Verification Neural Network

Nassim Arifette

26 March 2024

1 Introduction to Verification Neural Network

In the fast-growing field of artificial intelligence (AI), neural networks are fundamental to many modern technologies, such as self-driving cars, medical diagnostic systems, and language processing tools. As these neural networks become more complex and widespread, ensuring they are reliable, safe, and trustworthy is increasingly challenging. This is where the technique of Abstraction-Based Verification (ABV) becomes essential. ABV helps in checking if neural networks behave correctly according to set standards, thus reducing risks and improving the safety of these systems.

Abstraction-Based Verification involves methods that analyze neural network models to confirm they work within set limits and make safe, accurate decisions. The main strength of ABV is its ability to simplify the complex details of neural networks into easier, more manageable models while maintaining important functionalities. This simplification makes it easier to thoroughly check and fix any issues like potential biases or errors that could cause unexpected or harmful results.

The importance of ABV in neural network applications is critical, especially as neural networks are used more in important areas like healthcare, finance, and transportation. The risks of mistakes in these areas are high, and ABV provides a way to ensure these systems work as expected, even under new or changing conditions.

This report will cover the basics of Abstraction-Based Verification, how it works, and some advanced method using tropical algebra.

1.1 Motivation

At the heart of this method is the idea of overapproximating—a way to consider all possible states a neural network might reach with any input, but in a simplified way. The main goal here is to create a broad model that allows us to verify the network's behavior is correct.

This overapproximation has two main purposes. Firstly, it helps confirm the network's outputs are reliable and safe across different scenarios, ensuring the network behaves correctly. Secondly, it recognizes the challenges of dealing with complex systems. If the verification doesn't prove the network is correct, it doesn't mean the network is unreliable. Instead, it might indicate that our simplified model doesn't capture all the details needed. This is an important point because it prompts a careful check to see if the issue is with the network itself or just the model used to understand it.

Using this approximation approach, those working with neural networks have a robust way to manage the complexity of verifying these systems. This method balances the need for thorough analysis and the practicality of the computational efforts involved. Overapproximation not only helps in verifying the accuracy of neural networks but also helps in refining our understanding of these complex systems. Through ongoing refinement and testing, this method helps align theoretical models of how neural networks should work with the actual outcomes we see in real-world applications.

2 What we want to verify

Let $f : \mathbb{R}^n \to \mathbb{R}^m$ be a neural network with input $x \in \mathbb{R}^n$ and output $y = f(x) \in \mathbb{R}^m$.

As an example we can take a neural network that take a grey scale image and outputs a real number.

Let's take $x \in \mathbb{R}^n$ to be the input image and $y \in \mathbb{R}^m$ to be the output vector.

Let's say that we have $c \in \mathbb{R}^n$ an image that is slightly different from x in the sense that it has a small amount of noise added to it (e.g. a few pixels are changed, brightness is altered, etc.).

We can define a set $X = \{x \in \mathbb{R}^n : \|x - c\| < \epsilon\}$ that contains all images that are close to c in the sense that their L^2 distance is less than ϵ.

We take $r \in \mathbb{R}^m$ to be the output of the neural network on the input x, i.e. $r = f(x)$.

If we note $class(r)$ to be the class of the output r (e.g. the digit that the neural network predicts in the case of a digit recognition network), we want to verify that the neural network is robust to small perturbations in the input, i.e. that if $x \in X$ then $class(f(x)) = class(r)$.

The issue with this method is that even the number of images is finite, the number of images that are close to c is infinite (or at least really) and it is not feasible to check the output of the neural network on all of them.

Let's imagine that we can lift the neural network to a higher dimension and work with sets instead of points and so here work over sets of images instead of single images. So we need to define :

$$f^S : \mathcal{P}(\mathbb{R}^n) \to \mathcal{P}(\mathbb{R}^m)$$

such that $f^s(X) = \{y \mid x \in X, x = f(x)\}$.

