Journal of Machine Learning Research 21 (2020) 1-39 Submitted 6/19; Revised 2/20; Published 2/20

Branch and Bound for Piecewise Linear Neural Network Verification

Rudy Bunel * RUDY@ROBOTS.OX.AC.UK
Ilker Turkaslan ILKER.TURKASLAN@LMH.OX.AC.UK
Philip H.S. Torr PHILIP.TORR@ENG.OX.AC.UK
M. Pawan Kumar PAWAN@ROBOTS.OX.AC.UK
Department of Engineering Science
University of Oxford
Oxford OX1 3PJ

Jingyue Lu * JINGYUE.LU@SPC.OX.AC.UK
Department of Statistics
University of Oxford
Oxford OX1 3LB

Pushmeet Kohli PUSHMEET@GOOGLE.COM
Deepmind
London N1C 4AG

Editor: Amir Globerson

Abstract

The success of Deep Learning and its potential use in many safety-critical applications has motivated research on formal verification of Neural Network (NN) models. In this context, verification involves proving or disproving that an NN model satisfies certain input-output properties. Despite the reputation of learned NN models as black boxes, and the theoretical hardness of proving useful properties about them, researchers have been successful in verifying some classes of models by exploiting their piecewise linear structure and taking insights from formal methods such as Satisifiability Modulo Theory. However, these methods are still far from scaling to realistic neural networks. To facilitate progress on this crucial area, we exploit the Mixed Integer Linear Programming (MIP) formulation of verification to propose a family of algorithms based on Branch-and-Bound (BaB). We show that our family contains previous verification methods as special cases. With the help of the BaB framework, we make three key contributions. Firstly, we identify new methods that combine the strengths of multiple existing approaches, accomplishing significant performance improvements over previous state of the art. Secondly, we introduce an effective branching strategy on ReLU non-linearities. This branching strategy allows us to efficiently and successfully deal with high input dimensional problems with convolutional network architecture, on which previous methods fail frequently. Finally, we propose comprehensive test data sets and benchmarks which includes a collection of previously released testcases. We use the data sets to conduct a thorough experimental comparison of existing and new algorithms and to provide an inclusive analysis of the factors impacting the hardness of verification problems.

Keywords: Formal Verification, Branch and Bound, ReLU Branching

∗. equal contribution

1. Introduction

Despite their success in a wide variety of applications, Deep neural networks have seen limited adoption in safety-critical settings. The main explanation for this lies in their reputation for being black-boxes whose behaviour cannot be predicted. Current approaches to evaluate trained models mostly rely on testing using held-out data sets. However, as Edsger W. Dijkstra said “testing shows the presence, not the absence of bugs” (Buxton and Randell, 1970). If deep learning models are to be deployed in applications such as autonomous driving cars, we need to be able to verify safety-critical behaviours.

To this end, some researchers have tried to use formal methods. To the best of our knowledge, Zakrzewski (2001) was the first to propose a method to verify simple, one hidden layer neural networks. However, only recently were researchers able to work with non-trivial models by taking advantage of the structure of ReLU-based networks (Cheng et al., 2017b; Katz et al., 2017a). Even then, these works are not scalable to the large networks encountered in most real world problems.

This paper advances the field of NN verification by making the following key contributions:

1. By taking advantage of the Mixed Integer Linear Programming (MIP) formulation of the problem, we introduce the Branch-and-Bound framework for NN verification. The framework contains state of the art verification methods as special cases.

2. We identify the weakness and strengths of previous verification methods from the aspects of the way bounds are computed, the type of branching that are considered and the strategies guiding the branching. By retaining the strengths and correcting the identified flaws, we propose new methods that achieve considerable performance improvements when compared to the previous state of the art. In some cases, a speed-up of almost two orders of magnitudes is obtained. Specifically, we develop a new branching strategy that supports branching over ReLU non-linearities. Previous BaB based verification methods mainly focus on designing heuristics for branching over input domains. These heuristics, although they perform well on small-scale problems, are either computationally expensive for high dimensional input problems or ineffective for problems with convolutional network architecture. Similar issues are faced by the existing ReLU branching strategies. Our designed branching strategy is computationally cheap and explores the underlying network architecture to make a decision. Using on high dimensional input problems with convolutional network architectures, we demonstrate the benefits of our branching strategy over various verification methods that employ either input-domain branching or ReLU branching strategies.

3. We introduce comprehensive data sets consisting of trained as well as synthetic networks with fully connected and/or convolutional layers. Convolutional networks are widely used in computer vision tasks and should be an indispensable component for fair and complete evaluations of verification methods. Only recently did convolutional network data start to be included in the evaluation of verification methods. This takes the form of verification properties attempting to prove adversarial robustness on a $\mathcal{L}_\infty$ ball with a fixed perturbed distance ϵ. The difficulty level of a verification property is mainly determined by its network size. Our curated convolutional data sets differ from these data sets and are able to bring new insights by verifying properties on range of ϵ

values on the same network. We make two observations to strengthen our statement. Firstly, the difficulty of a verification property not only relies on the size of the network, but also the value of ϵ. Secondly, one bottleneck for BaB based methods is the time required for solving linear programs (LPs), which could increase significantly with the size of the network. Our data sets consist of verification properties with various difficulty levels on relatively small network architecture. This means they allow effective evaluations of branching heuristics or bounding decisions without suffering from the LP bottleneck. Additionally, we have introduced the synthetic TwinStream data set to facilitate the study of the relationship between bounding and branching strategies. Overall, the extensive test data sets not only allow thorough experimental analyses of existing methods, but also facilitate the understanding of verification problems and encourage the development of new methods.

A preliminary version of this work appeared in the proceedings of NeurIPS, 2018. The article significantly differs from the previous work by (i) improving the clarity of the BaB framework by providing a running toy example; (ii) designing novel branching strategies for the important class of NN with convolutional layers; (ii) introducing new data sets with convolutional networks and synthetic models; and (iv) including new baseline algorithms.

Section 2 and 3 specify the problem of verification and present different formulations of verification processes respectively. Section 4 presents the BaB framework, showing that previous methods can be seen as special cases of it. Section 5 builds on the observations in section 4 to highlight possible improvements within the BaB framework. New methods are proposed accordingly. The last two sections conduct detailed experimental studies of verification methods on our comprehensive data sets. Specifically, section 6 discusses the experimental setup and section 7 analyses the results.

2. Problem Specification

We now specify the problem of formal verification of neural networks. Given a network that implements a function $\mathbf{\hat{x}_n} = f(\mathbf{x_0})$, a bounded input domain $\mathcal{C}$ and a property P, we want to prove

$$\mathbf{x_0} \in \mathcal{C}, \quad \mathbf{\hat{x}_n} = f(\mathbf{x_0}) \implies P(\mathbf{\hat{x}_n}). \tag{1}$$

For example, the property of robustness to adversarial examples in $\mathcal{L}_\infty$ norm around a training sample $\mathbf{a}$ with label y_a would be encoded by using $\mathcal{C} \triangleq \{\mathbf{x_0} |\ \|\mathbf{x_0} - \mathbf{a}\|_\infty \leq \epsilon\}$ and $P(\mathbf{\hat{x}_n}) = \left\{\forall y, \quad \hat{x}_{n[y_a]} > \hat{x}_{n[y]}\right\}$. From now on, we use $x_{i[j]}$ to denote the jth element of $\mathbf{x_i}$.

In this paper, we are going to focus on Piecewise-Linear neural networks (PL-NN), that is, networks for which we can decompose $\mathcal{C}$ into a set of polyhedra $\mathcal{C}_i$ such that $\mathcal{C} = \cup_i\ \mathcal{C}_i$, and the restriction of f to $\mathcal{C}_i$ is a linear function for each i. While this prevents us from including networks that use activation functions such as sigmoid or tanh, PL-NNs allow the use of linear transformations such as fully-connected or convolutional layers, pooling units such as MaxPooling and activation functions such as ReLUs. In other words, PL-NNs represent the majority of networks used in practice. Operations such as Batch-Normalization or Dropout also preserve piecewise linearity at test-time.

The properties that we are going to consider are Boolean formulas over linear inequalities. In our robustness to adversarial example above, the property is a conjunction of linear inequalities, each of which constrains the output of the original label to be greater than the output of another label.

In general, we divide verification algorithms into three categories: algorithms are unsound if they can only prove some of the false properties are false; algorithms are incomplete if they can only prove some of the true properties are true; and algorithms are complete if they are able to report all correct properties. In this paper, we will only focus on complete algorithms. For unsound methods, we refer interested readers to Akintunde et al. (2018); Huang et al. (2017), Carlini and Wagner (2017) and Webb et al. (2019) and for incomplete methods, we refer interested readers to Xiang et al. (2017); Weng et al. (2018); Singh et al. (2018) and Dvijotham et al. (2018). In addition, among complete methods, the scope of this paper does not include approaches relying on additional assumptions such as twice differentiability of the network (see Hein and Andriushchenko, 2017; Zakrzewski, 2001), limitation of the activation to binary values (see Cheng et al., 2017b; Narodytska et al., 2017) or restriction to a single linear domain (see Bastani et al., 2016).

3. Verification Formalism

In this section, we present different formulations of verification process.

3.1. Verification as a Satisfiability Problem

The methods we involve in our comparison all leverage the piecewise-linear structure of PL-NN to make the problem more tractable. They all follow the same general principle: given a property to prove, they attempt to discover a counterexample that would make the property false. This is accomplished by defining a set of variables corresponding to the inputs, hidden units and output of the network, and the set of constraints that a counterexample would satisfy.

