Honour School of Engineering Science
B1: Computational Mathematics
Project B

Formal Verification of Neural Networks

- Analysing the Robustness of Multi-Layer Perceptron -

Candidate Number: 1025374

January, 2020

1 Introduction

Advances in deep learning have led to a wide implementation of neural networks in addressing complex real-world problems such as speech recognition and image classification. In recent years, however, it has been observed that DNNs can react in unexpected and incorrect ways to even the slightest perturbation of their inputs unnoticeable to the human eye. In the context of safety-critical applications such as autonomous cars, adversarial attacks can be targeted on vehicles where stickers and paint are used to generate an adversarial stop sign that the vehicles can misinterpret as another sign.

In the wake of such dangers, a greater emphasis has been placed on developing a means to formally verify the correctness of the neural systems. In this dissertation, I will explore different fundamental ways of addressing the formal verification problem. Specifically, the main contributions of this dissertation are:

- Design of the Branch and Bound Framework
- Analysis of the Unsound method, Interval Bound Propagation Method, Linear Programming Method under the Branch and Bound Framework
- Evaluation and further research on improvements of each method

2 Problem Formulation

2.1 Specification of the Neural Network

Consider a piecewise linear multi-layer perceptron with n layers which represents a certain function f. An input vector $\mathbf{x}$ is specified by a bounded input domain $\mathcal{X} = \{\mathbf{x} \mid \mathbf{x}^{min} \leq \mathbf{x} \leq \mathbf{x}^{max}\} \subseteq \mathbb{R}^{k_0}$ where k_0 represents the input dimension. An output vector $\mathbf{y} = f(\mathbf{x})$ is specified by an output domain $\mathcal{Y} \subseteq \mathbb{R}^{k_n}$ where k_n represents the output dimension.

To consider the hidden layers of the neural network, assume that each layer in $f : \mathbb{R}^{k_0} \to \mathbb{R}^{k_n}$ corresponds to a function $f_i : \mathbb{R}^{k_{i-1}} \to \mathbb{R}^{k_i}$ where k_i represents the dimension of the hidden variable $\mathbf{z_i}$ at the i^{th} layer:

$$f(\mathbf{x}) = f_n \circ f_{n-1} \circ ... \circ f_1(\mathbf{x})$$

The function at the i^{th} layer is:

$$\mathbf{z}_i = f_i(\mathbf{z}_{i-1}) = \sigma_i(\hat{\mathbf{z}}_i) \tag{1}$$

$$\hat{\mathbf{z}}_i = \mathbf{W}_i\mathbf{z}_{i-1} + \mathbf{b}_i \tag{2}$$

which consists of a linear transformation defined by a weight matrix $\mathbf{W_i} \in \mathbb{R}^{k_{i-1} \times k_i}$ and a bias matrix $\mathbf{b_i} \in \mathbb{R}^{k_i}$, and the ReLU activation function $\sigma_i : \mathbb{R}^{k_i} \to \mathbb{R}^{k_i}$.

Note that the final n^{th} layer does not have an activation function, giving the final output $\mathbf{z}_n = \hat{\mathbf{z}}_n$.

2.2 Problem Statement

Setting $\mathbf{x} = \mathbf{z}_0$ and $\mathbf{y} = \hat{\mathbf{z}}_n$, and letting $P(\hat{\mathbf{z}}_n)$ to denote the property of the neural network, the full problem statement of formal verification of neural networks is to prove:

$$\mathbf{z}_0 \in \mathcal{X}, \quad \hat{\mathbf{z}}_n = f(\mathbf{z}_0) \quad \Longrightarrow \quad P(\hat{\mathbf{z}}_n) \tag{3}$$

The property $P(\hat{\mathbf{z}}_n)$ is one that describes the property of robustness to adversarial examples. To verify the robustness of the neural network, all samples in the neighbourhood of a given input $\mathbf{z}_0$ all need to be classified with the same label. Supposing that the desired label is $i^* \in \{1, 2, ..., k_n\}$ and that ϵ is the maximum allowable disturbance in the input space, the input constraints, the output constraints and the property constraints for a true property are given by:

$$\mathcal{X} = \{\mathbf{z}_0 \mid \| \mathbf{z}_0 - \mathbf{k} \|_p < \epsilon\} \tag{4}$$

$$\mathcal{Y} = \{\hat{\mathbf{z}}_n \mid \hat{\mathbf{z}}_n = f(\mathbf{z}_0)\} \tag{5}$$

$$P(\hat{\mathbf{z}}_n) = \{\hat{\mathbf{z}}_{n_{[i^*]}} > \hat{\mathbf{z}}_{n_{[j]}} \ \forall \ i^* \neq j\} \tag{6}$$

