1 Task 1: Lower Bound for Neural Network Output

The method outlined in this task is essentially a trial-and-error approach to obtaining a lower bound to prove false properties. It is obvious that not all false properties are guaranteed to be proven false, especially for properties with mostly negative output domains. This means that even with very large values of *k*, which would also be computationally expensive, there is always a chance that this method will miss a false property.

1.1 Average amount of time taken to generate the outputs

In order to avoid having to load the files 500*k* times, we use an outer loop that loops through each file and an inner loop that loops through *k*. We use the *tic toc* function to get the time taken to generate the outputs for each $k \in \{1, 2, 3, ..., K\}$ for a single file, and store the values in a $(K \times 1)$ ***individual_time*** column vector. We then add up all the ***individual_time*** column vectors and divide by 500 element-wise to obtain the average amount of time taken to generate the outputs. When this data was plotted against *k*, there were large spikes present in the plot. Given that the measured times were very short and *tic toc* performs poorly when the code is too fast, the code was ran in a loop and the measured time was averaged for a single run [1]. The data is then plotted to give Figure 1a. Not surprisingly, the average amount of time taken to generate the outputs tends to increase as *k* gets larger. However, the large spikes are still present at the same positions of $k = 28, 59, 76$. This suggests that these spikes are not random events. One possible explanation is that these large spikes are intrinsic to the hardware of the device that the MATLAB code is being run on, due to the hardware-dependent behaviour of *tic toc* [2].

1.2 Average lower bound

We again use a nested loop with an outer loop that loops through each file and an inner loop that loops through *k*. Each time the outputs are computed, we use the *max()* function to determine the largest output, which would be the lower bound computed for that particular value of *k*. In a similar way to what was done in Section 1.1, a ***lower_bound*** row vector contains the lower bounds for each value of *k* for a particular file. The row vectors are then added to one another and divided by 500 element-wise. The resulting row vector is then plotted against *k*, giving Figure 1b.

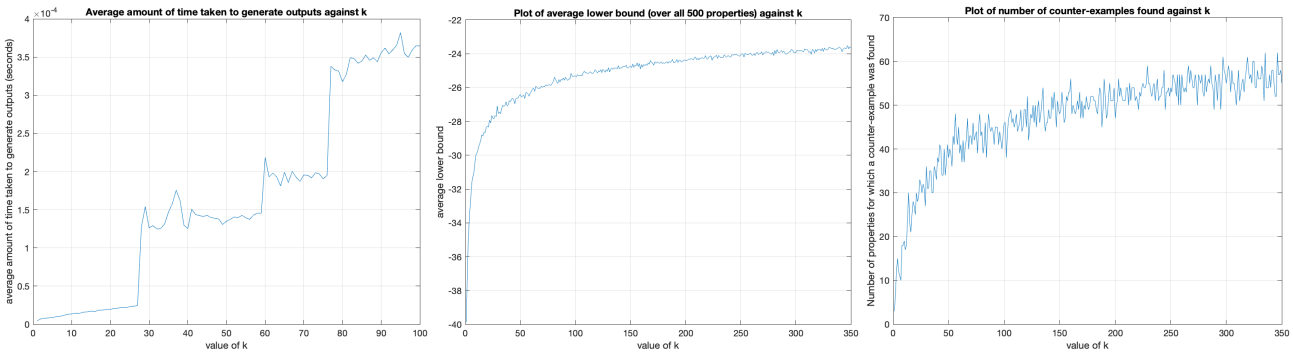
In Figure 1b, the average lower bound increases significantly, before starting to level off at $k = 25$. The fact that the average lower bound is so low for small values of *k* suggests that the output domains of most of the properties are largely, if not entirely, negative. This makes sense since 328 of the 500 properties are true, meaning that the output domains of a majority of the properties are entirely negative. As *k* increases, the computed lower bound for each property starts to approach the upper limit of its respective output domain, which brings up the average lower bound. It should also be noted

that the average lower bound remains negative even for very large values of k, further confirming that most of the properties are true.

1.3 Number of properties for which a counter-example was found

To obtain the number of counter-examples found for each value of k, we again make use of the ***lower_bound*** row vector as well as a ***false_flag*** and a ***total_flag*** row vector (each initialised as a $1 \times K$ zero vector). A '1' is assigned to the k^{th} element of ***false_flag*** if the k^{th} element of ***lower_bound*** is positive. We do this for a single property and add ***false_flag*** to ***total_flag***, before moving on to the next property and repeating the whole process. After looping through all 500 properties, ***total_flag*** can then be plotted against k to obtain Figure 1c.

