NEURAL NETWORK BRANCHING FOR NEURAL NETWORK VERIFICATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Formal verification of neural networks is essential for their deployment in safety-critical areas. Many available formal verification methods have been shown to be instances of a unified Branch and Bound (BaB) formulation. We propose a novel framework for designing an effective branching strategy for BaB. Specifically, we learn a graph neural network (GNN) to imitate the strong branching heuristic behaviour. Our framework differs from previous methods for learning to branch in two main aspects. Firstly, our framework directly treats the neural network we want to verify as a graph input for the GNN. Secondly, we develop an intuitive forward and backward embedding update schedule. Empirically, our framework achieves roughly 50% reduction in both the number of branches and the time required for verification on various convolutional networks when compared to the best available hand-designed branching strategy. In addition, we show that our GNN model enjoys both horizontal and vertical transferability. Horizontally, the model trained on easy properties performs well on properties of increased difficulty levels. Vertically, the model trained on small neural networks achieves similar performance on large neural networks.

1 INTRODUCTION

Despite their outstanding performances on various tasks, neural networks are found to be vulnerable to adversarial examples (Goodfellow et al., 2015; Szegedy et al., 2013). The brittleness of neural networks can have costly consequences in areas such as autonomous driving, finance and healthcare. When one requires robustness to adversarial examples, traditional model evaluation approaches, which test the trained model on a hold-out set, do not suffice. Instead, formal verification of properties such as adversarial robustness becomes necessary. For instance, to ensure self-driving cars make consistent correct decisions even when the input image is slightly perturbed, the required property to verify is that the underlying neural network outputs the same correct prediction for all points within a norm ball whose radius is determined by the maximum perturbation allowed.

Several methods have been proposed for verifying properties on neural networks (NN). Bunel et al. (2018) showed that many of the available methods can be viewed as instances of a unified Branch and Bound (BaB) framework. A BaB algorithm consists of two key components: branching strategies and bounding methods. Branching strategies decide how the search space is recursively split into smaller space. Bounding methods compute bounds of each split space to tighten the bounds of the final objective function over the whole search space. In this work, we focus on improving the branching strategies. By directly working with a general framework, our identified algorithmic improvements can be combined with any bounding method, leading to potential performance improvement for BaB based verification algorithms.

Branching strategy has a significant impact on the overall problem-solving process, as it directly decides the total number of steps, consequently the total time, required to solve the problem at hand. The quality of a branching strategy is even more important when NN verification problems are considered, which generally have a very large search space. Each input dimension or each activation unit can be a potential branching option and neural networks of interest often have high dimensional input and thousands of hidden activation units. With such a large search space, an effective branching strategy could mean a large reduction of the total number of branches required, consequently time required to solve a problem. Developing an effective strategy is thus of significant importance to the success of BaB based NN verification.

So far, to the best of our knowledge, branching rules adopted by BaB based verification methods are either random selection (Katz et al., 2017; Ehlers, 2017) or hand-designed heuristics (Wang et al., 2018b; Bunel et al., 2018; Royo et al., 2019; Bunel et al., 2019). Random selection is generally inefficient as the distribution of the best branching decision is rarely uniform. In practice, this strategy often results in exhaustive search to make a verification decision. On the other hand, hand designed heuristics often involve a trade-off between effectiveness and computational cost. For instance, strong branching is generally one of the best performing heuristics for BaB methods in terms of the number of branches, but it is computationally prohibitive as each branching decision requires an expensive exhaustive search over all possible options. The heuristics that are currently used in practice are either inspired by the corresponding dual problem when verification is formulated as an optimization problem (Bunel et al., 2018; Royo et al., 2019) or incorporating the gradient information of the neural network (Wang et al., 2018b). These heuristics normally have better computational efficiency. However, given the complex nature of the search space, it is unlikely that any hand-designed heuristics is able to fully exploit the structure of the problem and the data distribution encountered in practice. As mentioned earlier, for large size NN verification problems, a slight reduction of branching quality strategy could lead to exponential increase in the total number of branches required to solve the problem. A computationally cheap but high quality branching strategy is thus much needed.

In order to exploit the inherent structure of the problem and the data, we propose a novel machine learning framework for designing a branching strategy. Our framework is both computational efficient and effective, giving branching decisions that are of a similar quality to that of strong branching. Specifically, we make following contributions:

- We use a graph neural network (GNN) to exploit the structure of the neural network we want to verify. The embedding vectors of the GNN are updated by a novel schedule, which is both computationally cheap and memory efficient. In detail, we mimic the forward and backward passes of the neural network to update the embedding vectors. In addition, the proposed GNN allows a customised schedule to update embedding vectors via shared parameters. That means, once training is done, the trained GNN model is applicable to various verification properties on different neural network structures.
- We train GNNs via supervised learning. We provide ways to generate training data cheaply but inclusive enough to represent branching problems at different stages of a BaB process for various verification properties. With the ability to exploit the neural network structure and a comprehensive training data set, our GNN is easy to train and converges fast.
- Our learned GNN also enjoys transferability both horizontally and vertically. Horizontally, although trained with easy properties, the learned GNN gives similar performance on medium and difficult level properties. More importantly, vertically, given that all other parts of BaB algorithms remain the same, the GNN trained on small networks performs well on large networks. Since the network size determines the total cost for generating training data and is positively correlated with the difficulty of learning, this vertical transferability allows our framework to be readily applicable to large scale problems.
- We further enhance our framework via online learning. For a learned branching strategy, it is expected that the strategy can fail to output satisfactory branching decisions from time to time. To deal with this issue, we provide an online scheme for fine-tuning the GNN along the BaB process in order to best accommodate the verification property at hand.
- Finally, we supply a dataset on convolutional NN verification problems, covering problems at different difficulty levels over neural networks of different sizes. We hope that by providing a large problem dataset it could allow easy comparisons among existing methods and additionally encourage the development of better methods.

Since most verification methods available work on ReLU-based deep neural networks, we focus on neural networks with ReLU activation units in this paper. However, we point out that our framework is applicable to any neural network architecture.

2 Related Works

Learning has already been used in solving combinatorial optimization (Bello et al., 2016; Dai et al., 2017) and mixed integer linear programs (MILP) (Khalil et al., 2016; Alvarez et al., 2017; Hansknecht et al., 2018; Gasse et al., 2019). In these areas, instances of the same underlying structure are solved multiple times with different data value, which opens the door for learning. Among them, Khalil et al. (2016), Alvarez et al. (2017), Hansknecht et al. (2018), and Gasse et al. (2019) proposed learned branching strategies for solving mixed integer programs with BaB algorithms. All of these methods imitate the strong branching strategy. Specifically, Khalil et al. (2016) and Hansknecht et al. (2018) learn a ranking function to rank potential branching decisions while Alvarez et al. (2017) uses regression to assign a branching score to each potential branching choice. There are two main issues with these methods. Firstly, they rely heavily on hand-designed features and secondly, they use generic learning structure which is unable to exploit the neural network architecture.

The most relevant to ours is the recent concurrent work by Gasse et al. (2019). Gasse et al. (2019) managed to reduce feature reliance by exploiting the bipartite structure of MILP through GNN. Bipartite graph is capable of capturing the architecture information but is memory-consuming, requiring recording matrices that grow linearly with the total number of nodes of the neural network. This can be a serious issue for large scale NN verification problems. Our proposed framework is specifically designed for NN verification problems. By exploiting the neural network structure, and designing a customized schedule for embedding updates, our framework is able to scale elegantly both in terms of computation and memory.

3 Background

Formal verification of neural networks refers to the problem of proving or disproving a property over a bounded input domain. Properties are functions of neural network outputs. When a property can be expressed as a Boolean expression over linear forms, we can modify the neural network in a suitable way so the property can be simplified to checking the sign of the neural network output (Bunel et al., 2018). Note that all the properties studied in previous works satisfy this form, thereby allowing us to use the aforementioned simplification. Mathematically, given the modified neural network f, a bounded input domain $\mathcal{C}$, formal verification examines the truthfulness of the following statement:

$$\forall \boldsymbol{x} \in \mathcal{C}, \qquad f(\boldsymbol{x}) \geq 0. \tag{1}$$

If the above statement is true, the property holds. Otherwise, the property does not hold.

