Neural Lower Bounds for Verification

***Abstract*—Recent years have witnessed the deployment of branch-and-bound (BaB) frameworks for formal verification in deep learning—proving or disproving a desirable property of a neural network. The main computational bottleneck of BaB is the estimation of lower bounds via convex relaxations. 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 feasible 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. Using standard publicly available data sets, we show that our approach provides a significant speedup for formal verification compared to the state of the art solvers. Moreover, the GNN achieves good generalization performance on unseen networks.**

I. Introduction

The application of deep learning to safety critical domains such as autonomous vehicles [4] and personalized medicine [29] requires its formal verification, that is, proving or disproving its desirable properties. For instance, one desirable property is the robustness of a given network to so-called adversarial examples. These are examples that are similar to real images but ones which the neural network misclassifies with a high probability. They are obtained by applying small but deliberately chosen perturbations that are often imperceptible to the human eye.

Formal verification 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 works by hierarchically splitting the domain of the MIP into subdomains via a routine known as branching. For each subdomain it computes an upper and a lower bound of the MIP objective. If the upper bound of a subdomain is less than the lower bound of another, the latter subdomain can be pruned thereby reducing the search space for the optimal solution.

Branching is usually performed using an efficient heuristics that is either hand-designed or learnt [16, 18, 23]. The upper bound is also efficient to compute as it involves evaluating the objective for any feasible solution. In contrast, the lower bound computation requires solving a large convex relaxation. Typically, the relaxation is solved using either commercial solvers such as Gurobi [15] or traditional optimization algorithms such as subgradient descent or proximal minimization [5]. 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. In other words, lower bound estimation forms the main computational bottleneck for BaB.

A natural question that arises is why do traditional algorithms fail? We argue that by their very nature, they ignore the rich inherent structure of lower bound estimation for the problem of verification. Specifically, all lower bounds that one wishes to estimate across multiple subdomains of the same property, across multiple properties of the same network, and across multiple networks share the same form of variables, constraints and objectives. Furthermore, the coefficients of the objective and constraints are determined from network weights, which themselves are not random but are estimated using real data. Traditional optimization algorithms are agnostic to this complex high-dimensional structure as it is not "visible" to human intelligence. However, we argue that this is exactly the type of structure that can be readily exploited by artificial intelligence. To this end, we propose to use deep learning to efficiently estimate accurate lower bounds of neural networks.

Specifically, we propose the use of a graph neural network (GNN) whose architecture closely resembles that of the network we wish to verify. Given a subdomain and some initial dual variables 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. When training the GNN we consider a horizon with a decay factor to output an ascent direction that maximizes the dual objective function that directly corresponds to a lower bound on the final output of the network.

By using a parameterization of the GNN that depends only on the type of nodes and the order of the passes and not on the underlying architecture, we can train a GNN using one network and test it on another. Our approach can be used to initialize the dual problem for traditional optimization methods, but somewhat surprisingly, we found that it wasn't necessary. In all our experiments, we use our GNN-based approach as a stand alone method, that is, we directly use its lower bound in the BaB framework. 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. We consistently beat all baselines on over 80% of all images. 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. This is important as the training complexity is directly proportional to the difficulty of the training properties and the size of the model we do the training on.

II. Related Work

Neural network verification has been an active field of research in recent years with plenty of different methods being proposed in the literature [10, 17, 28, 30, 31]. Bunel et al. [6] introduced a unified framework for many of these methods by writing them as branch-and-bound frameworks (BaB). BaB is made up of three components: the branching strategy decides how to split the current domain into several smaller subdomains; the upper bound computation aims to find a counter-example of the property in each subdomain; and the computation of the lower bound, which is the focus of this work, that aims to return a certificate of robustness for each subdomain.

