Towards Continuous Constraint Programming for Sound Neural Network Verification

Yi-Nung Tsao[1,*], Pierre Talbot[1]

[1]University of Luxembourg, 2 Av. de l'Universite, 4365 Belval Esch-sur-Alzette

Abstract

Neural networks are susceptible to adversarial examples—inputs with subtle perturbations that trigger erroneous outputs and potentially catastrophic failures. Neural network verification addresses this by formally checking whether given postconditions are entailed by specific preconditions. Most state-of-the-art verifiers rely on abstract interpretation to overapproximate neuron bounds layer by layer, utilizing branch-and-bound methods to achieve completeness. However, a significant gap exists between theoretical soundness and practical implementation: standard floating-point arithmetic introduces rounding errors that accumulate during computation, potentially leading to unsound verification results. While the continuous constraint solving community has long addressed numerical inaccuracies by accounting for floating-point rounding errors, this rigorous treatment is not yet widespread in neural network verification. Our preliminary experimental results demonstrate that unsoundness issues manifest in several state-of-the-art neural network verifiers. To bridge this gap, we propose JET, a GPU-based sound continuous constraint solver. JET integrates sound floating-point interval arithmetic with constraint propagation and a search method, providing a framework that guarantees numerical soundness while leveraging the parallel performance on GPU.

Keywords

Neural network verification, continuous constraint satisfaction problem, constraint programming, abstract interpretation, GPU

1. Introduction

Neural networks are vulnerable to *adversarial examples* [1, 2]. These are inputs that are initially correctly classified, but their predictions will be changed (misclassifications) when they are perturbed with slight noise that is imperceptible to humans. For example, in an autonomous car system, a neural network can correctly identify a clean stop sign on the road, yet fail to classify it accurately if there is a specific sticker pattern on it or if the brightness differs from the training data set [3]. Such misclassifications can lead to critical errors and potentially result in accidents that endanger the driver and other road users. Given a certain input perturbated space, verifying if a neural network can consistently produce the expected results is therefore a crucial and necessary step before deploying it in safety-critical applications. This step is called *neural network verification*.

Although various efficient neural network verifiers [4, 5, 6, 7] claim theoretical soundness and completeness, few rigorously address the intricacies of floating-point arithmetic. Under the IEEE-754 standard [8, 9], a *floating-point number* is represented by three components: a sign bit, an exponent, and a significand (or mantissa). The precision is determined by the bit-width of these components; for instance, single-precision (32-bit) uses 8 bits for the exponent and 23 for the significand, while double-precision (64-bit) allocates 11 and 52 bits, respectively. While this standard facilitates consistent representation across diverse hardware environments, it inherently introduces rounding errors that can compromise verification reliability.

The primary issue is that most real numbers—including simple decimals such as 0.1 and 0.2 or irrational numbers—cannot be exactly expressed within this finite representation [8]. Consequently,

LOGICNN 2026: Workshop on Logical Methods for Neural Network Analysis, July 25, 2026, Lisbon, Portugal
*Corresponding author.
yi-nung.tsao@uni.lu (Y. Tsao); pierre.talbot@uni.lu (P. Talbot)
https://ytsao.github.io/ (Y. Tsao); https://ptal.github.io/ (P. Talbot)
0009-0001-4488-0236 (Y. Tsao); 0000-0001-9202-4541 (P. Talbot)

floating-point arithmetic exhibits critical limitations: first, it does not satisfy the *associative law* (e.g., $(0.1+0.2)+0.3 \neq 0.1+(0.2+0.3)$); second, arithmetic outcomes often vary across different precisions due to the distinct accumulation of rounding errors.

The critical nature of floating-point arithmetic in neural network verification has recently been highlighted by [10]. Indeed, some neural network verifiers might wrongly conclude the safety of a neural network w.r.t. the preconditions (Section 4). *Soundness* is the absence of such behavior. Further emphasizing this concern, the introduction of SoundBench [11] provides a dedicated benchmark to validate the correctness of verifiers. Despite this growing awareness, many existing tools still suffer from incorrect floating-point implementations, potentially leading to catastrophic consequences in safety-critical applications. To date, no solution has been proposed for this issue.

To mitigate these numerical issues, we propose to investigate continuous constraint programming—a rigorous solving method over possibly nonlinear continuous constraints—for neural network verification. The main idea is to overapproximate a real number r using an interval $[\ell, u]$ such that $r \in [\ell, u]$, where ℓ and u are the greatest floating-point number less than r and the smallest floating-point number greater than r, respectively. To ensure soundness during computation, we explicitly account for *floating-point rounding errors* by employing directed rounding modes: the lower bound is calculated using downward rounding, while the upper bound uses upward rounding. This approach is a cornerstone of several interval-based continuous constraint solvers, such as IBEX [12] and RealPaver [13]. Generally, these solvers integrate *rounded interval arithmetic* and *constraint propagation* with systematic search to explore more solutions.

While these techniques are standard for solving continuous constraint satisfaction problems (CCSPs), few neural network verifiers rigorously account for floating-point rounding errors during computation. We note that while certain frameworks [14, 15, 16] employ similar overapproximation techniques via directed rounding on both CPU and GPU, they rely on bound propagation, which is an *incomplete method*. To achieve completeness, these tools typically interface with MILP solvers, which may compromise floating-point soundness (Section 4). Although PyRAT [6] is among the few verifiers that incorporate directed rounding, its GPU backend relies on the PyTorch library [17], which does not support controlled floating-point rounding modes. Consequently, the GPU-based extension of PyRAT lacks the numerical soundness guarantees provided by its CPU implementation.

Another distinct advantage of interval-based continuous constraint solvers is that, unlike conventional modeling in mixed-integer linear programming (MILP) [18], they do not require auxiliary binary variables to represent piecewise linear functions (Section 3). This formulation offers a more concise representation and naturally extends to various nonlinear activation functions.

Furthermore, recent verification studies have demonstrated that taking the network's output constraints into account can further tighten the bounds of each neuron in the network such that the verification precision can be significantly improved [19, 20]. Our framework inherently facilitates this by leveraging a combination of *forward and backward propagation* (Section 3). Therefore, the network's output constraints are implicitly and automatically integrated into the verification process.

To bridge the gap between theoretical soundness and practical floating-point implementation, we introduce JET, the first sound GPU-based continuous constraint satisfaction solver. JET extends Turbo, a GPU-based discrete constraint solver [21, 22], by supporting continuous domains and constraints for neural network verification. By formulating verification as a CCSP, JET integrates sound interval arithmetic and constraint propagation within a systematic search framework, fully leveraging GPU parallelism while maintaining rigorous numerical guarantees.

