B1 Engineering Computation: Project B (2019-20)

Neural Network Verification

Course webpage: `http://mpawankumar.info/teaching/b1.html`

1 Introduction

Neural networks are routinely used in various applications of artificial intelligence. Some well-known examples include object detection (that is, automatically locating instances of object categories such as people or cars in a given image), machine translation (that is, automatically translating a text in one language into another), protein folding (that is, automatically estimating the 3D shape of a protein from its genetic sequence) and autonomous agents for highly complex games such as Go and Starcraft. Despite their success, neural networks have been shown to be highly susceptible to adversarial attacks: small changes (almost imperceptible to humans) in the input that drastically change the output of the network. This prevents the use of neural networks in safety critical domains such as autonomous navigation and personalised medicine, where erroneous outputs come with high risk.

In order to overcome the aforementioned deficiency of neural networks, researchers have recently started to focus on *neural network verification*, that is, formally proving that a given neural network satisfies a desirable property or disproving it by generating a counter-example. This B1 project is a taster for the types of methods that are being designed in the machine learning and optimisation communities. The general problem of neural network verification is highly complex. In order to fit the timeline of the B1 project, we will study a simplified version described below.

Notation. Throughout this write-up, we will denote scalars using non-bold lower-case or upper-case letters, e.g. y or L. Vectors will be denoted using bold lower-case letters, e.g. $\mathbf{x}$. The i-th element of a vector $\mathbf{x}$ will be denoted by $\mathbf{x}(i)$. Matrices will be denoted using bold upper-case letters, e.g. $\mathbf{W}$. The (i, j)-th element of a matrix $\mathbf{W}$ will be denoted by $\mathbf{W}(i, j)$. The size of the various vectors and matrices should be clear from context. Sets will be denoted using calligraphic letters, e.g. $\mathcal{X}$.

Multi Layer Perceptron (MLP). We will focus on a special type of neural network called a multi-layer perceptron (MLP). An example MLP is shown in figure 1. This visualisation is a helpful way of understanding how an MLP generates an output for a given input. Each rectangular box denotes a

