B1 Engineering Computation: Project B (2019-2020) Neural Network Verification

Candidate Number: 1093288

Contents

1 Introduction

Neural networks have emerged in recent years as a much faster method of pattern recognition than traditional computing, but though they are powerful, they are also vulnerable to adversarial attack. Due to their ubiquity, it is essential to develop methods for verifying neural networks. For example, consider a simplified network used in a self-driving vehicle designed to identify the speed limit on an image of a road sign: if an image shows 20mph, but the changing of a single pixel (this is our adversarial attack) causes that image to be misidentified as 80mph, this would be extremely dangerous. As a result, it is crucial to be able to prove the input region which will not unsafely change the image classification. We do this by defining properties with ranges of inputs, and formally proving which properties are true (i.e. those whose output domain lies entirely within that of 'safe' classifications), and which have counter examples proving them false (some outputs in the 'unsafe' error domain).

In this report we will explore various methods of neural network verification for a multi-layer perceptron. We will demonstrate their implementation in MATLAB, the advantages and disadvantages of each, and produce a final algorithm that quickly and efficiently gives a correct result across all properties.

2 Lower Bound for Neural Network Output

This is an unsound method that can be used to prove that some false properties are indeed false by generating a counter example. Whether a property is true or false is knowable if we know the maximum output over all possible inputs, denoted as y^*. We cannot find this easily, but by using this method we can find a lower bound on it. This is done simply by generating a number of random input sets within the input domain, using the function below:

```
function output = generate_inputs(xmin, xmax, k, input_vector_size)
output = ((rand(k,input_vector_size).*(xmax - xmin)) + xmin)';
```

Using $k = 1000$ random inputs per property, a matrix of 1000 column vectors is generated and fed into the following function:

```
function y = compute_nn_outputs(W, b, X)
n = 1;  % current layer
zn = X;
while n < size(W,2)
    zn = W{1,n}*zn + b{1,n};     % apply weights and biases
    zn = (zn + abs(zn))/2;       % ReLU function
    n = n+1;
end
zn = W{1,n}*zn + b{1,n};         % No ReLU at output
y = zn;
```

All 1000 outputs are solved simultaneously using efficient MATLAB matrix operations, and the largest one is our lower bound on y^*. Using a loop we perform this operation on all 500 properties and generate a vector of lower bounds on y^*, which we can turn into a vector of Boolean flags with the following line (note that all bounds tested in this report use a form of this line to turn bounds into labels):

```
results = (y_star_lb - abs(y_star_lb))./(2*y_star_lb);
```

To test `results` we use the `compare_with_groundtruth` function below:

```
function num_correct = compare_with_groundtruth(input)
load('groundtruth.mat');  % file of GT values generated using python script
results = input' + 2*groundtruth; % binary used to produce different combinations
num_correct = sum((zeros(size(input'))) == results); % correctly identified false
num_fails = sum((zeros(size(input'))+1) == results); % failed to prove false
num_errors = sum((zeros(size(input'))+2) == results); % incorrectly identified false
```

Using MATLAB's plot function, we can examine how the success of our method varied with the value of k, as shown in figure 1 and figure 2. This method is very fast, and is extremely useful in identifying false properties, but it cannot guarantee identifying a false property as false, and it

