Literature Implementation in Haskell: Verification of Neural Networks

Prithvi Padathara

April 2025

Contents

1 Introduction

A neural network is a computational model consisting of layers of interconnected nodes that process and learn from data. It uses weights, biases, and activation functions to transform input data into outputs, enabling tasks like classification, regression, and pattern recognition.

Due to their ability to automate decision making and recognize patterns in big data, they are used in a wide range of applications, from natural language processing to image recognition.

However, networks do not always classify or make predictions correctly. There are multiple reasons why they could fail, but one reason that will be looked at in this paper is misclassification due to adversarial attacks.

The objective of this paper is to look at the following literature on abstraction based verification of neural networks and implement the algorithms presented in a purely functional approach using Haskell.

- Introduction to Neural Network Verification by Aws Albarghouthi
- AI^2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation by Gehr et al.
- Fast and Effective Robustness Certification by Singh et al.

2 Motivation

In applications such as autonomous driving, medical diagnosis, or financial systems, ensuring the safety and reliability of the model is paramount. Verification helps to confirm that the model behaves correctly under all conditions, reducing the risk of failures that could lead to accidents or incorrect decisions.

One cause of incorrect behavior of networks is adversarial attacks. Adversarial attacks are techniques used to manipulate machine learning models by introducing small, carefully crafted changes to input data that cause the model to make incorrect predictions or classifications. These perturbations are often imperceptible to humans, but can significantly affect the model's performance.

2.1 Adversarial Attacks

There are multiple methods of perturbing input data, few such methods being by changing the pixel intensity values of the input and rotation of the input image till it is misclassified. In this paper, we will be looking at attacks involving the lightening/darkening of pixel intensities.

For example, as a part of this project I used Fast Gradient Sign Method (FGSM) to create adversarial images for this image of a great white shark. The network I was testing was the MobileNetV2 pretrained model. Feeding this image to the network, it correctly predicted it as a great white shark with 98% confidence.

Figure 1: Great White Shark [6]

To perform the FGSM attack, a perturbation image is created first. For the above image, it looked like this:

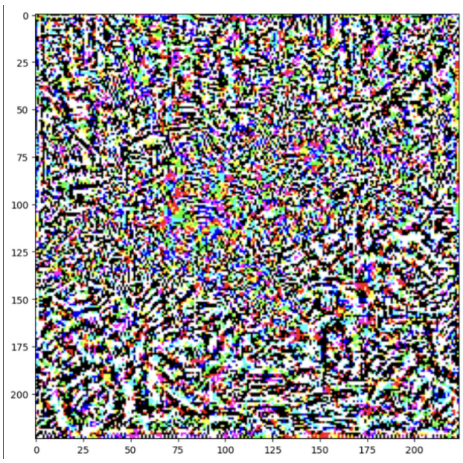
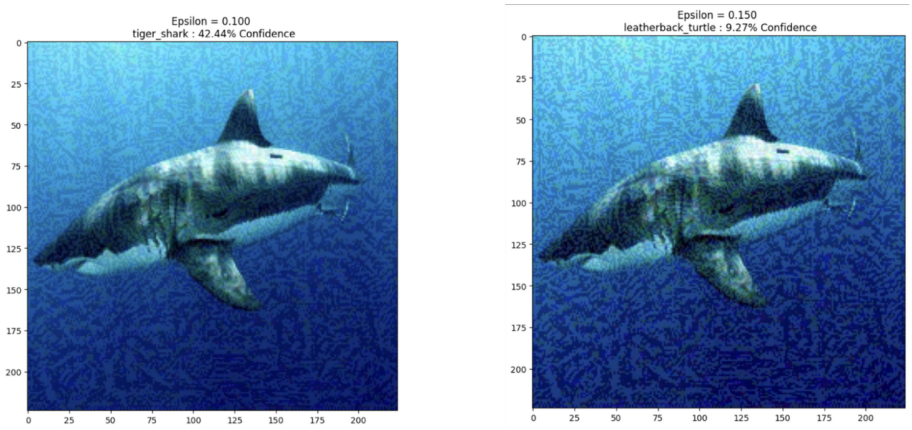


Figure 2: Perturbation Image

This perturbation image is multiplied by some constant ϵ and superimposed on the original image. I used constants of 0.1 and 0.15 for testing and the model misclassified it as a tiger shark and a turtle.



(a) $\epsilon = 0.1$ (b) $\epsilon = 0.15$

Figure 3: Original and Perturbation Images used in FGSM Attack

To verify if a neural network is safe from adversarial attacks, one can conduct adversarial testing by generating adversarial examples and evaluating the model's performance on them. Techniques like robustness testing involve creating a variety of adversarial inputs using methods like FGSM and checking if the model consistently misclassifies or remains robust.

However, the issue with this verification method is that the number of images generated and used for prediction becomes too large and computationally infeasible. A more efficient way of testing these large sets of images is by using formal verification methods which is what will be covered in this paper.

3 Background: Neural Networks

Before getting into formal verification, a mathematical background on neural networks is required. Networks are constructed by composing multiple layers together, and different types of layers have different functionalities.

The layers that are looked at in this work are:

- Dense or fully connected
- Activation functions
- Convolution
- Pooling

3.1 Dense Layer

The most basic transformation in all feedforward networks is the fully connected, dense or affine layer. These layers are termed "fully connected" because each

neuron in one layer is connected to every neuron in the preceding layer, creating a highly interconnected network.

