B1 Computational Project Report

Neural Network Verification

Candidate Number: 1044559

Michaelmas 2021

1 Introduction

Throughout this document we will refer to the Neural Networks (NN) from the collision detection dataset [2] from which the weights and biases have been provided as **examples**. We will **classify** examples as 1 (true) if it is proven that there is no possible output from the network that is positive, and 0 (false) if it is proven that there exists at least one positive output. The function `get_ground_truth` was written to help verify results given by the tasks. The function uses the `groundtruth.txt` file given and converts this into a 500×1 vector in which a 1 and 0 are have the same meanings as above. We will say an output has been **correctly classified** or **verified** if the classification of a given example is the same as the ground truth. We will define y^* as the maximum possible output from the network given the bounded inputs. The computations for each example are independent of one another and thus are prime candidates for parralleisation. We used the `parfor` loop from the `matlab` parralleisation toolbox to quarter computation time. We were able to test four examples at once since the computer we used had four cores.

2 Tasks

2.1 Task 1 - Random Sampling

2.1.1 Method

Function `generate_inputs` This function is expressed as follows:

```
X = (xmax-xmin)'.*rand(6,k) + xmin';
```

Function `compute_nn_outputs` This function works much like **Algorithm 1** from [3] however, we utilise a for loop iterating through the integers $1, 2, \ldots, L-1$ rather than a while loop.

Script The script for Task 1 is described by **Algorithm 1**. It is notable that the $\boldsymbol{y}$ vector is initialised containing $-\infty$. This is done as we are using the maximum function to update it and thus, if it were intialised as zeros, negative lower bounds would never be registered.

Algorithm 1 Task 1 script

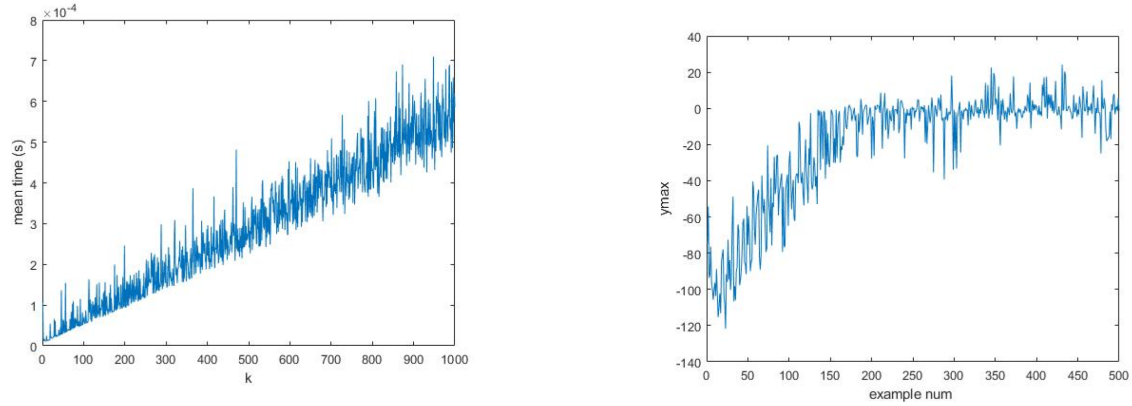
```
Initialise T              ▷ Representing time. Initialised as a matrix of zeros
Initialise y              ▷ Representing the lower bound on y*. Initialised as a vector of −∞
for each example do
    load in weights and biases
    for k ∈ 1, 2, ..., k_max do
        X ← generate inputs
        compute nn outputs
        update y with maximum y across all k and previous maximum for this example
    end for
end for
```

2.1.2 Results

The plots from Task 1 are shown in figure 1a and figure 1b. Figure 1a shows the mean time taken linearly increasing with k. There is a signifcant amount of noise present even when there is no other applications running in the foreground. We obtained a mean lower bound on y^* of -19.2 and with $k_{max} = 1000$ were able to correctly classify 125 out of a total of 172 possible examples as false.