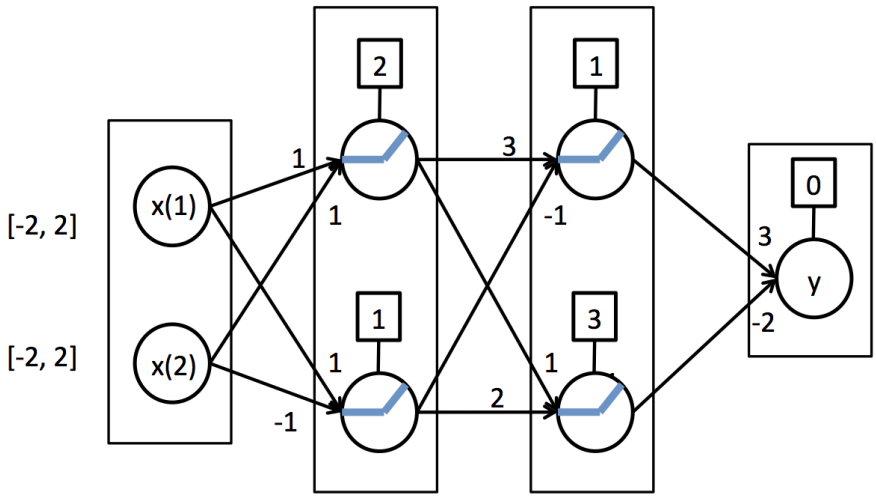

Figure 1: *An example multi-layer perceptron (MLP) with four layers, each shown as a rectangular box. Each layer consists of a set of neurons, which are depicted by circles. The leftmost layer (layer 0) denotes the input, which in this case is a vector of size 2. Each of the two elements of the input vector* $\mathbf{x}$ *can take any value from the interval* $[-2, 2]$*. The second layer (layer 1, which is the first hidden layer) consists of two neurons. Its output is computed in two stages. First, it applies a linear transformation as indicated by the weights (shown along the edges connecting the neurons of layer 0 to the neurons of layer 1) and the biases (shown as a square on top of the neurons of layer 1). Specifically, it computes* $\hat{\mathbf{z}}_1(1) = \mathbf{x}(1) + \mathbf{x}(2) + 2$ *and* $\hat{\mathbf{z}}_1(2) = \mathbf{x}(1) - \mathbf{x}(2) + 1$*. Next, it applies the ReLU non-linear activation function (as indicated by the two linear pieces inside the circle depicting the neurons of layer 1). In other words, it computes* $\mathbf{z}_1(1) = \max\{\hat{\mathbf{z}}_1(1), 0\}$ *and* $\mathbf{z}_1(2) = \max\{\hat{\mathbf{z}}_1(2), 0\}$*. The third layer (layer 2, which is the second hidden layer) consists of two neurons. Its output is computed in two stages. First, it applies a linear transformation as indicated by the weights (shown along the edges connecting the neurons of layer 1 to the neurons of layer 2) and the biases (shown as a square on top of the neurons of layer 2). Specifically, it computes* $\hat{\mathbf{z}}_2(1) = 3\mathbf{z}_1(1) - \mathbf{z}_1(2) + 1$ *and* $\hat{\mathbf{z}}_2(2) = \mathbf{z}_1(1) + 2\mathbf{z}_1(2) + 3$*. Next, it applies the ReLU non-linear activation function (as indicated by the two linear pieces inside the circle depicting the neurons of layer 2). In other words, it computes* $\mathbf{z}_2(1) = \max\{\hat{\mathbf{z}}_2(1), 0\}$ *and* $\mathbf{z}_2(2) = \max\{\hat{\mathbf{z}}_2(2), 0\}$*. The final layer (layer 3) denotes the scalar output* $y = 3\mathbf{z}_2(1) - 2\mathbf{z}_2(2)$*. Note that the final layer does not use ReLU non-linear activation.*

layer of the MLP, each consisting of a set of *neurons* shown as circles. We will denote the number of layers as $L + 1$. The leftmost layer (that is, layer 0) denotes the input $\mathbf{x}$. The rightmost layer (that is, layer L) denotes the output y. For simplicity, we will assume that the output will always be a scalar. The layers between the input and the output are referred to as hidden layers. Using its layers, an MLP computes an output y of an input $\mathbf{x}$ in L steps. At a step $l \in \{1, \cdots, L\}$, the layer l takes as its input the output of layer $l-1$ and generates its output via a composition of two functions: a linear mapping, followed by a non-linear activation.

Formally, let us denote the output of a layer l as $\mathbf{z}_l$. Furthermore, we denote $\mathbf{z}_0 = \mathbf{x}$, that is, by convention we assume that the output of the input layer is the input itself. Given the vector $\mathbf{z}_{l-1}$ (that is, the output of the layer $l-1$), layer l computes an intermediate vector $\hat{\mathbf{z}}_l = \mathbf{W}_l\mathbf{z}_{l-1} + \mathbf{b}_l$. The term $\mathbf{W}_l$ is called the weight matrix of the layer l. The elements of the weight matrix are shown

next to the solid lines connecting the neurons of layer $l-1$ to the neurons of layer l in figure 1. The term $\mathbf{b}_l$ is called the bias vector of the layer l. The elements of the bias vector are shown inside the squares that are on top of the neurons of layer l. Using the intermediate vector $\hat{\mathbf{z}}_l$, the output of layer l is computed as $\mathbf{z}_l = \sigma(\hat{\mathbf{z}}_l)$, where $\sigma(\cdot)$ denotes a vector valued activation function. For simplicity, we will only focus on a single type of activation function known as rectified linear unit (ReLU): $\mathbf{z}_l = \sigma(\hat{\mathbf{z}}_l) = \max\{0, \hat{\mathbf{z}}_l\}$, where the maximisation is done in an element-wise fashion. Another way of understanding the ReLU activation is that its output $\mathbf{z}_l$ is related to its input $\hat{\mathbf{z}}_l$ as follows:

$$\mathbf{z}_l(i) = \begin{cases} \hat{\mathbf{z}}_l(i), & \text{if } \hat{\mathbf{z}}_l(i) \geq 0, \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

In other words, the ReLU activation function replaces all negative elements in its input vector with zeroes, and leaves the remaining elements unchanged. In figure 1, the ReLU activation function is depicted by the two linear pieces drawn inside the circle representing the neuron of a hidden layer. Note that the ReLU activation function has been depicted for the neurons of the hidden layers but not the input and the output layers. As mentioned earlier, the output of the input layer is assumed to be the input itself. Furthermore, as is the convention, the ReLU activation function is not applied for the final output layer, that is, $y = \mathbf{z}_L = \hat{\mathbf{z}}_L$. The pseudo-code for generating the output of a given input using an MLP is provide in algorithm 1.