A neural network with function f is said to be **robust at** $\mathbf{z}_0$ if for all $\mathbf{q} = \mathbf{z}_0 - \mathbf{k}$ in the input domain $\mathcal{X}$, the weight of the correct label $\hat{\mathbf{z}}_{n_{[i^*]}}$ exceeds that of the undesired label $\hat{\mathbf{z}}_{n_{[j]}}$. By contrast, a neural network is said to be not robust at $\mathbf{z}_0$ if the weight of the undesired label $\hat{\mathbf{z}}_{n_{[j]}}$ is greater than or equal to that of the correct label $\hat{\mathbf{z}}_{n_{[i^*]}}$ for any $\mathbf{q}$. In this case, the vector $\mathbf{q}$, which is at a minimal distance $\mathbf{k}$ away from $\mathbf{z}_0$, is described as an **adversarial example** to the neural network.

The problem statement is either i) to prove that the property of the neural network is true for all values of the vector $\mathbf{q}$ or ii) to show that the property of the neural network is false by generating a counter-example that fails to satisfy equations (4) and (6).

2.3 Canonical Form of the Problem Statement

If the property of the neural network can be expressed as a boolean expression over linear forms, the neural network can be altered in a way that the process of proving the property is simplified to checking the sign of the output of the modified neural network $f'(\mathbf{z}_0)$. As illustrated in [2], if the property is given by $P(\hat{\mathbf{z}}_n) \triangleq \mathbf{c}^T \hat{\mathbf{z}}_n \leq b$, a layer with weight of $\mathbf{c}$ and a bias of $-b$ can be added to the final layer of the neural network to give an adjusted single output $\mathbf{c}^T \hat{\mathbf{z}}_n - b$.

With the addition of the final layer, the canonical form of the problem statement becomes either i) to prove that the property of the neural network is true by showing that the output $f'(\mathbf{z_0})$ is negative for all $\mathbf{z_0} \in \mathcal{X}$ or ii) to show that the property of the neural network is false by showing that there exists at least one $\mathbf{z_0} \in \mathcal{X}$ for which the corresponding output $f'(\mathbf{z_0})$ is positive.

3 Branch and Bound Framework

The **piecewise linear** property of the given neural network means that $f(\mathbf{z}_0)$ is a function composed of linear segments defined over the domain $\mathcal{X}$. That is, the input domain $\mathcal{X}$ can be divided into n collective exhaustive subsets { $\mathcal{X}_1, \mathcal{X}_2, ..., \mathcal{X}_n$ } with a linear function defined over each subset. This is an important property of the neural network key to forming the branch and bound framework.

The general structure of the branch and bound method is shown by Algorithm 1. The branch strategy and bound strategy are defined by the functions $compute\_BCs$ and $compute\_Bounds$ each respectively. Given the input domain $\mathcal{X}$, the bounding strategy calculates the corresponding lower bound $\hat{l}$ and upper bound $\hat{u}$ of the output $\mathbf{z}_n$. There are 3 possible cases that arise from the bound strategy:

Case 1: $\hat{l} > 0$ $f'(\mathbf{z}_0)$ is positive for at least one $\mathbf{z}_0 \in \mathcal{X}$ and hence the property is false

Case 2: $\hat{u} < 0$ $f'(\mathbf{z}_0)$ is negative for all $\mathbf{z}_0 \in \mathcal{X}$ and hence the property is true

Case 3: $\hat{l} < 0$ and $\hat{u} > 0$ Unable to determine the property of the neural network.

Algorithm 1: Branch and Bound Framework

```
function Branch_And_Bound(W, b, x^min, x^max)
  [x̄_1, x̄_2] ← compute_BCs(x^min, x^max)
  BCs ← store_BCs(x̄_1, x̄_2)
  [LBs, UBs] ← compute_Bounds(W, b, BCs)
  for i = 1 ... L do
    if any(LBs > 0) > 0 then
      flag = 0
      return flag
    end if
    if all(UBs < 0) > 0 then
      [maxUB, index] ← max(UB)
      if maxUB < 0 then
        flag = 1
        return flag
      else
        [x̄^min, x̄^max] ← load_BCs(BCs, index)
        [x̄_1, x̄_2] ← compute_BCs(x̄^min, x̄^max)
        BCs ← append_BCs(BCs, index, x̄_1, x̄_2)
        [LBs, UBs] ← compute_Bounds(W, b, BCs)
      end if
    end if
    i = i + 1
  end for
  return flag
end function
```