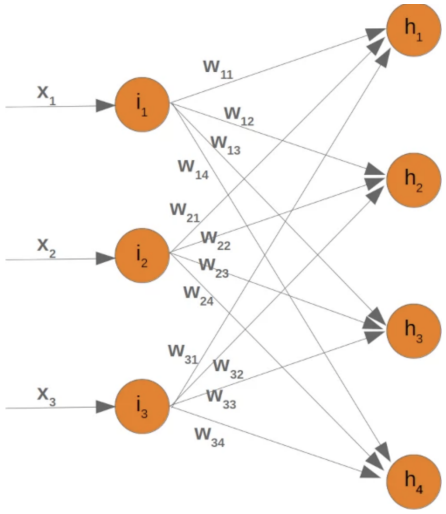


Figure 4: Dense Layer [7]

Each neuron in an FC layer receives inputs from all neurons of the previous layer, with each connection having a specific weight and each neuron incorporating a bias. The input to each neuron is a weighted sum of these inputs plus a bias:

$$y_j = \sum_{i=1}^{n} W_{ji} x_i + b_j$$

3.2 Activation Functions

> *Activation functions play a very crucial role in neural networks by learning the abstract features through non-linear transformations. Some common properties of activation functions are as follows: a) it should add the non-linear curvature in the optimization landscape to improve the training convergence of the network; b) it should not increase the computational complexity of the model extensively; c) it should not hamper the gradient flow during training; d) it should retain the distribution of data to facilitate the better training of the network. Several activation functions have been explored in recent years for deep learning to achieve the above mentioned properties. (https://arxiv.org/pdf/2109.14545).*

Some examples of activation functions are ReLU, sigmoid and tanh. In this paper, we will only be looking at ReLU.

The ReLU function takes in a value and outputs 0 if the value is negative, if not, it returns the value as is.

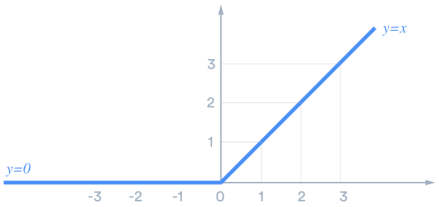


Figure 5: ReLU [8]

3.3 Convolution Layer

In 2D image convolution, each pixel in the output image is calculated by applying a filter (also called a kernel) to a small region of the input image. Mathematically, if I is the input image and K is the kernel, the convolution at position (i, j) in the output image O is:

$$O(i,j) = \sum_{m=-k}^{k} \sum_{n=-k}^{k} I(i+m, j+n) \cdot K(m,k)$$

In simpler terms, convolution is a process where you take an image and a smaller filter (a grid of numbers) and slide the filter over the image, one small step at a time. At each position, you multiply the numbers in the filter by the numbers in the image that it covers, then add them together to create a new value. This new value becomes part of a new image that highlights certain features, like edges or patterns. The filter helps the process focus on specific details in the image, and by repeating this across the entire image, convolution transforms the original into a new version that emphasizes important aspects.

3.4 Pooling Layer

A pooling layer in a neural network is used to reduce the spatial dimensions (height and width) of the input while retaining important features. It helps to lower the computational load, reduce memory usage, and prevent overfitting by summarizing the features in a local region of the input.

The most common types of pooling are:

- Max Pooling: Selects the maximum value from a small region (e.g., 2x2 or 3x3) in the input. This helps preserve the most prominent features.
- Average Pooling: Computes the average value from a small region in the input. It retains general feature information rather than focusing on extreme values.

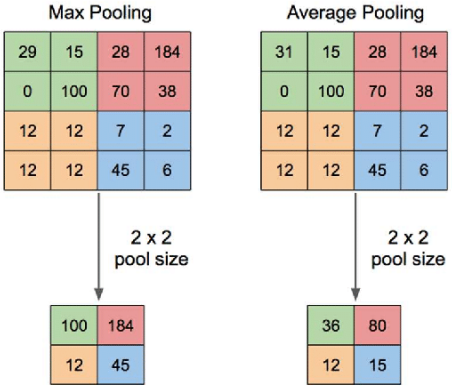


Figure 6: Pooling Example [9]

4 Abstract Interpretation

Abstract interpretation is a type of formal verification method that works on a set of images rather than individual images.

For some image vector c and a noise term $\epsilon > 0$, a set of images X can be defined as:

$$X = \{a \quad | \quad ||a - c|| < \epsilon\}$$

Now, to verify if X belongs to the right class, we need to apply the network's weights onto the set.

Let f be the function used to represent a network. Another function f^A is created that represents the abstract transformer of the network. Now, if $f(c) =$ class a then one needs to check if $f^A(X) \in$ class a.

This raises an important question. How can one represent the domain set in a way that makes it easy to manipulate data?

4.1 Zonotopes

Zonotopes provide a model that makes it easy to perform the operations of a neural network on a set of images, instead of having to do it on each individual image in the set. A zonotope is an n-dimensional **convex** polytope that can be described as the Minkowski sum of a finite set of line segments (vectors). In simpler terms, it's a geometric object formed by combining line segments in various directions.

Convexity is an important feature of zonotopes as it allows for representation of the set using only boundary points. Another important point to note is that all functions that are applied on the zonotope need to maintain convexity.

Every zonotope has a center point it starts from. From this center point it deviates to produce a range (or set) of values. The amount of deviation

is determined by error terms that are represented using ϵ_i where $\epsilon \in [-1, 1]$. Mathematically, it can be represented as follows:

$$\left\{ \left(\underbrace{c_{10} + \sum_{i=1}^{m} c_{1i} \cdot \epsilon_i}_{\text{first dimension}} \quad , \quad \underbrace{c_{n0} + \sum_{i=1}^{m} c_{ni} \cdot \epsilon_i}_{\text{nth dimension}} \right) \middle| \epsilon_i \in [-1, 1] \right\} c_{ij} \in \mathbb{R}$$

where m is the number of error terms and c_{ni} are the coefficients of the error terms.

Before getting into n-dimensional cases, starting with a 1 dimensional case makes things simpler to understand. Let's start off with no error terms, with the zonotope centered at '1'. That would look like this:

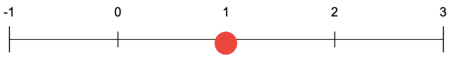


Figure 7: 1-D Zonotope: $\{1\}$

Now, let's introduce 2 error terms. Let the zonotope be 1 + 0.6 ϵ_1 + 0.4 ϵ_2. To get the possible range of values represented by this equation, one can substitute the boundary values of the ϵ terms to get the boundary values of the zonotope. That is depicted in the figure below.