Algorithm 1 *Computing the output y of an input $\mathbf{x}$ using an MLP.*

input The vector $\mathbf{x}$, the weights $\mathcal{W} = \{\mathbf{W}_l, l = 1, \cdots, L-1\} \bigcup \{\mathbf{w}_L\}$ and the biases $\mathcal{B} = \{\mathbf{b}_l, l = 1, \cdots, L\} \bigcup \{b_L\}$.

1: Initialise $\mathbf{z}_0 = \mathbf{x}$ and $l = 1$.
2: **while** $l < L$ **do**
3: Compute $\hat{\mathbf{z}}_l = \mathbf{W}_l \mathbf{z}_{l-1} + \mathbf{b}_l$.
4: Compute $\mathbf{z}_l = \sigma(\hat{\mathbf{z}}_l) = \max\{0, \hat{\mathbf{z}}_l\}$.
5: Increment $l = l + 1$.
6: **end while**
7: Compute $y = \mathbf{w}_L^\top \mathbf{z}_{L-1} + b_L$.

output The scalar y.

Verification of MLP. A verification instance will be specified by an MLP, that is, its weights $\mathcal{W}$ and biases $\mathcal{B}$, as well as an input domain $\mathcal{X}$. In practice, the input domain $\mathcal{X}$ will be defined by box constraints using two vectors $\mathbf{x}^{min}$ and $\mathbf{x}^{max}$. Formally, we define $\mathcal{X} = \{\mathbf{x}, \text{s.t. } \mathbf{x}^{min} \leq \mathbf{x} \leq \mathbf{x}^{max}\}$[1]. The property that we wish to verify is that for all possible inputs $\mathbf{x} \in \mathcal{X}$, the output of the MLP is less

[1]Two vectors $\mathbf{a}$ and $\mathbf{b}$ satisfy the inequality $\mathbf{a} \leq \mathbf{b}$ if they are of the same size and $\mathbf{a}(i) \leq \mathbf{b}(i)$ for all the elements i.

than or equal to 0. Recall that the final layer does not use a ReLU activation. Hence, the output can be of either sign. While the form of the property appears to be restrictive, very complex properties can be reduced to this canonical form by appending additional layers to a given neural network. For those who are interested, see [2].

A property is true if there exists no $\mathbf{x} \in \mathcal{X}$ for which the corresponding output is positive. If a given property is true, we would like to formally prove that it is true. A property is false if there exists at least one $\mathbf{x} \in \mathcal{X}$ for which the output is positive. If a given property is false, we would like to generate a counter-example, that is, an input for which the corresponding output is positive.

2 Data Set and File Format

Before we go into the details of how neural network properties can be verified, let us quickly describe the data set and file formats that will be used in this project. We will be using a slightly modified 'Collision Detection' data set [4], which is available to download from the course webpage. The only modification to the original data set is to reformulate the properties in the canonical form described above. The data set consists of 500 MAT files (property001.mat to property500.mat) and one text file (groundtruth.txt). Each MAT file defines a property by specifying the input domain and the neural network weights and biases. For all 500 properties, the neural network consists of 6 layers: (i) layer 0, the input layer, is 6 dimensional; (ii) layer 1 is 40 dimensional; (iii) layer 2 is 40 dimensional; (iv) layer 3 is 38 dimensional; (v) layer 4 is 19 dimensional; and (vi) layer 5 is 1 dimensional (the scalar output). The input domain is specified by box constraints. In more detail, each MAT file contains the following variables.

1. `xmin` and `xmax`: 6 dimensional row vectors that specify the box constraints of the input domain, that is, the minimum and maximum values of each element of the input respectively.

2. `W`: a cell array of size 5. The cell `W{1}` specifies the weight matrix of the l-th layer.

   (a) `W{1}`: a matrix of size 40×6, which contains the weights for layer 1.

   (b) `W{2}`: a matrix of size 40×40, which contains the weights for layer 2.

   (c) `W{3}`: a matrix of size 38×40, which contains the weights for layer 3.

   (d) `W{4}`: a matrix of size 19×38, which contains the weights for layer 4.

   (e) `W{5}`: a 19 dimensional row vector, which contains the weights for layer 5.

3. `b`: a cell array of size 5. The cell `b{l}` specifies the biases of the l-th layer.

   (a) `b{1}`: a 40 dimensional column vector, which contains the biases for layer 1.

   (b) `b{2}`: a 40 dimensional column vector, which contains the biases for layer 2.

   (c) `b{3}`: a 38 dimensional column vector, which contains the biases for layer 3.

   (d) `b{4}`: a 19 dimensional column vector, which contains the biases for layer 4.

   (e) `b{5}`: a scalar bias for layer 5.