2.1.3 Discussion

This method is very fast, taking less that 10^{-3}s with $k = 1000$, and produces a good quality lower bound on y^*. These small times are responsible for the noise in figure 1a, as the time taken for background processes on the machine becomes significant. Due to its random nature, it is not guaranteed to generate a good quality bound on y^*, and in turn a counter example for an example classified as false. It is not capable of classifying an example as true. Although this method alone is incomplete for calssifying examples, it is very useful during the `branch_and_bound` procedure in Task 3. Unlike other methods which find a lower bound on y, it finds a lower bound on y^*. This helps to raise the lower bound faster



(a) k against the mean time in seconds for the calculation of the neural network outputs across all examples. With $k_{max} = 1000$

(b) The value of the lower bound on y^* against the example number.

Figure 1

and speed up convergence. It is notable that the bounds on y^* as opposed to y allow us to classify the example and thus are the bounds we are more interested in.

2.2 Task 2 - Interval Bound Propogation

2.2.1 Method

Function `interval_bound_propogation` This function is best described by **Algorithm 2**.

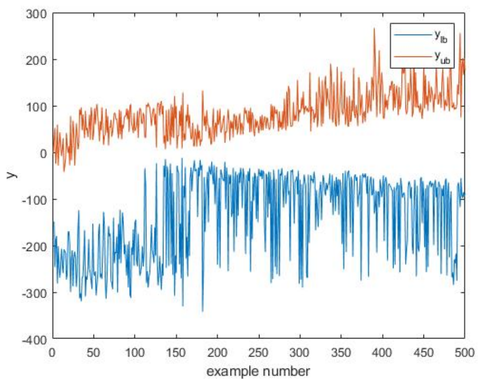
Algorithm 2 Interval Bound Propogation

Initialise $\boldsymbol{z}_{min}$ as $\boldsymbol{x}_{min}^T$
Initialise $\boldsymbol{z}_{max}$ as $\boldsymbol{x}_{max}^T$
for $l \in 1, 2, \ldots, L-1$ **do** ▷ The last layer does not have a ReLU
  $\boldsymbol{W}^- \leftarrow min(0, \boldsymbol{W}_l)$ ▷ min operates elementwise
  $\boldsymbol{W}^+ \leftarrow max(0, \boldsymbol{W}_l)$ ▷ max operates elementwise
  $\boldsymbol{z}_{min}^{temp} \leftarrow \boldsymbol{z}_{min}$
  $\boldsymbol{z}_{min} \leftarrow max(0, \boldsymbol{W}^+\boldsymbol{z}_{min} + \boldsymbol{W}^-\boldsymbol{z}_{max} + \boldsymbol{b}_l)$
  $\boldsymbol{z}_{max} \leftarrow max(0, \boldsymbol{W}^+\boldsymbol{z}_{max} + \boldsymbol{W}^-\boldsymbol{z}_{min} + \boldsymbol{b}_l)$
end for
$\boldsymbol{W}^- \leftarrow min(0, \boldsymbol{W}_L)$
$\boldsymbol{W}^+ \leftarrow max(0, \boldsymbol{W}_L)$
$\boldsymbol{z}_{min}^{temp} \leftarrow \boldsymbol{z}_{min}$
$z_{min} \leftarrow \boldsymbol{W}^+\boldsymbol{z}_{min} + \boldsymbol{W}^-\boldsymbol{z}_{max} + \boldsymbol{b}_L$
$z_{max} \leftarrow \boldsymbol{W}^+\boldsymbol{z}_{max} + \boldsymbol{W}^-\boldsymbol{z}_{min} + \boldsymbol{b}_L$
$y_{min} \leftarrow z_{min}$
$y_{max} \leftarrow z_{max}$

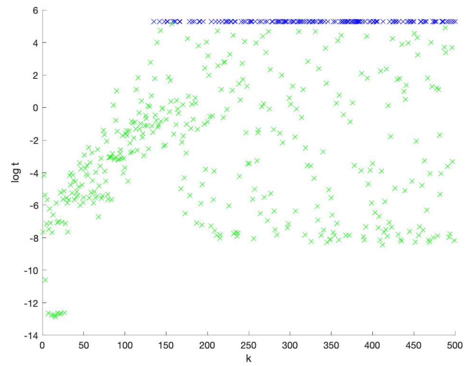
Script The script for Task 2 is described by **Algorithm 3**. It is notable that $\boldsymbol{Y}$ is initilaised as a 2×500 matrix as each column is the vector $\left[\begin{smallmatrix} y_{min} \\ y_{max} \end{smallmatrix}\right]$.

