Candidate Number: 1253748

B1 Engineering Computation: Project B

Neural Network Verification

Preamble

Context

This project is concerned with developing a complete verification method for the 500 neural networks described by the 'Collision Detection' dataset. We can imagine one realization of the problem to be as follows: 500 autonomous vehicles are each equipped with 6 sensors, the readings from which are used as the inputs to an MLP which outputs a single number. The MLP ('property') for each vehicle has undergone supervised training by backpropagation to determine its weights, W, and biases, b, and when tested each vehicle appears to successfully detect when a collision is about to take place. Further to this, a range of inputs xmin ≤ x ≤ xmax from the sensors is determined – any input within these constraints should result in an output, y ≤ 0, which means a collision is imminent. Though it seems each vehicle works as intended, we need to have certainty that there are no possible inputs (within our constraints) that return a positive output – this would lead to the catastrophic failure of an impending collision *not* being detected. Our aim is to formally prove whether or not each vehicle is possible of such an error. I found approaching the project with a real-world problem in mind helpful to consider different priorities, which we will return to.

Development, Testing and Verification

To carry out our aim, we need to implement successively better methods for determining upper and lower bounds to the maximum possible output $y*$ - if $y^*_{max} \leq 0$, we know that the property is true. Similarly, if $y^*_{min} > 0$, we know the property is false.

I employed the following iterative development method, looping back as necessary:

1. plan (pseudocode) → 2. implement → 3. test → 4. verify → 5. analyse results → 6

I decided to use the MATLAB Live Editor for my main scripts as this helped me more dynamically test and visualize my results. This was particularly helpful for unit testing functions against a single property or testing a particular line of code within a function. Where possible, I ensured that arrays were preallocated, loops vectorized and short-circuit logic conditions utilised to improve efficiency.

To begin the project, I wrote a function `get_groundtruth`, which takes the location of the groundtruth.txt file for our dataset and returns two variables – `gt`, a 500x1 logical column vector indicating which properties are true/false, and `true_count`, an integer representing the total number of true properties (328). These would allow me to check my results as the project progressed.

The methods for task 1-3 are well documented in the handout, so I will describe them sparingly.

Task 1: Lower Bound Approximation using Random Inputs

We start with a basic method to compute a valid lower bound of y* - randomly generate k inputs, compute the respective outputs, and store the largest values found for each property. If any of these values are positive, we have found a counterexample, which is sufficient to prove a property is false.