Most bounding methods create relaxations of the original problem. There are a variety of relaxations that have easily computable closed-form solutions such as Interval Bound Propagation (Gowal et al. [12]) or WK, the method introduced by Wong and Kolter [30]. 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: Planet introduced by Ehlers [10], Reluplex by Katz et al. [17], or the Anderson relaxation (Anderson et al. [2]). However, these often require an iterative solver to optimize them, which tends to not 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 [3, 7], and mixed integer linear programs (MILPs) [1, 11, 16, 18]. Often, different instances of these problems share a common structure, which can be exploited using learning. In the context of MILPs, learning has mostly been used to improve the branching strategies in BaB algorithms [16, 18, 23], including a recent approach that focuses on neural network verification [23]. Encouragingly, for the task of branching, the learning frameworks have been shown to outperform hand-designed heuristics. However, only limited work has been done on learning the bound computation in the BaB algorithm: Dvijotham et al. [8, 9] propose several different learned methods: one that treats every layer separately, and another that uses a simple forward-backward architecture. Gowal et al. [13] propose a method called predictor-verifier (PVT) that learns a robust training procedure and a verifier simultaneously. However, these methods end up beating Interval Bound propagation, a comparatively weak baseline, by a small margin only. There is thus a large scope for improvement which we explore in our work.

III. Background

In our work we focus on returning a lower bound on the output of the neural network we are trying to verify on a given subdomain. As the neural network is highly non-convex and thus hard to optimize over, Bunel et al. [5] introduced a decomposition based technique for the Planet relaxation [10]. We therefore begin with a brief description of the Planet relaxation [10] and the decomposition technique of [5] before describing our learning framework.

We are given a convex input domain C, an input vector $\mathbf{z}_0 \in C$, together with 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 it is widely used in machine learning in general [21, 24] as well as in neural network verification [6, 8, 10].

a) 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 $cvx\_hull_\sigma(\hat{\mathbf{z}}_k, \mathbf{z}_k, \mathbf{l}_k, \mathbf{u}_k)$, defined as follows:

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

Note that the computation of the convex hull requires the knowing of the lower and upper bounds (i.e. $\mathbf{l}_k$ and $\mathbf{u}_k$) for each intermediate node. The bounds do not have to be optimal, however, the tighter the bounds are, the tighter the relaxation will be as well. There are different ways of computing said bounds that have been proposed in the literature [12, 27, 30]. In our experiments we use the method proposed by Wong and Kolter [30]. For the sake of clarity, we introduce the following notations for the constraints corresponding to the input and the k-th layer respectively:

$$\mathscr{P}_0(\mathbf{z}_0, \hat{\mathbf{z}}_1) \equiv \begin{cases} \mathbf{z}_0 \in C \\ \hat{\mathbf{z}}_1 = W_1 \mathbf{z}_0 + \mathbf{b}_1 \end{cases}$$

and

$$\mathscr{P}_k(\hat{\mathbf{z}}_k, \hat{\mathbf{z}}_{k+1}) \equiv \begin{cases} \exists \mathbf{z}_k \text{ s.t.} \\ \mathbf{l}_k \leq \hat{\mathbf{z}}_k \leq \mathbf{u}_k \\ cvx\_hull_\sigma(\hat{\mathbf{z}}_k, \mathbf{z}_k, \mathbf{l}_k, \mathbf{u}_k) \\ \hat{\mathbf{z}}_{k+1} = W_{k+1} \mathbf{z}_k + \mathbf{b}_{k+1}. \end{cases} \quad (2)$$

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

$$\min_{\mathbf{z}, \hat{\mathbf{z}}} \hat{\mathbf{z}}_n \text{ s.t. } \mathscr{P}_0(\mathbf{z}_0, \hat{\mathbf{z}}_1); \mathscr{P}_k(\hat{\mathbf{z}}_k, \hat{\mathbf{z}}_{k+1}) \text{ for } k \in [1, \ldots, L-1]. \quad (3)$$

b) 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. [5] we will use the Lagrangian decomposition [14]. To this end, we first create two copies

$\hat{\mathbf{z}}_{A,k}, \hat{\mathbf{z}}_{B,k}$ of each variable $\hat{\mathbf{z}}_k$:

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

Next we obtain the dual by introducing Lagrange multipliers $\boldsymbol{\rho}$ corresponding to the equality constraints of the two copies of each variable:

$$\begin{aligned} q(\boldsymbol{\rho}) = \min_{\mathbf{z},\hat{\mathbf{z}}} \quad & \hat{\mathbf{z}}_{A,n} + \sum_{k=1,\dots,n-1} \boldsymbol{\rho}_k^\top (\hat{\mathbf{z}}_{B,k} - \hat{\mathbf{z}}_{A,k}) \\ \text{s.t.} \quad & \mathscr{P}_0(\mathbf{z}_0, \hat{\mathbf{z}}_{A,1}); \\ & \mathscr{P}_k(\hat{\mathbf{z}}_{B,k}, \hat{\mathbf{z}}_{A,k+1}) \ \text{ for } k \in [1,\dots,L-1]. \end{aligned} \tag{5}$$

