281193475053))
(assert (<= (* -1.0 X_1) 0.009549296585513092))
(assert (<= (* 1.0 X_1) 0.009549296585513092))
(assert (<= (* -1.0 X_2) -0.4933803235848431))
(assert (<= (* 1.0 X_2) 0.4997465213085185))
(assert (<= (* -1.0 X_3) -0.3))
(assert (<= (* 1.0 X_3) 0.5))
(assert (<= (* -1.0 X_4) -0.3))
(assert (<= (* 1.0 X_4) 0.3333333333333333))
```

Fig. 7 Screenshot of a "Generic SMT" property definition in NeVer2

6.3 Verification

After training the network, we can specify VNN- LIB properties concerning both the input and the output. We show in Fig. 7 the input bounds for property *P3* of the ACAS XU benchmark. Utilizing the "Generic SMT" option available in the input block, we simply select "Add property" to access a text area where we can input plain SMT-LIB constraints. This visualization is also the default when opening a property file.

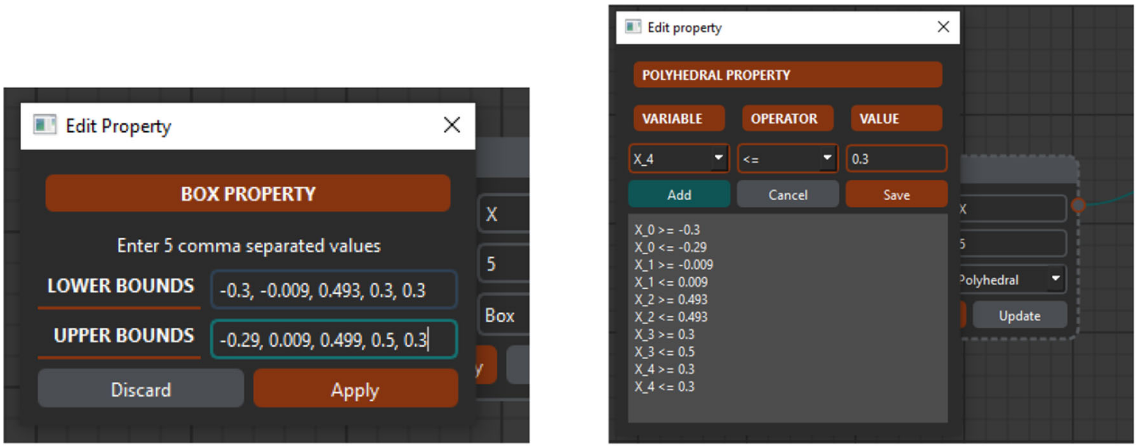
While the standard representation of properties in VNN-LIB follows the specific format presented, we also offer two more user-friendly interfaces to enable non-experts to define their own properties. In Fig. 8a and Fig. 8b, we present the same input bounds as depicted in Fig. 7, but defined using the "Box" option for lower and upper bounds, or the "Polyhedral" option for bounding variable-by-variable with constraints, respectively. Note that in these representations, bounding values are truncated for readability. Once a property is defined, a corresponding block is added to the canvas and connected to the input or output node, allowing visualization and modification of the property.

Finally, we can verify the property by selecting "Verification... → Verify..." from the menu. This action opens the verification dialog shown in Fig. 9, where we can choose the verification strategy based on different verification algorithms detailed in Demarchi et al (2022) and briefly described in the following.

Complete analysis With this algorithm, whenever a neuron is unstable for dimension j, the input star is "split" into two new stars: one considering all points where $z < 0$ (Θ_{low}), and the other considering points where $z \geq 0$ (Θ_{upp}) along dimension j. Both Θ_{low} and Θ_{upp} are derived by adding appropriate constraints to the input star *input[k]*. If the analysis at lines $20-21$ of Algorithm 1 is applied throughout the network, and the input abstraction is precise, then the abstract output will also be precise, meaning it will coincide with the concrete one. In this case, we refer to the analysis of NeVer2 as "complete". However, due to the worst-case exponential growth in the number of resulting stars, the complete analysis may become practically infeasible.

Over-approximate analysis With this algorithm, the ReLU function is abstracted using the over-approximation method proposed in Tran et al (2019). This approach is considerably less conservative compared to others, such as those based on zonotopes or abstract domains, and yields a tighter abstraction. If this analysis is applied across the entire network, the resulting output star will be a sound over-approximation of the concrete output range. In this case, we refer to the analysis of NeVer2 as "over-approximate". Although the number of stars remains constant throughout the analysis, each unstable neuron introduces a new predicate variable, which linearly increases the size of the program to be solved by GET_BOUNDS.