We can now run it on the following input set:

$$X = \{x \in \mathbb{R}^n : \|x - c\| < \epsilon\}$$

By computing $f^S(X)$ we get the prediction of the neural network on all images that are close to c and we can check if the class of the output is always the same.

So formaly we can simply check that $class(f^S(X)) \subseteq \{y \mid y = class(r)\}$.

Exemple 2.1: Let's take a neural network that takes a grey scale image and outputs a real number. We want to verify that the neural network is robust to small perturbations in the input.
We define the following correctness property:

$$\begin{gathered}\{\|x - c\| \leq 0.1\} \\ r \leftarrow f(x) \\ \{class(r) = 1\}\end{gathered}$$

If we pick any image x that is close to c but is slighly brighter or darker by at most 0.1, we want to verify that the neural network predicts the class 1 for all such images.
So we can define:

$$X = \{x \in \mathbb{R}^n : \|x - c\| \leq 0.1\}$$

which is the set of all images that verify our property. By computing $f^S(X)$ we can check if the class of the output is always 1:

$$f^S(X) \subseteq \{y \mid class(y) = 1\}$$

In other words, all runs of the neural network on images x that are close to c predict the class 1.

3 Verification using Zones

3.1 Some definitions

The idea of zones is to represent the set of reachable states of a neural network by a set of inequalities. We are trying to approximate a ReLU feed-forward neural network.

Définition 3.1 (ReLU): The ReLU activation function is defined as:

$$\text{ReLU}(x) = \max(0, x)$$

Définition 3.2 (Layer): A n-neurons ReLU network layer L with m inputs is a function $\mathbb{R}^m \to \mathbb{R}^n$ defined by, a weight matrix $W \in \mathcal{M}_{n,m}(\mathbb{R})$ and a bias vector $b \in \mathbb{R}^n$ and the ReLU activation function:

$$L(x) = \mathrm{ReLU}(Wx + b)$$

Définition 3.3 (Multilayer Perceptron): A multilayer perceptron F_N is given by a list of network layers $L_0, \ldots, L_N$, where layers $L_i(i = 0, \ldots, N-1)$ are n_{i+1}-neurons layers with n_i inputs. The action of F_N on inputs is defined by composing the action of successive layers: $F_N = L_N \circ L_O$.

Définition 3.4: Let $x \in \mathbb{R}^n$. The zone domain is defined by:

$$(\bigwedge_{i \leq i,j \leq n}^{n} x_i - x_j \leq c_{i,j}) \wedge (\bigwedge_{1 \leq i \leq n} a_i \leq x_i \leq b_i)$$

where $a_i, b_i, \in \mathbb{R}$ and $c_{i,j} \in \mathbb{R} \cup \{+\infty\}$.

```
struct Zone
    numinp::Int
    numvars::Int
    DBM::Matrix{Float64}
end
```

The variables a_i, b_i represent the lower and upper bounds of the ith coordinate of x. The variable $c_{i,j}$ represents the difference between the ith and jth coordinates of x and this can be represented using the matrix $C = (c_{i_j})$ named the difference bound matrix.

In order to encode interval constraints seamlessly, we need to introduce a special variable x_0 that is always equal to 0.

Définition 3.5: The difference bound matrix C is a matrix of size $(n+1) \times (n+1)$ where n is the number of neurons in the neural network. The square matrix C is defined by the following set of points in $\mathbb{R}^{\ltimes}$:

$$\gamma(C) = \{x \in \mathbb{R}^n \mid \forall i, j \in \{0, \ldots, n\}, x_i - x_j \leq c_{i,j} \wedge x_0 = 0\}$$

Exemple 3.1: Let's say we have $-3 \leq x_1 \leq 1$, $-1 \leq x_2 \leq 3$ and $-4 \leq x_1 - x_2 \leq 0$ then the difference bound matrix is:

$$C = \begin{pmatrix} 0 & 3 & 1 \\ 1 & 0 & 0 \\ 3 & 4 & 0 \end{pmatrix}$$

A DBM might not be the "smallest" matrix what do we mean by that is for example if $x_1 - x_2 \leq 3$ and $x_2 - x_3 \leq -1$ and $x_1 - x_3 \leq 4$ then we can see that $x_1 - x_3 \leq 4$ is not the smallest matrix since we can deduce that $x_1 - x_3 \leq 2$ from the two other inequalities.