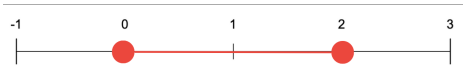
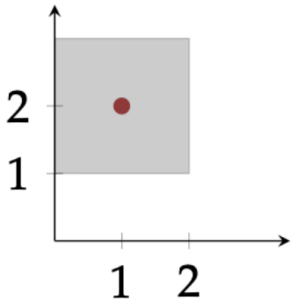
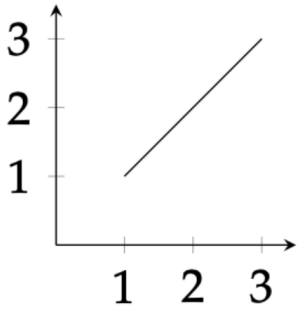


Figure 8: 1-D Zonotope: $\{1 + 0.6\epsilon_1 + 0.4\epsilon_2\}$

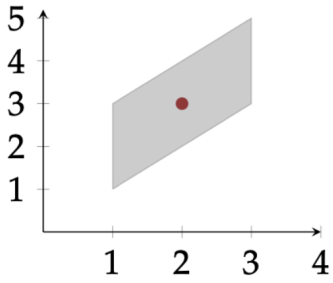
Zonotopes get more interesting in higher dimensions. Here are some 2 dimensional examples.



(a) $1 + 1\epsilon_1, 2 + 1\epsilon_2$



(b) $2 + 1\epsilon_1, 2 + 1\epsilon_1$



(c) $2 + 1\epsilon_1, 3 + 1\epsilon_1 + 1\epsilon_2$

Figure 9: 2-D Zonotopes [1]

To make these zonotopes even easier to work with, one can convert them from equation form to matrix form. In the matrix form, each row represents 1 dimension of the zonotope. The first column represents the center, and each subsequent column represents an error term.

For example, $2 + 1\epsilon_1, 3 + 1\epsilon_1 + 1\epsilon_2$ can be represented as:

$$\begin{bmatrix} 2 & 1 & 0 \\ 3 & 1 & 1 \end{bmatrix}$$

4.2 Generating a Zonotope from an Image

To understand zonotopes with regards to images better, let's look at a small example. Imagine a 1x2 image. Let its pixel values be (0.2,0.5). If this image is thought of as a point in space (as a vector), it could be plotted on a 2-D plane as (0.2,0.5). This image is unique and no other image could occupy the same space as this one.

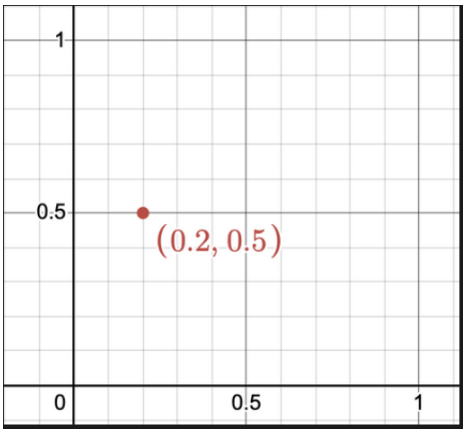


Figure 10: Image [0.2,0.5] as a point in space

Now suppose this image is perturbed slightly, each pixel by +0.1. The new image would be (0.3,0.6), which can be represented as a vector like before. The 2 points together would look like:

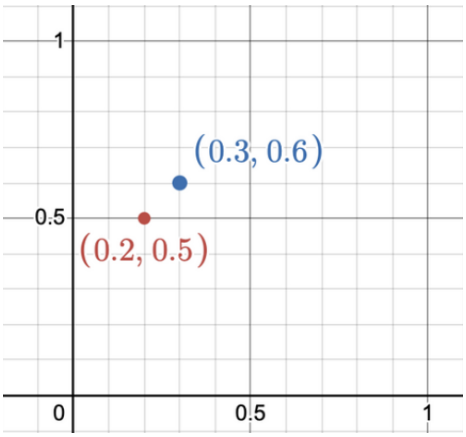


Figure 11: Image [0.3,0.6] in 2-D space

Now suppose each pixel in the image is perturbed up to 0.1 in either direction (plus and minus). By 'up to', it means that the image could be perturbed by 0.001, 0.01 etc. up to 0.1. Now this set of perturbed images can be represented as this yellow box (inclusive of all points inside the yellow borders) with the original image as the center.

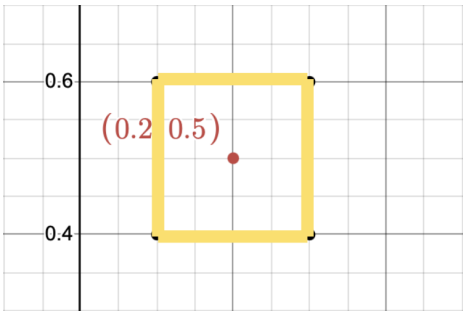


Figure 12: Set of images around [0.2,0.5]

This box represents 1000s of unique images (each minutely different than the other). During an adversarial attack, the attacker will perturb the original image slightly and the adversarial image is likely to belong to this box (if it doesn't, one can increase the size of the box by increasing the perturbation amount). To test the network traditionally, each image in this set would have to be sent to the model and checked if the predicted class matches the original image's class. As the size of the set increases this becomes more and more infeasible.

The solution is the zonotope model. Using zonotopes, the neural network's weights can be used to calculate how the model would transform the entire set of images to predict a class. In the example above, the box representing the images can be represented as a zonotope using the equation:

$$(0.2 + 0.1\epsilon_1, 0.5 + 0.1\epsilon_2)$$

Now that there's a neat model that can be used, another question needs to be answered; how does one find the appropriate representative transformations corresponding to layers of a neural network?

4.3 Neural Networks as a Composition of Functions

An important point to keep in mind is that a neural network can be represented as a composition of functions. As long as each function preserves convexity, they can be applied on the zonotope. However, there are cases in which they don't preserve convexity. In these cases, these functions need to be converted to some affine transformation, or provide some linear approximation of what the function would output.