Problem (5) is unconstrained with respect to $\boldsymbol{\rho}$. In other words any possible $\boldsymbol{\rho}$ is a feasible solution and thus by duality provides a lower bound for the primal problem (4). We therefore aim to maximize $q(\boldsymbol{\rho})$ to get the tightest possible lower bound. Given an assignment to the dual variables, Bunel et al. [5] showed that the minimization over $\mathbf{z}_0^*, \hat{\mathbf{z}}_A^*, \hat{\mathbf{z}}_B^*$ can be done efficiently. The supergradients can then be easily computed as $\nabla_{\boldsymbol{\rho}}(q) = \hat{\mathbf{z}}_B^* - \hat{\mathbf{z}}_A^*$. This is used to come up with lower bounds via supergradient ascent (see appendix X for a more detailed explanation). Unfortunately, this is known to be quite slow. We therefore take a different approach that learns to estimate a better ascent direction, thereby providing larger lower bounds more efficiently.

IV. 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. These include supergradient ascent and proximal maximization [5]. 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, and estimate the parameters using a training data set so as to exploit the problem and data structure more successfully. In what follows, we first provide an overview of our approach that achieves this generalization through graph neural networks (GNN). The remaining subsections describe the various components of the GNN and the forward and backward passes in greater detail.

A. Overview

Inspired by the approach of Lu and Kumar [23], who use a graph neural network (GNN) to come up with better branching decisions, we propose to use a GNN for efficient bound computation. Since previous bound computation approaches perform forward and backward passes on the network they wish to verify, it makes sense to use a GNN that mimics the architecture of that network as closely as possible. To this end, we treat the neural network as a graph $G_{NN} = (V_{NN}, E_{NN})$ and provide it as input for the GNN. We denote the GNN as an isomorphic graph to G_{NN}, that is, $G_{GNN} = (V_{GNN}, E_{GNN})$ where there is a one-to-one correspondence between the nodes V_{NN} and V_{GNN} and edges E_{NN} and E_{GNN}. For every node $v \in V_{GNN}$ we first compute a feature vector $\mathbf{f}$, which contains local information about the node. We then use this feature vector and a learned function g to compute an embedding vector $\boldsymbol{\mu}$, a high-dimensional vector that encapsulates a lot of the important information about the corresponding node, the structure of the neural network, and the state of the optimization algorithm. The embedding vectors are initialized based on the nodes' features and then updated using forward and backward passes in the GNN. Exchanging information with its neighbours ensures that the embedding vectors capture the global information of the structure of the problem. Once we have gotten a learned representation of each node we will convert the embedding vectors into a dual ascent direction, which will be used to update the dual variables. Having provided an overview we will now describe the GNN's main elements in greater detail.

B. GNN Components

a) Nodes.: We create a node $\mathbf{v}_k[i]$ 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}.

b) Node Features.: For each node $\mathbf{v}_k[i]$ 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:

$$\mathbf{f}_k[i] := \left(\boldsymbol{\rho}_k[i], \hat{\mathbf{z}}_{A,k}[i], \hat{\mathbf{z}}_{B,k}[i], \hat{\mathbf{z}}_{B,k}[i] - \hat{\mathbf{z}}_{A,k}[i]\right)^\top. \tag{6}$$

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.

c) 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_{ij}^k to be the edge connecting nodes $v_k[i]$ and $v_{k+1}[j]$ and assign it the value of W_{ij}^k.

d) 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:

$$\boldsymbol{\mu}_k[i] := g(\mathbf{f}_k[i]). \tag{7}$$

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:

$$\Theta_0 \in \mathbb{R}^{d \times p}, \quad \Theta_1, \dots, \Theta_{T_1} \in \mathbb{R}^{p \times p}, \quad \mathbf{b}_0, \dots, \mathbf{b}_{T_1} \in \mathbb{R}^p. \tag{8}$$

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

$$\boldsymbol{\mu}^0 = \Theta_0 \cdot \mathbf{f} + b_0, \quad \boldsymbol{\mu}^{l+1} = \Theta_{l+1} \cdot \text{relu}(\boldsymbol{\mu}^l) + \mathbf{b}_{l+1}. \tag{9}$$