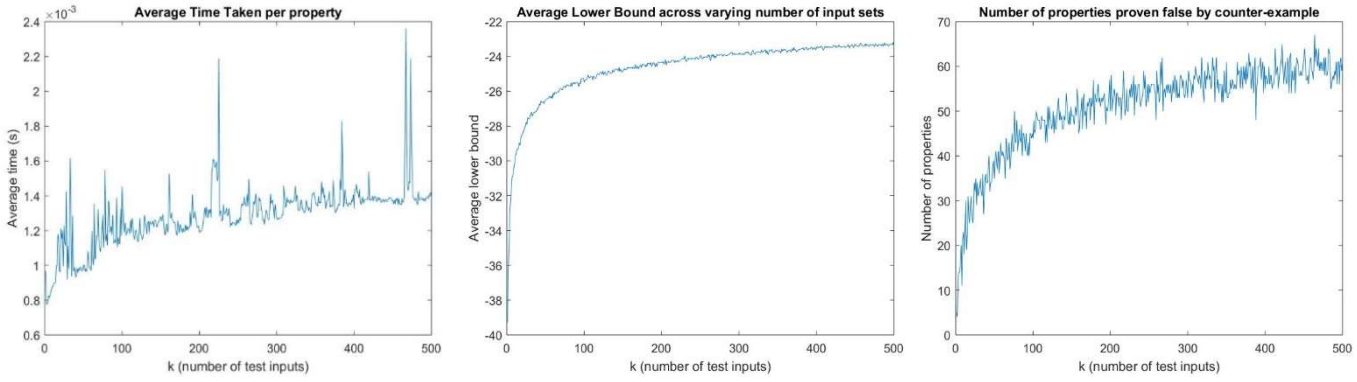

Figure 1: *Graphs of the average time taken per property (left), lower bound on y* found (centre) and number of properties proven false (right) for $k = 1 – 500$. For such low values of k, time taken per property is so low that fluctuations in our processor can drastically affect it. Lower bound and properties proven false appear to be roughly logarithmic.*

cannot prove properties true. To do that we must use a method to find an upper bound on y^*.

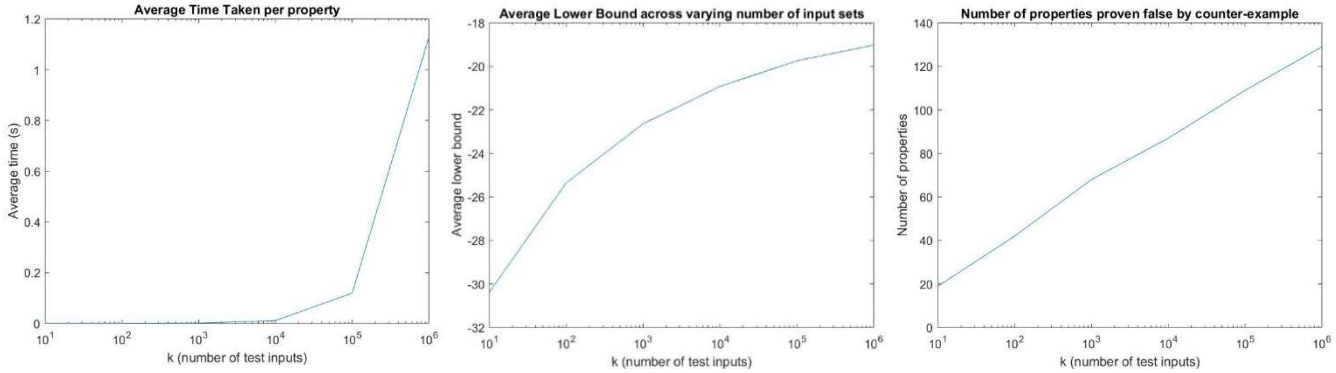

Figure 2: *The same graphs on a logarithmic x-axis varying from k = 10 to k = 1,000,000. For vales of high values of k, magnitude 10^4 and above, the method becomes significantly slower. Our logarithmic approximation for properties proven false proves reasonably accurate, as on a log scale we have roughly a straight line.*

3 Interval Bound Propagation

While our previous method was unsound, allowing us only to identify some false properties, this method is incomplete, allowing us to generate both upper and lower bounds on y^* for our properties, and therefore prove that some of our true properties are indeed true.

```
function [y_min, y_max] = . . .
interval_bound_propagation(W,b,x_min,x_max)

n = 1;  % current layer
zn_min = x_min';
zn_max = x_max';

while n < size(W,2)
    W_p = (W{1,n} + abs(W{1,n}))*0.5;
    W_n = (W{1,n} - abs(W{1,n}))*0.5;
    zn_max_hat = W_p*zn_max + W_n*zn_min + b{1,n};
    zn_min_hat = W_p*zn_min + W_n*zn_max + b{1,n};
    %ReLU
    zn_max = (zn_max_hat + abs(zn_max_hat))/2;
    zn_min = (zn_min_hat + abs(zn_min_hat))/2;
    % increment layer
    n = n+1;
end
% output layer (no ReLU)
W_p = (W{1,n} + abs(W{1,n}))/2;
W_n = (W{1,n} - abs(W{1,n}))/2;
y_max = W_p*zn_max + W_n*zn_min + b{1,n};
y_min = W_p*zn_min + W_n*zn_max + b{1,n};
```

Interval bound propagation finds bounds on the output by finding upper and lower bounds for each element of every layer, by using the upper or lower bounds (as appropriate) of the elements of the previous layer. As a result our bounds will be wider than the true output domain. Our IBP function is shown above, with results in figure 3. The IBP lower bound is limited in its usefulness. As we can see it is a much weaker lower bound on y^* than the one obtained from our stochastic method (as it is a lower bound on the entire output

domain and not y^* specifically). The IBP upper bound, however, gives us an upper bound on y^*, and allows us to prove 11 properties as true by showing that $y^* < 0$.

