Towards Sound Verification of Deployed Neural Networks

Attila Szász
University of Szeged, Hungary
szasz@inf.u-szeged.hu

Balázs Bánhelyi
University of Szeged, Hungary
banhelyi@inf.u-szeged.hu

Márk Jelasity
University of Szeged, and
HUN-REN–SZTE Research Group on AI, Szeged, Hungary
jelasity@inf.u-szeged.hu

Abstract

Verification methods aim at mathematically proving desirable properties of neural networks, such as robustness to adversarial perturbations. A verifier is sound if and only if it never claims that a neural network has the desired property when it does not. It was shown recently that none of the currently known verifiers that are claimed to be sound are guaranteed to be sound when considering the deployed version of the verified network. Due to this, all the known verifiers are vulnerable to certain backdoor attacks, where an adversarial network passes verification, but in reality, it exhibits adversarial behavior in specific deployment environments. So far, it has been suspected that sound verification is prohibitively expensive if we wish to verify all possible executions—including parallel and stochastic ones—in deployment. *We show that efficient and practically sound verifiers can be designed using a bound on the so-called backward error.* Using this bounding technique, we propose two verifiers: one based on interval bound propagation, and one using symbolic propagation. Both verifiers are proven to remain sound even if the deployment environment randomly selects a valid expression tree (an ordering and parenthesizing of the arithmetic operations) to compute the network, and even if numeric underflow or overflow occurs in any valid expression tree. This is especially interesting in the case of symbolic propagation, where the expression tree used by the verifier is completely different from the one used in deployment. We demonstrate empirically that our techniques introduce only a limited performance overhead while detecting all the known verifier backdoor attacks.

1 Introduction

The goal of the formal verification of neural networks is to provide mathematical guarantees for certain properties, such as robustness to adversarial perturbations. Numerous studies [1–3] have shown that existing verifiers focus on providing safety guarantees for the *theoretical model* of the network, while overlooking some of the practical aspects of floating-point environments [4, 5] resulting in *incorrect verification results in practice*.

Szász et al. [1] make the strongest claim, namely that none of the known verifiers are sound in practical deployment. This is due to the fact that the theoretical and practical networks are different mathematical objects, and the target of verification should be the practical network. In the simplest case of computing a sum, the theoretical computation is a well-defined operation with a unique result, while the practical computation depends on the ordering and parenthesizing of the addition

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

operations—in other words, the binary *expression tree* that is used to compute the sum—due to the non-associativity of floating-point arithmetic. However, the exact expression tree is normally not known in advance in deployment [4, 5]. Furthermore, this deficiency of current verification algorithms opens the door to practically viable *backdoor attacks* that are activated by certain targeted environments, while the network with the backdoor passes verification [1].

Here, we offer a solution to this problem. We build on a method proposed by Higham [6] to bound the error of a floating point inner product independently of addition order. We generalize the result to any expression tree and define a real-valued interval that includes the floating-point result of every possible expression tree for a given inner product in deployment. We then apply this technique to propose two practically sound verification techniques for ReLU networks: an interval bound propagation method and a symbolic propagation method. The main challenges included the application of the error bounds in symbolic propagation that use very different expression trees during verification and practical deployment, and dealing with potential *underflow and overflow*, in which case the original Higham bounds are not valid.

We prove that both verifiers are practically sound, that is, they never label a network robust when there is at least one possible expression tree that violates robustness. The only assumption we make is that no approximate algorithms (e.g., approximate matrix multiplication) are used in the deployment environment. Otherwise, we allow any expression tree to be selected dynamically.

We validate our theoretical claim in practice, namely that our two verifiers are able to verify adversarial networks that are known to fool the existing verifiers [1]. In addition, we demonstrate empirically that our bounding technique has a small overhead in terms of both runtime and output range approximation when verifying benign networks.

We summarize our main contributions below.

- We propose two novel verification algorithms by enhancing interval bound propagation and symbolic propagation with an error compensation technique building on Higham's approach on bounding backward error [6].
- Both novel verifiers are proven to be practically sound, that is, sound regardless of which expression tree of the network is used, also taking underflow and overflow into account.
- We demonstrate empirically that our novel algorithms are able to correctly verify a diverse set of adversarial networks from the literature while the bounding technique requires practically no compromise in terms of the tightness of the output approximation for benign networks.

2 Related Work

Due to extensive research in the field of neural network safety, a wide range of verifiers have been developed so far. Good overviews of the topic are offered by [7–10]. These verifiers can be classified in numerous ways, but here, we will focus on sound verifiers. A verification algorithm is called *sound* if every property it proves to be true is indeed true, and *complete* if it successfully proves every true property.

Since neural network verification that is both sound and complete is known to be NP-complete [11], most verifiers aim to over-approximate the behavior of the network, thereby achieving soundness but sacrificing completeness. These algorithms typically use interval propagation or symbolic propagation to compute bounds on the output. Interval propagation is based on interval arithmetic [12], often called interval bound propagation or IBP [13, 14] in the context of neural network verification. Symbolic approaches propagate interval definitions symbolically, which allows for tighter bounds. Most symbolic methods are based on a linear over-approximation of the ReLU function. Some employ single-neuron approximations [15–18], while others utilize multi-neuron approximations to achieve higher precision [19, 20].

Methods that are claimed to be sound and complete often formalize the verification problem as an SMT [11, 21] or mixed-integer linear programming (MILP) model [22, 23] and apply solvers to solve them. Some other algorithms are built upon the branch-and-bound (BaB) [24] framework. These algorithms apply fast sound algorithms during the bounding phase and then use different heuristics to perform the branching. Detailed explanations of these methods are offered by [18, 19, 25, 26].

In recent years, numerous studies have examined how deployment environments influence the behavior of numerical programs, including neural networks [4, 5, 27–30], as well as the reliability of verification algorithms [1–3]. These studies demonstrated that the theoretical and practical soundness of a verification algorithm do not necessarily align. They constructed malicious networks that are reported to be safe by state-of-the-art sound verifiers but exhibit incorrect behavior in certain deployment environments.

Zombori et al. [2] defined an adversarial network that exploits the numerical error introduced by LP solvers in LP/MILP model-based verification. Jia and Rinard [3] proposed an adversarial attack on verifiers by finding adversarial (input, network) pairs that are incorrectly verified by MIPVerify. Szász et al. [1] went a step further, arguing that the theoretical and deployed (practical) models are different mathematical objects, and verification must consider the details of the deployment environment. The authors defined different adversarial networks and showed that none of the existing verifiers are able to detect their malicious behavior. They highlighted that the primary source of the vulnerability is that none of the available bounding algorithms can compute bounds that, regardless of the order of additions, contain all possible outputs.

Shanmugavelu et al. [31] showed that—due to summation not being associative in floating-point—the correct classification of inputs near the decision boundary can strongly depend on the order of floating-point operations that would otherwise be associative in real arithmetic. The impact of numerical representation has also been investigated in the context of quantization: recent works demonstrated that quantization errors can be exploited to introduce harmful behaviors into models that appear safe in higher precision [32, 33].

We should also mention a different approach for circumventing the floating-point problem: using quantized neural networks that avoid rounding errors by using integer arithmetic [34–36].

Despite these research efforts, the problem of order-independent bounding of floating-point sums—and thus floating-point sound verification—remains, to the best of our knowledge, unresolved.

3 Background: Verification, Floating-Point, Practical Soundness

Let $f(.;\theta) : \mathbb{R}^n \mapsto \mathbb{R}^m$ be a neural network, where $\theta \in \mathbb{R}^k$ is a vector of real parameters. In this paper, we work with classifier neural networks, that is, m denotes the number of output classes. The class label of an input x is given by $y(x) = \arg\max_i f(x;\theta)_i$, that is, the index of the maximal output value.

In robustness verification, the central question is whether the classification result of a model remains unchanged when its input is perturbed within a bounded neighborhood of a given input point x^*. Let $D \subseteq \mathbb{R}^n$ denote this neighborhood, typically specified by a norm $\|\cdot\|_p$ and a perturbation radius ϵ: $D_{\epsilon,p}(x^*) = \{x : \|x - x^*\|_p \leq \epsilon\}$. We define a safety property $P \subseteq \mathbb{R}^n$ that contains all inputs whose predicted class is identical to that of x^*. This property can be expressed as $P(x^*) = \{x : y(x) = y(x^*)\}$. The robustness requirement is then $D_{\epsilon,p}(x^*) \subseteq P(x^*)$. To make the verification problem more tractable, we apply a transformation to the network by appending an affine layer whose i-th output component is $f(x;\theta)_{y(x^*)} - f(x;\theta)_i$. Under this reformulation, the verification task becomes proving

$$\forall x \in D_{\epsilon,p}(x^*),\ f(x;\theta) \geq 0, \tag{1}$$

where f now denotes the modified model. From now on, we will assume that $p = \infty$ and f is a ReLU network.

Floating-point. In deployment, neural networks are implemented in floating-point. Since floating-point arithmetic operates with finite precision, the results of operations are rounded. The most commonly used rounding mode is round-to-nearest, where the output is rounded to the nearest representable number. Rounding makes floating-point computations *order-dependent* (i.e., *non-associative*). For example, in the IEEE 754 standard, $2^{53} + 1 - 2^{53}$ equals 0 in *binary64* arithmetic, when using this fixed order. However, reordering the operations as $2^{53} - 2^{53} + 1$ equals 1 under the same settings. Floating-point is *precision-dependent* as well. For example, $2^{24} + 1 - 2^{24}$ evaluates to 0 in *binary32*, but to 1 in *binary64*.

Practically sound verification. Sound verification requires computing a lower bound on f over a domain, in an attempt to prove Eq. (1). Practically sound verification, however, requires *a lower*

bound on every possible floating-point result in a given implementation. This, as mentioned above, depends on the floating-point precision and on the expression tree that is applied to compute f in deployment, and every expression tree might result in a different numeric result. Here, we explain what problems this might cause for interval bound propagation and symbolic evaluation, that are sound but not practically sound, as pointed out by Szász et al. [1].

Interval bound propagation (IBP) executes each arithmetic operation using intervals. For example, the sum of the intervals $[l_1, u_1]$ and $[l_2, u_2]$ is $[l_1 + l_2, u_1 + u_2]$. Here, $[l_1 + l_2, u_1 + u_2]$ guarantees the bounding of all possible $s = a_1 + a_2$, where $a_1 \in [l_1, u_1]$ and $a_2 \in [l_2, u_2]$. Other operators like subtraction, multiplication, and division can also be extended to work on intervals. A big advantage of this method is that very efficient implementations are available that can exploit GPU tensor operations ([13, 37]). IBP is normally implemented in floating-point using outward rounding to guarantee the bounding of the real result. That is, the lower bound of the output interval is rounded towards $-\infty$ and the upper bound is rounded towards $+\infty$. For example, $[1, 1] + [2^{53}, 2^{53}]$ evaluates to $[2^{53}, 2^{53} + 2]$ in a *binary64* environment, successfully bounding the real-valued result $2^{53} + 1$. This means that IBP can be implemented in floating-point to be sound.

However, **IBP is not practically sound**: for example, the expression $([1, 1] + [1, 1]) + [2^{53}, 2^{53}]$ evaluates to $[2^{53}+2, 2^{53}+2]$ in *binary64*, which does not include the *binary64* result of $(2^{53}+1)+1 = 2^{53}$, which represents a valid expression tree. This means that IBP does not correctly bound the result of every possible expression tree in deployment [1].

Symbolic propagation methods maintain symbolic expressions instead of numeric intervals. The input is typically represented as variables (interval or scalar), and during propagation, only the coefficients of the expressions are updated. Here, we will consider only *linear interval expressions* developed by [38]

$$L(x) = [l_0, u_0] + \sum_{i=1}^{n} [l_i, u_i] x_i \tag{2}$$

that are the most common in neural network verification, as they are easy to represent numerically and efficient to compute with. This approach aims to reduce the dependency problem presented by IBP: the overestimation of bounds that occurs when multiple occurrences of the same variable are treated independently. Examples for verifiers that use symbolic propagation, along with additional techniques and optimizations, include DeepPoly ([16]), CROWN ([25]), and DeepZ ([15]).

To determine the range represented by a linear interval expression $L(x)$, a concretization function γ is used to compute a containing (over-approximated) interval for $L(x)$ at a given value of x. In notation, $[l(x), u(x)] = \gamma(L, x)$. In this work, we define $\gamma(L, x)$ as the result of evaluating the linear interval expression using floating-point sound interval arithmetic at x. As shown by [1], this approach **inherits the IBP bounding issue**, as the floating-point result computed with a different expression tree than the one used by γ may not be within the bounds returned by γ.

