Honour School of Engineering Science

B1 Engineering Computation: Project B (2021-22)

Neural Network Verification

Analysing Robustness of Multi-Layer Perceptron

Student: Tijana Petrovic

Supervisor: Prof. Pawan Mudigonda

January 2022

Department of Engineering Science

Oxford University

Table of Contents

1 Introduction

A neural network is a network or circuit of (artificial) neurons or nodes represented by their weights, biases, and an input domain χ. They can be trained via a dataset to model complex relationships between inputs and outputs or find patterns in data. Thanks to this self-learning property, they are widely used in various applications of artificial intelligence such as facial recognition, signature verification, stock market, and protein folding prediction. Despite this, their use is very limited in safety-critical applications (such as autonomous driving cars) due to their susceptibility to adversarial attacks where small perturbations in the input lead to drastic changes in the network output.

This project aims to overcome this deficiency by using the formal verification of neural networks to test the hypothesis whether the given neural network satisfies a desirable property. Our proposed methods are Unsound method, Projected Gradient Ascent, Interval Bound Propagation, and Linear Programming method under the Branch and Bound framework. The Collision Detection dataset consisting of 500 MAT files is used for testing.

1.1 Problem statement

A certain function f is represented by a piecewise linear multilayer perception (MLP) with n layers as $f_n \circ f_{n-1} \circ \ldots \circ f_1(\boldsymbol{x})$ where f_i represents i^{th} layer. The MLP is specified by its weights $\boldsymbol{W}$, biases $\boldsymbol{b}$ and an input domain $\chi = \{\boldsymbol{x} | \boldsymbol{x}_{min} \leq \boldsymbol{x} \leq \boldsymbol{x}_{max}\}$, where $\boldsymbol{x}_{min}$ and $\boldsymbol{x}_{max}$ are box constraints. Outputs of each layer can be obtained using $f_i(z_{i-1}) = z_i = \sigma(\hat{z}_i)$ and $\hat{z}_i = z_{i-1} + \boldsymbol{b}_i$ up until the last layer. The output of the last layer $z_n = \hat{z}_n$ is denoted by a vector $\boldsymbol{y} = z_n = f(z_0) = f(\boldsymbol{x})$.

Given that the neural network implements a function $z_n = f(z_0)$ over a bounded input region χ and a property of robustness to adversarial examples, P, we want to prove that if $z_0 = x \in \chi$ and $\hat{z}_n = f(z_0)$, then $P(\hat{z}_n)$ is true.

Consider ∞-norm around a training sample $\boldsymbol{a}$ with label y_a. The desired property is encoded by using $\chi = \{z_0 | \ ||z_0 - \boldsymbol{a}||_\infty \leq \varepsilon\}$ and $P(\hat{z}_n) = \{\forall y, \ \hat{z}_{n[y_a]} \geq \hat{z}_{n[y]}\}$. A neural network characterized by a function f is said to be robust at z_0 if for $\forall(z_0 - \boldsymbol{a})$ in the input domain $\chi = \{z_0 | \ ||z_0 - \boldsymbol{a}||_\infty \leq \varepsilon\}$ the weight of the correct label $\hat{z}_{n[y_a]}$ is greater or equal to the weight of undesired label $\hat{\boldsymbol{z}}_{n[y]}$ for the

property $P(\hat{z}_n) = \{\forall y,\ \hat{\mathbf{z}}_{n[y_a]} \geq \hat{\mathbf{z}}_{n[y]}\}$. Otherwise, a neural network is not robust at z_0, and $(z_0 - a)$ for which this is the case is described as an adversarial.

1.2 Problem statement in Canonical form

The problem can also be stated in canonical and mixed integer programming form.

For the canonical form, we want to reduce the verification problem to a simple global optimization problem and check the sign of the output of the modified neural network $f'(z_0)$. This is possible if the property can be expressed as a Boolean expression over linear forms.

If a property satisfies $P(\hat{z}_n) = W^{\mathrm{T}}\hat{\mathbf{x}}_{\mathbf{n}} \geq \boldsymbol{b}$, it is enough to add a final fully connected layer with one output, weight $\boldsymbol{W}$ and bias -$\boldsymbol{b}$. If the global minimum of the network is positive, it is indicated that $W^{\mathrm{T}}\hat{\mathbf{x}}_{\mathbf{n}} - \boldsymbol{b} \geq 0$, or $W^{\mathrm{T}}\hat{\mathbf{x}}_{\mathbf{n}} \geq \boldsymbol{b}$, which implies the property is true. Otherwise, the minimizer provides a counterexample.