For Case 3, an enhanced estimation of $\hat{l}$ and $\hat{u}$ are required in order to prove the property of the neural network. Using the piecewise linear property, the branch strategy bisections the input sub-domain $\bar{\mathcal{X}}$ that generates the greatest upper bound into sub-domains $\bar{\mathcal{X}}_1$ and $\bar{\mathcal{X}}_2$. As demonstrated by Algorithm 1, the bisection of the input sub-domain with the greatest upper bound is iterated continually to create tighter bounds of the output $\hat{\mathbf{z}}_n$ until the property of the neural network is shown to be either true or false. In the following sections, different ways of implementing branch strategy and bound strategy will be discussed. That most implementations of the bound strategy are incomplete methods that fail to verify all properties of the neural network means that the general branch and bound framework is essential to the verification problem. I will also explore efficient ways of partitioning the input domain into sub-domains in search for the optimal bounds on $\mathbf{z}_n$.

4 Branch Strategy

4.1 Splitting the Longest Edge

One common way is to implement the function $compute\_BCs$ is to split the longest edge of the input domain. Assuming the piecewise linear property, the function partitions the input domain into sub-domains by computing $\mathbf{s}(i)$ and j.

$$\mathbf{s}(i) = \frac{\bar{\mathbf{x}}^{max}(i) - \bar{\mathbf{x}}^{min}(i)}{\mathbf{x}^{max}(i) - \mathbf{x}^{min}(i)} \quad (7) \qquad j = \underset{i}{\operatorname{argmax}}(\mathbf{s}(i)) \quad (8)$$

where $\mathbf{x}^{min}$ and $\mathbf{x}^{max}$ are the box constraints of the input domain $\mathcal{X}$ and $\bar{\mathbf{x}}^{min}$ and $\bar{\mathbf{x}}^{max}$ are the box constraints of the input sub-domain $\bar{\mathcal{X}}$ which generates the highest upper bound.

To split the longest edge, the function $compute\_BCs$ searches for the i^{th} element of $\bar{\mathcal{X}}$ with the longest relative length $\mathbf{s}(i)$ and partitions $\bar{\mathcal{X}}$ into further equally sized sub-domains $\bar{\mathcal{X}}_1$ and $\bar{\mathcal{X}}_2$ by splitting $\bar{\mathbf{x}}(i)$ into two equally sized box constraints as shown by equations (9) and (10) in the project brief. That $\bar{\mathcal{X}}_1$ and $\bar{\mathcal{X}}_2$ are subsets of $\bar{\mathcal{X}}$ means that the each $\bar{\mathcal{X}}_1$ and $\bar{\mathcal{X}}_2$ generates upper bound less than or equal to that of $\bar{\mathcal{X}}$ and lower bound greater than or equal to that of $\bar{\mathcal{X}}$.

4.2 Gradient Interval Method

Another branch strategy is to split the input domain according to computed value of **gradient interval**. For this particular strategy, the gradient or the Jacobian matrix of the output $\hat{\mathbf{z}}_n$ with respect to each element of the input is computed using a different implementation of the function $compute\_BCs$. The notion behind this strategy is that the element of the input which gives the greatest value of gradient interval most influences the output.

Algorithm 2: Gradient Interval

```
function Gradient_Interval(W, x^min, x^max, R)
    g^max = g^min ← W{Final_Layer}
    g = [ g^min, g^max]
    for i = Number_Of_Layers − 1 ... 1 do
        for j = 1 ... Layer_Size do
            g = R[i, j] × g
        end for
        g = W{i} · g
    end for
    return g
end function
```

In theory, gradient interval can be calculated by differentiating $\hat{\mathbf{z}}_n$ with respect to each interval given by $\mathbf{x}^{min}(j)$ and $\mathbf{x}^{max}(j)$. The problem with this calculation arises due to the presence of an activation function. As shown in equation (1), ReLU activation function σ_i takes effect at the end of every layer except the final layer to give $\mathbf{z}_i$ and thus its gradient has to be taken into account. Consequently, the function Gradient_Interval has $\mathbf{R}$ as an input to take into account the gradient of ReLU activation function. $\mathbf{R}$ is a matrix of row size $Number\_of\_Layers$ and column size $Layer\_Size$ with each element $[i, j]$ containing the corresponding value of the gradient of ReLU activation function.