To help design a unified framework, we reduce all instances of verification problems to a canonical representation. Specifically, the whole satisfiability problem will be transformed into a global optimization problem where the decision will be obtained by checking the sign of the minimum. If the property is a simple inequality $P(\hat{\mathbf{x}}_{\mathbf{n}}) \triangleq \mathbf{c}^T\hat{\mathbf{x}}_{\mathbf{n}} > b$, it is sufficient to add to the network a final fully connected layer with one output, with weight of $\mathbf{c}$ and a bias of $-b$. If the global minimum of this network is positive, it indicates that for all $\hat{\mathbf{x}}_{\mathbf{n}}$, the original network output, we have $\mathbf{c}^T\hat{\mathbf{x}}_{\mathbf{n}} - b > 0 \implies \mathbf{c}^T\hat{\mathbf{x}}_{\mathbf{n}} > b$, and as a consequence the property is true. On the other hand, if the global minimum is negative, then the minimizer provides a counter-example. Clauses `OR` and `AND` in the property can similarly be expressed as additional layers, using MaxPooling units. Specifically, clauses specified using `OR` (denoted by $\bigvee$) can be encoded by using a MaxPooling unit. If the property is $P(\hat{\mathbf{x}}_{\mathbf{n}}) \triangleq \bigvee_i \left[\mathbf{c}_i^T\hat{\mathbf{x}}_{\mathbf{n}} > b_i\right]$, this is equivalent to $\max_i \left(\mathbf{c}_i^T\hat{\mathbf{x}}_{\mathbf{n}} - b_i\right) > 0$. Clauses specified using `AND` (denoted by $\bigwedge$) can be encoded similarly: the property $P(\hat{\mathbf{x}}_{\mathbf{n}}) = \bigwedge_i \left[\mathbf{c}_i^T\hat{\mathbf{x}}_{\mathbf{n}} > b_i\right]$ is equivalent to $\min_i \left(\mathbf{c}_i^T\hat{\mathbf{x}}_{\mathbf{n}} - b_i\right) > 0 \iff -\left(\max_i \left(-\mathbf{c}_i^T\hat{\mathbf{x}}_{\mathbf{n}} + b_i\right)\right) > 0$. We can formulate any Boolean formula over linear inequalities on the output of the network as a sequence of additional linear and max-pooling layers. From now on, we assume that a property is in a canonical form. Specifically, the output of the network is a scalar, and the property is true if the output is positive for all inputs in a given domain, and false otherwise. Assuming the network only contains ReLU activations between each layer, the satisfiability problem to find a counterexample can be expressed as:

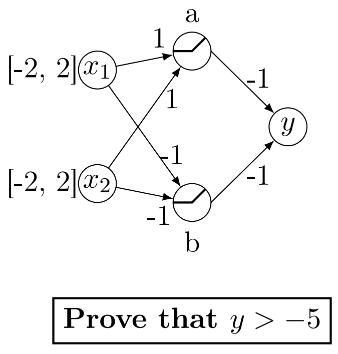


Figure 1: Example Neural Network. We attempt to prove the property that the network output is always greater than -5.

$$\mathbf{l_0} \leq \mathbf{x_0} \leq \mathbf{u_0} \quad (2a) \qquad \mathbf{\hat{x}_{i+1}} = W_{i+1}\mathbf{x_i} + \mathbf{b_{i+1}} \quad \forall i \in \{0,\ n-1\} \quad (2c)$$

$$\hat{x}_n \leq 0 \quad (2b) \qquad \mathbf{x_i} = \max\left(\mathbf{\hat{x}_i}, 0\right) \quad \forall i \in \{1,\ n-1\}. \quad (2d)$$

Equation 2a represents the constraints on the input and Equation 2b on the neural network output. Equation 2c encodes the linear layers of the network and Equation 2d the ReLU activation functions. If an assignment to all the values can be found, this represents a counterexample. If this problem is unsatisfiable, no counterexample can exist, implying that the property is true. We emphasise that we are required to prove that no counter-examples can exist, and not simply that none could be found.

While for clarity of explanation, we have limited ourselves to the specific case where only ReLU activation functions are used, this is not restrictive. The appendix contains a section detailing how each method specifically handles MaxPooling units, as well as how to convert any MaxPooling operation into a combination of linear layers and ReLU activation functions. Converting a verification problem into this canonical representation does not make its resolution simpler since the addition of the ReLU non-linearities Equation 2d transforms a problem that would have been solvable by simple Linear Programming into an NP-hard problem (Katz et al., 2017a). However, it does provide a formalism advantage. Specifically, it allows us to prove complex properties, containing several `OR` clauses, with a single procedure rather than having to decompose the desired property into separate queries as was done in previous work (Katz et al., 2017a). Operationally, a valid strategy for dealing with verification problems in the canonical form is to impose the constraints Equations 2a-2d and minimise the value of $\hat{x}_n$. Finding the exact global minimum is not necessary for verification. However, it provides a measure of satisfiability or unsatisfiability. If the value of the global minimum is positive, it will correspond to the margin by which the property is satisfied.

Toy Example *A toy-example of the Neural Network verification problem is given in Figure 1. On the domain* $\mathcal{C} = [-2; 2] \times [-2; 2]$*, we want to prove that the output* y *of the one hidden-layer network always satisfies the property* $P(y) \triangleq [y > -5]$*. We will use this as a running example to illustrate different formulations of the problem and introduce methods that can be reframed in our unified framework.*

For the network of Figure 1, the problem is formulated as follows. The variables would be $\{x_1, x_2, a_{in}, a_{out}, b_{in}, b_{out}, y\}$ *and the set of constraints would be:*

$$\begin{aligned}
&-2 \le x_1 \le 2 && -2 \le x_2 \le 2 && \text{(3a)}\\
&\hat{a} = x_1 + x_2 && \hat{b} = -x_1 - x_2 && \text{(3b)}\\
&a = \max(\hat{a}, 0) && b = \max(\hat{b}, 0) && \text{(3c)}\\
&y = -a - b && && \text{(3d)}\\
&y \le -5. && && \text{(3e)}
\end{aligned}$$

Here, $\hat{a}, \hat{b}$ *is the input value to hidden unit, while* a, b *is the value after the ReLU. Any point satisfying all the above constraints would be a counterexample to the property, as it would imply that it is possible to drive the output to -5 or less.*

3.2. Mixed Integer Linear Programming Formulation

A possible way to eliminate the non-linearities is to encode them with the help of binary variables, transforming the PL-NN verification problem Equation 2 into a Mixed Integer Linear Programming (MIP) problem. This can be done with the use of "big-M" encoding. The following encoding is from Tjeng and Tedrake (2019). Assuming we have access to lower and upper bounds on the values that can be taken by the coordinates of $\hat{\mathbf{x}}_\mathbf{i}$, which we denote $\mathbf{l_i}$ and $\mathbf{u_i}$, we can replace the non-linearities:

$$\begin{aligned}
\mathbf{x_i} = \max(\hat{\mathbf{x}}_\mathbf{i}, 0) \quad \Rightarrow \quad \boldsymbol{\delta_i} \in \{0,1\}^{h_i}, \quad &\mathbf{x_i} \ge 0, \qquad \mathbf{x_i} \le \mathbf{u_i} \cdot \boldsymbol{\delta_i} && \text{(4a)}\\
&\mathbf{x_i} \ge \hat{\mathbf{x}}_\mathbf{i}, \qquad \mathbf{x_i} \le \hat{\mathbf{x}}_\mathbf{i} - \mathbf{l_i} \cdot (1 - \boldsymbol{\delta_i}). && \text{(4b)}
\end{aligned}$$

It is easy to verify that $\delta_{i[j]} = 0 \Leftrightarrow x_{i[j]} = 0$ (replacing $\delta_{i[j]}$ in Equation 4a) and $\delta_{i[j]} = 1 \Leftrightarrow x_{i[j]} = \hat{x}_{i[j]}$ (replacing $\delta_{i[j]}$ in Equation 4b). From now on, we refer to $\mathbf{l_0}$ and $\mathbf{u_0}$, the lower and upper bounds for the input domain, as input bounds and $\mathbf{l_i}$ and $\mathbf{u_i}$ for $i \in \{1,\ n-1\}$, the lower and upper bounds for hidden units $\hat{\mathbf{x}}_\mathbf{i}$, as intermediate bounds.

Toy Example (MIP formulation) *In our example, the non-linearities of Equation 3c would be replaced by following conditions. Here, we give the detailed description for* $\hat{a}$*. The same is applied to* $\hat{b}$*.*

$$\begin{aligned}
&a \ge 0 && a \ge \hat{a} \\
&a \le \hat{a} - l_a(1 - \delta_a) && a \le u_a \delta_a && \text{(5)}\\
&\delta_a \in \{0, 1\}. &&
\end{aligned}$$

where l_a *is a lower bound of the value that* $\hat{a}$ *can take (such as -4) and* u_a *is an upper bound (such as 4). The binary variable* δ_a *indicates which phase the ReLU is in: if* $\delta_a = 0$*, the ReLU is blocked and* $a = 0$*, else the ReLU is passing and* $a = \hat{a}$*. The problem remains difficult due to the integer constraint on* δ_a*.*

By taking advantage of the feed-forward structure of the neural network, lower and upper bounds $\mathbf{l_i}$ and $\mathbf{u_i}$ can be obtained by applying interval arithmetic (Hickey et al., 2001) to propagate the bounds on the inputs, one layer at a time.

Thanks to this specific feed-forward structure of the problem, the generic, non-linear, non-convex problem has been rewritten into a MIP. Optimization of MIP is well studied and highly efficient off-the-shelf solvers exist. As solving them is NP-hard, performance is going to be dependent on the quality of both the solver used and the encoding. We now ask the following question: how much efficiency can be gained by using a bespoke solver rather than a generic one? In order to answer this, we present specialised solvers for the PL-NN verification task.

4. Branch and Bound for Verification

As described in Section 3.1, the verification problem can be rephrased as a global optimization problem. For non-convex problems, algorithms such as first order methods like Gradient Descent are not appropriate as they have no way of guaranteeing whether or not a stationary point is a global minimum. In this section, we present an approach to estimate the global minimum, based on the Branch-and-Bound paradigm (Land and Doig, 1960; Morrison et al., 2016). We also show that several published methods fit this framework. Among these methods, detailed studies are conducted on the previous state of art methods Reluplex and Planet, which are introduced as examples of Satisfiability Modulo Theories (SMT).