3.1 Branch and Bound

Verification tasks are often treated as a global optimization problem. We want to find the minimum of $f(\boldsymbol{x})$ over $\mathcal{C}$ in order to compare with the threshold 0. Specifically, we consider an L layer feed-forward neural network, $f : \mathbb{R}^{|\boldsymbol{x}|} \rightarrow \mathbb{R}$, with ReLU activation units such that for any $\boldsymbol{x}_0 \in \mathcal{C} \subset \mathbb{R}^{|\boldsymbol{x}|}$, $f(\boldsymbol{x}_0) = \hat{\boldsymbol{x}}_L \in \mathbb{R}$ where

$$\hat{\boldsymbol{x}}_{i+1} = W^{i+1}\boldsymbol{x}_i + \boldsymbol{b}^{i+1}, \qquad \text{for } i = 0, \ldots, L-1, \tag{2a}$$

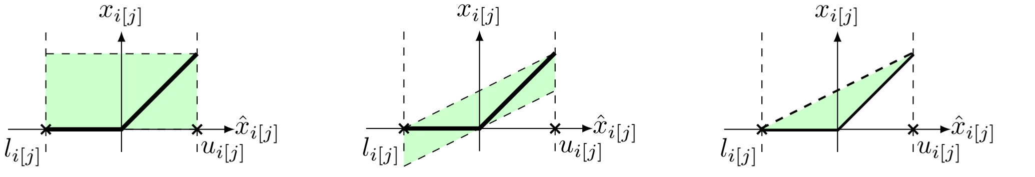
$$\boldsymbol{x}_i = \max(\hat{\boldsymbol{x}}_i, 0), \qquad \text{for } i = 1, \ldots, L-1. \tag{2b}$$

The terms W^i and $\boldsymbol{b}^i$ refer to the weights and biases of the i-th layer of the neural network f. Domain $\mathcal{C}$ can be an ℓ_p norm ball with radius ϵ. Finding the minimum of f is a challenging task, as the optimization problem is generally NP hard (Katz et al., 2017). To deal with the inherent difficulty of the optimization problem itself, BaB (Bunel et al., 2018) is generally adopted. In detail, BaB based methods divide $\mathcal{C}$ into sub-domains (branching), each of which defines a new sub-problem. They compute a relaxed lower bound of the minimum (bounding) on each sub-problem. The minimum of lower bounds of all generated sub-domains constitutes a valid global lower bound of the global minimum over $\mathcal{C}$. As a recursive process, BaB keeps partitioning the sub-domains to tighten the global lower bound. The process terminates when the computed global lower bound is above zero (property is true) or when an input with a negative output is found (property is false). A detailed description of the BaB is provided in the appendices. In what follows, we provide a brief description of the two components, bounding methods and branching strategies, that is necessary for the understanding of our novel learning framework.

3.2 Bounding

For NN verification problems, bounding consists of finding upper and lower bounds for the final output, the minimum of $f(\boldsymbol{x})$ over $\mathcal{C}$. An effective technique to compute a lower bound is to transform the original optimization problem into a linear program (LP) by introducing convex relaxations over ReLU activation units. As we can see in Eq. (2b), ReLU activation units do not define a convex feasible set, and hence, relaxations are needed. Denote the j-th element of the vector $\boldsymbol{x}_i$ as $x_{i[j]}$. Possible convex relaxations for a hidden node $x_{i[j]}$ that have been introduced so far are shown in

Figure 1. We replace ReLU with the shaded green area. The tighter the convex relaxation introduced, the more computational expensive it is to compute a bound but the tighter the bound is going to be. From Figure 1, we note that in order to introduce a convex relaxation, we need intermediate bounds $l_{i[j]}$ and $u_{i[j]}$. Thus intermediate bounds are required for building the LP for the final output lower bound. Given their purpose and the large number of intermediate bound computations, rough estimations are mainly used. On the other hand, the final output lower bound is directly used in making the pruning decision and hence a tighter lower bound is preferred as it avoids further unnecessary split on the sub-problem.



(a) Naive relaxation (b) Linear bounds relaxation (c) Planet relaxation (Ehlers, 2017)

Figure 1: Different convex relaxations introduced. For each plot, the black line shows the output of a ReLU activation unit for any input value between $l_{i[j]}$ and $u_{i[j]}$ and the green shaded area shows the convex relaxation introduced. Naive relaxation (a) is the loosest relaxation. Linear bounds relaxation (b) is tighter and is introduced in Weng et al. (2018). Finally, Planet relaxation (c) is the tightest linear relaxation among the three considered (Ehlers, 2017). Among them, (a) and (b) have closed form solutions which allow fast computations while (c) requires an iterative procedure to obtain an optimal solution.

3.3 BRANCHING

Branching is of equal importance as bounding in the Branch and Bound framework. Especially for large scale networks f, each branching step has a large number of putative choices. In these cases, the effectiveness of a branching strategy directly determines the possibility of verifying properties over these networks within a given time limit. On neural networks, two types of branching decisions are used: input domain split and hidden activation unit split.

Assume we want to split a parent domain $\mathcal{D}$. Input domain split selects an input dimension and then makes a cut on the selected dimension while the rest of the dimensions remaining the same as the parent for the two sub-domains. The common choice is to cut the selected dimension in half and the dimension to cut is decided by the branching strategy used. Available input domain split strategies are Bunel et al. (2018) and Royo et al. (2019). Royo et al. (2019) is based on sensitivity test of the LP on $\mathcal{D}$ while Bunel et al. (2018) uses the formula provided in Wong & Kolter (2018) to estimate final output bounds for sub-domains after splitting on each input dimension and selects the dimension that results in the highest output lower bound estimates.

In our setting, we refer to a ReLU activation unit $x_{i[j]} = \max(\hat{x}_{i[j]}, 0)$ as ambiguous over $\mathcal{D}$ if the upper bound $u_{i[j]}$ and the lower bound $l_{i[j]}$ for $\hat{x}_{i[j]}$ have different signs. Activation unit split chooses among ambiguous activation units and then divides the original problem into cases of different activation phase of the chosen activation unit. If a branching decision is made on $x_{i[j]}$, we divide the ambiguous case into two determinable cases: $\{x_{i[j]} = 0, l_{i[j]} \leq \hat{x}_{i[j]} \leq 0\}$ and $\{x_{i[j]} = \hat{x}_{i[j]}, 0 \leq \hat{x}_{i[j]} \leq u_{i[j]}\}$. After the split, the originally introduced convex relaxation is removed, since the above sets are themselves convex. We expect large improvements on output lower bounds of the newly generated sub-problems if a good branching decision is made. Apart from random selection, employed in Ehlers (2017) and Katz et al. (2017), available ReLU split heuristics are Wang et al. (2018a) and Bunel et al. (2019). Wang et al. (2018a) computes scores based on gradient information to prioritise ambiguous ReLU nodes. Similarly, Bunel et al. (2019) uses scores to rank ReLU nodes but scores are computed with a formula developed on the estimation equations in Wong & Kolter (2018). We note that for both branching strategies, after the split, intermediate bounds are updated accordingly on each new sub-problem. For NN verification problems, either domain split or ReLU split can be used at each branching step. When compared with each other, ReLU split is normally a more effective choice for large scale networks, as shown in Bunel et al. (2019).

All the aforementioned existing branching strategies use hand-designed heuristics. In contrast, we propose a new framework for branching strategies by utilizing a GNN to learn to imitate strong branching heuristics. This allows us to harness both the effectiveness of strong branching strategies and the efficiency of GPU computing power.

4 GNN FRAMEWORK

Overview We begin with a brief overview of our overall framework, followed by a detailed description of each of its components. A graph neural network G is represented by two components: a set of nodes V and a set of edges E, such that $G = (V, E)$. Each node and each edge has its set of features. A GNN uses the graph structure and node and edge features to learn a representation vector (embedding vector) for each node $v \in V$. GNN is a key component of our framework, in which we treat the neural network f as a graph G_f. A GNN takes G_f as an input and initializes an embedding vector for each node in V. The GNN updates each node's embedding vector by aggregating its own node features and all its neighbour embedding vectors. After several rounds of updates, we obtain a learned representation (an updated embedding vector) of each node. To make a branching decision, we treat the updated embedding vectors as inputs to a score function, which assigns a score for each node that constitutes a potential branching option. A branching decision is made based on the scores of potential branching decision nodes. Our framework is visualised in Figure 2. We now describe each component in detail.

