Towards Continuous Constraint Programming for Sound Neural Network Verification

Yi-Nung Tsao
University of Luxembourg

Pierre Talbot
University of Luxembourg

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.

2012 ACM Subject Classification Computing methodologies → Parallel computing methodologies

Keywords and phrases Neural network verification, continuous constraint satisfaction problems, constraint programming, abstract interpretation, GPU

Digital Object Identifier 10.4230/LIPIcs.CPSATDP.2026.23

1 Introduction

Neural networks are vulnerable to *adversarial examples* [17]. These are inputs that are initially correctly classified, but their predictions will be changed (misclassifications) when they are perturbated 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 [4]. Such misclassifications could lead to critical errors and potentially result in accidents that endanger the driver and other road users. Given a certain input perturbated space, verifying whether 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 [3, 12, 20] claim theoretical soundness and completeness, few rigorously address the intricacies of floating-point arithmetic. Under the IEEE-754 standard [7], a *floating-point number* is represented by three components: a sign bit, an exponent, and a significand. Precision is determined by the bit-width of these components. While this standard ensures consistent representation across diverse hardware environments, it also introduces rounding errors that can compromise verification reliability. The primary issue is that most real numbers cannot be exactly expressed within this finite representation. Consequently, floating-point arithmetic exhibits critical limitations: first, it

does not satisfy the *associative law*; second, arithmetic outcomes often vary across different precisions due to the distinct accumulation of rounding errors.

Because of these floating-point arithmetic limitations, some neural network verifiers might wrongly conclude the safety of a neural network w.r.t. the preconditions and postconditions (Section 4). *Soundness* is the absence of such behavior. Indeed, the critical nature of sound floating-point arithmetic in neural network verification has recently been highlighted by [16, 22]. 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 for sound neural network verification. The main idea is to overapproximate a real number r using an interval $[\ell, u]$ such that $r \in [\ell, u]$. To ensure soundness during computation, we explicitly account for *floating-point rounding errors* by employing directed rounding modes: the lower (upper) bound is calculated using downward (upward) rounding. This approach is a cornerstone of several interval-based continuous constraint solvers, such as `IBEX` [1] and `RealPaver` [5]. Generally, these solvers integrate *rounded interval arithmetic* and *constraint propagation* with systematic search to achieve completeness.

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, 13] 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 mixed-integer linear programming (MILP) solvers, which may compromise floating-point soundness (Section 4). Although `PyRAT` [12] is among the few verifiers that incorporate directed rounding, its GPU backend relies on the `PyTorch` library [8], which does not support controlled 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 MILP [19], 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 incorporating backward analysis significantly enhances verification precision [11], our framework inherently facilitates this by leveraging a combination of *forward* and *backward propagation*. As a result, backward analysis is 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 [18], by supporting continuous domains and constraints for neural network verification. `JET` fully leverages GPU parallelism while maintaining rigorous numerical guarantees by integrating sound interval arithmetic and constraint propagation within a systematic search framework.

The main contributions of this work are as follows:

- We propose a sound continuous constraint satisfaction solver, `JET`, for neural network verification.
- We evaluate our approach on the `SAT-ReLU` benchmark from VNN-COMP 2025 [10], demonstrating that `JET` consistently provides numerically sound verification results.
- Our analysis of the preliminary results reveals that neglecting floating-point rounding errors can lead to unsound conclusions in state-of-the-art verifiers.

2 Background

JET is a general continuous constraint satisfaction solver. To generalize the continuous constraint satisfaction problem (CCSP) formulation for it, let $\mathbb{X}$ be a finite set of continuous variables and $\mathbb{C}$ be a finite set of constraints. A *constraint network*[1] is then defined as a pair $\langle d, \mathbb{C}\rangle$, where $d \in \mathbb{X} \to \mathcal{P}(\mathbb{R})$ is a *domain function* that maps each variable to its corresponding domain. We denote the set of all domain functions by $\mathbb{D}$, which is ordered pointwise ($d \leq d' \Leftrightarrow \forall x \in \mathbb{X}, d(x) \subseteq d'(x)$). An assignment is a mapping $asn \in \mathbb{X} \to \mathbb{R}$, and **Asn** denotes the set of all possible assignments. The set of solutions satisfying a constraint $c \in \mathbb{C}$ is given by $rel(c) \subseteq \mathbf{Asn}$. Consequently, the solution set *sol* for a constraint network $\langle d, \mathbb{C}\rangle$ is defined as: $sol(d, \mathbb{C}) \triangleq \{asn \in \mathbf{Asn} \mid \forall x \in \mathbb{X}, asn(x) \in d(x) \wedge \forall c \in \mathbb{C}, asn \in rel(c)\}$. Notably, it is impossible to identify $sol(d, \mathbb{C})$ by enumerating all assignments in a constraint network $\langle d, \mathbb{C}\rangle$, since the set of all potential assignments are in $\mathcal{P}(\mathbb{R})$ which is an infinite set.