The main contributions of this work are as follows:

- We propose JET, a sound continuous constraint satisfaction solver for neural network verification.
- We evaluate our approach on the SAT-ReLU benchmark from VNN-COMP 2025 [23], demonstrating that JET consistently provides numerically sound verification results.
- Our analysis of these preliminary results reveals that neglecting floating-point rounding errors can lead to unsound verification conclusions in state-of-the-art verifiers, underscoring the necessity of our approach.

2. Background

2.1. Neural Network Verification

Neural network verification consists of three fundamental components that we define in turn: *neural network*, *preconditions*, and *postconditions*.

Neural networks A neural network N is composed of an input layer, multiple hidden layers, and an output layer where each layer is made up of several neurons. Each layer ℓ is a function $N_\ell \colon \mathbb{R}^{n_\ell} \to \mathbb{R}^{n_{\ell+1}}$, where n_ℓ is the number of neurons in the layer ℓ, defined as $N_\ell(\mathbf{x}) \triangleq \sigma(\mathbf{W}_\ell \mathbf{x} + \mathbf{b}_\ell)$. It processes a vector $\mathbf{x}$ according to a trained weight matrix $\mathbf{W}_\ell$, a trained bias vector $\mathbf{b}_\ell$ and an activation function σ. In general, σ can be any nonlinear function, but we focus on Rectified Linear Unit (ReLU) activation function, defined as $\sigma(\mathbf{W}_\ell \mathbf{x} + \mathbf{b}_\ell) \triangleq \max(0, \mathbf{W}_\ell \mathbf{x} + \mathbf{b}_\ell)$. It is the most commonly used activation function in deep learning [24, 25].

We denote by $\mathbf{x}_0$ and $\mathbf{y}$ the original input data vector and the output vector produced by the neural network and let L be the index set of layers. A neural network is a function $N \colon \mathbb{R}^{n_1} \to \mathbb{R}^{n_{|L|}}$ which can be obtained by functional composition of layer functions N_ℓ as follows:

$$N \triangleq N_{|L|} \circ N_{|L|-1} \circ \cdots \circ N_1$$
$$N(\mathbf{x}_0) = \mathbf{y} = N_{|L|}(N_{|L|-1}(\ldots N_1(\mathbf{x}_0)))$$

where $\mathbf{x}_0 \in \mathbb{R}^{n_1}$ and $\mathbf{y} \in \mathbb{R}^{n_{|L|}}$.

Preconditions The preconditions are defined in the input layer by a set $\mathbf{\Phi}(\mathbf{x}_0, \epsilon) \triangleq \{\mathbf{x} \in \mathbb{R}^{n_1} \mid p(\mathbf{x}, \mathbf{x}_0) \leq \epsilon\}$, where $p \colon \mathbb{R}^{n_1} \times \mathbb{R}^{n_1} \to \mathbb{R}$ is a function defining a perturbation and $\epsilon \in \mathbb{R}$ is the maximum allowable perturbation. The perturbation function can be different, depending on the application. For example, in the robustness analysis of neural network, the L_∞-norm distance is the most widely adopted perturbation function, where $\mathbf{\Phi}(\mathbf{x}_0, \epsilon) = \{x \mid \|\mathbf{x} - \mathbf{x}_0\|_\infty \leq \epsilon\}$ [14, 15, 26, 27]. Furthermore, in image classification tasks, perturbations may involve geometric transformations, such as rotation, defined by the rotation angle relative to the original image [28]. For autonomous aircraft applications (e.g., ACAS Xu [29]), preconditions are typically defined as specific regions in the input space to simulate aircraft behavior across diverse environments.

Postconditions Let $y_i = N(\mathbf{x}_0)_i$ be the output value of the i-th neuron in the output layer. The postconditions $\mathbf{\Psi}$ are defined as a set of predicates over the output layer, representing the expected behavior of the network. For simplicity, we introduce an *auxiliary layer* immediately following the output layer. The number of neurons in this auxiliary layer corresponds to the number of predicates in $\mathbf{\Psi}$, with the weights assigned as the coefficients of each predicate. Consequently, the verification task reduces to checking whether all outputs of the auxiliary layer are strictly positive.

For instance, in a classification task with three classes $\{0, 1, 2\}$ and a ground-truth label of 0, the postconditions can be formulated as $\mathbf{\Psi} \triangleq \{a_0 > 0 \wedge a_1 > 0 \mid a_0 = y_0 - y_1, a_1 = y_0 - y_2\}$.

Neural Network Verification The neural network verification problem is formally defined as:

$$\exists \mathbf{x} \in \mathbf{\Phi}(\mathbf{x}_0, \epsilon) \colon N(\mathbf{x}) \models \neg \mathbf{\Psi} \tag{1}$$

If there exists a perturbed input vector $\mathbf{x} \in \mathbf{\Phi}(\mathbf{x}_0, \epsilon)$ such that the output of N satisfies the negated predicates $\neg\mathbf{\Psi}$ (SAT), then $\mathbf{x}$ is identified as an *adversarial example* (or *counterexample*). If no such input exists, the network is proven safe (UNSAT) within the specified conditions. The complexity of neural network verification has been proven to be NP-complete [29].

2.2. Continuous Constraint Programming

Let X be a finite set of variables and C be a finite set of constraints. A *constraint network* is defined as a pair $\langle d, C\rangle$, where $d \in X \to \mathcal{P}(\mathbb{R})$ is a *domain function* that maps each variable to its corresponding domain. Internally, the constraint network is ternarized in JET to better align with the GPU architecture [22, 30]. The set of all domain functions are denoted by D, which is ordered pointwise ($d \leq d' \Leftrightarrow$ for each $x \in X, d(x) \subseteq d'(x)$). An assignment is a mapping $asn \in X \to \mathbb{R}$, and $\mathbf{Asn}$ denotes the set of all possible assignments. The set of solutions satisfying a constraint $c \in C$ is given by $rel(c) \subseteq \mathbf{Asn}$. For instance, we have $rel(x = y + 1.5) \triangleq \{asn \in \mathbf{Asn} \mid asn(x) = asn(y) + 1.5\}$. Hence, the solution set sol for a constraint network $\langle d, C\rangle$ is defined as:

$$sol(d, C) \triangleq \{asn \in \mathbf{Asn} \mid \text{ for each } x \in X, asn(x) \in d(x) \text{ and for each } c \in C, asn \in rel(c)\}. \quad (2)$$

Overapproximation and Underapproximation It is impossible to identify $sol(d, C)$ by enumerating all assignments in a constraint network $\langle d, C\rangle$. Because assignments are drawn from $\mathcal{P}(\mathbb{R})$, the set of all potential assignments is infinite. To address this, we follow the framework of *abstract interpretation* [31] and rely on an abstract set $\mathbb{A}$ overapproximating the set of solutions. The abstraction is connected to the set of solutions through a monotone *concretization function* $\gamma : \mathbb{A} \to \mathcal{P}(\mathbf{Asn})$. Soundness is the mathematical property guaranteeing that the overapproximation of the set of solutions. An element $a \in \mathbb{A}$ is soundly approximating the solutions' set $sol(d, C)$ iff $\gamma(a) \supseteq sol(d, C)$. It guarantees that a preserves all solutions of the constraint problem; however, it may introduce *spurious solutions*—elements in the abstract set $\mathbb{A}$ that do not satisfy the original constraints. Proving the soundness of a verifier amounts to demonstrating that every abstract element a soundly approximate the solutions of the constraint network. We illustrate these definitions with the *interval abstract domain*, the abstraction chosen in this work.