$\mathbf{R}$ can be calculated by using forward propagation, for instance, by having another step in addition to equation (6) of the project brief [1] as following: For the case where the output of the activation $\mathbf{z}_i$ is equal to $\hat{\mathbf{z}}_i$, the gradient is equal to 1. For the case where the output of the activation function $\mathbf{z}_i$ is zero, the gradient is also equal to zero. The values of the gradient of the activation function, each corresponding to lower and upper bounds, are stored in $\mathbf{R}[i, j]$.

The computation of gradient interval by using backward propagation is demonstrated by Algorithm 2. The function $Gradient\_Interval$ initialises $\mathbf{g}^{max}$ and $\mathbf{g}^{min}$ to be the values of the weight matrix at the final layer. For every j iteration, element-wise product between $\mathbf{R}$ and $\mathbf{g}$ is calculated. Then, for every i iteration, dot product between $\mathbf{W}\{i\}$ and $\mathbf{g}$ is performed to obtain the gradient interval.

$$\mathbf{s}(i) = \mathbf{g}^{max}(i) * (\mathbf{x}^{max}(i) - \mathbf{x}^{min}(i)) \tag{9}$$

The function $compute\_BCs$ searches for the i^{th} element of $\bar{\mathcal{X}}$ with the greatest gradient $\mathbf{s}(i)$ by using equations (9) and (8). Similar to the previous method, the function then partitions the $\bar{\mathcal{X}}$ into further equally sized sub-domains $\bar{\mathcal{X}}_1$ and $\bar{\mathcal{X}}_2$ according to equations (9) and (10) in the project brief.

5 Bound Strategy

The function $compute\_Bounds$ generates the lower and upper bounds corresponding to the partitioned set of input sub-domains computed by the branch strategy. In the subsequent sections, various methods of calculating lower and upper bounds of $\hat{\mathbf{z}}_\mathbf{n}$ are analysed and tested against the given dataset. The dataset consists of 500 MAT files each consisting of 6 layers. Each MAT file contains the following variables: i) $\mathbf{x}^{min}$ and $\mathbf{x}^{max}$: the box constraints of the input domain ii) $\mathbf{W}$: a cell array of weight matrix for each layer iii) $\mathbf{b}$: a cell array of bias matrix for each layer. Please refer to the project brief [1].

5.1 Unsound Method

The unsound method consists of functions $Generate\_Inputs$ and $Compute\_NN\_Outputs$. The former generates k number of random inputs within the box constraints set by $\mathbf{x}^{min}$ and $\mathbf{x}^{max}$, all of which are taken as inputs to the function. The equations (10) and (11) show that a uniformly distributed random number generator is used to create each of the random input and all the k random inputs generated are stored in an array $\mathbf{X}$. As equivalent to Algorithm 1 of the project brief, the function $Compute\_NN\_Outputs$ takes $\mathbf{X}$ as an input to calculate k corresponding outputs for each input. The maximum value among k outputs is a valid lower bound on $\hat{\mathbf{z}}_n$.

$$\mathbf{X}(i) = \mathbf{x}^{min}(i) + \mathbf{c}(i) * (\mathbf{x}^{max}(i) - \mathbf{x}^{min}(i)) \tag{10}$$