To address this, we follow the framework of *abstract interpretation* [2] and rely on an abstraction $\mathbb{A}$ 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 we overapproximate the set of solutions. An element $a \in \mathbb{A}$ is soundly approximating the solutions' set $sol(d, \mathbb{C})$ iff $\gamma(a) \supseteq sol(d, \mathbb{C})$. It ensures that a preserves all solutions of the constraint problem, however, it may introduce *spurious solutions*—elements in the abstract set that do not satisfy the original constraints. Proving soundness of a verifier amounts to proving that every intermediate element a soundly approximate the solutions of the constraint network. Given a sound solving procedure, if we obtain an element a such that $\gamma(a) = \{\}$, we can conclude the problem is unsatisfiable and the network satisfies the properties under considerations. In the opposite case ($\gamma(a) \neq \{\}$), we cannot reach any conclusion. To improve this situation, we shall attempt to extract from a an *underapproximation* u such that $\gamma(a) \supseteq \gamma(u)$ and $\gamma(u) \subseteq sol(d, \mathbb{C})$. In our setting, a non-empty underapproximation ($\gamma(u) \neq \{\}$) is a counterexample. We illustrate these definitions with the *interval abstract domain*, the abstraction chosen in this work.

2.1 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 [7]. Note that we exclude the `NaN` (Not-a-Number) symbol from $\mathbb{F}$, as it lacks a total ordering w.r.t. 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 $+\infty$ 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 then overapproximated by an interval $[\underline{r}, \bar{r}]$, where $\underline{r}, \bar{r} \in \mathbb{F}^\infty$ such that $r \in [\underline{r}, \bar{r}]$. To obtain the most precise overapproximation, $\underline{r}$ ($\bar{r}$) must be the greatest (least) floating-point number less (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.** *A set of floating-point intervals* $\mathbb{I}$ *is a complete lattice* $\langle \mathbb{I}, \sqsubseteq\rangle = \langle \{[\ell, u] \mid \ell \in \mathbb{F} \cup \{-\infty\}, u \in \mathbb{F} \cup \{+\infty\}, \ell \leq u\} \cup \{\bot\}, \sqsubseteq, \sqcap, \sqcup, \bot\rangle$. *Let* $[\ell, u], [\ell', u'] \in \mathbb{I}$, *the lattice operators and special elements are defined as follows:*

[1] Internally, the constraint network is ternarized in JET to better align with the GPU architecture [18].

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

The greatest element can be represented as the interval $[-\infty, +\infty]$, hence there is no need to add a special top element $\top$.

2.2 Soundness of Floating-Point Interval Propagation

To ensure soundness, we account for floating-point rounding errors by performing interval operations with directed rounding. Following standard conventions [2] for extended floating-point arithmetic on $\mathbb{F}^\infty$, we define: $\infty + \infty = \infty$, $\infty + c = \infty$, $c + \infty = \infty$, $-\infty + c = -\infty$, $c + (-\infty) = -\infty$, and $-\infty + (-\infty) = -\infty$, where $c \in \mathbb{F}$. 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 to, e.g., [2].
For all operations $\diamond \in \{\oplus, \ominus, \otimes\}$, $[\ell, u] \diamond \bot = \bot \diamond [\ell, u] = \bot$.

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

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

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

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: 1) **solution box ($\mathbb{S}$)**: A box where every point is guaranteed to be a solution. 2) **boundary box ($\mathbb{B}$)**: A box containing both solution and non-solution points,

typically arising in the presence of inequalities. 3) **unknown box ($\mathbb{U}$)**: 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 principles, we formally define the *sound floating-point interval propagator* for each constraint $c \in \mathbb{C}$ in Definition 3.