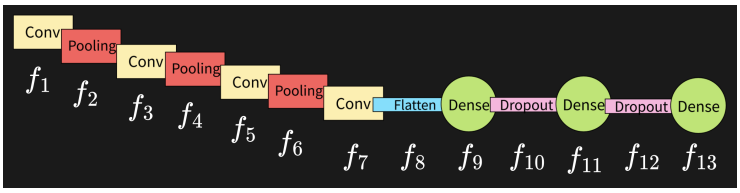


Figure 13: Neural Network as a Composition of Functions

Note from before that a zonotope can be written in matrix form. This matrix form can be represented as a list of column vectors. Each term in the formula below represents a vector.

$$\hat{c} + a_1\hat{\epsilon}_1 + ... + a_n\hat{\epsilon}_n$$

Now suppose a layer in the network is represented by T, and T is an affine transformation. Applying T on a zonotope,

$$T(\hat{c} + a_1\hat{\epsilon}_1 + ... + a_n\hat{\epsilon}_n) = T(\hat{c}) + \epsilon_1 T(\hat{a_1}) + ... + \epsilon_n T(\hat{a_n})$$

Therefore, to apply transformations onto a zonotope, one simply needs to apply the transformation onto each column vector from the matrix form of the zonotope.

4.4 Dense Layers

Applying a dense layer on a zonotope is very straightforward as these layers are already purely affine. As mentioned previously, a dense layer can be written mathematically as:

$$y_j = \sum_{i=1}^{n} W_{ji}x_i + b_j$$

This can be simplified to:

$$\hat{y} = T(\hat{x}) + \hat{b}$$

To apply it on a zonotope, one can apply it on each column vector as mentioned before.

$$y_{zonotope} = (T(\hat{x}) + \hat{b},\ \epsilon_1 T(a_1) + \hat{b},\ \ ...\ ,\ \epsilon_n T(a_n) + \hat{b})$$

4.5 Activation Functions: ReLU

The ReLU function takes in a value and outputs 0 if the value is negative, if not, it returns the value as is.

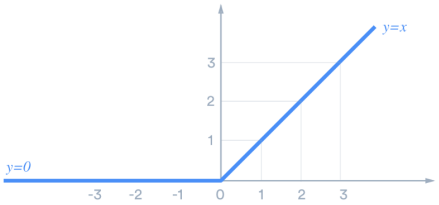


Figure 14: ReLU [8]

Creating an abstract transformer is not as straightforward for ReLU as it is not a linear function, only piecewise linear.

In a ReLU layer, ReLU is always performed on each node individually. This means the input to each ReLU function is a 1 dimensional zonotope. A 1D zonotope can only be a line, so checking the ReLU constraint becomes easier.

If the entire line segment is strictly greater than 0 i.e. the lower bound of the 1-D zonotope is greater than 0, it can be left as is after applying ReLU. If it is less than 0 i.e. the upper bound is less than 0, it can be set to 0. This is because ReLU is piecewise linear, and in each of these pieces it is linear.

The issue is when the line segment represented by the zonotope passes through 0, i.e. the lower bound of the zonotope is less than 0 and the upper bound is greater than 0. To capture it correctly, one needs to ensure that the transformation is purely linear while also trying to minimize the space covered by the zonotope. There are many ways to do this. The method used in this paper is by shearing the zonotope to fit tightly against the shape of ReLU.

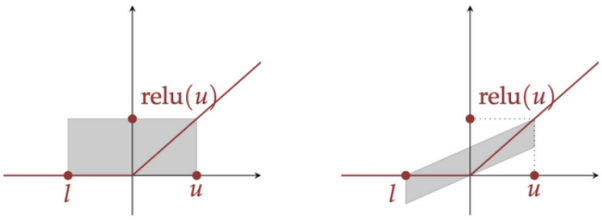


Figure 15: Over-approximation of ReLU [1]

In Figure 15, the box on the left is easier to construct, but the problem with it is that it approximates much more than the shape on the right. Both shapes guarantee soundness though, as they both include all possible ReLU outputs.

To construct the parallelogram on the right in Figure 15, the first thing to do is determine the equations for the top and bottom faces of the shape. This can be done using some simple algebra, which results in this:

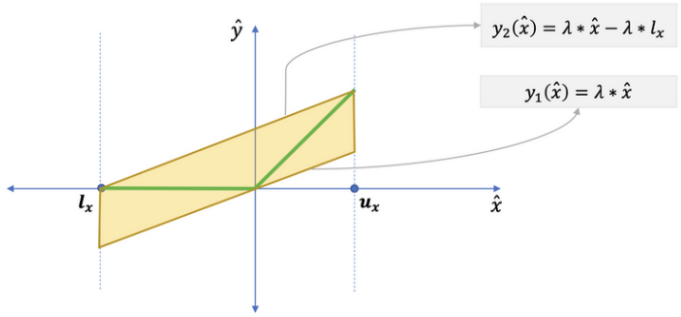


Figure 16: Creating an Abstract Transformer for ReLU [4]

All points between the two lines can be used to represent the output space of the over-approximation. This can be represented using the inequality:

$$\lambda * x \leq y(x) \leq \lambda * x - \lambda * l_x$$

However, this is not in zonotope format. The first thing to do would be to get it into equation format from an inequality.

$$y(x) = \lambda * x - c * \lambda * l_x \quad \text{where} \quad c \in [0, 1]$$

The problem with this is that all the error terms used so far were in the range [-1,1], while $c \in [0, 1]$. To fix this, a new error term can be introduced that is mathematically manipulated to be in the range [0,1].

$$c = (\epsilon_{new} + 1)/2$$

Substituting this in the previous equation, the new equation is in zonotope format:

$$y(x) = \lambda * x - (\epsilon_{new} + 1)/2 * \lambda * l_x$$

So for an input to ReLU like this:

$$x = \left\{ c_0 + \sum_{i=1}^{m} c_i \cdot \epsilon_i \;\middle|\; \epsilon_i \in [-1, 1] \right\}$$

The output would be an equation like this:

$$y(x) = \left\{ c_0 + \sum_{i=1}^{m+1} c_i \cdot \epsilon_i \;\middle|\; \epsilon_i \in [-1, 1] \right\}$$

4.6 Convolution

Convolution can be represented as matrix multiplication as this makes it an affine operation with a well defined algebraic structure.