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

C. 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:

$$\boldsymbol{\mu}'_k[i] = \text{relu}\Bigg(\Theta_1^{for} \boldsymbol{\mu}_k[i] + \Theta_2^{for} \left(W_k \boldsymbol{\mu}_{k-1} + \mathbf{b}_{k-1} \right)[i] + \Theta_3^{for} \left(\sum_{j \in N(i)} \boldsymbol{\mu}_{k-1}[j] / Q_{k+1}[j] \right)[i] \Bigg). \tag{10}$$

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

$$\boldsymbol{\mu}_k[i] = \text{relu}\Bigg(\Theta_1^{back} \boldsymbol{\mu}'_k[i] + \Theta_2^{back} (W_{k+1}^T \left(\boldsymbol{\mu}'_{k+1} - \mathbf{b}_{k+1} \right))[i] + \Theta_3^{back} \left(\sum_{j \in N'(i)} \boldsymbol{\mu}'_{k+1}[j] / Q'_{k+1}[j] \right)[i] \Bigg). \tag{11}$$

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 $T_2 = 1$.

D. 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 (5). 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:

$$\boldsymbol{\rho}^{t+1} = \boldsymbol{\rho}^t + \eta^{t+1} \hat{\boldsymbol{\rho}}^{t+1}. \tag{12}$$

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).

E. 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.

F. Running Standard Algorithms using the GNN

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 [5] (proof in appendix XI).

V. 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 $d_i = (\mathbf{x}, \varepsilon, W^i, \mathbf{b}^i, \mathbf{l}^i, \mathbf{u}^i, \boldsymbol{\rho}^i)$, 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 $(\mathbf{l}, \mathbf{u})$ computed via the WK method [30]; and the initial value of the dual variables $(\boldsymbol{\rho})$.

Recall that we do not use the GNN to directly compute the optimum dual solution. Instead, we run it iteratively, where each iteration computes an update direction for the dual variables. 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 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. Inference time of our GNN is shorter than supergradient ascent as we run it for significantly fewer iterations (100 and 500 respectively); so as long as the duals returned by the GNN are as good as those returned by supergradient ascent, the GNN will outperform the baseline in the BaB setting. Given the i-th training sample $d_i = (\mathbf{x}, \varepsilon, W^i, \mathbf{b}^i, \mathbf{l}^i, \mathbf{u}^i, \boldsymbol{\rho}^i) \in \mathscr{D}$, 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:

$$\mathscr{L}_i = -\sum_{t=1}^{K} q^i(\boldsymbol{\rho}^{i,t}_{GNN}) * \gamma^t * \mathbf{1}_{q^i(\boldsymbol{\rho}^{i,K}_{GNN}) < q^i_{SupG} + \kappa}. \tag{13}$$

Instead of maximizing over the dual value, we minimize over the negative dual instead. If the decay factor $\gamma \in (0,1)$ 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. Readers familiar with reinforcement learning may be reminded by the discount rates used in algorithms such as Q-learning and policy-gradient methods. We sum over the individual loss values corresponding to each data point to get the final training objective $\mathscr{L}$: $\mathscr{L} = \sum_{i=1}^{|D|} \mathscr{L}_i$.

VI. Experiments

We will now show the practical effectiveness of our method by comparing its performance with several state-of-the-art verification methods.

a) Setup.: All our experiments are performed on the test set of the CIFAR10 dataset [20]. We use the same properties used by Lu and Kumar [23]. They randomly chose the properties to verify against out of the list of false classes and compute the epsilon value that determines the size of the input domain using binary search and the BaBSR method. All properties considered are either true (i.e. the model is provably robust on the given input domain) or time out on all baselines. We will use three different neural networks to verify properties on, similar to [23]. We will use the main one, which we call the 'base model', to do all of our training and most of our testing on. It is trained robustly using the method introduced by Madry et al. [25] to achieve robustness against l_∞ perturbations of size up to ε = 8/255 (the amount typically considered in empirical works). We will further test the transferability of our GNN on two larger networks. The different network architectures are explained in greater detail in appendix XII.

b) 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. The depth, breadth and individual elements of the Branch and Bound tree depend on the branching, the upper bounding, and the lower bounding methods. Depending on the exact methods used, different subproblems are created and different ones pruned away. In other words, the subdomains found in the ith level depend on the lower bounding method used in all of the $i-1$ previous layers. This is why we cannot simulate the subdomains from the ith layer perfectly during the training of the GNN without running an entire Branch-and-Bound computation with the current version of the GNN. This would be very computationally expensive, as it would require rerunning the BaB verification on each property of the training dataset after every gradient update step in the GNN training phase. We will add this explanation to the revised version of the paper.