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

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

3 An Overview of JET for Neural Network Verification

While JET is designed as a general-purpose solver supporting continuous domains, in this study, we specifically investigate its application and efficacy in neural network verification. First, we build a CCSP model (2) by reformulating the neural network verification problem (Eq. (1)). Subsequently, we introduce a solving algorithm, *branch-and-prune algorithm* based on the interval abstract domain and sound floating-point propagation. These components are established in existing interval-based solvers [1, 5], JET thus integrates them into a GPU-based architecture. Developed as an extension of Turbo [18]—a state-of-the-art GPU-based discrete constraint solver.

3.1 A CCSP Model

The neural network verification problem is formally defined in Eq. (1). If there exists a perturbed input vector $\mathbf{x} \in \mathbf{\Phi}(\mathbf{x}_0, \epsilon)$ such that the output of the network 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 preconditions. The complexity of neural network verification has been proven to be NP-complete [9].

The neural network verification problem can also be formulated as a CCSP model (2). In CCSP model (2), we directly model each neuron in the ReLU layer[2] using a pointwise max function without introducing auxiliary binary variables to represent piecewise linear function. This modeling approach can be seamlessly applied to other general nonlinear activation functions, such as sigmoid and tanh.

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

$$\begin{aligned} \mathbf{x}_{\ell+1} &= \mathbf{W}_\ell \mathbf{x}_\ell + \mathbf{b}_\ell && \forall \ell \in \mathbb{L}^A \\ \mathbf{x}_{\ell+1} &= \max(0, \mathbf{x}_\ell) && \forall \ell \in \mathbb{L}^R \\ \mathbf{x}_1 &\in \mathbf{\Phi}(\mathbf{x}_0, \epsilon) \\ 0 &\leq \mathbf{W}_{|\mathbb{L}|} \mathbf{x}^{|\mathbb{L}|} \\ \mathbf{x}_\ell &\in \mathbb{R} && \forall \ell \in \mathbb{L} \\ \mathbf{y} &\in \mathbb{R} \end{aligned} \tag{2}$$

3.2 Branch-and-Prune Algorithm

We combine floating-point interval propagation with a *branch-and-prune algorithm* (BP) to achieve both soundness and completeness. BP is a search framework that interleaves

[2] In this study, we focus on Rectified Linear Unit (ReLU) activation function only.

branching and pruning strategies to reduce domains and determine whether a solution exists within a box under a given precision ϵ. The details of the BP are presented in Algorithm 1.

In Algorithm 1, the input parameters consist of an initial box $\mathbf{x}$ (representing an overapproximation of the problem), a constraint set $\mathbb{C}$, the postconditions $\mathbf{\Psi}$, and a precision ϵ. Although $\mathbf{\Psi}$ is included within $\mathbb{C}$ for the propagation step, we still need it in the underapproximation step. The algorithm initializes three structures: a stack $\mathbb{Q}$ for unexplored search tree nodes, a set $\mathbb{B}$ for boundary boxes, and a set $\mathbb{U}$ for unknown boxes. Since the CCSP (2) 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.

Algorithm 1 is a depth-first search algorithm exploring the search space. It terminates when either the entire search space has been explored or a counterexample is identified. The solving process alternates floating-point interval propagation to prune variable domains and search by bisecting a domain to generate subproblems.

Given the compositional structure of neural networks, it is unnecessary to branch on all variables in the CCSP (2); instead, branching can be restricted to the input variables, as the values of subsequent variables are determined by the networks' relational constraints. To determine the status of a current box $\mathbf{q}$, an *underapproximation* method is used to extract the midpoint $\mathbf{m}$ from the input neurons and verify if it is a solution by feeding it back to the network N. If $N(\mathbf{m})$ violates the postconditions $\mathbf{\Psi}$, it is identified as a counterexample (**sat**). Otherwise, the algorithm continues to branch on $\mathbf{q}$, or categorizes the box as **unknown** if its precision is less than or equal to ϵ.

Upon completing the search, if $\mathbb{U}$ is empty, the network is proven safe (**unsat**) under the given conditions. Otherwise, the presence of a counterexample cannot be determined, and the algorithm returns **unknown**. Note that Algorithm 1 terminates immediately after finding a counterexample. This distinguishes it from standard BPs, which typically aim to exhaustively identify all solutions within the search space.