Interval Abstract Domain Due to the inherent limitations of finite-precision floating-point representations, computers cannot exactly represent all real numbers. We employ *floating-point intervals* to overapproximate real numbers. Let $\mathbb{F}$ be the set of finite floating-point numbers representable under the IEEE-754 standard [9]. Notably, we exclude the NaN (Not-a-Number) symbol from $\mathbb{F}$, as it lacks a total ordering with respect to other floating-point numbers. We denote by MIN_FP and MAX_FP the least and greatest representable values in $\mathbb{F}$, respectively. Furthermore, we define the extended set $\mathbb{F}^\infty = \mathbb{F} \cup \{-\infty, \infty\}$, where $-\infty$ and ∞ serve as the lower and upper bounds, such that $\min \mathbb{F}^\infty = -\infty$ and $\max \mathbb{F}^\infty = \infty$.

A real number r is overapproximated by a floating-point interval $[\underline{r}, \bar{r}]$, where $\underline{r}, \bar{r} \in \mathbb{F}^\infty$ and $r \in [\underline{r}, \bar{r}]$. To obtain the most precise overapproximation, $\underline{r}$ must be the greatest floating-point number less than or equal to r, while $\bar{r}$ must be the least floating-point number greater than or equal to r. If $r <$ MIN_FP (or $r >$ MAX_FP), the resulting interval is defined as $[-\infty,$ MIN_FP$]$ (or $[$MAX_FP$, +\infty]$).

Definition 1. *[32] The set of floating-point intervals I is a complete lattice $\langle I, \sqsubseteq, \sqcap, \sqcup, \bot\rangle$, where $I = \{[\ell, u] \mid \ell \in \mathbb{F} \cup \{-\infty\}, u \in \mathbb{F}\{+\infty\}, \ell \leq u\}$. Let $[\ell, u], [\ell', u'] \in I$, the lattice operators are defined as follows:*

- $[\ell, u] \sqsubseteq [\ell', u'] \Leftrightarrow \ell \geq \ell' \wedge u \leq u'$,
- $[\ell, u] \sqcap [\ell', u'] \triangleq \begin{cases} [\max\{\ell, \ell'\}, \min\{u, u'\}] & \text{if } \max\{\ell, \ell'\} \leq \min\{u, u'\} \\ \bot & \text{otherwise} \end{cases}$,
- $[\ell, u] \sqcup [\ell', u'] \triangleq [\min\{\ell, \ell'\}, \max\{u, u'\}]$,
- $\gamma(a) \triangleq \begin{cases} \{\} & \text{if } a = \bot \\ \{r \in \mathbb{R} \mid \ell \leq r \leq u\} & \text{if } a = [\ell, u] \end{cases}$.

The greatest element can be represented as the interval $[-\infty, +\infty]$, hence there is no need to add a special top element $\top$. Note that when at least one argument is $\bot$, for each $[\ell, u] \in I, \bot \sqsubseteq [\ell, u]$; $\bot$ is neutral for $\sqcup$ and absorbing for $\sqcap$.

Soundness of Floating-Point Interval Arithmetic To ensure soundness, we account for floating-point rounding errors by performing interval operations with directed rounding. Following standard conventions for extended floating-point arithmetic $\mathbb{F}^{\infty}$, we define: $\infty + \infty = \infty$, $\infty + c = \infty$, $c + \infty = \infty$, $-\infty + c = -\infty$, $c + (-\infty) = -\infty$, and $-\infty + (-\infty) = -\infty$. Let $[\ell, u], [\ell', u'], \bot \in \mathbb{I}$. The sound interval arithmetic operators, incorporating the $\max$ operator for ReLU activations, are defined below in accordance with, e.g., [33].

For all operations $\diamond \in \{\oplus, \ominus, \otimes\}$, $[\ell, u] \diamond \bot = \bot \diamond [\ell, u] = \bot$.

$$
\begin{aligned}
[\ell, u] \oplus [\ell', u'] &= [\underline{\ell + \ell'}, \overline{u + u'}] \\
\ominus [\ell, u] &= [-u, -\ell] \\
[\ell, u] \ominus [\ell', u'] &= [\underline{\ell - u'}, \overline{u - \ell'}] \\
[\ell, u] \otimes [\ell', u'] &= [\min\{\underline{\ell\ell'}, \underline{\ell u'}, \underline{u\ell'}, \underline{uu'}\}, \max\{\overline{\ell\ell'}, \overline{\ell u'}, \overline{u\ell'}, \overline{uu'}\}] \\
[\ell, u] \oslash [\ell', u'] &\triangleq \begin{cases} [\min\{\underline{\ell/\ell'}, \underline{\ell/u'}, \underline{u/\ell'}, \underline{u/u'}\}, \max\{\overline{\ell/\ell'}, \overline{\ell/u'}, \overline{u/\ell'}, \overline{u/u'}\}] & 0 \notin [\ell', u'] \\ [-\infty, \infty] & \text{otherwise} \end{cases} \\
\max([\ell, u], [\ell', u']) &= [\max\{\ell, \ell'\}, \max\{u, u'\}], \quad \max([\ell, u], \bot) = [\ell, u]
\end{aligned}
$$

Definition 2. *The interval abstract domain $\langle \mathbb{A}_I, \dot{\sqsubseteq} \rangle$ is a complete lattice $\langle X \to I, \dot{\sqsubseteq}, \dot{\sqcap}, \dot{\sqcup}, \dot{\bot} \rangle$, where all operators are lifted pointwise from $\langle I, \sqsubseteq, \sqcap, \sqcup, \bot \rangle$. Let $d, d' \in X \to I$, the pointwise lattice operators and special elements are defined as follows:*

- $d \dot{\sqsubseteq} d' \Leftrightarrow$ *for each* $x \in X : d(x) \sqsubseteq d'(x)$,
- $d \dot{\sqcap} d' \triangleq x \in X \mapsto d(x) \sqcap d'(x)$,
- $d \dot{\sqcup} d' \triangleq x \in X \mapsto d(x) \sqcup d'(x)$,
- $\dot{\bot} \triangleq x \in X \mapsto \bot$,
- $\dot{\gamma}(d) \triangleq \{asn \in X \to \mathbb{R} \mid$ *for each* $x \in X, asn(x) \in \gamma(d(x))\}$.

