NEURAL LOWER BOUNDS FOR VERIFICATION

Florian Jaeckle and M. Pawan Kumar
Department of Engineering Science
University of Oxford
{florian,pawan}@robots.ox.ac.uk

ABSTRACT

Recent years have witnessed the deployment of branch-and-bound (BaB) frameworks for formal verification in deep learning. The main computational bottleneck of BaB is the estimation of lower bounds. Past work in this field has relied on traditional optimization algorithms whose inefficiencies have limited their scope. To alleviate this deficiency, we propose a novel graph neural network (GNN) based approach. Our GNN architecture closely resembles the network we wish to verify. During inference, it performs forward-backward passes through the GNN layers to compute a dual solution of the convex relaxation, thereby providing a valid lower bound. During training, its parameters are estimated via a loss function that encourages large lower bounds over a time horizon. We show that our approach provides a significant speedup for formal verification compared to state-of-the-art solvers and achieves good generalization performance on unseen networks.

1 INTRODUCTION

Formal verification of neural networks is typically carried out using the branch-and-bound (BaB) framework. The BaB framework solves a mixed integer programming (MIP) formulation of the verification problem. It uses a branching strategy to split the domain of the MIP into subdomains. For each subdomain it computes an upper and a lower bound of the MIP objective. As both branching and the computation of upper bounds can be implemented efficiently, the lower bound estimation forms the main computational bottleneck for BaB. Estimating the lower bound requires solving a large convex relaxation. Typically, the relaxation is solved using either commercial solvers such as Gurobi (Gurobi Optimization, 2020) or traditional optimization algorithms such as supergradient descent or proximal minimization (Bunel et al., 2020). However, neither approach scales elegantly with the size of the relaxation, which prevents current formal verification methods from being applied to deep state-of-the-art networks.

To this end, we propose the use of a graph neural network (GNN) whose architecture closely resembles that of the network we wish to verify. For each subdomain it repeatedly computes a direction of ascent. Every single run of the GNN is made up of one or more forward and backward passes that mimic a run of the network that we're verifying. By using a parameterization of the GNN that depends only on the type of nodes and layers and not on the underlying architecture, we can train a GNN using one network and test it on another. Our method leads to a significant reduction in verification time by reducing the number of subdomains visited in the BaB framework due to the computation of more accurate lower bounds. Moreover, the GNN shows good generalization performance in two ways: when trained purely on easy properties it performs well on difficult properties as well; moreover, when trained on a small network it also generalizes well to larger networks.

2 RELATED WORK

Most bounding methods proposed in the literature create relaxations of the original problem. There are a variety of relaxations that have easily computable closed-form solutions (Gowal et al., 2018; Wong & Kolter, 2017). However, these relaxations tend to be quite loose and therefore lead to bad estimates of the final layer output of a neural network. Hence different linear programming (LP) relaxations were proposed that provide tighter bounds (Ehlers, 2017; Katz et al., 2017; Anderson et al., 2019). However, these often require an iterative solver to optimize them, which tends not to scale well. We will therefore use machine learning approaches to come up with better bounds than the best current iterative methods.

Our work is similar in spirit to the recent machine learning approaches for a variety of combinatorial optimization problems (Bello et al., 2017a; Dai et al., 2016), and mixed integer linear programs (MILPs) (Alvarez et al., 2017; Gasse et al., 2019; Hansknecht et al., 2018; Khalil et al., 2016). In the context of MILPs, learning has mostly been used to improve the branching strategies in BaB algorithms (Hansknecht et al., 2018; Khalil et al., 2016; Lu & Kumar, 2020b). However, only limited work has been done on learning the estimation of lower bounds (Dvijotham et al., 2018a;b; Gowal et al., 2019) and existing methods beat Interval Bound propagation, a comparatively weak baseline, by a small margin only. There is thus a large scope for improvement which we explore below.

3 Background

We focus on returning a lower bound on the output of the neural network we are trying to verify. We are given a convex input domain C, an input vector $\mathbf{z}_0 \in C$, the network weights W and biases $\mathbf{b}$, and a non-linear activation function $\sigma(\cdot)$. In our case we use the ReLU activation defined as $\sigma(x) = \max(x, 0)$ As the neural network is highly non-convex and thus hard to optimize over, Bunel et al. (2020) introduced a decomposition based technique for the Planet relaxation (Ehlers, 2017).

Planet Relaxation. We denote the output of the k-th layer before the application of the ReLU as $\hat{\mathbf{z}}_k$ and the output of applying the ReLU to $\hat{\mathbf{z}}_k$ as $\mathbf{z}_k$. Given the lower bounds $\mathbf{l}_k$ and upper bounds $\mathbf{u}_k$ of the values of $\hat{\mathbf{z}}_k$, we relax the ReLU activations $\mathbf{z}_k = \sigma(\hat{\mathbf{z}}_k)$ to its convex hull $cvx\_hull_\sigma(\hat{\mathbf{z}}_k, \mathbf{z}_k, \mathbf{l}_k, \mathbf{u}_k)$:

$$cvx\_hull_\sigma(\hat{\mathbf{z}}_k, \mathbf{z}_k, \mathbf{l}_k, \mathbf{u}_k) \equiv \begin{cases} \mathbf{z}_k[i] \geq 0 \quad \mathbf{z}_k[i] \geq \hat{\mathbf{z}}_k[i] & \\ \mathbf{z}_k[i] \leq \frac{\mathbf{u}_k[i](\hat{\mathbf{z}}_k[i]-\mathbf{l}_k[i])}{\mathbf{u}_k[i]-\mathbf{l}_k[i]} & \text{if } \mathbf{l}_k[i] < 0 \text{ and } \mathbf{u}_k[i] > 0 \\ \mathbf{z}_k[i] = 0 & \text{if } \mathbf{u}_k[i] \leq 0 \\ \mathbf{z}_k[i] = \hat{\mathbf{z}}_k[i] & \text{if } \mathbf{l}_k[i] \geq 0. \end{cases} \tag{1}$$