Mixed analysis In Guidotti et al (2021), a novel approach is introduced to manage varying levels of abstraction during the analysis. Algorithm 1 is devised to control the abstraction at the individual neuron level, allowing for each neuron to have its own refinement level. This algorithm strikes a balance between the complete and over-approximate ones. To try to minimize the approximation error, neurons within each layer are ranked based on the area of their linear relaxation. Intuitively, neurons with wider bounds contribute to a



Edit Property

BOX PROPERTY

Enter 5 comma separated values

LOWER BOUNDS: -0.3, -0.009, 0.493, 0.3, 0.3

UPPER BOUNDS: -0.29, 0.009, 0.499, 0.5, 0.3

(a)

Edit property

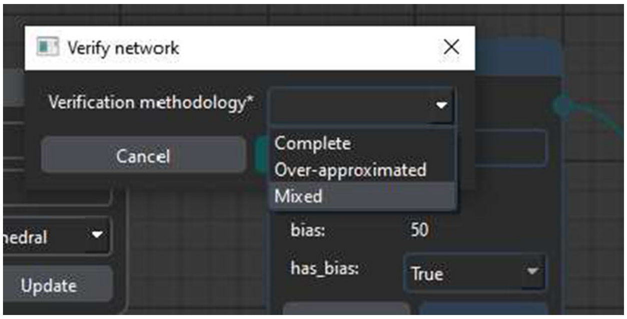
POLYHEDRAL PROPERTY

VARIABLE: X_4 OPERATOR: <= VALUE: 0.3

```
X_0 >= -0.3
X_0 <= -0.29
X_1 >= -0.009
X_1 <= 0.009
X_2 >= 0.493
X_2 <= 0.493
X_3 >= 0.3
X_3 <= 0.5
X_4 >= 0.3
X_4 <= 0.3
```

(b)

Fig. 8 Property definition alternatives in NeVer2 with Box (left) and Polyhedral (right) interfaces. **a** Screenshot of a “Box” property definition in NeVer2. The number of lower and upper bounds must be consistent with the network input. **b** Screenshot of a “Polyhedral” property definition in NeVer2. Here each variable can be bounded separately



Verify network

Verification methodology*

bias: 50

has_bias: True

Fig. 9 Screenshot of the verification window in NeVer2. The three alternatives refer to different abstract propagation algorithms that are either faster or more precise

broader area and consequently a larger approximation error. We then split the star along the neuron with the widest bounds and propagate the approximate method across the remaining neurons in the layer, ensuring that each layer undergoes at most a single split. This significantly reduces computational costs, as the growth becomes quadratic in the number of layers, and the complexity increase due to the approximation is contained. We refer to the analysis of NeVer2 in this case as “mixed”.

The verification dialog records the outcomes of the verification process and returns either “True” or “False” based on whether the network is deemed safe or not.

7 Evaluation and benchmarks

In this evaluation, we assess NeVer2 using a variety of verification tasks drawn from prior research studies, including some case studies proposed by us. It is worth noting that while the verification community has made significant strides in developing innovative methodologies, there remains a shortage of widely accepted benchmarks. Among the few available, the ACAS XU benchmark, introduced in 2017, remains the most prominent (Katz et al 2017). In our selection process, we aimed to choose tasks relevant to practical applications while ensuring that the neural networks involved were sufficiently small for existing verification techniques to be effectively applied.

Table 2 Details of the benchmarks on which we evaluate the performances of NeVer2 and other systems

Case study	Networks	Neurons	Properties	Instances
ACAS XU	45	300	10	172
ACC	3	30—90	5	15
Cartpole	1	128	100	100
Lunar Lander	1	192	100	100
Dubins Rejoin	1	768	96	96
Drone hovering	8	48—448	16	16
TOTAL	59		91	499

Each case study involves one or more networks which differ in the architecture or in the learning process, and that should be verified against multiple properties

In Table 2, we present the selection of benchmarks considered in our evaluation. We include the *ACAS XU* benchmark both as a reference point and because it features the “deepest” models, comprising five layers with 30 neurons each. The safety properties for testing are the standard ones outlined in Katz et al (2017). Additionally, we introduce the *ACC* case study from our prior work Demarchi et al (2022).

Table 3 Result of the performance evaluation of NeVer2, NNV and α,β- CROWN

Benchmark	Instances	Solved instances Over-approx	Mixed	Complete	NNV	α,β- CROWN
ACAS XU	172	61	78	100	61	170
ACC	15	0	0	15	–	–
Cartpole	100	95	95	100	100	100
Lunar Lander	100	18	18	99	96	100
Dubins Rejoin	96	5	5	6	2	96
Drone hovering	16	0	0	12	12	16
TOTAL	499	179	196	332	271	482

For each benchmark we report the number of solved instances by each tool and setting. Note that for the abstract algorithms of NeVer2 we consider an instance solved only if it returns "True"

This case study involves training three network architectures with a total neuron count ranging from 30 to 90, and we assess five safety properties for each network. We also include three Reinforcement Learning benchmarks, namely *Cartpole*, *Lunar Lander*, and *Dubins Rejoin*, sourced from VNN-COMP 2022 (Müller et al 2022). These benchmarks feature networks of increasing complexity, with neuron counts ranging from 128 to 768, and entail a set of 100 local robustness properties, except for the Dubins Rejoin network, which has 96 properties. Lastly, we introduce the *Drone hovering* Reinforcement Learning benchmark, where the task is to control a drone to hover at a specified height by adjusting the RPM of its four rotors. We present eight architectures with neuron counts ranging from 48 to 448, along with two local robustness properties for each architecture (Demarchi 2023).