By Definition 2, each variable is abstracted as a floating-point interval. The Cartesian product of each floating-point interval is called a *box*. The precision (width) of an interval $[\ell, u]$ is computed as $\overline{u - \ell}$. By employing upward rounding, we ensure that the computed width is a safe overapproximation, thereby guaranteeing that the algorithm only terminates when the true interval width is strictly within the precision ϵ. Accordingly, the precision of a box is determined by the maximum width among its intervals. In practice, a predefined precision ϵ is employed to facilitate algorithmic convergence; in this study, we set $\epsilon = 10^{-6}$ for the evaluations in Section 4.

To characterize the solution space, we categorize boxes into three distinct types based on their solution status:

- *solution box*: a box where every point is guaranteed to be a solution,
- *boundary box*: a box containing both solution and non-solution points, typically arising in the presence of inequalities,
- *unknown box*: a box where no solution can be definitively identified, yet which is pruned from the search tree because its width has reached the precision ϵ.

Based on these definitions and arithmetic rules, we formally define the *sound floating-point interval propagator* for each constraint $c \in C$ in Definition 3, followed by an illustrative example.

Definition 3. *A floating-point propagator for a constraint $c \in C$ is a function $p_c \in \mathbb{A}_I \to \mathbb{A}_I$ such that for each $a \in \mathbb{A}_I, p_c(a)$ respects the following properties:*

- *reductive:* $p_c(a) \dot{\sqsubseteq} a$,
- *monotone:* $a \dot{\sqsubseteq} a' \Rightarrow p_c(a) \dot{\sqsubseteq} p_c(a')$,
- *soundness:* $sol(a, \{c\}) \subseteq sol(p_c(a), \{\})$.

For example, consider the rounded floating-point arithmetic above and a constraint c defined as $x = y + z$. For any $a \in \mathbb{A}_I$, the corresponding propagator $p_c(a)$ is defined as:

$$
\begin{aligned}
&p_c(a) \triangleq a \mathbin{\dot{\sqcap}} b \\
&\quad \text{where} \;\; \text{for each } w \in X \setminus \{x, y, z\},\; b(w) = a(w) \text{ and} \\
&\qquad b(x) = a(y) \oplus a(z) \\
&\qquad b(y) = a(x) \ominus a(z) \\
&\qquad b(z) = a(x) \ominus a(y)
\end{aligned}
$$

Similarly, other propagators are defined by applying sound floating-point interval arithmetic rules specified in, e.g. [34].

3. JET: A Novel Neural Network Verification Tool

JET is designed as a general-purpose solver supporting continuous domains. In this study, we specifically investigate its application and efficacy in the context of neural network verification. This section presents a detailed overview of JET and its verification workflow, as illustrated in Fig. 1.

The verification process takes a neural network along with its specified preconditions and postconditions as input. Given these inputs, we first describe the reformulation of the neural network verification problem (Eq. (1)) into a CCSP (Eq. (3)). Subsequently, we introduce a solving algorithm, *branch-and-prune algorithm* based on the interval abstract domain and sound floating-point propagation. While these core components are well-established in traditional interval-based solvers [12, 13], JET uniquely integrates them into a parallel GPU-based architecture. Developed as an extension of Turbo [21, 22]—a state-of-the-art GPU-based discrete constraint solver.

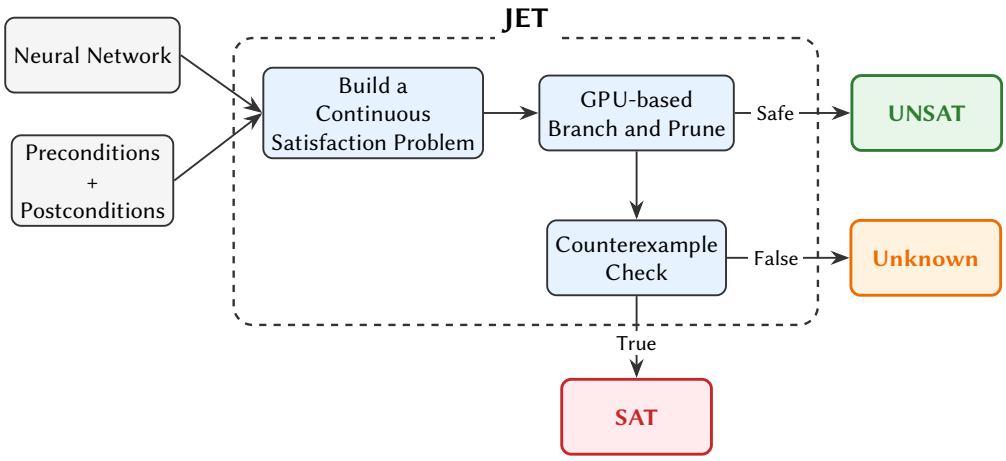

Figure 1: The overview of JET for neural network verification.

A CCSP Model of Neural Network Verification In this section, we decompose the network layers (excluding the input and output layers[1]) into distinct functional components. Specifically, we define two index sets to identify affine layers, L^A, and ReLU layers, L^R, respectively. It follows that $L = L^A \cup L^R$, where auxiliary layers are included in L^A. Furthermore, each ReLU layer is succeeded by an affine layer, i.e., for each $\ell \in L^R$: $\ell + 1 \in L^A$.

The advantage of this structural decomposition is that each layer represents a single function, rather than the nested functional structure defined in Section 2.1. Additionally, this modification aligns with the graph structure used in onnx [35], a standard neural network exchange format. Similar architectural representations are widely adopted by other neural network verification tools [15, 16, 14, 36, 37].

[1]The input and output layers are not considered compositional functions.

To verify neural networks by solving CCSPs, we first reformulate Eq. (1) into the CCSP model (Eq. (3)) presented below. Leveraging the notation introduced above, the affine and ReLU layers are formally defined by the first two equations in Eq. (3). The third equation specifies that layer $|L|$ is the output layer; thus, an identity relation is established between the outputs $\mathbf{x}^{|L|}$ of the final layer and the output variables $\mathbf{y}$. Furthermore, the preconditions $\mathbf{\Phi}(\mathbf{x}_0, \epsilon)$ and negated postconditions $\neg\mathbf{\Psi}$[2] are defined over the input vector $\mathbf{x}_0$ and the output variables $\mathbf{y}$, respectively. Finally, all variables are constrained within continuous domains, as specified in the last two relations of the formulation.

$$
\begin{aligned}
\mathbf{x}_{\ell+1} &= \mathbf{W}_\ell \mathbf{x}_\ell + \mathbf{b}_\ell && \text{for each } \ell \in L^A \\
\mathbf{x}_{\ell+1} &= \max(0, \mathbf{x}_\ell) && \text{for each } \ell \in L^R \\
\mathbf{y} &= \mathbf{x}^{|L|} \\
\mathbf{x}_1 &\in \mathbf{\Phi}(\mathbf{x}_0, \epsilon) \\
\textstyle\bigvee_{i=1}^{m} \mathbf{W}_{|L|}[i,:] \cdot \mathbf{y} &\leq 0 \\
\mathbf{x}_\ell &\in \mathbb{R} && \text{for each } \ell \in L \\
\mathbf{y} &\in \mathbb{R}
\end{aligned}
\tag{3}
$$

Consequently, the verification task reduces to determining the feasibility of the CCSP model (Eq. (3)). A feasible solution to the CCSP constitutes a valid counterexample, yielding a SAT result for the neural network verification problem (Eq. (1)). On the other hand, if the CCSP is proven infeasible, the verification result is UNSAT, guaranteeing that the network satisfies the specification.