Assuming the network only contains ReLU activations between each layer, the satisfiability problem to find a counter example could be expressed as:

$\boldsymbol{l}_0 \leq \boldsymbol{x}_0 \leq \boldsymbol{u}_0, \qquad \hat{x}_n \leq 0, \qquad \hat{\boldsymbol{x}}_{i+1} = \boldsymbol{W}_{i+1}\boldsymbol{x}_{\mathbf{i}} + \boldsymbol{b}_{\mathbf{i}+1}$ and $\boldsymbol{x}_{\mathbf{i}} = max(\hat{\mathbf{x}}_i, 0)$ for $\forall i \in \{0, n-1\}$.

Now, we can simplify the problem into computing the upper and lower bounds of the output where:

- the property is true if its upper bound is non-positive.
- the property is false if its lower bound is positive.

2 Lower bound for Neural Network output

It has been shown that the property is false if its lower bound is positive. Here, we will consider two different methods for obtaining lower bounds of the neural network: Unsound method and Projected Gradient Ascent.

2.1 Unsound method

The unsound method consists of two functions: *generate_inputs* and *compute_nn_outputs*. The former generates k random inputs within the box constraints set by $\boldsymbol{x}_{min}$ and $\boldsymbol{x}_{max}$. For this, we use a uniformly distributed random number generator: $\boldsymbol{X}(i) = \boldsymbol{x}_{min} + (\boldsymbol{x}_{max} - \boldsymbol{x}_{min}) \cdot rand(0, 1)$. Next, the

function *compute_nn_outputs* takes the input array, X, to calculate k corresponding outputs for each input using Algorithm 1. The maximum value among k outputs is then a valid lower bound on $\hat{z}_n$.

Algorithm 1 Compute NN Outputs

function compute nn outputs(**W**, b, **X**)

$z \leftarrow transpose(\mathrm{X})$
for l $1, ..., L-1$ **do**
  $\hat{z} \leftarrow W_l \cdot z + b_l$
  $z \leftarrow max(\hat{z}, 0)$
end for
$y \leftarrow W_L \cdot z + repmat(\mathrm{b}_L, 1, size(\mathrm{W}_L \cdot z, 2))$
$y \leftarrow transpose(y)$
return y

2.2 Projected Gradient Ascent

The unsound method helps us to find counterexamples quickly. However, results could be improved if we considered verification as a problem of finding the optimal solution of maximising $f(\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{x}^{\min}, \boldsymbol{x}^{\max})$ and perform gradient ascent iteratively.

Algorithm 2 Projected Gradient Ascent

function projected gradient ascent(**W**, b, X, x^{min}, x^{max})

for i $= 1, ...size(X)$ **do**
  $X_k = transpose(X(k,:))$
  for iteration $= 1, ...,$ l **do**
    $a = X_k$
    for $i = 1, ...$Number of Layers **do**
      $z = W(i) \times a + b(i)$
      for $j = 1, ...$ size(z) **do**
        Populate gradient matrix:
        if $z(j) > 0$ **then** $dfdx(j,j) = 1$
        else $dfdx(j,j) = 0$
        Compute the gradient of the current layer
        if $i \leq$ Number of layers-1 **then** $dadx = dfdx \times W(i)$
        else $dadx = W(i)$
        Compute the product of gradients of the all previous layers
        $a = max(z, 0)$
        Update X_k as $X_k + lrate \times transpose(propagated\ dadx)$
        Clip X
      end for
      $refinedX(k) = transpose(X_k)$
    end for
  end for
end for
return $refinedX$

Random input vector x_0 will be generated and updated as follows:

$$\boldsymbol{x}'_{t+1} = \boldsymbol{x}_t + \eta_t \nabla f(\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{x}_t)$$

Here, η_t represents the step-size at iteration t and $\nabla f(\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{x}_t)$ is the gradient of the function f at the point $\boldsymbol{x}_t$. We will also approximate the derivative at 0 as 0 and clip all the values $\boldsymbol{x}'_{t+1}$ that lie outside the domain χ to their nearest value in domain χ.

The pseudocode can be seen in Algorithm 2 given on the left.

2.3 Results

Results from the Unsound method and Projected Gradient Ascent are given in figures below.