Using the above notation, the Planet relaxation for computing the lower bound can be written as:

$$\min_{\mathbf{z}, \hat{\mathbf{z}}} \hat{\mathbf{z}}_n \text{ s.t. } \mathscr{P}_0(\mathbf{z}_0, \hat{\mathbf{z}}_1); \mathscr{P}_k(\hat{\mathbf{z}}_k, \hat{\mathbf{z}}_{k+1}) \text{ for } k \in [1, \ldots, L-1], \text{where} \tag{2}$$

$$\mathscr{P}_0(\mathbf{z}_0, \hat{\mathbf{z}}_1) \equiv \begin{cases} \mathbf{z}_0 \in C \\ \hat{\mathbf{z}}_1 = W_1 \mathbf{z}_0 + \mathbf{b}_1 \end{cases} \qquad \mathscr{P}_k(\hat{\mathbf{z}}_k, \hat{\mathbf{z}}_{k+1}) \equiv \begin{cases} \exists \mathbf{z}_k \text{ s.t.} \\ \mathbf{l}_k \leq \hat{\mathbf{z}}_k \leq \mathbf{u}_k \\ cvx\_hull_\sigma(\hat{\mathbf{z}}_k, \mathbf{z}_k, \mathbf{l}_k, \mathbf{u}_k) \\ \hat{\mathbf{z}}_{k+1} = W_{k+1} \mathbf{z}_k + \mathbf{b}_{k+1}. \end{cases} \tag{3}$$

Lagrangian Decomposition. We often merely need approximations of the bounds rather than the precise values of them. We can therefore make use of the primal-dual formulation of the problem as every feasible solution to the dual problem provides a valid lower bound for the primal problem. Following the work of Bunel et al. (2020) we will use the Lagrangian decomposition. To this end, we first create two copies $\hat{\mathbf{z}}_{A,k}, \hat{\mathbf{z}}_{B,k}$ of each variable $\hat{\mathbf{z}}_k$ and obtain the dual by introducing Lagrange multipliers $\boldsymbol{\rho}$ corresponding to the equality constraints of the two copies of each variable:

$$\begin{aligned} q(\boldsymbol{\rho}) = \min_{\mathbf{z}, \hat{\mathbf{z}}} \quad & \hat{\mathbf{z}}_{A,n} + \sum_{k=1,\ldots,n-1} \boldsymbol{\rho}_k^\top (\hat{\mathbf{z}}_{B,k} - \hat{\mathbf{z}}_{A,k}) \\ \text{s.t.} \quad & \mathscr{P}_0(\mathbf{z}_0, \hat{\mathbf{z}}_{A,1}); \; \mathscr{P}_k(\hat{\mathbf{z}}_{B,k}, \hat{\mathbf{z}}_{A,k+1}) \text{ for } k \in [1, \ldots, L-1]. \end{aligned} \tag{4}$$

We aim to maximize $q(\boldsymbol{\rho})$ to get the tightest possible lower bound for the primal problem (2).

4 GNN Framework

The key observation of our work is based on the fact that several previously known lower bound computation techniques can be thought of as performing forward-backward style passes through the network to update the dual variables. However, the exact form of the passes is restricted to those suggested by standard optimization algorithms, which are agnostic to the special structure of neural lower bound computation. This observation suggests a natural generalization: parameterize the forward and backward passes using a GNN, and estimate the parameters using a training data set so as to exploit the problem and data structure more successfully. We will now describe the GNN's main elements in greater detail before outlining the forward-backward passes, the final update step, and the training of the GNN.

4.1 GNN Components

Nodes. We create a node $v_k[i]$ in our GNN for every dual variable $\boldsymbol{\rho}_k[i]$. Every dual variable corresponds to the output of the non-linear activation and the input to the next linear layer. We denote the set of all nodes in the GNN by V_{GNN}.

Node Features. For each node $v_k[i]$ we define a corresponding d-dimensional feature vector $\mathbf{f}_k[i] \in \mathbb{R}^d$ describing the current state of that node. We use the current dual variables as well as the corresponding duplicated primal variables as features. While more complex features could be included, we deliberately chose the simple features and rely on the power of GNNs to efficiently compute an accurate ascent direction.

Edges. We denote the set of all the edges connecting the nodes in V_{GNN} by E_{GNN}. The edges are equivalent to the weights in the neural network that we are trying to verify. We define e^k_{ij} to be the edge connecting nodes $v_k[i]$ and $v_{k+1}[j]$ and assign it the value of W^k_{ij}.

Embeddings. For every node $v_k[i]$ we compute a corresponding p-dimensional initial embedding vector $\boldsymbol{\mu}_k[i] \in \mathbb{R}^p$ using a learned function $g : \mathbb{R}^p \to \mathbb{R}^d$ to get $\boldsymbol{\mu}_k[i] := g(\mathbf{f}_k[i])$.

4.2 Forward and Backward Passes

So far the embedding vector $\boldsymbol{\mu}$ solely depends on the current state of that node and does not take the underlying structure of the problem or the neighbouring nodes into consideration. We therefore introduce a method that updates the embedding vectors by simulating the forward and backward passes in the original network using learnt functions F and B. Specifically, during the forward update, for $k = 1, \dots, L-1$, we have, for all possible i,

$$\boldsymbol{\mu}_k[j] \longleftarrow F(\mathbf{f}_k[i], \boldsymbol{\mu}_{k-1}, \boldsymbol{\mu}_k[j], e^k; \Theta_1). \tag{5}$$

During the backward update, for $i = L-1, \dots, 1$, we have

$$\boldsymbol{\mu}_k[i] \longleftarrow B(\mathbf{f}_k[i], \boldsymbol{\mu}_{k+1}, \boldsymbol{\mu}_k[i], e^{k+1}; \Theta_2). \tag{6}$$

4.3 Update Step

Finally, we need to reduce each p-dimensional embedding vector to a single value to get an ascent direction $\hat{\boldsymbol{\rho}}^{t+1}_k$. We simply use a linear output function $\boldsymbol{\Theta}^{out}$ to get: $\hat{\boldsymbol{\rho}}^{t+1}_k = \boldsymbol{\Theta}^{out}\boldsymbol{\mu}_k$. As it is not feasible for the GNN to output a new ascent direction that will lead us directly to the global optimum of equation (4) we propose to run the GNN a small number of times to return ascent directions that gradually move towards the optimum. Given a step size η^{t+1}, previous dual variables $\boldsymbol{\rho}^t$, and the new ascent direction $\hat{\boldsymbol{\rho}}^{t+1}$ we update the dual variables as follows: $\boldsymbol{\rho}^{t+1} = \boldsymbol{\rho}^t + \eta^{t+1}\hat{\boldsymbol{\rho}}^{t+1}$.