4 Preliminary Experimental Results

4.1 Soundness Issues in MILP Solver

In this section, we first 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` [6] (version 12.0.3) with `IBEX` (version 2.9.1.0) [1], an interval-based constraint solver[3]. 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.

We evaluate an instance (`Kin1.bch`) provided by the `IBEX` team. The solution[4] obtained from `Gurobi` is as follows:

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

Due to its interval-based solving strategy, `IBEX` returns 16 solution boxes. We select the

[3] Both solvers use their default parameter settings.
[4] We present the solutions using 15-digit precision.

Algorithm 1 Branch and Prune Algorithm for Neural Network Verification

Input: $\mathbf{x}, \mathbb{C}, \mathbf{\Psi}, \epsilon$
Result: **unsat** / **sat** / **unknown**

$\mathbb{Q} = \{\mathbf{x}\}, \mathbb{B} = \emptyset, \mathbb{U} = \emptyset$; `// initialization`
while $\mathbb{Q} \neq \emptyset$ **do**
  $\mathbf{q} = \text{pop}(\mathbb{Q})$; `// select next sub-problem as q`
  forall $c \in \mathbb{C}$ **do**
    $p_c(\mathbf{q})$; `// Definition 3`
  if $\mathbf{q} \neq \bot$ **then**
    if $\mathit{width}(\mathbf{q}) \leq \epsilon$ **then**
      if $\mathit{underapproximate}(\mathbf{q}, \mathbf{\Psi}, \epsilon)$ **then**
        $\mathbb{B} = \mathbb{B} \cup \{\mathbf{q}\}$;
        return sat ; `// terminate when there is a counterexample`
      else
        $\mathbb{U} = \mathbb{U} \cup \{\mathbf{q}\}$;
    else
      $\mathbb{Q} = \mathbb{Q} \cup \mathit{branch}(\mathbf{q}, \epsilon)$; `// generate 2 subproblems in node list Q`
return $\mathbb{U} = \emptyset$? **unsat** : **unknown** ;

box most comparable to Sol_{gurobi} for evaluation:

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

Critically, we observe that $\forall i \in \{2, 3, 4\}, Sol_{gurobi}(t_i) \notin Sol_{ibex}(t_i)$. 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.

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 VNN-COMP 2025 [10]. `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. 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.

We evaluate the performance of `JET` against with `IBEX` [1][5] and two state-of-the-art neural

[5] As experiments for `IBEX` are still ongoing, its complete evaluation results are omitted from Table 1. For our preliminary offline experiments, `IBEX` was evaluated under its default configuration using 20 small instances from the `SAT-ReLU` benchmark. Among these, `IBEX` successfully verified only 2 instances

Table 1 Verification results from JET compare with $\alpha\beta$-CROWN and PyRAT.
Values outside and inside the parentheses denote the results of the CPU and GPU implementations.

Verifiers	SAT	UNSAT	UNKNOWN	TIMEOUT	WRONG
$\alpha\beta$-CROWN'25	50 (50)	50 (50)	0 (0)	0 (0)	0 (0)
$\alpha\beta$-CROWN	0 (1)	0 (1)	97 (8)	0 (88)	3 (2)
PyRAT	0 (0)	17 (10)	0 (0)	83 (87)	0 (3)
JET	18 (18)	2 (4)	0 (22)	80 (55)	0 (0)

network verifiers on SAT-ReLU: $\alpha\beta$-CROWN [20, 21] and PyRAT [12]. These verifiers utilize diverse constraint solving techniques and abstract domains[6].

Given their maturity, many of these tools incorporate integrated falsification methods (attacks) that can identify counterexamples during a preprocessing stage, potentially bypassing the verification process. To ensure a fair comparison focused strictly on constraint solving capabilities, we disabled these attack-based features. 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[7]. In the following, we discuss several key insights and observations derived from the results in Table 1.

Although $\alpha\beta$-CROWN verified all instances in the SAT-ReLU benchmark without incorrect results during VNN-COMP 2025 [10], our analysis reveals potential underlying issues. While $\alpha\beta$-CROWN'25 produces 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. Notably, even with the use of a MILP solver, numerical instability remains a concern; as demonstrated in the previous experiment.

Floating-point soundness is exclusively guaranteed within the CPU implementation in PyRAT [12]. Due to this limitation, 3 incorrect verification outcomes were observed when employing the GPU version, which currently lacks floating-point soundness guarantees.