As the value of k increases, the average lower bound obtained using the Unsound method also increases. This is because taking a larger number of inputs that satisfy the box constraints leads to a greater probability of finding z_0 that generates a stricter lower bound.

Comparing two figures, it is apparent that Projected Gradient Ascent gives better limits and higher number of found counterexamples than the Unsound method as predicted.

Figure 1 (Unsound method):

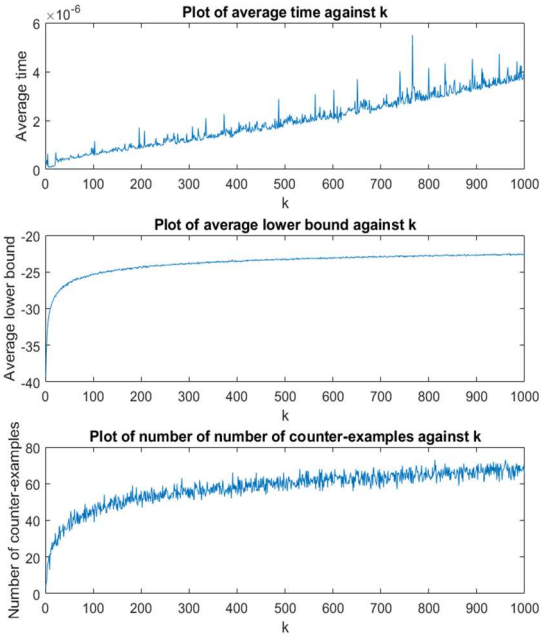
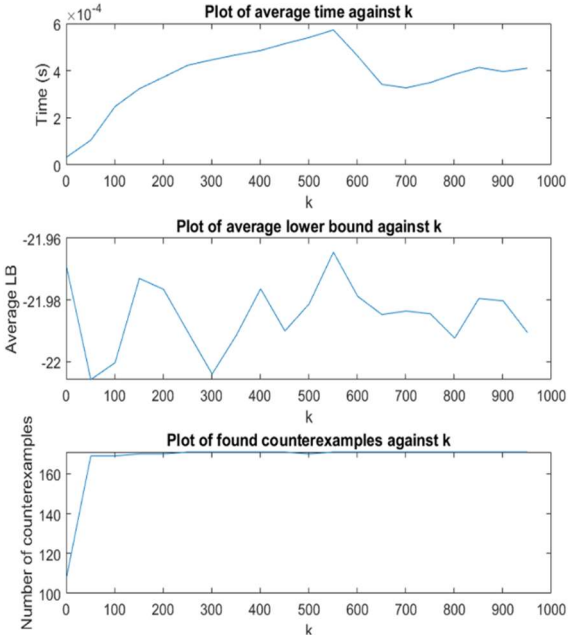


Figure 2 (Projected Gradient Ascent):



2.4 Further considerations

The Unsound method could be improved by using a grid search over the input domain instead of random sampling. This would create better coverage of the input domain and counterexamples could be found more easily.

3 Upper bound for Neural Network output

Previously introduced methods can be used efficiently for finding counterexamples of the neural network. However, they cannot be used for proving which properties are true. This is a huge limitation for the purpose of this project. In this chapter, we propose Interval Bound Propagation as a method of finding upper bound of the neural network.

3.1 Interval Bound Propagation

The Interval Bound Propagation algorithm is based on the artificial deconstruction of the weight matrix at each layer into matrices with positive and negative entries ($W_1 = W_l^- + W_l^+$). Then upper and lower bounds are computed for the output of each layer using these new matrices and the bounds of the previous layer as shown in the Algorithm 3.

Algorithm 3 Interval Bound Propagation

function inteval bound propagation($\mathbf{W}$, $\mathbf{b}$, x^{min}, x^{max})

$z_l^{min} \leftarrow transpose(\mathrm{x}^{min})$
$z_l^{max} \leftarrow transpose(\mathrm{x}^{max})$
for i $= 1, ..., L-1$ **do**
  $W_l^+ = max(0, W_l)$
  $W_l^- = min(0, W_l)$
  $z_l^{\hat{max}} = W_l^+ \cdot z_l^{max} + W_l^- \cdot z_l^{min} + b_l$
  $z_l^{\hat{min}} = W_l^+ \cdot z_l^{min} + W_l^- \cdot z_l^{max} + b_l$
  $z_l^{max} = max(0, \hat{z_l}^{max})$
  $z_l^{min} = max(0, \hat{z_l}^{min})$
end for
$y^{min} = W_L^+ \cdot z_L^{min} + W_L^- \cdot z_L^{max} + b_L$
$y^{max} = W_L^+ \cdot z_L^{max} + W_L^- \cdot z_L^{min} + b_L$
$=0$