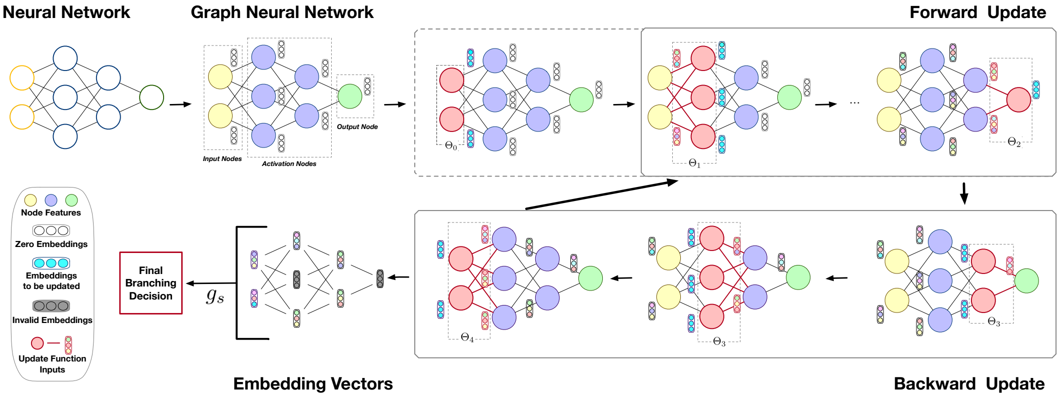


Figure 2: Illustration of our proposed GNN framework. An all zeros embedding network mimicking the neural network is initialised. Embedding vectors are updated via several rounds of forward backward passes using updating Eqs. (3)-(7). We obtain the final branching decision by calling a score function g_s over all embedding vectors of potential branching decision nodes.

Nodes Given a neural network f, V consists of all input nodes $v_{0[j]}$, all hidden activation nodes $v_{i[j]}$ and an output node v_L. In our framework, we combine every pre-activation variable and its associated post-activation variable and treat them as a single node. Pre- and post-activation nodes together contain the information about the amount of convex relaxation introduced at this particular activation unit, so dealing with the combined node simplifies the learning process. In terms of the Eq. (2), let $x'_{i[j]}$ denote the combined node of $\hat{x}_{i[j]}$ and $x_{i[j]}$. The nodes $v_{0[j]}$, $v_{i[j]}$ and v_L are thus in one-to-one correspondence with $x_{0[j]}$, $x'_{i[j]}$ and x_L. We note that V is larger than the set of all potential branching decisions as it includes unambiguous activation nodes and output nodes.

Node features Different types of nodes have different sets of features. In particular, input node features contain the corresponding domain lower and upper bounds and primal solution. For activation nodes, the node features consist of associated intermediate lower and upper bounds, layer bias, primal and dual solutions and new terms computed using previous features. Finally, the output node has features including the associated output lower and upper bounds, layer bias and primal solution. Other types of features could be used and some features could be excluded if they are expensive to compute. We denote input node features as $\boldsymbol{z}_{0[j]}$, activation node features as $\boldsymbol{z}_{i[j]}$ and output node feature as $\boldsymbol{z}_L$. Our framework uses simple node features and does not rely on extensive feature engineering. Nonetheless, by relying on the powerful GNN framework, it provides highly accurate branching decisions.

Edges E consists of all edges connecting nodes in V, which are exactly the connecting edges in f. Edges are characterized by the weight matrices that define the parameters of the network f such that for an edge e^i_{jk} connecting $x'_{i[j]}$ and $x'_{i+1[k]}$, we assign $e^i_{jk} = W^i_{jk}$.

Embeddings We associate a p-dimensional embedding vector $\boldsymbol{\mu}_v$ for each node $v \in V$. All embedding vectors are initialised as zero vectors.

Forward and Backward embedding updates In general, a graph neural network learns signals from a graph by acting as a function of two inputs: a feature matrix $\mathcal{X} \in \mathbb{R}^{|V| \times p}$, where each row is the embedding vector $\boldsymbol{\mu}_v$ for a node $v \in V$, and an adjacency matrix $\mathcal{A}$ representing the graph structure. Under this formulation, all node embedding vectors are updated at the same time and there is no particular order between nodes. In this work, instead, we propose an update scheme where only the nodes corresponding to the same layer of the network f are updated at the same time, so embedding vector updates are carried out in a layer-by-layer forward-backward way.

We argue that the forward-backward updating scheme is a natural fit for our problem. In more detail, for a given domain $\mathcal{D}$, each branching decision (an input node or an ambiguous activation node) will generate two sub-domains s_1 and s_2, with each sub-domain having an output lower bound $l^L_{s_1}$, $l^L_{s_2}$ equal to or higher than the lower bound $l^L_{\mathcal{D}}$ that of $\mathcal{D}$. Strong branching heuristic uses a predetermined function to measure the combined improvement of $l^L_{s_1}$, $l^L_{s_2}$ over $l^L_{\mathcal{D}}$ and makes the final branching decision by selecting a node that gives the largest improvement. Thus, to maximise the performance of a graph neural network, we want a node embedding vector to maximally capture all information related to computing $l^L_{s_1}$, $l^L_{s_2}$ if a split is conducted on the node. For estimating $l^L_{s_1}$, $l^L_{s_2}$ of splitting on a potential branching decision node v, we note that these values are closely related to two factors. The first factor is the amount of convex relaxations introduced at a branching decision node v, if v corresponds to an ambiguous activation node. The second factor considers that if the node v is split, the impact it will have on the convex relaxations introduced to nodes on layers after that of v. Recall that, if there are no ambiguous activation nodes, the neural network f is simply a linear operator, whose minimum value can be easily obtained. When ambiguous activation nodes are present, the total amount relaxation introduced determines the tightness of the lower bound to f. We thus treat embedding vectors as a measure of local convex relaxation and its contribution to other nodes' convex relaxation.

As shown in Figure 1, at each ambiguous activation node $x'_{i[j]}$, the area of convex relaxation introduced is determined by the lower and upper bounds of the pre-activate node $\hat{x}_{i[j]}$. Since intermediate lower and upper bounds of a node $\hat{x}_{i[j]}$ are completely decided by the layers prior to it and have to be computed in a layer-by-layer fashion, we respect this structure and the order of computation by utilising a forward layer-by-layer update on node embedding vectors, which should allow these embedding vectors to capture the local relaxation information. In terms of the impact of local relaxation change to that of other nodes, we note that by splitting an ambiguous node into two fixed cases, all intermediate bounds of nodes on later layers will affected, leading to relaxation changes at those nodes. We thus employ a backward layer-by-layer update to account for the impact the local change has over other nodes. Theoretically, by fixing an ambiguous ReLU node, intermediate bounds of nodes at previous layers and on the same layer might change as well. For a naturally trained neural network, these changes for these nodes should be relatively small compared to nodes on the later layers. To account for these changes, we rely on multiple rounds of forward-and-backward updates.

In summary, during the forward update, for $i = 1, \ldots, L-1$, we have, for all possible j,

$$\boldsymbol{\mu}_{0[j]} \longleftarrow F_{inp}(\boldsymbol{z}_{0[j]}; \boldsymbol{\theta}_0), \quad \text{if } \boldsymbol{\mu}_{0[j]} = \mathbf{0}, \tag{3}$$

$$\boldsymbol{\mu}_{i[j]} \longleftarrow F_{act}(\boldsymbol{z}_{i[j]}, \boldsymbol{\mu}_{i-1}, e^i; \boldsymbol{\theta}_1), \tag{4}$$

$$\boldsymbol{\mu}_{L} \longleftarrow F_{out}(\boldsymbol{z}_{L}, \boldsymbol{\mu}_{L-1}, e^L; \boldsymbol{\theta}_2) \tag{5}$$

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

$$\boldsymbol{\mu}_{i[j]} \longleftarrow B_{act}(\boldsymbol{z}_{i[j]}, \boldsymbol{\mu}_{i+1}, e^{i+1}; \boldsymbol{\theta}_3), \tag{6}$$

$$\boldsymbol{\mu}_{0[j]} \longleftarrow B_{inp}(\boldsymbol{z}_{0[j]}, \boldsymbol{\mu}_{1}, e^{1}; \boldsymbol{\theta}_4). \tag{7}$$

Update functions F and B take the form of multi-layered fully-connected networks with ReLU activation functions or composites of these simple update networks. The terms $\boldsymbol{\theta}_i$ denote the parameters of the networks. A detailed description of update functions is provided in the appendices.

We point out that our forward-backward update scheme does not depend on the underlying neural network structure and thus should be generalizable to network architectures that differ from the one we use for training. However, it does rely on the information used to compute convex relaxation, so underlying data distribution, features and bounding methods are assumed to be same when the trained model is applied to different networks. Furthermore, our forward-backward update is memory efficient, as we are dealing with one layer at a time and only the updated embedding vectors of the layer are used to update the embedding vectors in the next (forward-pass) and the previous (backward-pass) layer. This makes it readily applicable to large networks.