The text file contains the ground-truth information for each property. The i-th line of the text file specifies whether the i-th property is true or false. Recall that a property is true if no input in the specified domain results in an output that is positive, and is false if there exists at least one input whose output is positive. The ground-truth information has been provided only for debugging purposes. In practice, we will not know whether a property is true or false until we have verified it.

3 Lower Bound for Neural Network Output

We begin with a simple *unsound method* for neural network verification, that is, a method that can prove that some of the false properties are indeed false by generating a counter-example. We denote the maximum output over all possible inputs $\mathbf{x} \in \mathcal{X}$ as y^*. If we knew the value of y^*, verification would boil down to checking its sign. If y^* is positive, the property is false. Otherwise, the property is true. The computational challenge of verification lies in obtaining the value of y^*. While its exact value is hard to compute, we can readily obtain a lower bound as follows. We generate k random inputs that lie in the input domain. We compute the output for each of the k random inputs. The maximum value among the k outputs is a valid lower bound on y^*. If the lower bound itself is positive, it means that the property is false since a counter-example has been identified.

Note that the guarantee provided by the above unsound method is very weak. A given property may be false, but we may not be able to prove that it is false as we may fail to generate a counter-example. A given property may be true, in which case we won't be able to give any guarantees at all. Nonetheless, this approach of generating a lower bound will play an important part when we develop our final strategy to verify neural networks in section 6.

Task 1: Lower Bound for Neural Network Output.

- Write a function `generate_inputs`, which takes as its input the vectors `xmin` and `xmax` as well as an integer k and returns a matrix of size $k \times 6$ that contains a valid input in each of its k rows. Recall that each input is 6 dimensional in our data set.

  ```
  X = generate_inputs(xmin,xmax,k)
  ```

  You would need to use the in-built Matlab function `rand` to generate the inputs.

- Write a function `compute_nn_outputs`, which takes as its inputs the matrix `X` as well as the neural network weights and biases, and returns a vector of size $k \times 1$ that contains the outputs of the given inputs.

  ```
  y = compute_nn_outputs(W,b,X)
  ```

  You may be tempted to loop over the k inputs, but remember that Matlab is very slow when it comes to loops. A more efficient implementation would use matrix operations to simultaneously evaluate all k outputs.

- Increase the value of k from 1 to some maximum value. For each value of k, plot the average amount of time (averaged over all 500 properties) taken to generate the outputs. Also plot the average lower bound as well as the number of properties for which you were able to successfully find a counter-example. You would need to use the in-built Matlab function `plot` or `bar`.

4 Interval Bound Propagation

In the previous section, we discussed an unsound method that computes a lower bound on the maximum neural network output over a given input domain. This allows it to prove that some (but not all) false properties are indeed false by generating a counter-example. In this section, we will discuss an *incomplete method* that computes an upper bound on the maximum neural network output over a given domain. The upper bound can be used to show that some (but not all) true properties are indeed true. Specifically, if the upper bound itself is non-positive, then the property has to be true. However, if the upper bound is negative, then the property could be true or could be false.

The method that we will adopt to generate an upper bound is called interval bound propagation. To understand interval bound propagation, we will use the example of the small MLP shown in figure 1. The input is a vector of size 2 and the input domain $\mathcal{X}$ is specified by box constraints where $\mathbf{x}^{min}(1) = \mathbf{x}^{min}(2) = -2$ and $\mathbf{x}^{max}(1) = \mathbf{x}^{max}(2) = 2$. In other words, we have been provided with both a lower and an upper bound for the values of the neurons in layer 0. Using these bounds, we can compute lower and upper bounds for the values of the neurons in layer 1. Specifically, recall that $\hat{\mathbf{z}}_1(1) = \mathbf{x}(1) + \mathbf{x}(2) + 2$ and $\hat{\mathbf{z}}_1(2) = \mathbf{x}(1) - \mathbf{x}(2) + 1$. One simple way to obtain upper bounds on $\hat{\mathbf{z}}_1$ (denoted by $\hat{\mathbf{z}}^{max}$) is to note that

$$\begin{aligned} \hat{\mathbf{z}}_1(1) &\leq \mathbf{x}^{max}(1) + \mathbf{x}^{max}(2) + 2, \forall \mathbf{x} \in \mathcal{X}, \\ \hat{\mathbf{z}}_1(2) &\leq \mathbf{x}^{max}(1) - \mathbf{x}^{min}(2) + 1, \forall \mathbf{x} \in \mathcal{X}. \end{aligned} \tag{2}$$