Table 3 and Fig. 10 present the results of our experimental evaluation, conducted on a cluster comprising identical PCs featuring 12th Gen Intel Core i7-127000KF processors @3.61GHz and 32GB of RAM, running Ubuntu Linux 20.04.6 LTS. All experiments were executed with a timeout of 15 min, which exceeds the VNN-COMP timeout of 5 min to accommodate the substantial difference in computing infrastructure between our setup and that of VNN-COMP. We compare the performance of NeVer2 across three settings: *Over-approx*, *Mixed*, and *Complete*, against the solvers NNV and α,β- CROWN in a complete verification setting. We focus on these tools because NNV is the sole VNN-COMP contestant employing reachability analysis using stars, while α,β- CROWN has emerged as the winner in the last three VNN-COMP editions. This comparative analysis enables us to position NeVer2 in relation to other state-of-the-art tools.

In Fig. 10, we present cactus plots illustrating the performance of the five algorithms across each benchmark class outlined in Table 2. In cactus plots, the results of each solver on each benchmark (specific network and property) are arranged independently from the other solvers. Consequently, points at the same point of the x-axis may not correspond to the same benchmark, and benchmarks where solvers time out are excluded from consideration. Conceptually, a cactus plot provides an indication of "how far a solver can progress" relative to a family of benchmarks. Generally, the further the cactus arm extends to the right and the lower it is on the y-axis, the better the solver's performance. In Fig. 10, we illustrate the outcomes of algorithms that returned either "True" or "False" in a complete verification setting, and only "True" for the incomplete verification settings of NeVer2. This distinction is made because in incomplete verification, "False" is not guaranteed to be a correct answer due to potential approximation errors. To complement our experimental findings, we present Table 3, which highlights the number of solved instances per benchmark. This tabular format is chosen as cactus plots are not conducive to displaying such detailed information.

Upon reviewing the plots, it becomes evident that the abstract algorithms employed by NeVer2 consistently yield responses in less time compared to the complete algorithm, whenever applicable. Conversely, the complete algorithm consistently delivers a conclusive answer, successfully identifying both safe and unsafe networks. Specifically, in both the ACC and Drones benchmarks, all properties are deemed unsafe, leading to the absence of records for abstract algorithms in the plots. For Cartpole, abstract algorithms verify 95 benchmarks, yet the difference in time between abstract and complete algorithms is negligible, making it challenging to distinguish between them. In Lunar Lander, abstract algorithms verify 18 instances, coinciding with the markers for α,β- CROWN in the plot. Under the 15-min timeout, the complete algorithm manages to verify all instances in the ACC and Cartpole classes, nearly all instances in the Drones and Lunar Lander classes, and more than half of the ACAS XU instances. However, the Dubins Rejoin class poses the greatest challenge, with only a few instances solved within the timeout period. This difficulty is expected due to the exponential growth in the number of stars corresponding to the increasing number of neurons, particularly when dealing with layers featuring 256 neurons right from the start, which significantly impact performance.