To do so, a weight matrix W^F is defined, whose entries are taken from the weights of the filter. Suppose the input is an image x, the filter is f and the output after convolution is y. The input image is flattened out into a single row vector z. Every row of W^F will be multiplied with z to produce one value in the output y. Each row of W^F will have the filter weights positioned according to which elements of z they need to be multiplied with.

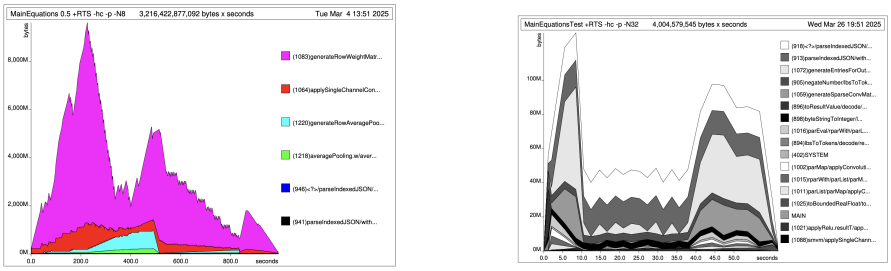
For example, let x be 5x5 and f be 3x3. Then the window that is multiplied first will be the points highlighted in yellow:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{bmatrix}$$

This window will produce the first value of y. Therefore row 1 in W^F will take the weights from the filter and place it at positions 1,2,3,6,7,8,11,12,13. The remaining 16 positions will be occupied by zeroes. Similarly, this should be done for all values of y. The total number of rows will be the total number of elements in y.

An issue that arose from this was that generation of the W^F matrix took too much time and memory because the size of W^F would grow exponentially. For a 10x10 image with a 3x3 filter, the size of W^F would be 64x100. However, the actual data stored in the matrix only takes 900 spaces out of 6400. The remaining values are all 0s.

To make better use of computational resources, we use sparse matrices in this paper. The matrix is now generated as a list of tuples. Each tuple is of the type (row index, column index, value). A tuple is only generated for points in the matrix where the value is not equal to 0. This greatly saved computational memory as seen in the heap profile figures below.



(a) Full Matrix (b) Sparse Matrix

Figure 17: Heap Profile during Runtime

Now, once the weight matrix for convolution is generated, it can be used for multiplication with the zonotope. Similar to the dense layer, the weights and bias are applied to the center of the zonotope, and just the weights are applied to the generators.

Below is an example for what generation of W^F would look like:

Let the original image be:

$$\begin{bmatrix} 12 & 2 & 34 \\ 5 & 10 & 8 \\ 9 & 12 & 1 \end{bmatrix}$$

Let the filter be:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

Then, W^F will be:

$$\begin{bmatrix} a & b & 0 & c & d & 0 & 0 & 0 & 0 \\ 0 & a & b & 0 & c & d & 0 & 0 & 0 \\ 0 & 0 & 0 & a & b & 0 & c & d & 0 \\ 0 & 0 & 0 & 0 & a & b & 0 & c & d \end{bmatrix}$$

4.7 Average Pooling

Tradtionally, most networks use max pooling instead of average pooling. However, max is a non-linear function that cannot be linearised. A solution for this was the use of average pooling instead.

Average pooling is a technique used to reduce the size of the data while retaining important features. It works by dividing the input into small regions (usually 2x2 or 3x3 blocks) and taking the average of the values from each region. This helps to make the model more efficient and less sensitive to small changes in the input.

Similar to convolution, the standard average pooling can't be applied to zonotopes, and the transformation needs to be linearised.

Average pooling is applied to each point in the zonotope (each column) individually and then combined. A single point is first taken along with the pooling dimensions and the original image dimensions. For example, let the pooling dimensions be 2x2, the original image dimensions be 5x5, then the zonotope will be represented in 25 dimensions. One column is taken from the zonotope (25x1) z, and this is reshaped into the original image dimensions - $zMatrix$. This is done to determine the sub-groups for max-pooling. The matrix below shows what the pooling groups for the above example would be.

$$\begin{bmatrix} \mathbf{1} & \mathbf{2} & \mathbf{3} & \mathbf{4} & \mathbf{5} \\ \mathbf{6} & \mathbf{7} & \mathbf{8} & \mathbf{9} & \mathbf{10} \\ \mathbf{11} & \mathbf{12} & \mathbf{13} & \mathbf{14} & \mathbf{15} \\ \mathbf{16} & \mathbf{17} & \mathbf{18} & \mathbf{19} & \mathbf{20} \\ \mathbf{21} & \mathbf{22} & \mathbf{23} & \mathbf{24} & \mathbf{25} \end{bmatrix}$$

Now once the indices that need to be pooled together are determined, the next thing to determine is how much to scale each element by. This is determined by the pooling size. For example, if it is 2x2, each pooling group will have 4 elements, so they can all be scaled by 0.25 and summed to find the average.

Below is an example of what the matrices used would look like for a 4x4 input with 2x2 pooling.

$$\begin{bmatrix} 0.25 & 0.25 & 0 & 0 & 0.25 & 0.25 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.25 & 0.25 & 0 & 0 & 0.25 & 0.25 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.25 & 0.25 & 0 & 0 & 0.25 & 0.25 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.25 & 0.25 & 0 & 0 & 0.25 & 0.25 \end{bmatrix}$$

Note, there are 16 columns in this weight matrix as there are 16 (4x4) dimensions for the input zonotope. There are 4 rows because the result of 2x2 pooling on the input will be a 4-D zonotope.

5 Implementation

The implementation of the algorithms presented can be found in this repository: `https://github.com/prithvipaddy/Neural-Network-Verification`

5.1 Technology Stack

For the implementation of the algorithms discussed earlier, the language used was Haskell. I chose Haskell because its strong static type system catches a host of potential errors at compile time, ensuring that tensor shapes, weight dimensions, and function interfaces all line up before the program ever runs.