4.4 GNN Training

Having described the structure of the GNN we will now show how to train its learnable parameters. Our training dataset $\mathscr{D}$ consists of a set of samples $d_i = (\mathbf{x}^i, \varepsilon^i, W^i, \mathbf{b}^i, \mathbf{l}^i, \mathbf{u}^i, \boldsymbol{\rho}^i)$, each with the following components: a natural input to the neural network we wish to verify ($\mathbf{x}$), for example an image; a domain for which we wish to compute the lower bound (ε), which in our case is an ℓ_∞ ball; the weights and biases of the neural network ($W, \mathbf{b}$); the intermediate bounds of the neural network ($\mathbf{l}, \mathbf{u}$); and the initial value of the dual variables ($\boldsymbol{\rho}$). In order for the training procedure to more closely resemble its behaviour at inference time, it is crucial to train the GNN using a loss function that takes into account the dual values across a large number of iterations K. In order to ensure that a

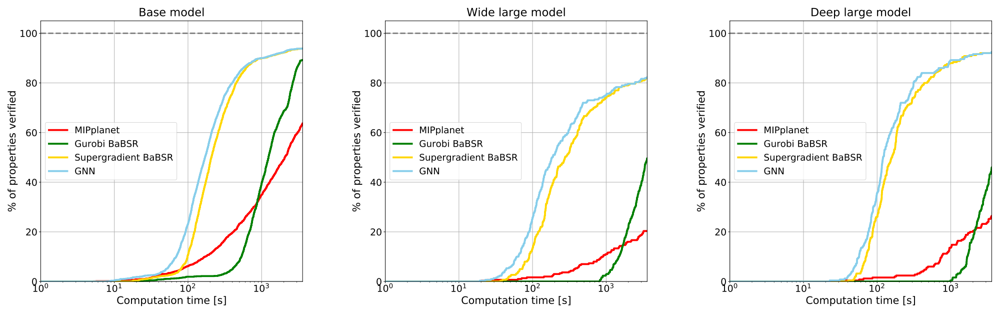

Figure 1: Cactus plots for the base, wide and deep models. For each, we compare the bounding methods and complete verification algorithms by plotting the percentage of solved properties as a function of runtime.

single training sample does not dominate the loss by reaching a large positive value, we truncate the loss values for each sample. The natural point to clamp the individual losses at is the value returned by supergradient ascent (q^i_{SupG}) plus a small positive threshold κ. Given the i-th training sample $d_i \in \mathscr{D}$, the corresponding dual objective q^i, and the dual value returned by supergradient ascent q^i_{SupG}, we define its loss $\mathscr{L}_i$ to be:

$$\mathscr{L}_i = -\sum_{t=1}^{K} q^i(\boldsymbol{\rho}^{i,t}_{GNN}) * \gamma^t * \mathbf{1}_{q^i(\boldsymbol{\rho}^{i,K}_{GNN}) < q^i_{SupG} + \kappa}, \tag{7}$$

given a decay factor $\gamma \in (0,1)$. If γ is low then we encourage the model to make as much progress in the first few steps as possible, whereas if γ is closer to 1, then more emphasis is placed on the final output of the GNN, sacrificing progress in the early stages. We sum over the individual loss values corresponding to each data point to get the final training objective $\mathscr{L}$: $\mathscr{L} = \sum_{i=1}^{|D|} \mathscr{L}_i$.

5 Experiments

Setup. We will now show the practical effectiveness of our method by comparing its performance with several state-of-the-art verification methods. All our experiments are performed on the test set of the CIFAR10 dataset (Krizhevsky et al., 2009). We use the same verification dataset used by Lu & Kumar (2020b) which consists of 3 different networks to verify (see Appendix E). We will use the main one, which we call the Base model, to do all of our training and most of our testing on. We will further test the transferability of our GNN on two larger networks. **Baselines.** We compare our work to the baselines used in both Bunel et al. (2018) and Lu & Kumar (2020b). We use MIPplanet, which is a mixed integer solver by the commercial solver GUROBI, BaBSR, a BaB based method that uses an LP solver by GUROBI to compute bounds for the subdomains, and supergradient ascent together with Adam as proposed by Bunel et al. (2020) (for a more detailed description see Appendix C). **Results - Base Model.** We first run experiments on the base model using a GNN that is trained on 20 easy properties only. We compare our method with previous work by showing how many properties are verified for any given amount of time in seconds (Figure 1). Our method leads to an over 70% reduction in terms of average time taken compared to BaBSR and MIPplanet a 10% reduction compared to Supergradient ascent. In fact, the GNN beats supergradient ascent on 93.80% of all properties. Even though the GNN is trained on easy properties only it generalizes well to harder ones. **Transferability: Larger Models.** We show further the generalization performance of our GNNs without the need to perform fine-tuning or online learning by testing it on two different larger neural networks. As shown in Figure 1, the GNN still outperforms all baselines on both unseen networks. The GNN beats each baseline on over 85% of all properties. For a more detailed analysis of our empirical results see Appendix F.4.

6 Discussion

We have shown how to improve the complete verification procedure using GNNs that learn how to use the underlying structure of the problem to return better bounds more quickly. We show that our method consistently beats the existing state-of-the-art algorithms. Our GNN trained on easy properties on a small network shows good generalization performance on harder properties and on larger unseen networks.

References

Alejandro Marcos Alvarez, Quentin Louveaux, and Louis Wehenkel. A machine learning-based approximation of strong branching. *INFORMS Journal on Computing*, 29(1):185–195, 2017.

Ross Anderson, Joey Huchette, Christian Tjandraatmadja, and Juan Pablo Vielma. Strong mixed-integer programming formulations for trained neural networks. In *International Conference on Integer Programming and Combinatorial Optimization*, pp. 27–42. Springer, 2019.

Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *The International Conference on Learning Representations Workshop*, 2017a.

Rudy Bunel, Alessandro De Palma, Alban Desmaison, Krishnamurthy Dvijotham, Pushmeet Kohli, Philip Torr, and M Pawan Kumar. Lagrangian decomposition for neural network verification. In *Conference on Uncertainty in Artificial Intelligence*, pp. 370–379. PMLR, 2020.

Rudy R Bunel, Ilker Turkaslan, Philip Torr, Pushmeet Kohli, and M Pawan Kumar. A unified view of piecewise linear neural network verification. In *Advances in Neural Information Processing Systems*, pp. 4790–4799, 2018.

Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data. In *International conference on machine learning*, pp. 2702–2711, 2016.

Krishnamurthy Dvijotham, Sven Gowal, Robert Stanforth, Relja Arandjelovic, Brendan O'Donoghue, Jonathan Uesato, and Pushmeet Kohli. Training verified learners with learned verifiers. *arXiv preprint arXiv:1805.10265*, 2018a.

Krishnamurthy Dvijotham, Robert Stanforth, Sven Gowal, Timothy A Mann, and Pushmeet Kohli. A dual approach to scalable verification of deep networks. In *Conference on Uncertainty in Artificial Intelligence*, pp. 550–559, 2018b.

Ruediger Ehlers. Formal verification of piece-wise linear feed-forward neural networks. In *International Symposium on Automated Technology for Verification and Analysis*, pp. 269–286. Springer, 2017.

Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks. In *Advances in Neural Information Processing Systems*, pp. 15554–15566, 2019.

Sven Gowal, Krishnamurthy Dvijotham, Robert Stanforth, Rudy Bunel, Chongli Qin, Jonathan Uesato, Timothy Mann, and Pushmeet Kohli. On the effectiveness of interval bound propagation for training verifiably robust models. *arXiv preprint arXiv:1810.12715*, 2018.

Sven Gowal, Krishnamurthy Dvijotham, Robert Stanforth, Timothy Mann, and Pushmeet Kohli. A dual approach to verify and train deep networks. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, pp. 6156–6160. AAAI Press, 2019.

LLC Gurobi Optimization. Gurobi optimizer reference manual, 2020. URL `http://www.gurobi.com`.

Christoph Hansknecht, Imke Joormann, and Sebastian Stiller. Cuts, primal heuristics, and learning to branch for the time-dependent traveling salesman problem. *arXiv preprint arXiv:1805.01415*, 2018.

Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks. In *International Conference on Computer Aided Verification*, pp. 97–117. Springer, 2017.

Elias Boutros Khalil, Pierre Le Bodic, Le Song, George Nemhauser, and Bistra Dilkina. Learning to branch in mixed integer programming. *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

Jingyue Lu and M Pawan Kumar. Neural network branching for neural network verification. In *International Conference on Learning Representations*, 2020a.

Jingyue Lu and M Pawan Kumar. Neural network branching for neural network verification. In *International Conference on Learning Representations*, 2020b.

Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *International Conference on Learning Representations*, 2018.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. *Automatic differentiation in pytorch*, 2017.

Gagandeep Singh, Jonathan Maurer, Christoph Mller, Matthew Mirman, Timon Gehr, Adrian Hoffmann, Petar Tsankov, Dana Drachsler Cohen, Markus Pschel, and Martin Vechev. Eth robustness analyzer for neural networks (eran), 2020. URL `https://github.com/eth-sri/eran`.

VNN-COMP. International verification of neural networks competition (vnn-comp), 2020. URL `https://sites.google.com/view/vnn20/vnncomp`.

Eric Wong and J Zico Kolter. Provable defenses against adversarial examples via the convex outer adversarial polytope. *arXiv preprint arXiv:1711.00851*, 2017.

A Branch and Bound

We will now describe the branch-and-bound algorithm referenced throughout this work. We use a slightly modified version as we are merely interested in whether the final lower bound is positive or negative instead of its precise value; we prune away all subdomains with positive lower bounds. We use the same hand-designed ReLU splitting branching strategy used in Lu & Kumar (2020b).

Algorithm 1 Branch and Bound

```
1: function BAB(net, problem)
2:     global_lb ← compute_LB(net, problem)                              ▷ global lower bound
3:     global_ub ← compute_UB(net, problem)                              ▷ global upper bound
4:     probs ← [(global_lb, problem)]                                    ▷ set of all current domains
5:     while probs is not empty do
6:         (_ , prob) ← pick_out(probs)          ▷ the pick˙out function picks an ambiguous ReLU to split on
7:         [subprob_1, subprob_2] ← split(prob)
8:         for $i = 1, 2$ do
9:             sub_lb ← compute_LB(net, subprob_i)
10:            sub_ub ← compute_UB(net, subprob_i)
11:            if sub_ub $< 0$ then
12:                return SAT                                            ▷ we've found an adversarial example
13:            end if
14:            if sub_lb $< 0$ then
15:                probs.append((sub_lb, subprob_i))
16:            end if                                ▷ if sub_lb $> 0$ then the subdomain gets pruned away
17:        end for
18:        global_lb ← min{lb | (lb, prob) ∈ probs}   ▷ If probs is non-empty then global_lb is negative
19:    end while
20:    return UNSAT          ▷ all subproblems have a positive lower bound, therefore global_lb is positive
21: end function
```

We will now describe the parallelized version of the BaB algorithm used whenever we use supergradient ascent or the GNN to compute final bounds:

Algorithm 2 Branch and Bound — parallelized version

```
function BAB(net, problem)
    global_lb ← compute_LB(net, problem)                ▷ global lower bound
    global_ub ← compute_UB(net, problem)                ▷ global upper bound
    probs ← [(global_lb, problem)]                      ▷ set of all current domains
    while probs is not empty do
        $s = \min\{$batch_size$/2, $len(probs)$\}$
        subproblems = []
        for $i = 1 \dots s$ do
            (_ , prob) ← pick_out(probs)    ▷ the pick˙out function picks an ambiguous ReLU to split on
            [subprob_$i_1$, subprob_$i_2$] ← split(prob)
            subproblems ← subproblems + [subprob_$i_1$, subprob_$i_2$]
        end for
        [sub_lb$_{1_1}$, sub_lb$_{1_2}$, ..., sub_lb$_{s_1}$, sub_lb$_{s_2}$] ← compute_LBs(net, subproblems)
        [sub_ub$_{1_1}$, sub_ub$_{1_2}$, ..., sub_ub$_{s_1}$, sub_ub$_{s_2}$] ← compute_UBs(net, subproblems)
        for $i = 1 \dots s$ do
            for $j = 1, 2$ do
                if sub_ub$_{i_j} < 0$ then
                    return SAT                          ▷ we've found an adversarial example
                end if
                if sub_lb$_{i_j} < 0$ then
                    probs.append((sub_lb$_{i_j}$, subprob_$i_j$))
                end if
            end for
        end for
        global_lb ← min{lb | (lb, prob) ∈ probs}    ▷ If probs is non-empty then global_lb is negative
    end while
    return UNSAT    ▷ all subproblems have a positive lower bound, therefore global_lb is positive
end function
```

B GNN ARCHITECTURE

We now describe the components of the GNN, the forward and backward passes, the update step, the fail-safe strategy, and the generation of the training dataset in greater depth.

B.1 GNN COMPONENTS

Nodes. We create a node $\boldsymbol{v}_k[i]$ in our GNN for every dual variable $\boldsymbol{\rho}_k[i]$. Every dual variable corresponds to the output of the non-linear activation and the input to the next linear layer. We denote the set of all nodes in the GNN by V_{GNN}.