$$\mathbf{c}(i) = rand(0, 1) \tag{11}$$

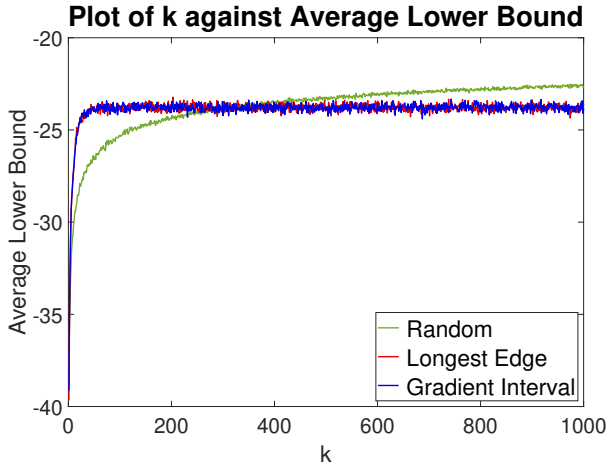

Figure 1: Average Lower Bound

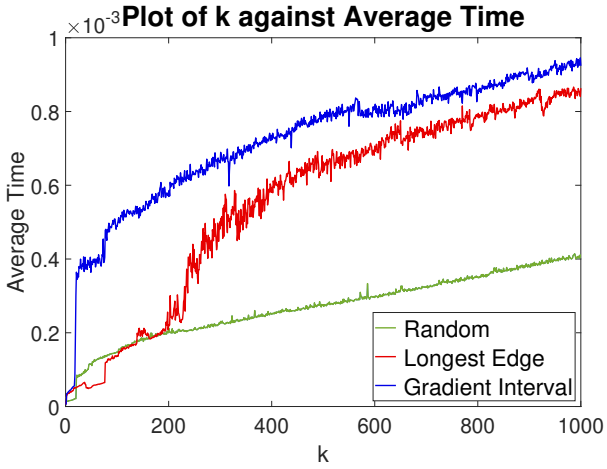

Figure 2: Average Time

Figure 1, the value of k ranges from 1 to 1000. For its every value, k number of inputs and the corresponding outputs are generated for all 500 properties and for each property, the maximum of the k number of outputs is taken as the lower bound. The average lower bound is calculated by taking the average of the lower bounds over all 500 properties for the given value of k. As the value of k increases, the average lower bound also increases as expected. This is because taking a greater number of inputs that satisfy the box constraints leads to a greater probability of finding $\mathbf{z}_0$ that generates a stricter lower bound.

However, simply creating a set of random inputs that satisfy the box constraints given by $\mathbf{x}^{min}$ and $\mathbf{x}^{max}$ is an inefficient process of finding such $\mathbf{z}_0$. Using the ideas from the branch strategy, creating a set of random inputs that satisfy the new box constraints $\bar{\mathbf{x}}^{min}$ and $\bar{\mathbf{x}}^{max}$ set by splitting the input domain with the longest edge or the greatest gradient interval is a more efficient process. As demonstrated by Figures 1 and 3, for small values of k in the range 1 to 200, the gradient interval and splitting along the longest edge methods are better able to find stricter lower bound and are thus better able to find greater number of counter-examples. Referring to Figure 2, on the other hand, the average computation time for the original method that only uses $\bar{\mathbf{x}}^{min}$ and $\bar{\mathbf{x}}^{max}$ require far less computational time than the other two methods require. Figure 2 verifies that using the original method with value of k equal to 500 is able to generate lower bound as accurate as the lower bounds generated more efficiently.

Figure 3, however, reveals a setback of the unsound method. The maximum number of counter-examples generated across all the methods used is 78. Given that 172 of the 500 properties provided are false, the unsound method does not guarantee sufficient number of counter-examples. In fact, without splitting the input domain, we may even fail to generate counter-examples for the 78 false properties found, as we are taking random inputs from the original input domain $\mathcal{X}$. Moreover, the unsound method does not provide a valid upper bound and therefore cannot provide any guarantee for a true property.

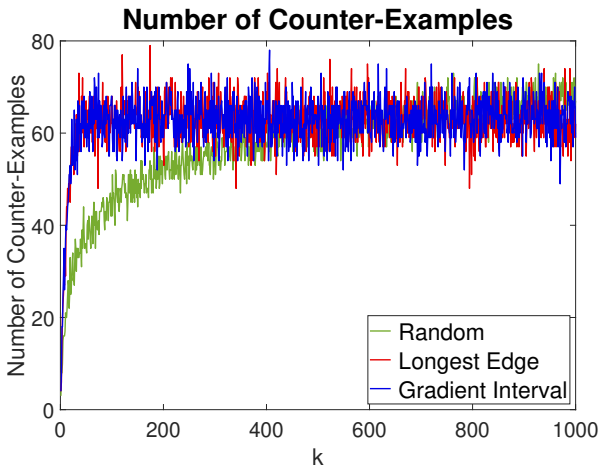

Figure 3: Number of False Properties Proved

On the other hand, the unsound method is indeed a computationally efficient means to generate strict lower bounds. When used in combination with other methods that upper bounds in the branch and bound framework, the unsound method proves to be an effective and efficicient way of finding counter-examples to disprove the property of the neural network. For any further usage of the unsound method, the value of 500 will be used as k, as previously justified.

5.2 Interval Bound Propagation Method

In the previous subsection, we have seen that using concrete values of $\mathbf{z}_0$ has a disadvantage of generating only a single bound. In this subsection, interval arithmetic is explored as a means to generate lower and upper bounds of the given neural network.