Here, we will focus on the practical soundness of vanilla IBP and vanilla symbolic propagation of linear interval expressions, as these are fundamental building blocks of modern verifiers. Additional techniques could also be made practically sound using our approach; we leave this for future work.

4 Sound Verification in Deployment

As mentioned previously, we are not aware of *any practically sound verification algorithms* that provably verify all the possible expression trees that could occur in deployment for computing the neural network. Here, we propose two approaches that are able to do this: an enhanced IBP algorithm and a symbolic propagation approach.

Assumptions. We work with ReLU networks that use only the inner product and the ReLU nonlinearity as building blocks. This covers bias terms as well if we append a constant 1 to the input vector. The sensitivity domain is defined by the ∞-norm. We will also assume that the maximal fan-in n in the network is such that $n\mu < 1$ where μ is the machine epsilon. We assume that the deployment environment allows for all possible expression trees that compute the neural network numerically (and not symbolically). That is, before computing the output of a neuron, we first compute the numeric value of each input neuron. The minimum floating-point precision is known, but higher precision is allowed in any operation. Note that in practice, there could be additional errors due to the

application of numerical approximation algorithms that we do not consider here. Initially, we assume that no under- or overflow occurs, but this assumption is relaxed in Section 4.4.

The proofs not included in the following sections can be found in the Appendix.

4.1 Bounding Inner Products

In the case of ReLU networks, as we will detail later, it suffices to deal with the case of inner products, because the network is composed of inner products and ReLU functions, and the ReLU functions introduce no numeric issues (in the case of IBP) or are approximated by linear expressions (in the case of symbolic methods). Here, we first present Higham's bounds for the floating-point inner product $\theta^\top x$ ($x, \theta \in \mathbb{R}^n$) [6] and we then use them to define an interval that covers the output of any expression tree.

Let us first fix the expression tree $\theta^\top x = (\cdots(\theta_1 x_1 + \theta_2 x_2) + \theta_3 x_3) + \cdots + \theta_n x_n)$. Let $\hat{s}$ denote the floating-point sum following this expression tree. We can express the floating-point relative errors committed in each step of the computation by multiplying the exact result of each addition and multiplication with the exact relative error term $(1 + \delta_i)$, where $\delta_i \in \mathbb{R}$, getting

$$\hat{s} = (\cdots(\theta_1 x_1(1+\delta_1) + \theta_2 x_2(1+\delta_2))(1+\delta_3) + \theta_3 x_3(1+\delta_4))(1+\delta_5) + \cdots))(1+\delta_{2n-1}). \quad (3)$$

Rearranging Eq. (3) we get $\hat{s} = \sum_{i=1}^{n} \theta_i x_i \Delta_i$ where Δ_i is the product of at most n relative error terms.

We want to find good bounds on Δ_i that will be crucial for our sound verification approach. First, consider that $\forall i,\ |\delta_i| \leq \mu$ where μ is the so-called *machine precision*, that is, the difference between 1 and the next larger floating-point number. Using the machine precision μ, the following lemma helps us bound Δ_i.

Lemma 1 (Higham [6])**.** *Let $\delta(n)$ be defined by the equation $\prod_{i=1}^{n}(1+\delta_i) = 1 + \delta(n)$. Then, if $|\delta_i| \leq \mu$ and $n\mu < 1$ then $|\delta(n)| \leq \frac{n\mu}{1-n\mu}$.*

Let $\Delta = \frac{n\mu}{1-n\mu}$. Based on the lemma, it is clear that $\Delta_i \in [1-\Delta, 1+\Delta]$ for any $i \leq n$, because Δ is monotone increasing in n. This means that the interval

$$[l_\Delta, u_\Delta] = \theta_1 x_1[1-\Delta, 1+\Delta] + \cdots + \theta_n x_n[1-\Delta, 1+\Delta] \quad (4)$$

is guaranteed to contain both $\hat{s}$ and the exact value of $\theta^\top x$.

It should be evident that $[l_\Delta, u_\Delta]$ also contains the floating-point result of every expression tree that has the same structure as in Eq. (3) but with a different ordering of the members of the sum. This is because, independently of i, we replaced all Δ_i with $[1-\Delta, 1+\Delta]$. However, we can state more. It is also true that $[l_\Delta, u_\Delta]$ contains the floating-point result of *every possible expression tree*.

Proposition 1. *Let $\hat{s}_o$ be the result of computing the inner product $\theta^\top x$ ($x, \theta \in \mathbb{R}^n$) in floating-point using expression tree o using any rounding method. Let $[l_\Delta, u_\Delta]$ be defined as in Eq.* (4) *where $\Delta = \frac{n\mu}{1-n\mu}$, and μ is the machine precision. Then, for any valid expression tree o, we have $\hat{s}_o \in [l_\Delta, u_\Delta]$.*

4.2 Deep Linear Networks with IBP and Symbolic Methods

Now that we have described how to bound an inner product to include every possible floating-point result, we can describe our main contribution, namely an approach to enhance existing methodologies for verification using the derived relative error bound Δ.

Deep linear networks. Here, we focus on deep linear networks, that is, directed acyclic networks of neurons that compute the inner product of their input and their weight vector, and use the identity function as activation function. The ReLU activation function and other implementation details will be added as well in Section 4.3 where we discuss our final algorithms.

IBP. To implement sound verification through IBP, we need to be able to handle interval-valued inputs. Let x be a vector of intervals, with $x_i = [l_i, u_i]$. We can compute $[l_\Delta, u_\Delta]$ using Eq. (4), but replacing every scalar x_i with the interval $[l_i, u_i]$.

$$[l_\Delta, u_\Delta] = \theta_1[l_1, u_1][1-\Delta, 1+\Delta] + \cdots + \theta_n[l_n, u_n][1-\Delta, 1+\Delta], \quad (5)$$

The following proposition is analogous to Proposition 1 but adapted to interval-valued inputs.

Proposition 2. *Let $[\hat{l}_o, \hat{u}_o]$ be the result of computing the inner product $\theta^\top x$ ($x = [l, u]$ and $\theta, l, u \in \mathbb{R}^n$) in interval floating-point with outward rounding using expression tree o. Let $[l_\Delta, u_\Delta]$ be defined as in Eq.* (5) *where $\Delta = \frac{n\mu}{1-n\mu}$, and μ is the machine precision. Then, for any valid expression tree o, we have $[\hat{l}_o, \hat{u}_o] \subseteq [l_\Delta, u_\Delta]$.*

It is easy to see that this result indicates that if we propagate the inputs through the network using Eq. (5) then the output interval will also contain every floating-point result according to any expression tree of the entire deep linear network, because the expression tree of the entire network is simply a composition of the inner product expression trees. The following corollary formulates this statement.

Corollary 1. *Let $L(x)$ denote the output neuron of a deep linear network ($x = [l, u]$ and $l, u \in \mathbb{R}^n$). Let $[\hat{l}_o, \hat{u}_o]$ be the result of computing $L(x)$ using floating-point arithmetic and the expression tree o. Let $[l_\Delta, u_\Delta]$ be the real interval of the output neuron as computed using IBP based on the application of Eq.* (5) *in each neuron. Then, for any valid expression tree o, we have $[\hat{l}_o, \hat{u}_o] \subseteq [l_\Delta, u_\Delta]$.*

Symbolic methods. Here, let us consider the case of using interval linear expressions as defined in Eq. (2). The deep linear network is a directed acyclic expression graph, with linear expressions at its nodes (that is, at its neurons) that depend on the input x. Let neuron i be computed symbolically as

$$L^i(x) = \theta_{i,1} L^{i,1}(x) + \cdots + \theta_{i,n_i} L^{i,n_i}(x), \tag{6}$$

where the expressions $L^{i,j}(x)$ represent the jth input neuron for neuron i, or $L^{i,j}(x) = x_j = [l_j, u_j]$ for an input node. For neuron i, we can also define our usual extended version with the help of Δ:

$$L^i_\Delta(x) = \theta_{i,1} L^{i,1}_\Delta(x)[1-\Delta, 1+\Delta] + \cdots + \theta_{i,n_i} L^{i,n_i}_\Delta(x)[1-\Delta, 1+\Delta], \tag{7}$$

where the input expressions $L^{i,j}_\Delta(x)$ are also defined by this equation recursively. Here, n_i is the number of inputs of neuron i, and for simplicity, we define $\Delta = (n_{\max}\mu)/(1 - n_{\max}\mu)$, where $n_{\max} = \max_i n_i$, the maximal input dimension over the neurons.

Proposition 3. *Let $L(x)$ denote the output neuron of a deep linear network ($x = [l, u]$ and $l, u \in \mathbb{R}^n$). Let $[\hat{l}_o, \hat{u}_o]$ be the smallest interval that includes the result of computing $L(x^*)$ for any $x^* \in x$ using floating-point arithmetic and the expression tree o. Let $[l_\Delta, u_\Delta] = L_\Delta(x)$ be the real interval of the output neuron, where the linear expression $L_\Delta(x)$ is computed recursively applying Eq.* (7)*, where $\Delta = (n_{max}\mu)/(1 - n_{max}\mu)$, μ is the machine precision, and $n_{max} = \max_i n_i$, the maximal input dimension over the neurons. Then, for any expression tree o such that neurons are computed numerically and not symbolically, we have $[\hat{l}_o, \hat{u}_o] \subseteq [l_\Delta, u_\Delta]$.*

This result is analogous to Corollary 1 and is the basis of our symbolic verification algorithm, discussed in the next section.

4.3 Putting it together

In Section 3, we discussed two vanilla techniques: IBP and symbolic forward propagation of linear expressions (or *Symbolic* for short). We are now ready to make these techniques practically sound. Our previous results—most importantly Corollary 1 (for IBP) and Proposition 3 (for Symbolic)—provide the foundation of implementing sound versions of the corresponding methods even when we allow for all possible expression trees in deployment.

The practically sound versions of the two vanilla methods will be called *FPSoundIBP* and *FPSoundSymbolic*. We should stress that the original versions, IBP and Symbolic, are **not novel algorithms**, our contribution here is providing a practically sound version of these basic algorithms. As we will see later, although FPSoundSymbolic is a practically viable algorithm, there are several, more advanced verifiers that outperform FPSoundSymbolic due to more sophisticated overapproximation methods of the nonlinearities, or more advanced symbolic methods such as backsubstitution [15, 16]. We leave extending our bounding techniques to these more advanced algorithms for future work. This is a trivial task in many cases (such as any linear overapproximations) and more interesting for techniques such as backsubstitution.

FPSoundIBP. Corollary 1 defines most of the algorithm, which is a simple IBP using our error bounding technique. We need to address two practical issues. The first is that $[l_\Delta, u_\Delta]$ is an exact interval, that is, no floating-point rounding errors are accounted for while computing the right-hand

side of Eq. (5). However, it is easy to account for floating-point errors using outward rounding of the interval boundaries, resulting in an interval $[\hat{l}_\Delta, \hat{u}_\Delta]_o$, where o is the expression tree. Due to the outward rounding, for any expression tree o we have $[l_\Delta, u_\Delta] \subseteq [\hat{l}_\Delta, \hat{u}_\Delta]_o$, so an implementation can use any expression tree to over-approximate $[l_\Delta, u_\Delta]$. The second issue is implementing the ***ReLU function***. But this is trivial, as we simply have to cut the negative half (if any) of the propagated interval, that is, $\text{ReLU}([l, u]) = [\max(0, l), \max(0, u)]$. This operation has no numeric issues.

FPSoundSymbolic. Proposition 3 defines the basic algorithm, which is recursively computing the coefficients of $L_\Delta(x)$ using Eq. (7), and then computing the interval it represents. Here, like for IBP, we need to address two issues. The first is that Eq. (7) defines an exact computation, but in practice, the coefficient intervals of the input variables x have to be properly over-approximated when using interval arithmetic. However, any expression tree can be used, because they all contain the real interval coefficient. The resulting expression $\hat{L}_\Delta(x)$ can then be concretized for a given input x, again, using over-approximation, but with an arbitrary expression tree, because every expression tree will contain the real concretization that in turn contains all the floating-point results as stated by the proposition.

The other issue is computing $\text{ReLU}(L(x))$ for a neuron $L(x)$, when $L(x)$ is unstable (that is, it can take on both positive and negative values). In that case, the output range cannot be represented exactly using linear expressions, so we need to apply a linear over-approximation of the ReLU function. In this work, we adopt a technique similar to the one used by DeepZ ([15]). Let $[l, u] = \gamma(L(x))$ be an over-approximating concretization. Then,

$$\text{ReLU}(L(x)) = \begin{cases} L(x), & \text{if } l > 0, \\ [0, 0], & \text{if } u \leq 0 \\ [\lambda_l, \lambda_u] \cdot L(x) + x_{new} & \text{otherwise,} \end{cases} \tag{8}$$

where $[\lambda_l, \lambda_u]$ is a floating-point interval, which includes the (real) parameter $\lambda = u/(u - l)$ and $x_{new} \in [0, -l]$ is a new variable. This represents a linear envelope for the ReLU function over $[l, u]$.