Based on these preliminary experimental results, JET successfully demonstrates its floating-point soundness. 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. Nevertheless, for certain instances in this benchmark, JET failed to reach a conclusive result within the timeout, potentially stemming from insufficient precision in both the overapproximation and underapproximation processes.

5 Conclusion

This paper represents an initial step toward leveraging continuous constraint solving techniques for sound neural network verification. Currently, achieving conclusive results within reasonable time limits remains a significant challenge. We identify two primary bottlenecks: first, the underlying overapproximation may lack the precision required to establish tight verification bounds; second, the underlying underapproximation may be insufficient for identifying a counterexample.

[6] The domains include: polyhedra ($\alpha\beta$-CROWN) and zonotopes (PyRAT).
[7] The CPU implementation of PyRAT is configured to employ sound floating-point arithmetic modes.

References

1. Frédéric Benhamou and Laurent Granvilliers. Chapter 16 - continuous and interval constraints. In *Handbook of Constraint Programming*. 2006.
2. Patrick Cousot. *Principles of abstract interpretation*. 2021.
3. Hai Duong, ThanhVu Nguyen, and Matthew B Dwyer. Neuralsat: A high-performance verification tool for deep neural networks. In *International Conference on CAV*, 2025.
4. Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Chaowei Xiao, Atul Prakash, Tadayoshi Kohno, and Dawn Song. Robust physical-world attacks on deep learning visual classification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018.
5. Laurent Granvilliers and Frédéric Benhamou. Algorithm 852: Realpaver: an interval solver using constraint satisfaction techniques. *ACM Transactions on Mathematical Software*, 2006.
6. Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2026.
7. IEEE. 754-2019 - ieee standard for floating-point arithmetic, July 2019.
8. Sagar Imambi, Kolla Bhanu Prakash, and GR Kanagachidambaresan. Pytorch. In *Programming with TensorFlow: solution for edge computing applications*. 2021.
9. Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks. In *International conference on computer aided verification*, 2017.
10. Konstantin Kaulen, Tobias Ladner, Stanley Bak, Christopher Brix, Hai Duong, Thomas Flinkow, Taylor T Johnson, Lukas Koller, Edoardo Manino, ThanhVu H Nguyen, et al. The 6th international verification of neural networks competition (vnn-comp 2025): Summary and results. *arXiv preprint arXiv:2512.19007*, 2025.
11. Suhas Kotha, Christopher Brix, J Zico Kolter, Krishnamurthy Dvijotham, and Huan Zhang. Provably bounding neural network preimages. *Advances in Neural Information Processing Systems*, 2023.
12. Augustin Lemesle, Julien Lehmann, Tristan Le Gall, and Zakaria Chihani. Verifying neural networks with pyrat. In *International Static Analysis Symposium*, 2025.
13. Christoph Müller, François Serre, Gagandeep Singh, Markus Püschel, and Martin Vechev. Scaling polyhedral neural network verification on gpus. *MLSys*, 2021.
14. Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin Vechev. Fast and effective robustness certification. *Advances in neural information processing systems*, 2018.
15. Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages*, 2019.
16. Attila Szász, Balázs Bánhelyi, and Márk Jelasity. No soundness in the real world: On the challenges of the verification of deployed neural networks. In *ICML*, 2025.
17. Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.
18. Pierre Talbot. A gpu-based constraint programming solver. *Proceedings of the AAAI Conference on Artificial Intelligence*, 2026.
19. Vincent Tjeng, Kai Xiao, and Russ Tedrake. Evaluating robustness of neural networks with mixed integer programming. *arXiv preprint arXiv:1711.07356*, 2017.
20. Shiqi Wang, Huan Zhang, Kaidi Xu, Xue Lin, Suman Jana, Cho-Jui Hsieh, and Zico Kolter. Beta-crown: efficient bound propagation with per-neuron split constraints for neural network robustness verification. 2021.
21. Huan Zhang, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh, and Luca Daniel. Efficient neural network robustness certification with general activation functions. *Advances in neural information processing systems*, 2018.
22. Xingjian Zhou, Keyi Shen, Andy Xu, Hongji Xu, Cho-Jui Hsieh, Huan Zhang, and Zhouxing Shi. Soundnessbench: A soundness benchmark for neural network verifiers. *arXiv preprint arXiv:2412.03154*, 2024.