Trinity Term 2020

B1 Engineering Computation: Project B Report

1042767

1 Introduction

The advancement of deep-learning technologies allows them to be utilised in a range of industries, from autonomous vehicles to banking. The use of neural networks in these sensitive settings makes the task of verifying that a certain neural network is behaving according to a set of desirable properties imperative for our safety [1]. This paper explores various algorithms that perform Neural Network Verification. The property to be satisfied here is: for any input **x**, the neural network produces a negative output.The neural network under consideration is a Multilayer Perceptron (MLP), which is a feed-forward neural network composed of an input/output layer and an arbitrary number of hidden layers [2]. A ground truth document is available that labels the 172 false and 328 true properties.

2 Implementation

2.1 Input Generation

For the neural network to generate an output and be verified, we need an appropriate input. The data provided contains the box constraints [xmin,xmax] for an acceptable input, from which a set of inputs will be randomly generated. To implement this, I've first written a function that generates k-inputs of the size specified, within the boundaries. In this specific case this function generates a kx6 array. As seen from Code 1, the generation uses the rand function which generates normalised random values that are then shifted to the range xmin, xmax.

```matlab
function x = generate_inputs(xmin,xmax,k)
    i = 1:size(xmin,2); %row size
    x = xmin + (xmax-xmin).*rand(k,1); %array multiplication
end
```

Matlab Code 1: Input Generation

2.2 Output Computation

The second main task before the verification process is to generate the outputs that will be verified. The neural network is composed of simple linear and ReLu functions. It can be written as:

$$y = z_5 = \mathcal{W}_5\left(R_4\left(\mathcal{W}_4\left(R_3\left(\mathcal{W}_3\left(R_2\left(\mathcal{W}_2\left(R_1\left(\mathcal{W}_1\mathbf{z}_0 + b_1\right)\right) + b_2\right)\right) + b_3\right)\right) + b_4\right) + b_5 \quad (1)$$

Where R represents the ReLu function $R(x) = max(0, x)$ and $\mathbf{z}_0$ is the input vector.The function compute_nn_outputs (shown below) inputs the weights and biases of all the layers of the Neural Network and the $\mathbf{z}_0$, generated by the former function. It then returns the output of the Neural Network through the implementation of equation (1) with a while loop.

```matlab
function y = compute_nn_outputs(W,b,x)
    z = x';
    i = 1;
    while i < size(W,2)
        z = W{i}*z + b{i};
        z(z<0) = 0;
        i = i + 1;
    end
    y = W{size(W,2)}*z + b{size(W,2)};
end
```

Matlab Code 2: Output Computation

2.3 Interval Bound Propagation (IBP)

For some of the algorithms the maxima/minima of the output corresponding to a certain input is needed to verify a property. A simple way of acquiring the maximum output is to start with the box constraints and calculate the bounds for the output of the first layer by applying the positive weights to xmax and negative weights to xmin and vice versa for the minimum. The bounds generated are the input bounds for the next layer, and upon repetition of the same calculation the output bounds are acquired. Matlab Code 3 shows the boundary calculation for one layer.

```matlab
    W_max = W{i};W_max(W_max < 0) = 0;
    W_min = W{i};W_min(W_min > 0) = 0;

    z_max = W_max*zmax + W_min*zmin + b{i};
    z_min = W_max*zmin + W_min*zmax + b{i};
```

Matlab Code 3: Interval Bound Generation

2.4 Projected Gradient Ascent

Instead of using randomly generated inputs, we can choose to refine these inputs according to a certain optimisation objective. Since we are exploring the maxima and minima of the output, it is critical to refine the inputs so that the extremes of the output range are generated. Essentially, if we feed the neural network with inputs that satisfy the optimization of $\max_{\mathbf{x}\in\mathcal{X}} f(\mathbf{x};\mathcal{W},\mathcal{B})$, where function *f* represents our neural network, we will achieve a more accurate verification. To this end, we need to iteratively refine the randomly generated **x** using the following update formula:

$$\mathbf{x}'_{t+1} = \mathbf{x}_t + \eta_t \nabla f\left(\mathbf{x}_t;\mathcal{W},\mathcal{B}\right)$$