3.2 Results

Running the Interval Bound Propagation algorithm on the Collision Detection dataset took an average of 0.0016 seconds per property. It verified 11 out of 328 properties as true and has not proven any properties to be false. The average computed lower bound was -137.9, and the average upper bound was 82.7. The performance is poor because the algorithm alters the network quite drastically which leads to loose bounds that rarely approach zero for the given dataset.

It is worth noting that Interval Bound Propagation has low computational cost but lower bounds computed using both, Unsound method and Projected Gradient Ascent are much higher and therefore better.

4 Linear programming method

The disadvantage of interval bound propagation is that obtained bounds are very loose which means that we will not be able to prove a significant number of properties. Although using the Unsound method together with Interval Bound Propagation could improve the lower bounds there is still space for further improvement. Here, we introduce an algorithm that uses optimization at each layer and then solves for the minimum and maximum output of all previous layers. The computational difficulty of this problem is in the non-linearity and non-convexity of the final sets of constraints due to ReLU activations. This converts our problem into an NP-hard problem that can be solved by approximating non-linear constraints to linear constraints.[2]

Algorithm 4 Linear Programming Bound

function linear programming bound(**W**, **b**, x^{min}, x^{max})

$z_0^{min} \leftarrow transpose(\mathrm{x}^{min})$
$z_0^{max} \leftarrow transpose(\mathrm{x}^{max})$
Define matrices f, A and B
for i $= 1, ..., L$ **do**
  for j $= 1, ...,$ Number of Neurons in the Layer **do**
    Apply constraints
  end for
  Compute linear program for each Neuron in the Layer
  $z_l^{min} = max(0, \hat{z}_l^{min})$
  $z_l^{max} = max(0, \hat{z}_l^{max})$
  Update matrices A and B
end for
LB $= \hat{z}_L^{min}$
UB $= \hat{z}_L^{max}$
return [LB, UB]

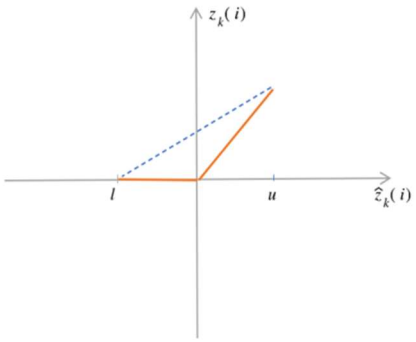
The pseudocode of the method is given in the Algorithm 4. The built-in MATLAB function *linprog* was used for optimisations, while A and B represent equality matrices.

4.1 Planet Relaxation of ReLU function

Planet Relaxation operates by explicitly attempting to assign a number to the phase of the non-linearities. It assigns either 0 or 1 to each $\delta_{i[j]}$ variable, verifying at each step the feasibility of the partial assignment. This allows us to prune infeasible partial assignment early.

The search strategy enumerates all the domains that have not been pruned yet, while the branching rule fixes the decision variable $\delta_{i[j]}=0$ which is equivalent to choosing $x_{i[j]}=0,\ \hat{x}_{i[j]}\leq 0$ and fixing $\delta_{i[j]}=1$ which is equivalent to $x_{i[j]}=0,\ \hat{x}_{i[j]}\geq 0$.

Figure 3 (Planet Relaxation):



$$z_k^{min}(i) \equiv min\ z_k(i)$$

- $z_k(i) > 0$
- $z_k(i) > \hat{z}_k(i)$
- $z_k(i) < m\left(z_k(i) - z_k^{min}(i)\right)$

Where $m = \dfrac{\hat{z}_k^{max}}{\hat{z}_k^{max} - \hat{z}_k^{min}}$

Planet Relaxation does not include any heuristic to prioritise which decision variable should be split over or a mechanism for early termination based on a heuristic search of a feasible point. Consequently, only when a full complete assignment identified the solution is returned. However, we can detect incoherent assignments if we approximate the non-linear constraints by a set of linear constraints representing the non-linearities' convex hull. It also uses conflict analysis to discover combinations of splits that cannot lead to satisfiable assignments, allowing them to perform further pruning of the domains.