Algorithm 1 Branch and Bound

```
 1: function BaB(net, problem, $\epsilon$)
 2:   global_ub $\leftarrow$ inf
 3:   global_lb $\leftarrow -$inf
 4:   probs $\leftarrow$ [(global_lb, problem)]
 5:   while global_ub $-$ global_lb $> \epsilon$ do
 6:     (_ , prob) $\leftarrow$ pick_out(probs)
 7:     [subprob_1, ..., subprob_s] $\leftarrow$ split(prob)
 8:     for $i = 1 \dots s$ do
 9:       prob_ub $\leftarrow$ compute_UB(net, subprob_i)
10:       prob_lb $\leftarrow$ compute_LB(net, subprob_i)
11:       if prob_ub $<$ global_ub then
12:         global_ub $\leftarrow$ prob_ub
13:         prune_problems(probs, global_ub)
14:       end if
15:       if prob_lb $<$ global_ub then
16:         problems.append((prob_lb, subprob_i))
17:       end if
18:     end for
19:     global_lb $\leftarrow \min\{$lb $\mid$ (lb, prob) $\in$ probs$\}$
20:   end while
21:   return global_ub
22: end function
```

Algorithm 1 describes its generic form. The original problem is repeatedly split into sub-problems (either split the input domain into sub-domains or an unfixed ReLU activation unit [1] into different phases) (line 7), over which lower and upper bounds of the minimum are computed (lines 9-10). The best upper-bound found so far serves as a candidate for the global minimum. Any sub-problem whose lower bound is greater than the current global upper bound can be pruned away as it cannot contain the global minimum (line 13, lines 15-17). By iteratively splitting the (sub-)problems, it is possible to compute tighter lower bounds. We keep track of the global lower bound on the minimum by taking the minimum over the lower bounds of all sub-problems (line 19). When the global upper bound and the global lower bound differ by less than a small scalar ϵ (line 5), we consider that we have converged.

Algorithm 1 shows how to optimise and obtain the global minimum. If all that we are interested in is the satisfiability problem, the procedure can be simplified by initialising the

1. We refer to a ReLU activation unit $x_i[j] = \max(\hat{x}_{i[j]}, 0)$ as unfixed if, given the upper and lower bounds $u_{i[j]}, l_{i[j]}$ of $x_{i[j]}$, $x_i[j]$ can take either the value of $\hat{x}_{i[j]}$ or 0.

global upper bound with 0 (in line 2). Any sub-problem with a lower bound greater than 0 (and therefore not eligible to contain a counterexample) will be pruned out (by line 15). The computation of the lower bound can therefore be replaced by the feasibility problem (or its relaxation) imposing the constraint that the output is below zero without changing the algorithm. If it is feasible, there might still be a counterexample and further branching is necessary. If it is infeasible, the sub-problem can be pruned out. In addition, if any upper bound improving on 0 is found on a sub-problem (line 11), it is possible to stop the algorithm as this already indicates the presence of a counterexample.

The description of the verification problem as optimization and the pseudo-code of Algorithm 1 are generic and would apply to verification problems beyond the specific case of PL-NN. To obtain a practical algorithm, it is necessary to specify several elements.

A search strategy, defined by the `pick_out` function, which chooses the next problem to branch on. Several heuristics are possible, for example those based on the results of previous bound computations. For satisfiable problems or optimization problems, this allows us to discover good upper bounds, enabling early pruning.

A branching rule, defined by the `split` function, which takes a problem `prob` and returns its partition into subproblems such that $\bigcup_i \texttt{subprob\_i} = \texttt{prob}$ and that $(\texttt{subprob\_i} \cap \texttt{subprob\_j}) = \emptyset, \ \forall i \neq j$. This will determine the attributes of the (sub-)problems, which impacts the hardness of computing bounds. In addition, choosing the right partition can greatly impact the quality of the resulting bounds.

Bounding methods, defined by the `compute_{UB,LB}` functions. These procedures estimate respectively upper bounds (`prob_ub`) and lower bounds (`prob_ub`) over the minimum output that the network `net` can reach over a given (sub-)problem. We want the lower bound to be as high as possible, so that this (sub-)problem can be pruned easily. This is usually done by introducing convex relaxations of the (sub-)problem and minimising them. On the other hand, the computed upper bound should be as small as possible, so as to allow pruning out other (sub-)problems or discovering counterexamples. As any feasible point corresponds to an upper bound on the minimum, heuristic methods are sufficient.

4.1. BaB Reformulations

We now give a general discussion of published work in the verification literature through the lens of the Branch-and-Bound framework. We first briefly mention some methods that do not rely on SMT solvers and then conduct detailed studies on SMT based methods Reluplex and Planet. ReluVal is a complete method introduced by Wang et al. (2018b). In ReluVal, an input domain branching rule is used and the split function decides which domain dimension to split on by computing influence metrics based on input-output gradient. For the bounding method, it uses a novel technique called symbolic interval propagation, which replaces lower and upper bounds by linear equations of input variables and propagates these linear equations in a layer by layer order. By doing so, symbolic intervals can preserve more dependency information than interval arithmetic, and are thus able to achieve tighter final bounds. Inspired by the branching rule used by ReluVal, Royo et al. (2019) proposed a verification procedure via shadow prices. This method also adopts a Branch-and-Bound framework. In detail, for the branching rule, the method makes an input split decision through sensitivity studies of the bounds of unfixed ReLU nodes to the change of the input domain. The bounding method is not specified but the same logic has been used to check whether a sub-domain should be stored and further split on or should be pruned. Also based on ReluVal is a complete method called Neurify (Wang et al., 2018a). Neurify is an improved version of ReluVal, which also uses gradient metrics to make a split decision but it splits on unfixed ReLU nodes. Neurify uses a different bounding method than ReluVal by calling an LP solver instead of computing symbolic interval bounds. For all the methods mentioned above, no specific search strategy is implemented. All sub-problems are simply enumerated.

4.2. Reluplex

Katz et al. (2017a) present a procedure named Reluplex to verify properties of Neural Network containing linear functions and ReLU activation units. Reluplex functions as an SMT solver using the splitting-on-demand framework (Barrett et al., 2006). The principle of Reluplex is to always maintain an assignment to all of the variables, even if some of the constraints are violated.

Starting from an initial assignment, it attempts to fix some violated constraints at each step. It prioritises fixing linear constraints (Equations 2a, 2b, 2c and some relaxation of Equation 2d) using a simplex algorithm, even if it leads to violated ReLU constraints. If no solution to this relaxed problem containing only linear constraints exists, the counterexample search is unsatisfiable. Otherwise, either all ReLU are respected, which generates a counterexample, or Reluplex attempts to fix one of the violated ReLU, potentially leading to newly violated linear constraints. This process is not guaranteed to converge. Thus, to make progress, non-linearities that get fixed too often are split into two cases. Two new problems are generated, each corresponding to one of the phases of the ReLU. In the worst setting, the problem will be split completely over all possible combinations of activation patterns, at which point the sub-problems will all be simple LPs.

This algorithm is a special case of Branch-and-Bound for satisfiability. The **search strategy** is handled by the SMT core and to the best of our knowledge does not prioritise any (sub-)problems. The **branching rule** is implemented by the ReLU-splitting procedure: when neither the upper bound search, nor the detection of infeasibility are successful, one non-linear constraint over the j-th neuron of the i-th layer $x_{i[j]} = \max\left(\hat{x}_{i[j]}, 0\right)$ is split out into two sub-problems: $\{x_{i[j]} = 0, \hat{x}_{i[j]} \leq 0\}$ and $\{x_{i[j]} = \hat{x}_{i[j]}, \hat{x}_{i[j]} \geq 0\}$. The prioritisation of ReLUs that have been frequently fixed is a heuristic to decide among possible partitions.

As Reluplex only deals with satisfiability, the analogue of the lower bound computation is an over-approximation of the satisfiability problem. The **bounding method** used is a convex relaxation, obtained by dropping some of the constraints. The following relaxation is applied to ReLU units for which the sign of the input is unknown ($l_{i[j]} \leq 0$ and $u_{i[j]} \geq 0$).

$$\mathbf{x_i} = \max\left(\mathbf{\hat{x}_i}, 0\right) \quad \Rightarrow \quad \mathbf{x_i} \geq \mathbf{\hat{x}_i} \qquad (6a) \qquad\qquad \mathbf{x_i} \geq 0 \qquad (6b) \qquad\qquad \mathbf{x_i} \leq \mathbf{u_i}. \qquad (6c)$$

If this relaxation is unsatisfiable, this indicates that the subdomain cannot contain any counterexample and can be pruned out. The search for an assignment satisfying all the ReLU constraints by iteratively attempting to correct the violated ReLUs is a heuristic that is equivalent to the search for an upper bound lower than 0: success implies the end of the procedure.

Step	$\mathbf{x_1}$	$\mathbf{x_2}$	$\hat{\mathbf{a}}$	**a**	$\hat{\mathbf{b}}$	**b**	**y**
1	0	0	0	0	0	0	0
	Fix linear constraints						
	0	0	0	1	0	4	-5
2	0	0	0	1	0	4	-5
	Fix a ReLU						
	0	0	0	1	4	4	-5
3	0	0	0	1	4	4	-5
	Fix linear constraints						
	-2	-2	-4	1	4	4	-5
	...						

Figure 2: Evolution of the Reluplex algorithm. Red cells corresponds to value violating linear constraints, and orange cells corresponds to value violating ReLU constraints. Resolution of violation of linear constraints are prioritised.

Toy Example (Running Reluplex) *Figure 2 shows the initial steps of a run of the Reluplex algorithm on the example of Figure 1. Starting from an initial assignment, it attempts to fix some violated constraints at each step. It prioritises fixing linear constraints (Equations 3a, 3b and 3e in our illustrative example) using a simplex algorithm, even if it leads to violated ReLU constraints Equation 3c. This can be seen in step 1 and 3 of the process.*

If no solution to the problem containing only linear constraints exists, this shows that the counterexample search is unsatisfiable. Otherwise, all linear constraints are fixed and Reluplex attempts to fix one violated ReLU at a time, such as in step 2 of Figure 2 (fixing the ReLU b), potentially leading to newly violated linear constraints. In the case where no violated ReLU exists, this means that a satisfiable assignment has been found and that the search can be terminated early.