In Figure 1c, it can be seen that the number of properties for which a counter-example is found generally increases as k increases, which is expected. Since the k inputs are randomly generated, a larger lower bound is more likely to be found if k is large. The random generation of the k inputs also explains the fluctuations in the plot. An increase in k only increases the probability that a counter-example is found for a false property, it does not guarantee that a property for which a counter-example was found before will have a counter-example found again. In addition, it can also be observed that the plot starts to level off at around 50 to 60 proven false properties for larger values of k. Since the total number of false properties is known to be 172, it is implied that it is relatively harder to find a counter-example for the remaining false properties, and that a far larger k is needed to prove all the false properties are indeed false.



(a) Average amount of time taken to generate outputs (seconds) against k

(b) Average lower bound against k

(c) Number of properties for which a counter-example was found against k

Figure 1: Task 1

2 Task 2: Interval Bound Propagation

The modified weights $\mathbf{W}_l^+$ and $\mathbf{W}_l^-$ used in this method are modified in a rather simplistic way that loosens the bound constraints considerably. Hence, we do not expect this method to give very good lower and upper bounds. However, interval bound propagation is computationally inexpensive.

2.1 Average values of the lower and upper bounds

The lower and upper bound for each property is computed using the *interval_bound_propagation* function and cumulatively summed up in order to get the total lower bound and the total upper bound. Dividing both values by 500 would give an average lower bound of -137.882246646741 and an average upper bound of 82.7293950164992. From Figure 1b, it can be seen that the average lower bound for $k = 350$ obtained in Task 1 is slightly larger than -24, which is far larger than the average lower bound computed using interval bound propagation. In fact, the average lower bound computed in Task 1 is larger than that computed in Task 2 even for small values of *k*. Therefore, it seems that the method in Task 1 is superior to interval bound propagation at computing lower bounds.

2.2 Number of properties proved

We initialise two 1×500 vectors (***false_flag*** and ***true_flag***) as zero vectors. *if* statements are used to check the values of the lower and upper bounds of each property. If the lower bound of the i^{th} property is positive, the property is false and a '1' is assigned to the i^{th} element of ***false_flag***. If the upper bound of the i^{th} property is non-positive, the property is true and a '1' is assigned to the i^{th} element of ***true_flag***. The elements of ***false_flag*** and ***true_flag*** are then summed up separately at the end of the loop to give zero proven false properties and 11 proven true properties respectively. This confirms that interval bound propagation is inferior at proving false properties compared to the method in Task 1, which was able to prove some false properties even for small values of *k*. Furthermore, interval bound propagation is only able to prove 11 out of the 328 true properties, which suggests that this method is not too good at proving true properties either. It can be concluded that interval bound propagation is unable to prove many properties because its computed bounds are too loose to be very useful.

3 Task 3: Branch-and-Bound

The branch-and-bound algorithm creates a rooted tree, which is a tree in which one vertex is set as the root [3], and evaluates all the leaf nodes (nodes with no children). In this case, the root is the original input domain $\mathbf{X}_{original}$, and the leaf nodes are the subdomains. The branch-and-bound algorithm finds the leaf node with the highest upper bound and adds two children nodes to it (the subdomain with the largest upper bound is split into two new subdomains). Notice that this means smaller trees are contained within larger trees. This will be useful later.

$$P = \{\mathbf{X}_1, \mathbf{X}_2, \mathbf{X}_3, ..., \mathbf{X}_n\}, \qquad \textbf{\textit{bounds}} = \begin{bmatrix} \bar{y}_1^{min} & \bar{y}_2^{min} & \bar{y}_3^{min} & ... & \bar{y}_n^{min} \\ \bar{y}_1^{max} & \bar{y}_2^{max} & \bar{y}_3^{max} & ... & \bar{y}_n^{max} \end{bmatrix}^T$$

As shown above, we use a cell array *P* in the code to contain the set of the subdomains obtained so

far. We also use a matrix ***bounds*** to store the lower and upper bounds. $\bar{y}_j^{min}$ and $\bar{y}_j^{max}$ are the lower and upper bounds respectively for the neural network output computed when the input is restricted to $\mathbf{X}_j$, where *j* $\in \{1, 2, 3, ..., n\}$. Since we know the method in Task 1 is superior at computing the lower bound, interval bound propagation is only used to compute the upper bound while the Task 1 method (using 20 inputs in this case since the average lower bound in Task 1 starts levelling off at around $k = 20$) is used to compute the lower bound.