Haskell also features immutable variables. Immutable variables eliminated side-effects in algorithms, which simplifies reasoning about how each zonotope transformation propagates through the network and also makes the code inherently thread-safe. This means that a piece of code can be executed by multiple threads at the same time safely. In an imperative language, threads often share and mutate the same memory, so one needs locks or other synchronization primitives to prevent one thread from, say, updating a variable while another is reading it. In Haskell, however, values are immutable by default—once a data structure is created, no thread can change it—so there's no shared mutable state to contend over.

Beyond these technical benefits, I was curious to test the feasibility of coding these algorithms in a purely functional language. The functional programming approach, with its focus on clear, predictable functions and building complex systems from simple parts, made it a nice challenge to implement zonotope arithmetic. Additionally, it gave me confidence that once I proved the types and functions were correct, the system would work reliably and predictably with any input.

Apart from the code for verification, it was also necessary to create and train neural networks to test the code on. For this, I used TensorFlow in python as it provides high-level APIs for ease of use, and advanced features that enable building complex deep learning systems efficiently.

5.2 Code Explanation

For the code to run, first the weights of a network need to be extracted. Currently, the code only supports networks built using TensorFlow in python. Once the weights are extracted, they need to be fed into the Haskell program, along with a base image to test around, and a perturbation amount to test within. Once these are inputted to the program, it will parse through each layer, apply the corresponding functions and finally reach the output layer. The figure below depicts how the zonotope's dimensionality changes through an example network.

```
parsed layer: conv2d
img dimensions: (43,43)
zonotope dimensions: (3,1849,1850)
parsed layer: average_pooling2d
img dimensions: (41,41)
zonotope dimensions: (32,1681,1851)
parsed layer: conv2d_1
img dimensions: (20,20)
zonotope dimensions: (32,400,1851)
parsed layer: average_pooling2d_1
img dimensions: (18,18)
zonotope dimensions: (64,324,1852)
parsed layer: conv2d_2
img dimensions: (9,9)
zonotope dimensions: (64,81,1852)
parsed layer: average_pooling2d_2
img dimensions: (7,7)
zonotope dimensions: (128,49,1853)
parsed layer: conv2d_3
img dimensions: (3,3)
zonotope dimensions: (128,9,1853)
parsed layer: flatten
img dimensions: (1,1)
zonotope dimensions: (256,1,1854)
parsed layer: dense
img dimensions: (1,1)
zonotope dimensions: (1,256,1854)
parsed layer: dropout
img dimensions: (1,1)
zonotope dimensions: (1,256,1855)
parsed layer: dense_1
img dimensions: (1,1)
zonotope dimensions: (1,256,1855)
parsed layer: dropout_1
img dimensions: (1,1)
zonotope dimensions: (1,128,1856)
parsed layer: dense_2
img dimensions: (1,1)
zonotope dimensions: (1,128,1856)
Final zonotope dimensions: (1,43,1856)
Final zonotope classification: [True]
```

Figure 18: Output Log

The network used in this example was trained on the German Traffic Sign Recognition Benchmark (GTSRB) dataset. There are 43 classes in the network, which is why there are 43 dimensions in the final zonotope.

5.2.1 Generating the Zonotope

Once an image is fed into the program, it is first flattened into a list of vectors (it is a list due to images possibly having multiple color channels). Once the image is flattened, it is fed into the function in Listing 1.

```
createEquations :: U.Vector Double -> Double -> [U.Vector Double]
createEquations zonotope perturbation = let
```

```
  !size = U.length zonotope
  !identity' = M.identity size :: Matrix Double
  !perturbedIdentity = scaleMatrix perturbation identity'
  !perturbedIdentityVectors = map U.fromList (M.toLists perturbedIdentity
    )
  !newZ = zonotope : perturbedIdentityVectors
  in newZ
```

Listing 1: Function to construct perturbed zonotope equations

This function perturbs each dimension by some value. It does so by creating a unique error term of the same value for each dimension. This can be done easily by creating a scaled identity matrix and joining that with the image vector (Note: this follows the matrix representation of zonotopes discussed earlier). This creates a hypercube around the original image.

5.2.2 Parsing through the Layers

As mentioned previously, a network can be thought of as a composition of functions. Each layer is passed through recursively using the function in Listing 2. The layer name is checked and its functionality is applied accordingly onto the zonotope.

```
parseLayers :: Handle -> [LayerInfo] -> [[U.Vector Double]] -> (Int,Int)
    -> IO [[U.Vector Double]]
parseLayers _ [] zonotope _ = return zonotope
parseLayers logFile (l:layers) zonotope (imgRows,imgCols) = ...
```

Listing 2: Function to parse through the network's layers

The function takes the following inputs:

- layers :: [LayerInfo]:

  Each item in the list is a layer of the network. LayerInfo stores the name and type of the layer, along with any relevant information corresponding to the layer (like kernels, weights, biases)

- zonotope :: [[U.Vector Double]]:

  The zonotope that is inputted to the function.

- (imgRows,imgCols) :: (Int,Int):

  This is what the dimensions of the image would be before flattening. This is important for convolution and pooling as it is needed to determine the new dimensions of the zonotope after passing through these layers. These get updated as needed at each function call of parseLayers.

The next sections will go through the different cases for 'layers'.

5.2.3 Dense Layer

If the layer being parsed is a dense layer, the code in Listing 3 shows what happens to the zonotope.

```
"<class 'keras.src.layers.core.dense.Dense'>" ->
      let !weights' = fromMaybe [] (weights l)
          !weightsMatrix = M.transpose (M.fromLists weights')
          !zonotopeMatrix = M.transpose (M.fromLists (map U.toList (head
    zonotope)))
          !newZ2 = M.toLists (multStd weightsMatrix zonotopeMatrix)
          !biases' = fromMaybe [] (biases l)
          !newZ' = zipWith (\x row -> (head row + x) : tail row) biases'
    newZ2
          !newZ'' = Data.List.transpose newZ'
          !activationFunction' = fromMaybe [] (activationFunction l)
          !newZ = if activationFunction' == "relu"
            then map (map U.fromList) (applyRelu [newZ''])
            else [map U.fromList newZ'']
      in parseLayers logFile layers newZ (imgRows,imgCols)
```

Listing 3: Case where the layer is a dense layer

The weights of the layer are extracted and converted to Haskell's matrix form. The zonotope is also converted to matrix form from the original unboxed vector form. These are multiplied and the bias is added to the product. Finally, the layer is checked for the presence of ReLU and ReLU is applied accordingly.