Except for the planet relaxation, parallel relaxation and mixed-integer encoding could be used for the same purpose. [1][3]

4.2 Results

Linear Programming Method has proved 260 properties, which took on average 6.6 seconds per property. The obtained lower limit is -58.7 and upper limit is -13.2. It is apparent that it preforms significantly better than Interval Bound Propagation but at cost of higher computational time.

5 Branch and Bound for Verification

In this section, an approach of estimating the global minimum is introduced. As the name suggests, Branch and Bound (BaB) iteratively partitions the input domain into a set of subdomains that are collectively exhaustive and uses bounding methods for estimating lower and upper bounds. This is possible due to the piecewise linear property of the neural network. A partition $\mathcal{P}$ of size n is a set $\{\chi_1, \chi_2, \cdots, \chi_n\}$ such that $\chi_1 \cup \chi_2 \cup \cdots \cup \chi_n = \chi$ and is initialized as $\mathcal{P}_0 = \{\chi\}$. The general structure of the BaB is shown in Algorithm 5.

Algorithm 5 Branch And Bound

function branch and bound(**W**, b, x^{min}, x^{max})

1: $\bar{z}^{min} \leftarrow transpose(\mathrm{x}^{min})$
2: $\bar{z}^{max} \leftarrow transpose(\mathrm{x}^{max})$
3: $bounds \leftarrow getBounds(x^{min}, x^{max})$
4: **if** LowerBound> 0 **then**
5: flag $= 0$
6: **return** flag
7: **end if**
8: **for** $i = 1, ..., L$ **do**
9: [maxUB, index] $\leftarrow max$(UpperBounds)
10: **if** maxUB< 0 **then**
11: flag $= 1$
12: **return** flag
13: **end if**
14: $[x_1^{min}, x_1^{max}, x_2^{min}, x_2^{max}] \leftarrow partOut$(bounds)
15: $bounds_1 \leftarrow getBounds(x_1^{min}, x_1^{max})$
16: $bounds_2 \leftarrow getBounds(x_2^{min}, x_2^{max})$
17: **if** $LowerBound_1 > 0$ or $LowerBound_2 > 0$ **then**
18: flag $= 0$
19: **return** flag
20: **end if**
21: **update** Bounds
22: **end for**
23: flag $= 2$
24: **return** flag

The algorithm has three elements:

Search strategy – chooses the next domain to branch on. We always choose the domain with the maximum upper bound because if the highest obtained upper bound is negative, all the other upper bounds are negative and consequently, the property is true.

Branching strategy – takes the previously chosen domain and returns a partition in its subdomain such that $subdomain_1 \cup subdomain_2 = domain$ and $subdomain_1 \cap subdomain_2 = \emptyset$

Bounding strategy – produces upper and lower bounds over the minimum output that the network can reach over a given input domain.

There are two possible cases that arise can from the bound strategy:

- **Case 1**: $l > 0$ $f'(x_0)$ is positive for at least one $z_0 \in \chi$ ⇒ property is false
- **Case 2**: $u < 0$ $f'(x_0)$ is negative for all $z_0 \in \chi$ ⇒ property is true

However, although the procedure will terminate after a finite number of iterations, this might take too long, and, therefore, third case of slow convergence arises:

- **Case 3**: $l < 0$ and $u > 0$ unable to determine the validity of property of the neural network.

Encountering the case 3 would mean that our method is incomplete and should be replaced by another one. Therefore, it is important to find methods that provide tighter bounds on z_n and efficient ways of partitioning the input domain into sub-domains in search for the optimal bounds on z_n.

5.1 Branch Strategy

5.1.1 Splitting the longest edge

Assuming the piecewise linear property, the function partitions the input domain into subdomains by finding the dimension of the longest relative length $s(i)$: $s(i) = \frac{\bar{x}^{max}(i) - \bar{x}^{min}(i)}{x^{max}(i) - x^{min}(i)}$ and $j = \underset{i}{argmax}(s(i))$, where x^{min} and x^{max} are the box constraints of the input domain χ, and $\bar{x}^{min}$ and $\bar{x}^{max}$ are the box constraints of the input sub-domain $\bar{\chi}$ which generates the highest upper bound.