5.2.1 Naive Interval Bound Propagation

One way of calculating the lower and upper bound of the output is by setting input features as intervals and following the same arithmetic performed in the neural network. Referring to the small network in Figure 1 of the project brief, when the identical arithmetic is applied on the interval [-2,2] on each elements of the input vector, the output interval [-38,51] is obtained. However, this approach leads to a loose estimation of lower and upper bound and it is evident that there is no input that generates output value of 51. The loose estimation results from a **dependency problem**, which refers to overestimation of the resulting lower and upper bound arising from the ignorance of the input dependencies during interval propagation.

5.2.2 Symbolic Interval Bound Propagation

Symbolic Interval Bound Propagation attempts to minimise such errors arising from the dependency problem by maintaining symbolic equations during intermediate computations as shown by Algorithm 3. Assume that the upper and lower bounds of the previous layer have been computed to give an interval $\left[\, \mathbf{z}_{l-1}^{min},\ \mathbf{z}_{l-1}^{max} \,\right]$ and let $\mathbf{W}_l^+$ and $\mathbf{W}_l^-$ each respectively denote all the positive elements and all the negative elements of the $\mathbf{W}_l$ matrix. The multiplication between the interval and the positive weight parameters results in the output $\left[\, \mathbf{w}_l^+ \mathbf{z}_{l-1}^{min},\ \mathbf{w}_l^+ \mathbf{z}_{l-1}^{max} \,\right]$. Taking the sign into consideration, the multiplication between the interval and the negative weight parameters results in the output $\left[\, \mathbf{w}_l^- \mathbf{z}_{l-1}^{max},\ \mathbf{w}_l^- \mathbf{z}_{l-1}^{min} \,\right]$. Summing up the outputs from the two multiplications and adding the bias matrix $\mathbf{b}_l$ gives lines 4 and 5 of Algorithm 3.

Algorithm 3: Interval Bound Propagation

function $Interval\_Propagation(\mathbf{W}, \mathbf{b}, \mathbf{x}^{min}, \mathbf{x}^{max})$
  $[\, \mathbf{z}_0^{min},\ \mathbf{z}_0^{max}] \leftarrow [\, \mathbf{x}^{min},\ \mathbf{x}^{max}]$
  for $l = 1 \ldots Number\_Of\_Layers - 1$ **do**
    $\hat{\mathbf{z}}_l^{max} \leftarrow \mathbf{W}_l^{+}\mathbf{z}_{l-1}^{max} + \mathbf{W}_l^{-}\mathbf{z}_{l-1}^{min} + \mathbf{b}_l$
    $\hat{\mathbf{z}}_l^{min} \leftarrow \mathbf{W}_l^{+}\mathbf{z}_{l-1}^{min} + \mathbf{W}_l^{-}\mathbf{z}_{l-1}^{max} + \mathbf{b}_l$
    $[\, \mathbf{z}_l^{min},\ \mathbf{z}_l^{max}\,] \leftarrow \sigma(\hat{\mathbf{z}}_l^{min},\ \hat{\mathbf{z}}_l^{max})$
  end for
  $\hat{\mathbf{z}}_l^{max} \leftarrow \mathbf{W}_l^{+}\mathbf{z}_{l-1}^{max} + \mathbf{W}_l^{-}\mathbf{z}_{l-1}^{min} + \mathbf{b}_l$
  $\hat{\mathbf{z}}_l^{min} \leftarrow \mathbf{W}_l^{+}\mathbf{z}_{l-1}^{min} + \mathbf{W}_l^{-}\mathbf{z}_{l-1}^{max} + \mathbf{b}_l$
  return $\hat{\mathbf{z}}_l^{min}, \hat{\mathbf{z}}_l^{max}$
end function

The function $Interval\_Propagation$ computes the average lower and upper bounds to be -137.88 and 82.73. The lower and upper bounds are vastly over-estimated, however, to an extent where only 11 properties are proved to be true and none of the properties are shown to be false. We thus require an enhanced estimate of the upper bound.

5.3 Linear Programming Method

The problem of computing upper bound of $\hat{\mathbf{z}}_n$ can be formulated as a maximisation problem demonstrated by equation (7) of the project brief. The problem of computing lower bound $\hat{\mathbf{z}}_n$ is a minimisation problem with the same set of constraints given by the equation. The computational difficulty of the optimisation problems lies on the non-linearity of the final sets of constraints. The presence of the ReLU function essentially converts the linear problem into a computationally demanding NP-hard problem. A possible way of resolving the issue is to approximate non-linear constraints with linear constraints.