4.4 Underflow and Overflow

Let us now sketch the basic ideas behind dealing with overflow and underflow. To illustrate the ideas, here, we focus on sums. The full technical discussion is included in Appendix B.

Overflow. We need to compute the bounds using Eq. (4) for the positive and negative addends separately, and if the computed bounds have a larger absolute value than the maximum representable number, then there is a possibility of overflow in deployment, otherwise we can exclude overflow. If verification uses the same precision as deployment, then an overflow of the above bounds indicates the possibility of overflow in deployment.

Underflow needs more consideration. The basic idea is that if all the addends are integer multiples of the smallest possible representable floating-point number then there can be no underflow. This condition trivially holds if the deployment uses so called subnormal floating-point numbers. If there are no subnormals, then this condition can be checked. If the condition does not hold, then we cannot exclude an underflow. However, as we explain in Appendix B, this does not mean that the verification process has to stop, as we can widen the intervals very slightly so that all the endpoints are integer multiples of the smallest possible representable number. We will prove that this way, the final bound remains valid, despite possible expression trees that result in an underflow. Also, note that this widening is necessary only extremely rarely, so it might even indicate an adversarial attempt.

5 Numerical Results

Here, we present empirical results with our two algorithms FPSoundIBP and FPSoundSymbolic that we implemented in Julia ([39]), where the interval arithmetic operations are based on the IntervalArithmetic.jl package ([40]). Please refer to Appendix D for more details.

Verifying adversarial networks. Our motivation was to provide sound verification in deployment, allowing for every expression tree and floating point representation. Thus, our algorithms should be able to correctly verify all the adversarial networks presented in [1] and they indeed do, as Table 4 in

Table 1: The networks used in the empirical evaluation, and the performance of some baseline verifiers. Sym, Poly, and Zono stand for Symbolic, DeepPoly [16] and DeepZ [15].

ID	Dataset	Model	Training	#Layers; Nodes; Pars.	Verified safe (%)				Runtime (s)	
					IBP	Sym.	Zono	Poly	Zono	Poly
MS	MNIST	ConvSmall [19]	Normal	3; 3 604; 89 608	0	10	19	20	0.22	0.45
ML	$\epsilon = 0.1$	ConvBig [19]	DiffAI [41]	6; 34 688; 1 974 764	90	92	92	94	1.72	2.24
MXL		ConvSuper [19]	DiffAI [41]	6; 88 500; 10 985 964	83	95	95	97	124.96	139.31
CS	CIFAR10	ConvSmall [19]	PGD [42]	3; 4 852; 125 324	0	25	27	30	0.35	0.95
CL	$\epsilon = 2/255$	ConvBig [19]	PGD [42]	6; 62 464; 2 466 864	0	17	17	19	29.81	42.31
M	MNIST	Base [43]	PGD [42]	3; 4 814; 166 406	0	90	94	96	0.28	1.13
MO3	$\epsilon = 0.1$	Order3 [1]	PGD [42]	5; 5 572; 246 749	0	90	94	96	0.73	1.33

Appendix C demonstrates. Although this is not surprising, as theory guarantees sound verification, these results validate the correctness of our implementation.

In the remainder of the section, we focus on the extra cost of our techniques to achieve practical soundness. We discuss verification accuracy and running time.

5.1 Setup

We will use seven networks to test our algorithms, listed in Table 1. Five of these networks are adopted from [19], and the remaining two were used in the adversarial network construction (with MO3 being the adversarial version of M) in [1].

The MO3 network is composed of the pre-trained MNIST base model M, a detector neuron, and the definition of an adversarial behavior that is triggered if the detector neuron is activated. The output of the detector neuron is designed to highly depend on the expression tree with which the sum is evaluated. MO3 has a detector neuron that computes $1 + \cdots + 1 + \omega - \omega$, where the sum contains 512 1s. Here, ω is the smallest number such that the next representable number after ω is $\omega + 2$. Only those expression trees will evaluate to 0, in which $1 + \omega$ is computed first (note that floating-point addition is commutative, so $\omega + 1$ gives the same result), then the 1s are added one by one, and $-\omega$ is added last. These expression trees are highly extreme, so those verifiers that do not cover all the possible expression trees are not likely to include 0 in the computed range of the expression. The detailed explanation of MO3 (along with other adversarial networks) can be found in Appendix C.

Some baseline performance metrics are included in Table 1 to anchor the performance of our simple verifiers. IBP and Symbolic are the non-sound versions of our methods FPSoundIBP and FPSoundSymbolic: they do not apply the interval-widening term Δ. Not surprisingly, in terms of verification performance Symbolic (a vanilla symbolic forward propagation method) performs worse than both DeepZ and DeepPoly (two well-known symbolic verifiers) although its performance is not far from DeepZ on the robust models. Note that with more advanced over-approximation methods for the ReLU function we could also improve these metrics, but this was not our focus here. Running times are averages over 100 test inputs. They are extremely hard to compare [44] but our implementation is comparably fast so our overhead estimations are well grounded. The naive IBP verifier performs poorly, as expected, except for well-behaved networks, but we still analyze its overhead for reference.

MO3 was explicitly designed to break the soundness of the baseline verifiers. On MO3, the correct percentage of practically safe inputs is 0%, while—as shown in Table 1—the baseline verifiers ignore the adversarial backdoor in MO3 and treat it as if it was identical to model M. The slight increase in running time is due to the extra neurons in the backdoor.

5.2 Width of the Output Interval

Widening is negligible. We evaluated the output interval width of our verification methods on our model benchmark on the first 100 test samples of the respective datasets, the results are included in Tables 2 and 3. The size of the input perturbation in ℓ_∞ norm is given by ϵ. We indicate both the average and maximum interval size. In the case of the benign networks, the intervals are practically identical for the naive and the sound algorithm, so we also explicitly show the difference, that is, the amount of widening of the output intervals. The overhead is negligible, it is in the range of numeric

Table 2: Overhead of the IBP method family (output interval width and running time) based on the first 100 test examples.

ID	ϵ	IBP		FPSoundIBP		overhead (FPSoundIBP−IBP)		Avg. running time (s)			
		avg	max	avg	max	avg	max	IBP	+FPS	+O	+U
MS	0.1	71.94	106.80	71.94	106.80	$2.15 \cdot 10^{-11}$	$3.28 \cdot 10^{-11}$	0.01	0.02	0.03	0.04
ML	0.1	1.82	6.03	1.82	6.03	$2.37 \cdot 10^{-12}$	$3.53 \cdot 10^{-12}$	0.33	0.43	0.53	0.57
MXL	0.1	4.79	9.78	4.79	9.78	$2.02 \cdot 10^{-11}$	$3.19 \cdot 10^{-11}$	1.68	2.20	2.43	2.86
CS	2/255	167.60	212.17	167.60	212.17	$2.02 \cdot 10^{-10}$	$2.26 \cdot 10^{-10}$	0.01	0.025	0.03	0.06
CL	2/255	4373.54	6834.75	4373.54	6834.75	$8.20 \cdot 10^{-9}$	$8.25 \cdot 10^{-9}$	0.40	0.60	0.80	0.93
M	0.1	165.74	220.27	165.74	220.27	$6.84 \cdot 10^{-11}$	$9.01 \cdot 10^{-11}$	0.02	0.03	0.05	0.06
MO3	0.1	165.74	220.27	2003.25	2018.41	1837.51	1798.14	0.02	0.04	0.06	0.07

Table 3: Overhead of the Symbolic method family (output interval width and running time) based on the first 100 test examples.

ID	ϵ	Symbolic		FPSoundSymbolic		overhead (FPS.S.−Sym.)		Avg. running time (s)			
		avg	max	avg	max	avg	max	Sym.	+FPS	+O	+U
MS	0.1	13.06	20.60	13.06	20.60	$6.84 \cdot 10^{-11}$	$9.01 \cdot 10^{-11}$	0.60	1.01	1.12	1.42
ML	0.1	1.80	5.99	1.80	5.99	$3.23 \cdot 10^{-12}$	$4.95 \cdot 10^{-12}$	2.18	3.40	3.62	4.69
MXL	0.1	2.75	6.35	2.75	6.35	$4.95 \cdot 10^{-11}$	$1.01 \cdot 10^{-10}$	95.04	177.10	177.60	220.70
CS	2/255	3.92	6.21	3.92	6.21	$1.68 \cdot 10^{-9}$	$2.26 \cdot 10^{-9}$	1.87	3.34	3.40	4.34
CL	2/255	2.20	4.14	2.20	4.14	$1.18 \cdot 10^{-8}$	$1.20 \cdot 10^{-8}$	53.83	106.10	106.79	132.33
M	0.1	5.76	10.55	5.76	10.55	$6.84 \cdot 10^{-11}$	$9.01 \cdot 10^{-11}$	0.80	1.54	1.58	1.96
MO3	0.1	5.76	10.55	$6.33 \cdot 10^{6}$	$6.33 \cdot 10^{6}$	$6.33 \cdot 10^{6}$	$6.33 \cdot 10^{6}$	0.83	1.57	1.62	2.01

error. This means that we achieve soundness without sacrificing precision. Indeed, the sound and naive versions can verify exactly the same inputs in our benchmark.

Wide output is a red flag. The only exception is the adversarial network MO3. However, *this is the intended behavior*: a sound verifier should cover the full feasible output range, which is wide ($[0, 512]$ for the detector neuron of MO3). Our sound verifiers produce wider intervals, for example, FPSoundIBP estimates the post-ReLU output of the detector neuron as $[0, 2573]$, and FPSoundSymbolic estimates $[-964.39, 2569]$, which then propagate through the network and amplify the output width. We stress again that for the benign networks the extra accumulated error is negligible. When our sound verifiers output an abnormally wide interval relative to those returned by the naive methods, it indicates that the network being verified has some unusual problems that need further investigation.

5.3 Running Time Overhead

The increased running time is mainly due to calculating the Δ term for every single neuron and ensuring numerical stability against underflow and overflow. That is, computing the inner product requires us to first calculate and inspect the individual terms scaled by the weights before summing them, which breaks standard hardware optimizations. Another potential source of overhead is that the number of unstable ReLU gates might increase due to the wider approximations of the output ranges of the neurons, which also increases running time, because more internal products have to be computed and more neurons need to be approximated. However, this effect is negligible, as we demonstrated earlier.

We evaluated the runtime of our verification methods on the first 100 test samples of the respective datasets, the average runtimes are included in Tables 2 and 3. We present an ablation by cumulatively adding the enhancements one-by-one to the naive versions (i.e., IBP and Symbolic): the interval-widening technique for floating-point soundness (+FPS), the overflow detection technique (+O) and underflow detection (+U). The size of the input perturbation is given by ϵ.

The IBP family is very fast in general. In the Symbolic family, practical floating-point soundness roughly doubles the running time, overflow detection has an insignificant cost, and underflow detection adds another 30% running time (note that in our benchmark no underflow or overflow was ever

detected). While these measurements strongly depend on the implementation details, clearly, the running time is expected to stay in the same order of magnitude as the naive versions.

6 Conclusions and Limitations

We proposed improved verification techniques for the first time that are able to cover every possible expression tree that is possible in deployment. We used an approximation of the relative error Δ that can be used in IBP or symbolic approaches alike to correctly over-approximate the range of ReLU networks, in a practically sound manner. Our empirical measurements indicate a tolerable overhead.

We also address the case when overflow and underflow are possible in any possible expression tree: we can detect and flag potential overflow, and we can correctly bound those networks that can potentially experience underflow in deployment. We note that in our model benchmark, no potential underflow or overflow was detected.

As for limitations, the FPSoundIBP algorithm is mostly of theoretical interest, as in practical situations it produces intervals that are too wide. FPSoundSymbolic is practical, however, it could be improved further by applying more sophisticated, perhaps adaptive overapproximation techniques. This is possible, because our interval-widening technique can be generalized to any linear approximations.

Also, it is a challenge if the deployment uses very low precision while the neurons in the model have a very large fan-in, due to the $n\mu < 1$ constraint. Another problem is when there are sources of errors other than what we considered, for example, numerically approximated matrix operations that deviate from the valid expression trees of the verified model. Such situations require specific environment-dependent approaches. Note, though, that additional standard floating-point operations such as Fused Multiply-Add (FMA), and non-IEEE representations can easily be adopted.