Thus, we can obtain a valid upper bound $\hat{\mathbf{z}}^{max}(1) = 6$ and $\hat{\mathbf{z}}^{max}(2) = 5$. Similarly, we can obtain lower bounds on $\hat{\mathbf{z}}_1$ (denoted by $\hat{\mathbf{z}}^{min}$) by noting that

$$\begin{aligned} \hat{\mathbf{z}}_1(1) &\geq \mathbf{x}^{min}(1) + \mathbf{x}^{min}(2) + 2, \forall \mathbf{x} \in \mathcal{X}, \\ \hat{\mathbf{z}}_1(2) &\geq \mathbf{x}^{min}(1) - \mathbf{x}^{max}(2) + 1, \forall \mathbf{x} \in \mathcal{X}. \end{aligned} \tag{3}$$

The above inequalities provide a valid lower bound $\hat{\mathbf{z}}^{min}(1) = -2$ and $\hat{\mathbf{z}}^{min}(2) = -3$. Given the upper and lower bounds for $\hat{\mathbf{z}}(1)$, it is now possible to compute upper and lower bounds for the output of the neurons of layer 1 (denoted by $\mathbf{z}^{max}$ and $\mathbf{z}^{min}$ respectively) using the relationship $\mathbf{z}_1(i) = \max\{0, \hat{\mathbf{z}}_1(i)\}$ for each neuron i in layer 1. Thus, we obtain $\mathbf{z}^{min}(1) = 0$, $\mathbf{z}^{min}(2) = 0$, $\mathbf{z}^{max}(1) = 6$ and $\mathbf{z}^{max}(1) = 5$. Applying this process starting from the leftmost layer and continuing to the rightmost layer provides us with valid upper and lower bounds for the output.

Let us define the process of interval bound propagation formally. We assume that we have already computed the upper and lower bounds of layer $l-1$, which we denote by $\mathbf{z}_{l-1}^{max}$ and $\mathbf{z}_{l-1}^{min}$ respectively. Recall that the upper and lower bounds for $\mathbf{z}_0$ will be specified as part of the property we wish to verify. We will use these bounds to compute the upper and lower bounds for layer l. To this end, we define the following two matrices:

$$\mathbf{W}_l^+ = \max\{0, \mathbf{W}_l\}, \mathbf{W}_l^- = \min\{0, \mathbf{W}_l\}, \tag{4}$$

where the maximisation and minimisation is performed element-wise. In other words, $\mathbf{W}_l^+$ contains all the positive elements of the matrix $\mathbf{W}_l$ and zeros in place of all its negative elements. Similarly, $\mathbf{W}_l^-$ contains all the negative elements of the matrix $\mathbf{W}_l$ and zeros in place of all its positive elements. Using the above notation, we define

$$\begin{aligned}\hat{\mathbf{z}}_l^{max} &= \mathbf{W}_l^+ \mathbf{z}_{l-1}^{max} + \mathbf{W}_l^- \mathbf{z}_{l-1}^{min} + \mathbf{b}_l, \\ \hat{\mathbf{z}}_l^{min} &= \mathbf{W}_l^+ \mathbf{z}_{l-1}^{min} + \mathbf{W}_l^- \mathbf{z}_{l-1}^{max} + \mathbf{b}_l. \end{aligned} \tag{5}$$

The upper and lower bounds for $\mathbf{z}_l$ can then be computed as

$$\mathbf{z}_l^{max} = \max\{0, \hat{\mathbf{z}}_l^{max}\}, \mathbf{z}_l^{min} = \max\{0, \hat{\mathbf{z}}_l^{min}\}. \tag{6}$$

Note that, at termination, interval bound propagation provides both an upper and a lower bound for the output of the neural network. In practice, the lower bound computed in this manner is smaller (that is, of inferior quality) than the one computed by random sampling of inputs as in the previous section. Thus, interval bound propagation is typically only used to compute an upper bound.

Task 2: Interval Bound Propagation.

- Write a function `interval_bound_propagation`, which takes as its inputs the neural network weights and biases as well as the upper and lower bounds of the input, and returns the upper and lower bounds of the output.

  ```
  [ymin,ymax] = interval_bound_propagation(W,b,xmin,xmax)
  ```

- Report the average values of the upper and lower bounds computed using interval bound propagation over all 500 properties. How does the lower bound computed using interval bound propagation compare with the one computed in the previous task? Also report the number of properties that you were able to prove using interval bound propagation.