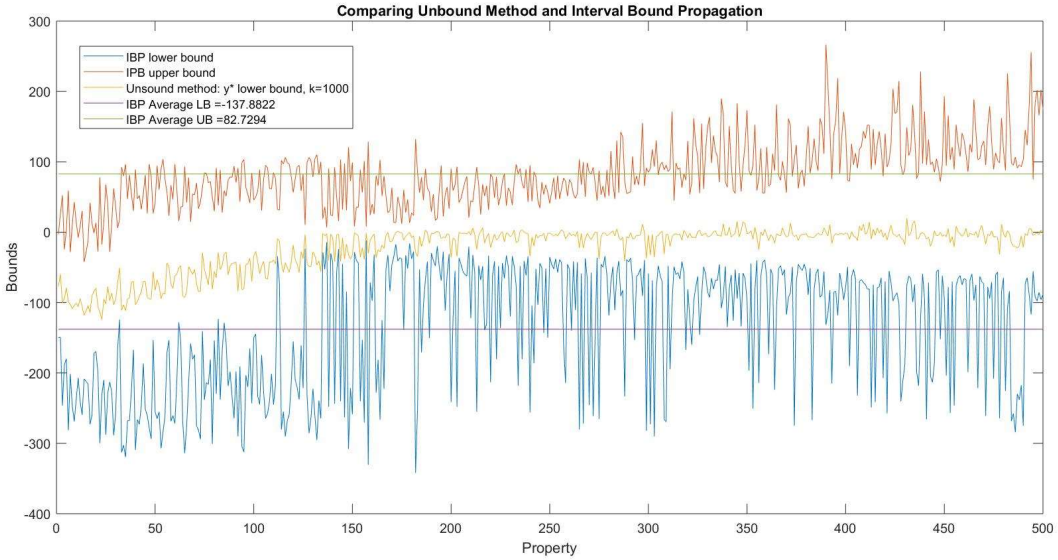

Figure 3: *Upper and lower bounds calculated across all 500 properties using interval bound propagation, as well as the bound averages and the lower bounds on y^* found earlier.*

4 Linear Programming Bound

This method is also incomplete but can improve enormously on the bounds given by interval bound propagation. In the previous method, we determine the bounds of each element in each layer based on the bounds of the previous layer, but this tells us nothing of whether inputs exist that will produce these bounds. In this method we constrain the output domain by using linear programming to account for bounds of all layers. Were it not for the ReLU function this would give us the exact output domain of the property, but as it is nonlinear we are forced to approximate it, producing a loosened domain (though still improved over IBP).

In order to pose this as a linear programming problem, we use the MATLAB function `linprog`, which finds $\min \boldsymbol{f}^T \boldsymbol{v}\ such\ that\ 1\text{:}\{\boldsymbol{A}_{eq} \cdot \boldsymbol{v} = \boldsymbol{b}_{eq}\}\ ,\ 2\text{:}\ \{\boldsymbol{lb} \leq \boldsymbol{v} \leq \boldsymbol{ub}\}\ ,\ 3\text{:}\{\boldsymbol{A} \cdot \boldsymbol{v} \leq \boldsymbol{b}_{nq}\}$ (1)

i.e. an equality constraint, a bound constraint and an inequality constraint.

4.1 Equality Constraint

The value of each element in a layer, calculated using weights of elements in the previous layer plus biases, is an equation we must represent by constraint 1. Thus we must determine `linprog` inputs $\boldsymbol{A}_{eq}$ and $\boldsymbol{b}_{eq}$.