Acknowledgments and Disclosures

Supported by project 2024-1.2.3-HU-RIZONT-2024-00017 that has been implemented with the support provided by the Ministry of Culture and Innovation of Hungary from the National Research, Development and Innovation Fund, financed under the 2024-1.2.3-HU-RIZONT funding scheme.

The authors have no competing interests to declare.

References

[1] Attila Szász, Balázs Bánhelyi, and Márk Jelasity. No soundness in the real world: On the challenges of the verification of deployed neural networks. In *International Conference on Machine Learning (ICML)*, 2025. URL `https://arxiv.org/abs/2506.01054`.

[2] Dániel Zombori, Balázs Bánhelyi, Tibor Csendes, István Megyeri, and Márk Jelasity. Fooling a complete neural network verifier. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL `https://openreview.net/forum?id=4IwieFS44l`.

[3] Kai Jia and Martin Rinard. Exploiting verified neural networks via floating point numerical error. In Cezara Drăgoi, Suvam Mukherjee, and Kedar Namjoshi, editors, *Static Analysis*, pages 191–205, Cham, 2021. Springer International Publishing. ISBN 978-3-030-88806-0.

[4] Alexander Schlögl, Nora Hofer, and Rainer Böhme. Causes and effects of unanticipated numerical deviations in neural network inference frameworks. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL `http://papers.nips.cc/paper_files/paper/2023/hash/af076c3bdbf935b81d808e37c5ede463-Abstract-Conference.html`.

[5] Oreste Villa, Daniel Chavarría-Miranda, Vidhya Gurumoorthi, Andrés Márquez, and Sriram Krishnamoorthy. Effects of floating-point non-associativity on numerical computations on massively multithreaded systems. In *Proceedings of the Cray User Group Meeting (CUG)*, 2009. URL `https://cug.org/5-publications/proceedings_attendee_lists/CUG09CD/S09_Proceedings/pages/authors/01-5Monday/4C-Villa/villa-paper.pdf`.

[6] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, second edition, 2002. URL https://epubs.siam.org/doi/abs/10.1137/1.9780898718027.

[7] Linyi Li, Tao Xie, and Bo Li. SoK: Certified robustness for deep neural networks. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*, pages 1289–1310. IEEE, 2023. URL https://doi.org/10.1109/SP46215.2023.10179303.

[8] Aws Albarghouthi. Introduction to neural network verification. *Foundations and Trends in Programming Languages*, 7(1-2):1–157, 2021. ISSN 2325-1107. URL http://dx.doi.org/10.1561/2500000051.

[9] Xiaowei Huang, Daniel Kroening, Wenjie Ruan, James Sharp, Youcheng Sun, Emese Thamo, Min Wu, and Xinping Yi. A survey of safety and trustworthiness of deep neural networks: Verification, testing, adversarial attack and defence, and interpretability. *Comput. Sci. Rev.*, 37: 100270, 2020. URL https://doi.org/10.1016/j.cosrev.2020.100270.

[10] Changliu Liu, Tomer Arnon, Christopher Lazarus, Christopher A. Strong, Clark W. Barrett, and Mykel J. Kochenderfer. Algorithms for verifying deep neural networks. *Found. Trends Optim.*, 4(3-4):244–404, 2021. URL https://doi.org/10.1561/2400000035.

[11] Guy Katz, Clark Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In Rupak Majumdar and Viktor Kunčak, editors, *Computer Aided Verification*, pages 97–117, Cham, 2017. Springer International Publishing. ISBN 978-3-319-63387-9. URL https://doi.org/10.1007/978-3-319-63387-9_5.

[12] Götz Alefeld and Jürgen Herzberger. *Introduction to Interval Computation*. Academic Press, 1983. ISBN 9780120498208.

[13] Sven Gowal, Krishnamurthy Dvijotham, Robert Stanforth, Rudy Bunel, Chongli Qin, Jonathan Uesato, Relja Arandjelovic, Timothy A. Mann, and Pushmeet Kohli. On the effectiveness of interval bound propagation for training verifiably robust models. *CoRR*, abs/1810.12715, 2018. URL http://arxiv.org/abs/1810.12715.

[14] Kaidi Xu, Zhouxing Shi, Huan Zhang, Yihan Wang, Kai-Wei Chang, Minlie Huang, Bhavya Kailkhura, Xue Lin, and Cho-Jui Hsieh. Automatic perturbation analysis for scalable certified robustness and beyond. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/0cbc5671ae26f67871cb914d81ef8fc1-Abstract.html.

[15] Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin T. Vechev. Fast and effective robustness certification. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 10825–10836, 2018. URL https://proceedings.neurips.cc/paper/2018/hash/f2f446980d8e971ef3da97af089481c3-Abstract.html.

[16] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. An abstract domain for certifying neural networks. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. URL https://doi.org/10.1145/3290354.

[17] Huan Zhang, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh, and Luca Daniel. Efficient neural network robustness certification with general activation functions. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 4944–4953, 2018. URL https://proceedings.neurips.cc/paper/2018/hash/d04863f100d59b3eb688a11f95b0ae60-Abstract.html.

[18] Kaidi Xu, Huan Zhang, Shiqi Wang, Yihan Wang, Suman Jana, Xue Lin, and Cho-Jui Hsieh. Fast and Complete: Enabling complete neural network verification with rapid and massively parallel incomplete verifiers. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=nVZtXBI6LNn.

[19] Claudio Ferrari, Mark Niklas Müller, Nikola Jovanovic, and Martin T. Vechev. 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, 2022. URL https://openreview.net/forum?id=l_amHf1oaK.

[20] Mark Niklas Müller, Gleb Makarchuk, Gagandeep Singh, Markus Püschel, and Martin Vechev. PRIMA: general and precise neural network certification via scalable convex hull approximations. *Proc. ACM Program. Lang.*, 6(POPL), January 2022. URL https://doi.org/10.1145/3498704.

[21] Rüdiger Ehlers. Formal verification of piece-wise linear feed-forward neural networks. In Deepak D'Souza and K. Narayan Kumar, editors, *Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings*, volume 10482 of *Lecture Notes in Computer Science*, pages 269–286. Springer, 2017. URL https://doi.org/10.1007/978-3-319-68167-2_19.

[22] Souradeep Dutta, Susmit Jha, Sriram Sankaranarayanan, and Ashish Tiwari. Output range analysis for deep feedforward neural networks. In Aaron Dutle, César A. Muñoz, and Anthony Narkawicz, editors, *NASA Formal Methods - 10th International Symposium, NFM 2018, Newport News, VA, USA, April 17-19, 2018, Proceedings*, volume 10811 of *Lecture Notes in Computer Science*, pages 121–138. Springer, 2018. URL https://doi.org/10.1007/978-3-319-77935-5_9.

[23] Vincent Tjeng, Kai Y. Xiao, and Russ Tedrake. Evaluating robustness of neural networks with mixed integer programming. In *International Conference on Learning Representations*, 2017. URL https://arxiv.org/abs/1711.07356.

[24] Rudy Bunel, Jingyue Lu, Ilker Turkaslan, Philip H. S. Torr, Pushmeet Kohli, and M. Pawan Kumar. Branch and bound for piecewise linear neural network verification. *Journal of Machine Learning Research*, 21(42):1–39, 2020. URL http://jmlr.org/papers/v21/19-468.html.

[25] Shiqi Wang, Huan Zhang, Kaidi Xu, Xue Lin, Suman Jana, Cho-Jui Hsieh, and J. Zico Kolter. Beta-CROWN: Efficient bound propagation with per-neuron split constraints for neural network robustness verification. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan, editors, *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 29909–29921, 2021. URL https://proceedings.neurips.cc/paper/2021/hash/fac7fead96dafceaf80c1daffeae82a4-Abstract.html.

[26] Alessandro De Palma, Harkirat S. Behl, Rudy Bunel, Philip H. S. Torr, and M. Pawan Kumar. Scaling the convex barrier with active sets. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL https://openreview.net/forum?id=uQfOy7LrlTR.

[27] Sanjif Shanmugavelu, Mathieu Taillefumier, Christopher Culver, Oscar R. Hernandez, Mark Coletti, and Ada Sedova. Impacts of floating-point non-associativity on reproducibility for HPC and deep learning applications. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis, Atlanta, GA, USA, November 17-22, 2024*, pages 170–179. IEEE, 2024. URL https://doi.org/10.1109/SCW63240.2024.00028.

[28] Lucas C. Cordeiro, Matthew L. Daggitt, Julien Girard-Satabin, Omri Isac, Taylor T. Johnson, Guy Katz, Ekaterina Komendantskaya, Augustin Lemesle, Edoardo Manino, Artjoms Sinkarovs, and Haoze Wu. Neural network verification is a programming language challenge. In Viktor Vafeiadis, editor, *Programming Languages and Systems - 34th European Symposium on Programming, ESOP 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3-8, 2025, Proceedings, Part I*, volume 15694 of *Lecture Notes in Computer Science*, pages 206–235. Springer, 2025. URL https://doi.org/10.1007/978-3-031-91118-7_9.

[29] Cheng Zhang, Hanna Foerster, Robert D. Mullins, Yiren Zhao, and Ilia Shumailov. Hardware and software platform inference. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=kdmjVF1iDO.

[30] Jonas Möller, Lukas Pirch, Felix Weissberg, Sebastian Baunsgaard, Thorsten Eisenhofer, and Konrad Rieck. Adversarial inputs for linear algebra backends. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=R0WHut6jWo.

[31] Sanjif Shanmugavelu, Mathieu Taillefumier, Christopher Culver, Vijay Ganesh, Oscar R. Hernandez, and Ada Sedova. Robustness of deep learning classification to adversarial input on GPUs: asynchronous parallel accumulation is a source of vulnerability. *CoRR*, abs/2503.17173, 2025. URL https://doi.org/10.48550/arXiv.2503.17173.

[32] Kazuki Egashira, Mark Vero, Robin Staab, Jingxuan He, and Martin Vechev. Exploiting LLM quantization. *Advances in Neural Information Processing Systems*, 2024. URL https://papers.nips.cc/paper_files/paper/2024/file/496720b3c860111b95ac8634349dcc88-Paper-Conference.pdf.

[33] Kazuki Egashira, Robin Staab, Mark Vero, Jingxuan He, and Martin Vechev. Mind the gap: A practical attack on GGUF quantization. *International Conference on Machine Learning*, 2025. URL https://arxiv.org/abs/2505.23786.

[34] Thomas A. Henzinger, Mathias Lechner, and Đorđe Žikelić. Scalable verification of quantized neural networks. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(5):3787–3795, May 2021. doi: 10.1609/aaai.v35i5.16496. URL https://ojs.aaai.org/index.php/AAAI/article/view/16496.

[35] Pei Huang, Haoze Wu, Yuting Yang, Ieva Daukantas, Min Wu, Yedi Zhang, and Clark Barrett. Towards efficient verification of quantized neural networks. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38:21152–21160, 03 2024. doi: 10.1609/aaai.v38i19.30108. URL https://ojs.aaai.org/index.php/AAAI/article/view/30108/31956.

[36] Yedi Zhang, Zhe Zhao, Guangke Chen, Fu Song, Min Zhang, Taolue Chen, and Jun Sun. QVIP: An ILP-based formal verification approach for quantized neural networks. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450394758. URL https://doi.org/10.1145/3551349.3556916.

[37] Huan Zhang, Hongge Chen, Chaowei Xiao, Sven Gowal, Robert Stanforth, Bo Li, Duane Boning, and Cho-Jui Hsieh. Towards stable and efficient training of verifiably robust neural networks. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=Skxuk1rFwB.

[38] Antoine Miné. Relational abstract domains for the detection of floating-point run-time errors. In David Schmidt, editor, *Programming Languages and Systems*, pages 3–17, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-24725-8.

[39] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017. URL https://epubs.siam.org/doi/10.1137/141000671.

[40] David P. Sanders and Luis Benet. IntervalArithmetic.jl, 2014. URL https://github.com/JuliaIntervals/IntervalArithmetic.jl.

[41] Matthew Mirman, Timon Gehr, and Martin T. Vechev. Differentiable abstract interpretation for provably robust neural networks. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, Proceedings of Machine Learning Research, pages 3575–3583. PMLR, 2018. URL http://proceedings.mlr.press/v80/mirman18b.html.

[42] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=rJzIBfZAb.

[43] Eric Wong and Zico Kolter. Provable defenses against adversarial examples via the convex outer adversarial polytope. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 5286–5295. PMLR, 10–15 Jul 2018. URL https://proceedings.mlr.press/v80/wong18a.html.