4.3. Planet

Ehlers (2017a) also proposed an approach based on SMT. Unlike Reluplex, the proposed tool, named Planet, operates by explicitly attempting to find an assignment to the phase of the non-linearities. Reusing the notation of Section 3.2, it assigns a value of 0 or 1 to each $\delta_{i[j]}$ variable, verifying at each step the feasibility of the partial assignment so as to prune infeasible partial assignment early.

As in Reluplex, the **search strategy** is not explicitly encoded and simply iterates over the (sub-)problems that have not yet been pruned. The **branching rule** is the same as for Reluplex, as fixing the decision variable $\delta_{i[j]} = 0$ is equivalent to choosing $\{x_{i[j]} = 0, \hat{x}_{i[j]} \leq 0\}$ and fixing $\delta_{i[j]} = 1$ is equivalent to $\{x_{i[j]} = \hat{x}_{i[j]}, \hat{x}_{i[j]} \geq 0\}$. Note however that Planet does not include any heuristic to prioritise which decision variables should be split over. As a result, there is no mechanism that based on a heuristic search of a feasible point to encourage early termination in Planet.

For satisfiable problems, only when a full complete assignment is identified is a solution returned. In order to detect incoherent assignments, Ehlers (2017a) introduces a global linear approximation to a neural network, which is used as a **bounding method** to over-approximate the set of values that each hidden unit can take. In addition to the existing linear constraints (Equations 2a, 2band 2c), the non-linear constraints are approximated by sets of linear constraints representing the convex hull of each non-linearity treated independently. Specifically, ReLUs with input of unknown sign are replaced by the set of equations:

$$\mathbf{x_i} = \max\left(\hat{\mathbf{x}}_\mathbf{i}, 0\right) \;\Rightarrow\; \mathbf{x_i} \geq \hat{\mathbf{x}}_\mathbf{i} \quad \text{(7a)} \qquad \mathbf{x_i} \geq 0 \quad \text{(7b)} \qquad x_{i[j]} \leq u_{i[j]} \frac{\hat{x}_{i[j]} - l_{i[j]}}{u_{i[j]} - l_{i[j]}}. \quad \text{(7c)}$$

Recall that $x_{i[j]}$ corresponds to the value of the j-th coordinate of $\mathbf{x_i}$. An illustration of the convex hull is provided in the supplementary material.

Compared with the relaxation of Reluplex Equation 6, the Planet relaxation is tighter. Specifically, Equations 6a and 6b are identical to Equations 7a and 7b but Equation 7c implies Equation 6c. Indeed, given that $\hat{x}_{i[j]}$ is smaller than $u_{i[j]}$, the fraction multiplying $u_{i[j]}$ is necessarily smaller than 1, implying that this provides a tighter bounds on $x_{i[j]}$.

To use this approximation to compute better bounds than the ones given by simple interval arithmetic, it is possible to leverage the feed-forward structure of the neural networks and obtain bounds one layer at a time. Having included all the constraints up until the i-th layer (not including the i-th layer), it is possible to optimize over the resulting linear programming problem and obtain bounds for all the units of the i-th layer, which in turn will allow us to create the constraints Equation 7 for the next layer.

In addition to the pruning obtained by the convex relaxation, both Planet and Reluplex make use of conflict analysis (Marques-Silva and Sakallah, 1999) to discover combinations of splits that cannot lead to satisfiable assignments, allowing them to perform further pruning of (sub-)problems.

Toy Example (Running Planet) *Planet first computes initial bounds via interval arithmetic for nodes a, b and y. Then it builds a linear program to approximate the network by using the linear constraints Equations 7a, 7b and 7c. Upper and lower bounds of a, b and y are refined by calling an LP solver. In this toy example, after refinement, the lower bound and upper bound of y are replaced with* -3 *and* 5 *respectively. Since it is sufficient to prove the property with the refined lower bound, the algorithm exits before entering the main loop where a Satisifiability solver is called.*

5. Improved BaB for NN Verification

As can be seen, previous approaches to neural network verification have relied on methodologies developed in three communities: optimization, for the estimation of upper and lower bounds; verification, especially SMT; and machine learning, especially the feed-forward nature of neural networks for the creation of relaxations. A natural question that arises is "Can other existing literature from these domains be exploited to further improve neural network verification?" Our Branch-and-Bound framework makes it easy to answer this question. With its help, we can easily identify and consequently provide a non-exhaustive list of techniques to speed-up verification algorithms.

5.1. Better Bounding

While the relaxation proposed by Ehlers (2017a) is tighter than the one used by Reluplex, it can be improved further still. Specifically, after a splitting operation, on a newly generated (sub-)problem, we can refine all the $\mathbf{l_i}, \mathbf{u_i}$ bounds by, for instance, formulating corresponding linear programming problems with added constraints from the split. Then, we solve for minimum for $l_{i[j]}$ and maximum for $u_{i[j]}$. With refined upper and lower bounds, we are able to introduce smaller convex relaxation hulls to obtain tighter relaxations. We show the importance of this in the experiments section with the **reluBaB** method that performs splitting on the activation like Planet but updates its intermediate bounds approximation completely at each step. However, it should be noted there is a trade-off between the benefits of tighter relaxation and the overall computational efficiency. Since updating all the $\mathbf{l_i}, \mathbf{u_i}$ bounds could be computationally expensive, we also show in the experiments section that, depending on the problem at hand, it is sometimes sufficient to update bounds for some of the layers. The overall gain from a tighter relaxation is not in a linear relationship with the number of intermediate bounds updated.

5.2. Better Branching

In the following, we discuss two possible ways to improve branching strategies.

5.2.1. Branching on Input Domains

In the previous discussions, both Planet and Reluplex adopt the decision to split on the activation of the ReLU non-linearities. Although the decision is intuitive as it provides a clear set of decision variables to fix, these methods ignore another natural branching strategy, namely, splitting on the input domain. There are two main advantages of input domain splitting strategies. Firstly, it is simple and straightforward to apply. Once an input dimension to split on is decided, we only need to modify the associated input constraints for each sub-domain, generated by the split step of the BaB algorithm. There is no need to deal with potential conflicts (e.g. infeasible sub-problem) that could be introduced by fixing a ReLU node. Secondly, it can be argued that since the function encoded by the neural networks are piecewise linear in their input, splitting the input domain could result in the computation of high quality upper and lower bounds. With tighter input bounds, tighter intermediate bounds at all layers can be easily re-evaluated, which might not be the case for splitting on a ReLU node, at least for layers prior to the ReLU node we branch on.

To demonstrate the benefits of input domain splitting, we propose two novel input split algorithms. We will show in experiments sections that domain splitting strategies incorporating our proposed heuristics are highly effective for small scale verification problems with low dimensional input. The first and the most direct algorithm is **BaB** algorithm. Based on a domain with input constrained by Equation 2a, the `split` function would return two subdomains where bounds would be identical in all dimension except for the dimension with the largest length, denoted $i^\star$. The bounds for each subdomain for dimension $i^\star$ are given by $l_{0[i^\star]} \leq x_{0[i^\star]} \leq \frac{l_{0[i^\star]}+u_{0[i^\star]}}{2}$ and $\frac{l_{0[i^\star]}+u_{0[i^\star]}}{2} \leq x_{0[i^\star]} \leq u_{0[i^\star]}$.

In order to exploit the benefits of input domain splitting to the fullest, we introduce the second splitting heuristic by using the highly efficient lower bound computation approach of Wong and Kolter (2018). This approach was initially proposed in the context of robust optimization. However, our unified framework opens the door for its use in verification. We propose a smart branching method **BaBSB** to replace the longest edge heuristic of **BaB**. We proceed as follows. For each input dimension i, we split on it and generate two subdomains, denoted by sub_{i_0} and sub_{i_1}. Then, by using the fast approach in Wong and Kolter (2018), we are able to compute lower bound estimations $f^n_{i_0}$ and $f^n_{i_1}$ of subdomains sub_{i_0} and sub_{i_1} respectively [2]. Finally, we make the split decision by choosing the dimension that improves the domain's lower bound the most after the split, which is the solution of $\arg\max_i(\min(f^n_{i_0}, f^n_{i_1}))$. Here, we have used the minimum of $f^n_{i_0}, f^n_{i_1}$ to represent the improvement achieved over the split on a input dimension. It is possible to replace it with other criteria such as the $\max(f^n_{i_0}, f^n_{i_1})$ or the product of $f^n_{i_0}$ and $f^n_{i_1}$, as used in Khalil et al. (2016) in the context of other MIP problems.

In terms of computing a lower bound f^n for a sub-domain, we assume the verification property has been reformulated and added as final layers to the network that we need to verify on. The modified network has a scalar output. This should be easily doable as discussed in section 3.1. Then, a rough estimate of a lower bound can be obtained by using the following formula introduced in Wong and Kolter (2018): assume an arbitrary sub-domain is upper bounded by the vector $\mathbf{u}_0$ and lower bounded by the vector $\mathbf{l}_0$,

$$\begin{aligned} f^n = &- \sum_{k=1}^{n} \nu_{k+1}^T b_k - \mathbf{u}_0^T[\hat{\nu}_1]_+ + \mathbf{l}_0^T[\hat{\nu}_1]_- \\ &+ \sum_{k=2}^{n} \sum_{j\in\mathcal{I}_i} \frac{u_{k[j]} l_{k[j]}}{u_{k[j]} - l_{k[j]}} [\hat{\nu}_{k[j]}]_+ \end{aligned} \tag{8}$$

where

$$\begin{aligned} \nu_{n+1} &= -1 \\ \hat{\nu}_k &= W_k^T \nu_{k+1}, k = n, \dots, 1 \\ \nu_{k,j} &= \begin{cases} 0 & \text{if } u_{k[j]} > 0 \quad (j \in \mathcal{I}_k^-) \\ \hat{\nu}_{k[j]} & \text{if } l_{k[j]} > 0 \quad (j \in \mathcal{I}_k^+) \\ \frac{u_{k[j]}}{u_{k[j]}-l_{k[j]}}[\hat{\nu}_{k[j]}]_+ - \frac{u_{k[j]}}{u_{k[j]}-l_{k[j]}}[\hat{\nu}_{k[j]}]_- & \text{otherwise} \quad (j \in \mathcal{I}_k) \end{cases} \\ &\text{for } k = n, \dots, 2. \end{aligned}$$

Here, $[v]_-, [v]_+$ represent negative and positive parts of an element v respectively. The term I_k^- denotes the set of activation nodes whose upper bounds are negative while I_k^+ contains activation nodes whose lower bounds are positive. The rest of activation nodes belong to I_k. This approach is computationally efficient as one computation of f^n is equivalent to one backward pass due the recursive nature of v_k and $\hat{v}_k$. Here, we have directly used all intermediate bounds $u_{k[j]}$ and $l_{k[j]}$. These intermediate bounds are actually computed beforehand in a similar fashion with above formula. Each $l_{k[j]}$ can be treated as a lower bound to a sub-network consisting of layers prior to it and $u_{k[j]}$ are obtained by negating the signs of the sub-network. Given the input constraints, we proceed layer by layer to compute all intermediate bounds $u_{k[j]}$ and $l_{k[j]}$.