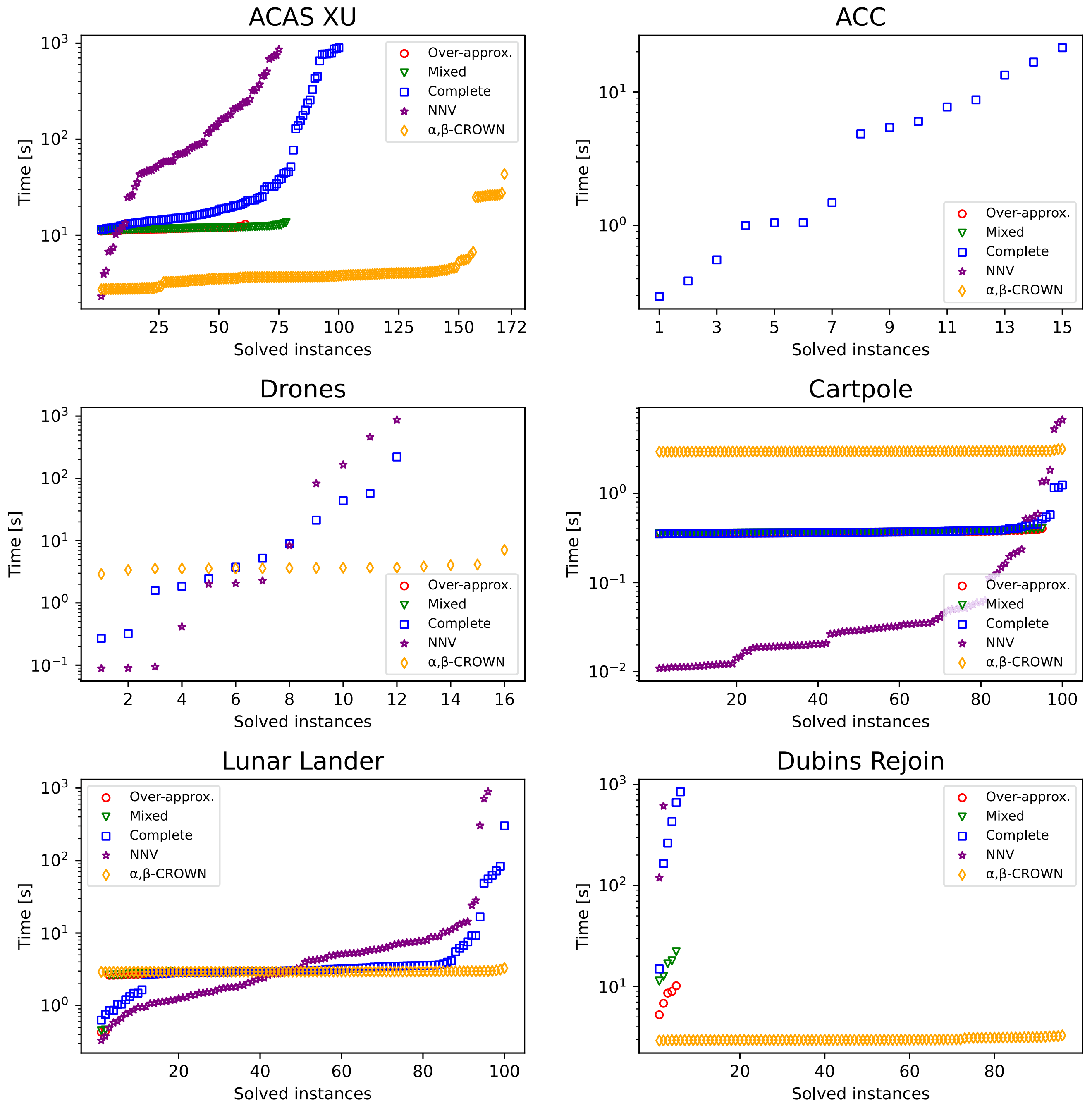


Fig. 10 Graphical analysis of the performance evaluation of NEVER2, NNV and α,β- CROWN using cactus plots. For each benchmark class we report the CPU time took by the verification algorithms for solving an instance. As in Table 3, for NEVER2 we consider only "True" answers as valid data when using abstract algorithms

Upon evaluating the performances of NNV and α,β-CROWN, it becomes evident that NEVER2 performs quite well compared to NNV, albeit being slower than α,β-CROWN. However, it demonstrates promising results particularly in scenarios involving over-approximation and mixed algorithms. When examining the Cartpole benchmark, we can evaluate the setup overhead of the tools due to its simplicity. Here, NNV demonstrates that its MATLAB implementation incurs less setup overhead compared to NEVER2 and α,β- CROWN. The performance gap is nearly two orders of magnitude; however, it is important to note that the runtimes are still relatively small for all the tools in the comparison. Additionally, it is worth mentioning that on this benchmark, NNV scales slightly worse than both

NeVer2 and α,β- CROWN. The findings from the case studies ACAS XU, Drones, and Lunar Lander exhibit notable similarities: α,β- CROWN, leveraging rapid bound propagation algorithms and GPU-based optimizations, effectively manages nearly every instance and typically outperforms both NNV and NeVer2 in terms of speed and instance resolution. NeVer2 consistently demonstrates faster performance than NNV, except in cases where its longer setup time hinders its efficacy. For completeness, it is worth noting that α,β- CROWN prematurely halted on two instances of ACAS XU due to a memory leak. The Dubins Rejoin benchmark clearly demonstrates the superiority of α,β- CROWN: while NeVer2 occasionally achieves comparable performance using incomplete abstract methods, α,β- CROWN consistently delivers conclusive results. Finally, the ACC case study presents a relatively straightforward benchmark. However, both NNV and α,β- CROWN struggled to address it due to the inability to represent the required pre-conditions, which cannot be represented as a hyper-rectangle in the input domain, that is, as the "standard" robustness property pre-condition. In the case of NNV, it is due to a problem with the property parser, since the verification algorithm itself is pretty similar to the one of NeVer2 and should be capable of successfully handling these benchmarks.

8 Conclusions

In this paper, we have presented NeVer2, the sole system presently integrating design, training, and verification functionalities for a significant subset of DNNs. Despite being a research prototype, NeVer2 facilitates the verification of small-to-medium scale networks which hold practical utility in control applications. Both NeVer2 and its verification backend, pyNeVer, are engineered to be easily extendable; their code is clear and well-documented to foster further contributions and extensions from the research community.