5.2.4 ReLU

```
composeLambdaAndNuRelu :: [Double] -> [Double]
composeLambdaAndNuRelu eq = let
    !u = reluUpper eq
    !l = reluLower eq
    !lambda = u / (u-l)
    !nu = (u * (1 - lambda)) / 2
    !lambdaScaled = map (* lambda) eq ++ [0]
    !lengthLambdaScaled = length lambdaScaled
    !nScaled = [nu] ++ replicate (lengthLambdaScaled - 2) 0 ++ [nu]
    in zipWith (+) lambdaScaled nScaled

applyReluPerDimension :: [Double] -> [Double]
applyReluPerDimension eq = let
  !l = reluLower eq
  !u = reluUpper eq
  !eqLength = length eq
  in if l > 0
    then eq ++ [0]
    else if u < 0
      then replicate (eqLength + 1) 0
```

```
    else
      composeLambdaAndNuRelu eq

applyRelu :: [[[Double]]] -> [[[Double]]]
applyRelu zonotope = let
  !zonotopeT = map Data.List.transpose zonotope
  !result = map (map applyReluPerDimension) zonotopeT
  !resultT = map Data.List.transpose result
  result' = if all (all (== 0) . last) resultT then map init resultT else
     resultT
  in result'
```

Listing 4: ReLU

The algorithm described earlier for ReLU is implemented using the code in Listing 4. The zonotope is inputted to the function and each dimension is acted on individually.

The upper and lower bounds for each dimension are calculated and they are used to find the values of λ and ϵ_{new} (represented by nu in the code). The input is scaled by λ and ϵ_{new} is added to the scaled zonotope dimension.

5.2.5 Convolution

```
generateSparseConvMatrix :: [[Double]] -> (Int, Int) -> [(Int, Int,
    Double)]
generateSparseConvMatrix kernel (inpRows, inpCols) =
  let !kernelRows = length kernel
      !kernelCols = length (head kernel)
      !yRows = inpRows - kernelRows + 1
      !yCols = inpCols - kernelCols + 1
      !outputSize = yRows * yCols
      !kernelFlat = U.fromList (concat kernel)

      -- Parallel generation of matrix entries
      !entries = concat $ parMap rdeepseq (generateEntriesForOutputIndex
    (inpCols, kernelRows, kernelCols, kernelFlat))
                  [0..outputSize-1]
      in entries
```

Listing 5: Generating W^F for convolution

In Listing 5 the output size after convolution is calculated to determine how many rows need to be generated in the sparse matrix. This is determined easily using the input image's dimensions and the kernel's dimensions. Each row index is passed to the function in Listing 6.

```
generateEntriesForOutputIndex :: (Int, Int, Int, U.Vector Double) -> Int
    -> [(Int, Int, Double)]
generateEntriesForOutputIndex (inpCols, kr, kc, kernelFlat) outIdx =
  let (!i, !j) = outIdx `divMod` (inpCols - kc + 1)
```

```
      !baseRow = i * inpCols + j
  in [ (outIdx, baseRow + dj + di * inpCols, kernelFlat `U.unsafeIndex` (
    di * kc + dj))
     | di <- [0..kr-1]
     , dj <- [0..kc-1]
     ]
```

Listing 6: Generating W^F for convolution

The function in Listing 6 constructs a list of sparse matrix entries representing the contribution of input elements to a specific output index in a 2D convolution operation.

The line '(i, j) = outIdx divMod (inpCols - kc + 1)' converts outIdx into 2D coordinates (i, j) in the output matrix. From this, baseRow is calculated which is the starting index in the flattened input matrix where the kernel is applied. From the base row, for each kernel element (di, dj), the kernel element is assigned an index.

This function generates a list of (rowIndex, columnIndex, weight) tuples, representing how the input (with weight) contributes to the output.

5.2.6 Average Pooling

```
generateSparsePoolingMatrix :: Int -> Int -> [[Double]] -> [(Int, Int,
    Double)]
generateSparsePoolingMatrix p q x =
  let !rows = length x
      !cols = length (head x)
      !numOutputRows = rows `div` p
      !numOutputCols = cols `div` q
      !outputSize = numOutputRows * numOutputCols
      !poolSize' = fromIntegral (p * q)

      -- Parallel generation of entries
      !entries = concat $ parMap rdeepseq
                 (\outIdx -> generatePoolingEntries p q cols numOutputCols
     outIdx poolSize')
                 [0..outputSize-1]
  in entries

generatePoolingEntries :: Int -> Int -> Int -> Int -> Int -> Double -> [(
    Int, Int, Double)]
generatePoolingEntries p q cols numOutputCols outIdx poolSize' =
  let (!i, !j) = outIdx `divMod` numOutputCols
      !inputStartRow = i * p
      !inputStartCol = j * q
      !scaleFactor = 1.0 / poolSize'
  in [ (outIdx, (inputStartRow + di) * cols + (inputStartCol + dj),
    scaleFactor)
     | di <- [0..p-1]
```

```
    , dj <- [0..q-1]
    ]
```

Listing 7: Generating W^{AP} for average pooling

Similar to convolution, a sparse weight matrix is generated for average pooling. The difference is in the positioning of the weights and the weights themselves. The values of the weights are decided by the pooling size, and the position is determined by the pooling groups.