Définition 3.6: The closed matrix $C*$ is the smallest matrix that represents the same set of points as C. And can be computed by the Floy-Warshall algorithm to update $(c_{i,j})_{0 \leq i,j \leq n}$ as follows:

1. Initialize $c^*_{i,j} = c_{i,j}$ for all $i, j \in \{0, \ldots, n\}$.
2. For all $i, j, k \in \{0, \ldots, n\}$,, $c^*_{i,j} = \min(c^*_{i,j}, c^*_{i,k} + c^*_{k,j})$.

```julia
function zone_closure(input::Zone)
    c = input.DBM
    n = input.numvars + input.numinp + 1
    for k in 1:n
        for i in 1:n
            for j in 1:n
                c[i, j] = min(c[i, j], c[i, k] + c[k, j])
            end
        end
    end
    return Zone(input.numinp, input.numvars, c)
end
```

3.2 Abstraction of linear maps

We want to abstract the graph $\mathcal{G}_f = \{(x, y) \mid y = f(x)\}$ of a linear map $f(x) = Wx + b$ with $x \in [\underline{x}_1, \overline{x}_1] \times \ldots \times [\underline{x}_m, \overline{x}_m]$ and $y \in [\underline{y}_1, \overline{y}_1] \times \ldots \times [\underline{y}_n, \overline{y}_n]$, where $W \in \mathcal{M}_{n,m}(\mathbb{R})$ and $b \in \mathbb{R}^n$.

We want to implement the following proposition which correspond to Propositon 3 of SAS paper:

Proposition 3.1 (Optimal approximation of a linear layer by a zone):
Let $n, m \in \mathbb{N}$ and $f : \mathbb{R}^m \to \mathbb{R}^n$ an affine transformation defined, for all $x \in \mathbb{R}^m$ and $i \in [1, n]$, by $\big(f(x)\big)_i = \sum_{j=1}^m w_{i,j} x_j + b_i$. Let $K \subset \mathbb{R}^m$ be an hypercube defined as $K = \prod_{\{} 1 \leq j \leq m\}[\underline{x}_j, \overline{x}_j]$, with $\underline{x}_j, \overline{x}_j \in \mathbb{R}$. Then, the tightest zone $\mathcal{H}_f$ of $\mathbb{R}^m \times \mathbb{R}^n$ containing $S := \Big\{\big(x, f(x)\big) \,\Big|\, x \in K\Big\}$ is the set of all $(x, y) \in \mathbb{R}^m \times \mathbb{R}^n$ satisfying

$$\Big(\bigwedge_{1\leq j\leq m} \underline{x}_j \leq x_j \leq \overline{x}_j\Big) \wedge \Big(\bigwedge_{1\leq i\leq n} m_i \leq y_i \leq M_i\Big) \wedge \Big(\bigwedge_{1\leq i_1, i_2\leq n} y_{i_1} - y_{i_2} \leq \Delta_{i_1,i_2}\Big)$$
$$\wedge \Big(\bigwedge_{1\leq i\leq n, 1\leq j\leq m} m_i - \overline{x}_j + \delta_{i,j} \leq y_i - x_j \leq M_i - \underline{x}_j - \delta_{i,j}\Big),$$

where, for all $i, i_1, i_2 \in [1, n]$ and $j \in [1, m]$:

$$m_i = \sum_{w_{i,j}<0} w_{i,j}\overline{x}_j + \sum_{w_{i,j}>0} w_{i,j}\underline{x}_j + b_i,$$
$$M_i = \sum_{w_{i,j}<0} w_{i,j}\underline{x}_j + \sum_{w_{i,j}>0} w_{i,j}\overline{x}_j + b_i,$$
$$\Delta_{i_1,i_2} = \sum_{w_{i_1,j}<w_{i_2,j}} (w_{i_1,j} - w_{i_2,j})\underline{x}_j + \sum_{w_{i_1,j}>w_{i_2,j}} (w_{i_1,j} - w_{i_2,j})\overline{x}_j + (b_{i_1} - b_{i_2}),$$
$$\delta_{i,j} = \begin{cases} 0, & \text{if } \ w_{i,j} \leq 0 \\ w_{i,j}(\overline{x}_j - \underline{x}_j), & \text{if } \ 0 \leq w_{i,j} \leq 1 \\ (\overline{x}_j - \underline{x}_j), & \text{if } \ 1 \leq w_{i,j} \end{cases}$$

3.3 Abstraction of activation function ReLU

To interpret the ReLU activation function within the context of zones represented by difference bound matrices, we need to understand how it works on interval constraints.

For a given interval constraint $x \in [l, u]$, the ReLU function can be interpreted as follows:

$$\text{ReLU}([l, u]) = \begin{cases} [l, u] & \text{if } l \geq 0 \\ [0, 0] & \text{if } u \leq 0 \\ [0, u] & \text{if } l < 0 < u \end{cases}$$

```
function zone_approximate_act_map(act::ActivationFunction, input::
    Zone)
    if act == Id()
        return input
    elseif act == ReLU()
```

```julia
        db = copy(input.DBM)
        n = input.numvars + input.numinp + 1
        for i in (input.numinp+2):(input.numinp+input.numvars+1)
            db[1, i] = max(db[1, i], 0)
            if db[i, 1] < 0
                db[i, 1] = 0
            end
            for j in 1:n
                if db[i, j] < 0
                    db[i, j] = 0
                end
                if db[j, i] < 0
                    db[j, i] = 0
                end
            end
        end
        return Zone(input.numinp, input.numvars, db)
    end
    error("Activation function not supported")
end
```

4 Verification using Tropical Geometry

What's the idea behind tropical geometry?

The idea is to replace the usual operations of addition and multiplication by the tropical operations of taking the minimum and the sum.

Définition 4.1: Let's define $\mathbb{R}_{\max} = \mathbb{R} \cup \{-\infty\}$.
Let $a, b \in \mathbb{R}_{\max}$, we define the tropical sum and product by:

$$a \oplus b = \max(a, b)$$
$$a \otimes b = a + b$$

The neutral element for the tropical sum is $-\infty$ and the neutral element for the tropical product is 0.
No inverse for the sum but inverse on $\mathbb{R}$ (and not on $\mathbb{R}_{\max}$) for the product.
The usual order on $\mathbb{R}$ is the same as the order on $\mathbb{R}_{\max}$ with $-\infty$ being the smallest element. $x \leq y$ if and only if $x \oplus y = \max(x, y) = y$.

Remarque 4.1: Some definition of tropical geometry use the minimum instead of the maximum for the tropical sum. This is equivalent since the maximum is the minimum in the tropical semiring.

4.1 Tropical Polynomials

Just as classical polynomials are sums of monomials, tropical polynomials are tropical sums of tropical monomials.

Définition 4.2 (Tropical Monomial): A tropical monomial in variables $x_1, \dots, x_n$ is an expression of the form

$$c \otimes x_1^{\otimes \alpha_1} \otimes \cdots \otimes x_n^{\otimes \alpha_n}$$

which in classical notation is

$$c + \alpha_1 x_1 + \cdots + \alpha_n x_n$$

where $c, \alpha_1, \dots, \alpha_n \in \mathbb{R}$.

Définition 4.3 (Tropical Polynomial): A tropical polynomial is a tropical sum of tropical monomials:

$$p(x) = \bigoplus_{i=1}^{m} (c_i \otimes x_1^{\otimes \alpha_{i1}} \otimes \cdots \otimes x_n^{\otimes \alpha_{in}})$$

which in classical notation is

$$p(x) = \max_{i=1}^{m} (c_i + \alpha_{i1} x_1 + \cdots + \alpha_{in} x_n)$$

4.2 Tropical Rational Functions

Tropical rational functions are particularly important for understanding the behavior of ReLU networks.