5.3.1 Relaxation of the ReLU function

The figures below show the various approximations of the non-linear ReLU function using linear constraints.

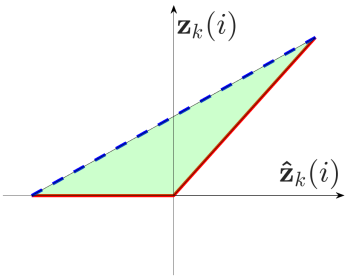

Figure 4: Planet Relaxation

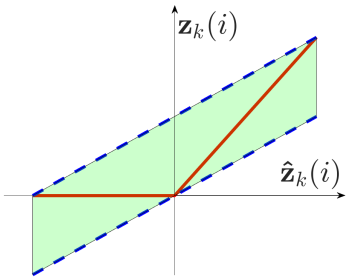

Figure 5: Parallel Relaxation

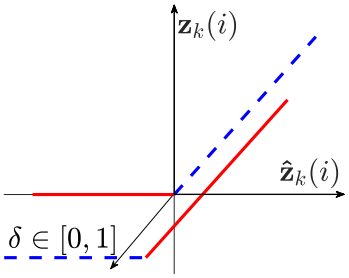

Figure 6: Mixed Integer Encoding

$$\begin{aligned} &\mathbf{z}_k^{min}(i) \equiv\ min\ \mathbf{z}_k(i) \\ &\text{s.t.}\ \ \mathbf{z}_k(i) > 0 \\ &\qquad \mathbf{z}_k(i) > \hat{\mathbf{z}}_k(i) \\ &\qquad \mathbf{z}_k(i) < m(\hat{\mathbf{z}}_k(i) - \mathbf{z}_k^{min}(i)) \end{aligned} \tag{12}$$

$$\begin{aligned} &\mathbf{z}_k^{min}(i) \equiv\ min\ \mathbf{z}_k(i) \\ &\text{s.t.}\ \ \mathbf{z}_k(i) < m(\hat{\mathbf{z}}_k(i) - \hat{\mathbf{z}}_k^{min}(i)) \\ &\qquad \mathbf{z}_k(i) > m(\hat{\mathbf{z}}_k(i)) \end{aligned} \tag{13}$$

$$\begin{aligned} &\mathbf{z}_k^{min}(i) \equiv\ min\ \mathbf{z}_k(i) \\ &\text{s.t.}\ \ \mathbf{z}_k(i) \geq 0 \\ &\qquad \mathbf{z}_k(i) > \mathbf{z}_k(i) \\ &\qquad \mathbf{z}_k(i) \leq \hat{\mathbf{z}}_k^{max}(i)) \cdot \boldsymbol{\delta}(i) \\ &\qquad \mathbf{z}_k(i) \leq \hat{\mathbf{z}}_k(i) - \hat{\mathbf{z}}_k^{min}(i) \cdot (1 - \boldsymbol{\delta}(i)) \end{aligned} \tag{14}$$

where $m = -\dfrac{\hat{\mathbf{z}}_k^{max}}{\hat{\mathbf{z}}_k^{max} - \hat{\mathbf{z}}_k^{min}}$, $\boldsymbol{\delta}(i) \in [0, 1]$ s.t. $\sum_{i=1}^{u} \boldsymbol{\delta}(i) = 1$

Planet relaxation uses the three linear constraints formed by equations of lines passing through the points ($\hat{\mathbf{z}}_k^{max}$, $\hat{\mathbf{z}}_k^{max}$), $(0,0)$ and ($\hat{\mathbf{z}}_k^{min}$,0). Parallel relaxation uses the two linear constraints form by the equation of the line passing through the points ($\hat{\mathbf{z}}_k^{max}$, $\hat{\mathbf{z}}_k^{max}$) and ($\hat{\mathbf{z}}_k^{min}$,0), and the equation of the vertical shift of the line by $-m\hat{\mathbf{z}}_k^{min}$. With reference to [3], the ReLU activation function can be approximated by formulating a maximum function in a mixed integer linear programming model and introducing a variable $\boldsymbol{\delta}$. With exactly one of the u elements having the value of 1, $\boldsymbol{\delta}$ indicates whether the given set of equations (14) are binding or non-binding. To be more specific, $\boldsymbol{\delta}$ value of 1 denotes that the second and the fourth equations are binding, giving $\mathbf{z}_k(i) = \hat{\mathbf{z}}_k(i)$ and $\boldsymbol{\delta}$ value of 0 means that none of the equations are binding.