After the subdomain $\mathbf{X}_J = [\mathbf{x}_J{}^{min}, \mathbf{x}_J{}^{max}]$ with the largest upper bound is found, $\mathbf{X}_J$ is split into two new subdomains. The lower and upper bounds for the two new subdomains are computed using the Task 1 method and interval bound propagation respectively. The two new subdomains are added to *P* while $\mathbf{X}_J$ is removed. The respective bounds are also added to and removed from **bounds**. Of course, the lower and upper bounds are checked between partitions, and the algorithm is terminated if the property is proven.

3.1 Execute

We want to perform *p* partitions, and obtain the number of verified properties as well as time taken for the branch-and-bound algorithm, and plot the former against the latter. Hence, it is tempting to use a nested loop to loop through each property for each value of *p* in order to find the time taken and the number of proven properties for each value of *p*. However, as mentioned before, smaller trees (less partitions) are contained within larger trees (more partitions). Using a nested loop would involve a total of $(1 + 2 + 3 + ... + P\_max)$ partitions for properties that remain unproven even after *P_max* partitions, where *P_max* is the maximum value of *p*. For properties that get proven before being partitioned *P_max* times, the total number of branchings performed is smaller. How much smaller it is will depend on how early the branching procedure is terminated. Either way, this is a lot of unnecessary calculations and is inefficient as a result. Instead, we should perform *branch_and_bound* on each property just once with a maximum partition limit of *P_max*, and obtain the time taken for the i^{th} branching (for convenience, we define 'branching' to include both the partitioning and the evaluation of the leaf nodes).

To begin, we have to modify *branch_and_bound*. Within *branch_and_bound*, a $(1 \times P\_max)$ vector **time** is initialised as a zero vector. The *tic toc* function is used to measure the time taken for each i^{th} branching, and the time is stored as the i^{th} element of **time**.

If *i* branchings take $T(i)$ time, then the time taken for *i+k* branchings is $T(i) + t(i+1) + t(i+2) + ... + t(i+k)$ where $t(i)$ is the time taken for the *(i)*$^{\text{th}}$ branching. Note that if branching terminates early (property is proven) at $i = I$, the elements of **time** after the *(I)*$^{\text{th}}$ element will be zero. We also know that the time taken for *I* and more than *I* branchings will all be the same (if the property is proven after $T(I)$

has elapsed, running the branch-and-bound procedure for longer than that is meaningless). Hence, we can use the MATLAB function *cumsum* to cumulatively sum up all the time taken for each individual branching from the first branching to the last (since $T(1)$ is trivially equal to $t(1)$). If *I* exists, $T(I)$ will be cumulatively added to zeros, ensuring that the time taken to perform more than *I* branchings is the same as the time taken to perform *I* branchings. This is what we want since branch-and-bound would terminate regardless after *I* branchings. This does mean that we need to add additional output arguments (**time** and *I*) for *branch_and_bound*.

Outside *branch_and_bound*, two (1×500) vectors (**true_flag**, **false_flag**) and two $(1 \times P\_max)$ vectors (**total_time**, **total_proven**) are initialised as zero vectors. As we loop through each file, *branch_and_bound* is called and **time** for each property is added to **total_time**. *if* statements are also used to determine if a property is proven by checking the function output *flag*. If so, a counter is added to the I^{th} element of **total_proven**. Obviously, if a property is proven after *I* branchings, it will also be proven for branchings larger than *I*. Following a similar logic as with **time**, *cumsum* is used to cumulatively sum up the elements of **total_proven**. **true_flag** and **false_flag** are just used to count the total number of proven true and false properties separately as an additional data to include on the plot.

In Figure 2, there is a very sharp increase in number of properties that have been verified, before it starts plateauing and increasing very slowly. In this case, the maximum number of partitions was set to be one million, and it took 4.0137×10^4 seconds (roughly 11 hours) to run the procedure. Yet, there are still 52 properties left unproven. Extrapolating from the plot, it is likely that proving all 500 properties would take far longer than 11 hours. That being said, using branch-and-bound still gives better upper bounds than just using interval bound propagation alone. The same goes for lower bounds with the method in Task 1. However, branch-and-bound is evidently too computationally expensive and we will need to find better ways of computing the lower and upper bounds in order to prove all properties in a much more reasonable amount of time.