[44] Konstantin Kaulen, Tobias Ladner, Stanley Bak, Christopher Brix, Hai Duong, Thomas Flinkow, Taylor T. Johnson, Lukas Koller, Edoardo Manino, ThanhVu H Nguyen, and Haoze Wu. The 6th international verification of neural networks competition (vnn-comp 2025): Summary and results, 2025. URL https://arxiv.org/abs/2512.19007.

[45] Huan Zhang, Shiqi Wang, Kaidi Xu, Linyi Li, Bo Li, Suman Jana, Cho-Jui Hsieh, and J. Zico Kolter. General cutting planes for bound-propagation-based neural network verification. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022. URL http://papers.nips.cc/paper_files/paper/2022/hash/0b06c8673ebb453e5e468f7743d8f54e-Abstract-Conference.html.

[46] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. Boosting robustness certification of neural networks. In *International Conference on Learning Representations (ICLR)*. 2019.

A Proofs

Lemma 1 Let $\delta(n)$ be defined by the equation $\prod_{i=1}^{n}(1+\delta_i) = 1+\delta(n)$. Then, if $|\delta_i| \leq \mu$ and $n\mu < 1$ then $|\delta(n)| \leq \frac{n\mu}{1-n\mu}$.

Proof. The proof is by induction (after [6]). For $n = 1$, $(1+\delta_1) = 1+\delta(1)$ where $|\delta(1)| = |\delta_1| \leq \mu \leq \frac{\mu}{1-\mu}$. For the inductive step, let us express $\delta(n)$ using $\delta(n-1)$.

$$\prod_{i=1}^{n}(1+\delta_i) = (1+\delta_n)(1+\delta(n-1)) = 1+\delta(n) \text{ thus } \delta(n) = \delta_n + (1+\delta_n)\delta(n-1). \quad (9)$$

Now, from the induction hypothesis,

$$|\delta(n)| \leq \mu + (1+\mu)\frac{(n-1)\mu}{1-(n-1)\mu} = \frac{\mu(1-(n-1)\mu) + (1+\mu)(n-1)\mu}{1-(n-1)\mu} = \quad (10)$$

$$\frac{n\mu}{1-(n-1)\mu} \leq \frac{n\mu}{1-n\mu}. \quad (11)$$

□

Proposition 1 Let $\hat{s}_o$ be the result of computing the inner product $\theta^\top x$ ($x, \theta \in \mathbb{R}^n$) in floating-point using expression tree o using any rounding method. Let $[l_\Delta, u_\Delta]$ be defined as in Eq. (4) where $\Delta = \frac{n\mu}{1-n\mu}$, and μ is the machine precision. Then, for any valid expression tree o, we have $\hat{s}_o \in [l_\Delta, u_\Delta]$

Proof. It is easy to see that for any expression tree o, after adding error terms and rearranging, we get the floating-point sum in the form of $\hat{s}_o = \sum_{i=1}^{n} \theta_i x_i \Delta_i$. All we need to prove is that Δ_i is the product of at most n relative error terms, because in that case, due to Lemma 1, $\Delta_i \in [1-\Delta, 1+\Delta]$, which in turn means that $l_\Delta \leq \hat{s}_o \leq u_\Delta$.

But the height of the expression tree $h(o)$ is at most n (and $h(o) = n$ if we first specify an ordering of the terms and then apply left-to-right addition) and the maximum number of error terms in any Δ_i is bounded by $h(o)$, and thus for any expression tree the number of error terms in the product Δ_i is at most n. □

Proposition 2 Let $[\hat{l}_o, \hat{u}_o]$ be the result of computing the inner product $\theta^\top x$ ($x = [l, u]$ and $\theta, l, u \in \mathbb{R}^n$) in interval floating-point with outward rounding using expression tree o. Let $[l_\Delta, u_\Delta]$ be defined as in Eq. (5) where $\Delta = \frac{n\mu}{1-n\mu}$, and μ is the machine precision. Then, for any valid expression tree o, we have $[\hat{l}_o, \hat{u}_o] \subseteq [l_\Delta, u_\Delta]$.

Proof. For any $x^* \in x$ and any expression tree o, the application of Proposition 1 gives us that $\hat{s}^*_o$, the floating-point result of $\theta^\top x^*$ as computed with o will be in the interval $[l^*_\Delta, u^*_\Delta]$, as given by substituting x^* into Eq. (4). However, since $x^* \in x$, we have $[l^*_\Delta, u^*_\Delta] \subseteq [l_\Delta, u_\Delta]$. □

Lemma 2. *Let* $[a, b]$, $[c, d]$, *and* $[n, m]$ *be real intervals. Then* $([a, b] + [c, d]) \cdot [n, m] \subseteq [a, b] \cdot [n, m] + [c, d] \cdot [n, m]$.

Proof. Let $s \in [a, b] + [c, d]$, hence $s = s_1 + s_2$ with $s_1 \in [a, b]$, $s_2 \in [c, d]$. Take any $t \in [n, m]$. Then $(s_1 + s_2) \cdot t = s_1 t + s_2 t$. Since $s_1 t \in [a, b] \cdot [n, m]$ and $s_2 t \in [c, d] \cdot [n, m]$, we obtain $s \cdot t \in [a, b] \cdot [n, m] + [c, d] \cdot [n, m]$. This holds for all $s \in [a, b] + [c, d]$ and $t \in [n, m]$. □

Lemma 3. *Let* $[a, b], [c, d]$ *be real intervals, and let* k *be a constant. Then* $([a, b] + [c, d]) \cdot k = [a, b] \cdot k + [c, d] \cdot k$.

Proof. If $k \geq 0$, then $[u, v] \cdot k = [uk, vk]$. Thus $([a, b] + [c, d]) \cdot k = ([a + c,\ b + d]) \cdot k = [ak + ck, bk + dk]$. On the other hand, $[a, b] \cdot k + [c, d] \cdot k = [ak, bk] + [ck, dk] = [ak + ck, bk + dk]$, which is equal to the previous expression.

If $k < 0$, then $[u, v] \cdot k = [vk, uk]$, and the same reasoning can be applied. □

Proposition 3 Let $L(x)$ denote the output neuron of a deep linear network ($x = [l, u]$ and $l, u \in \mathbb{R}^n$). Let $[\hat{l}_o, \hat{u}_o]$ be the smallest interval that includes the result of computing $L(x^*)$ for any $x^* \in x$ using floating-point arithmetic and the expression tree o. Let $[l_\Delta, u_\Delta] = L_\Delta(x)$ be the real interval of the output neuron, where the linear expression $L_\Delta(x)$ is computed recursively applying Eq. (7), where $\Delta = (n_{\max}\mu)/(1 - n_{\max}\mu)$, μ is the machine precision, and $n_{\max} = \max_i n_i$, the maximal input dimension over the neurons. Then, for any expression tree o such that neurons are computed numerically and not symbolically, we have $[\hat{l}_o, \hat{u}_o] \subseteq [l_\Delta, u_\Delta]$.

Proof. This proof is by induction according to the directed acyclic computational graph of the deep linear network. The input nodes are intervals $[l_i, u_i]$ ($i = 1, \ldots, n$). We will represent the input nodes $[l_i, u_i]$ also as linear expressions with the coefficient 1 for input i and coefficient 0 for the other inputs.

First, let's assume that the inputs of $L^i_\Delta(x)$ contain only source nodes (that is, neuron i is connected to the input directly). In this case, $L^i_\Delta(x)$ simplifies to Eq. (5), since $L^{i,j}_\Delta(x) = [l_j, u_j]$, where we can use Proposition 2.

As for the inductive step, we use a similar technique to that of Proposition 2 in that we prove the proposition for any scalar input $x^* \in x$. Consider the equation

$$\tilde{L}^i_\Delta(x) = \theta_{i,1} L^{i,1}(x)[1 - \Delta, 1 + \Delta] + \cdots + \theta_{i,n_i} L^{i,n_i}(x)[1 - \Delta, 1 + \Delta], \tag{12}$$

which is similar to Eq. (7) except we used the exact input neurons $L^{i,j}(x)$. Substituting x^*, $L^{i,j}(x^*) = a_{i,j} \in \mathbb{R}$ is a scalar and $\tilde{L}^i_\Delta(x^*) = [\tilde{l}^i, \tilde{u}^i]$ is a real interval. From Proposition 1 we known that $[\tilde{l}^i, \tilde{u}^i]$ contains all the floating-point realizations of the inner product $\sum_{j=1}^{n_i} \theta_{i,j} a_{i,j}$.

Now, if we could find intervals that contain every possible floating-point realization of $L^{i,j}(x^*)$ in deployment then we could substitute these intervals in Eq. (12), and the interval computed by this formula would cover every possible floating-point realization of $L^i(x^*)$. This is because we allow only for realizations o that compute neurons numerically, that is, expression trees that first compute the inputs of every neuron and then compute the inner product with the neuron parameter vector. Note that this excludes expression trees represented by symbolic computation that first compute the aggregated coefficients of each input and then in a final step multiply them with the input; but this is fine, because in deployment such expression trees are never used.

Fortunately, we do have intervals that contain every possible floating-point realization of $L^{i,j}(x^*)$, namely $[l^{i,j}_\Delta, u^{i,j}_\Delta] = L^{i,j}_\Delta(x^*)$, due to the inductive assumption! Substituting these into the right-hand side of Eq. (12) we get

$$[\overline{l}^i_\Delta, \overline{u}^i_\Delta] = \theta_{i,1}[l^{i,1}_\Delta, u^{i,1}_\Delta][1 - \Delta, 1 + \Delta] + \cdots + \theta_{i,n_i}[l^{i,n_i}_\Delta, u^{i,n_i}_\Delta][1 - \Delta, 1 + \Delta], \tag{13}$$

and from Proposition 2 we know that $[\overline{l}^i_\Delta, \overline{u}^i_\Delta]$ contains all possible floating-point realizations of $L^i(x^*)$, that is, $[\overline{l}^i_\Delta, \overline{u}^i_\Delta] \supseteq [\hat{l}^{*,i}_o, \hat{u}^{*,i}_o]$ for every allowed expression tree o.

This is nice, however, unfortunately $L^i_\Delta(x^*) \neq [\overline{l}^i_\Delta, \overline{u}^i_\Delta]$ in general, and so we need to prove $L^i_\Delta(x^*) \supseteq [\overline{l}^i_\Delta, \overline{u}^i_\Delta]$ to complete the proof. To do this, we will transform $[\overline{l}^i_\Delta, \overline{u}^i_\Delta]$ into $L^i_\Delta(x^*)$ using interval algebra, while showing that none of the steps makes the interval narrower.

Let us first expand the linear interval expression

$$L^{i,j}_\Delta(x) = x_1[l^j_1, u^j_1] + \cdots + x_n[l^j_n, u^j_n] = \sum_{k=1}^{n} x_k[l^j_k, u^j_k] \tag{14}$$

Using this notation, we can expand the right-hand side of Eq. (13) and then rearranging, we get

$$\sum_{j=1}^{n_i} \theta_{i,j}[1-\Delta, 1+\Delta] \sum_{k=1}^{n} x^*_k[l^j_k, u^j_k] \subseteq \sum_{j=1}^{n_i} \sum_{k=1}^{n} \theta_{i,j}[1-\Delta, 1+\Delta] x^*_k[l^j_k, u^j_k] \tag{15}$$

due to Lemma 2. Next, we rearrange again to get $L^i_\Delta(x)$ as

$$\sum_{j=1}^{n_i} \sum_{k=1}^{n} \theta_{i,j}[1-\Delta, 1+\Delta] x^*_k[l^j_k, u^j_k] = \sum_{k=1}^{n} x^*_k \sum_{j=1}^{n_i} \theta_{i,j}[1-\Delta, 1+\Delta][l^j_k, u^j_k] = L^i_\Delta(x^*). \tag{16}$$

The equation above holds because, first, the order of addition of the intervals does not matter, the interval sum is commutative (recall that we are using real arithmetic here). Second, since x^* is a vector of scalars, according to Lemma 3 the real interval result does not change due to factoring out x^*_k.

This proves that $L^i_\Delta(x^*) = [l^{*,i}_\Delta, u^{*,i}_\Delta] \supseteq [\overline{l}^i_\Delta, \overline{u}^i_\Delta] \supseteq [\hat{l}^{*,i}_o, \hat{u}^{*,i}_o]$. Since $x^* \in x$, $L^i_\Delta(x)$ contains all floating-point realizations of every point in x, which concludes the proof.

□

B Overflow and Underflow

Let us first consider the case of a regular inner product

$$\hat{s} = \sum_{i=1}^{n} \theta_i x_i, \tag{17}$$

where θ_i and x_i are scalars representable in the floating-point representation of the deployment.