We assessed the performance of NeVer2 by comparing it with two other tools showcased in VNN-COMP: α,β- CROWN and NNV. The comparison results indicate that, while NeVer2 is slower than α,β- CROWN—the winner of the last three VNN-COMPs—it outperforms NNV, the sole other VNN-COMP contestant employing similar algorithms and data structures. Additionally, NeVer2 demonstrates the ability to handle more intricate safety and robustness specifications, as highlighted by the ACC case study.

Regarding verification capabilities, NeVer2 is applicable only to a subset of state-of-the-art verification benchmarks featuring feed-forward neural networks with ReLU activation functions. Furthermore, at this time NeVer2 does not support all operators available in pyTorch, meaning that certain DNNs trainable in pyTorch may not be visualized or verified in NeVer2. Nevertheless, commonly used operators are already accessible, and the set can be expanded to encompass additional ones.

NeVer2 and pyNeVer are open-source and can be freely downloaded for research and educational purposes from:

http://www.neuralverification.org/

Acknowledgements The research of Dario Guidotti has been supported by Fondazione di Sardegna, project "Tecniche e strumenti per la verifica di reti neurali". The research of Luca Pulina has been partially funded by the University of Sassari, research fund FAR2020PULINAL. The design of NeVer2 interface was part of a M.Sc. project carried by Andrea Gimelli and Giacomo Rosato at University of Genova. We also would like to thank the authors of α,β- CROWN and NNV for support in running their tools and the organizers of VNN-COMP for providing all researchers with a yearly assessment of state-of-the-art DNN verification tools.

Author Contributions All authors contributed to the study conception and design. Material preparation, data collection and analysis were performed by Stefano Demarchi. The first draft of the manuscript was written by Stefano Demarchi and Armando Tacchella and all authors commented on previous versions of the manuscript. All authors read and approved the final manuscript.

Funding Open access funding provided by Università degli Studi di Genova within the CRUI-CARE Agreement. This work was supported by Fondazione di Sardegna and Università degli Studi di Sassari (Grant numbers CUP J85F21001000005 and FAR2020PULINAL). The authors have no other relevant financial or non-financial interests to disclose.

Data Availability The experimental data and results of the current study are available in the GitHub repository https://github.com/NeVerTools/pyNeVer/tree/main/examples/submissions/2023_SoftComputing.

Declarations

Conflict of interest The authors have not disclosed any competing interests.

References

Abadi M, Barham P, Chen J, et al (2016) Tensorflow: a system for large-scale machine learning. In: 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016. USENIX Association, pp 265–283

Al-Waisy AS, Al-Fahdawi S, Mohammed MA et al (2023) Covid-chexnet: hybrid deep learning framework for identifying COVID-19 virus in chest x-rays images. Soft Comput 27(5):2657–2672. https://doi.org/10.1007/s00500-020-05424-3

Bak S (2021) nnenum: Verification of relu neural networks with optimized abstraction refinement. In: NASA Formal Methods - 13th International Symposium, NFM 2021, Virtual Event, May 24-28, 2021. Proceedings, Lecture Notes in Computer Science, vol 12673. Springer, pp 19–36. https://doi.org/10.1007/978-3-030-76384-8_2

Bak S, Duggirala PS (2017) Simulation-equivalent reachability of large linear systems with inputs. In: Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017. Proceedings, Part I, Lecture Notes in Computer Science, vol 10426. Springer, pp 401–420. https://doi.org/10.1007/978-3-319-63387-9_20

Bak S, Tran H, Hobbs K, et al (2020) Improved geometric path enumeration for verifying relu neural networks. In: Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I, Lecture Notes in Computer Science, vol 12224. Springer, pp 66–96. https://doi.org/10.1007/978-3-030-53288-8_4

Barrett C, Stump A, Tinelli C (2010) The SMT-LIB Standard: Version 2.0. In: Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK), p 14

Botoeva E, Kouvaros P, Kronqvist J, et al (2020) Efficient verification of relu-based neural networks via dependency analysis. In: The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020. AAAI Press, pp 3291–3299. https://doi.org/10.1609/AAAI.V34I04.5729

Brix C, Noll T (2020) Debona: decoupled boundary network analysis for tighter bounds and faster adversarial robustness proofs. CoRR abs/2006.09040

Chappat E (2023) AiFiddle.io. https://aifiddle.io

Chen Y, Guo J, Huang J et al (2022) A novel method for financial distress prediction based on sparse neural networks with $l_{1/2}$ regularization. Int J Mach Learn Cybern 13(7):2089–2103. https://doi.org/10.1007/s13042-022-01566-y

Cohen J, Rosenfeld E, Kolter JZ (2019) Certified adversarial robustness via randomized smoothing. In: Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA, Proceedings of Machine Learning Research, vol 97. PMLR, pp 1310–1320