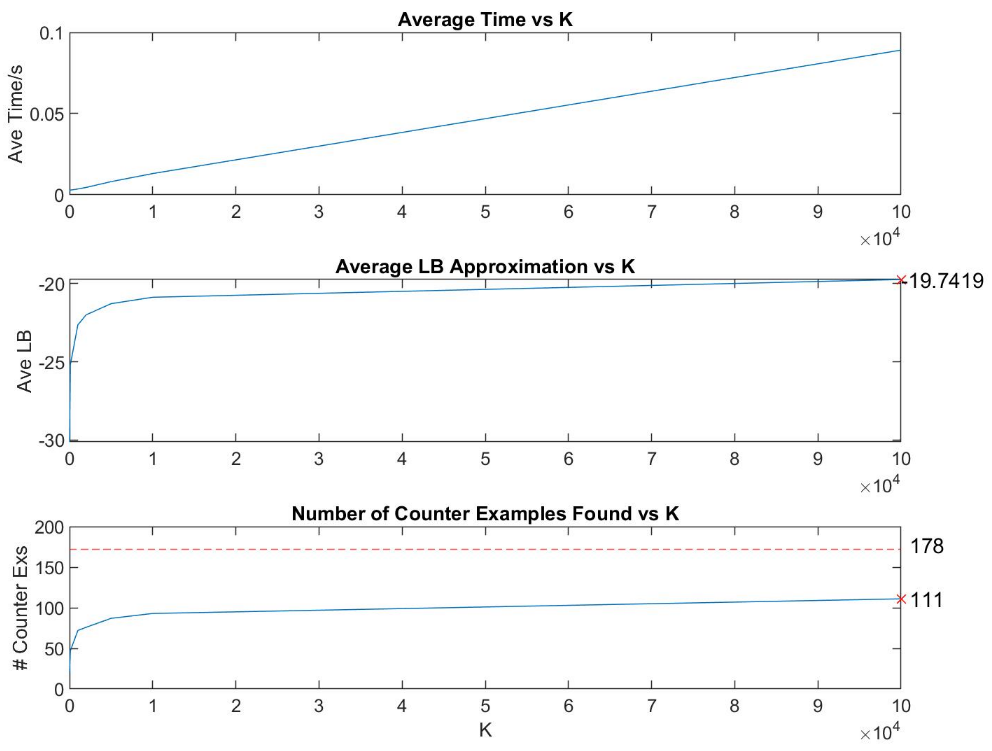

Figure 1: Task 1 Results

From Fig. 1 we can see that the time cost per property is directly proportional to the size of k, or of order O(k). However, we see exponential decay in the rates at which the average LB approximation to `y*` improves and the number of counterexamples found increases. While we could theoretically generate enough random inputs to eventually find a counterexample for all 178 false properties, this is not computationally feasible.

Unless otherwise stated, I used k = 1000 for future tasks (at least as a starting point), as it seemed to be suitably balanced between performance and time required.

Task 2: Interval Bound Propagation

We next turn to interval bound propagation – an incomplete method (see p.6-8 of the handout) to determine upper and lower bounds of the entire output y. This means our ymin and ymax values are both valid, albeit very loose, lower/upper bounds to y*. I found the upper/lower bounds (averaged across all 500 properties) to be: `Ave_U = 82.7294` `Ave_L = -137.882`

These results were able to verify 11 properties as true and 0 false, taking a total time of 1.2929 seconds (averaged over 10 trials), or 0.0026 seconds per property. We see that the average lower bound is much worse than the one calculated in task 1, and that not many properties can be verified using this method alone.

Task 3: Branch-and-Bound

We progress to our first attempt at implementing a *complete* method for NN verification: branch-and-bound (see p.9-11 of the handout). Here `max_iter` refers to the maximum partition size.

Figure 2 shows two possible ways of arranging the partition – I decided to use the left arrangement, as using a 3-dimensional set of 'frames' would be more intuitive to handle than indexing in sets of 6.

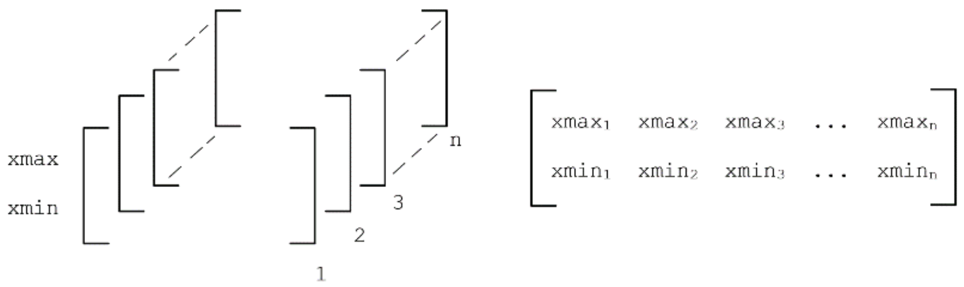

Figure 2: Partition Arrangements for BnB

I was interested to see the effects of varying `max_iter` and k on the branch-and-bound method – you can see the results of these tests in Fig. 3. Increasing either variable improved the results, but they both seemed to indicate that there were two strata of properties – a set amount (comprising mostly false properties) that were easily verified, and the rest which all demanded significantly more iterations.