Node Features. For each node $\boldsymbol{v}_k[i]$ we define a corresponding d-dimensional feature vector $\mathbf{f}_k[i] \in \mathbb{R}^d$ describing the current state of that node as follows:

$$\mathbf{f}_k[i] := \left(\boldsymbol{\rho}_k[i], \hat{\mathbf{z}}_{A,k}[i], \hat{\mathbf{z}}_{B,k}[i], \hat{\mathbf{z}}_{B,k}[i] - \hat{\mathbf{z}}_{A,k}[i]\right)^\top. \tag{8}$$

Here, $\boldsymbol{\rho}_k$ is the current assignment to the corresponding dual variables and $\hat{\mathbf{z}}_{B,k}$ and $\hat{\mathbf{z}}_{A,k}$ are the closed-form solutions to the inner minimization problem of the dual problem as explained above. The term $\hat{\mathbf{z}}_{B,k}[i] - \hat{\mathbf{z}}_{A,k}[i]$ corresponds to the supergradient of q. While more complex features could be included, we deliberately chose the simple features described above and rely on the power of GNNs to efficiently compute an accurate ascent direction.

Edges. We denote the set of all the edges connecting the nodes in V_{GNN} by E_{GNN}. The edges are equivalent to the weights in the neural network that we are trying to verify. We define e^k_{ij} to be the edge connecting nodes $v_k[i]$ and $v_{k+1}[j]$ and assign it the value of W^k_{ij}.

Embeddings. For every node $v_k[i]$ we compute a corresponding p-dimensional embedding vector $\boldsymbol{\mu}_k[i] \in \mathbb{R}^p$ using a learned function g:

$$\boldsymbol{\mu}_k[i] := g(\mathbf{f}_k[i]). \tag{9}$$

In our case g is a multilayer perceptron (MLP), which is made up of a series of linear layers Θ_i and non-linear activations σ. We have the following set of trainable parameters:

$$\Theta_0 \in \mathbb{R}^{d\times p}, \quad \Theta_1,\ldots,\Theta_{T_1} \in \mathbb{R}^{p\times p}, \quad \mathbf{b}_0,\ldots,\mathbf{b}_{T_1} \in \mathbb{R}^{p}. \tag{10}$$

Given a feature vector $\mathbf{f}$ we compute the following set of vectors:

$$\boldsymbol{\mu}^0 = \Theta_0 \cdot \mathbf{f} + b_0, \quad \boldsymbol{\mu}^{l+1} = \Theta_{l+1} \cdot \text{relu}(\boldsymbol{\mu}^l) + \mathbf{b}_{l+1}. \tag{11}$$

We initialize the embedding vector to be $\boldsymbol{\mu} = \boldsymbol{\mu}^{T_1}$, where $T_1 + 1$ is the depth of the MLP.

B.2 Forward and Backward Passes.

So far the embedding vector $\boldsymbol{\mu}$ solely depends on the current state of that node and does not take the underlying structure of the problem or the neighbouring nodes into consideration. We therefore introduce a method that updates the embedding vectors by simulating the forward and backward passes in the original network. The forward pass consists of a weighted sum of three parts: the first term is the current embedding vector, the second is the embedding vector of the previous layer passed through the corresponding linear or convolutional filters, and the third is the average of all neighbouring embedding vectors:

$$\boldsymbol{\mu}'_k[i] = \text{relu}\left(\Theta_1^{for}\boldsymbol{\mu}_k[i] + \Theta_2^{for}\left(W_k\boldsymbol{\mu}_{k-1} + \mathbf{b}_{k-1}\right)[i] + \Theta_3^{for}\left(\sum_{j\in N(i)} \boldsymbol{\mu}_{k-1}[j]/Q_{k+1}[j]\right)[i]\right). \tag{12}$$

Both the second and the third term can be implemented using existing deep learning functions. Similarly, we perform a backward pass as follows:

$$\boldsymbol{\mu}_k[i] = \text{relu}\left(\Theta_1^{back}\boldsymbol{\mu}'_k[i] + \Theta_2^{back}(W_{k+1}^T\left(\boldsymbol{\mu}'_{k+1} - \mathbf{b}_{k+1}\right))[i] + \Theta_3^{back}\left(\sum_{j\in N'(i)} \boldsymbol{\mu}'_{k+1}[j]/Q'_{k+1}[j]\right)[i]\right). \tag{13}$$

Here $\Theta_1^{for}, \Theta_2^{for}, \Theta_3^{for}, \Theta_1^{back}, \Theta_2^{back}, \Theta_3^{back} \in \mathbb{R}^{p\times p}$ are all learnable parameters. To ensure better generalization performance to unseen neural networks with a different network architecture we include normalization parameters Q and Q'. These are matrices whose elements are the number of neighbouring nodes in the previous and following layer respectively for each node. We repeat this process of running a forward and backward pass T_2 times. The high-dimensional embedding vectors are now capable of expressing the state of the corresponding node taking the entire problem structure into consideration as they are directly influenced by every single other node, even if we set $T_2 = 1$.

B.3 Update Step

Finally, we need to reduce each p-dimensional embedding vector to a single value to get an ascent direction $\hat{\boldsymbol{\rho}}_k^{t+1}$. We simply use a linear output function $\boldsymbol{\Theta}^{out}$ to get: $\hat{\boldsymbol{\rho}}_k^{t+1} = \boldsymbol{\Theta}^{out}\boldsymbol{\mu}_k$.

Ideally the GNN would output a new ascent direction that will lead us directly to the global optimum of equation (4). However, as the dual problem is complex this may not be feasible in practice without making the GNN very large, thereby resulting in computationally prohibitive inference. Instead, we propose to run the GNN a small number of times to return ascent directions that gradually move towards the optimum. Given a step size η^{t+1}, previous dual variables $\boldsymbol{\rho}^t$, and the new ascent direction $\hat{\boldsymbol{\rho}}^{t+1}$ we update the dual variables as follows:

$$\boldsymbol{\rho}^{t+1} = \boldsymbol{\rho}^t + \eta^{t+1}\hat{\boldsymbol{\rho}}^{t+1}. \tag{14}$$

Similar to many iterative optimization methods we decay our stepsize as we want to take smaller steps the closer we get to the optimal solution. Given an initial step size η_0, we define the step size at time t as follows: $\eta^t = \eta_0 * \sqrt{t}$.

The hyper-parameters for the GNN computation of the duals are the depth of the MLP (T_1), how many forward and backward passes we run (T_2), and the embedding size (p).

B.4 Fail-safe Strategy.

As with almost all machine learning based optimization algorithms, we do not have any convergence guarantees. For a few subdomains our GNN might diverge rather than improve on the value returned for its parent. We therefore introduce a fail-safe strategy that ensures our algorithm performs well even when our GNN fails. We compare whether the final bound of a given subdomain outputted by the GNN beats the bound returned for its parent domain by a given absolute threshold. If it fails to do so, then we add the subdomain into a second set of current subdomains. We use supergradient ascent to solve these subdomains on which our GNN performed poorly. This way we reduce the risk of our branch-and-bound algorithm timing out on certain properties.