5.3.2 Primal and Dual

The duality principle refers to the fact that every optimisation problem can be viewed as a primal problem and the corresponding dual problem. With reference to [3], the duality relationship can be formed as:

$$\begin{aligned} &minimise\ \mathbf{c}^T\mathbf{x} \\ &\text{s.t.} \quad \mathbf{A}\mathbf{x} \leq \mathbf{b} \\ &\qquad \mathbf{x} \geq 0 \end{aligned} \tag{15}$$

$$\begin{aligned} &maximise\ -\mathbf{b}^T\mathbf{v} \\ &\text{s.t.} \quad \mathbf{A}^T\mathbf{v} + c \geq 0 \\ &\qquad \mathbf{v} \geq 0 \end{aligned} \tag{16}$$

The advantage of the duality principle is that the same optimisation problem can be solved either as a primal problem or a dual problem. In the case of parallel relaxation, the set of primal constraints formed in equations (14) can be translated into simpler set of dual constraints. That is, the set of constraints given by $m(\hat{\mathbf{z}}_k(i)) < \mathbf{z}_k(i) < m(\hat{\mathbf{z}}_k(i) - \hat{\mathbf{z}}_k^{min}(i))$ simply becomes a single linear constraint given by $\hat{\mathbf{v}}_j(i)\mathbf{z}_k(i) \leq -m\hat{\mathbf{v}}_j(i)\hat{\mathbf{z}}_k(i)$. The single linear constraint of the parallel relaxation of the ReLU function provides a computationally efficient means to estimate lower and upper bounds using linear programming.

5.3.3 Remarks on Linear Programming Method

In this dissertation, the planet relaxation has been chosen as a means to approximate the non-linear ReLU activation function. Combining the constraints given by the equation (7) of the project brief with the set of constraints given by equation (12) of this research, all the constraints required to solve the optimisation problem are found. To formulate the optimisation problem using the in-built $linprog$ function in matlab, following matrices have been defined in order to calculate the lower and upper bounds of $\mathbf{z}_n$: $\mathbf{f}$ was used to define the specific element $\hat{\mathbf{z}}_k$ from $\mathbf{Z}$, the combination of all the optimising variables; $\mathbf{A}$ was the equality matrix formed by rearranging the equation (7) and combining the matrix with identity matrices to elicit relevant variables. $\mathbf{A}_{eq}$ was defined from the planet approximation that we have set out in the form of equation (12). The result of $linprog$ was as follows:

6 Results

The figures below show all the results obtained from the lower and upper bounds using all the bounding strategy. The combination of the lower bound generated by the unsound method and the upper bound generated by the linear programming method resulted in a desirable result.

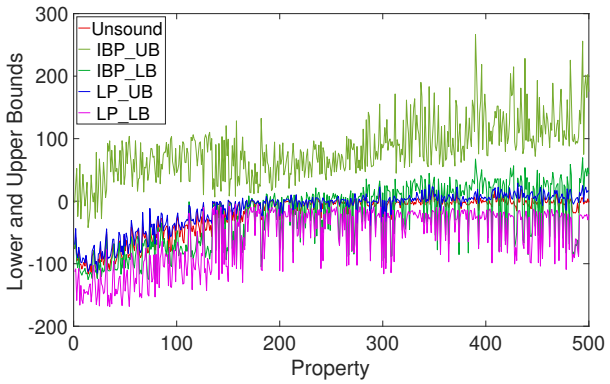

Figure 7: Lower and Upper Bounds

LB Method	UB Method	% of True Proved	% of False Proved	Accuracy	Time Taken(s)
Unsound	LP	100	100	100	4512
LP	LP	71.23	61.19	66.21	18108
IBP	IBP	67.99	30.83	55.22	46603

Figure 8: Accuracy and Time Taken over BAB

7 Conclusion

In conclusion, the bounding strategy alone did not prove sufficient number of properties in order to verify the neural network. But using the branch and bound framework to combine bound strategy with branching strategy, we were able to see a significant increase in accuracy. In the case of the combination of linear programming bound and the unsound method, an increase from 77.4% from the simple bounds calculation to 100% verification using the branching and bound framework was particularly notable.

8 References

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

[2] 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*.

[3] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochendefer. Reluplex: An efficient SMT solver for verifying deep neural networks. In *International Conference on Computer-Aided Verification, 2017*

[4] S. Wang, K. Pei, W. Justin, J. Yang, and S. Jana. Formal security analysis of neural networks using symbolic intervals. *27th USENIX Security Symposium, 2018*.