2. For clarity, $f^n_{i_0}$ is a lower bound estimation of the minimum output of the network **net** can reach on sub_{i_0}

Once a domain splitting decision is made, two sub-domains are generated. On each sub-domain, we use an LP solver to refine intermediate bounds, as mentioned in Section 5.1. Finally, we call the LP solver again to compute the domain lower bound (`prob_ub`). Performance of **BaB** and **BaBSB** are included in the experiments section.

5.2.2. Branching on ReLU Activation Nodes

Despite their success in small-scale verification problems, input domain splitting methods are often found to be inadequate for large scale networks, as there are several limitations of input domain branching strategies. Firstly, for some methods (e.g. **BaBSB**), their computation cost for making a branching decision increases at least linearly with input dimensions. High computational cost renders these method infeasible for high input dimensional problems. Secondly and more importantly, potential input branching decisions constitute a fairly small portion of all potential branching decisions for large-scale network architecture, which contains a sizable number of unfixed ReLU activation nodes (each is a valid potential branching point). Focusing on input domain splitting alone significantly limits the power of verification methods. Given the wide and almost dominant usage of deep convolutional network in various tasks, developing an effective and computationally cheap heuristic for branching on ReLU non-linearities is of considerable importance for verification.

In this section, we propose a new smart branching algorithm **BaBSR** on the activation of the ReLU non-linearities. So far, to the best of our knowledge, existing ReLU node splitting methods are **Reluplex**, **Planet** and **Neurify**. We show that **BaBSR** enjoys various benefits over the existing methods, although all methods use the same **branching rule**: for a given ReLU node (a node $x_{i[j]} = \max\left(\hat{x}_{i[j]}, 0\right)$ is split out into two subdomains:$\{x_{i[j]} = 0, \hat{x}_{i[j]} \leq 0\}$ and $\{x_{i[j]} = \hat{x}_{i[j]}, \hat{x}_{i[j]} \geq 0\}$). To start, unlike **Planet** which does not have any heuristic to make splitting decisions, **BaBSR** uses a simple and fast heuristic to prioritise which unfixed ReLU node to split on. **Reluplex** uses the SMT core to handle the splitting order and **Neurify** computes gradient scores to prioritise ReLU nodes. We show in experiments that the prioritising strategy of **BaBSR** is much more successful than that of **Reluplex** and **Neurify** in selecting an effective ReLU node. Additionally, **BaBSR** has a convergence guarantee and encourages early termination, which means verification problems can be solved completely and efficiently. Once a ReLU node has been selected, **BaBSR** calls a commercial solver to obtain a lower bound for each sub-problem with the added constraint $\hat{x}_{i[j]} \leq 0$ or $\hat{x}_{i[j]} \geq 0$ respectively. The algorithm continues until line 5 in Algorithm 1 is satisfied. Convergence guarantee is inherently supported by the algorithm while early termination through finding adversarial examples is assisted by the prioritisation we used in generating sub-problems.

The heuristic used for prioritising ReLU nodes is based on the similar idea to the one used in **BaBSB**. For an arbitrary picked-out domain, we refer to the lower bound of the network minimum on the domain as f^n. In order to decide which unfixed ReLU nodes to split on, for each potential splitting option (any unfixed ReLU node), we attempt to compute a rough estimate of the potential improvement to the lower bound. We then make the splitting decision by choosing the unfixed node with the largest estimated improvement.

In detail, we estimate the improvement on splitting an arbitrary unfixed ReLU node via a modified application of Equation 8. We first observe that when imposing a constraint on an arbitrary unfixed ReLU node $x_{i[j]}$, we will force $v_{i[j]}$ to be either 0 (ReLU is in a blocking state) or $\hat{v}_{i[j]}$ (ReLU is in a completely passing state). As a direct result, the associated terms $v_{i[j]}b_{i-1[j]}$ and $\frac{u_{i[j]}}{u_{i[j]}-l_{i[j]}}[\hat{\nu}_{i[j]}]_+$ in Equation 8 will change accordingly. Furthermore, v_k, $\hat{v}_k$ for $k < i$ as products of v_i will take different values as well and so do their associated terms in Equation 8 . It is possible to compute lower bounds f^n by assuming $v_{i[j]} = 0$ and $v_{i[j]} = \hat{v}_{i[j]}$ and then take the minimum (or maximum, product etc.) of the two cases to represent the improvement made if splitting on $x_{i[j]}$. However, doing this would require two full or partial (only v_k, $\hat{v}_k$ for $k < i$ need to be updated) backward passes for one ReLU splitting choice, which can be computationally expensive when the number of unfixed ReLU nodes is large. To deal with this issue, we use a key observation when dealing with different data sets: when the weights (similar for bias) on each layer are of same magnitudes, the potential improvement of each $x_{i[j]}$ is generally dominated by the changes of terms $v_{i[j]}b_{i-1[j]}$ and $\frac{u_{i[j]}}{u_{i[j]}-l_{i[j]}}[\hat{\nu}_{i[j]}]_+$. Since a rough estimation is sufficient in this scenario, we thus propose to evaluate each ReLU split choice $x_{i[j]}$ by computing a ReLU score $s_{i[j]}$, defined as

$$s_{i[j]} = \left|\max(v_{i[j]}b_{i-1[j]},\ (v_{i[j]} - 1)b_{i-1[j]}) - \frac{u_{i[j]}}{u_{i[j]} - l_{i[j]}}[\hat{\nu}_{i[j]}]_+\right|. \tag{9}$$

We make a decision by picking the ReLU with the largest score. One major benefit of this heuristics is that it is highly computationally efficient. As the same v_i and $\hat{v}_i$, for $1 < i < n-1$, are used for computing all unfixed ReLUs scores, the recursive formulations of v_i and $\hat{v}_i$ enable us to compute all ReLU scores within one single backward pass, regardless the total number of unfixed ReLU nodes.

To maximize the performance of the heuristic, we have also incorporated into the heuristic other useful observations. Firstly, we refer to a convolution layer as sparse if it contains mostly zeros when linearlized. We found that splitting on the sparsest layer is often ineffective in terms of improving the global lower bound when compared with choices on other layers. In addition, ReLU scores are likely to fail in giving good indications when all of them are close to zero. Thus, given a network that contains a large convolution layer with a small kernel, we do not consider the unfixed ReLU nodes on this layer until all other improvements computed are relatively small. When this happens, we consider the heuristic used in prioritising ReLU nodes to be no longer effective. Hence, other selecting strategies should be used, such as, random selections with a preference over non-sparse layers.

Finally, unlike **BaBSB**, **BaBSR** does not call an LP solver to compute tight intermediate bounds. The main applications of **BaBSR** should be networks with large convolution layers. For these networks, computing tight intermediate bounds via an LP solver is computationally infeasible. Thus, once a ReLU split choice is made, we explicitly replace the upper or lower bound of the corresponding ReLU node to 0 for each newly generated sub-problem and then update intermediate bounds by the better of interval arithmetic bounds and bounds computed using the method of Wong and Kolter (2018). Overall, with rough improvement estimations of ReLU choices and loose intermediate bounds, **BaBSR** is significantly cheap for each branching step. While many more branches might be required to solve a property, we often find this trade-off is worthwhile as will be seen in the experiments section.

5.3. Other Potential Improvements

We also list some potential improvements that could be made in future research. One possible area of improvement lies in the tightness of the bounds used. We note that Equation 7 is very closely related to the Mixed Integer Formulation of Equation 4. In fact, it corresponds to level 0 of the Sherali-Adams hierarchy of relaxations (Sherali and Adams, 1994). One possible improvement is to use stronger relaxations by exploring higher levels of the hierarchy. This would jointly constrain groups of ReLUs, rather than linearising them independently. A related work is that of Anderson et al. (2019), in which a MIP formulation for neural networks using stronger relaxations is proposed.

One advantage of the Branch-and-Bound framework is that it is not restricted to piecewise linear networks, which is not true for methods such as **Reluplex**, **Planet** or the MIP encoding. Any type of networks for which an appropriate bounding function can be found will be verifiable with a Branch-and-Bound based method. In order for Branch-and-Bound to achieve good performance on various kinds of networks, developing appropriate bounding functions is necessary. Recent advances on bounds for activations such as sigmoid or hyperbolic tangent have been made in Dvijotham et al. (2018). While their focus is on incomplete methods, our Branch-and-Bound framework makes it readily usable for complete verification as well.

6. Experimental Setup

The problem of PL-NN verification has been shown to be NP-complete (Katz et al., 2017a). Meaningful comparison between approaches therefore needs to be experimental. We use a timeout of two hours for each experiment, unless otherwise stated.