Définition 4.4 (Tropical Rational Function): A tropical rational function is a difference of two tropical polynomials:

$$r(x) = p(x) \oslash q(x) = p(x) - q(x)$$

where p and q are tropical polynomials.

4.3 Relation to ReLU Neural Networks

ReLU neural networks can be represented as tropical rational functions, which provides a powerful framework for verification. Let's explore this connection in detail.

Proposition 4.1: The ReLU activation function can be expressed in tropical notation as:

$$\mathrm{ReLU}(x) = \max(0, x) = 0 \oplus x$$

Theorème 4.1: Any function computed by a ReLU neural network can be expressed as a tropical rational function of the form:

$$f(x) = \max_j (\alpha_j^0 + \sum_i \alpha_j^i x_i) - \max_k (\beta_k^0 + \sum_l \beta_k^l x_l)$$

Proof. The proof follows by induction on the number of layers. For a single layer with ReLU activation, we can directly express the output as a tropical polynomial. For multiple layers, we compose these functions and use the properties of tropical algebra to simplify the resulting expression. □

4.4 Tropical Hypersurfaces and Neural Network Decision Boundaries

A key insight from tropical geometry is that the decision boundaries of ReLU neural networks correspond to tropical hypersurfaces.

Définition 4.5 (Tropical Hypersurface): The tropical hypersurface of a tropical polynomial p is the set of points where the maximum in p is achieved by at least two different terms.

Proposition 4.2: The decision boundaries of a ReLU neural network correspond to the tropical hypersurfaces of its tropical polynomial representation.

This correspondence allows us to analyze the geometric properties of neural network decision boundaries using tools from tropical geometry.

4.5 Verification Algorithm using Tropical Geometry

Now, we present an algorithm for verifying properties of ReLU neural networks using tropical geometric methods.

Neural Network Verification via Tropical Geometry

1. Express the neural network as a tropical rational function f.
2. Express the property to be verified as a constraint g on the output of f.
3. Formulate the verification problem as checking whether f satisfies g for all inputs in a given domain D.
4. Use tropical geometric operations to compute the regions where f may violate g.
5. If no such regions exist in D, the property is verified.