In simple terms, we are updating the input vector **x** so that it follows the upward gradient with a step-size/learning rate η_t towards a maximum point of $f\left(\mathbf{x}_t;\mathcal{W},\mathcal{B}\right)$. The neural network is composed of simple linear and ReLu functions as shown in equation (1). In order to calculate the derivative we need to utilise the chain rule. For the sake of this project, we will assume that the derivative of the ReLu function at 0 is 0 despite being infinite. For a successive layer-pair, the gradient can be represented with equation (3), where j represents the number of the layer, i the dimension of x and A the activation function. For layers 2-4, A is the ReLU function and for layers 1 and 5 it is 1.

$$\frac{\partial z_j}{\partial x_i} = \frac{\partial z_j}{\partial A_j}\frac{\partial A_j}{\partial z_{j-1}}\frac{\partial z_{j-1}}{\partial x_i} \tag{3}$$

This indicates that the gradient is a combination of the weights and the derivative of the ReLu function where appropriate. The salient part of gradient calculation is shown in Matlab Code 5 below.

```
...
C{1} = W{1};
i = 1;
    while i < size(W,2)
        z = W{i}*z + b{i};
        z(z<0) = 0;
        r = z;r(r>0) = 1;
        C{i+1} = r'.* W{i+1};
        i = i + 1;
    end
   grad = C{5}*C{4}*C{3}*C{2}*C{1};
   ...
```

Matlab Code 4: Gradient Calculation

2.5 Linear Programming Bound (LPB)

It is possible to improve the loose bounds provided by Interval Bound Propagation by converting the task into a layer-by-layer optimisation problem using the upper and lower bounds of the neurons in the previous layer and the ReLU activation (for all layers but the previous) as constraints for each optimisation. The task is to optimise the input of each neuron so that the output is a maximum/minimum.

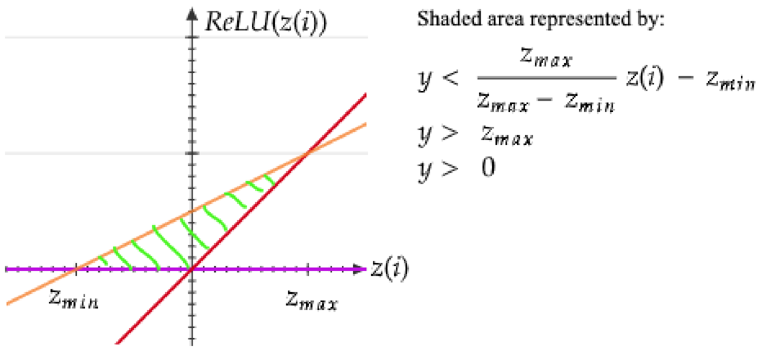

ReLU(z(i))
Shaded area represented by:
$y < \frac{z_{max}}{z_{max} - z_{min}} z(i) - z_{min}$
$y > z_{max}$
$y > 0$
z(i)
z_{min}
z_{max}

Fig. 1: Linearised ReLu Constraints

Unless the bounds provided are both positive or negative which simplifies the ReLU constraint to $y = z(i)$ or $y = 0$ respectively, the ReLU constraint is non-linear. To make this a linear optimization problem the ReLU function is approximated by breaking it down into three separate linear constraints as shown in Figure 4. The optimization problem is constructed by implementing the constraints and solved for every neuron in every layer using the linprog function of MATLAB. The function inputs f,A,B which encapsulate the objective function and the constraints. If we set $f = [b(n), W(n,:)]$ and constrain the solution, x_{1x7}, so that the first dimension which gets multiplied by b is 1, and the remaining is z_{1x6}, the input to the layer, $f' * x$ will represent the objective function $W * z + b$. The constraints are represented by A and B in the form $A * x <= B$. For the first layer we have 14 constraints, 2 for the b multiplier to equal 1 $(x(1) <= 1, -x(1) <= -1)$ and 12 corresponding to box constraints $(x(i) <= xmax(i), -x(i) <= -xmin(i))$. Then for the second layer we add the ReLu constraints to A and B, 3 per node, and continue for every layer until we get the output bounds.