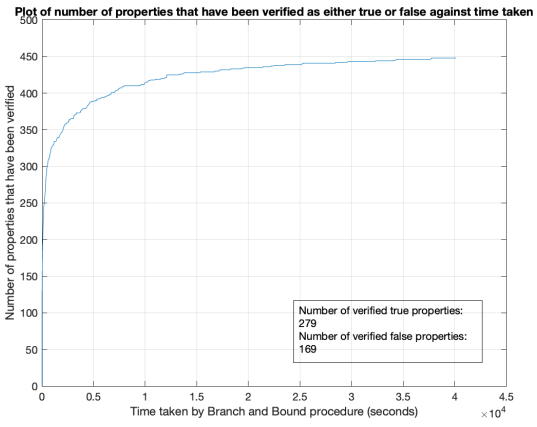


Figure 2: Branch-and-bound

4 Task 4: Projected Gradient Ascent for Lower Bounds

Before coding the function, we have to look into how to to calculate the gradient of *f*, which is the function that describes the neural network. By applying the chain rule, we get:

$$\frac{\partial f}{\partial \mathbf{x}} = \left(\mathbf{W}_5 \frac{\partial ReLU(\hat{\mathbf{z}}_4)}{\partial \hat{\mathbf{z}}_4} \mathbf{W}_4 \frac{\partial ReLU(\hat{\mathbf{z}}_3)}{\partial \hat{\mathbf{z}}_3} \mathbf{W}_3 \frac{\partial ReLU(\hat{\mathbf{z}}_2)}{\partial \hat{\mathbf{z}}_2} \mathbf{W}_2 \frac{\partial ReLU(\hat{\mathbf{z}}_1)}{\partial \hat{\mathbf{z}}_1} \mathbf{W}_1\right)^T$$

Hence, the outputs $\hat{\mathbf{z}}_i$ are computed, and the derivative of the ReLU function is applied on each of them element-wise. The result will then have to be diagonalised to give $\frac{\partial ReLU(\hat{\mathbf{z}}_i)}{\partial \hat{\mathbf{z}}_i}$ [4]. Note that there will be *k* number of $\frac{\partial ReLU(\hat{\mathbf{z}}_i)}{\partial \hat{\mathbf{z}}_i}$ for each value of *i* if we have *k* inputs. While we could loop through *k*, it is better to construct a 3D array of *k* number of $\frac{\partial ReLU(\hat{\mathbf{z}}_i)}{\partial \hat{\mathbf{z}}_i}$ in the pages (MATLAB term for the third dimension). The MATLAB *pagemtimes* function is then used to multiply all the terms together to obtain $\frac{\partial f}{\partial \mathbf{x}}$ for each of the *k* inputs. This is akin to **array broadcasting** in NumPy.

4.1 Average amount of time

Similar to what was done in Task 1, the average amount of time is measured and calculated for each value of *k* and then plotted against *k* as seen in Figure 3a. Not surprisingly, the average amount of time increases as *k* increases, and it appears to be a linear relationship.

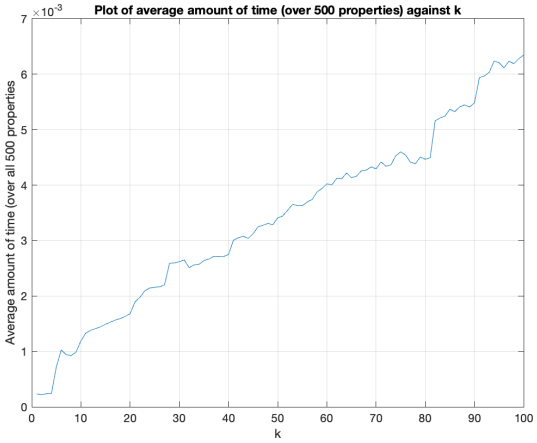
4.2 Average lower bound

The average lower bound for each value of *k* is obtained using a similar approach as that in Section 1.2. In Figure 3b, there is a sharp increase from $k = 0$ to $k = 5$, before the plot levels off. Compared to Figure 1b, the levelling off starts at a lower value of *k*, and the levelling off portion is flatter. This suggests that projected gradient ascent approaches the true maximum outputs of the neural network faster than the method in Task 1.