5 Linear Programming Bound

The advantage of interval bound propagation is that it is easy to understand and relatively easy to code. The disadvantage of interval bound propagation is that the upper bound computed in this manner is usually very loose (that is, its value tends to be significantly higher than the actual value of the maximum output of the neural network over the input domain). This looseness of the upper bound means that we won't be able to show that a significant number of true properties are indeed true. How can we remedy this? To answer this question, let us first look at an ideal scenario where computational complexity is not an issue. Similar to the previous section, let us assume that we have already computed (or have been given) upper and lower bounds for the neurons of all the layers up to $l-1$. We now wish to compute bounds for the neurons of layer l. We can formulate this mathematically as the following optimisation problem:

$$\begin{aligned}
\hat{\mathbf{z}}_l^{max}(i) \equiv \max \quad & \hat{\mathbf{z}}_l(i) \\
\text{s.t.} \quad & \mathbf{x}^{min} \leq \mathbf{z}_0 \leq \mathbf{x}^{max}, \\
& \mathbf{z}_k^{min} \leq \mathbf{z}_k \leq \mathbf{z}_k^{max}, \forall k \in \{1, \cdots, l-1\}, \\
& \hat{\mathbf{z}}_k = \mathbf{W}_k \mathbf{z}_{k-1} + \mathbf{b}_k, \forall k \in \{1, \cdots, l\}, \\
& \mathbf{z}_k = \max\{0, \hat{\mathbf{z}}_k\}, \forall k \in \{1, \cdots, l-1\}.
\end{aligned} \tag{7}$$

The above problem computes the upper bound of the i-th element of $\hat{\mathbf{z}}_l$. An analogous problem can be used to compute the lower bound of the i-th element of $\hat{\mathbf{z}}_l$ by simply changing the problem to a minimisation problem instead of a maximisation one. Once all the elements of $\hat{\mathbf{z}}_l^{max}$ and $\hat{\mathbf{z}}_l^{min}$ have been computed, we obtain $\mathbf{z}_l^{max} = \max\{0, \hat{\mathbf{z}}_l^{max}\}$ and $\mathbf{z}_l^{min} = \max\{0, \hat{\mathbf{z}}_l^{min}\}$.

The computational difficulty of the above optimisation problem is that the final set of constraints are in fact not linear. In other words, the above problem is not a linear program, which makes it hard to solve[2]. This observation forces us to approximate the problem for the sake of efficiency. To this end, it is beneficial to visualise the non-linear constraint as shown in figure 2 (left). The x-axis is the value of $\hat{\mathbf{z}}_k(i)$ and the y-axis is the corresponding value of $\mathbf{z}_k(i)$. This set of points shown in red cannot be represented using linear constraints. However, figure 2 (right) shows a superset of the points (that is, all the points inside or on the boundary of the blue triangle), which can be specified using three linear inequality constraints. Note that the exact specification of the linear inequality constraints would

[2]Technically, it is its non-convexity that makes it hard to solve.

require the knowledge of $\hat{\mathbf{z}}_k^{min}(i)$ and $\hat{\mathbf{z}}_k^{max}(i)$. However, since $k \in \{0, \cdots, l-1\}$, by our assumption we already know these values when we are computing the bounds for layer l.

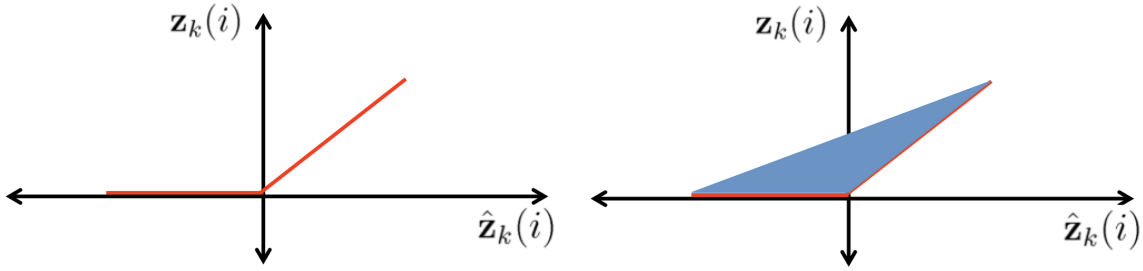

Figure 2: *Visualisation of the non-linear constraint corresponding to the ReLU activation function (left) and its approximation that is suitable for linear programming (right).*

The observation that the non-linear constraints can be approximated using linear constraints leads to an effective strategy to compute upper and lower bounds on the value of the neurons in layer l. Specifically, we approximate problem (7) as a linear program by replacing the non-linear constraints with the linear constraints that specify the triangular superset. Note that we have effectively loosened the constraint set, which means that the optimal value of the linear program will necessarily be greater than or equal to the optimal value of problem (7). Since the optimal value of problem (7) provides a valid upper bound, it follows that the potentially higher optimal value of the linear program will also be a valid upper bound. A similar argument also holds when we are computing a lower bound via a linear programming approximation.