Scores At the end of the forward-backward updates, embedding vectors for potential branching decision nodes (all input nodes and ambiguous activation nodes) are gathered and treated as inputs for a score function $g_s(\cdot; \boldsymbol{\theta}_5) : \mathbb{R}^p \rightarrow \mathbb{R}$, which takes the form of a fully-connected network with parameters $\boldsymbol{\theta}_5$. It assigns a scalar score for each input embedding vector. The final branching decision is determined by picking the node with the largest score.

5 Parameter Estimation

Training We train a GNN via supervised learning. To estimate $\boldsymbol{\Theta} := (\boldsymbol{\theta}_0, \boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \boldsymbol{\theta}_3, \boldsymbol{\theta}_4, \boldsymbol{\theta}_5)$, we propose a new hinge rank loss function that is specifically designed for our framework. Before we give details of the loss, we introduce a relative improvement measure m first. Given a domain $\mathcal{D}$, for each branching decision node v, the two generated sub-problems have output lower bounds $l^L_{s_1}$ and $l^L_{s_2}$. We measure the relative improvement of splitting at the node v over the output lower bound $l^L_{\mathcal{D}}$ as follows

$$m_v := (\min(l^L_{s_1}, 0) + \min(l^L_{s_2}, 0) - 2 \cdot l^L_{\mathcal{D}})/(-2 \cdot l^L_{\mathcal{D}}). \tag{8}$$

Intuitively, m ($0 \leq m \leq 1$) measures the average relative sub-problem lower bound improvement to the maximum improvement possible, that is $-l^L_{\mathcal{D}}$. Any potential branching decision node v can be compared and ranked via its relative improvement value m_v. Since we are only interested in branching nodes with large improvement measures, ranking loss is a natural choice. Direct pairwise rank loss might be difficult to learn for NN verification problems, given the large number of branching decision nodes on each domain $\mathcal{D}$. In addition, many branching decisions may give similar performance, so it is redundant and potentially harmful to the learning process if we learn a ranking among these similar nodes. To deal with these issues, we develop our loss by first dividing all potential branching nodes into M classes (M is much smaller than the total number of branching decision nodes) through the improvement value m_v of a node. We denote the class label as Y_v for a node v. Labels are assigned in an ascending order such that $Y_v >= Y_{v'}$ if $m_v > m_{v'}$. We then compute the pairwise hinge-rank loss on these newly assigned labels as

$$loss_{\mathcal{D}}(\boldsymbol{\Theta}) = \frac{1}{K} \sum_{i=1}^{N} \big(\sum_{j=1}^{N} \phi(g_s(\boldsymbol{\mu}_j; \boldsymbol{\Theta}) - g_s(\boldsymbol{\mu}_i; \boldsymbol{\Theta})) \cdot \mathbf{1}_{\mathrm{Y_j > Y_i}} \big), \tag{9}$$

where $\phi(z) = (1-z)_+$ is the hinge function, N is the total number of branching decision nodes and K is total number of pairs where $Y_j > Y_i$ for any branching decision nodes v_i, v_j. The loss measures the average hinge loss on score difference $(g_s(\boldsymbol{\mu}_j; \boldsymbol{\Theta}) - g_s(\boldsymbol{\mu}_i; \boldsymbol{\Theta}))$ for all pairs of branching decision nodes v_i, v_j such that $Y_j > Y_i$. Finally, we evaluate $\boldsymbol{\Theta}$ by solving the following optimization problem:

$$\boldsymbol{\Theta} = \arg\min_{\boldsymbol{\Theta}} \frac{\lambda}{2} \|\boldsymbol{\Theta}\|^2 + \frac{1}{n} \sum_{i}^{n} loss_{\mathcal{D}_i}(\boldsymbol{\Theta}), \tag{10}$$

where the $loss_{\mathcal{D}_i}$ is the one introduced in Eq. (9) and n is the number of training samples.

Fail-safe Strategy We introduce a fail-safe strategy employed by our framework to ensure that consistent high-quality branching decisions are made throughout a BaB process. The proposed framework uses a GNN to imitate the behavior of the strong branching heuristic. Although computationally cheap, in some cases, the output decision by the learned graph neural network might be suboptimal. When this happens, it could lead to considerably deteriorated performances for two reasons. Firstly, we observed that for a given domain, which requires multiple splits to reach a conclusion on this domain, if a few low-quality branching decisions are made at the beginning or the middle stage of the branching process, the total number of splits required might increase substantially. The total BaB path is thus, to some extent, sensitive to the quality of each branching decision apart from those made near the end of the BaB process. Secondly, once a low-quality decision is made on a given domain, a decision of similar quality is likely to be made on the two newly generated sub-domains, leading to exponential decrease in performance. Features for newly generated sub-domains are normally similar to those of the parent domain, especially in the cases where the branching decision of the parent domain is made on the later layers and loose intermediate bounds are used. Thus, it is reasonable to expect the GNN fails again on the sub-domains.

To deal with issue, we keep track of the output lower bound improvement for each branching decision, as introduced in Eq. (8). We then set a pre-determined threshold parameter. If the improvement is below the threshold, a computationally cheap heuristic is called to make a branching decision. Generally, the back-up heuristic is able to give an above-threshold improvement and generate subdomains sufficiently different from the parent domain to allow the learned GNN to recover from the next step onwards.

Online Learning Online learning is a strategy to fine-tune the network for a particular property after we have learnt $\mathbf{\Theta}$. It can be seen as an extension of the fail-safe strategy employed. Every time a heuristic branching decision node v_h is used instead of the node v_{gnn} chosen by the GNN, we can use v_h and v_{gnn} to update GNN accordingly. Since a correct GNN model should output embedding vector $\boldsymbol{\mu}_h$ resulting in a higher score $g_s(\boldsymbol{\mu}_h; \mathbf{\Theta})$ for the heuristic decision, a loss can be developed based on the two scores $g_s(\boldsymbol{\mu}_h; \mathbf{\Theta})$ and $g_s(\boldsymbol{\mu}_{gnn}; \mathbf{\Theta})$ to generate optimization signals for correcting the GNN behaviour. For example, the loss used in our experimental setting is:

$$loss_{online}(\mathbf{\Theta}) = g_s(\boldsymbol{\mu}_{gnn}; \mathbf{\Theta}) - g_s(\boldsymbol{\mu}_h; \mathbf{\Theta}) + \gamma \cdot ((m_h - m_{gnn}) > t). \tag{11}$$

The last term is used to amplify ($\gamma > 0$) the loss if the relative improvement made by the heuristic decision is more than t percent higher than that by the GNN. We update $\mathbf{\Theta}$ of GNN by taking one gradient step with small learning rate of the following minimization problem.

$$\mathbf{\Theta} = \arg\min_{\mathbf{\Theta}} \frac{\lambda}{2}\|\mathbf{\Theta}\|^2 + loss_{online}(\mathbf{\Theta}). \tag{12}$$

Online learning is property specific: it uses the decisions made by heuristics to fine tune the GNN model so it can best accommodate the property at hand. As will be shown in our experiments, a small but significant improvement in performance is achieved when online learning is used.

6 Experiments

We now validate the effectiveness of our proposed framework through comparative experiments against other available NN verification methods. A comprehensive study of NN verification methods has been done in Bunel et al. (2019). We thus design our experiments based on the results presented in Bunel et al. (2019).

6.1 Setup

We are interested in verifying properties on large network architectures with convolutional layers. In Bunel et al. (2019), existing NN methods are compared on a robustly trained convolutional network on MNIST. We adopt a similar network structure but using a more challenging dataset, namely CIFAR-10, for an increased difficulty level. We compare against the following two methods: (i) MIPplanet, a mixed integer solver backed by the commercial solver Gurobi; and (ii) BaBSR, a BaB based method utilising a ReLU-split heuristic. Our choice is motivated by their superior performance over other methods for MNIST verification problems in the previous work (Bunel et al., 2019).