Dansana D, Kumar R, Bhattacharjee A et al (2023) Early diagnosis of covid-19-affected patients based on x-ray and computed tomography images using deep learning algorithm. Soft Comput 27(5):2635–2643. https://doi.org/10.1007/s00500-020-05275-y

Dash S, Parida P, Mohanty JR (2023) Illumination robust deep convolutional neural network for medical image classification. Soft Comput. https://doi.org/10.1007/s00500-023-07918-2

Demarchi S (2023) Experimenting with constraint programming techniques in artificial intelligence: automated system design and verification of neural networks. PhD thesis, University of Genoa, Italy. https://hdl.handle.net/11567/1117675

Demarchi S, Guidotti D, Pitto A, et al (2022) Formal verification of neural networks: A case study about adaptive cruise control. In: Hameed IA, Hasan A, Alaliyat SA (eds) Proceedings of the 36th ECMS International Conference on Modelling and Simulation, ECMS 2022, Ålesund, Norway, May 30 - June 3, 2022. European Council for Modeling and Simulation, pp 310–316. https://doi.org/10.7148/2022-0310

Demarchi S, Guidotti D, Pulina L, et al (2023) Supporting standardization of neural networks verification with vnnlib and coconet. In: Proceedings of the 6th Workshop on Formal Methods for ML-Enabled Autonomous Systems, Kalpa Publications in Computing, vol 16. EasyChair, pp 47–58. https://doi.org/10.29007/5pdh

Eaton-Rosen Z, Bragman FJS, Bisdas S, et al (2018) Towards safe deep learning: Accurately quantifying biomarker uncertainty in neural network predictions. In: Medical Image Computing and Computer Assisted Intervention - MICCAI 2018 - 21st International Conference, Granada, Spain, September 16-20, 2018, Proceedings, Part I, Lecture Notes in Computer Science, vol 11070. Springer, pp 691–699. https://doi.org/10.1007/978-3-030-00928-1_78

Eramo R, Fanni T, Guidotti D, et al (2022) Verification of neural networks: challenges and perspectives in the aidoart project (short paper). In: Proceedings of the 10th Italian workshop on Planning and Scheduling (IPS 2022), RCRA Incontri E Confronti (RiCeRcA 2022), and the workshop on Strategies, Prediction, Interaction, and Reasoning in Italy (SPIRIT 2022) co-located with 21st International Conference of the Italian Association for Artificial Intelligence (AIxIA 2022), November 28 - December 2, 2022, University of Udine, Udine, Italy, CEUR Workshop Proceedings, vol 3345. CEUR-WS.org

Ferrari C, Müller MN, Jovanovic N, et al (2022) Complete verification via multi-neuron relaxation guided branch-and-bound. In: The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022. OpenReview.net

Gehr T, Mirman M, Drachsler-Cohen D, et al (2018) AI2: safety and robustness certification of neural networks with abstract interpretation. In: 2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA. IEEE Computer Society, pp 3–18. https://doi.org/10.1109/SP.2018.00058

Girard-Satabin J, Alberti M, Bobot F, et al (2022) CAISAR: a platform for characterizing artificial intelligence safety and robustness. In: Proceedings of the Workshop on Artificial Intelligence Safety 2022 (AISafety 2022) co-located with the Thirty-First International Joint Conference on Artificial Intelligence and the Twenty-Fifth European Conference on Artificial Intelligence (IJCAI-ECAI-2022), Vienna, Austria, July 24-25, 2022, CEUR Workshop Proceedings, vol 3215. CEUR-WS.org

Giunchiglia E, Lukasiewicz T (2021) Multi-label classification neural networks with hard logical constraints. J Artif Intell Res 72:759–818. https://doi.org/10.1613/JAIR.1.12850

Giunchiglia E, Stoian MC, Lukasiewicz T (2022) Deep learning with logical constraints. In: Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022. ijcai.org, pp 5478–5485. https://doi.org/10.24963/IJCAI.2022/767

Goodfellow IJ, Shlens J, Szegedy C (2015) 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

Guidotti D, Leofante F, Castellini C, et al (2019a) Repairing learned controllers with convex optimization: a case study. In: Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 16th International Conference, CPAIOR 2019, Thessaloniki, Greece, June 4-7, 2019, Proceedings, Lecture Notes in Computer Science, vol 11494. Springer, pp 364–373. https://doi.org/10.1007/978-3-030-19212-9_24

Guidotti D, Leofante F, Pulina L, et al (2019b) Verification and repair of neural networks: a progress report on convolutional models. In: AI*IA 2019 - Advances in Artificial Intelligence - XVIIIth International Conference of the Italian Association for Artificial Intelligence, Rende, Italy, November 19-22, 2019, Proceedings, Lecture Notes in Computer Science, vol 11946. Springer, pp 405–417. https://doi.org/10.1007/978-3-030-35166-3_29

Guidotti D, Leofante F, Pulina L, et al (2020) Verification of neural networks: Enhancing scalability through pruning. In: ECAI 2020—24th European Conference on Artificial Intelligence, 29 August-8 September 2020, Santiago de Compostela, Spain, August 29 - September 8, 2020 - Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020), Frontiers in Artificial Intelligence and Applications, vol 325. IOS Press, pp 2505–2512. https://doi.org/10.3233/FAIA200384