The primary benefit of adopting the CCSP formulation (3) is that it eliminates the need for auxiliary binary variables to represent ReLU activation functions. Instead, each neuron in the ReLU layer is modeled directly via a pointwise max function. This modeling approach is highly versatile, seamlessly extending to other general nonlinear activation functions, such as sigmoid and tanh, by directly incorporating their respective nonlinear functions.

Forward and Backward Propagation By applying the sound floating-point propagators defined in Definition (3), we can iteratively tighten the bounds of each variable in the CCSP model (Eq. (3)). This process is performed via forward and backward propagation. Unlike most existing symbolic bound propagation methods [15, 38, 14, 16, 36, 37, 39, 40, 26], which rely solely on *forward propagation* and ignore the information from the negated postconditions $\neg\mathbf{\Psi}$, our approach propagates constraints in both directions to maximize bound reduction.

Branch-and-Prune Algorithm Since floating-point interval propagation is a *sound but incomplete* method, we combine it with a *branch-and-prune algorithm* (BPA) to explore more solutions. BPA is a depth-first search algorithm that exploring the state-space. The solving procedure includes sound floating-point propagation to safely prune variable domains, search steps by bisecting a domain with the largest width to generate subproblems, and an underapproximation (Algorithm 2) to determine whether a solution exists within a box under a given precision ϵ. The details of the BPA are presented in Algorithm 1.

The proposed BPA requires an initial box $\mathbf{x}$ (representing an overapproximation of the problem), a constraint set C, the postconditions $\mathbf{\Psi}$, and a precision ϵ. Although $\mathbf{\Psi}$ is included within C for the propagation step, we need it in the underapproximation step.

Initially, we create a stack Q for unexplored search tree nodes, a set B for boundary boxes, and a set U for unknown boxes in Line 2. Given that the CCSP model (Eq.(3)) includes both equations and inequalities, verifying whether all points within a continuous box are solutions remains non-trivial; this complexity precludes the explicit consideration of solution boxes in our approach.

[2]Suppose the postconditions $\mathbf{\Psi} \triangleq \{\bigwedge_{i=1}^{m} \mathbf{W}_{|L|}[i,:] \cdot \mathbf{y} > 0\}$, where m denotes number of rows in $\mathbf{W}_{|L|}$ and $\mathbf{W}_{|L|}[i,:]$ represents the i-th row in $\mathbf{W}_{|L|}$.

Because the compositional and deterministic structure of neural networks, it is unnecessary to branch on all variables within the CCSP model (Eq. (3)). Instead, branching can be strategically restricted to the input variables only, as the values of all subsequent layers are uniquely determined by the network's relational constraints. To evaluate the status of a current box $\mathbf{q}$, an *underapproximation* method (Algorithm 2) is invoked to extract its midpoint $\mathbf{m}$ and check its feasibility. Because the CCSP model (Eq. (3)) embeds the negated postconditions $\neg\mathbf{\Psi}$, identifying $\mathbf{m}$ as a valid solution directly yields a counterexample for the original neural network verification problem (Eq. (1)). Subsequently, the current box $\mathbf{q}$ is classified into the boundary box set B, and the algorithm returns SAT in Line 8. Otherwise, if $\mathbf{m}$ is not a solution, the algorithm categorizes it into the unknown box set U if its maximum width drops below or equal to ϵ in Line 10 or continues to branch on $\mathbf{q}$ in Line 12.

Upon completing the search, if the set of unknown boxes U remains empty—meaning that all explored boxes have been proven infeasible and pruned—the network is formally proven safe (UNSAT) under the given conditions. Otherwise, if U is non-empty and no counterexample was found, the presence of a counterexample remains inconclusive, and the algorithm returns UNKNOWN. In such scenarios, users can increase the precision (by setting a smaller ϵ) to achieve a more granular evaluation, thereby resolving the inconclusiveness of the verification outcome. Notably, the discovery of the first counterexample triggers the immediate termination of Algorithm 1; in contrast, standard BPAs aim to exhaustively identify all solutions within the search space.

Overall, given this sound solving procedure, if there exists an abstract element a such that $\gamma(a) = \{\}$, the problem is then unsatisfiable, implying that the network satisfies the properties under considerations (UNSAT). In the opposite case ($\gamma(a) \neq \{\}$), we cannot reach any conclusion. To improve this situation, we shall attempt to extract an *underapproximation* u from a such that $\gamma(a) \supseteq \gamma(u)$ and $\gamma(u) \subseteq sol(d, C)$. In our proposed framework, a non-empty underapproximation ($\gamma(u) \neq \{\}$) is a counterexample (SAT).

Algorithm 1 Branch-and-Prune Algorithm for Neural Network Verification

1: **function** $search(\mathbf{x}, C, \epsilon)$
2: $Q \leftarrow \{\mathbf{x}\}, B \leftarrow \{\}, U \leftarrow \{\}$ ▷ *Initialization*
3: **while** $Q \neq \{\}$ **do**
4: $\mathbf{q} \leftarrow pop(Q)$ ▷ *Select the next subproblem*
5: $propagate(\mathbf{q}, C)$ ▷ *Sound floating-point propagation*
6: **if** $underapproximate(\mathbf{q}, \mathbf{\Psi})$ **then** ▷ *Algorithm 2*
7: $B = B \cup \{\mathbf{q}\}$
8: **return** SAT ▷ *Early termination upon finding a counterexample*
9: **else if** $width(\mathbf{q}) \leq \epsilon$ **then**
10: $U = U \cup \{\mathbf{q}\}$
11: **else**
12: $Q = Q \cup branch(\mathbf{q}, \epsilon)$ ▷ *Generate 2 subproblems in the node list Q*
13: **end if**
14: **end while**
15: **return** $U = \{\}$? UNSAT : UNKNOWN
16: **end function**

Algorithm 2 Midpoint Extraction (Underapproximation)

1: **function** $underapproximate(\mathbf{q}, \mathbf{\Psi})$
2: $\mathbf{m} = extract(\mathbf{q})$ ▷ *Extract the midpoint for each floating-point interval*
3: **if** $N(\mathbf{m}) \models \neg\mathbf{\Psi}$ **then**
4: **return true**
5: **end if**
6: **return false**
7: **end function**

4. Preliminary Experimental Results

4.1. Soundness Issues in MILP Solver

First, we demonstrate that the MILP solver Gurobi has a soundness issue; but similar issues occur in other MILP solvers. In our preliminary experiments, we compare Gurobi (version 12.0.3) [41] with IBEX (version 2.9.1.0) [12], an interval-based constraint solver. The experiment is conducted on a machine equipped with a 2.3GHz 6-core AMD Ryzen 5 PRO 5650U CPU and 16 GB of memory, with both solvers using their default parameter settings.