We provide the detailed experimental setup through four perspectives: bounding method, branching strategy, network structure, verification properties tested. (**Bounding methods**) We compute intermediate bounds using linear bounds relaxation (Figure 1(b)). For the output lower bound, we use Planet relaxation (Figure 1(c)) and solve the corresponding LP with Gurobi. For the output upper bound, we compute it by directly evaluating the network value at the input provided by the LP solution. (**Branching strategy**) We focus on ReLU split only in our experiments. As shown in Bunel et al. (2019), domain split only outperforms ReLU split on low input dimensional and small scale networks. Also, since one of the Baseline method BaBSR employs a ReLU-split heuristic, we consider ReLU split only for a fair comparison. However, we emphasize that our framework is readily applicable to work with a combined domain and ReLU split strategy. (**Network structures**) Three neural network structures will be studied. The base one is of the similar structure and size to the one used in Bunel et al. (2019). It has two convolutional layers, followed by two fully connected layers and is trained robustly using the method provided in Wong & Kolter (2018). This particular choice of network size is made because the time required for solving each LP increases substantially with the size of the network. To best evaluate the performance of the branching strategy, we have to work with a medium sized network so that within the given timeout, a sufficient amount of branching decisions can be made to allow effective comparisons. When testing the transferability of framework, two larger networks will be tested but their sizes are still restricted by the LP bottleneck. Detailed network architecture is provided in the appendices. (**Verification properties**) Finally, we consider the following verification properties. For a image $\boldsymbol{x}$, that the model correctly predicted the label y_c, we randomly choose a label $y_{c'}$ such that for a given ϵ, we want to prove

$$(\boldsymbol{e}^{(c)} - \boldsymbol{e}^{(c')})^T f'(\boldsymbol{x}') > 0 \qquad \forall \boldsymbol{x}' \text{ s.t } \quad \|\boldsymbol{x} - \boldsymbol{x}'\|_\infty \leq \epsilon, \tag{13}$$

where f' is the original neural network, $\boldsymbol{e}^{(c)}$ and $\boldsymbol{e}^{(c')}$ are one-hot encoding vectors for labels y_c and $y_{c'}$. We want to verify that for a given ϵ, the trained network will not make a mistake by labelling the image as $y_{c'}$. Since BaBSR is claimed to be the best performing method on convolutional networks, we use it to determine the ϵ values, which govern the difficulty level of verification properties. Small ϵ values mean that most ReLU activation units are fixed so their associated verification properties are easy to prove while large ϵ values could lead to easy detection of counter-examples. The most challenging ϵ values are those at which a large number of activation units are ambiguous. We use binary search with BaBSR method to find suitable ϵ values. We only consider ϵ values that result in true properties and timed out properties. Binary search process is simplified by our choice of robustly trained models. Since these models are trained to be robust over a δ ball, the predetermined value δ can be used as a starting value for binary search.

6.2 Training Dataset

In order to generate training data, we firstly pick 565 random images and for each image, we randomly select an incorrect class. For each property, the ϵ value is determined by running binary search with BaBSR and 800 seconds timeout, so the final set of properties consists of mainly easily solvable properties and a limited number of timed out properties.

We collect training data along a BaB process for solving a verification property. At each given domain, given the large number of potential branching decisions, we perform the strong branching heuristic on a selected subset of all potential branching decisions. The subset consists of branching decisions that are estimated to be of high quality by the BaBSR heuristic and randomly selected ones, which ensure a minimum 5% coverage on each layer.

To construct a training dataset that is representative enough of the whole problem space, we need to cover a large number of properties. In addition, within a BaB framework, it is important to include training data at different stages of a BaB process. However, running a complete BaB process with the strong branching heuristic for hundreds of properties is computationally expensive and considerably time consuming. We thus propose the following procedure for generating a training dataset to guarantee a wide coverage both in terms of the verification properties and BaB stages. For generated verification properties, we randomly select 25% of non-timeout property to conduct a complete BaB process with the strong branching heuristic. For the rest of the properties, we try to generate at least $\mathcal{B} = 20$ training data for each verification property. Given the maximum number of branches $q = 10$ and an effective and computationally cheap heuristic, we first generate a random integer k from $[0, q]$. Then, we run a BaB process with the selected cheap heuristic for k steps. Finally, we call the strong branching heuristic to generate a training sample. We repeat the process until B training samples are generated or the BaB process terminated. A detailed algorithm is provided in the appendices.

6.3 Base model

We test our learned model on the same model structure but on properties at three different difficulty levels. Testing verification properties are generated by binary search with BaBSR and 3600s timeout. We categorise verification properties solved within 800s as easy, which is consistent with training data generated, between 800s and 2400s as medium and more than 2400s as hard. In total, we generated 467 easy properties, 773 medium properties and 426 hard properties.

Results are given in the Table 1. Methods are compared in three perspectives: the average time over all properties, average number of branches required over the properties that are solved by all methods (we exclude timed out properties) and also the ratio of timed out properties. Since the properties are generated based on BaBSR, the timed out ratios of BaBSR on easy and medium properties are not comparable with that of other methods. All other numbers should give a fair evaluation of the effectiveness of our branching strategy. BaBSR, GNN and GNN-online only differ in the branching strategy used.

On all three sets of properties, we see that our learned branching strategy has led to a more than 50% reduction in the total average number of branches required for a property. As a direct result, the average time required achieves at least a 50% reduction as well. Our framework is thus an effective scheme and enjoys horizontal transferability. A further performance improvement is obtained through instance-specific online learning. Among all 1666 tested verification properties, GNN with online-learning solves 61.52% of properties with fewer number of branches and 60.20% of properties in less time when compared to standard GNN.

Table 1: Methods' Performance on the Base model. For easy, medium and difficult level verification properties, we compare methods' average solving time, average number of branches required and the percentage of timed out properties. GNN-Online outperforms other methods in all aspects.

	Easy			Medium			Hard		
Method	time(s)	branches	%Timeout	time(s)	branches	%Timeout	time(s)	branches	%Timeout
BaBSR	545.721	578.828	0.0	1370.395	1405.301	0.0	3128.00	2493.87	0.4131
MIPplanet	1499.375		0.165	2240.980		0.430	2250.624		0.462
GNN	272.695	285.682	0.002	592.118	583.210	0.004	1577.156	995.437	0.216
GNN-online	**208.822**	**252.807**	**0.002**	**556.163**	**501.595**	**0.001**	**1369.326**	**813.502**	**0.183**

Table 2: Methods' Performance on Large Models. For verification properties on Wide large model and Deep large model, we compare methods' average solving time, average number of branches required and the percentage of timed out properties. GNN-Online outperforms other methods in all aspects.

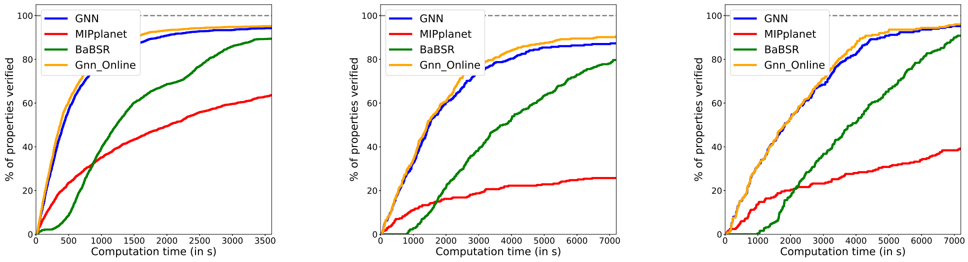
	Wide			Deep		
Method	time	branches	%Timeout	time	branches	%Timeout
BaBSR	4137.467	843.476	0.0	4016.336	416.824	0.0
MIPplanet	5855.059		0.743	5426.160		0.608
GNN	2367.693	418.684	0.127	2308.612	202.731	0.048
GNN-online	**2179.306**	**374.945**	**0.095**	**2220.351**	**193.697**	**0.040**

We also provide a time cactus plot (Figure 3a) for all properties on the Base model. Time cactus plots for each category of properties can be found in the appendices. All these time cactus plots look similar. Although BaBSR performs better than the commercial solver encoded method MIPplanet overall, MIPplanet wins on a subset of properties. The learned model GNN, however, is capable of giving consistent high quality performance over all properties tested.

6.4 Transferability: larger models

We also robustly trained two larger networks. One has the same layer structure as the Base model but more hidden units on each layer, which we refer to as Wide model. The other has a similar number of hidden units on each layer but more layers, which we refer to as Deep model. The detailed network architecture is provided in the appendices. Apart from the network structure, everything else is kept the same as for the Base model experiments. We use BaBSR and timeout of 7200s to generate 300 properties for the Wide model and 250 properties for the Deep model. For these two models, each LP called for solving a sub-problem output lower bound is much more time consuming, especially for the Deep model. This is reason that the average number of branches considered is much fewer that those of the Base model within the given time limit.

The model learned on the Base network is tested on verification properties of large networks. Experimental results are given in the Table 2 and time cactus plots (Figures 3b, 3c) are also provided. All results are similar to what we observed on the Base model, which show that our framework enjoys vertical transferability.



(a) Base model (b) Wide large model (c) Deep large model

Figure 3: Cactus plots for the Base model (left), Wide large model (middle) and Deep large model (right). For each model, we plot the percentage of properties solved in terms of time for each method. Consistent performances are observed on all three models. BaBSR beats MIPplanet on the majority of properties. GNN consistently outperforms BaBSR and MIPplanet. Further small improvements can be achieved through online-learning.