Algorithm 3 Task 2 script

Initialise $\boldsymbol{Y}$ ▷ Initialised as a 2×500 matrix of zeros
for each example **do**
  load in weights and biases
  interval bound propogation
  update $\boldsymbol{Y}$ matrix with y_{min} and y_{max}
end for



(a) The upper and lower bounds on y as obtained by `interval_bound_propogation` in Task 2 against example number



(b) The log of the time taken for the `branch_and_bound` procedure to converge in Task 3 against the example number. Examples classified correctly are shown in green and examples which did not converge are shown in blue.

Figure 2

2.2.2 Results

The plot for Task 2 is show in figure 2a. We obtained a mean lower bound on y of -137.9 and a mean upper bound on y of 82.7. We were also able to correctly classify 0 examples as false and 11 examples as true.

2.2.3 Discussion

The `interval_bound_propogation` method produces valid bounds, however, they are very loose as shown by the small number of examples for which we obtained a classification and the poor lower bound in comparison to the `random_sampling` in Task 1. This is becuase the bounds produced are on y rather than y^* i.e. the lower bound is a bound on the lowest possible output of the neural network. The upper bound of y is the same as that of y^*.

2.3 Task 3 - Branch and Bound

2.3.1 Method

Function `random_sampling` This works the same as Task 1, however, it has been reformulated into a function outputting the maximum output generated.

Function `branch_and_bound` This function works much like **Algorithm 1** from [1] however, because our criteria for example classification are opposite, we maximise the lower bound rather than minimise the upper bound. We use `interval_bound_propogation` to generate the upper bounds and the maxmimum lower bound from `random_sampling`, and `interval_bound_propogation` as our lower bound. It is possible that the lower bound generated is greater than the upper bound. When this occurs we take the upper bound of the subdomain to be the same as the lower bound. In place of the `pick_out` function we choose the domain with the maximum upper bound. This domain is removed from the set and split into subdomains. In place of the `split` function we cut along the longest edge as stated in [3]. As we are only looking for a classification we followed the recommendation of [1] and set the global lower bound to zero. It is thus uncessary to prune domains as any situation in which this would be necessary would be a counter example and so we return a false classification. Due to some examples taking a very long time to run we set a limit for each example. If the result did not converge within this time we set the flag to -1 to indicate there is no classification.

Script In order to evaluate `branch_and_bound` we first initialise our $\boldsymbol{y}_{flags}$ and $\boldsymbol{t}$ vectors. Then for each example we: load in the weights and biases; run the `branch_and_bound` procedure while recording the time it takes; record the output into the $\boldsymbol{y}_{flags}$ vector and the time taken into the $\boldsymbol{t}$ vector; and finally we verify our outputs and plot the graph.

2.3.2 Results

The plot for Task 3 is shown in figure 2b. Out of the 500 examples 367 converged and we were able to verify all of these outputs. We used a time limit of 200 seconds for each example, $\epsilon = 0.1$ and $k = 300$ for `random_sampling`. The script was left running overnight and took around 8 hours to complete.

2.3.3 Discussion

The `branch_and_bound` procedure provides a very flexible and powerful method for classifying examples. The main constraint on the rate of convergence is the quality of the bounds on y^* calculated for each subdomain. When the bounds have converged we have classified the examples as tentatively true, as it is likely the upper bound will continue to decrease below zero. However, it is possible that $0 < y^* < \epsilon$. In this case our tentative classification would be incorrect. The probability of this occuring decreases with ϵ but the computation time increases. We chose our value of ϵ as it is small enough to allow correct classification of most examples, but not to prohibitively increase computation time. We chose our value of k to balance the quality of the bounds generated and the computation time.