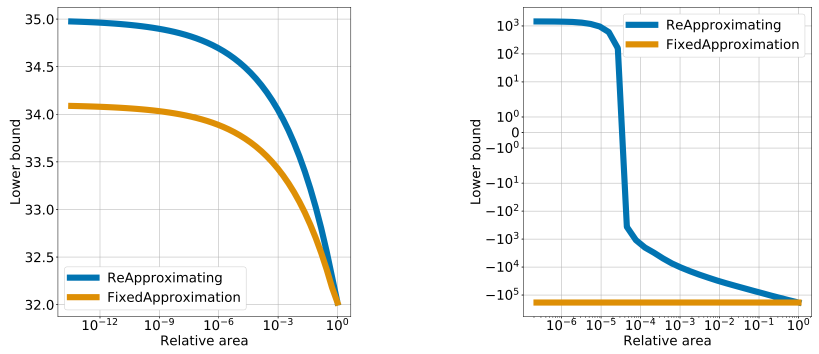
6.1. Methods

The simplest baseline we refer to is **BlackBox**, a direct encoding of Equation 2 into the Gurobi solver, which will perform its own relaxation, without taking advantage of the problem's structure.

For the SMT based methods, **Reluplex** and **Planet**, we use the publicly available versions (Ehlers, 2017b; Katz et al., 2017b). Both tools are implemented in C++. We wrote software to support conversion between the input formats of both solvers in both directions. However, it is worth noting that, we do not change the underlying GLPK solver for linear programming. All the other methods use a potentially faster Gurobi LP solver. The reader is reminded to take this key difference into account when studying our results.

We also evaluate the potential of using MIP solvers, based on the formulation of Equation 4. Due to the lack of availability of open-sourced methods at the time of our experiments, we reimplemented the approach in Python, using the Gurobi MIP solver. We report results for a variant called **MIPplanet**, which uses bounds derived from Planet's convex relaxation rather than simple interval arithmetic. Both the **MIPplanet** and **BlackBox** are not treated as simple feasibility problem but are encoded to minimize the output $\hat{x}_n$ of Equation 2b, with a callback interrupting the optimization as soon as a negative value is found. Additional discussions on encodings of the MIP problem can be found in the appendix.

In our benchmark, we include the methods derived from our Branch-and-Bound analysis. Our implementation follows Algorithm 1 faithfully, is implemented in Python and uses



(a) Approximation on a CollisionDetection net.

(b) Approximation on a deep net from ACAS.

Figure 3: Quality of the linear approximation, depending on the size of the input domain. We plot the value of the lower bound as a function of the area on which it is computed (higher is better). The domains are centered around the global minimum and repeatedly shrunk. Rebuilding completely the linear approximation at each step allows to create tighter lower-bounds thanks to better $\mathbf{l_i}$ and $\mathbf{u_i}$, as opposed to using the same constraints and only changing the bounds on input variables. This effect is even more significant on deeper networks.

Gurobi to solve LPs. The `pick_out` strategy consists in prioritising the (sub-)problem that currently has the smallest lower bound. Upper bounds are generated by randomly sampling points on the considered (sub-)problem or directly compute the network value at the lower bound solution provided by Gurobi, for which we use the convex approximation of Ehlers (2017a) to obtain lower bounds. Motivated by the observation shown in Figure 3, which demonstrates the significant improvements it brings especially for deeper networks, we do not always using a single approximation of the network as was done in Ehlers (2017a). Bearing in mind the trade-off between benefits of tighter relaxation and computational costs, we rebuild the approximation via calling an LP solver to recompute none or partial or all intermediate bounds for each sub-problem. This decision should be made on a case by case basis depending on the size of a network architecture, the number of the input dimensions and the magnitude and the correlation of layer weights. To better study the trade-off, we have incorporated different approximation strategies into different algorithms and compared them on several data sets.

For `split`, we focus on two types of split: input domain split and ReLU activation split. In each case, we consider naive split methods and improved versions. Specifically, in terms of input domain split, the naive methods (e.g. **BaB**) simply performs branching by splitting the input domain in half along its longest edge, while the improved methods (e.g. **BaBSB**) does it by splitting the input domain along the dimension that gives the estimated best improvement to the global lower bound. Estimations are made through the fast bounds formula provided in Wong and Kolter (2018). Regarding the ReLU node split, we study methods which always split on the first or a random unfixed ReLU node on the first layer containing unfixed ReLU nodes. More advanced methods (e.g. **BaBSR** and **BaBSRL**) prioritize ReLU nodes through the criteria introduced in Equation 9. Each specific method is a combination of the choice of approximation strategies (how to rebuild approximations) and branching strategies (what branching heuristics to use). We summarize all Branch-and-Bound methods considered in Table 1.[3] We use abbreviations **imb** for intermediate bounds and `prob_ub` and `prob_lb` are the same as those defined in Algorithm 1.

Method	**Branching Type**	**Branching Heuristics**	**Approximations**	
BaBSB	**Input**	fast bounds Wong and Kolter (2018)	**imb:**	LP solver
			`prob_lb`	LP solver
			`prob_ub`	random sampling
BaB		longest edge	**imb:**	LP solver
			`prob_lb`	LP solver
			`prob_ub`	random sampling
reluBaB	**ReLU**	the first unfixed ReLU node on the first layer containing unfixed ReLU nodes	**imb:**	LP solver
			`prob_lb`	LP solver
			`prob_ub`	random sampling
BaBSR		prioritization via the criteria defined in Equation 9	**imb:**	better of interval bounds and bounds of Wong and Kolter (2018)
			`prob_lb`	LP solver
			`prob_ub`	solution of the LP
BaBSRL		prioritization via the criteria defined in Equation 9	**imb:**	update the bounds of the layer right before the ReLU node selected via an LP solver; update the rest of the layers via better of interval bounds and bounds of Wong and Kolter (2018)
			`prob_lb`	LP solver
			`prob_ub`	solution of the LP

Table 1: All Branch-and-Bound methods considered in the experiments section.

6.2. Data Sets

We perform verification on six data sets of properties and report the comparison results.

The **CollisionDetection** data set (Ehlers, 2017a) attempts to predict whether two vehicles with parameterized trajectories are going to collide. 500 properties are extracted from problems arising from a binary search to identify the size of the region around training examples in which the prediction of the network does not change. The network used is

3. We mention one implementation strategy used to achieve improved performances. For all BaB methods, an LP solver is called to compute the lower bound of a subdomain. In our case, the LP solver used is Gurobi. We found that when Gurobi is used to compute all intermediate bounds, it is faster to reintroduce the LP problem in a layer by layer order to Gurobi for each sub-problem. That is, given a sub-problem, we first create a new Gurobi model instance and introduce all constraints up to the first non-activation layer. Then we compute all upper and lower bounds for the layer via Gurobi. After this is done, we add constraints up to the second non-activation layer and compute all bounds. This procedure continues until we reach the final layer and obtain a lower bound of the sub-problem by solving the corresponding LP problem in its complete form. However, for methods that do not require tight intermediate bounds, obtained via an LP solver, it is cheaper to create a single Gurobi model instance at the start. Then for each sub-problem, we only update the constraints of the Gurobi model to be consistent with the LP of the sub-problem and compute a lower bound for it.

relatively shallow but due to the process used to generate the properties, some lie extremely close between the decision boundary between satisfiable (SAT) and unsatisfiable (UNSAT). Recall that satisfiable refers to properties that are false (a counterexample if found) while unsatisfiable refers to properties that are true (no counterexample exists). Results presented in Figure 4a therefore highlight the accuracy of methods.

The **Airborne Collision Avoidance System (ACAS)** data set, as released by Katz et al. (2017a) is a neural network based advisory system recommending horizontal manoeuvres for an aircraft in order to avoid collisions, based on sensor measurements. Each of the five possible manoeuvres is assigned a score by the neural network and the action with the minimum score is chosen. The 188 properties to verify are based on some specification describing various scenarios. Due to the deeper network involved, this data set is useful in highlighting the scalability of the various algorithms.

The **Robust MNIST Network** is adopted from the network trained with the strategies proposed in Wong and Kolter (2018). The network contains 2 convolution layers followed by 2 fully connected layers with a total number of 4804 activation nodes. Since on a network of this size each LP requires more than 2 seconds, the total number of branches that could be taken within a timeout is low. To better evaluate the performance of the branching heuristic used in **BaBSR**, we also introduce a reduced version, **reduced Robust MNIST Network**. The reduced network has the same structure as the original one but fewer hidden nodes on each hidden layer. The total number of ReLU nodes in the reduced network is 1226. On the reduced network, each LP requires only 0.17 seconds, which allows a large number of branching decisions to be made before timeout. For MNIST networks, the natural properties to verify are whether the predicted label changes if each input image is allowed to be perturbed within an ϵ-infinity norm ball.[4] Since each combination of an image in the MNIST test set and an ϵ constitute a valid property, we randomly select test images and verify properties at a set of pre-specified epsilons ranging from 0.14 to 0.175 for **Robust MNIST Network** and from 0.11 to 0.14 for **reduced Robust MNIST Network**. Recall that, on the same network, epsilon values determine the difficulty level of verification properties. We use a set of epsilon values for both networks to allow comprehensive evaluations. Higher value of ϵ is used for the large network, as the network is more robust. Due to the large number of properties, we restrict the timeout to be one hour on these two data sets.

Existing data sets do not allow us to explore the impact of various problem/model parameters such as depth, number of hidden units, input dimensionality and correlation between hidden nodes on the same layer. Our data sets, **PCAMNIST** and **TwinStream**, remove this deficiency, and can prove helpful in analysing future verification approaches as well. They are generated in a way to give control over different parameters. Specifically, **PCAMNIST** is mainly used for evaluating methods over different network architecture. It has a much wider range in terms of depth, number of hidden units and input dimensionality than **TwinStream** but no particular layer correlation is introduced. On the other hand,

4. For a given image x with the predicted label y^{targ} and a given ϵ, the property to be verified is $\max(f(x')_{y^*} - f(x')_{y^{\text{targ}}}) < 0$ for $\forall x'$ s.t. $\|x' - x\|_\infty < \epsilon$, where y^* is any label. For **MIPplanet**, the encoding of the max function is given in Appendix B.1. For other methods, the max function is encoded as a combination of linear functions and ReLUs as introduced in Appendix B.2. Although it is conceptually simpler to deal with max function directly, we saw improved performance when we replace max function with ReLUs and hence the encoding decisions.