7 Discussion

The key observation of our work is that the neural network we wish to verify can be used to design a GNN to improve branching strategies. This observation can be used in enhancing the performances of other aspects of BaB. Possible future works include employing GNN to find fast-converging starting values for solving LPs on a neural network and utilising GNN to develop a lazy verifier, that only solves the corresponding LP on a domain when it could lead to pruning.

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.

Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

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

Rudy Bunel, Jingyue Lu, Ilker Turkaslan, Philip H.S Torr, Pushmeet Kohli, and M. Pawan Kumar. Branch and bound for piecewise linear neural network verification. *arXiv preprint arXiv:1909.06588*, 2019.

Hanjun Dai, Elias B. Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. *Conference on Neural Information Processing Systems*, 2017.

Ruediger Ehlers. Formal verification of piece-wise linear feed-forward neural networks. *Automated Technology for Verification and Analysis*, 2017.

Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks. *arXiv preprint arXiv:1906.01629*, 2019.

Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *The International Conference on Learning Representations*, 2015.

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 Dill, Kyle Julian, and Mykel Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks. *International Conference on Computer Aided Verification*, 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.

Vicenc Rubies Royo, Roberto Calandra, Dusan M Stipanovic, and Claire Tomlin. Fast neural network verification via shadow prices. *arXiv preprint arXiv:1902.07247*, 2019.

Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Efficient formal safety analysis of neural networks. *Conference on Neural Information Processing Systems*, 2018a.

Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Formal security analysis of neural networks using symbolic intervals. *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pp. 1599–1614, 2018b.

Tsui-Wei Weng, Huan Zhang, Hongge Chen, Zhao Song, Cho-Jui Hsieh, Duane Boning, Inderjit S Dhillon, and Luca Daniel. Towards fast computation of certified robustness for relu networks. *International Conference on Machine Learning*, 2018.

Eric Wong and Zico Kolter. Provable defenses against adversarial examples via the convex outer adversarial polytope. *International Conference on Machine Learning*, 2018.

APPENDIX A. BRANCH AND BOUND ALGORITHM

The following generic Branch and Bound Algorithm is provided in Bunel et al. (2019). Given a neural network *net* and a verification property *problem* we wish to verify, the BaB procedure examines the truthfulness of the property through an iterative procedure. During each step of BaB, we first use the *pick_out* function (line 6) to choose a problem *prob* to branch on. The split function (line 7) determines the branching strategy and splits the chosen problem *prob* into sub-problems. We compute output upper and lower bounds on each sub-problem with functions *compute_UB* and *compute_LB* respectively. Newly computed output upper bounds are used to tighten the global upper bound, which allows more sub-problems to be pruned. We prune a sub-problem if its output lower bound is greater than or equal to the global upper bound, so the smaller the global upper bound the better it is. Newly calculated output lower bounds are used to tighten the global lower bound, which is defined as the minimum of the output lower bounds of all remained sub-problems after pruning. We consider the BaB procedure converges when the difference between the global upper bound and the global lower bound is smaller than ϵ.

In our case, our interested verification problem Eq. (1) is a satisfiability problem. We thus can simplify the BaB procedure by initialising the global upper bound *global_ub* as 0. As a result, we prune all sub-problems whose output lower bounds are above 0. In addition, the BaB procedure is terminated early when a below 0 output upper bound of a sub-problem is obtained, which means a counterexample exits.

Algorithm 1 Branch and Bound

```
 1: function BAB($\texttt{net}, \texttt{problem}, \epsilon$)
 2:     $\texttt{global\_ub} \leftarrow \inf$
 3:     $\texttt{global\_lb} \leftarrow -\inf$
 4:     $\texttt{probs} \leftarrow [(\texttt{global\_lb}, \texttt{prob})]$
 5:     while $\texttt{global\_ub} - \texttt{global\_lb} > \epsilon$ do
 6:         $(\_\,, \texttt{prob}) \leftarrow \texttt{pick\_out}(\texttt{probs})$
 7:         $[\texttt{subprob\_1}, \ldots, \texttt{subprob\_s}] \leftarrow \texttt{split}(\texttt{prob})$
 8:         for $i = 1 \ldots s$ do
 9:             $\texttt{sub\_ub} \leftarrow \texttt{compute\_UB}(\texttt{net}, \texttt{subprob\_i})$
10:             $\texttt{sub\_lb} \leftarrow \texttt{compute\_LB}(\texttt{net}, \texttt{subprob\_i})$
11:             if $\texttt{sub\_ub} < \texttt{global\_ub}$ then
12:                 $\texttt{global\_ub} \leftarrow \texttt{sub\_ub}$
13:                 $\texttt{prune\_probs}(\texttt{probs}, \texttt{global\_ub})$
14:             end if
15:             if $\texttt{sub\_lb} < \texttt{global\_ub}$ then
16:                 $\texttt{probs.append}((\texttt{sub\_lb}, \texttt{subprob\_i}))$
17:             end if
18:         end for
19:         $\texttt{global\_lb} \leftarrow \min\{\texttt{lb} \mid (\texttt{lb}, \texttt{prob}) \in \texttt{probs}\}$
20:     end while
21:     return $\texttt{global\_ub}$
22: end function
```

APPENDIX B. IMPLEMENTATION OF FORWARD AND BACKWARD PASSES

We give implementation details of forward and backward updates for embedding vectors for the model used in the experiments section. Choices of forward and backward update functions are based on the bounding methods used. In our experiments, we used linear bound relaxations for computing intermediate bounds and Planet relaxation for computing the final output lower bound. We start with a graph neural network mimicking the structure of the network we want to verify. We denote domain lower and upper bounds as $\boldsymbol{l}_0$ and $\boldsymbol{u}_0$ respectively. Similarly, we denote the intermediate bounds (pre-activation) for layers $i = 1, \ldots, L-1$ as $\boldsymbol{l}_i$ and $\boldsymbol{u}_i$. Since an LP solver is called for the final output lower bound, we have primal values for all nodes of V and dual values for all ambiguous nodes of V. Finally, let $W^1, \ldots, W^L$ be the layer weights and $\boldsymbol{b}^1, \ldots, \boldsymbol{b}^L$ be the layer biases of the network f, which we wish to verify.

B.1 Forward Pass

Unless otherwise stated, all functions F_* are 2-layer fully connected network with ReLU activation units.

B.1.1 Input nodes

We update the embedding vectors of input nodes only during the first round of forward pass. That is we update $\boldsymbol{\mu}_{0[j]}$ when it is zero for all j. After that, input nodes embedding vectors are updated only in backward pass. For each input node, we form the feature vector $\boldsymbol{z}_{0[j]}$ as a vector of $l_{0[j]}$, $u_{0[j]}$ and its associated primal solution. The input node embedding vectors are computed as

$$\boldsymbol{\mu}_{0[j]} = F_{inp}(\boldsymbol{z}_{0[j]}; \boldsymbol{\theta}_0). \tag{14}$$

B.1.2 Activation nodes

The update function F_{act} can be broken down into three parts: 1) compute information from local features 2) compute information from neighbour embedding vectors and 3) combine information from 1) and 2) to update current layer's embedding vectors.

Information from local features Since we compute the final lower bound with the Planet relaxation (Figure 1(c)), we introduce a new feature related to the relaxation: the intercept of the relaxation triangle, shown in Figure 4. We denote an intercept as β and compute it as

$$\beta_{i[j]} = \frac{-l_{i[j]} \cdot u_{i[j]}}{u_{i[j]} - l_{i[j]}}. \tag{15}$$

The intercept of a relaxation triangle can be used as a measure of the amount of relaxation introduced at the current ambiguous node.

Therefore, the local feature vector $\boldsymbol{z}_{i[j]}$ of an ambiguous node $x'_{i[j]}$ consists of $l_{i[j]}$, $u_{i[j]}$, $\beta_{i[j]}$, its associated layer bias value, primal values (one for pre-activation variable and one for post-activation variable) and dual values. We obtain information from local features via

$$R_{i[j]} = \begin{cases} F_{act-lf}(\boldsymbol{z}_{i[j]}; \boldsymbol{\theta}_1^0) & \text{if } x'_{i[j]} \text{ is ambiguous,} \\ \mathbf{0} & \text{otherwise.} \end{cases} \tag{16}$$

where $R_{i[j]} \in \mathbb{R}^p$.