In layer 1, our working vector $\boldsymbol{v}_1$ is $\begin{bmatrix}\hat{\boldsymbol{z}}_1\\ \boldsymbol{x}\end{bmatrix}$ with $\hat{\boldsymbol{z}}_1$ being as-of-yet undetermined, pre-ReLU layer values, and $\boldsymbol{x}$ being the inputs. Therefore we can rewrite the equation $\hat{\boldsymbol{z}}_1 = \boldsymbol{W}_1 \cdot \boldsymbol{x} + \boldsymbol{b}_1$ as $[I \quad -\boldsymbol{W}_1] \cdot \boldsymbol{v}_1 = \boldsymbol{b}_1$, where $\boldsymbol{b}_1$ is the layer 1 biases and $\boldsymbol{W}_1$ is the 40×6 weight matrix.

In layer 2, $\boldsymbol{v}_2$ becomes, $[\hat{\boldsymbol{z}}_2 \quad \boldsymbol{z}_1 \quad \hat{\boldsymbol{z}}_1 \quad x]^T$ (with $\boldsymbol{z}_1$ being $\hat{\boldsymbol{z}}_1$ after applying the approximate ReLU function), our $\boldsymbol{b}_{eq}$ is $\begin{bmatrix}\boldsymbol{b}_2\\ \boldsymbol{b}_1\end{bmatrix}$ and our $\boldsymbol{A}_{eq}$ becomes $\begin{bmatrix} I & -\boldsymbol{W}_2 & & \\ & I & -\boldsymbol{W}_1 \end{bmatrix}$, and this continues in a similar manner for all subsequent layers, so that the general equality constraint is:

$$\boldsymbol{A}_{eq} \cdot [\hat{\boldsymbol{z}}_6 \quad \boldsymbol{z}_5 \quad \hat{\boldsymbol{z}}_5 \quad \cdots \quad \hat{\boldsymbol{z}}_1 \quad \boldsymbol{x}]^T = \boldsymbol{b}_{eq} \text{ where } \boldsymbol{b}_{eq} = \begin{bmatrix}\boldsymbol{b_6}\\ \vdots \\ \boldsymbol{b}_1\end{bmatrix} \text{ and } \boldsymbol{A}_{eq} = \begin{bmatrix} I & -\boldsymbol{W}_6 & & \\ & \ddots & & \\ & & I & -\boldsymbol{W}_1 \end{bmatrix} \quad (2)$$

$\boldsymbol{A}_{eq}$ and $\boldsymbol{b}_{eq}$ are initialised for layer 1 and formed in subsequent layers as follows:

```
Aeq = blkdiag([eye(layer_size{j}), -W{j}],Aeq);
beq = [b{j}; beq];
```

4.2 Bounds

In each layer we must account for the bounds of the previous layer and fill in the current layer's bounds. For layer 1, $ub = [\infty_{40\times1} \quad \boldsymbol{x}_{max}]^T$ and $lb = [-\infty_{40\times1} \quad \boldsymbol{x}_{min}]^T$, as we do not yet know the bounds of $\hat{\boldsymbol{z}}_1$. In layer 2, where our $\boldsymbol{v}_2 = [\hat{\boldsymbol{z}}_2 \quad \boldsymbol{z}_1 \quad \hat{\boldsymbol{z}}_1 \quad \boldsymbol{x}]^T$, $ub = [\infty_{40\times1} \quad \boldsymbol{z}_1^{max} \quad \hat{\boldsymbol{z}}_1^{max} \quad \boldsymbol{x}_{max}]^T$ (and similar for lb), with the bounds for layer 1 being calculated with `linprog`. Due to the ReLU function, $\boldsymbol{z}_1^{max} = \max\{0, \hat{\boldsymbol{z}}_1^{max}\}$.

The upper bound implementation in code for layers 2-6 is as follows, with similar code for lb:

```
ub{j} = [Inf*ones(layer_size{j},1); max(0,z_hat_max{j-1}); ub{j-1}];
```

4.3 Inequality Constraint

Constraint 3 involves approximating the ReLU function to make it suitable for linear programming, as shown in figure 4. We must determine $\boldsymbol{A}$ and $\boldsymbol{b}_{nq}$ that will replicate the inequalities. In the first layer we are only finding the bounds of $\hat{\boldsymbol{z}}_1$, so there is no need for ReLU and we pass `linprog` empty matrices for $\boldsymbol{A}$ and $\boldsymbol{b}_{nq}$.