Guidotti D, Pulina L, Tacchella A (2021) pynever: a framework for learning and verification of neural networks. In: Automated Technology for Verification and Analysis - 19th International Symposium, ATVA 2021, Gold Coast, QLD, Australia, October 18-22, 2021, Proceedings, Lecture Notes in Computer Science, vol 12971. Springer, pp 357–363. https://doi.org/10.1007/978-3-030-88885-5_23

Guidotti D (2022) Verification of neural networks for safety and security-critical domains. In: Proceedings of the 10th Italian workshop on Planning and Scheduling (IPS 2022), RCRA Incontri E Confronti (RiCeRcA 2022), and the workshop on Strategies, Prediction, Interaction, and Reasoning in Italy (SPIRIT 2022) co-located with 21st International Conference of the Italian Association for Artificial Intelligence (AIxIA 2022), November 28 - December 2, 2022, University of Udine, Udine, Italy, CEUR Workshop Proceedings, vol 3345. CEUR-WS.org

Guidotti D, Masiero R, Pandolfo L, et al (2023a) Vector reconstruction error for anomaly detection: preliminary results in the IMOCO4.E project. In: 28th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2023, Sinaia, Romania, September 12-15, 2023. IEEE, pp 1–4. https://doi.org/10.1109/ETFA54631.2023.10275396

Guidotti D, Pandolfo L, Pulina L (2023b) Detection of component degradation: A study on autoencoder-based approaches. In: 19th IEEE International Conference on e-Science, e-Science 2023, Limassol, Cyprus, October 9-13, 2023. IEEE, pp 1–2. https://doi.org/10.1109/E-SCIENCE58273.2023.10254890

Guidotti D, Pandolfo L, Pulina L (2023) Leveraging satisfiability modulo theory solvers for verification of neural networks in predictive maintenance applications. Information 14(7):397. https://doi.org/10.3390/INFO14070397

Guidotti D, Pandolfo L, Pulina L (2023d) Verification of nns in the IMOCO4.E project: Preliminary results. In: 28th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2023, Sinaia, Romania, September 12-15, 2023. IEEE, pp 1–4. https://doi.org/10.1109/ETFA54631.2023.10275345

Guidotti D, Pandolfo L, Pulina L (2023e) Verifying neural networks with SMT: an experimental evaluation. In: 19th IEEE International Conference on e-Science, e-Science 2023, Limassol, Cyprus, October 9-13, 2023. IEEE, pp 1–2. https://doi.org/10.1109/E-SCIENCE58273.2023.10254877

Gul Y, Müezzinoglu T, Kilicarslan G et al (2023) Application of the deep transfer learning framework for hydatid cyst classification using CT images. Soft Comput 27(11):7179–7189. https://doi.org/10.1007/s00500-023-07945-z

Henriksen P, Lomuscio AR (2020) Efficient neural network verification via adaptive refinement and adversarial search. In: ECAI 2020 - 24th European Conference on Artificial Intelligence, 29 August-8 September 2020, Santiago de Compostela, Spain, August 29 - September 8, 2020 - Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020), Frontiers in Artificial Intelligence and Applications, vol 325. IOS Press, pp 2513–2520. https://doi.org/10.3233/FAIA200385

Henriksen P, Lomuscio A (2021) DEEPSPLIT: an efficient splitting method for neural network verification via indirect effect analysis. In: Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021. ijcai.org, pp 2549–2555. https://doi.org/10.24963/IJCAI.2021/351

Henriksen P, Leofante F, Lomuscio A (2022) Repairing misclassifications in neural networks using limited data. In: SAC '22: The 37th ACM/SIGAPP Symposium on Applied Computing, Virtual Event, April 25 - 29, 2022. ACM, pp 1031–1038. https://doi.org/10.1145/3477314.3507059

Hornik K, Stinchcombe MB, White H (1989) Multilayer feedforward networks are universal approximators. Neural Netw 2(5):359–366. https://doi.org/10.1016/0893-6080(89)90020-8

Hu Z, Ma X, Liu Z, et al (2016) Harnessing deep neural networks with logic rules. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers. The Association for Computer Linguistics. https://doi.org/10.18653/V1/P16-1228

Jahanbakhti H, Pourgholi M, Yazdizadeh A (2023) Online neural network-based model reduction and switching fuzzy control of a nonlinear large-scale fractional-order system. Soft Comput 27(19):14063–14071. https://doi.org/10.1007/s00500-023-07922-6

Joseph FJJ, Nonsiri S, Monsakul A (2021) Keras and tensorflow: A hands-on experience. In: A Practical Approach, Advanced Deep Learning for Engineers and Scientists, pp. 85–111

Katz G, Barrett CW, Dill DL, et al (2017) Reluplex: an efficient SMT solver for verifying deep neural networks. In: Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I, Lecture Notes in Computer Science, vol 10426. Springer, pp 97–117. https://doi.org/10.1007/978-3-319-63387-9_5

Katz G, Huang DA, Ibeling D, et al (2019) 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, Lecture Notes in Computer Science, vol 11561. Springer, pp 443–452, https://doi.org/10.1007/978-3-030-25540-4_26