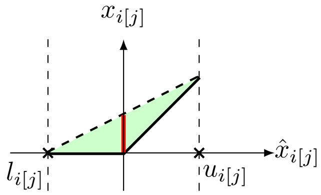
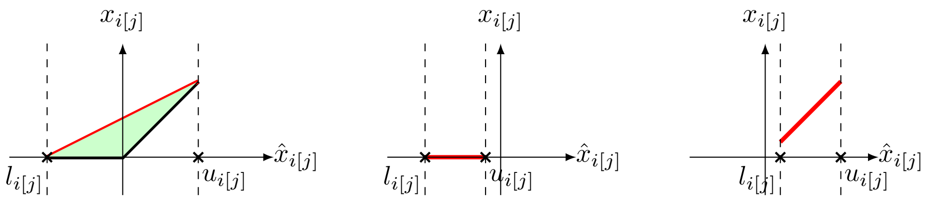


Figure 4: Red line represents the intercept of the convex relaxation. It is treated as a measure of the shaded green area.

Information from neighbour embedding vectors During the forward pass, we focus on embedding vectors of the previous layer only. To update an embedding vector on layer i, we first combine embedding vectors of the previous layer with edge weights via

$$E_{i[j]} = \sum_k W^i_{kj} \cdot \boldsymbol{\mu}_{i-1[k]}. \tag{17}$$

To compute the information from neighbour embedding vectors to an arbitrary activation node $x'_{i[j]}$, we consider each activation unit as a *gate*. We observe that the amount of the information from neighbour embedding vectors that remains after passing through a *gate* is dependent on the its lower bound $l_{i[j]}$ and upper bound $u_{i[j]}$. When $l_{i[j]}$ and $u_{i[j]}$ are of different signs, $x'_{i[j]}$ is an ambiguous



(a) Ambiguous Node (b) Complete Blocking (c) Complete Passing

Figure 5: Depending on the value of $l_{i[j]}$ and $u_{i[j]}$, relaxed activation function can take three forms. The left figure shows the case where $l_{i[j]}$ and $u_{i[j]}$ are of different signs. In this case, for any input value between $l_{i[j]}$ and $u_{i[j]}$, the maximum output achievable is indicated by the red line. The middle figure shows the case where both $l_{i[j]}$ and $u_{i[j]}$ are no greater than zero. In this case, the activation function completely blocks all input information by outputting zero for any input value. The right figure shows the case where $l_{i[j]}$ and $u_{i[j]}$ are greater or equal to zero. In this case, the activation function allows complete information passing by outputting a value equal to the input value.

node. With relaxation, for any input value between $l_{i[j]}$ and $u_{i[j]}$, the maximum output achievable after passing an activation unit is shown by the red slope in Figure 5(a). The red slope $s_{i[j]}$ is computed as

$$s_{i[j]}(\hat{x}_{i[j]}) = \frac{u_{i[j]}}{u_{i[j]} - l_{i[j]}} \cdot \hat{x}_{i[j]} + \beta_{i[j]}. \tag{18}$$

Thus, the amount of information from neighbour embedding vectors that remains after passing through an ambiguous *gate* is related to the ratio $\alpha := \frac{u_{i[j]}}{u_{i[j]} - l_{i[j]}}$. When $u_{i[j]}$ is no greater than zero, the activation node $x'_{i[j]}$ completely blocks all information. For any input value, the output value is zero after passing the activation unit, as shown by the red line in Figure 5(b). We have $\alpha = 0$ in this case. Finally, when $l_{i[j]}$ is no less than 0, the activation node $x'_{i[j]}$ allows a complete passing of information and $\alpha = 1$. It is shown by the red line in Figure 5(c). We incorporate these observations into our evaluations and compute the information from neighbour embedding vectors as

$$N_{i[j]} = f_{act-nb}([\alpha \cdot E_{i[j]}, \alpha' \cdot E_{i[j]}]; \boldsymbol{\theta}_1^1), \tag{19}$$

where $\alpha' = 1 - \alpha$ when $0 < \alpha < 1$ and $\alpha' = \alpha$ otherwise. Here, we use $[\boldsymbol{a}, \boldsymbol{b}]$ to denote the concatenation of two vectors $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^p$ into a vector $\mathbb{R}^{2p}$. We introduce α' to be more informative. We do not consider the information that relate to the intercept $\beta_{i[j]}$ in the ambiguous case for the sake of simplicity. Improved performance could be expected if the $\beta_{i[j]}$ related information is incorporated as well.

Combine previous information Finally, we combine the information from local features and the information from neighbour embedding vectors to update the embedding vectors of activation nodes. Specifically,

$$\boldsymbol{\mu}_{i[j]} = F_{act-com}([R_{i[j]}, N_{i[j]}]; \boldsymbol{\theta}_1^2). \tag{20}$$

B.1.3 Output node

Embedding vectors of output nodes are updated in a similar fashion to that of activation nodes. We first compute information from local features.

$$R_{Lj} = F_{out-lf}(\boldsymbol{z}_{Lj}; \boldsymbol{\theta}_2^0) \tag{21}$$

For output nodes, the vector of local features $\boldsymbol{z}_L$ consists of output lower bound, output upper bound, primal solution and layer bias. F_{out-lf} is a one-layer fully-connected network with ReLU activation units. We then compute information from neighbour embedding vectors. Since the output node does not have an activation unit associated with it, we directly compute the information of neighbour embedding vectors as

$$E_{L[j]} = \sum_k W_{kj}^L \cdot \boldsymbol{\mu}_{L-1[k]}. \tag{22}$$

Finally, we update the embedding vector of the output node as

$$\boldsymbol{\mu}_{Lj} = F_{out-com}([R_{L[j]}, E_{L[j]}]; \boldsymbol{\theta}_2^1). \tag{23}$$

B.2 Backward Pass

During backward message passing, for $i = L-1, \dots, 1$, we update embedding vectors for activation nodes and input node. Again, all functions B_* are 2-layer fully-connected networks unless specified otherwise.

B.2.1 Activation nodes

Similar to updates of embedding vectors carried out for activation nodes in a forward pass, we update embedding vectors of activation nodes using the same three steps in the backward pass, but with minor modifications.

Information from local features We use the same feature $\boldsymbol{z}_{i[j]}$ as the one used in the forward pass and compute the information from local features as

$$R^b_{i[j]} = \begin{cases} B_{act-lf_1}(\boldsymbol{z}_{i[j]}; \boldsymbol{\theta}^0_3) & \text{if } x'_{i[j]} \text{ is ambiguous,} \\ \mathbf{0} & \text{otherwise.} \end{cases} \tag{24}$$

We recall that a dual value indicates how the final objective function is affected if its associated constraint is relaxed by a unit. To better measure the importance of each relaxation to the final objective function, we further update the information from local features by

$$R^{b'}_{i[j]} = \begin{cases} B_{act-lf_2}([\boldsymbol{d}_{i[j]} \odot R^b_{i[j]}, R^b_{i[j]}]; \boldsymbol{\theta}^1_3) & \text{if } R^b_{i[j]} \neq \mathbf{0} \\ \mathbf{0} & \text{otherwise.} \end{cases} \tag{25}$$

Here, $\boldsymbol{d}_{i[j]}$ is the vector of dual values corresponding to the activation node $x'_{i[j]}$. We use $\odot$ to mean that we multiply $R^b_{i[j]}$ by each element value of $\boldsymbol{d}_{i[j]}$ and concatenate them as a singe vector.

Information from neighbour embedding vectors During the backward pass, we focus on embedding vectors of the next layer only. In order to update an embedding vector on layer i, we compute the neighbourhood embedding vectors as

$$E^b_{i[j]} = \sum_k W^{i+1}_{jk} \cdot \boldsymbol{\mu}_{i+1[k]}. \tag{26}$$

We point out that there might be an issue with computing $E_{i[j]}$ if the layer $i+1$ is a convolutional layer in the backward pass. For a convolutional layer, depending on the padding number, stride number and dilation number, each node $x'_{i[j]}$ may connect to a different number of nodes on the layer $i+1$. Thus, to obtain a consistent measure of $E_{i[j]}$, we divide $E_{i[j]}$ by the number of connecting node on the layer $i+1$, denoted as $E^{b'}_{i[j]}$ and use the averaged $E^{b'}_{i[j]}$ instead. Let

$$E^{b*}_{i[j]} = \begin{cases} E^{b'}_{i[j]} & \text{if layer i+1 convolutional,} \\ E^b_{i[j]} & \text{otherwise.} \end{cases} \tag{27}$$