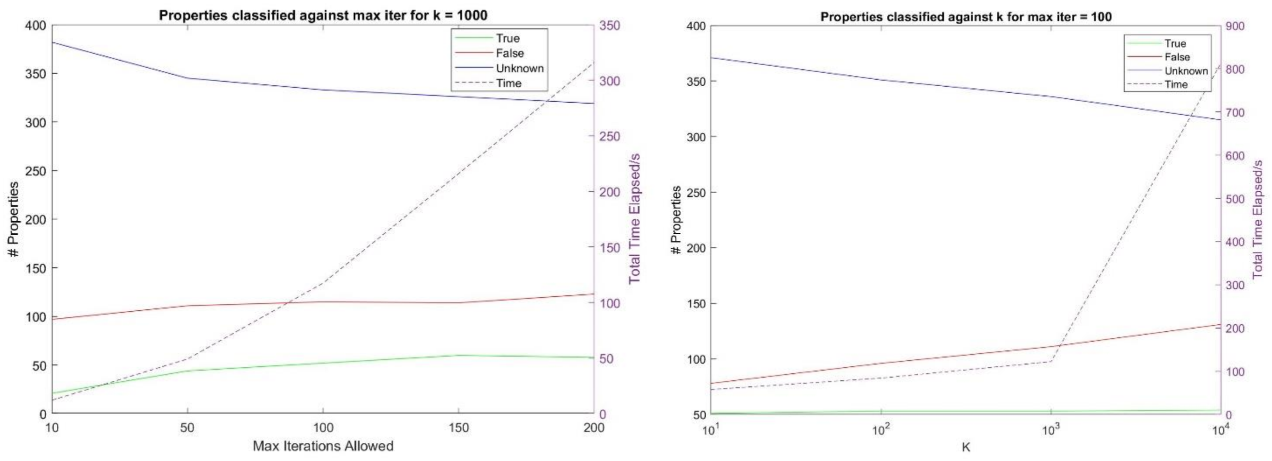

Figure 3: BnB Tests (L-R) Fixed k & varying max_iter, Fixed max_iter, varying k

I found that confusion charts were a simple but effective way to compare my results to `gt` for further analysis. Figure 4 shows two examples at different values of `max_iter`.

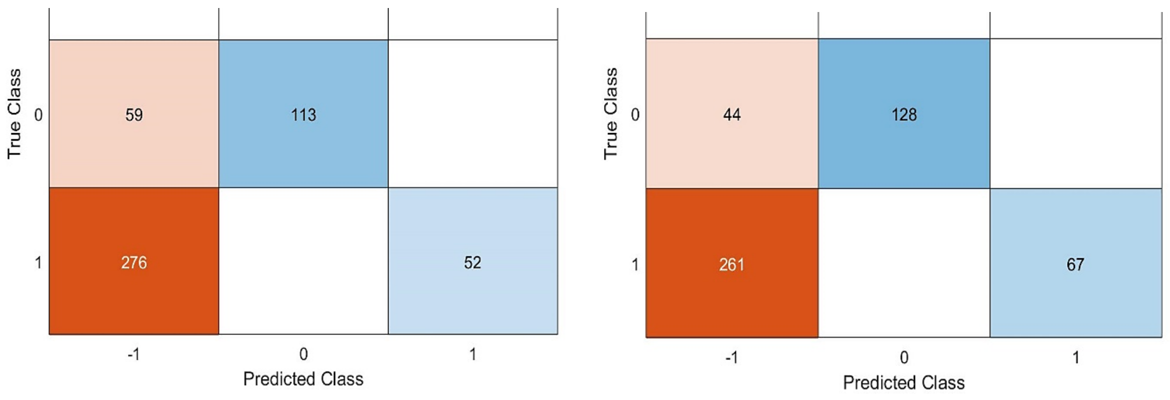

Figure 4: (L-R) Confusion Charts for `max_iter` = 100 and 500 at fixed k = 1000

We are able to quickly see that this algorithm is better at verifying false properties than it is true, and that the greatest source of error is from true properties reaching `max_iter` before being verified. The limited improvement we see between the max partition sizes further backs up the results from Fig. 3 – that there are quite a few properties that can be classified quickly, but each successive improvement demands more and more time. Most importantly, we also see two empty squares in each chart, indicating that there are no instances of a false property being identified as true and vice versa. This is a good indication that the method is working as intended, even if it would take too long to verify all properties (projected in the order of days).