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 and Kumar [23] 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 [19] with no weight decay.

c) Baselines.: We compare our work to the baselines used in both Bunel et al. [6] and Lu and Kumar [23]. 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. [5] (for a more detailed description see appendix X). 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).

d) Implementation.: The implementation of our method is based on Pytorch [26]. We compute the intermediate bounds of the network using the method introduced by Wong and Kolter [30]. The two baselines using GUROBI are run on one CPU each, as done by Lu and Kumar [23]. We run both the supergradient baseline and our GNN on a single GPU and

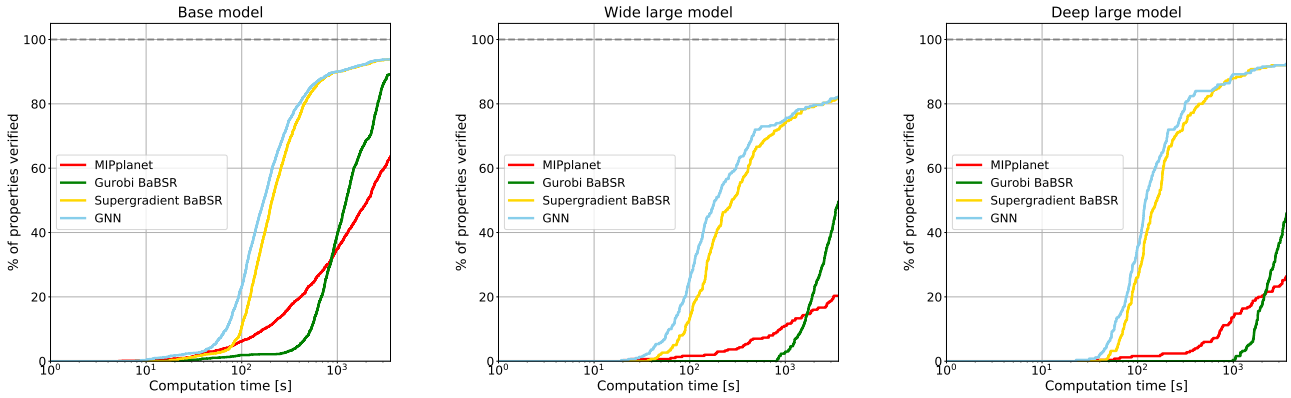

Fig. 1: Cactus plots for the base, wide and deep models. For each, we compare the bounding methods and complete verification algorithms by plotting the percentage of solved properties as a function of runtime.

	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
ADAM	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 I: 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.

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.

e) Base Model.: We first run experiments on the base model using a GNN that is trained on 20 easy properties only. We test the GNN on 425 easy properties, 704 medium properties, and 387 hard properties taken from the dataset created by Lu and Kumar [23]. The difficulty of the properties is determined by how long BaBSR takes to solve them. 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 (Table I) and times out on significantly fewer properties. Our GNN also outperforms supergradient ascent. It leads to a 10% reduction in both time taken and number of subdomains visited (and a 20% improvement when using the median or the geometric mean, see appendix XIII-A, and XIII-B). In fact, the GNN beats supergradient ascent on 93.80% of all properties as can be seen in appendix XIII-C. Even though the GNN is trained on easy properties only it generalizes well to harder ones (see appendix XIII-D).

f) 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. One of them is wider than the base network, and the other one is deeper. A more detailed description of the different architectures can be found in appendix XII. To get better generalization performance, we increase the size of the training dataset to 100 images and train a new GNN based on the base architecture. As shown in Figure 1 and Table I, the GNN still outperforms all baselines on both unseen networks. The GNN beats each baseline on over 85% of all properties (appendix XIII-C). Good generalisation performance from easy properties to difficult ones and from small networks to larger ones is beneficial as the complexity of training the GNN depends on both the difficulty of the training properties and the size of the model. Moreover, it allows us to train a single GNN and use it for many different verification tasks on various networks.