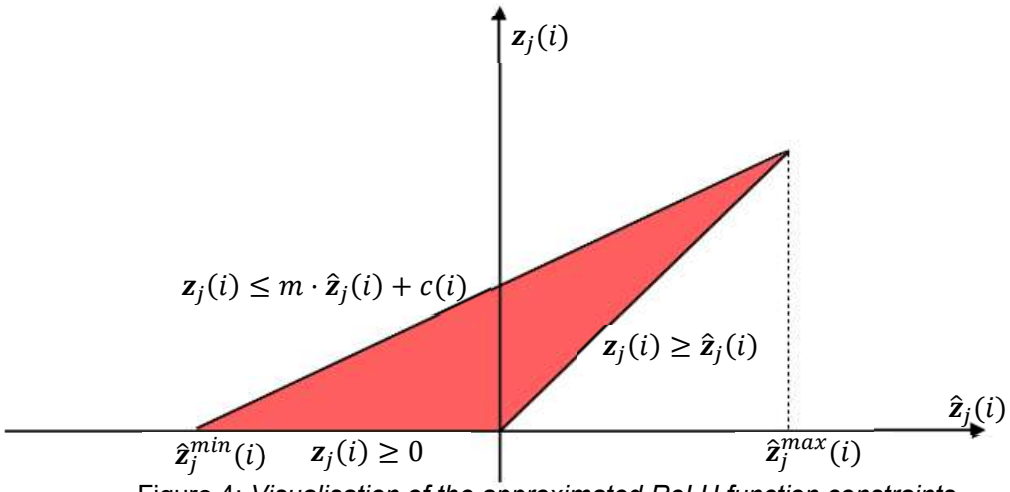

Figure 4: *Visualisation of the approximated ReLU function constraints.*

In subsequent layers we must account for the ReLU, and to do this we firstly need equations for the three lines of our triangle in figure 4: $\boldsymbol{z}_j(i) \geq 0$, $\boldsymbol{z}_j(i) \geq \hat{\boldsymbol{z}}_j(i)$, and $\boldsymbol{z}_j(i) \leq m \cdot \hat{\boldsymbol{z}}_j(i) + c$, where m and c are the gradient and intercept, implemented in code as shown on the right:

```
m = z_hat_max{j-1}./(z_hat_max{j-1} - z_hat_min{j-1});
m = min(max(0,m),1);
M = diag(m(1:layer_size{j-1})); % diagonalised matrix
c = -z_hat_min{j-1}.*m;
c = max(c,0);
c = c(1:layer_size{j-1});
```

Note that though we are in layer j, the ReLU approximation is applied to $\hat{\boldsymbol{z}}_{j-1}$. Also note that, as the triangle cannot be formed if $\hat{\boldsymbol{z}}_{min} \geq 0$ or $\hat{\boldsymbol{z}}_{max} \leq 0$, we constrain $0 \leq m \leq 1$ and $c \geq 0$. Now, using our triangle inequalities, we will find $\boldsymbol{A}$ and $\boldsymbol{b}_{nq}$:

$\boldsymbol{z}_{j-1}(i) \geq 0$ corresponds to $-I \cdot \boldsymbol{z}_{j-1}(i) + 0 \cdot \hat{\boldsymbol{z}}_{j-1} + 0 \cdot \boldsymbol{x} \leq 0$

$\boldsymbol{z}_{j-1}(i) \geq \hat{\boldsymbol{z}}_{j-1}(i)$ corresponds to $-I \cdot \boldsymbol{z}_{j-1} + I \cdot \hat{\boldsymbol{z}}_{j-1} + 0 \cdot \boldsymbol{x} \leq 0$ (3)

$\boldsymbol{z}_{j-1}(i) \leq m \cdot \hat{\boldsymbol{z}}_{j-1}(i) + c$ corresponds to $I \cdot \boldsymbol{z}_j - \boldsymbol{M}_{j-1} \cdot \hat{\boldsymbol{z}}_{j-1} + 0 \cdot \boldsymbol{x} \leq \boldsymbol{c}_{j-1}$, therefore:

$$\overline{\boldsymbol{A}}_{j,centre} = \begin{bmatrix} -\boldsymbol{I} & 0 \\ -\boldsymbol{I} & \boldsymbol{I} \\ \boldsymbol{I} & -\boldsymbol{M}_{j-1} \end{bmatrix} \text{ and } \overline{\boldsymbol{b}}_{nq,j} = \begin{bmatrix} 0 \\ 0 \\ \boldsymbol{c}_{j-1} \end{bmatrix}. \quad (4)$$