B.5 Training Dataset.

We would like to train the GNN on the same samples that we will encounter during inference time. However, that is impossible as the structure and the elements of the BaB tree computed at test time depend on the lower bound computation and thus on the GNN. To resolve that problem we dynamically create a training dataset as follows. We first pick a fixed number of images from the training dataset used in Lu & Kumar (2020b) together with the corresponding properties that we are verifying our network against and epsilon values defining the input domain. We then create the first part of the training dataset by running a complete BaB algorithm on these properties using the supergradient method. We record the intermediate bounds and parent dual variables for each subdomain visited to create a dataset to train a first GNN on. Once we have finished training the first version of the GNN we extend the dataset by running another complete BaB algorithm on the same properties; this time using the first version of the GNN instead of supergradient ascent to compute the lower bounds. We subsequently resume training the first GNN on the extended dataset for a fixed number of epochs to get a second GNN. We then repeat this process of extending the dataset and further training the GNN for a fixed number of iterations. For most properties in the training dataset we acquire a large number of samples over the different iterations. To speed up training, we reduce the proportion of the training dataset on which we train our GNNs by only picking a small subset of the samples for each property. We make sure to pick subdomains from different stages of the BaB algorithm in order to get a more diverse training dataset. We train the GNN using the loss function described in the previous section and using the Adam optimizer Kingma & Ba (2015) with no weight decay.

C Supergradient Method

We will now outline the supergradient ascent method used in Bunel et al. (2020). We run supergradient ascent for 500 steps with a learning rate of 1e-4 (both of these hyper-parameters have been optimized over the validation dataset).

Algorithm 3 Supergradient method

```
1: function SUPERG_COMPUTE_BOUNDS($\{W_k, \mathbf{b}_k, \mathbf{l}_k, \mathbf{u}_k\}_{k=1..n}$)
2:     Initialise dual variables $\boldsymbol{\rho}^0$ using the duals of the parent domain or the algo of Wong & Kolter (2017)
3:     for nb_iterations do
4:         $\hat{\mathbf{z}}^*$, $\hat{\mathbf{z}}_A^*$ $\hat{\mathbf{z}}_B^* \leftarrow$ inner minimization as proposed by Bunel et al. (2020)
5:         Compute supergradient using $\nabla_{\boldsymbol{\rho}} q(\boldsymbol{\rho}^t) = \hat{\mathbf{z}}_B^* - \hat{\mathbf{z}}_A^*$
6:         $\boldsymbol{\rho}^{t+1} \leftarrow$ Adam's update rule Kingma & Ba (2015)
7:     end for
8:     return $q(\boldsymbol{\rho})$
9: end function
```

D REGAINING SUPERGRADIENT ASCENT

As mentioned earlier, the motivation behind our GNN framework is to offer a parameterized generalization of previous methods for lower bound computation. We now formalize the generalization using the following proposition.

Proposition 1 *Our GNN architecture can simulate supergradient ascent Bunel et al. (2020).*

The supergradient ascent step is equivalent to update step $\boldsymbol{\rho}^{t+1} = \boldsymbol{\rho}^t + \eta^{t+1}\hat{\boldsymbol{\rho}}^{t+1}$, where

$$\hat{\boldsymbol{\rho}}_k^{t+1} = \hat{\mathbf{z}}_{B,k} - \hat{\mathbf{z}}_{A,k}. \tag{15}$$

Let Θ_0 be the zero-matrix with non-zero elements $\Theta_0[1,4] = 1$, $\Theta_0[2,4] = -1$. Moreover, setting $T_1 = 1$, $\Theta_1 = \mathbb{1}$ and $\mathbf{b}_0 = \mathbf{b}_1 = \mathbf{0}$, we get

$$\boldsymbol{\mu}_k^0 = \left(\hat{\mathbf{z}}_{B,k} - \hat{\mathbf{z}}_{A,k}, -\hat{\mathbf{z}}_{B,k} + \hat{\mathbf{z}}_{A,k}, \mathbf{0}, \dots, \mathbf{0}\right)^\top, \tag{16}$$

$$\boldsymbol{\mu} = \left(\left(\hat{\mathbf{z}}_{B,k} - \hat{\mathbf{z}}_{A,k}\right)_+, -\left(\hat{\mathbf{z}}_{B,k} - \hat{\mathbf{z}}_{A,k}\right)_-, \mathbf{0}, \dots, \mathbf{0}\right)^\top. \tag{17}$$

If we set $\Theta_2^{for} = \Theta_3^{for} = \Theta_2^{back} = \Theta_3^{back} = \mathbf{0}$ and $\Theta_1^{for} = \Theta_1^{back} = \mathbb{1}$, then the forward and backward passes don't change the embedding vector. We now just need to set $\boldsymbol{\Theta}^{out} = (1, -1, 0, \dots, 0)^\top$ to get the final ascent direction:

$$\hat{\boldsymbol{\rho}}_k^{t+1} = \left(\hat{\mathbf{z}}_{B,k} - \hat{\mathbf{z}}_{A,k}\right)_+ + \left(\hat{\mathbf{z}}_{B,k} - \hat{\mathbf{z}}_{A,k}\right)_- = \hat{\mathbf{z}}_{B,k} - \hat{\mathbf{z}}_{A,k}. \tag{18}$$

We have shown that we can simulate supergradient ascent using our GNN architecture.

E Network Architectures

We perform all our experiments on the same three neural networks used by Lu & Kumar (2020b). All three networks are trained robustly using the method introduced by Madry et al. (2018) to achieve robustness against l_∞ perturbations of size up to $\varepsilon = 8/255$ (the amount typically considered in empirical works).

The neural network architectures differ in the size and number of convolutional layers used. However, all of them have two fully connected layers of 100 and 10 units respectively as the final layers. Each layer but the last is followed by a ReLU activation function.

Network Name	**No. of Properties**	**Network Architecture**
BASE Model	Training: 430 Easy: 425 (Lu & Kumar (2020b): 467) Medium: 704 (Lu & Kumar (2020b): 773) Hard: 387 (Lu & Kumar (2020b): 426)	Conv2d(3,8,4, stride=2, padding=1) Conv2d(8,16,4, stride=2, padding=1) linear layer of 100 hidden units linear layer of 10 hidden units (Total ReLU activation units: 3172)
WIDE	300	Conv2d(3,16,4, stride=2, padding=1) Conv2d(16,32,4, stride=2, padding=1) linear layer of 100 hidden units linear layer of 10 hidden units (Total ReLU activation units: 6244)

DEEP	250	Conv2d(3,8,4, stride=2, padding=1)
		Conv2d(8,8,3, stride=1, padding=1)
		Conv2d(8,8,3, stride=1, padding=1)
		Conv2d(8,8,4, stride=2, padding=1)
		linear layer of 100 hidden units
		linear layer of 10 hidden units
		(Total ReLU activation units: 6756)

Table 1: Network Architectures

F Further Experimental Results

We will now compare our method with the different baselines in greater depth.