4.6 Implementation in Julia

Let's implement the basics of tropical algebra and verification in Julia:

```julia
module TropicalAlgebra

export TropicalNumber,  ,  , tropical_sum, tropical_product

# Define a tropical number type
struct TropicalNumber
    value::Float64

    # Constructor with special handling for -Inf
    TropicalNumber(x::Real) = x == -Inf ? new(-Inf) : new(convert(Float64, x))
```

```julia
end

# Define tropical addition (max)
tropical_sum(a::TropicalNumber, b::TropicalNumber) = TropicalNumber(max(a.value, b.value))
   (a::TropicalNumber, b::TropicalNumber) = tropical_sum(a, b)

# Define tropical multiplication (addition)
tropical_product(a::TropicalNumber, b::TropicalNumber) = TropicalNumber(a.value + b.value)
   (a::TropicalNumber, b::TropicalNumber) = tropical_product(a, b)

# Define pretty printing
Base.show(io::IO, x::TropicalNumber) = print(io, x.value == -Inf ? "-   " : x.value)

end # module

module TropicalVerification

using LinearAlgebra
using ..TropicalAlgebra

export tropical_affine_transform, tropical_relu, tropical_network_forward

# Implement tropical affine transformation
function tropical_affine_transform(W::Matrix{Float64}, b::Vector{Float64}, x::Vector{TropicalNumber})
    n = size(W, 1)
    result = Vector{TropicalNumber}(undef, n)

    for i in 1:n
        result[i] = TropicalNumber(b[i])
        for j in 1:length(x)
            term = W[i,j]     x[j]
            result[i] = result[i]     term
        end
    end

    return result
end

# Implement tropical ReLU
function tropical_relu(x::TropicalNumber)
    zero = TropicalNumber(0)
    return x     zero
end

function tropical_relu(x::Vector{TropicalNumber})
    return [tropical_relu(xi) for xi in x]
end

# Forward pass through a tropical neural network
function tropical_network_forward(weights::Vector{Matrix{Float64}},
                                  biases::Vector{Vector{Float64}},
                                  input::Vector{TropicalNumber})
    x = input
    for i in 1:length(weights)
        x = tropical_affine_transform(weights[i], biases[i], x)
        if i < length(weights)  # Apply ReLU to all but the last layer
            x = tropical_relu(x)
        end
    end
    return x
end

# Verify a property of a neural network
function verify_property(weights::Vector{Matrix{Float64}},
                         biases::Vector{Vector{Float64}},
                         input_domain::Matrix{Float64},
                         property::Function)
    # TODO: Implement verification using tropical geometry
    # This would check if the network satisfies the property
    # over the entire input domain

    # Placeholder for the actual implementation
    return true
end

end # module

# Example usage
using .TropicalAlgebra
using .TropicalVerification

# Define a simple neural network
weights = [
    [1.0 -1.0; 1.0 1.0],  # First layer weights
    [1.0 1.0; -1.0 1.0]   # Second layer weights
]

biases = [
    [-1.0, 1.0],  # First layer biases
    [0.0, 0.0]    # Second layer biases
]

# Convert input to tropical numbers
input = [TropicalNumber(0.5), TropicalNumber(-0.5)]

# Forward pass
output = tropical_network_forward(weights, biases, input)
println("Network output: ", output)

# Define an input domain and a property to verify
input_domain = [-1.0 1.0; -1.0 1.0]  # Bounds for each input dimension
property(output) = output[1].value ≥ output[2].value  # Class 1 always
    has higher score than class 2

# Verify the property
verified = verify_property(weights, biases, input_domain, property)
println("Property verified: ", verified)
```

4.7 Tropical Polyhedra for Verification

Tropical polyhedra provide a powerful abstraction for representing the behavior of ReLU networks.

Définition 4.6 (Tropical Polyhedron): A tropical polyhedron is a set of points satisfying a finite number of tropical linear inequalities:

$$\bigoplus_{j=1}^{n} a_j \otimes x_j \oplus c \leq \bigoplus_{j=1}^{n} b_j \otimes x_j \oplus d$$

To verify properties of a neural network, we can represent its behavior using tropical polyhedra and then check if the property holds for all points in the polyhedra.

```julia
struct TropicalPolyhedron
    A::Matrix{Float64}  # Left-hand side coefficients
    b::Vector{Float64}  # Left-hand side constants
    C::Matrix{Float64}  # Right-hand side coefficients
    d::Vector{Float64}  # Right-hand side constants
end

function is_in_polyhedron(poly::TropicalPolyhedron, x::Vector{
    Float64})
    m = size(poly.A, 1)
    n = size(poly.A, 2)

    for i in 1:m
        lhs_max = poly.b[i]
        for j in 1:n
            lhs_max = max(lhs_max, poly.A[i,j] + x[j])
        end

        rhs_max = poly.d[i]
        for j in 1:n
            rhs_max = max(rhs_max, poly.C[i,j] + x[j])
        end

        if lhs_max > rhs_max
            return false
        end
    end

    return true
end

function project_polyhedron(poly::TropicalPolyhedron, dims::Vector{
    Int})
    # Placeholder for projection algorithm
    # In practice, this would compute a new polyhedron that is the
    # projection of the original one onto the specified dimensions
    return poly
end

```

```
function intersect_polyhedra(poly1::TropicalPolyhedron, poly2::
    TropicalPolyhedron)
    # Placeholder for intersection algorithm
    # Would combine the constraints from both polyhedra
    return poly1
end
```