This sets up the inequalities, but to form $\boldsymbol{A}$ and $\boldsymbol{b}_{nq}$ for layer j we must concatenate with prior layers, and add zeros either side of $\boldsymbol{A}_{centre}$ to account for $\hat{\boldsymbol{z}}_j$ and $\boldsymbol{x}$ in $\boldsymbol{v}_j$:

$$\boldsymbol{A}_{centre} = \begin{bmatrix} \overline{\boldsymbol{A}}_{j,centre} & & \\ & \ddots & \\ & & \overline{\boldsymbol{A}}_{2,centre} \end{bmatrix}, \text{ then } \boldsymbol{A} = [0 \quad \boldsymbol{A}_{centre} \quad 0], \text{ and } \boldsymbol{b_{nq}} = \begin{bmatrix} \overline{\boldsymbol{b}}_{nq,j} & \cdots & \overline{\boldsymbol{b}}_{nq,2} \end{bmatrix} \quad (5)$$

Now we have all of our `linprog` inputs, we use the following for loop to perform the linear program. `linprog` minimises the function $\boldsymbol{f}^T\boldsymbol{v}$, so to maximise/minimise each element, $\boldsymbol{f}$ is a zero vector with the appropriate element set to -1 for max and +1 for min. Results shown in figure 4.

```
for i = 1:layer_size
    f = zeros(x_size,1);
    f(i) = 1;
    tmp =
linprog(f,A,bnq,Aeq,beq,lb,ub,options);
    z_hat_min(i) = tmp(i);
    f(i) = -1;
    tmp =
linprog(f,A,bnq,Aeq,beq,lb,ub,options);
    z_hat_max(i) = tmp(i);
end
```

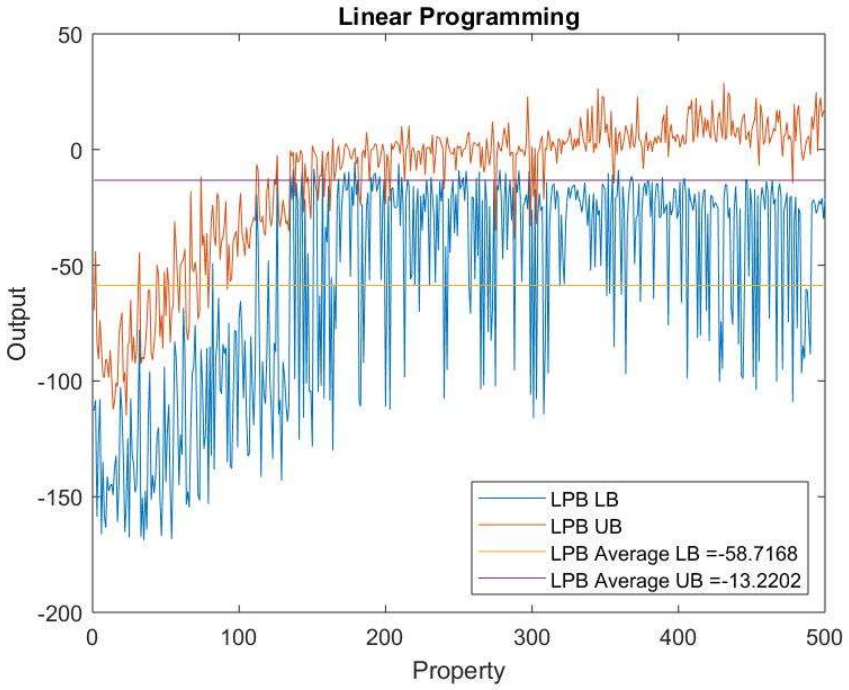

Figure 4: *Upper and lower bounds calculated across all 500 properties, as well as the averages. As expected, these bounds are much tighter than those found by interval bound propagation, and we have been able to prove 260 properties true by showing that a lower bound on their y^* is negative.*