F.1 Implementation.

The implementation of our method is based on Pytorch Paszke et al. (2017). We compute the intermediate bounds of the network using the method introduced by Wong & Kolter (2017). The two baselines using GUROBI are run on one CPU each, as done by Lu & Kumar (2020b). We run both the supergradient baseline and our GNN on a single GPU and CPU each. One advantage of our method compared to off-the-shelf solvers is precisely that we can run it on GPUs and can therefore use more efficient parallelized implementations of mathematical operations. To speed up the BaB algorithm we parallelize over the lower bound computation for the different subdomains. We run both the GNN and supergradient ascent with a batch-size of 300. We run our GNN for 100 steps with an absolute fail-safe threshold of 0.1.

F.2 Mean

We compare the different methods on the Base, the Wide, and the Deep model using the arithmetic mean in Table 2.

	Base			Wide			Deep		
Method	time(s)	subdomains	%Timeout	time(s)	subdomains	%Timeout	time(s)	subdomains	%Timeout
GUROBI BABSR	1592.30	**1346.44**	10.75	2906.71	**632.59**	50.33	3007.237	**299.913**	54.00
MIPPLANET	2042.46		36.28	3112.11		79.67	2997.115		73.60
SUPERGRADIENT BABSR	513.19	6348.84	6.20	986.68	5416.54	18.00	525.77	2616.97	8.00
GNN	**473.05**	5322.27	**6.07**	**920.16**	4599.56	**17.67**	**494.18**	2335.16	**7.60**

Table 2: We compare average (mean) solving time, average number of subdomains solved, and the percentage of properties that the methods time out on when using a cut-off time of 3600s. The best performing method for each subcategory is highlighted in bold.

F.3 MEDIAN

Unlike the arithmetic mean, the median is not skewed by outliers. The randomly picked threshold at which we stop experiments (3600s) has a larger impact on the mean than the median so as long as methods time-out on less than half of all images.

	Base			Wide			Deep		
Method	time(s)	subdomains	%Timeout	time(s)	subdomains	%Timeout	time(s)	subdomains	%Timeout
GUROBI BABSR	1239.56	**1028.00**	10.75	3600.00	**530.00**	50.33	3600.00	**264.00**	54.00
MIPPLANET	2063.54		36.28	3600.00		79.67	3600.00		73.60
SUPERGRADIENT BABSR	208.00	3922.00	6.20	276.33	3321.00	18.00	158.55	1708.00	8.00
GNN	**170.39**	2892.00	**6.07**	**185.57**	2038.00	**17.67**	**120.62**	1289.00	**7.60**

Table 3: We compare average (median) solving time, average (median) number of subdomains solved, and the percentage of properties that the methods time out on when using a cut-off time of 3600s. The best performing method for each subcategory is highlighted in bold.

F.4 GEOMETRIC MEAN

The geometric mean is less skewed than the arithmetic mean but still encapsulates a lot more information than the median, and is arguably the best suited measure to compare the different methods. The geometric mean of a set of numbers $x_1, \ldots, x_n$ is defined to be: $(\prod_{i=1}^{n} x_i)^{\frac{1}{n}}$. As shown in Table 4 the GNN is about 20% faster than supergradient ascent and more than 6 times faster than GUROBI and MIPplanet when using the geometric mean.

	Base			Wide			Deep		
Method	time(s)	subdomains	%Timeout	time(s)	subdomains	%Timeout	time(s)	subdomains	%Timeout
GUROBI BABSR	1228.08	**1015.15**	10.75	2727.59	**542.03**	50.33	2869.54	**264.58**	54.00
MIPPLANET	1217.43		36.28	2603.76		79.67	2502.48		73.60
SUPERGRADIENT BABSR	256.07	3988.68	6.20	392.87	3480.53	18.00	214.33	1750.46	8.00
GNN	**205.40**	2921.26	**6.07**	**301.34**	2419.67	**17.67**	**179.60**	1406.00	**7.60**

Table 4: We use the geometric mean to compare solving time, number of subdomains solved, and the percentage of properties that the methods time out on when using a cut-off time of 3600s. The best performing method for each subcategory is highlighted in bold.

F.5 Head to Head Performance

The following table outlines the 1-on-1 comparison between the different methods. The GNN beats each individual baseline on more than 82% of all images on the base and on the wide model and on over 74% on the deep model.

Base					
Method	Gurobi BaBSR	MIPplanet	Supergradient	GNN im 20	Best
Gurobi BaBSR		**60.44**	0.77	0.70	0.07
MIPplanet	39.56		18.10	17.96	17.96
Supergradient BaBSR	**99.23**	**81.90**		6.95	4.05
GNN	**99.30**	**82.04**	**93.05**		**77.92**

Wide					
Method	Gurobi BaBSR	MIPplanet	Supergradient	GNN im 100	Best
Gurobi BaBSR		**71.67**	1.61	1.61	0.37
MIPplanet	28.33		13.81	14.23	13.43
Supergradient BaBSR	**98.39**	**86.19**		14.57	12.31
GNN	**98.39**	**85.77**	**85.43**		**73.88**

Deep					
Method	Gurobi BaBSR	MIPplanet	Supergradient	GNN im 100	Best
Gurobi BaBSR		**61.97**	1.29	1.28	0.41
MIPplanet	38.03		9.21	8.71	8.26
Supergradient BaBSR	**98.71**	**90.79**		25.86	22.73
GNN	**98.72**	**91.29**	**74.14**		**68.60**

Table 5: We compare head-to-head performance of the different methods on the base, wide, and deep model. The numbers refer to the percentage of images on which the method in the row header beats the method in the column header. The final column represents the percentage of images on which the method beats all other methods. If two models timeout on an image than that image is not included in their head-to-head performance as neither method beats the other. The better performing method is highlighted in bold

F.6 Base Model Experiments

We now provide a more in-depth analysis of the results on the base model in Figure 2. The properties are separated into three sets based on the time t_i it takes GUROBI BaBSR to solve them: "easy" ($t_i < 800$), "medium" and a "hard " ($t_i > 2400$). The GNN is trained on easy properties only, but it beats the baselines on all three types of properties thus showing good generalization performance.