In our offline test, we find an instance (Kin1.bch as described below) provided by the IBEX team. Since the original problem is in the .bch format—which is incompatible with Gurobi—we reimplemented the problem using the Gurobi C++ API to ensure a direct and fair comparison.

$$
\begin{aligned}
& t_i \in [0, 2\pi], \quad i = 1, \ldots, 6 \\
& (-0.4077) + \cos(t_2)\cos(t_6) + \cos(t_3)\cos(t_6) + \cos(t_4)\cos(t_6) + \cos(t_5)\sin(t_2)\sin(t_6) \\
& \quad - \cos(t_5)\sin(t_3)\sin(t_6) - \cos(t_5)\sin(t_4)\sin(t_6) = 0 \\
& (-1.9115) + \cos(t_5)\sin(t_1) + \cos(t_1)\cos(t_2)\sin(t_5) + \cos(t_1)\cos(t_3)\sin(t_5) \\
& \quad + \cos(t_1)\cos(t_4)\sin(t_5) = 0 \\
& (-1.9791) + \sin(t_2)\sin(t_5) + \sin(t_3)\sin(t_5) + \sin(t_4)\sin(t_5) = 0 \\
& (-4.0616) + 3\cos(t_1)\cos(t_2) + 2\cos(t_1)\cos(t_3) + \cos(t_1)\cos(t_4) = 0 \\
& (-1.7172) + 3\cos(t_2)\sin(t_1) + 2\cos(t_3)\sin(t_1) + \cos(t_4)\sin(t_1) = 0 \\
& (-3.9701) + 3\sin(t_2) + 2\sin(t_3) + \sin(t_4) = 0
\end{aligned}
$$

We present the solutions using 15-digit precision. The solution obtained from Gurobi is as follows:

$$
\begin{aligned}
Sol_{\texttt{gurobi}} = \{ & t_1 \mapsto 3.541589115876881, t_2 \mapsto 2.528671886043\mathbf{263}, t_3 \mapsto 2.200031289188\mathbf{270}, \\
& t_4 \mapsto 2.463425269084\mathbf{723}, t_5 \mapsto 1.392156881933631, t_6 \mapsto 1.827868726794066\}.
\end{aligned}
$$

Due to its interval-based solving strategy, IBEX returns 16 solution boxes. We select the box most comparable to $Sol_{\texttt{gurobi}}$ for evaluation:

$$
\begin{aligned}
& Sol_{\texttt{ibex}} = \{ \\
& t_1 \mapsto [3.541589115876879, 3.541589115876882], t_2 \mapsto [2.528671886043\mathbf{245}, 2.528671886043\mathbf{252}], \\
& t_3 \mapsto [2.200031289188\mathbf{303}, 2.200031289188\mathbf{314}], t_4 \mapsto [2.463425269084\mathbf{697}, 2.463425269084\mathbf{708}], \\
& t_5 \mapsto [1.392156881933624, 1.392156881933633], t_6 \mapsto [1.827868726794062, 1.827868726794066]\}.
\end{aligned}
$$

Critically, we observe that for each $i \in \{2, 3, 4\}$, $Sol_{\texttt{gurobi}}(t_i) \notin Sol_{\texttt{ibex}}(t_i)$. Theoretically, each solution box produced by IBEX is guaranteed to contain a true solution. This discrepancy indicates that, due to the inherent limitations of standard floating-point arithmetic in MILP solvers, Gurobi cannot guarantee that its solutions are numerically sound.

This specific instance involves nonlinear functions such as sin and cos, which likely contributes to the observed inaccuracy in the MILP solver. Given that neural networks are characterized by highly nonlinear and non-convex functions, ensuring a rigorous guarantee against counterexamples is paramount in the verification field. Without such soundness, the presence or absence of violations remains uncertain. Therefore, we adopt interval-based constraint solving techniques as the foundation for our neural network verification framework.

4.2. SAT-ReLU, VNN-COMP 2025

To validate the soundness of floating-point arithmetic in JET across both CPU and GPU platforms, we conducted experiments using the SAT-ReLU benchmark from international verification of neural networks competition 2025 (VNN-COMP 2025) [23]. SAT-ReLU is specifically designed to verify the

correctness of neural network verification tools, consisting of 100 instances equally divided into 50 SAT and 50 UNSAT cases. Each instance features a neural network with a single hidden layer and one ReLU layer, with the number of input and hidden neurons varying across instances.

For these preliminary experiments, we selected 9 instances across 3 size categories (small, medium, and large). The experiments were performed on a system equipped with dual 14-core Intel Xeon E5-2680v4 CPUs (2.4GHz), 128 GB of memory, and an NVIDIA Tesla V100-SXM2 GPU.

Neural Network Verifier Settings We evaluate the performance of JET against four state-of-the-art neural network verifiers on the selected 9 instances: $\alpha\beta$-CROWN [7, 26], NeuralSAT [5], PyRAT [6], and Marabou [4]. These verifiers utilize diverse constraint solving techniques and abstract domains[3]. It is worth noting that IBEX was excluded from this comparison due to the execution errors (`Segmentation fault`) during the solving phase.

Given their maturity, many of these tools incorporate integrated falsification methods (attacks) that can identify counterexamples during a preprocessing stage, potentially bypassing the core verification process. To ensure a fair comparison focused strictly on constraint solving capabilities, we disabled these attack-based features[4]. A timeout of 5 minutes was set for each instance, encompassing both preprocessing and the overall solving procedure, with all other parameters kept at their default values[5].

Notably, all selected tools support both CPU and GPU execution, with the exception of Marabou, which is CPU-only. In the following, we discuss several key insights and observations derived from the results in Table 1.

Observation 1 Although $\alpha\beta$-CROWN and NeuralSAT verified all instances in the SAT-ReLU benchmark without incorrect results during VNN-COMP 2025 [23], our analysis reveals potential underlying issues. While both tools produce correct results under their competition-specific configurations—which rely on a MILP solver paired with falsification attacks—discrepancies arise when such attacks are disabled. Specifically, we observe inconsistencies when utilizing the default branch-and-bound algorithm in $\alpha\beta$-CROWN.