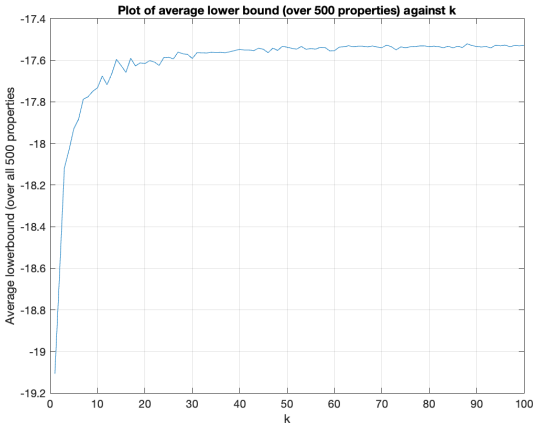
4.3 Number of properties for which a counter-example is found

This can be found using the approach in Section 1.3. From Figure 3c, it can be seen that for values of *k* larger than 17, the plot does not change significantly. Hence, relatively small values of *k* are enough to prove most but not all of the false properties. This is much better than the results in Task 1. As can be seen in Figure 1c, the method in Task 1 is not close to proving all the false properties despite using a larger value of *k*. Figure 3c also suggests that a small minority of false properties are harder to prove than the rest. These harder-to-prove properties likely have a lot of local maxima that are non-positive, making it harder for the gradient ascent method to find a positive maxima. Due to the randomised nature of the inputs, whether the gradient ascent method can find any positive maxima for each of the inputs to a false property is down to luck. Of course, the larger the value of *k*, the higher the chances of generating an input that is close enough to a positive maximum for the gradient ascent method to find it. Additionally, 171 of the false properties are proven multiple times for

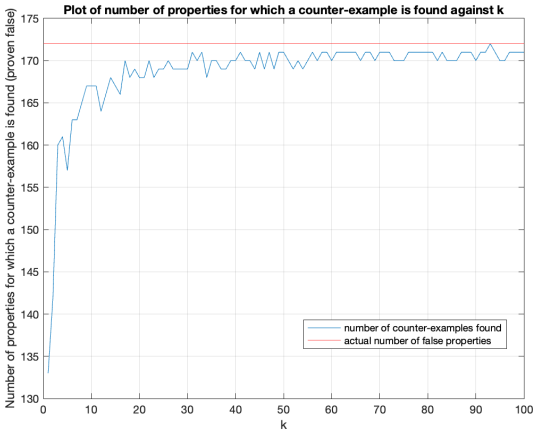
different values of k. All 172 false properties are proven only once, at $k = 93$. This suggests that one property in particular is significantly harder to prove than the others, and that it was only proven false by luck as it was not proven false for $93 < k \leq 100$ as can be seen in Figure 3c. Indeed, re-running the code confirms that not all 172 false properties are guaranteed to be proven false. Hence, a value of k larger than 100 will be required to increase the chances of finding all the false properties, which will be more computationally expensive. The step size and cutoff can also be tweaked to improve the performance of this method. That being said, projected gradient ascent is superior to the method in Task 1 and could be used to calculate lower bounds within branch-and-bound.



(a) Average time (over all properties) against k



(b) Average lower bound against k



(c) Number of properties for which a counter-example is found against k

Figure 3: Projected Gradient Ascent (step size = 0.0005, cutoff = 0.00001)

5 Task 5: Linear Programming Bound

There are two ways to approach this: use linear programming on each layer of the network successively to obtain the bounds for $\hat{\mathbf{z}}_k$, $\mathbf{z}_k$, and output y where $k \in \{1, 2, 3, 4\}$, or to use interval bound propagation to first obtain the bounds for $\hat{\mathbf{z}}_k$, $\mathbf{z}_k$ and then these bounds in the linear programming applied on just the last layer. The latter approach is chosen for this task. Note that it does provide looser bounds but it is also less computationally expensive.