We now have some confidence that we have a complete method – relating back to our example problem, this means we are now able to determine whether or not each of our vehicles is truly safe. The question now is whether this outcome is sufficient: for a company, a few days (or even faster with better computers) in the development stage may be insignificant. That being said, our MLP is extremely simple, we only have 500 properties to verify and if a software update required in-field verification, downtime of a few days for a customer would be unacceptable. We will therefore see what further improvements we can make.

Task 4: Projected Gradient Ascent

We turn to projected gradient ascent, a method to help improve our lower bound of `y*`. We want to implement a method similar to that used in Task 1, but with a refined set of inputs. We do this by updating our inputs according to equation 10 in the handout. The main difficulty was in figuring out how to determine the gradient of the NN function; the key was understanding that the ReLU activation function is applied elementwise, which leads us to a derivation:

$$f(x; W, B) = W_5(ReLU(W_4(ReLU(\dots) + b_4) + b_5$$

$$let\ D_l = \frac{d}{dx} ReLU(W_l z_{l-1} + b_l) = \frac{d}{dx} ReLU|_{W_l z_{l-1} + b_l}$$

$$\frac{df}{dx} = W_5 \cdot D_4 * W_4 \cdot D_3 * W_3 \cdot D_2 * W_2 \cdot D_1 * W_1, \quad D_l = \begin{cases} 1, & W_l z_{l-1} + b_l > 0 \\ 0, otherwise \end{cases}$$

Where **·** and * represent elementwise and matrix multiplication respectively. My function `projected_gradient_ascent` employs a while loop with two exit conditions – the maximum number of allowed iterations being reached, or the maximum absolute difference between successive inputs x_t being smaller than a specified tolerance.

```
while ((counter < maxiter) && (max(max(abs(xt-xtold))) > tol))
```

Within this, the values of D_l for each layer are generated by computing the neuron values for each layer, `zhat`, and using a logical condition to get the desired matrix:

```
ReLU_derivatives{l} = (zhat > 0).';
```

The gradient is then calculated as defined above, making sure to multiply from left to right to ensure easy dimension matching (and no use of repmat or bsxfun), and the successive inputs calculated. As these can lie outside of our box constraints, we are required to project, or clip, these values: `xt = max(xmin,xt); xt = min(xmax,xt);`

When we reach a sufficient exit condition, we compute the outputs for our refined inputs and check for counterexamples.