VII. 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. We've taken an important step towards creating verification methods for larger state-of-the-art networks. Further work might include extending our approach to work on different relaxations, such as the Anderson relaxation, which is tighter then Planet but has significantly more constraints. Alternatively, one could learn a lazy verifier that only solves subdomains for which there is

a high chance of pruning and further divides them into more subdomains otherwise.

VIII. Broader Impact Statement

Our work expands the applicability of deep learning to safety critical domains as it is essential to be able to verify the robustness of neural networks used in these situations. This has the potential to save lives when applied to areas like health care or autonomous driving. However, it may also lead to machines taken over jobs previously carried out by humans and can therefore lead to an increase in unemployment.

References

[1] Alejandro Marcos Alvarez, Quentin Louveaux, and Louis Wehenkel. A machine learning-based approximation of strong branching. In *INFORMS Journal on Computing*, volume 29, pages 185–195. INFORMS, 2017.

[2] 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*, pages 27–42. Springer, 2019.

[3] Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *ICLR Workshop*, 2017a.

[4] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[5] Rudy Bunel, Alessandro De Palma, Alban Desmaison, Krishnamurthy Dvijotham, Pushmeet Kohli, Philip HS Torr, and M Pawan Kumar. Lagrangian decomposition for neural network verification. *arXiv preprint arXiv:2002.10410*, 2020.

[6] 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*, pages 4790–4799, 2018.

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

[8] 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*, 2018.

[9] 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*, pages 550–559, 2018.

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

[11] 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*, pages 15554–15566, 2019.

[12] 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.

[13] 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*, pages 6156–6160. AAAI Press, 2019.

[14] Monique Guignard and Siwhan Kim. Lagrangean decomposition: A model yielding stronger lagrangean bounds. *Mathematical programming*, 39(2):215–228, 1987.

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

[16] 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.

[17] 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*, pages 97–117. Springer, 2017.

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

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

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

[21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

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

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

[24] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3, 2013.

[25] 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.

[26] 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.

[27] Aditi Raghunathan, Jacob Steinhardt, and Percy Liang. Certified defenses against adversarial examples. In *International Conference on Learning Representations*, 2018.

[28] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. Boosting robustness certification of neural networks. In *International Conference on Learning Representations*, 2019.

[29] Jeremy C Weiss, Sriraam Natarajan, Peggy L Peissig, Catherine A McCarty, and David Page. Machine learning for personalized medicine: Predicting primary myocardial infarction from electronic health records. *AI Magazine*, 33(4):33–33, 2012.

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

[31] Radosiaw R Zakrzewski. Verification of a trained neural network accuracy. In *IJCNN'01. International Joint Conference on Neural Networks. Proceedings (Cat. No. 01CH37222)*, volume 3, pages 1657–1662. IEEE, 2001.

IX. 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 and Kumar [23].

Algorithm 1 Branch and Bound

```
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
        (_ , prob) ← pick_out(probs)                        ▷ the pick_out function picks an ambiguous ReLU to split on
        [subprob_1, subprob_2] ← split(prob)
        for i = 1, 2 do
            sub_lb ← compute_LB(net, subprob_i)
            sub_ub ← compute_UB(net, subprob_i)
            if sub_ub < 0 then
                return SAT                                  ▷ we've found an adversarial example
            end if
            if sub_lb < 0 then
                probs.append((sub_lb, subprob_i))
            end if                                          ▷ if sub_lb > 0 then the subdomain gets pruned away
        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
```