	Easy			Med			Hard		
Method	time(s)	subdomains	%Timeout	time(s)	subdomains	%Timeout	time(s)	subdomains	%Timeout
Gurobi BaBSR	550.48	**584.53**	**0.00**	1374.32	**1411.19**	**0.00**	3132.97	**2588.51**	42.12
MIPplanet	1535.05		16.71	2239.23		42.76	2241.74		45.99
Supergradient	155.12	2350.02	0.47	258.80	5304.62	0.00	1369.20	14 574.71	23.77
GNN	**131.86**	1809.59	0.47	**217.70**	4174.40	0.00	**1312.24**	13 046.03	**23.26**

Table 6: We compare average (mean) solving time, average number of subdomains solved, and the percentage of properties solved for easy, medium, and hard properties on the base model. The best performing method for each subcategory is highlighted in bold. (Note that by definition Gurobi BaBSR doesn't time out on easy and med experiments)

.

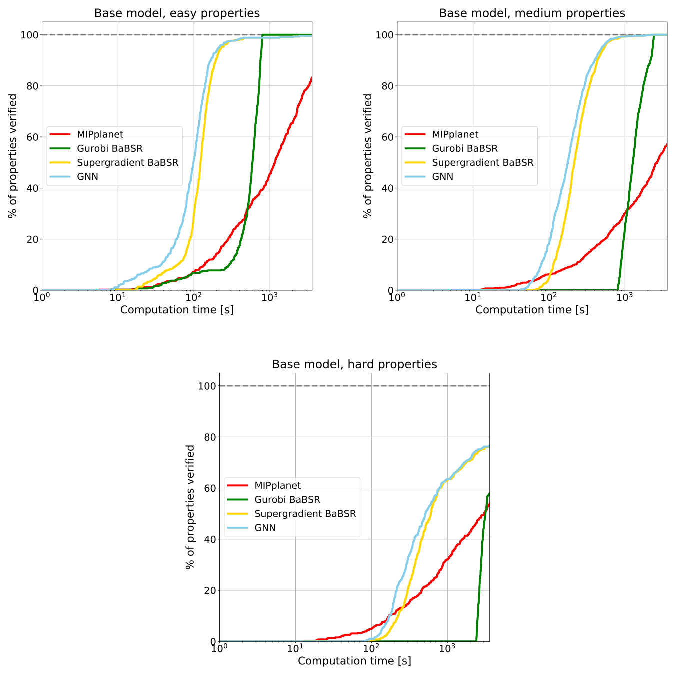

Figure 2: Cactus plots for the base model, separated into three different graphs based on the difficulty of the properties. We compare the different bounding methods by plotting the percentage of properties that have been solved for any given time.

F.7 Comparison against ERAN and branching GNN

We will now compare our results against ERAN (Singh et al., 2020), a state-of-the-art complete verification method as well as the branching GNN method proposed by Lu & Kumar (2020a). We run ERAN, GNN-Branching the baselines described above, and our methods on a subset of the OVAL dataset used in the VNN-COMP competition (VNN-COMP, 2020). Our method significantly outperforms GNN-Branching on all three models both in terms of average time taken to verify and percentage of properties successfully verified. Our method is also faster than ERAN, however ERAN times out on fewer properties. When using a large timeout of 3600s or more ERAN might be stronger method while our method is stronger when using a shorter timeout of less than 3600s.

	Base			Wide			Deep		
Method	time(s)	subdomains	%Timeout	time(s)	subdomains	%Timeout	time(s)	subdomains	%Timeout
GUROBI BABSR	2367.15	1313.41	36.00	2860.04	1072.58	48.98	2750.12	441.26	39.00
MIPPLANET	2849.10		68.00	2423.04		45.92	2302.79		40.00
SUPERGRADIENT	922.35	13 738.41	21.00	743.50	13 184.12	13.27	348.59	3672.74	5.00
ERAN	805.89		**5.00**	635.48		**9.18**	545.72		**0.00**
GNN-BRANCHING	1794.88	734.93	33.00	1360.97	397.77	15.31	1055.06	128.14	4.00
GNN	**758.28**	9875.30	17.00	**599.50**	9371.14	10.20	**285.25**	3392.71	4.00

Table 7: We compare average (mean) solving time, average number of subdomains solved, and the percentage of properties that the methods time out on when using a cut-off time of 3600s. The best performing method for each subcategory is highlighted in bold.

G Experiment Setup

We will now explain in greater detail how the experiments described in this paper were run. We first describe the training procedure used to train the two different GNNs mentioned above. We then further detail all hyperparameters used in the verification experiments.

G.1 GNN Training

We train two different GNNs, one on 20 properties and the other on 100. All of the training properties used are easy; that is BaBSR takes less than 800 seconds to solve them. We showed above that it suffices to use 20 training images to get good performance on the base model. In order to achieve better generalization performance on the wide and deep networks we had to train a second GNN on more images. We train both GNNs for three iterations, as explained above. For each iteration we train the GNN on 10,000 subdomains for a total of 50 epochs. At the start of each of the three iterations, we randomly select the subdomains to train on, choosing the same number of subdomains for each property. We aim to minimize the loss function (7) using a horizon of 100 and a decay factor $\gamma = 0.99$. We train the GNN using the Adam optimizer Kingma & Ba (2015) with a learning rate of $1e^{-2}$ and no weight decay; we manually decay the learning rate by a factor of 10 if the loss function doesn't improve for two consecutive epochs. For the update step we use an initial step size of $\mu = 1e^{-3}$, and decay it as explained above. We set the embedding size to be 32 for all GNNs. Moreover, we set $T_1 = 1$ and $T_2 = 1$. That is, we use a 2-layer MLP to initialize the embedding vectors and perform just one set of forward and backward passes. At the beginning of the first iteration we create the dataset by running the BaB algorithm using supergradient ascent and Adam to compute the lower bounds; we set the learning rate to be $1e^{-4}$. For the second and third iterations we further extend the dataset, this time using the current version of the GNN to compute the final lower bounds.

G.2 Verification Experiments

For all verification experiments mentioned in this work we run the BaB algorithm outlined in appendix A. We run 100 iterations of the GNN compared to 500 when using supergradient ascent. This together with the significant reduction in subdomains visited in the BaB algorithm when using the GNN more than compensates for the fact that one iteration of the GNN takes longer than one iteration of supergradient ascent. If the GNN performs poorly on a subdomain and the fail-safe method is used, it is likely to also not do well on the child subdomains. We therefore use supergradient ascent to solve all subdomains that result from further subdividing the current one. For all experiment we use a batch-size of 300 for both the GNN and supergradient descent. For the base experiments we store all current subdomains in memory because it is quicker; for the deep and wide models we store them as files because the experiments are more memory expensive.