5 Branch and Bound

This is an example of a complete method, as, with enough time, it can guarantee a correct true or false identification of a property. The essence of this method is to continually split the input subdomain with the largest upper bound output, $\bar{\chi}$, into two equal partitions, the output bounds of which will be improved over the original. We can continue creating partitions until we either find a counterexample or prove that none exist. In our first version we will use linear programming to find upper and lower bounds on each partition. Our LPB from the previous section serves as a first pass over the entire input domain. We open a loop that terminates when the `COMPLETE` flag is set to 1 and we have a definitive answer – on each pass we find the domain (index indicated by `idx1`)

with the largest output value (`ymax`), and immediately check if this value is negative, proving that the property is true. If not, we record the input bounds and cut the longest edge (as shown below) to split the input domain in two.

```
[YMIN, YMAX] = . . . linear_programming_bound(W,b,XMIN,XMAX);
while COMPLETE == 0
    % idx is the index of xbar
    [ymax, idx1] = max(YMAX);
    if ymax < 0
        COMPLETE = 1;
        flag = 1; % proven true
        break
    end
    xbarmin = XMIN(idx1,:);
    xbarmax = XMAX(idx1,:);

    S = (xbarmax-xbarmin)./(xmax-xmin); % cutting the longest edge
    [~, idx2] = max(S); % idx2 is index of dimension with longest relative length
```

Once we have generated a new pair of input domains, we can find their output bounds using LPB again, and this time check for positive lower bounds on y^* that will allow us to set the flag and terminate. If the procedure fails to terminate, we simply increase the partition size and return to the top of the loop, being sure to update the subdomain inputs and outputs `XMIN`, `XMAX`, `YMIN` and `YMAX`.

When tested across all properties this program returns a 1×500 vector of Boolean flags that matches the ground truth. We thus have a complete method that guarantees whether every property is true or false. However, due to some properties requiring a large number of iterations, this method is slow, taking about 18 hours to complete. We can speed it up by implementing elements of our previous methods. For property168 our current method returns the correct flag after 20 iterations in 166.8s. Though it provides weaker bounds, IBP is much faster than linear programming, and we may be able to obtain a correct result in less time by using more iterations. To test this we replace all instances of `linear_bound_propagation` with `interval_bound_propagation` in our code and run again, and this time when we run the correct result is given after 25952 iterations, but in only 55.70s. This seems promising, however, when tested with property499, the LPB based method gives the correct flag after 273.6s and 35

iterations, while the IPB method runs for over 10 minutes and 90,000 iterations without terminating. To explore this further, we run our IBP based branch-and-bound on all 500 properties and find that 211 fail to terminate within 10,000 iterations. We can conclude that IBP is much more effective than LPB for some properties and completely ineffective for others, and thus it would seem logical to have our algorithm switch methods after a set number of iterations. We achieve this simply by placing our algorithm within another `while` loop controlled by a flag, `RESTART`, as shown on the right.

```
while RESTART == 1
RESTART = 0;
. . .
  while COMPLETE == 0
    if i == max_iter
      METHOD = 2;
      RESTART = 1;
      break
    end
. . .
```

We can further speed up our algorithm by making use of the stochastic method. As shown in figure 1, this method is very successful in quickly identifying false properties, which also tend to be the most time consuming in LBP based branch-and-bound. It is reasonable to assume that since input subdomain $\bar{\chi}$ has the largest upper bound on y^*, it will also give us the best chance of randomly generating a positive output, so we implement the stochastic method as shown above on the right after finding $\bar{\chi}$ in each iteration. Note that we only apply this method in conjunction with LPB, as the time taken becomes significant over the many thousands of iterations possibly required for IBP based branch-and-bound. It is also applied once before branch-and-bound begins

```
if METHOD == 2
  X_rand = …
    generate_inputs(xbarmax, xbarmin, k, 6);
  y = compute_nn_outputs(W,b,X_rand);
  if max(y) > 0
    flag = 0;
    COMPLETE = 1;
    break
  end
```