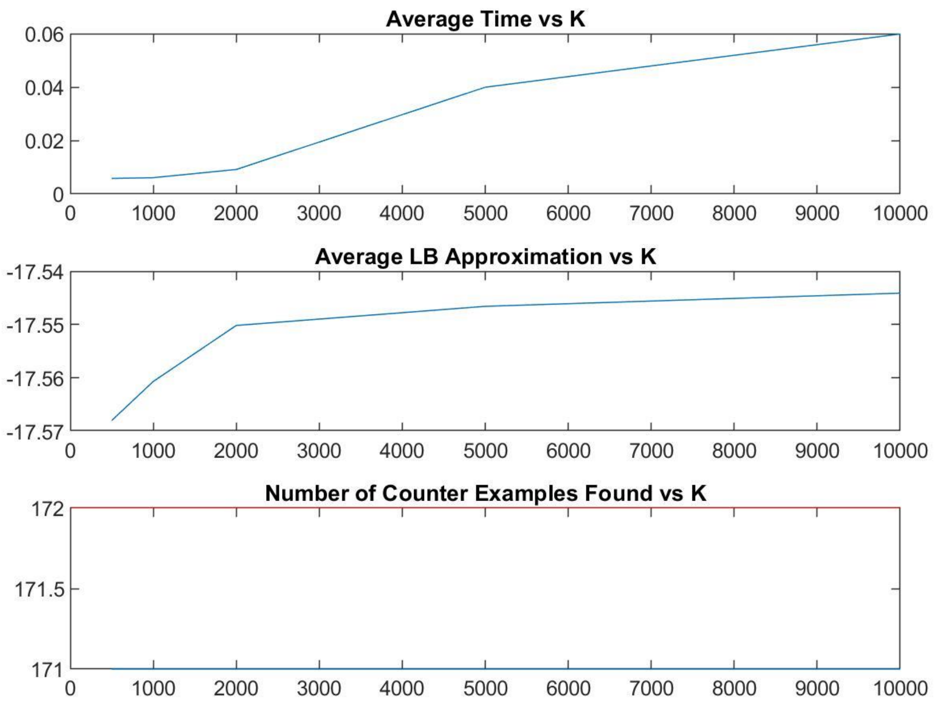

Figure 5: Projected Gradient Ascent Results

Using `tol` = 0.1 and η = 0.01, I found that this combined method very quickly verified 171 out of 172 false properties. Crosschecking with groundtruth, these were all correct – only property 309 failed to generate a counterexample. This could be due to an inappropriate step-size leading to the gradient ascent method being stuck in a local maximum, or the tolerance being too high. Further experimentation only led to significantly longer computation times with little improvements to results – I decided to leave this property to be solved by a different method and prioritise the speed at which the other false properties could be identified. Here we see again the relative ease at which we can verify the false properties. With the problem of testing our vehicles, if we are sufficiently confident in our ability to detect such properties, it *may* be decided that it is enough for some tests to only search for these and assume the other properties are true, even if it is difficult to prove that.

Task 5: Linear Programming Bound

Having improved on our lower bound approximation, we try to improve on our upper bound. We can do this by solving a set of linear programming problems using the MATLAB function `linprog` – for each neuron, this will find the optimal values of the neurons in the previous layer in order to minimise its value. We see the form of the function here:

`x = linprog(f,A,b,Aeq,beq,lb,ub),` which finds $$\min_x f^T x \text{ such that } \begin{cases} A \cdot x \leq b, \\ Aeq \cdot x = beq, \\ lb \leq x \leq ub. \end{cases}$$

Therefore, setting our output variables `x = [z₀; ẑ₁; z₁; ẑ₂; … ẑₗ; zₗ]`, we get

`fᵀ = [zeros(1,size_f), W{l}(t,:)]` where l and t give the layer and number of the current neuron and size_f is used to ensure dimensions match.

We then need to formulate matrices/vectors to contain our inequality, equality, and bound constraints.

I started by setting the lower and upper bounds of z_0 and $\hat{z}_0$ as defined by the box constraints of the property. The constraint matrices A, B, Aeq and beq are initialised as empty arrays, and the vectors `lb` and `ub` are defined as the bounds for z0. I defined a vector to hold the size of each layer, `sizes,` to help match dimensions, which is easily found from W/b. Each matrix is then progressively built layer-by-layer to be used for the appropriate function. As the first layer does not yet involve a ReLU, no further conditions need to be imposed on the problem. For every following layer, vectors `lb` and `ub` are each extended by two sets of columns to represent the new $\hat{z}_l$ and z_l.

Matrix `Aeq` and vector `beq,` are used to represent our equality constraint $\hat{z}_k = W_k z_{k-1} + b_k$. The matrix is expanded using the MATLAB function `blkdiag` and an extra array of zeros added to rightmost part which corresponds to z_a. The vector, `beq` is extended by the new biases for $\hat{z}_k$.

An equivalent method is used to create the inequality constraint matrix/vector A/B. Before we can do this, we are first required to linearise the ReLU function used in our MLP. Fig. 6 shows a visualisation of this, with the three possible scenarios we might face. The ReLU function is shown in red, and the blue line is the linear constraint approximation. For the second scenario (going from left to right) we see that if $\hat{z}^{max} \leq 0$, the gradient of this line should be zero – similarly, if $\hat{z}^{min} > 0$, the gradient should be set to 1. This can be calculated as follows:

```
grad = zmax_hat./(zmax_hat-zmin_hat);
grad(zmin_hat>0) = 1; grad(zmax_hat<=0) = 0
```