2.4 Task 4 - Projected Gradient Descent

2.4.1 Method

Function `projected_gradient_ascent` This function takes the $n \times m$ matrix $\boldsymbol{X}$ and for each column $\boldsymbol{x}$ calculates an updated version $\boldsymbol{x}^{new}$. The calculation of the gradient wrt $\boldsymbol{x}$ for this neural network is shown in **Equation 1**. We iterate through the neural network where in each layer we calculate a value of $\boldsymbol{z}$ and track the gradient at this layer i.e. calculate $\frac{\partial \boldsymbol{z}_l}{\partial \boldsymbol{x}}$. We then update $\boldsymbol{x}$ with the calculated gradient at a given learning rate η. Finally we take take $x_i^{new} = min(x_i^{new}, x_i^{max})$ and $x_i^{new} = max(x_i^{new}, x_i^{min})$ to keep $\boldsymbol{x}^{new}$ within the domain. We repeat this for n iterations for every $\boldsymbol{x} \in columns(\boldsymbol{X})$.

$$\begin{aligned}
\frac{\partial y}{\partial \boldsymbol{x}} &= \frac{\partial y}{\partial \boldsymbol{z}_4}\frac{\partial \boldsymbol{z}_4}{\partial \hat{\boldsymbol{z}}_4}\frac{\partial \hat{\boldsymbol{z}}_4}{\partial \boldsymbol{z}_3}\frac{\partial \boldsymbol{z}_3}{\partial \hat{\boldsymbol{z}}_3}\frac{\partial \hat{\boldsymbol{z}}_3}{\partial \boldsymbol{z}_2}\frac{\partial \boldsymbol{z}_2}{\partial \hat{\boldsymbol{z}}_2}\frac{\partial \hat{\boldsymbol{z}}_2}{\partial \boldsymbol{z}_1}\frac{\partial \boldsymbol{z}_1}{\partial \hat{\boldsymbol{z}}_1}\frac{\partial \hat{\boldsymbol{z}}_1}{\partial \boldsymbol{x}} && \text{where } \boldsymbol{x} \in \mathbb{R}^n \text{ and the } n \times n \text{ matrix,} \\
&= \boldsymbol{W}_5 R'(\hat{\boldsymbol{z}}_4)\boldsymbol{W}_4 R'(\hat{\boldsymbol{z}}_3)\boldsymbol{W}_3 R'(\hat{\boldsymbol{z}}_2)\boldsymbol{W}_2 R'(\hat{\boldsymbol{z}}_1)\boldsymbol{W}_1 && R'(\boldsymbol{x}) = \begin{cases} 0, & \text{if } i \neq j \\ 0, & \text{if } x_{i,j} \leq 0 \text{ and } i = j \\ 1, & \text{if } x_{i,j} > 0 \text{ and } i = j \end{cases} \\
&= \boldsymbol{W}_5 \prod_{k=4,3,2,1} R'(\hat{\boldsymbol{z}}_k)\boldsymbol{W}_k \qquad (1)
\end{aligned}$$

It is notable that the calculations for each column $\boldsymbol{x}$ are independent of each other and thus ideal candidates for parralleisation. We used the parfor loop from parralleisation toolbox in matlab to run these calculations in parrallel.

Script This is structured much like Algorithm 1 however, before computing the NN outputs we use `projected_gradient_ascent` to refine the inputs.

2.4.2 Results

The plots for Task 4 are shown in figures 3 and 4. We can see in figure 4a that the mean time for calculation increases roughly linearly with the value of k_{max}. However, with parralleisation we saw calculation times fall by a factor of 4, the same as the number of cores on our machine, allowing us to try for higher values of k_{max}. We see a fall of computation time by roughly a factor of four when parralleisation is implemented, with the time for $k = 30$ dropping from 1.17s to 0.36s. We found the mean lower bound on y^* to be -17.7 and were able to correctly classify 169 out of 172 false examples. We used $\eta = 0.05$, 1000 iterations and $k_{max} = 30$.