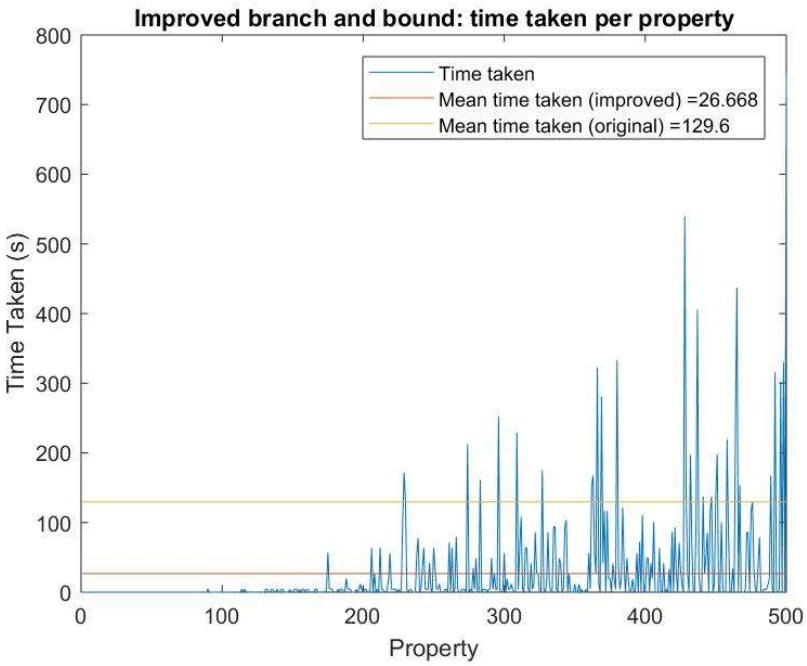

Figure 5: *Time taken for each property to be correctly identified by improved branch-and-bound algorithm.*

in order to eliminate a large number of properties quickly without the need for calling `linprog`. Using k = 10,000 and `max_iter` = 3000, the results of our improved branch-and-bound are shown in figure 5. The total running time has been reduced to 13340s, approximately 3 hours 40 minutes, almost five times faster than our initial algorithm, with all 500 properties being correctly labelled.

6 Conclusion

While it is by no means optimally efficient, we have shown that a branch-and-bound method using linear programming will eventually guarantee correct identification of true and false properties, and we can enormously speed this up by employing RNG based proof by counterexample and using faster bounding methods like interval propagation up to a sensible partition size. Another method we could explore is mixed-integer linear programming (MILP) in place of linear programming, standard form on left[1] below.

$$\begin{array}{ll} \text{maximize} & \mathbf{c}^{\mathrm{T}}\mathbf{x} \\ \text{subject to} & A\mathbf{x} + \mathbf{s} = \mathbf{b}, \\ & \mathbf{s} \geq \mathbf{0}, \\ & \mathbf{x} \geq \mathbf{0}, \\ \text{and} & \mathbf{x} \in \mathbb{Z}^n, \end{array}$$

MILP standard form. Note $\boldsymbol{r}_k$ is a binary vector of flags indicating if the ReLU is active, and N_k is the number of elements

$$\mathbf{z}_k = \max(\hat{\mathbf{z}}_k, 0) \Rightarrow \begin{cases} \mathbf{z}_k \leq \hat{\mathbf{z}}_k - \hat{\mathbf{z}}_k^{\min}(1 - \mathbf{r}_k) \\ \mathbf{z}_k \geq \hat{\mathbf{z}}_k \\ \mathbf{z}_k \leq \hat{\mathbf{z}}_k^{\max}\mathbf{r}_k \\ \mathbf{z}_k \geq 0 \\ \mathbf{r}_k \in \{0,1\}^{N_k} \end{cases}$$

ReLU approximation in MILP. Note that $\boldsymbol{s}$ represents slack variables to remove inequalities

MILP is an optimisation program in which some of the variables are constrained as integers. Similar to what we explored in section 4.3, this requires reformulation of the ReLU activation function, as shown on the right above[2]. MILP is very difficult to solve as it is NP-complete (essentially deterministically solvable by branching in polynomial time), and likely overkill for our simple dataset, but for more complex verification we would use MILP with a commercial solver such as Gurobi.

References

[1]: Papadimitriou, C. H.; Steiglitz, K. (1998). *Combinatorial optimization: algorithms and complexity*. Mineola, NY: Dover. ISBN 0486402584.

[2]: Andreas Venzke, Spyros Chatzivasileiadis (2019). Verification of Neural Network Behaviour: Formal Guarantees for Power System Applications. arXiv:1910.01624