The following steps are the same as the forward pass. We first evaluate

$$N^b_{i[j]} = B_{act-nb}([\alpha \cdot E^{b*}_{i[j]}, \alpha' \cdot E^{b*}_{i[j]}]; \boldsymbol{\theta}^2_3), \tag{28}$$

and the update embedding vectors as

$$\boldsymbol{\mu}_{i[j]} = B_{act-com}([R^{b'}_{i[j]}, N^b_{i[j]}]; \boldsymbol{\theta}^3_3). \tag{29}$$

B.2.2 Input nodes

Finally, we update the input nodes. We use the feature vector $\boldsymbol{z}^b_0$, which consists of domain upper bound and domain lower bound. Information from local features is evaluated as

$$R_{0j} = B_{inp-lf}(\boldsymbol{z}^b_{0[j]}; \boldsymbol{\theta}^0_4). \tag{30}$$

We compute the information from neighbour embedding vectors in the same manner as we do for activation nodes in the backward pass, shown in Eq (27). Denote the computed information as $E^{b*}_{0[j]}$. The embedding vectors of input nodes are updated by

$$\boldsymbol{\mu}_{0[j]} = B_{inp-com}([R^{b'}_{0[j]}, E^{b*}_{0[j]}]; \boldsymbol{\theta}^1_4). \tag{31}$$

Appendix C. Algorithm for generating Training Dataset

Algorithm 2 outlines the procedure for generating the training dataset. The algorithm ensures the generated training date have a wide coverage both in terms of the verification properties and BaB stages while at the same time is computationally efficient. Specifically, we randomly pick 25% of all properties that do not time out and run a complete BaB procedure on each of them with the strong branching heuristic to generate training samples (line 3-5). For the remaining properties, we attempt to generate B training samples for each of them. To cover different stages of a BaB process of a property, we use a computationally cheap heuristic together with the strong branching heuristic. Given a property, we first use the cheap heuristic for k steps (line 10-15) to reach a new stage of the BaB procedure and then call the strong branching heuristic to generate a training sample (line 16). We repeat the process until B training samples are generated or the BaB processs terminates.

Algorithm 2 Generating Training Dataset

```
 1: Provided: total $P$ properties; minimum $B$ training data for each property; a maximum $q$
    branches between strong branching decisions
 2: for $p = 1, \dots, P$ do:
 3:     $\alpha \longleftarrow$ random number from $[0, 1]$
 4:     if $p$ is not a timed out property and $\alpha \leq 0.25$ then
 5:         Running a complete BaB process with the Strong Branching Heuristic
 6:     else
 7:         $b = 0$
 8:         while $b \leq B$ do
 9:             $k \longleftarrow$ random integer from$[0, q]$
10:             while $k > 0$ do
11:                 Call a computationally cheap heuristic
12:                 if BaB process terminates then return
13:                 end if
14:                 $k = k - 1$
15:             end while
16:             Call the strong branching heuristic and generate a training sample
17:             if BaB process terminates then return
18:             end if
19:             $b = b + 1$
20:         end while
21:     end if
22: end for
```

Appendix D. Experiments Details

All the hyper-parameters used in the experiments are determined by testing a small set of numbers over the validation set. Due to the limited number of tests, we believe better sets of hyper-parameters could be found.

D.1 Training Details

Training dataset Recall that to generate a training dataset, 565 random images are selected. Binary serach with BaBSR and 800 seconds timeout are used to determine ϵ. Among 565 verification properties determined, we use 430 properties to generate 17958 training samples and the rest of properties to generate 5923 validation samples. Training samples and validation samples are generated using Algorithm 2 with $B = 20$ and $q = 10$.

Training We initialise a GNN by assigning each node a 32-dimensional zero embedding vector. GNN updates embedding vectors through two rounds of forward and backward updates. To train the GNN, we use hinge rank loss (Eq (9)) with $M = 10$. Parameters $\mathbf{\Theta}$ are computed and updated through Adam optimizer with weight decay rate $\lambda = 1e^{-4}$ and learning rate $1e^{-4}$. If the validation loss does not decrease for 10 consecutive epoches, we decrease the learning rate by a factor of 5. If the validation loss does not decrease for 20 consecutive epoches, we terminate the learning procedure. The batch size is set to 2.

D.2 Experimental Details

We ran all experiments in parallel on 16 CPU cores, with one property being verified on one CPU core. We observed that although we specifically set the thread number to be one for MIPplanet (backed by the commercial solver Gurobi), the time required for solving a property depends on the total number of CPUs used. For a machine with 20 cpu cores, MIPplanet requires much less time on average for proving the same set of properties on fewer (say 4) CPU cores in parallel than on many (say 16) CPU cores in parallel (the rest of CPU cores remain idle). Since BaBSR, GNN and GNN-online all use Gurobi for the bounding problems, similar time variation, depending on the number of CPU cores used, are observed. We ran each method in the same setting and on 16 CPUs in parallel, so our reported results and time are comparable. However, we remind readers to take time variation into consideration when replicating our experiments or using our results for comparison.

Fail-safe strategy Since, to the best of our knowledge, the branching heurisitc of BaBSR is the best performing one on convolutional neural networks so far, we choose it for our fail-safe strategy. The threshold is set to be 0.2. Every time when the relative improvement m_{gnn} of a GNN branching decision v_{gnn} is less than 0.2, we call the heuristic to make a new branching decision v_h. We solve the corresponding LPs for the new branching decision and compute its relative improvement m_h. The node with higher relative improvement is chosen to be the final branching decision.

Online learning We take a conservative approach in terms of online learning. We refer to a GNN decision as a failed decision if the relative improvement offered by heuristic branching is better than the one offered by the GNN. We record all GNN failed decisions and only update the GNN model online when the same failed decision is made at least twice. To update the GNN model, we use Adam optimizer with weight decay rate $\lambda = 1e^{-4}$ and learning rate $1e^{-4}$. The GNN model is updated with one gradient step only with respect to the optimization problem Eq. (12), where $\gamma = 1$ and $t = 0.1$ in the loss function $loss_{online}$, defined in Eq. (11).

Methods In Bunel et al. (2019), methods including MIPplanet, BaBSR, planet (Ehlers, 2017), reluBaB and reluplex (Katz et al., 2017) are compared on a small convolutional MNIST network. Among them, BaBSR and MIPplanet significantly outperform other methods. We thus evaluate our methods against these two methods only in the experiments section. We mention that Bunel et al. (2019) excludes the method Neurify (Wang et al., 2018a), because, during their personal correspondence with the authors of Neurify, the authors of Neurify mentioned the version available online is not theoretically correct and the correct version does not perform well on convolutional networks. Details can be found in Bunel et al. (2019). We do not compare against Neurify for the same reason.

D.3 Model Architecture

We provide the architecture detail of the neural networks verified in the experiments in the following table.

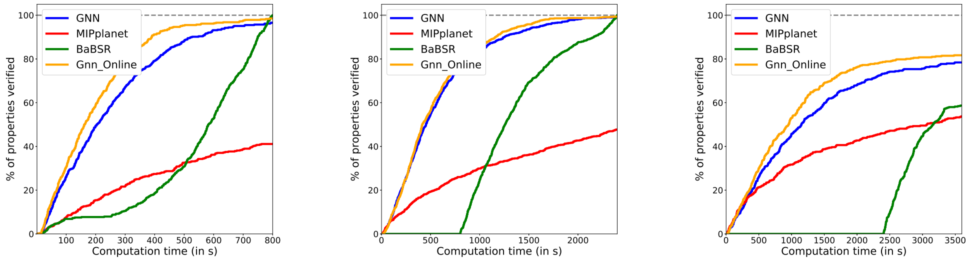
Network Name	No. of Properties	Network Architecture
Base Model	Easy: 467 Medium: 773 Hard: 426	Conv2d(3,8,4, stride=2, padding=1) Conv2d(8,16,4, stride=2, padding=1) linear layer of 100 hidden units linear layer of 10 hidden units (Total ReLU activation units: 3172)
WIDE	300	Conv2d(3,16,4, stride=2, 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 3: For each experiment, the network architecture used and the number of verification properties tested.

Appendix E. Additional Plots

Finally, we provide cactus plots for the Base model on easy, medium and hard difficulty level properties respectively.



(a) Easy properties

(b) Medium properties

(c) Hard properties

Figure 6: Cactus plots for easy properties (left), medium properties (middle) and hard properties (right) on the Base model. For each model, we plot the percentage of properties solved in terms of time for each method. BaBSR beats MIPplanet on easy and medium properties overall. On hard properties, although BaBSR manages to solve more properties, its average performance is worse than MIPplanet in terms of time. GNN consistently outperforms BaBSR and MIPplanet on all three levels of properties, demonstrating the horizontal transferrability of our framework. Again, further small improvements can be achieved through online-learning.