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...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...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
```

X. Supergradient Method

We will now outline the supergradient ascent method used in Bunel et al. [5].

Algorithm 3 Supergradient method

function SUPERG_COMPUTE_BOUNDS($\{W_k, \mathbf{b}_k, \mathbf{l}_k, \mathbf{u}_k\}_{k=1..n}$)
  Initialise dual variables $\boldsymbol{\rho}^0$ using the duals of the parent domain or the algo of Wong and Kolter [30]
  for `nb_iterations` **do**
    $\hat{\mathbf{z}}^*, \hat{\mathbf{z}}^*_A \hat{\mathbf{z}}^*_B \leftarrow$ inner minimization as proposed by Bunel et al. [5]
    Compute supergradient using $\nabla_{\boldsymbol{\rho}} q(\boldsymbol{\rho}^t) = \hat{\mathbf{z}}^*_B - \hat{\mathbf{z}}^*_A$
    $\boldsymbol{\rho}^{t+1} \leftarrow$ Adam's update rule [19]
  end for
  return $q(\boldsymbol{\rho})$
end function

XI. Regaining Supergradient Ascent

We show that our method is strictly more expressive than supergradient ascent by showing that it can simulate it exactly. 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

$$\hat{\boldsymbol{\rho}}^{t+1}_k = \hat{\mathbf{z}}_{B,k} - \hat{\mathbf{z}}_{A,k}. \tag{14}$$

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

$$\boldsymbol{\mu}^0_k = \left(\hat{\mathbf{z}}_{B,k} - \hat{\mathbf{z}}_{A,k}, -\hat{\mathbf{z}}_{B,k} + \hat{\mathbf{z}}_{A,k}, \mathbf{0}, \ldots, \mathbf{0}\right)^\top, \tag{15}$$

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

If we set $\Theta^{for}_2 = \Theta^{for}_3 = \Theta^{back}_2 = \Theta^{back}_3 = \mathbf{0}$ and $\Theta^{for}_1 = \Theta^{back}_1 = \mathbb{1}$, then the forward and backward passes don't change the embedding vector. We now just need to set $\boldsymbol{\Theta}^{out} = (1, -1, 0, \ldots, 0)^\top$ to get the final ascent direction:

$$\hat{\boldsymbol{\rho}}^{t+1}_k = \left(\hat{\mathbf{z}}_{B,k} - \hat{\mathbf{z}}_{A,k}\right)_+ + \left(\hat{\mathbf{z}}_{B,k} - \hat{\mathbf{z}}_{A,k}\right)_- = \hat{\mathbf{z}}_{B,k} - \hat{\mathbf{z}}_{A,k}. \tag{17}$$

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

XII. Network Architectures

We perform all our experiments on the same three neural networks used by Lu and Kumar [23]. All three networks are trained robustly using the method introduced by Madry et al. [25] 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 and Kumar [23]: 467) Medium: 704 (Lu and Kumar [23]: 773) Hard: 387 (Lu and Kumar [23]: 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 II: Network Architectures

XIII. Further Experimental Results

We will now compare our method with the different baselines in greater depth. We first use different statistics to more accurately explain the performances of the different methods.

A. 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
Adam	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 III: 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.

B. 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: $\left(\prod_{i=1}^{n} x_i\right)^{\frac{1}{n}}$. As shown in Table IV 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
ADAM	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 IV: 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.

C. 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	**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	**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	**98.71**	**90.79**		25.86	22.73
GNN	**98.72**	**91.29**	**74.14**		**68.60**

TABLE V: 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