5.2.7 Classifying the Results

Once the zonotope reaches the output layer, it needs to be checked for which class it belongs to. Traditionally, most networks classify results using argmax, which just checks which output node has the highest value. However each node here will have a 1-D zonotope for which you can not just check max value so easily. The work around is to ensure that the lower bound for the expected class is greater than all other classes.

$$\text{If the expected output is class i,} \quad \forall j \neq i, \quad ub_j < lb_i$$

Where ub and lb represent the upper bound and lower bound for the zonotope's dimension.

6 Experiments

To verify the safety of the GTSRB network, some images from the dataset were fed back into the network as zonotopes. One input was the image below:

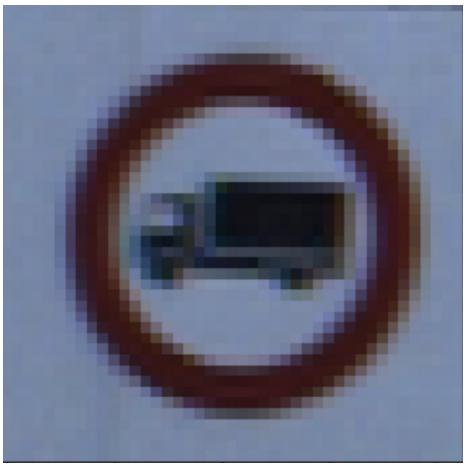
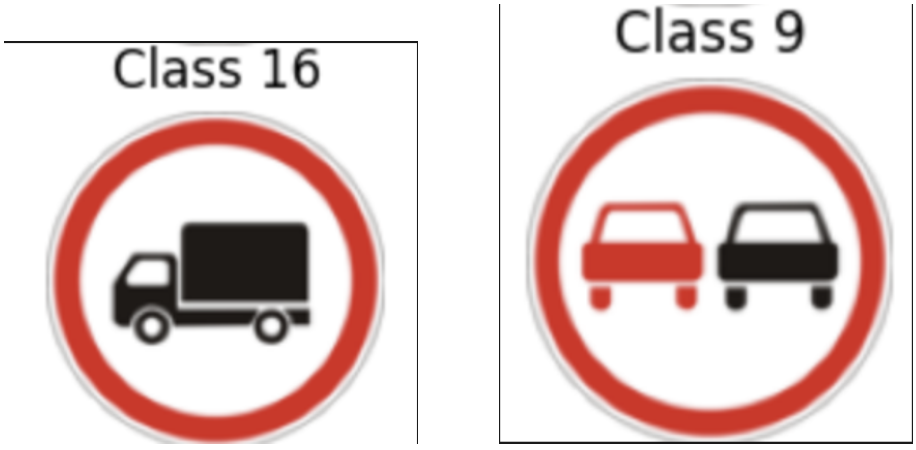


Figure 19: Input Image [5]

For a perturbation amount of 0.0142, the network always classifies all images in the zonotope correctly, that is in the expected class '16'. When perturbation is increased to 0.0143 the network becomes unsafe. Class 9's upper bound is higher than the lower bound for the expected class which means that for some images with pixels perturbed by 0.0143, the network may misclassify it as class 9.



(a) Expected Class (b) Misclassified Class

Figure 20: Classes from GTSRB Dataset [5]

7 Challenges

Formal verification is a very computationally intensive process, and coding everything from scratch was quite a challenge. After getting all the functions to perform correctly on small datasets, running them for a real network crashed my laptop. I then connected to a high performance computer (HPC) with 256 GB RAM and 64 cores to test the code, but even on that the code crashed. The next step was to find possible optimisations.

The first optimisation was changing the data structures used. For the matrix representation of zonotopes, I was initially using Haskell's built in list structure. This was very inefficient and I switched to a library that uses Matrices instead. After refactoring the code to the new data type, the code ran for a while, but there was too much memory build up and it crashed after running for around 2 hours.

The memory build up was due to Haskell's lazy evaluation system. An easy fix to this was the use of strict evaluators, which prevent lazy evaluation by forcing the program to perform any computations then and there. In addition to this I also added parallelisation to the code.

After these two additions, the code was running for longer and passing through some layers. However, for each run, after around 20 hours it crashed

again due to memory build up.

The next optimisation was the use of sparse matrices as mentioned earlier in this paper. That greatly improved efficiency, and the code was able to pass through the entire network in around 3 hours.

This was a big improvement, but still not sufficient. To improve further, I added more parallelisation and changed the data structure used from Matrices to Unboxed Vectors, which are more efficient than Matrices. The final version of the code now passes through a network in less than 5 minutes on the HPC.

References

[1] Aws Albarghouthi. *Introduction to Neural Network Verification*. University of Wisconsin–Madison, 2021.

[2] Gehr et al. *AI^2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation*. ETH Zurich, 2018.

[3] Singh et al. *Fast and Effective Robustness Certification*. ETH Zurich, 2018.

[4] Martin Vechev. *Lecture 6: Reliable and Interpretable Artificial Intelligence*, ETH Zurich, 2020. `https://files.sri.inf.ethz.ch/website/teaching/riai2020/materials/lectures/LECTURE6_ZONO.pdf`, accessed 2025-04-19.

[5] German Traffic Sign Recognition Benchmark Dataset, Kaggle. `https://www.kaggle.com/datasets/meowmeowmeowmeowmeow/gtsrb-german-traffic-sign`, accessed 2025-04-21.

[6] Great White Shark Image from WorldWildLife, `https://files.worldwildlife.org/wwfcmsprod/images/Great_White_Shark_Circle_and_Hero_Image/hero_full/56ksgfki3k_Species_Great_White_Shark_WWF_Carlos_Aguilera.jpg`, accessed 2025-04-21.

[7] Dense Layer Image from Medium, `https://python-course.eu/images/machine-learning/weights_input2hidden_1.webp`, accessed 2025-04-21.

[8] ReLU Image from Medium, `https://miro.medium.com/v2/resize:fit:1400/1*DfMRHwxY1gyyDmrIAd-gjQ.png`, accessed 2025-04-21.

[9] Pooling Image from Medium, `https://miro.medium.com/v2/resize:fit:517/0*lIcR0gOMK0xzTr6_.png`, accessed 2025-04-21.