We split the subdomain into two along its j[th] dimension using following rule:

$$\bar{x}_1^{min} = \bar{x}^{min}, \qquad \bar{x}_1^{max} = \begin{cases} \bar{x}^{max}(i) & \text{if } i \neq j, \\ \bar{x}^{min}(i) + \frac{\bar{x}^{max}(i) - \bar{x}^{min}(i)}{2} & \text{otherwise.} \end{cases}$$

$$\bar{x}_2^{max} = \bar{x}^{max}, \qquad \bar{x}_2^{min} = \begin{cases} \bar{x}^{min}(i) & \text{if } i \neq j, \\ \bar{x}^{min}(i) + \frac{\bar{x}^{max}(i) - \bar{x}^{min}(i)}{2} & \text{otherwise.} \end{cases}$$

χ_1 and χ_2 computed in this way are guaranteed to generate an upper bound less than or equal to that of the χ, and a lower bound greater than or equal to that of χ.

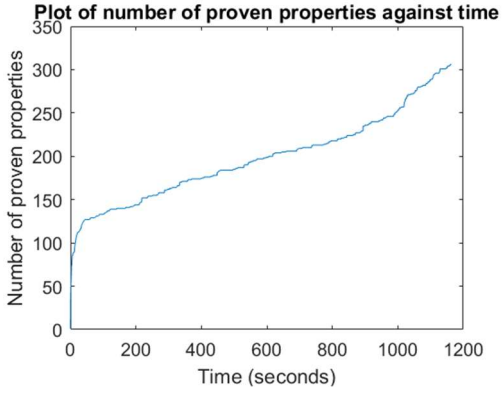
Other branching methods that could be used are random splitting and largest edge splitting. Random splitting reduces time complexity of branching, while largest edge method splits the domain so that it has the greatest effect on the output of the network. [1]

5.2 Results

5.2.1 Interval Bound Propagation and Unsound Method

As shown in the Figure 4, it took 3.8 seconds per property to prove 306 properties using a combination of Unsound method and Interval Bound Propagation under the Branch and Bound framework. Although we did not manage to prove all properties using the chosen convergence limit, this method is very useful and has good computational cost.

Figure 4 (BaB using IBP and UM):

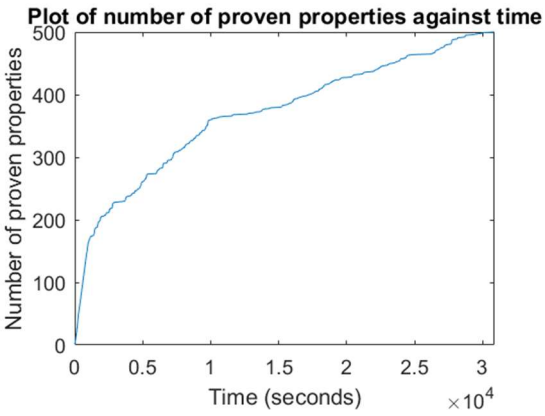


5.2.2 Linear Programming Bounds and Unsound Method

Under the Branch and Bound framework, a combination of Linear Programming and Unsound method took on average 60.1 seconds per property to prove all 500 properties correctly. Plot of

number of proven properties against time can be seen in the Figure 5 on the right. All considered, this method is considerably better than all previously considered ones.

Figure 5 (BaB using LP):



5.2.3 Hybrid

The Hybrid method first tries to prove a property using the Unsound method and Interval Bound Propagation and then moves onto the Linear Programming if it was not successful. It proved all properties and took about 40 seconds for each which is considerably better than all previously considered methods.

6 Conclusions

To conclude, the bounding strategies alone did not prove enough properties to verify the neural network. However, combining bounding strategies with branching strategies withing the Branch and Bound framework led to significant improvement in accuracy. It is worth noting that the Collision Detection dataset is relatively small, so it is not hard to achieve a reasonable performance. With larger networks, the performance of these methods would be considerably worse, and, therefore, it is difficult to say without further research whether this is an effective general method for neural network verification in its current form.

7 References

[1] R. Bunel, I. Turkaslan, P. Torr, P. Kohli, and M. P. Kumar. A unified view of piecewise linear neural network verification. In Advances in Neural Information Processing Systems, 2018.

[2] R. Anderson, J. Huchette, W. Ma, C. Tjandraatmadja, and J. P. Vielma. Strong mixed-integer programming formulations for trained neural networks. In International Confernece on Integer Programming and Combinatorial Optimization, 2019.

[3] Rudiger Ehlers: Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks, 2017