D. 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.

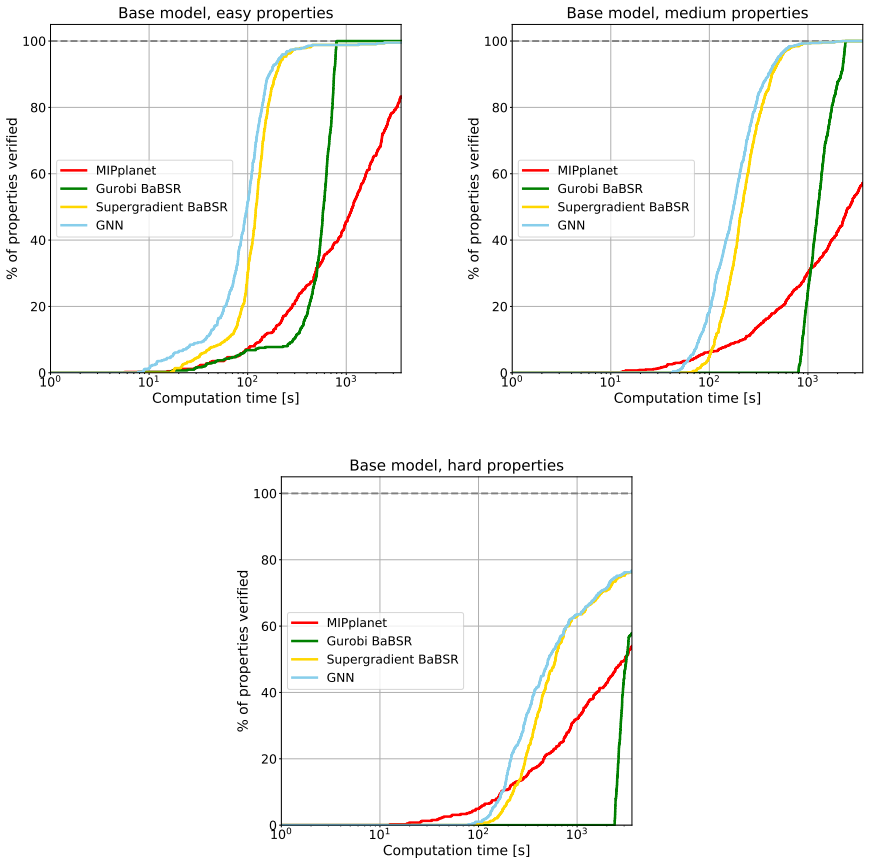

Fig. 2: Cactus plots for the base model, separated into three different graphs based on the difficulty of the properties. We compare the different bounding methods by plotting the percentage of properties that have been solved for any given time.

	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 VI: 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)

XIV. 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.

A. 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 (13) using a horizon of 100 and a decay factor $\gamma = 0.99$. We train the GNN using the Adam optimizer [19] 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.

B. Verification Experiments

For all verification experiments mentioned in this work we run the BaB algorithm outlined in appendix IX. 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.

XV. Comparison to Previous Work

Whilst [22] and our work both focus on using GNN for neural network verification problems, there are several key differences between the two papers. Firstly, they focus on different elements of the Branch-and-Bound algorithm. Whilst our method outputs an ascent direction for the dual variables, the branching GNN returns a branching score for each ReLU node. The two papers use different loss functions: we train the GNN in a self-supervised way by optimise the dual objective, whilst [22] train their GNN in a supervised manner based on the actual improvement score, that evaluates the possible gain from branching at each node. They aim to imitate the strong branching strategy. Another key difference between the two methods is the training dataset for the GNN, as they use the strong branching heuristic to create some training properties which is very expensive. The two papers also differ in the GNN architectures they use: they have different feature vectors, and different types of forward and backward update steps. Finally, Lu and Kumar [22] use online-learning to fine-tune their GNN at inference time, which we do not.

XVI. GNN

We now present the GNN Bounding Method as an algorithm below.

Algorithm 4 GNN Bounding

Input: natural image $\mathbf{x}$, perturbation norm ε, weights and biases $W, \mathbf{b}$, lower and upper bounds $\mathbf{l}, \mathbf{u}$, dual variables $\boldsymbol{\rho}$
function GNN($\mathbf{x}, \varepsilon, W^i, \mathbf{b}^i, \mathbf{l}^i, \mathbf{u}^i, \boldsymbol{\rho}^i$)
  $\boldsymbol{\rho}^0 := \boldsymbol{\rho}$ ▷ Initialize dual variables
  for $t = 0, \cdots, K-1$ **do**
    $\mathbf{f}_k[i] := \left(\boldsymbol{\rho}_k[i], \hat{\mathbf{z}}_{A,k}[i], \hat{\mathbf{z}}_{B,k}[i], \hat{\mathbf{z}}_{B,k}[i] - \hat{\mathbf{z}}_{A,k}[i]\right)^\top$. ▷ Initialize Feature vectors for all layers k and nodes i
    $\boldsymbol{\mu}_k[i] := g(\mathbf{f}_k[i])$ ▷ Initialize Embedding vectors for all layers k and nodes i
    $\boldsymbol{\mu}_k[i] := F(\boldsymbol{\mu}_{k-1}, \boldsymbol{\mu}_k, W_k)$ ▷ Forward Pass as defined in (10) from the second to the last layer
    $\boldsymbol{\mu}_k[i] := B(\boldsymbol{\mu}_k, \boldsymbol{\mu}_{k+1}, W_{k+1})$ ▷ Backward pass as defined in (11) from the penultimate to the first layer
    $\hat{\boldsymbol{\rho}}_k^{t+1} = \boldsymbol{\Theta}^{out} \boldsymbol{\mu}_k$. ▷ Compute a new ascent direction
    $\boldsymbol{\rho}^{t+1} = \boldsymbol{\rho}^t + \eta^{t+1} \hat{\boldsymbol{\rho}}^{t+1}$ ▷ Compute new dual variables
  end for
  if $q(\boldsymbol{\rho}^K) > 0$ **then**
    return UNSAT
  else
    return $\boldsymbol{\rho}^K$ and $q(\boldsymbol{\rho}^K)$
  end if
end function