Numerical stability in floating-point arithmetic remains a long-standing challenge, as evidenced by the inconsistent verification outcomes observed even among state-of-the-art verifiers. Notably, even with the use of a MILP solver, numerical instability remains a concern; as demonstrated in our preliminary experiments, MILP solvers can return unsound solutions. Supporting evidence was observed in the results of NeuralSAT, which utilizes a MILP solver as its linear theory solver. Although it successfully verified 8 instances, it encountered a critical execution error on the final instance—`NotImplementedError: Unknown MIP solver error`—across both CPU and GPU platforms.

Observation 2 In PyRAT, floating-point soundness is exclusively guaranteed in its CPU implementation [6]. Due to the substantial computational overhead associated with frequent switching of hardware rounding modes, PyRAT was able to verify only a single instance within the specified time limit. Furthermore, we observed 2 incorrect verification outcomes when using the GPU version of PyRAT, as it does not currently provide floating-point soundness guarantees on GPU architectures.

Observation 3 While Marabou efficiently identifies counterexamples without relying on falsification attacks, it struggles to prove UNSAT instances within the specified time limits. Notably, although no incorrect results were observed for Marabou in these preliminary experiments, its potential issues regarding soundness and correctness have been noticed in recent literature [11].

[3]The domains include: polyhedra ($\alpha\beta$-CROWN and Marabou), Boolean abstraction (NeuralSAT), and zonotopes (PyRAT).
[4]Marabou does not employ any attack methods.
[5]The CPU implementation of PyRAT is configured to employ sound floating-point arithmetic modes.

Table 1
Verification results from JET compare with $\alpha\beta$-CROWN, NeuralSAT, PyRAT, and Marabou. (Background colors: green: correct, red: wrong, pink: lack of precision, and gray: timeout.)

Instances	$\alpha\beta$-CROWN		NeuralSAT		PyRAT		Marabou	JET	
	CPU	GPU	CPU	GPU	CPU	GPU	CPU	CPU	GPU
small 1	unsat	unsat	sat	sat	timeout	unsat	sat	sat	sat
small 2	unknown	unknown	unsat	unsat	unsat	unsat	timeout	timeout	timeout
small 3	unsat	unknown	sat	sat	timeout	unsat	sat	sat	sat
medium 1	timeout	timeout	sat	sat	timeout	timeout	sat	timeout	timeout
medium 2	timeout	timeout	sat	sat	timeout	timeout	sat	timeout	timeout
medium 3	timeout	timeout	unsat	unsat	timeout	timeout	timeout	timeout	timeout
large 1	timeout	timeout	sat	sat	timeout	timeout	sat	timeout	timeout
large 2	timeout	timeout	sat	sat	timeout	timeout	sat	timeout	timeout
large 3	timeout	timeout	error	error	timeout	timeout	timeout	timeout	timeout

Observation 4 Based on these preliminary and limited experimental results, JET successfully demonstrates its soundness, yielding no incorrect verification outcomes. Furthermore, JET leverages a simplified yet efficient overapproximation strategy to achieve competitive performance against existing tools that rely on more complex abstract domains (e.g., zonotopes or polyhedra) and sophisticated search heuristics. However, for larger instances in this benchmark, JET was unable to reach a conclusive result within the timeout, potentially due to either the substantial expansion of the ternary constraint network or the insufficient precision of the overapproximation method.

5. Conclusion

Summary In this study, we propose JET, a sound GPU-based continuous constraint solver designed to address the floating-point soundness issues in neural network verification. Developed as an extension of Turbo—a GPU-based discrete constraint solver—JET rigorously implements sound floating-point interval arithmetic on both CPU and GPU architectures. Our preliminary experiments demonstrate that JET ensures both soundness and correctness for neural network verification. Furthermore, to the best of our knowledge, JET is the first GPU-based sound continuous solver capable of handling general CCSPs.

Limitations and Next Steps This paper represents our initial step toward leveraging continuous constraint solving techniques for neural network verification. Currently, achieving conclusive results within reasonable time limits remains a significant challenge. We identify two primary bottlenecks: first, the rapid expansion of the ternary constraint network, which substantially hinders both counterexample search and formal safety proofs; and second, the underling overapproximation, which may lack the precision necessary for tighter verification bounds.

To address these challenges, our future work will focus on developing alternative decomposition strategies for affine functions to mitigate the complexity of the constraint network, thereby enhancing overall solving efficiency. Additionally, we plan to transition from the interval abstract domain to more sophisticated relational abstract domains to capture inter-neuron dependencies more effectively. Furthermore, we intend to conduct extensive evaluations across a broader range of neural network verification benchmarks and general CCSPs to further demonstrate the versatility and scalability of JET.

Acknowledgments

We are grateful to the reviewers for the useful comments. This work is supported by the Luxembourg National Research Fund, INTER project ADHOC (INTER/ANR/22/17166749/ADHOC).

Declaration on Generative AI

During the preparation of this work, the author(s) used Gemini (Google) in order to: Grammar and spelling check. After using these tool(s)/service(s), the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

References

[1] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, R. Fergus, Intriguing properties of neural networks, arXiv preprint arXiv:1312.6199 (2013).

[2] I. J. Goodfellow, J. Shlens, C. Szegedy, Explaining and harnessing adversarial examples, arXiv preprint arXiv:1412.6572 (2014).

[3] K. Eykholt, I. Evtimov, E. Fernandes, B. Li, A. Rahmati, C. Xiao, A. Prakash, T. Kohno, D. Song, Robust physical-world attacks on deep learning visual classification, in: Proceedings of the IEEE conference on computer vision and pattern recognition, 2018, pp. 1625–1634.

[4] H. Wu, O. Isac, A. Zeljić, T. Tagomori, M. Daggitt, W. Kokke, I. Refaeli, G. Amir, K. Julian, S. Bassan, et al., Marabou 2.0: a versatile formal analyzer of neural networks, in: International Conference on Computer Aided Verification, Springer, 2024, pp. 249–264.

[5] H. Duong, T. Nguyen, M. B. Dwyer, Neuralsat: A high-performance verification tool for deep neural networks, in: International Conference on Computer Aided Verification, Springer, 2025.

[6] A. Lemesle, J. Lehmann, T. L. Gall, Z. Chihani, Verifying neural networks with pyrat, in: International Static Analysis Symposium, Springer, 2025, pp. 11–33.

[7] S. Wang, H. Zhang, K. Xu, X. Lin, S. Jana, C.-J. Hsieh, Z. Kolter, Beta-crown: efficient bound propagation with per-neuron split constraints for neural network robustness verification, in: Proceedings of the 35th International Conference on Neural Information Processing Systems, NIPS '21, Curran Associates Inc., Red Hook, NY, USA, 2021.