Task 3: Linear Programming Bound.

- Write a function `linear_programming_bound`, which takes as its inputs the neural network weights and biases as well as the upper and lower bounds of the input, and returns the upper and lower bounds of the output.

  ```
  [ymin,ymax] = linear_programming_bound(W,b,xmin,xmax)
  ```

  To solve each linear program, you would need to use the in-built Matlab function `linprog`.

- Report the average value of the upper and lower bounds computed using linear programming (averaged over all 500 properties). How do the linear programming bounds compare with those computed in the previous two tasks? Also report the number of properties which you were able to prove as true or false using linear programming bounds.

6 Branch-and-Bound

So far we have only considered either unsound or incomplete methods, which have the obvious flaw that they can sometime fail to provide any guarantee for a given property. An unsound method may fail to generate a counter-example (that is, provide a negative lower bound), in which case the property may be true or may be false. Similarly, an incomplete method may provide a positive upper bound, which again implies that we won't be able to find out whether a property is true or false. We will now design a *complete method* for neural network verification: one that will always be able to prove that a property is true, or provide a counter-example to show that it is false. Specifically, we will focus on the branch-and-bound approach. As indicated by its name, branch-and-bound uses methods for estimating lower and upper bounds are key subroutines.

Branch-and-bound iteratively partitions the input domain. A partition $\mathcal{P}$ of $\mathcal{X}$ is a set of subdomains that are collectively exhaustive. In other words, a partition $\mathcal{P}$ of size n is a set $\{\mathcal{X}_1, \mathcal{X}_2, \cdots, \mathcal{X}_n\}$ such that $\mathcal{X}_1 \bigcup \mathcal{X}_2 \bigcup \cdots \bigcup \mathcal{X}_n = \mathcal{X}$. The partition is initialised as $\mathcal{P}_0 = \{\mathcal{X}\}$, that is, the size of the starting partition is 1 and its only element is the entire input domain $\mathcal{X}$. At each iteration, the partition size increases by 1 (that is, $|\mathcal{P}_{t+1}| = |\mathcal{P}_t| + 1$, where $|\cdot|$ denotes the cardinality of a set), until a termination criterion is reached. In what follows, we describe the operations that need to be performed at each iteration of branch-and-bound.

Let us denote the partition at the start of iteration t as $\mathcal{P}_{t-1} = \{\mathcal{X}_1, \mathcal{X}_2, \cdots, \mathcal{X}_n\}$. We will assume that we have already computed upper and lower bounds for the neural network outputs when the input is restricted to each of the n subdomains. We now choose the subdomain with the largest value of the upper bound among all n subdomains. Let us denote this subdomain by $\bar{\mathcal{X}} \in \mathcal{P}$, and the vectors representing its box constraints as $\bar{\mathbf{x}}^{min}$ and $\bar{\mathbf{x}}^{max}$.

If the upper bound of $\bar{\mathcal{X}}$ is negative, then it means that the upper bounds of all the subdomains in the partition $\mathcal{P}_{t-1}$ are negative (since $\bar{\mathcal{X}}$ is the subdomain with the largest value of the upper bound). However, since the partition covers the entire input domain $\mathcal{X}$, it follows that the largest value of the output among all the inputs in the domain $\mathcal{X}$ cannot be greater than the largest upper bound among all the subdomains in the partition. Hence, if the upper bound of $\bar{\mathcal{X}}$ is negative, then we can terminate the branch-and-bound procedure and declare that the property is true.

If the upper bound of $\bar{\mathcal{X}}$ is positive, we will try to refine it by partitioning $\bar{\mathcal{X}}$ further into two equal parts. One possible way to achieve this is to *cut the longest edge*. Formally, we compute

$$\mathbf{s}(i) = \frac{\bar{\mathbf{x}}^{max}(i) - \bar{\mathbf{x}}^{min}(i)}{\mathbf{x}^{max}(i) - \mathbf{x}^{min}(i)}. \tag{8}$$