3 Verification Algorithms & Results

3.1 Lower Bound for Neural Network Output

If the maximum potential output over all the possible inputs is known, checking its sign would be enough for verification. However obtaining the value of the maximum possible output is difficult to compute. This verification algorithm produces k-output values from k-randomly generated inputs and the maximum output among the computed outputs is considered as a valid lower-bound for the maximum output for all possible inputs. This is implemented using functions mentioned in 2.1 and 2.2. It can be seen that this is an unsound method, as the property will not be proven false unless a counter-example is randomly generated.

3.1.1 Results

Fig.2(a) shows the relevant data from the Lower Bound Verification iterated over k-values ranging from 1 to 500. It can be observed that as the number of inputs increase, all of the variables plotted are increasing towards a steady-state. The spikes in computational time (eg. k = 80,170) can be explained by the dynamic memory allocation of Matlab, ie if the size of the array is beyond the available contiguous memory of its original location, MATLAB must make a copy of the array and move the copy into a memory block with sufficient space [3]. The accuracy increases significantly with k initially, but for k>70 the value for the lower bound and number of counter examples found stabilises. Increasing the value for k after that point only increases the computational time. The counter-example count fluctuates towards 42, and after a k value of around 60 it remains around 42 consistently. Similarly the average lower bound increases to reach -32 as k increases, stabilising around a k value of 60. Lower Bound Verification is based on the assumption that all properties are true unless a counter-example is found. This means the accuracy is measured by the correct identification of the false properties. With a k of 500, ie the maximum amount of outputs generated, 42 of 172 false cases (24.4% accuracy) were identified. Generating more inputs will increase the likelihood of a counter-example being generated and therefore the accuracy up to a certain saturation point. In this case it can be seen that a similar accuracy can be achieved with a lower k, and consequently lower computational time. With a bigger dataset it is critical to identify the saturation point to optimise computation time.