TwinStream is specially designed such that hidden nodes on the same layer are highly correlated. It allows us to explore the trade-off between different bounding strategies. Details of the data set construction are given in the appendix.

Finally, we summarize in Table 2 the characteristics of all of the data sets used for the experimental comparison.

Data set	**Count**	**Model Architecture**
Collision Detection	500	6 inputs 40 hidden unit layer, MaxPool 19 hidden unit layer 2 outputs
ACAS	188	5 inputs 6 layers of 50 hidden units 5 outputs
PCAMNIST	27	10 or {5, 10, 25, 100, 500, 784} inputs 4 or {2, 3, 4, 5, 6, 7} layers of 25 or {10, 15, 25, 50, 100} hidden units, 1 output, with a margin of +1000 or {-1e4, -1000, -100, -50, -10, -1 ,1, 10, 50, 100, 1000, 1e4}
reduced ROBUST MNIST	1200	28 by 28 inputs Conv2d(1,4,4, stride=2, padding=1) Conv2d(4,8,4, stride=2, padding=1) linear layer of 50 hidden units linear layer of 10 hidden units $\mathcal{L}_\infty$ ball radius selected are {0.11, 0.115, 0.12, 0.125, 0.127, 0.13, 0.14}
ROBUST MNIST	1000	28 by 28 inputs Conv2d(1,16,4, stride=2, padding=1) Conv2d(16,32,4, stride=2, padding=1) linear layer of 100 hidden units linear layer of 10 hidden units $\mathcal{L}_\infty$ ball radius selected are {0.14, 0.15, 0.155, 0.16, 0.165, 0.17, 0.175}
TwinStream	81	{5, 10, 25} inputs {2, 4, 5} layers of {5, 10, 25} hidden units, 1 output, with a margin of {1e2, 1, 10}

Table 2: Details of all the data sets. For PCAMNIST, we use a base network with 10 inputs, 4 layers of 25 hidden units and a margin of 1000. We generate new problems by changing one parameter at a time, using the values inside the brackets.

6.3. Evaluation Criteria

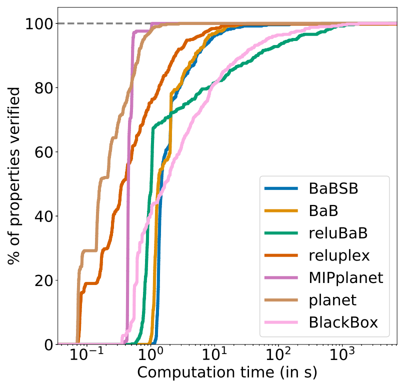
For each of the data sets, we compare different methods using the same protocol. We attempt to verify each property with a timeout of two hours (with an exception of one hour timeout for the reduced Robust Network experiment and the Robust Network experiment due to the large number of properties), and a maximum allowed memory usage of 20GB, on a single core of a machine with an i7-5930K CPU. We measure the time taken by the solvers to either prove or disprove the property. If the property is false and the search problem is therefore satisfiable, we expect from the solver to exhibit a counterexample. If the returned input is not a valid counterexample, we don't count the property as successfully proven, even if the property is indeed satisfiable. All code and data necessary to replicate our analysis have been released.

7. Analysis

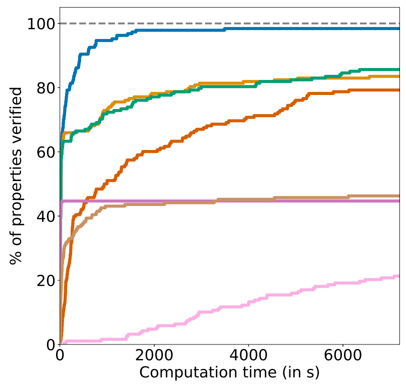
We perform an ablation study to evaluate the performance of different methods on various data sets.

7.1. Small Networks

We first consider small networks with low dimensional input. These are networks of CollisionDetection and ACAS. When networks are small, computing all intermediate bounds via an LP solver is computationally affordable. In these cases, gains from tighter relaxations are often significant and the tightest intermediate bounds should be used to achieve an ideal performance. As a result, methods like **BaBSR** and **BaBSRL** that employ rough estimated intermediate bounds are not included in this section. In Figure 4a, on the shallow networks



(a) CollisionDetection data set.



(b) ACAS data set.

Figure 4: Proportion of properties verifiable for varying time budgets depending on the methods employed. A higher curve means that for the same time budget, more properties will be solvable. All methods solve CollisionDetection quite quickly except **reluBaB**, which is much slower and **BlackBox** which produces several incorrect counterexamples.

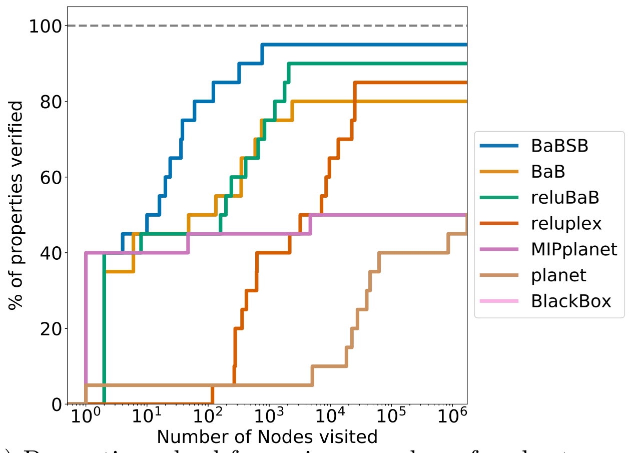
of CollisionDetection, most solvers succeed against all properties in about 10 seconds. In particular, the SMT inspired solvers **Planet**, **Reluplex** and the MIP solver are extremely fast. On the deeper networks of ACAS, in Figure 4b, no errors are observed but most methods timeout on the most challenging testcases. The best baseline is **Reluplex**, which reaches 79.26% success rate at the two hour timeout, while our best method, **BaBSB**, already achieves 98.40% with a budget of one hour. To reach Reluplex's success rate, the required runtime is two orders of magnitude smaller.

We are also able to identify the factors that allow our methods to perform well. We point out that the only difference between **BaBSB** and **BaB** is the smart branching, which represents a significant part of the performance gap. Furthermore, on networks with low dimensional inputs, branching over the ReLU activation nodes rather than over the inputs does not contribute much, as shown by the small difference between **BaB** and **reluBaB**. The rest of the performance gap can be attributed to using better bounds: **reluBaB** significantly outperforms **planet** while using the same branching strategy and the same convex relaxations. The improvement comes from the benefits of rebuilding the approximation at each step shown in Figure 3.

Figure 5 presents additional analysis on a 20-property subset of the ACAS data set, showing how the methods used impact the need for branching. Smart branching and the use of better lower bounds reduce heavily the number of subdomains to explore.

Method	**Average time per Node**
BaBSB	1.81s
BaB	2.11s
reluBaB	1.69s
reluplex	0.30s
MIPplanet	0.017s
planet	1.5e-3s

Table 3: Average time to explore a node for each method.



(a) Properties solved for a given number of nodes to explore (log scale).

Figure 5: Trade-off between bounding cost and total number of branches required. Figure 5a shows how many subdomains needs to be explored before verifying properties while Table 3 shows the average time cost of exploring each subdomain. Our methods have a higher cost per node but they require significantly less branching, thanks to better bounding. Note also that between **BaBSB** and **BaB**, the smart branching reduces by an order of magnitude the number of nodes to visit.

7.2. Large Networks

We then study the performance of various methods on the reduced Robust MNIST Network and the Robust MNIST Network. Due to the large number of properties (1200 and 1000 respectively in order to cover a wide range of difficulties), we treat **MIPplanet**[5] as the benchmark and rule out methods which could not perform at least at similar level to **MIPplanet** over simple properties, that is, those that could be solved within 100 seconds by **MIPplanet**. For a comprehensive study of available complete methods, we have also included **ERAN**, ETH Robustness Analyser for neural networks. It is developed on a series of work (Singh et al., 2018, 2019a,b) that apply abstract interpretation for neural network verification. **ERAN**[6] mainly focuses on incomplete verification of MNIST, CIFAR10 and a subset of ACAS properties but also supports a complete mode, rendering itself a fair candidate for comparison studies. We observe from Figure 6 that most methods fail even on simple properties. **BaBSB** becomes incompetent when the input dimension is high. Although **planet**, **reluplex** and **BaBSR** use the same branching rule and loose bounds of different extent for approximation, the significantly improved performance of **BaBSR** is attributed to its effective ReLU prioritization heuristic and early termination feature. The **reluBaB** method is the only ReLU split method that uses the tightest bounds available throughout the procedure. However, the fact it could not solve a single property reemphasizes the issue of finding a balance between the computational cost and the quality of the relaxation introduced. With limited computing resources, more gains might be achieved via a good branching strategy than a tighter relaxation.

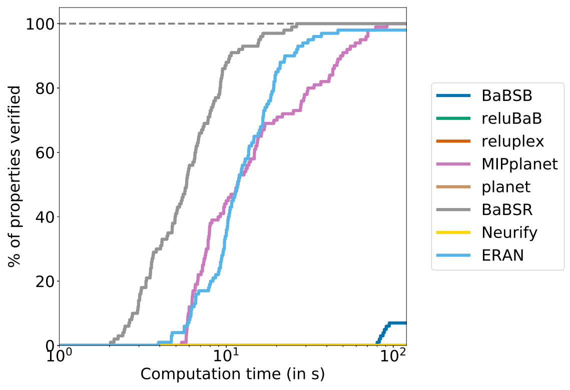


Figure 6: On simple properties, all **planet**, **reluBaB**, **reluplex** and **Neurify** timed out on every single property for 100s time limit. **BaBSB** managed to solve some properties but its performance is significantly worse than that of **MIPplanet**. Only **BaBSR** and **ERAN** perform better than and similarly to **MIPplanet** and are thus ran on all properties of the reduced Robust MNIST Network and Robust MNIST Network.