The i-th element of $\mathbf{s}$ denotes the relative length of the i-th dimension of the subdomain $\bar{\mathcal{X}}$ compared to the original domain $\mathcal{X}$. We choose the dimension with the longest relative length, that is, $j = \arg\max_i \mathbf{s}(i)$, and split the subdomain into two along this dimension. In other words, we partition $\bar{\mathcal{X}}$ into two equal halves denoted by $\bar{\mathcal{X}}_1$ and $\bar{\mathcal{X}}_2$. The subdomain $\bar{\mathcal{X}}_1$ is defined by box constraints with vectors $\bar{\mathbf{x}}_1^{min}$ and $\bar{\mathbf{x}}_1^{max}$, where $\bar{\mathbf{x}}_1^{min} = \bar{\mathbf{x}}^{min}$ and

$$\bar{\mathbf{x}}_1^{max}(i) = \begin{cases} \bar{\mathbf{x}}^{max}(i) & \text{if } i \neq j, \\ \bar{\mathbf{x}}^{min}(i) + \frac{\bar{\mathbf{x}}^{max}(i) - \bar{\mathbf{x}}^{min}(i)}{2} & \text{otherwise.} \end{cases} \tag{9}$$

Similarly, the subdomain $\bar{\mathcal{X}}_2$ is defined by box constraints with vectors $\bar{\mathbf{x}}_2^{min}$ and $\bar{\mathbf{x}}_2^{max}$, where $\bar{\mathbf{x}}_2^{max} = \bar{\mathbf{x}}^{max}$ and

$$\bar{\mathbf{x}}_2^{min}(i) = \begin{cases} \bar{\mathbf{x}}^{min}(i) & \text{if } i \neq j, \\ \bar{\mathbf{x}}^{min}(i) + \frac{\bar{\mathbf{x}}^{max}(i) - \bar{\mathbf{x}}^{min}(i)}{2} & \text{otherwise.} \end{cases} \tag{10}$$

For each of the two new subdomains $\bar{\mathcal{X}}_1$ and $\bar{\mathcal{X}}_2$ we compute upper and lower bounds of the neural network output. Since both $\bar{\mathcal{X}}_1$ and $\bar{\mathcal{X}}_2$ are subsets of $\bar{\mathcal{X}}$, it follows that the upper bounds of $\bar{\mathcal{X}}_1$ and $\bar{\mathcal{X}}_2$ are less than or equal to the upper bound of $\bar{\mathcal{X}}$ and the lower bounds of $\bar{\mathcal{X}}_1$ and $\bar{\mathcal{X}}_2$ are greater than or equal to the lower bound of $\bar{\mathcal{X}}$. In other words, partitioning $\bar{\mathcal{X}}$ into two equal halves has resulted in better upper and lower bounds. Two possible cases arise:

1. The lower bound of either $\bar{\mathcal{X}}_1$ or $\bar{\mathcal{X}}_2$ is positive, in which we have found a counter-example and the property is false. Hence, we can terminate.

2. If both the lower bounds are negative, we create a new partition $\mathcal{P}_t$ which replaces $\bar{\mathcal{X}}$ with $\bar{\mathcal{X}}_1$ and $\bar{\mathcal{X}}_2$, thereby increasing the size of the partition by 1.

The branch-and-bound procedure will terminate after a finite number of iterations. In theory, the number of iterations can be large. However, the 'Collision Detection' data set is a fairly simple one. So if you have implemented all the subroutines correctly, the branch-and-bound procedure should terminate within a few minutes for each property. Note that the branch-and-bound procedure should

always terminate with the right answer. However, you may need to pay attention to numerical instability issues that arise due to approximations in floating point arithmetic.

Task 4: Branch-and-Bound.

- Write a function `branch_and_bound`, which takes as its inputs the neural network weights and biases as well as the upper and lower bounds of the input, and returns a Boolean flag indicating whether the property is true or false.

  ```
  flag = branch_and_bound(W,b,xmin,xmax)
  ```

  You should try all possible combinations of upper and lower bounds that you have implemented as part of the previous tasks within your branch-and-bound framework.

- Plot the number of properties that have been correctly verified as either true or false as a function of the time taken by the branch-and-bound procedure.

7 Extensions

There are several possible extensions. You may wish to try running your code on a more challenging scenario, such as the ACAS data set [5]. The more challenging scenario may require better lower bounds [3], better upper bounds [1] and/or better branching strategies [2]. You could also try to formulate the verification problem as a mixed integer linear program [2], and use it with commercial solvers such as Gurobi or Mosek.

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 Confernece 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] N. Carlini and D. Wagner. Towards evaluating the robustness of neural networks. In *IEEE Symposium on Security and Privacy*, 2017.

[4] R. Ehlers. Formal verification of piece-wise linear feed-forward neural networks. In *Automated Technology for Verification and Analysis*, 2017.

[5] 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.