B.1 Overflow

Detecting overflows during the multiplications $\theta_i x_i$ is straightforward, so we turn our focus to the sum. We can easily flag possible overflows by applying a worst-case analysis. We only need to compute a sound lower and upper bound to the possible smallest and largest result, respectively. Given a sum $\sum_{i=1}^{n} \theta_i x_i$, the idea is that we compute the lower bound considering only the negative summands and compute the upper bound considering only the positive summands. We use the notation $S_+ = \{\theta_i x_i | \theta_i x_i > 0\}$ and $S_- = \{\theta_i x_i | \theta_i x_i < 0\}$. Then, let

$$\check{l} = \sum_{i \in S_-} \theta_i x_i[1-\Delta, 1+\Delta], \text{and } \check{u} = \sum_{i \in S_+} \theta_i x_i[1-\Delta, 1+\Delta], \tag{18}$$

where we used Eq. (4) for both sums.

If, when computing $\check{l}$ and $\check{u}$, we use the same precision used in deployment then if $\check{u} \neq$ Inf and $\check{l} \neq -$Inf then overflow is not possible. Otherwise we signal a potential overflow. (If we use a higher precision to compute the bounds, then we have to check whether $\check{u}$ or $\check{l}$ would translate to Inf or $-$Inf, respectively, in the minimal deployment precision.)

B.2 Underflow

Underflow is possible during both multiplication and addition. We discuss these separately.

B.2.1 Multiplication

Multiplication can easily underflow. Now, let us assume that during the summation there is no underflow, but one $\theta_k x_k$ term underflows for some k, that is, $\theta_k \neq 0$, $x_k \neq 0$, and $\theta_k x_k = 0$ using the deployment precision. We can show that even in this scenario, our bounding can be made to remain floating-point sound.

We need to make sure that $\hat{s} \in [l_\Delta, u_\Delta]$ (defined in Eq. (4)). Now, consider the following bounds: $[l^*_\Delta, u^*_\Delta] = \sum_{i \neq k} \theta_i x_i [1-\Delta, 1+\Delta]$. It is evident that $\hat{s} \in [l^*_\Delta, u^*_\Delta]$, so we only need to make sure that the $\theta_k x_k [1-\Delta, 1+\Delta]$ term within the original sum just widens this bound, or in other words, it contains 0.

We again look at two cases. First, if the verification uses the same precision as the deployment then this is guaranteed because we compute the multiplication using interval arithmetic with outward rounding, so the interval result of $[\check{l}, \check{u}] = \theta_k x_k$ bounds both 0 (the floating-point result) and a non-zero number (the real result). Computing $[\check{l}, \check{u}][1-\Delta, 1+\Delta]$, we get $[\check{l} \cdot (1+\Delta), \check{u} \cdot (1+\Delta)]$ that obviously contains 0. If the verification uses a higher precision, than we make sure we use the deployment (lower) precision for the outward rounding of the multiplication. Note that widening any intervals always keeps the soundness property of verification.

To sum up, this means that if any of the multiplications underflow, then the bounding of $\hat{s}$ will remain to be sound.

B.2.2 Summation

Here, we need to consider two cases based on whether the environment uses subnormal numbers. If the environment computes with subnormal numbers, then underflow during the summation is not possible, we only need to show that our bounding is sound even with subnormal numbers. When subnormal numbers are rounded to zero (flush-to-zero mode), then underflow is possible, in this case our original widening is no longer sound, but we can extend it with an additional, extremely small widening that ensures our algorithm remains sound even in the presence of potential underflow.

In this section, since we have dealt with the multiplication operation first, we need to consider only the sum function $\hat{s} = \sum_{i=1}^{n} x_i$ where x_i are representable.

With subnormal numbers we only need to show that the backward error expression defined in Eq. (3) is valid even if subnormal numbers are allowed as addends x_i. To do this, it suffices to consider the binary case $(x_1 + x_2)(1+\delta)$, and show that it can express the floating point error even if x_1, x_2, or the floating point result of $x_1 + x_2$ is a subnormal number. In fact, we only need to consider whether the real value of the sum $x_1 + x_2$ is subnormal or not. If not, then the error term works by definition. If the sum is subnormal, we can use the following statement.

Lemma 4. *Let $\mathbb{F}$ be an IEEE 754 standard binary floating-point representation. For any $x, y \in \mathbb{F}$, if the computed floating-point sum $s = fl(x+y)$ is a subnormal number or zero, then the addition is exact, that is, $fl(x+y) = x + y$.*

Proof. Let m denote the smallest positive representable floating-point number in $\mathbb{F}$ (the minimum positive subnormal number), and let N_{min} denote the smallest positive normal number. According to the floating-point standard, the distance between two adjacent representable numbers is m within the subnormal domain, or $m \cdot 2^k$ for some k within the normal domain. Consequently, every representable floating-point number in $\mathbb{F}$ is an integer multiple of m. Therefore, for our given $x, y \in \mathbb{F}$, there exist integers $A, B \in \mathbb{Z}$ such that:

$$x = A \cdot m \tag{19}$$

$$y = B \cdot m \tag{20}$$

The exact mathematical sum can be expressed as

$$x + y = (A \cdot m) + (B \cdot m) = (A + B)m \tag{21}$$

We assumed that $(A+B)m < N_{min}$, that is, $(A+B)m$ is a subnormal number and an integer multiple of m, which implies that it is representable. □

This of course means that $(x_1 + x_2)(1 + \delta) = x_1 + x_2$ and so it is safe to use the backward error estimation approach on sums, if subnormal numbers are allowed. In other words, in this case a sum never results in an underflow.

Without subnormal numbers. In some deployment environments, subnormal numbers are rounded to zero due to some performance gains (some compilers support the DAZ (denormals-are-zero) and FTZ (flush-to-zero) flags). Considering such an environment, our widening approach is no longer sound, since we cannot apply the multiplicative non-zero relative error term $(1 + \delta)$ to model the floating point result of a sum within real arithmetic, because the sum can now underflow. To deal with this case, we developed an algorithm that applies an additional widening to include potentially underflowing additions. This way, the verification can remain sound even in the presence of potential underflows.

First, let us make two observations.

Observation 1. *When every number in a sum is of the form $k \cdot N_{min}$, where N_{min} is the smallest normal number and $k \in \mathbb{Z}$ is an integer, then every intermediate result (and the final sum) will be a normal number or zero.*

Observation 2. *Let $\mathbb{F}$ be an IEEE 754 standard binary floating-point representation, $s_o = x_1 + \cdots + x_n$ be the floating-point sum, where $s_o, x_1, \cdots, x_n \in \mathbb{F}$, and o denote a binary expression tree determining the evaluation order. Then $s_{min} \leq z_{min}$, where $s_{min} = \min_o s_o$, $z_{min} = \min_o z_o$ and $z_o = y_1 + \cdots + y_n$, where $x_i \leq y_i$ for all i. This property remains valid even when DAZ and FTZ modes are enabled.*

Proof. Observation 2 follows directly from the monotonicity of binary floating-point addition, which remains valid even when subnormals are rounded to zero. □

IBP. Consider the sum of floating-point intervals $[l_o, u_o] = [l_1, u_1] + \cdots + [l_n, u_n]$, where $l_i, u_i \in \mathbb{F}$ and o is a given expression-tree. Based on the definition of interval addition, we get $l_o = l_1 + \cdots + l_n$ and $u_o = u_1 + \cdots + u_n$ computing the bounds using expression-tree o (l_o is computed by rounding the non-representable sub-results down, and u_o is computed by rounding them up). If every bound l_i, u_i is of the form $k \cdot N_{min}$, Observation 1 guarantees that every intermediate result will be of the form $k \cdot N_{min}$. Consequently, our original bounds l_Δ, u_Δ remain sound, since no intermediate results will be in the subnormal range, and so we can apply the $(1 + \delta)$ error term to model the numerical error and compute $[l_\Delta, u_\Delta]$ as usual.

To handle cases where a bound is not of the form $k \cdot N_{min}$ (making $[l_\Delta, u_\Delta]$ unsound), *we simply round such bounds outward to the nearest integer multiple of N_{min}*. We can do this because Observation 2 ensures monotonicity, so widening individual addends will widen the resulting interval as well, which is always safe to do from the point of view of soundness.

Symbolic. In the symbolic case, we can apply an approach similar to that of IBP. We only need to ensure that the real bounds of each expression $\check{L}^{i,j}(x) = \theta_{i,j} L^{i,j}(x)$ are of the form $k \cdot N_{min}$ after concretization. If this condition holds, the proof of Proposition 3 holds, since Eq. (13) remains sound (and the back-substitution step strictly widens the output bounds).

Let the concretized real-valued bounds of $\check{L}^{i,j}(x)$ be $[\check{l}_\Delta^{i,j}, \check{u}_\Delta^{i,j}]$. We need to compute tight, sound bounds for the real values $\check{l}_\Delta^{i,j}$ and $\check{u}_\Delta^{i,j}$, and then check whether the floating point numbers inside these bounds are all of the form $k \cdot N_{min}$.

Fortunately, we can use interval arithmetic to compute the bounds $\check{l}_\Delta^{i,j} \in [L^{\check{l}_\Delta^{i,j}}, U^{\check{l}_\Delta^{i,j}}]$ and $\check{u}_\Delta^{i,j} \in [L^{\check{u}_\Delta^{i,j}}, U^{\check{u}_\Delta^{i,j}}]$. We only need to check whether every floating-point representable number within these intervals is an integer multiple of N_{min}. This can be done by checking whether $[L^{\check{l}_\Delta^{i,j}}, U^{\check{l}_\Delta^{i,j}}] \cap D = \emptyset$ and $[L^{\check{u}_\Delta^{i,j}}, U^{\check{u}_\Delta^{i,j}}] \cap D = \emptyset$, where $D = (-N_{min}/\mu, 0) \cup (0, N_{min}/\mu)$ and μ is the machine epsilon. Note that we prefer this method because it is a fast and sound test, but it may give a false negative result.

If this condition is not met (that is, if there exists a representable number $y \in [L^{\check{l}_\Delta^{i,j}}, U^{\check{l}_\Delta^{i,j}}] \cap \mathbb{F}$ or $y \in [L^{\check{u}_\Delta^{i,j}}, U^{\check{u}_\Delta^{i,j}}] \cap \mathbb{F}$ such that $y \neq k \cdot N_{min}$ for any $k \in \mathbb{Z}$) we must widen the real range of

the expression to maintain soundness. In practice, just as with IBP, we can easily achieve this by adjusting the constant term of $\check{L}^{i,j}(x)$.

B.3 Putting it together

We have described how to deal with interval-valued inner products. Although we have touched upon this, let us summarize here, how to integrate these techniques into our verifiers.

In FPSoundIBP, these techniques can be directly included while propagating the interval bounds through the networks, checking (and maybe widening) each interval that is being computed.

In the case of FPSoundSymbolic, we need to temporarily concretize the interval linear expression of each intermediate neuron and check these concretized intervals using the technique presented above. These concretized intervals are then thrown away, because the propagation is symbolic, however, we might widen the interval linear expression that is being propagated by adjusting the constant terms if we detect a potential underflow (note that this never happened over our benchmark of models, and the widening is extremely small even if it happens), and we might detect and flag potential overflows as well.

C Backdoors

Here, we describe the motivation for the kinds of backdoors that are activated by deployment environment features, and describe specific backdoor construction techniques that are applied in our empirical analysis.

The content of this section is mostly a recap of related work, specifically: [1, 2], but we also include results with our new algorithms in Table 4 at the end of the section.

C.1 Motivation and Attack Scenarios

The backdoors we used were introduced by [1]. Their main observation was that deployed networks and theoretical networks are different mathematical objects, and deployed networks in different deployment environments are also different mathematical objects. Therefore deployed networks might exhibit behavior that cannot be observed in a full-precision evaluation, moreover, a deployed network in one environment might produce behaviors that cannot be observed in another environment.

Now, the question is whether we can artificially enlarge the differences in behavior between different deployment environments? The main motivation would be to create neural network definitions that are especially suitable for testing practical verifiers. However, if we can create such networks, they would also represent an attack vector, where an attacker can craft a network that behaves entirely differently in a specific environment.

[1] showed that we can indeed create networks that behave completely differently in different deployment environments, and they proposed a methodology as well to create such networks. The key idea is that we identify a specific property of the targeted environment, for example, a unique expression tree that is characteristic to the environment, or maybe simply the applied floating-point format. We then create a so-called detector neuron, that returns zero in the target environments, and non-zero in the rest of the environments. This output is then used as a switch to select between arbitrarily different subsets of the network to produce the output.

C.2 Backdoor Construction

The adversarial networks we tested were introduced in [1] and rely on the backdoor structure illustrated in Fig. 1. The exact configuration of this structure, including the definitions of the α and β parameters, is given in the Appendix of [1].