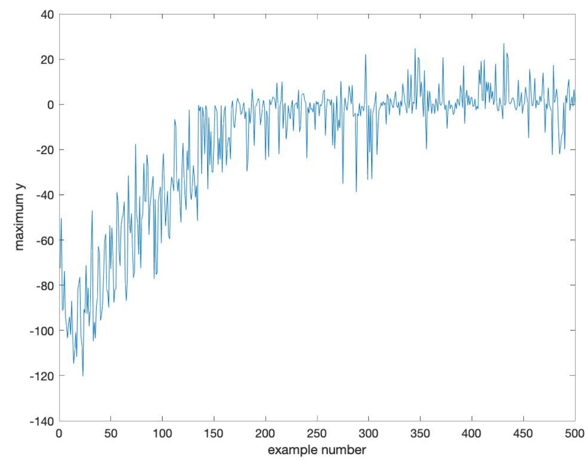
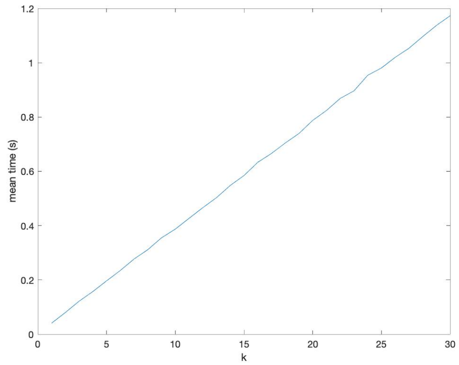


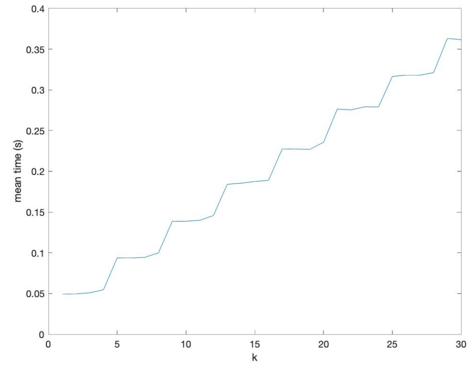
Figure 3: The value of the lower bound on y^* from Task 4 against the example number.

2.4.3 Discussion

`projected_gradient_ascent` does not provide sufficiently improved bounds over `random_sampling` to justify the multiple order of magnitude increase in computation time required. In the `branch_and_bound` procedure we see many fewer convergences as the time per iteration is increased and thus fewer iterations are completed within the time limit. The main problem with the procedure comes from keeping $x^{refined}$ within the domain. A possible improvement to this procedure would be the use of techniques to adjust the learning rate as the procedure runs for example learning rate decay or momentum. Our choices of hyperparameters in this task reflects the need to balance the quality of the bounds and the



(a) Before parralleisation



(b) After parralleisation

Figure 4: The mean time in seconds for the calculation of the refined neural network outputs across all examples from Task 4 against k . With $k_{max} = 30$ before and after parralleisation

computation time.

2.5 Task 5 - Linear Programming Bound

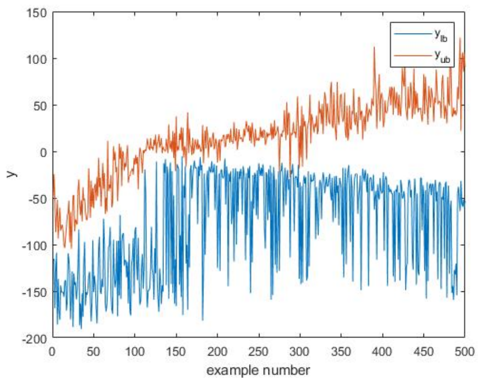
2.5.1 Method

Function `interval_bound_propogation_comprehensive` This function works much like the function from Task 2 `interval_bound_propogation`. However, it records and returns the bounds on every layer of the network.

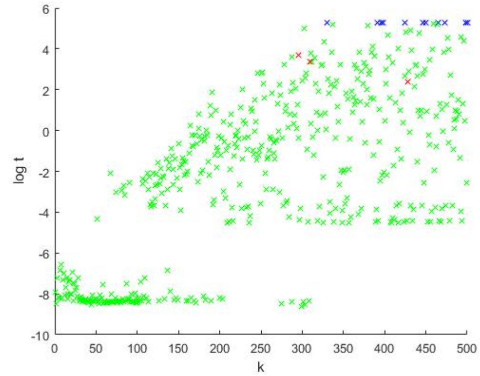
Function `calculate_num_constraints` This calculates the number of equalities and inequalities that will be generated by the program in order to be able to allocate the correct amount of memory for the matracies.