[8] D. Goldberg, What every computer scientist should know about floating-point arithmetic, ACM computing surveys (CSUR) 23 (1991) 5–48.

[9] IEEE, 754-2019 - ieee standard for floating-point arithmetic, 2019. doi:10.1109/IEEESTD.2019.8766229.

[10] A. Szász, B. Bánhelyi, M. Jelasity, No soundness in the real world: On the challenges of the verification of deployed neural networks, arXiv preprint arXiv:2506.01054 (2025).

[11] X. Zhou, K. Shen, A. Xu, H. Xu, C.-J. Hsieh, H. Zhang, Z. Shi, Soundnessbench: A soundness benchmark for neural network verifiers, arXiv preprint arXiv:2412.03154 (2024).

[12] F. Benhamou, L. Granvilliers, Chapter 16 - continuous and interval constraints, in: F. Rossi, P. van Beek, T. Walsh (Eds.), Handbook of Constraint Programming, volume 2 of *Foundations of Artificial Intelligence*, Elsevier, 2006, pp. 571–603. URL: https://www.sciencedirect.com/science/article/pii/S1574652606800209. doi:https://doi.org/10.1016/S1574-6526(06)80020-9.

[13] L. Granvilliers, F. Benhamou, Algorithm 852: Realpaver: an interval solver using constraint satisfaction techniques, ACM Transactions on Mathematical Software (TOMS) 32 (2006) 138–156.

[14] G. Singh, T. Gehr, M. Mirman, M. Püschel, M. Vechev, Fast and effective robustness certification, Advances in neural information processing systems 31 (2018).

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

[16] C. Müller, F. Serre, G. Singh, M. Püschel, M. Vechev, Scaling polyhedral neural network verification on gpus, Proceedings of Machine Learning and Systems 3 (2021) 733–746.

[17] S. Imambi, K. B. Prakash, G. Kanagachidambaresan, Pytorch, in: Programming with TensorFlow: solution for edge computing applications, Springer, 2021, pp. 87–104.

[18] V. Tjeng, K. Xiao, R. Tedrake, Evaluating robustness of neural networks with mixed integer programming, arXiv preprint arXiv:1711.07356 (2017).

[19] H. Wu, C. Barrett, M. Sharif, N. Narodytska, G. Singh, Scalable verification of gnn-based job schedulers, Proceedings of the ACM on Programming Languages 6 (2022) 1036–1065.
[20] S. Kotha, C. Brix, J. Z. Kolter, K. Dvijotham, H. Zhang, Provably bounding neural network preimages, Advances in Neural Information Processing Systems 36 (2023) 80270–80290.
[21] P. Talbot, F. G. Pinel, P. Bouvry, A variant of concurrent constraint programming on gpu, Proceedings of the AAAI Conference on Artificial Intelligence 36 (2022) 3830–3839. URL: https://ojs.aaai.org/index.php/AAAI/article/view/20298. doi:10.1609/aaai.v36i4.20298.
[22] P. Talbot, A GPU-based Constraint Programming Solver, in: Proceedings of the AAAI Conference on Artificial Intelligence, volume 40, 2026, pp. 14331–14341. doi:10.1609/aaai.v40i17.38448.
[23] K. Kaulen, T. Ladner, S. Bak, C. Brix, H. Duong, T. Flinkow, T. T. Johnson, L. Koller, E. Manino, T. H. Nguyen, et al., The 6th international verification of neural networks competition (vnn-comp 2025): Summary and results, arXiv preprint arXiv:2512.19007 (2025).
[24] X. Glorot, A. Bordes, Y. Bengio, Deep sparse rectifier neural networks, in: Proceedings of the fourteenth international conference on artificial intelligence and statistics, JMLR Workshop and Conference Proceedings, 2011, pp. 315–323.
[25] V. Nair, G. E. Hinton, Rectified linear units improve restricted boltzmann machines, in: Proceedings of the 27th international conference on machine learning (ICML-10), 2010, pp. 807–814.
[26] H. Zhang, T.-W. Weng, P.-Y. Chen, C.-J. Hsieh, L. Daniel, Efficient neural network robustness certification with general activation functions, Advances in neural information processing systems 31 (2018).
[27] N. Carlini, D. Wagner, Towards evaluating the robustness of neural networks, in: 2017 ieee symposium on security and privacy (sp), Ieee, 2017, pp. 39–57.
[28] M. Balunovic, M. Baader, G. Singh, T. Gehr, M. Vechev, Certifying geometric robustness of neural networks, Advances in Neural Information Processing Systems 32 (2019).
[29] G. Katz, C. Barrett, D. L. Dill, K. Julian, M. J. Kochenderfer, Reluplex: An efficient smt solver for verifying deep neural networks, in: International conference on computer aided verification, Springer, 2017, pp. 97–117.
[30] P. Talbot, Decomposition and preprocessing of ternary constraint networks, 2025. URL: https://arxiv.org/abs/2511.11872. arXiv:2511.11872.
[31] P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, 1977, pp. 238–252.
[32] A. Miné, Tutorial on static inference of numeric invariants by abstract interpretation, Found. Trends Program. Lang. 4 (2017). URL: https://doi.org/10.1561/2500000034. doi:10.1561/2500000034.
[33] T. Hickey, Q. Ju, M. H. Van Emden, Interval arithmetic: From principles to implementation, J. ACM 48 (2001) 1038–1068. URL: https://doi.org/10.1145/502102.502106. doi:10.1145/502102.502106.
[34] P. Cousot, Principles of abstract interpretation, MIT Press, 2021.
[35] J. Bai, F. Lu, K. Zhang, et al., Onnx: Open neural network exchange, https://github.com/onnx/onnx, 2019.
[36] G. Singh, R. Ganvir, M. Püschel, M. Vechev, Beyond the single neuron convex barrier for neural network certification, Advances in Neural Information Processing Systems 32 (2019).
[37] G. Singh, T. Gehr, M. Püschel, M. Vechev, Boosting robustness certification of neural networks, in: International conference on learning representations, 2019.
[38] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, M. Vechev, Ai2: Safety and robustness certification of neural networks with abstract interpretation, in: 2018 IEEE symposium on security and privacy (SP), IEEE, 2018, pp. 3–18.
[39] S. Wang, K. Pei, J. Whitehouse, J. Yang, S. Jana, Formal security analysis of neural networks using symbolic intervals, in: 27th USENIX Security Symposium (USENIX Security 18), 2018.
[40] S. Wang, K. Pei, J. Whitehouse, J. Yang, S. Jana, Efficient formal safety analysis of neural networks, Advances in neural information processing systems 31 (2018).
[41] Gurobi Optimization, LLC, Gurobi Optimizer Reference Manual, 2026. URL: https://www.gurobi.com.