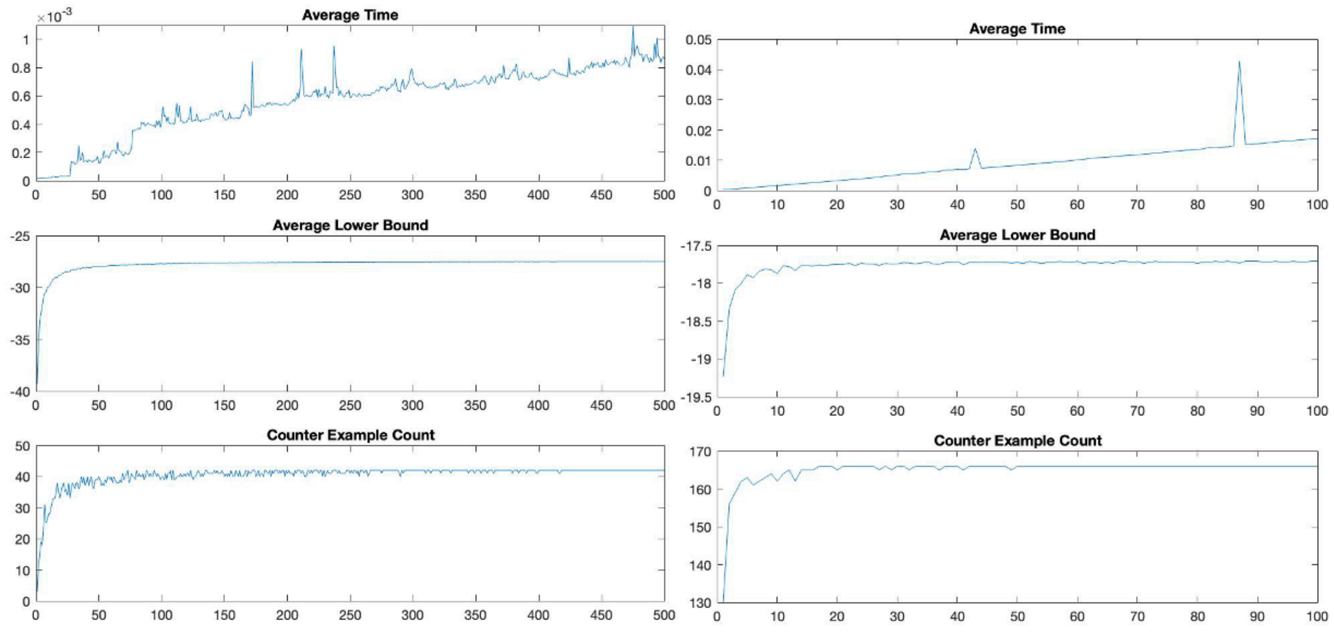


Fig. 2: Task1 Plots: (a)Original (b)Refined

3.1.2 Improvements

This method can be significantly improved if inputs are optimised so that they yield the maximum output rather than randomly generated. To this end, the inputs generated by the projected gradient ascent were used for the Lower Bound Verification. As shown in Fig.2(b) the results are significantly improved with just one fifth of the k value used in Section.3.1.1: 100. A new average lower bound of -18 and over 160 counter-examples are identified, pushing the accuracy up to 93.0%. The computation time is increased to 0.04 seconds, due to the gradient calculations for inputs with higher k, however the accuracy of the method accounts for this increase in time.

3.2 Interval Bound Propagation

Despite being a base function IBP can be used as a verification algorithm. A simple check of the signs of the maximum/minimum output that can be generated by the input boundaries (xmin,xmax) can be used to verify the properties. If the upper bound of the output is non-positive, we can definitely say that this property is true, if it is positive we cannot draw any conclusions. Similarly if the lower bound is positive the property is proved to be false, and no conclusions can be drawn otherwise. As we are only considering the two extremes of the given domain, this becomes an inefficient and incomplete method for verification.

3.2.1 Results

The calculation of the upper/lower bounds of the outputs generated by [xmin,xmax] took 0.1040 seconds and could only prove that 11 of the properties are definitely true. This corresponds to a 6.4% accuracy in identifying true properties which is very low. The average upper and lower bound calculated by this algorithm is 82.7 and -137 respectively. The lower bound is significantly lower and therefore of inferior quality than the average lower-bound calculated via the lower bound verification algorithm, which was around -31. Therefore this algorithm fails to identify any of the properties to be definitely false. Despite being inaccurate and inefficient when applied to the box constraints (xmin,xmax) only, the Branch and Bound algorithm uses this function to achieve a complete verification that yields significantly better results.

3.2.2 Improvements

For this simple verification to yield more accurate results we need tighter bounds on the output. This can be achieved by using the Linear Programming Bound function. As expected, using LPB considerably changed the outcome and the computational time which increased from 0.104 seconds to 41 minutes due to the multi-layered optimization problem. The average upper and lower bound calculated is -13.2 and -58.7 respectively. Both bounds decreased in magnitude which means they are of better quality when compared to the 82.7 and -137 in Section 3.2.1, however the lower bound is still looser compared to -31 and -18 of Section 3.1.1 and its improved version Section 3.1.2 respectively. 260 properties are proved to be true, which corresponds to a 79.2% accuracy . This is a significant increase from the IBP. The remaining 68 cases of false negatives indicate the LPB overestimates the upper-bound which makes sense considering the bounds are loosened through the approximation of the ReLU constraint. Similar to Section 3.2.2, the lower-bounds are underestimated and none of the properties are proven to be false by a lower-bound verification.