Function `generate_constraints` This creates the $\boldsymbol{A}$, $\boldsymbol{A}_{eq}$, $\boldsymbol{b}$ and $\boldsymbol{b}_{eq}$ matracies coding for the given constraints. It is notable that the number of equalities and inequalities varies with the bounds on each neuron due to the nature of the ReLU. If $z_{l_i}^{max} < 0$ then we are able to set $z_{l_i} = 0$ and if $z_{l_i}^{min} > 0$ then we are able to set $z_{l_i} = \hat{z}_{l_i}$. Otherwise we need to use three inequalities to bound the ReLU as explained in [3].

Function `linear_programming_bound` This generates an upper and lower bound on y using the `matlab` function `linprog`. First we generate bounds using `interval_bound_propogation_comprehensive` and



(a) The upper and lower bounds on y as obtained by `linear_programming_bound` in Task 5 against example number



(b) The log of the time taken for the `branch_and_bound` procedure to converge in Task 5 against the example number. Examples classified correctly are shown in green and examples which did not converge are shown in blue.

Figure 5: The bounds generated by `linear_programming_bound` in Task 5 and the results when this is used in the `branch_and_bound` procedure.

pass that, along with the result from `calculate_num_constraints`, into `generate_constraints`. We then define our cost function to be minimised, which is simply $\hat{z}_5$ for the lower bound and $-\hat{z}_5$ for the upper bound. We finally run `linprog` with each of these two cost functions to obtain y_{lb} and y_{ub}.

Script This works much like **Algorithm 3** however we use `linear_programming_bound` in the place of `interval_bound_propogation`.

Branch and Bound We ran the `branch_and_bound` procedure using a version of `linear_programming_bound` which only generated upper bounds to save on computation time. We generated lower bounds with `random_sampling`.

2.5.2 Results

The plots for Task 5 are show in figure 5. We obtained a mean lower bound on y of -77.6 and a mean upper bound on y of 14.8. We were also able to correctly classify 0 examples as false and 126 examples as true. When used in the `branch_and_bound` procedure with $\epsilon = 0.025$ and $k = 500$ for `random_sampling` we were able to correctly classify all 500 properties. The computation time was significantly shorter than when using `interval_bound_propogation` to genereate upper bounds.

2.5.3 Discussion

The `linear_programming_bound` method produces much tighter bounds than `interval_bound_propogation`. The upper bounds are significantly decreased from the previous best, however, compared to `random_sampling` and `projected_gradient_ascent` the lower bounds are very loose. This is because this method finds bounds on y rather than y^* much like `interval_bound_propogation`. We chose to only optimize the bounds of the output node in this task as it significantly reduced potential computation time. A way we could improve the bounds is to perform the optimisation on every node in the network from input to output. This would result in very tight bounds. When used with the `branch_and_bound` method we found some false examples were falsely classified with $\epsilon \geq 0.3$ due to their y^* being less than this. Once we lowered ϵ to 0.25 all examples were correctly classified.

3 Conclusion

We have discussed and evaluated our approaches to Neural Network Verification methods. We were able to correctly classify all 500 examples from the collision detection dataset through choosing the most computationally effective methods and adjusting hyperparameters. Our work has only explored methods of bounding y and y^* which were utilised in the `branch_and_bound` procedure to classify examples. Further work could explore different methods of bounding such as the extended `linear_programming_bound` method discussed in Task 5. Other areas for further work could include looking into other methods for picking out and splitting domains in the `branch_and_bound` procedure.

References

[1] Rudy R Bunel, Ilker Turkaslan, Philip Torr, Pushmeet Kohli, and Pawan K Mudigonda. A unified view of piecewise linear neural network verification. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[2] Rüdiger Ehlers. Formal verification of piece-wise linear feed-forward neural networks. *CoRR*, abs/1705.01320, 2017.

[3] Pawan K. Neural network verification, 2020.