The base model in all cases is an MNIST network first published by [43]. The network consists of two convolutional layers with stride 2, using 16 and 32 filters of size 4×4, respectively, followed by a fully connected layer with 100 neurons. All layers employ ReLU activations. The model was trained to be robust against adversarial perturbations within an l_∞-ball of radius 0.1.

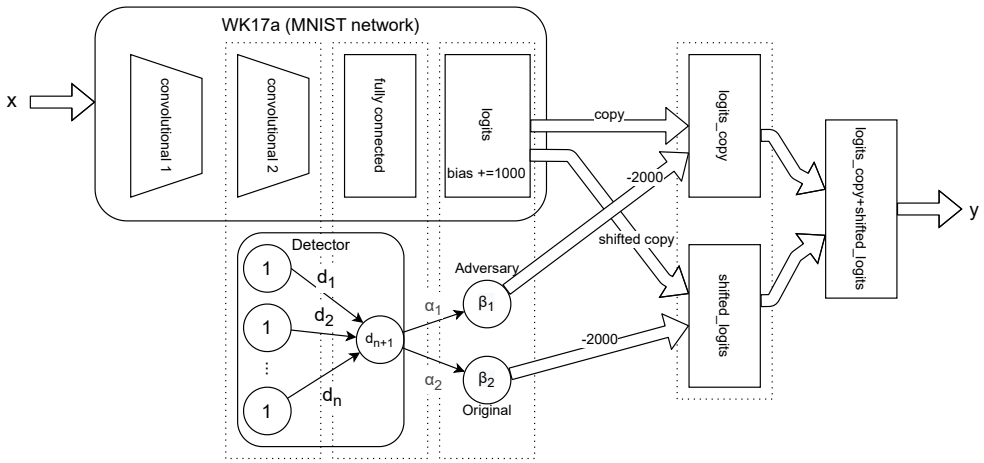

Figure 1: Backdoor integration. Circles denote ReLU neurons, with the bias term indicated inside each circle. Simple arrows represent connections, with their corresponding weights.

The adversarial networks mainly differ in whether the adversarial behavior exploits the limited precision or the non-associative property of floating-point representation. In all these networks, there is a detector neuron whose output highly depends on the expression tree or the precision with which the sum is evaluated. More precisely, these adversarial networks build on the fact that the output of their detector neuron can be either zero or a positive number. Each environment is classified as safe or adversarial based on the output of the detector neuron. The adversarial structure also incorporates the definition of an adversarial mechanism—for example, shifting the network's output by one position—which is activated when the detected environment is adversarial.

For the definitions, we use a special number ω, which is defined as the smallest representable number such that the next representable number after ω is $\omega + 2$. For binary32 format $\omega = 2^{24}$ and for binary64 format $\omega = 2^{53}$. We assume that the rounding mode is round-to-nearest, which results in $\omega + 1 = \omega$.

Exploiting numerical precision. The adversarial network that exploits numerical precision has a detector neuron defined as $c = \omega + 1 - \omega$. We have $c = 0$ if the precision of the deployment is the same as the precision represented by ω, and $c = 1$ if higher precision is used. To detect both behaviors, our verifier always assumes the lowest precision that may occur in deployment—32-bit for these networks—and propagates expressions with that precision. Since lower precision leads to larger numerical errors, these bounds also cover the outputs of more precise arithmetic.

Exploiting non-associativity. [1] presented three expression-tree based adversarial networks. They mainly differ in how easily their adversarial behavior can be detected using interval arithmetic. The authors used the term *default order* to refer to the standard left-to-right addition, typically used in single-threaded CPU environments, and configured their networks so that the detector neuron's output following this order results in normal behavior.

The detector neuron of the *Order1* network is defined as

$$c = \underbrace{\underbrace{\frac{\omega}{h_1} + \cdots + \frac{\omega}{h_1}}_{h_1\times} + 1 + \underbrace{\frac{-\omega}{h_1} + \cdots + \frac{-\omega}{h_1}}_{h_1\times} + \cdots}_{h_2\times}.$$

The default order produces output $c = 0$ in floating-point, indicating normal behavior, while any other output results in adversarial behavior. The full-precision value (which is $c = h_2$) is always bounded by interval arithmetic regardless of the order, ensuring that adversarial behavior is detected in any expression-tree if outward rounding is used.

Table 4: The vulnerability of the verifiers to the precision attack (Pr.), and three order attacks (Order1 (O1), Order2 (O2), Order3 (O3)), as well as to the attack of [2]. U means the verifier fails (unsound), S means the verifier does not fail (sound). We used a precision attack that is adversarial to the environment used by the given verifier. Related work results are taken from ([1]).

Verifier	Ver. Env.	Bounding	Pr.	O1	O2	O3	[2]
MIPVerify ([23])	64-bit, CPU	IBP	U	S	U	U	U
MN-BAB ([19])	64-bit, GPU	Symbolic	U	S	U	U	U
β-CROWN BaB ([25])	32-bit, CPU	Symbolic	U	S	U	U	[no 32-bit model]
β-CROWN BaB ([25])	64-bit, CPU	Symbolic	U	S	U	U	S
β-CROWN BaB ([25])	64-bit, GPU	Symbolic	U	S	U	U	S
GCP-CROWN ([45])	64-bit, CPU	Symbolic	U	S	U	U	S
DeepPoly ([16])	64-bit, CPU	Symbolic	U	S	S	U	S
RefinePoly ([16])	64-bit, CPU	Symbolic	U	S	S	U	U
DeepZ ([15])	64-bit, CPU	Symbolic	U	S	S	U	S
RefineZono ([46])	64-bit, CPU	Symbolic	U	S	S	U	[Gurobi error]
FPSoundIBP (ours)	32-bit, CPU	IBP	S	S	S	S	S
FPSoundSymbolic (ours)	32-bit, CPU	Symbolic	S	S	S	S	S

The detector of *Order2* is defined as

$$c = \underbrace{\frac{2}{h} + \cdots + \frac{2}{h}}_{h\times} + \omega - \omega.$$

This adversarial network is configured so that 0 triggers the adversarial behavior (the output according to the default expression tree is 2). Bounding 0 is harder in this case, because interval arithmetic using the default order will not cover 0. However, many other expression trees cover 0, namely those where ω appears before position $h/2$.

Order3 has a detector neuron defined as

$$c = \underbrace{1 + \cdots + 1}_{h\times} + \omega - \omega.$$

Considering an interval arithmetic, only two expression trees will result in an interval that bounds 0: when ω is the first or the second summand. These expression trees are highly extreme, and the probability that they will be evaluated during verification is extremely low.

[2] defined the detector neuron as $c = \omega - \omega + 1$. Their backdoor structure is slightly different from [1], as the detector neuron's trigger also depends on the input, but this is not important for our discussion. This backdoor does not directly target the bounding algorithms described here, but rather LP/MILP-based verifiers, where the solver might evaluate c starting the addition with 1.

We tested the two algorithms introduced in Section 4.3: FPSoundIBP and FPSoundSymbolic. To implement our algorithms, we used Julia [39], where the interval arithmetic operations are based on the IntervalArithmetic.jl package ([40]). The results are summarized in Table 4, along with several sound verifiers from the literature, indicating that our algorithms do not fail on any of the attacks, as predicted by theory, unlike the rest of the verifiers.

D Experimental Details

Hardware and Software Details. Our algorithms are implemented in Julia [39] (tested on version 1.12), using the JuliaIntervals package for interval computations. All experiments were conducted on a workstation equipped with an Intel(R) Core(TM) Ultra 7 265KF CPU and 64 GB of RAM. For baseline comparisons involving the DeepZ [15] and DeepPoly [16] algorithms, we used their original implementations provided by the ERAN framework[1]. ERAN relies on a Python frontend and a highly optimized C-based backend (ELINA) for efficient operations over abstract domains.

Tables. Tables 2 and 3 summarize the output width statistics collected across all verification instances. During the verification process of each instance, we recorded the average and maximum logit widths.

[1] `https://github.com/eth-sri/eran`

These instance-level statistics were aggregated over the test set to compute the global mean and maximum values (rounded to two decimal places). The average value is the average of instance-level averages, and the maximum value is the maximum of instance level maximums. The width overhead is the difference of the precise value of the previous corresponding columns (avg and max).

Regarding the runtime measurements reported in our evaluations, we excluded the initial compilation overhead inherent to Julia. Since Julia is a Just-In-Time (JIT) compiled language, including the first-run compilation time would misrepresent the actual computational complexity of the verification algorithms, therefore, all reported execution times reflect the pure algorithmic runtime after an initial warm-up phase.

Software release. We release the software under the Apache 2.0 license at `https://github.com/szegedai/sound-verification-deployed`.

Software Dependencies To ensure the reproducibility of our results, we list the exact versions of the core software libraries utilized in our environments.

Proposed Framework (Julia Environment):

- Julia: v1.12.6
- IntervalArithmetic.jl: v0.23.0
- Flux.jl: v0.16.5
- NNlib.jl: v0.9.31
- MLDatasets.jl: v0.7.18

ERAN (Python Environment):

- Python: v3.8.20
- PyTorch: v2.1.2
- TensorFlow: v2.13.1
- NumPy: v1.23.5
- ONNX: v1.8.0 (with onnxruntime v1.19.2)

E Measurements Regarding over-Approximation

Here, we examine in more detail the amount of over-approximation of our method. Since our theoretical results do not imply that our bounds are tight in general, we can expect over-approximation that might depend on a number of features of the verified network.

First, we analyze over-approximation by looking at the scaling of our error bound Δ. We then examine how over-approximation depends on the number of summands, when bounding the range of a large sum.

E.1 Scaling the Parameter Δ

Here, we consider the sensitivity of our error parameter Δ, and experiment with scaled versions $\alpha\Delta$, with $\alpha \in [0, 1]$. Recall that, when $\alpha < 1$, soundness is not currently guaranteed, our theoretical guarantees require $\alpha = 1$. Note that we have not proven that Δ is tight.

We first repeated the experiments in Table 4 with $\alpha < 1$, the results are shown in Table 5. We can observe that the verification result remains sound when $\alpha = 0.25$, for all the tested networks. However, for $\alpha \leq 0.2$, the result is incorrect for Order3, the verifier misses the backdoor.

However, one needs to be careful when interpreting these results. The smallest α that preserves correctness—which determines the practical extent of over-estimation—depends strongly on both *the specific numerical pattern of weights in the adversarial network and the order in which the interval sum is evaluated*. We illustrate this with the backdoor pattern of the Order3 network, where the sum is

$$\underbrace{1 + \cdots + 1}_{512\times} + \omega - \omega. \tag{22}$$

Table 5: Sensitivity ablation via using $\alpha\Delta$ instead of Δ for several α values. The vulnerability of the verifiers to the precision attack (Pr.), and three order attacks (Order1 (O1), Order2 (O2), Order3 (O3)), as well as to the attack of [2]. U means the verifier fails (unsound), S means the verifier does not fail (sound). We used a precision attack that is adversarial to the environment used by the given verifier.

Verifier	Ver. Env.	Pr.	O1	O2	O3	[2]
FPSoundIBP (ours)	32-bit, CPU	S	S	S	S	S
FPSoundSymbolic (ours)	32-bit, CPU	S	S	S	S	S
FPSoundIBP (ours) $\alpha = 0.25$	32-bit, CPU	S	S	S	S	S
FPSoundSymbolic (ours) $\alpha = 0.25$	32-bit, CPU	S	S	S	S	S
FPSoundIBP (ours) $\alpha = 0.2$	32-bit, CPU	S	S	S	U	S
FPSoundSymbolic (ours) $\alpha = 0.2$	32-bit, CPU	S	S	S	U	S
FPSoundIBP (ours) $\alpha = 0$	32-bit, CPU	S	S	U	U	S
FPSoundSymbolic (ours) $\alpha = 0$	32-bit, CPU	S	S	U	U	S

Table 6: The output interval on the $1 + 1 + ... + 1 + 2^{53} - 2^{53}$ (Order3 detector) pattern.

Algorithm	**Bound** $[l, u]$
IBP	$[512, 512]$
IBP + [38]	$[500, 524]$
FPSoundIBP	$[-1548, 2573]$

When evaluated in the default left-to-right order using interval arithmetic, the result is $[512, 512]$, meaning that no numerical errors arise in this particular summation order. However, the floating-point values that need to be bounded are in $[0, 512]$. In contrast, FPSoundIBP with $\alpha = 1$ yields the interval $[-1548, 2573]$, which introduces considerable over-estimation. Table 6 includes these output intervals, along with the one used by the (not practically sound) error bounding technique of [38], while Table 7 shows the output ranges obtained for different values of the scaling parameter α with FPSoundIBP.