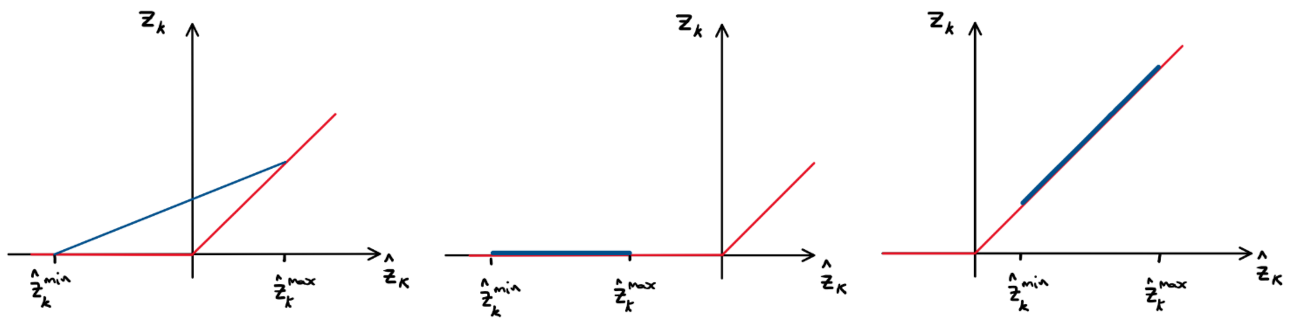

Figure 6: Visualisations of our Different Inequality Constraints

Now that we have all the pieces, we can run through each layer and run two sets of `linprog` evaluations – a minimisation to find the LB approximation and a maximisation (by replacing f with -f) to find the UB. It is important to note that the biases for each layer can be added after the `linprog` step has run as these will not affect the optimisation, and z^{min} and z^{max} can be defined as the ReLU of $\hat{z}^{min}$ and $\hat{z}^{max}$ for the next layer.

Running my `linear_programming_bound` function on all properties took an average of 3.9174 seconds per property, with: `Ave_U = -13.2202` `Ave_L = -58.7168`

Like with Task 2, our lower bound is still quite loose, but we see a huge improvement on the value for our upper bound. This method verified 260 true properties and no false, which is a significant improvement on our previous attempts. I then created a duplicate function, `linprogbound_finalonly` to use interval bound propagation up until the penultimate layer and then use `linprog` for the final set of problems. As expected, this was much quicker (averaging 0.85s per property) but only verified 126 properties – still a good result relative to the time taken.

I then attempted to modify my branch-and-bound function (referred to as modified BnB 1) to use `linear_programming_bound` in place of `interval_bound_propagation`. Even with a low `max_iter` (around 25) this method took so long that I had to quit before the execution could finish. Fig. 7a and 8. show results for the first 250 properties using this method. It performed very well, and we see it taking variable times for properties rather than many all timing out.