Kingma DP, Ba J (2015) Adam: a method for stochastic optimization. In: 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings

Kouvaros P, Kyono T, Leofante F, et al (2021) Formal analysis of neural network-based systems in the aircraft domain. In: Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings, Lecture Notes in Computer Science, vol 13047. Springer, pp 730–740. https://doi.org/10.1007/978-3-030-90870-6_41

LeNail A (2019) NN-SVG: publication-ready neural network architecture schematics. J Open Source Softw 4(33):747. https://doi.org/10.21105/JOSS.00747

Leofante F, Henriksen P, Lomuscio A (2023) Verification-friendly networks: the case for parametric relus. In: International Joint Conference on Neural Networks, IJCNN 2023, Gold Coast, Australia, June 18-23, 2023. IEEE, pp 1–9, https://doi.org/10.1109/IJCNN54540.2023.10191169

Müller MN, Brix C, Bak S, et al (2022) The third international verification of neural networks competition (VNN-COMP 2022): summary and results. CoRR abs/2212.10376. https://doi.org/10.48550/ARXIV.2212.10376

Nagisetty V (2021) Domain knowledge guided testing and training of neural networks. Master's thesis, University of Waterloo

Paszke A, Gross S, Massa F, et al (2019) Pytorch: an imperative style, high-performance deep learning library. In: Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada, pp 8024–8035

Pavithra A, Kalpana G, Vigneswaran T (2023) Deep learning-based automated disease detection and classification model for precision agriculture. Soft Comput. https://doi.org/10.1007/s00500-023-07936-0

Pulina L, Tacchella A (2010) An abstraction-refinement approach to verification of artificial neural networks. In: Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings, Lecture Notes in Computer Science, vol 6174. Springer, pp 243–257, https://doi.org/10.1007/978-3-642-14295-6_24

Roeder L (2023) Netron.app. https://netron.app

Sako K, Mpinda BN, Rodrigues PC (2022) Neural networks for financial time series forecasting. Entropy 24(5):657. https://doi.org/10.3390/e24050657

Seide F, Agarwal A (2016) CNTK: microsoft's open-source deep-learning toolkit. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016. ACM, p 2135, https://doi.org/10.1145/2939672.2945397

Singh G, Gehr T, Püschel M, et al (2019a) An abstract domain for certifying neural networks. Proc ACM Program Lang 3(POPL):41:1–41:30. https://doi.org/10.1145/3290354

Singh G, Gehr T, Püschel M, et al (2019b) Boosting robustness certification of neural networks. In: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019. OpenReview.net

Sotoudeh M, Thakur AV (2021) Provable repair of deep neural networks. In: PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021. ACM, pp 588–603, https://doi.org/10.1145/3453483.3454064

Szegedy C, Zaremba W, Sutskever I, et al (2014) Intriguing properties of neural networks. In: 2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings

Tjeng V, Xiao KY, Tedrake R (2019) 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. OpenReview.net

Tran H, Lopez DM, Musau P, et al (2019) Star-based reachability analysis of deep neural networks. In: Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings, Lecture Notes in Computer Science, vol 11800. Springer, pp 670–686, https://doi.org/10.1007/978-3-030-30942-8_39

Tran H, Yang X, Lopez DM, et al (2020) NNV: the neural network verification tool for deep neural networks and learning-enabled cyber-physical systems. In: Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I, Lecture Notes in Computer Science, vol 12224. Springer, pp 3–17, https://doi.org/10.1007/978-3-030-53288-8_1

Wang S, Zhang H, Xu K, et al (2021) Beta-crown: efficient bound propagation with per-neuron split constraints for neural network robustness verification. In: Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual, pp 29,909–29,921

Xu K, Zhang H, Wang S, et al (2021) Fast and complete: enabling complete neural network verification with rapid and massively parallel incomplete verifiers. In: 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021. OpenReview.net

Yang F, Chen J, Liu Y (2023) Improved and optimized recurrent neural network based on PSO and its application in stock price prediction. Soft Comput 27(6):3461–3476. https://doi.org/10.1007/s00500-021-06113-5

Zhang H, Weng T, Chen P, et al (2018) Efficient neural network robustness certification with general activation functions. In: Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada, pp 4944–4953

Zhang H, Wang S, Xu K, et al (2022) General cutting planes for bound-propagation-based neural network verification. In: NeurIPS

Zhang B, Huang W, Zhao F (2023) An available-flow neural network for solving the dynamic groundwater network maximum flow problem. Soft Comput. https://doi.org/10.1007/s00500-023-07912-8

Zheng Y (2019) Computing bounding polytopes of a compact set and related problems in n-dimensional space. Comput Aided Des 109:22–32. https://doi.org/10.1016/J.CAD.2018.12.002