We use *linprog* to minimise

$$[\textbf{zeros}(1, 261), \mathbf{W}_5][\mathbf{x}, \hat{\mathbf{z}}_1, \mathbf{z}_1, \hat{\mathbf{z}}_2, \mathbf{z}_2, \hat{\mathbf{z}}_3, \mathbf{z}_3, \hat{\mathbf{z}}_4, \mathbf{z}_4]^T$$

$$-[\textbf{zeros}(1, 261), \mathbf{W}_5][\mathbf{x}, \hat{\mathbf{z}}_1, \mathbf{z}_1, \hat{\mathbf{z}}_2, \mathbf{z}_2, \hat{\mathbf{z}}_3, \mathbf{z}_3, \hat{\mathbf{z}}_4, \mathbf{z}_4]^T$$

which is essentially minimising $y - b_5 = \mathbf{W}_5\mathbf{z}_4$ and $-(y - b_5) = -\mathbf{W}_5\mathbf{z}_4$ respectively. Note that the variables that minimise $-(y - b_5)$ would maximise $y - b_5$. The inequality constraints matrices are assembled using the approximated linear inequality constraints:

$$\mathbf{z}_k(i) - C\hat{\mathbf{z}}_k(i) \leq -C\hat{\mathbf{z}}_k^{min}(i), \text{where } C = \frac{\hat{\mathbf{z}}_k^{max}(i)}{\hat{\mathbf{z}}_k^{max}(i) - \hat{\mathbf{z}}_k^{min}(i)}$$

$$\hat{\mathbf{z}}_k(i) - \mathbf{z}_k(i) \leq 0,$$

$$\mathbf{z}_k(i) \geq 0.$$

Note that this third inequality constraint is already implied by the lower bounds calculated using interval bound propagation (since the lower bound of $\mathbf{z}_k(i)$ must be non-negative), and hence can be omitted. However, these approximated constraints only apply if $\hat{\mathbf{z}}_k^{min}(i)$ is negative and $\hat{\mathbf{z}}_k^{max}(i)$ is positive. If both are negative, $\mathbf{z}_k(i)$ is just zero. If both are positive, $\mathbf{z}_k(i)$ is equal to $\hat{\mathbf{z}}_k(i)$. Hence, we have to check the signs of every element of $\hat{\mathbf{z}}_k^{min}(i)$, $\hat{\mathbf{z}}_k^{max}(i)$ and modify the corresponding elements in the inequality constraints matrices by making them zero such that $0 * \hat{\mathbf{z}}_k(i) + 0 * \mathbf{z}_k(i) \leq 0$ if the approximated constraints do not hold. The equality matrices are also assembled accordingly such that $0 * \hat{\mathbf{z}}_k(i) + \mathbf{z}_k(i) = 0$ for negative $\hat{\mathbf{z}}_k^{min}(i)$ and $\hat{\mathbf{z}}_k^{max}(i)$, and $\hat{\mathbf{z}}_k(i) - \mathbf{z}_k(i) = 0$ for positive $\hat{\mathbf{z}}_k^{min}(i)$ and $\hat{\mathbf{z}}_k^{max}(i)$. The $\hat{\mathbf{z}}_k - \mathbf{W}_k\mathbf{z}_{k-1} = 0$ (where $\mathbf{z}_0 = \mathbf{x}$) constraints are also included in the equality matrices. All these constraints as well as the lower and upper bounds of the variables already calculated using interval bound propagation are then used within *linprog* to minimise $y - b_5$ and $-(y - b_5)$. As mentioned before, this would give us the minimum and maximum value of $y - b_5$ respectively, which is essentially just the lower and upper bounds of y after adding b_5.

5.1 Average value of the lower and upper bounds

Similar to what was done in Section 2.1, we obtain an average lower bound of -77.602404683211560 and an average upper bound of 14.819882426533335. This is a tighter bound than what was obtained in Task 2, especially the upper bound. Hence, we can expect this method to prove more properties.

5.2 Number of properties proven

Again, we re-use the code from Section 2.2 to show that 126 true properties and zero false properties are proven. While there is no difference in the number of proven false properties, there is

a significant increase in number of proven true properties from 11 in Task 2 to 126. Hence, linear programming should be chosen over just interval bound propagation on its own when it comes to computing upper bounds.

5.3 Incorporate linear programming bounds within Branch-and-bound

The code from Task 3 is modified to use linear programming to compute upper bounds instead of using interval bound propagation. However, it is not used to compute lower bounds given its poor performance at proving false properties. Instead, projected gradient descent from Task 4 is used. Figure 3c shows that the plot levels off at around $k = 20$, so k will be set to this value in this implementation of projected gradient ascent in order to balance between performance and computation cost.