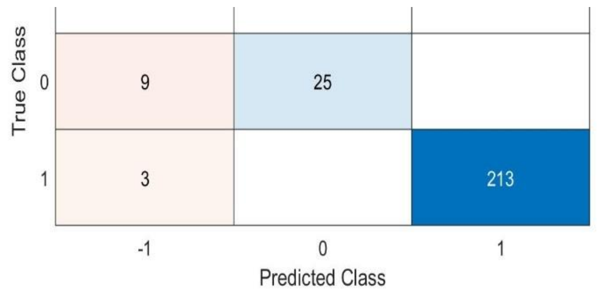

Figure 7a: Confusion Chart for Modified BnB 1

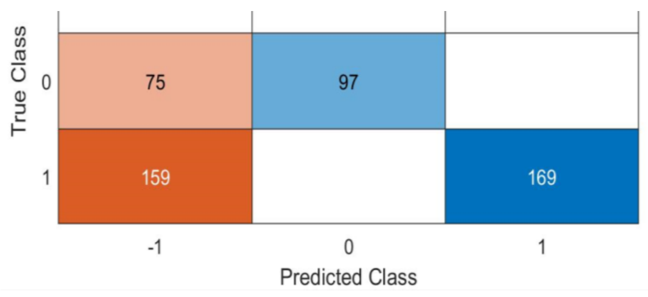

Figure 7b: Confusion Chart for Modified BnB 2

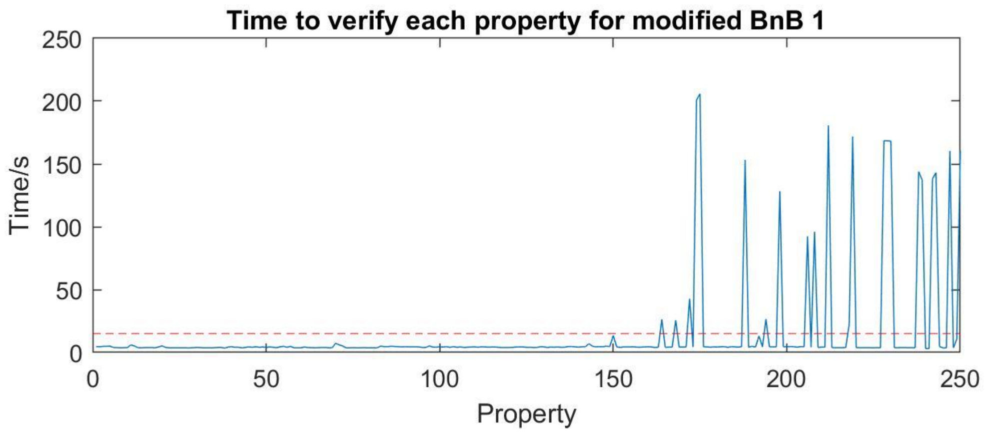

Figure 8: Time to Verify each Property for Modified BnB 1 (first 250 properties only)

I then tried a second approach, using `linprogbound_finalonly` in place of `interval_bound_propagation`. Results can be seen in Fig. 7b and 9 – this was much faster and performed well relatively well true properties but performed worse at identifying false properties compared to the results of Fig. 4 – as the algorithm for identifying counterexamples should be the same between these, I believe the source of this difference is the lower value of `max_iter` used (50).

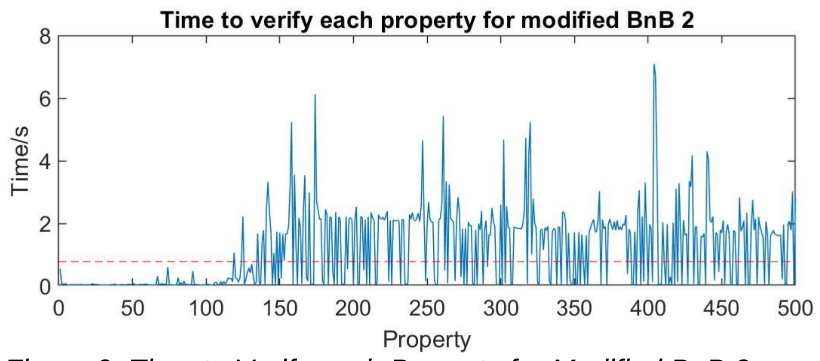

Figure 9: Time to Verify each Property for Modified BnB 2

Further Considerations

Having made considerable improvements in our upper and lower bound approximations to `y*`, we can plan a theoretical combined method which will verify all 500 properties in the least amount of time. We can summarise our investigation results in a table:

Method	*Projected Gradient*	*Linprog Full*	*Linprog Final*	*BnB 1*	*BnB 2*
Time/property/s	0.04	3.9	0.85	20	0.9
Num True	0	260	126	213	169
Num False	171	0	0	25	97

Using these results and parameters from experiments throughout the report, I suggest:

1. For each property:
    a. apply projected gradient ascent to a set of (k=) 1000 random inputs to try to find a counterexample.
    b. Then, use full linear programming bound to try to verify the problem as true.
2. While any remaining property remains unverified, progressively attempt:
    a. modified branch-and-bound 2 with a maximum of 50 iterations.
    b. modified branch-and-bound 1 with a high (>100) number of maximum iterations.

Final Note

Hopefully, this approach would allow us to prove all of our imagined 'vehicles' to be safe in the most efficient way possible – although the problem may not be time sensitive, minimising time, cost and energy should always be prime considerations for engineers.