3.3 Branch and Bound

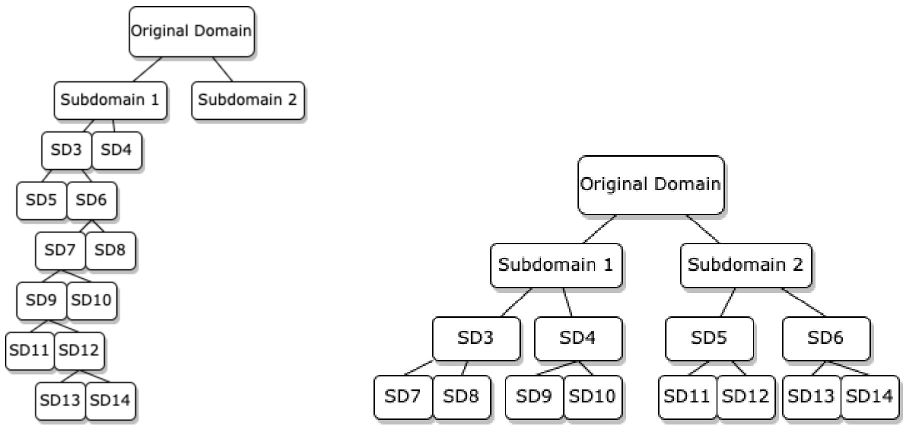
The branch and bound algorithm is a complete method for verification. The input domain [xmin,xmax] is iteratively partitioned and the upper/lower bounds are verified for each partition. This produces better upper/lower boundaries, until a counter-example is found or termination conditions are reached. In this case, the input vector is 6-dimensional and we partition from the dimension with the longest length relative to the original domain. This will ensure that all dimensions will get an even number of partitioning over the iterations. The initial partitioning of the original domain is done across a random dimension as all dimensions have the same relative length of 1. The chosen dimension is then split in half producing a lower and upper bound set for the subdomains: [x1min,x1max], [x2min,x2max], which produce the original domain when combined. Matlab Code 4 shows the implementation of the partitioning. The relative length (s) is calculated for each dimension, and the dimension j with the highest s is identified. Then the input domain is evenly split across j.

```
%split the partition with the max upper bound across largest dimension
s = (x_max(i,:) - x_min(i,:))./(xmax-xmin); %i corresponds to the partition
[~,j] = max(s);                             %  with the max upper bound
x1min = x_min(i,:); x2max = x_max(i,:);
x1max = x_max(i,:); x1max(j) = x_min(i,j) + (x_max(i,j)-x_min(i,j))/2;
x2min = x_min(i,:); x2min(j) = x_min(i,j) + (x_max(i,j)-x_min(i,j))/2;
```

Matlab Code 5: Partitioning

Once the new subdomains are obtained, the output boundaries for both are calculated using the interval bound propagation function in section 2.3.2. If the value for the minimum output (ymin) for any of the boundary sets is positive, iterations stop as a counter-example is found.

The problem of how we continue partitioning beyond the first split of the original domain can be tackled in different ways as seen represented in Figure 4.



(a) Method 1 (b) Method 2

Fig. 3: Visual Representation of Partitioning Methods

Given that both satisfy $ymin < 0$, the first method is as instructed in the project notes: the maximum possible output for the two sub-domains are compared and the partitioning is continued over the one with the higher maximum output value. While this is computationally more easy, if both of the sub-domains have a positive maximum output both can still contain the sub-domain that will yield the counter-example ie. $ymin > 0$. Additionally, the partition with the higher $ymax$ might have a true upper bound that is negative, which will flag the property as true while the unexplored subdomain may have a positive $ymax$ causing a false positive. So by tracking the sub-domain with the higher $ymax$ we are risking verifying a false property. The second method, which is a more complete way of doing this is by partitioning every sub-domain that yields a positive $ymax$. However this is computationally very challenging due to the exponential growth of the number of partitions. Hence, although this method guarantees the exploration of a complete range of sub-domains, the iterations are limited due to time constraints and the exploration is not as ”deep” as the first method. The upper/lower boundaries get better as the exploration continues deeper and the sub-domains get smaller, so the accuracy of the second method is significantly limited. This once again will cause the verification of false properties. When computational constraints are taken into account the two methods represent trade-off between vertical and horizontal exploration as shown in the flowcharts in Figure 4.