Running the modified code produces Figure 4, which shows that all 500 properties are proven after 1.1607×10^4 seconds (roughly 3 hours). This algorithm only performed a maximum of 48966 partitions. This is vastly superior to the Task 3 algorithm, which was not able to prove all the properties even after 11 hours and a million partitions. It should also be noted that the majority of the properties were proven very early on, with a minority being significantly harder to prove. This is in line with what we observed in earlier tasks.

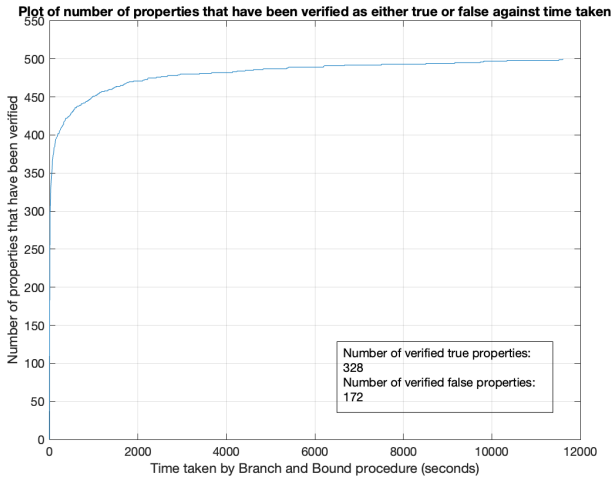


Figure 4: Linear programming bound within branch-and-bound

6 Test bench

We extract the information from *groundtruth.txt* and put it in the vector ***testbench*** such that the i^{th} element is '1' if the i^{th} property is true, and '-1' if false. For all the tasks in this report, ***flag*** vectors with different notations had been used to count the number of proven properties. These vectors can be modified to follow the same notation as ***testbench*** with '1' for proven true properties, '-1' for proven false properties, and '0' for unproven properties. Equating a modified ***flag*** vector with ***testbench*** will give a vector of logical '1's and '0's. Summing up all the elements in this vector should then give the total number of correctly proven properties. Since the number of proven properties for each of the

tasks is already known, we check if the total number of correctly proven properties matches it. If so, the code for that particular task passes the test bench and can be taken to be correct. If there are any wrongly labelled properties (true property gets proven as false or vice versa), the two numbers would not match and the code would then have to be corrected. All the tasks have passed this test bench.

7 Conclusion

The method used in Section 5.3 is the only algorithm that proved all 500 properties within a reasonable amount of time. Hence, it seems that a branch-and-bound algorithm using both projected gradient ascent and linear programming would be best suited for network verification of this kind. However, it should be noted that not all combinations of the algorithms have been tested. The value of k used in projected gradient ascent could also be tweaked in order to achieve better performance. Most importantly, all the algorithms in this report have only been tested on one dataset. More datatsets would be needed to test the robustness of the algorithms. To conclude, a good approach to a verification problem would be to use projected gradient ascent on its own to determine an optimal value of k, and then use that value of k within the Section 5.3 algorithm.

References

[1] "Measure the Performance of Your Code," Measure the Performance of Your Code - MATLAB & Simulink - MathWorks United Kingdom. [Online]. Available: https://uk.mathworks.com/help/matlab/matlab_prog/measure-performance-of-your-program.html. [Accessed: 08-Jan-2022].

[2] M. Knapp-Cordes and B. McKeeman, "Improvements to TIC and TOC functions for measuring absolute elapsed time performance in Matlab," MATLAB & Simulink, 2011. [Online]. Available: https://uk.mathworks.com/company/newsletters/articles/improvements-to-tic-and-toc-functions-for-measuring-absolute-elapsed-time-performance-in-matlab.html. [Accessed: 17-Dec-2021].

[3] E. A. Bender and S. G. Williamson, "Lists, Decisions and Graphs. With an Introduction to Probability," 2010. [Online]. Available: https://cseweb.ucsd.edu/ gill/BWLectSite/Resources/LDGbook-COV.pdf. [Accessed: 19-Dec-2021].

[4] K. Clark, "Computing Neural Network Gradients." [Online]. Available: https://web.stanford.edu/class/cs224n/readings/gradient-notes.pdf. [Accessed: 30-Nov-2021].