Let us now consider a different ordering of the sum, namely

$$\omega + \underbrace{1 + \cdots + 1}_{512\times} - \omega. \tag{23}$$

When evaluated with interval arithmetic (that is, $\alpha = 0$), the resulting interval is $[0, 1024]$, which correctly bounds the floating-point range $[0, 512]$. This means that α can safely be set to 0 for this expression tree while preserving soundness (IBP is already sound). In contrast, FPSoundIBP ($\alpha = 1$) yields the interval $[-2059, 3083]$—an interval that is wider than the one obtained with the original expression-tree. This example clearly demonstrates that the degree of over-estimation introduced by our method is sensitive to the expression tree. Also note that *the soundness itself is not sensitive to the expression tree: this is our main result*. But the width of the interval does depend on the particular expression tree.

E.2 The Effect of the Number of Summands

We generated n i.i.d. random numbers uniformly distributed in $[0, 1)$ and compared the diameters of the resulting output intervals produced by no error bounding (IBP), the technique of [38], and our FPSoundIBP. The results are shown in Table 8. It can be seen that the amount of *relative* over approximation also grows with n, a phenomenon predicted by theory, since Δ depends on n. Fortunately, in practice the fan-in of neurons is of a relatively moderate size.

F Illustrative example for FPSoundIBP

Let us compute FPSoundIBP in full detail, step-by-step, for a simple input expression:

$$1 + 1 + 2^{53} - 2^{53}, \tag{24}$$

Table 7: Lower and upper bounds obtained in the default order of the Order3 backdoor pattern by FPSoundIBP with various values of α.

Scale	Lower	Upper
0.00	512.0	512.0
0.05	406.0	618.0
0.10	304.0	721.0
0.15	200.0	825.0
0.20	98.0	926.0
0.25	**-6.0**	**1030.0**
0.30	-108.0	1133.0
0.35	-210.0	1234.0
0.40	-314.0	1338.0
0.45	-416.0	1441.0
0.50	-520.0	1545.0
0.55	-621.0	1646.0
0.60	-724.0	1749.0
0.65	-828.0	1853.0
0.70	-929.0	1954.0
0.75	-1033.0	2058.0
0.80	-1136.0	2161.0
0.85	-1237.0	2262.0
0.90	-1341.0	2366.0
0.95	-1444.0	2469.0
1.00	-1548.0	2573.0

Table 8: Comparison of the output interval lengths on sums of n i.i.d. random numbers drawn uniformly from $[0, 1)$ using the binary64 floating-point representation.

n	IBP	FPSoundIBP	IBP + [38]
10^1	$4.99 \cdot 10^{-16}$	$3.67 \cdot 10^{-14}$	$7.93 \cdot 10^{-15}$
10^2	$8.53 \cdot 10^{-14}$	$3.63 \cdot 10^{-12}$	$6.91 \cdot 10^{-13}$
10^3	$1.06 \cdot 10^{-12}$	$3.49 \cdot 10^{-10}$	$8.23 \cdot 10^{-12}$
10^4	$3.43 \cdot 10^{-11}$	$3.58 \cdot 10^{-8}$	$2.49 \cdot 10^{-10}$
10^5	$2.32 \cdot 10^{-9}$	$3.57 \cdot 10^{-6}$	$1.63 \cdot 10^{-8}$
10^6	$1.42 \cdot 10^{-7}$	$3.55 \cdot 10^{-4}$	$9.98 \cdot 10^{-7}$
10^7	$2.62 \cdot 10^{-6}$	$3.54 \cdot 10^{-2}$	$1.84 \cdot 10^{-5}$
10^8	$4.87 \cdot 10^{-5}$	3.54	$3.41 \cdot 10^{-4}$
10^9	$6.4 \cdot 10^{-3}$	354.328	$4.48 \cdot 10^{-2}$

assuming a binary64 floating-point representation. In this representation, the true range of every possible expression tree for this sum is $[0, 2]$. For example, if we sum from left to right we get 2, and if we sum in the order of $1 + 2^{53} + 1 - 2^{53}$, we get zero. No values outside this range are possible (one can test every possible ordering).

Verified expression tree. First of all, FPSoundIBP works with an explicit expression tree, that we need to fix. We proved that the output interval will contain every possible expression tree, independently of which expression tree we chose here, nevertheless, the method needs a specific tree, and the actual output interval depends on this tree. Let this expression tree be represented by computing $((1 + 1) + 2^{53}) - 2^{53}$ from left to right.

Plain IBP fails. The IBP algorithm without our error bounding computes the interval $[2, 2]$ on this expression tree, because every operation in the tree (namely, $1 + 1$, $2 + 2^{53}$, and $(2 + 2^{53}) - 2^{53}$) can be performed without any rounding errors in floating-point, thus the result is a zero-length interval around the exact value. However, as we saw above, there is another expression tree which results in zero, and zero is not included in $[2, 2]$. Also note that if we used the order $1 + 2^{53} + 1 - 2^{53}$ then IBP would be sound (but only by chance).

The computations of FPSoundIBP. First, let us compute Δ according the its definition $\Delta = \frac{n\mu}{1-n\mu}$ given in Section 4.1. Here, $n = 4$ is the number of summands and $\mu = 2^{-52}$ is the machine epsilon, which gives

$$\Delta = \frac{4 \cdot 2^{-52}}{1 - 4 \cdot 2^{-52}} \approx 8.88 \cdot 10^{-16}. \tag{25}$$

Now, we can substitute into Eq. (4) which gives

$$1[1 - \Delta, 1 + \Delta] + 1[1 - \Delta, 1 + \Delta] + 2^{53}[1 - \Delta, 1 + \Delta] - 2^{53}[1 - \Delta, 1 + \Delta] = [-18, 23]. \tag{26}$$

This interval over-approximates the true interval ($[0, 2]$), nonetheless, it is a sound over-approximation that is guaranteed to cover every possible expression tree.

Note that FPSoundIBP works with Eq. (5) with interval inputs, but if the input contains scalars (that is, zero length intervals like $[1, 1]$) then this equation falls back to the inner product case that we used above.

G Verifying an Interesting Example Symbolically

Here, we present an interesting neural network instance to demonstrate an issue with symbolic propagation methods. The issue is that—due to the extremely different expression trees that are used during verification and in deployment—the output of numerically and symbolically evaluated networks can drastically differ when using floating-point arithmetic. Our example network is designed to illustrates this, and we also explain how our symbolic error propagation method FPSoundSymbolic bounds both symbolic and numeric evaluations correctly for this example.

Consider a neural network with two inputs $x = (x_1, x_2)$, n layers, and a single output. Let each hidden layer contain two neurons: $L^{i,1}$, and $L^{i,2}$, except for the final layer, which has only one neuron $L^{n,2}$. The upper neuron $L^{i,1}$ is connected to both neurons in the subsequent layer, while the lower neuron $L^{i,2}$ is connected only to the lower neuron of the next layer, $L^{i+1,2}$. All connection weights are fixed to 1, and all biases are set to 0. The structure is illustrated in Fig. 2.

It is straightforward to observe that, due to the connectivity pattern and the choice of unit weights and zero biases, the contribution of x_1 accumulates linearly across the n layers, while x_2 propagates directly to the output. Consequently, under symbolic evaluation, the network output is $x_2 + nx_1$.

However, when the network is evaluated using standard floating-point arithmetic, numerical errors may arise. For instance, if we choose $x_1 = 1$ and $x_2 = 2^{53}$ then, under the assumption that the sum $2^{53} + 1$ is rounded down to 2^{53} due to limited precision in IEEE 754 double-precision arithmetic (the contribution of x_1 is effectively lost at each layer). Consequently, the network will numerically evaluate to 2^{53}, instead of the symbolic output, which would be close to $2^{53} + n$, with only a small possible deviation due to floating-point rounding. This means that *arbitrarily large discrepancies can arise between symbolic and floating-point evaluations*, growing with the number of layers n.

In double-precision floating-point arithmetic, the machine epsilon is given by $\mu = 2^{-52}$. In our network, there are two types of neurons: those with a single input and those with two inputs.

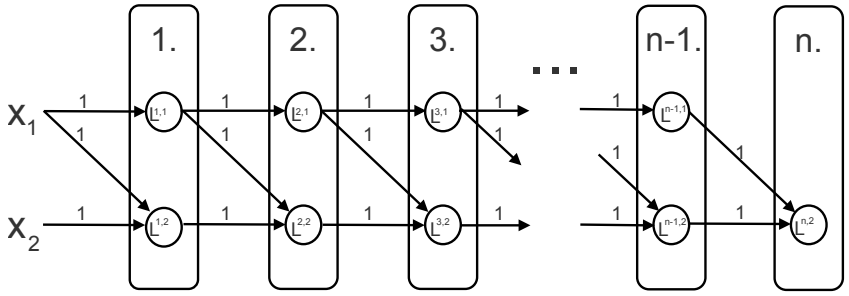

Figure 2: The structure of the example network to illustrate FPSoundSymbolic.

Accordingly, we have two types of Δ, denoted as $\Delta^{(1)}$ and $\Delta^{(2)}$. The intervals generated by these are denoted as $I_i = [1 - \Delta^{(i)}, 1 + \Delta^{(i)}]$, where $i = 1, 2$. Note that in the main text we used a common Δ for every layer (using the maximal number of inputs) for simplicity, however, one can use layer-wise Δ in this manner.

It can be seen that the extended symbolic expression for the i-th layer neurons are given by the following formulas:

$$L_\Delta^{i,1} = I_1^i x_1 + 0 x_2, \text{ where } i = 1, \ldots, n-1$$

$$L_\Delta^{i,2} = \left(\left(\ldots \left((I_2^2 + I_1 I_2) I_2 + I_1^2 I_2 \right) \ldots \right) I_2 + I_1^{i-1} I_2 \right) x_1 + I_2^i x_2 \text{ where } i = 1, \ldots, n$$

Thus, the symbolic result for the output neuron is

$$L_\Delta^{n,2} = \left[\left(\ldots \left((I_2^2 + I_1 I_2) I_2 + I_1^2 I_2 \right) \ldots \right) I_2 + I_1^{n-1} I_2 \right] x_1 + I_2^n x_2$$

Let us look at the $L_\Delta^{n,2}$ interval formula and examine whether, for $x_1 = 1$ and $x_2 = 2^{53}$, it contains 2^{53}. Since the exact coefficient of x_1 is n, we know that the interval coefficient of x_1 will include n, hence multiplying with x_1 will include nx_1. For the second part, the coefficient contains $[1 - n\mu, 1]$, because $\Delta^{(2)} > \mu$. Hence the x_2 term will include the $[x_2(1 - n\mu), x_2]$ interval. Since $\mu x_2 \geq x_1$, the final result will therefore also contain x_2.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The claims we make are supported by our theoretical as well as our empirical results.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: The limitations are discussed in Section 6.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [Yes]

Justification: Most of our contributions are theoretical. The complete set of assumptions, along with the statement of our propositions is included in Section 4, while the complete proofs are included in Appendix A. In addition, the technical details of the treatment of overflow and underflow are included in Appendix B.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: Reproducibility is facilitated by presenting all the necessary code in the supplementary material, and references to third party code in the main paper in Section 5 and in the appendix in Appendix D along with the description of the hardware and software environment we used.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in

some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: We provide open access to our implementation, with sufficient instructions.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Our experiments do not involve training and testing models, we do formal verification. We provide every necessary detail that includes the model checkpoints (taken from related work) and the verification task parameters.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: Our algorithms are deterministic formal verification algorithms that are tested on a benchmark of frozen model checkpoints and a fixed test set of inputs. Following the general practice of the area, we report the verification percentage over these test inputs, and we provide statistics (average and maximum) of the inclusion interval widths.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.

- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Our experiments report the execution times in Section 5, and the used hardware and software components are described in Appendix D.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: We confirm with the Code of Ethics: our research uses only common benchmark datasets and models and is targeted directly at improving AI safety.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [N/A]

Justification: Our work is foundational in nature. Since we work on formal verification, positive societal impacts are intended via verifying the adversarial robustness of deployed models.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

Answer: [N/A]

Justification: We do not release data or models.

Guidelines:

- The answer [N/A] means that the paper poses no such risks.
- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes]

Justification: We use only openly released model checkpoints and formal verifiers, where the license allows for academic use.

Guidelines:

- The answer [N/A] means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.

- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: We release our software implementation of the new verifiers, along with sufficient documentation, under the open source Apache 2.0 license. We mention this in Appendix D.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: We include use no crowdsourcing or research with human subjects,

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: We include use no crowdsourcing or research with human subjects,

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [N/A]

    Justification: We took advantage of the LLM feedback option, which helped to fix a number of presentation issues, but it affected the paper in no important ways.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.