3.3.1 Results

The results of the implementation of method 1 clearly indicate that the accuracy of this verification algorithm heavily depends on the iteration constraints. The relevant data is showcased in Table 1, where n and t represent the maximum number of iterations and maximum run-time allowed per property, respectively. It is important to consider that the algorithm works on the assumption that a property is true if no counter-example is found, and the majority of the properties we are verifying are true, the accuracy of this algorithm is measured through its accuracy in identifying the false properties. Among 172 false properties, 122 of the false properties were correctly identified in the best test run, achieving 70.9% accuracy. However this test run was the longest in duration (28 hours) showcasing the critical trade-off between accuracy and computation time. Especially for a larger-scale project with millions of properties increasing the accuracy marginally will take a considerable amount of time. Property 175 is a false property flagged

Table 1: Branch and Bound Data

Constraint	n = 1000	n = 7000	n = 10000	t <600s
#Correct Verification	329	343	348	440
Accuracy	0.581%	8.72%	11.6%	70.9%
Time Elapsed	31s	216s	338s	28 hours

true by the 28 hour test run. When ran without any time constraints Branch and Bound takes just over 50 minutes to identify this property as false. This means that at the very least the time limit should be set to over 50 minutes to approach perfect accuracy, which bumps up the worst-case total run time to around 500 hours. Method 2 takes even more time bounding the output and following this the results clearly show that the algorithm fails to identify any of the false properties when the same iteration constraints are applied as Method 1. This is, again, due to the computational challenge arisen by storing exponentially increasing number of partitions. Limited vertical exploration indicates an inferior quality of boundary values and therefore inaccurate results. Method 1 proves to be more efficient and accurate.

3.3.2 Improvements

From Section 3.2.2 it is known that replacing the boundaries generated by IBP with LPB yields more accurate results but increases the computational time for calculating the bounds. However since LPB provides tighter and more accurate bounds, the Branch and Bound algorithm should take a shorter amount of time to flag false properties. When ran on Property 175 with LPB bounds, it took 296 seconds for Branch and Bound to correctly flag the property as false, decreasing the run-time more than tenfold compared to IBP bound. The magnitude of this time reduction indicates that the program will take a longer time to run for the same iteration constraints as in Section 3.3.1 , however will achieve a more accurate result. Linear Programming bounds were used in implementing Method 1 Branch and Bound with a maximum iteration number of 1000 to limit the boundary computation time. The algorithm ran for just over 20 hours, which is a significant increase in run-time compared to the 31 seconds of Interval Bound Branch and Bound with identical constraints. As expected, there is also a significant increase in the accuracy: 500 of the 500 properties were identified correctly by the algorithm. On the contrary, in Section 3.3.1 for a maximum accuracy of 70.9% the program ran for 28 hours. Overall, Linear Bound Programming takes more time to calculate bounds, hence time per iteration is increased compared to Section 3.3.1. However, the boundaries produced are better quality and therefore less iterations are required to correctly verify a property, resulting in a highly accurate verification algorithm with improved time-efficiency.

4 Discussion

Three main verification algorithms were implemented, improved and analysed for this report.A summary table of the results are shown in Table 2. Branch and Bound is the only complete and therefore reliable algorithm that is able to yield an accuracy of 100% but its most time-efficient version still takes 20 hours to achieve this. Lower Bound is an unsound method that achieves highly accurate results when combined with Projected Gradient Ascent, with a run-time of under a second. Interval Bound Propagation's best accuracy is the lowest compared to the other three making it the worst performer. The accuracy required and the size of the data should be considered when implementing a verification algorithm.

Table 2: Summary Data

Method	Lower Bound	Interval Bound	Branch and Bound
Max Accuracy	93.0%	79.2%	100%
Time Elapsed	0.04s	41 mins	20 hours

References

[1] Aws Albarghouthi. Introduction to neural network verification, 2021.

[2] S. Abirami and P. Chitra. Chapter fourteen. In Pethuru Raj and Preetha Evangeline, editors, *The Digital Twin Paradigm for Smarter Systems and Environments: The Industry Use Cases*, Advances in Computers. Elsevier, 2020.

[3] MathWorks(2021). How matlab allocates memory.