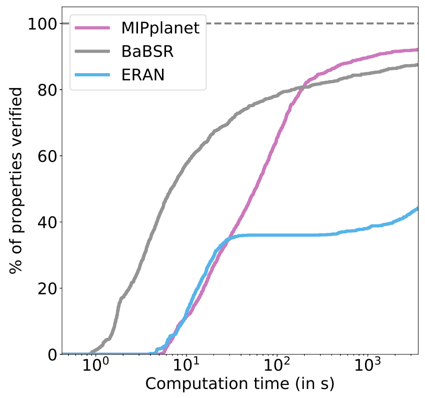
We thus only compare among **BaBSR**, **ERAN** and **MIPplanet** on all properties of reduced Robust MNIST Network and the Robust MNIST Network. Overall, **BaBSR** achieved the best performance on both data sets. In detail, when the reduced Robust MNIST network

5. For large networks, calling an LP solver to compute all intermediate bounds in computationally expensive. To ensure a fair comparison between **BaBSR** and **MIPplanet**, the same intermediate bounds as that of **BaBSR** are used for building the LP model.
6. For our experiments, we have used the complete version of **ERAN** with refinepoly domain and 10 seconds MILP timeout, as suggested in Singh et al. (2019b). All the rest of hyper-parameters are set as default values. Each process is restricted to a single cpu core.

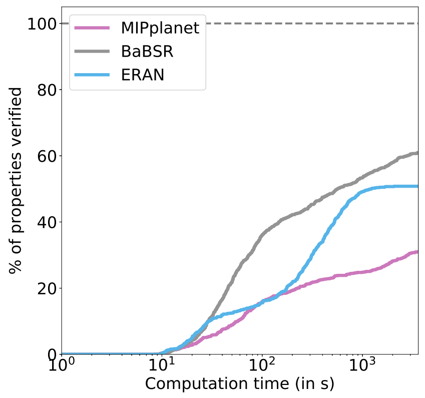
is considered, **BaBSR** outperforms **MIPplanet** significantly on easy properties, but the performance gap decreases when properties become more and more difficult. On the most challenging ones, **MIPplanet** slightly wins over **BaBSR**. A likely cause for the declined performance of **BaBSR** could be the split heuristic used. After a certain number of branches, the rough estimations used by the split heuristic are no longer fair representations of potential improvements that could be made by available branching decisions. This conjecture is consistent with what we observe on the Robust MNIST Network, where **BaBSR** significantly outperforms **MIPplanet** on all properties. Since each LP is expensive (requires more than 5 seconds) on the large network, the total number of branches that could be taken within the time limit is at least 10 times smaller than that on the reduced network, which means the heuristic of **BaBSR** probably remains effective throughout the verification procedure. **ERAN** is not as competent as the other two methods on the reduced Robust data set but it beats **MIPplanet** on the Robust data set. Compared to **MIPplanet**, **ERAN**, in its complete mode, adopts a similar idea of solving a MIP instance. However, different convex relaxations and intermediate bounds are used. Specifically, in these experiments, **ERAN** uses intermediate bounds collected by running RefinePoly analysis. Those computationally more expensive but potentially tighter intermediate bounds used by **ERAN** might explain its varying performance to that of **MIPplanet** on these data sets. In addition, when the network size increases, the time required by the LP solver increased exponentially, which becomes the main bottleneck in verifying properties on large networks in the Branch-and-Bound framework. Thus, in order to tackle real life verification problems, which often involve networks considerably larger than the Robust MNIST Network, it is important to develop an efficient LP solver that, by exploiting the special structure of neural networks, scales well with their size. At the same time, developing a better split strategy that is computationally cheap yet capable of giving high quality decisions throughout the whole Branch-and-Bound process is also key to the success of Branch-and-Bound methods on dealing with large network verification problems. One recent work (Lu and Kumar, 2020), which employs graph neural networks to imitate strong branching decisions, has demonstrate some success in achieving desired branching strategies.

7.3. Varying Parameters

Finally, we study how the performance of each method is impacted by various parameters. Firstly, consider the case of various network architectures of the PCAMNIST data set. In the graphs of Figure 8, the trend for almost all methods are similar, which seems to indicate that hard properties are intrinsically hard and not just hard for a specific solver. Figure 8a shows an expected trend: the larger the number of inputs, the harder the problem is. Similarly, Figure 8b shows unsurprisingly that wider networks require more time to solve, which can be explained by the fact that they have more non-linearities. The impact of the margin, as shown in Figure 8c is also clear. Properties that are true or false with large satisfiability margin are easy to prove, while properties that have small satisfiability margins are significantly harder. It is interesting to see the inconsistent performance of **MIPplanet**, which could be due to the different strategies used by Gurobi for different sized problems. In addition, consistent results of **BaBSR** outperforming **MIPplanet** on easy problems can be observed. We point out that the PCAMNIST data set is a small data



(a) All properties of reduced Robust MNIST Network.

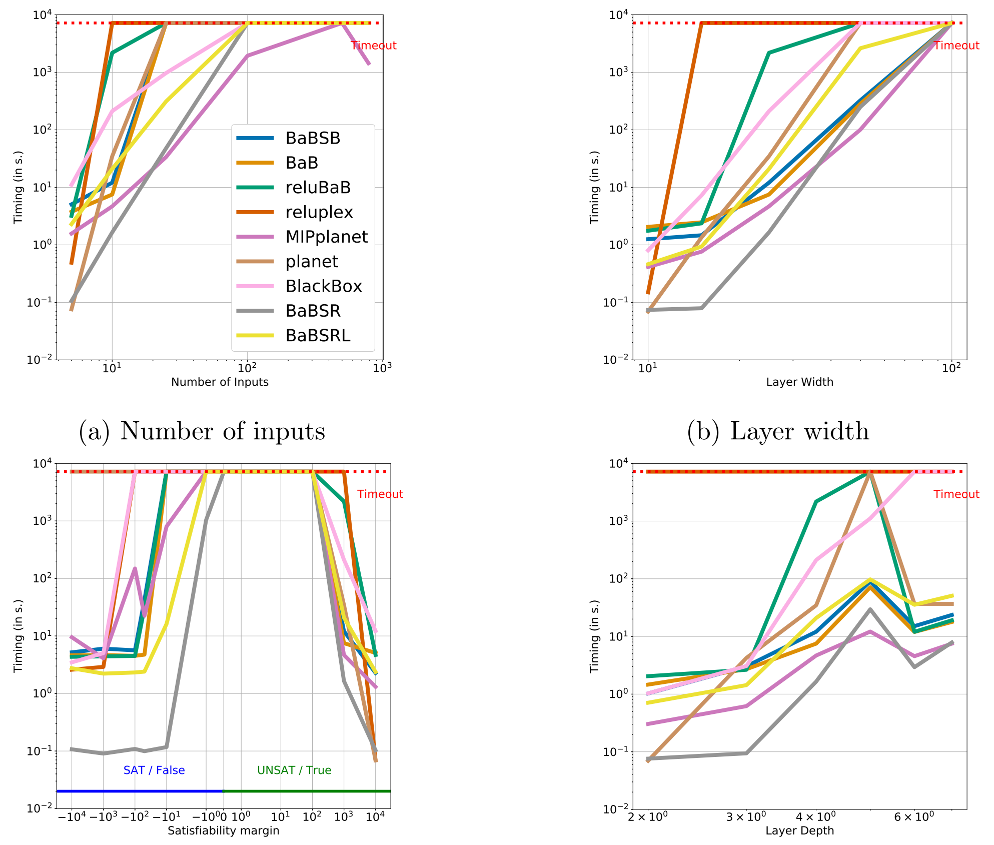


(b) All properties of the Robust MNIST Network.

Figure 7: Cactus plots of properties solved by **MIPplanet**, **BaBSR** and **ERAN** on the reduced Robust MNIST Network (left) and the Robust MNIST Network (right). **BaBSR** outperforms **MIPplanet** except on the difficult properties of the reduced Robust MNIST Network. The declined performance of **BaBSR** might be caused by the ineffectiveness of the branching heuristics at later stage. The better performance of **ERAN** than that of **MIPplanet** on the Robust MNIST Network could be explained by the possible tighter intermediate bounds used by **ERAN**.

set with only 27 networks. Observations should be made with care in terms of generalisability.

The TwinStream data set introduces a possible standpoint to take when selecting splitting methods and bounding methods for BaB algortihms or other non-BaB methods to best solve the properties at hand. In Figure 9, we see that **MIPplanet** performed the best over all properties and input split methods **BaBSB** and **BaB** performed the worst. Despite the fact that all twinladder networks are small, ReLU split strategy should be preferred when highly correlated layers are present. In terms of ReLU split based methods, the method with the tightest relaxation **reluBaB** and the method with the supposedly effective ReLU prioritizing heuristics **BaBSR** performed the worst. This is expected, as when the correlation among the hidden nodes of the same layer is high (by which we mean if we write each node as a linear combination of hidden nodes on the previous layer, the coefficient vectors are highly correlated), the ReLU score used in **BaBSR** will become way too loose to be of any use. For example, if several ReLU nodes $x_{i[p]}$ are highly correlated with the node $x_{i[j]}$, forcing $\hat{x}_{i[j]} \leq 0$ or $\hat{x}_{i[j]} \geq 0$ will lead to notable changes to the upper or lower bound of $x_{i[p]}$ respectively. In extreme cases, it means $\hat{x}_{i[p]} \leq 0$ or $\hat{x}_{i[p]} \geq 0$ if the weights associate with $x_{i[j]}$ and $x_{i[k]}$ have a correlation of one. Estimating improvements by keeping all other terms the same is thus unreasonable in this case. The **reluBaB** method mainly suffers from its computational cost. The method **BaBSRL** lies between **BaBSR** and **reluBaB**. Since only one layer is updated via the LP, **BaBSRL** has tighter relaxations than **BaBSR** so the ReLU prioritising heuristic of **BaBSR** can make better branching decisions in the following steps. Yet, the computational cost of **BaBSRL** is much lower than that of **reluBaB**. An improved performance of **BaBSRL** over **BaBSR** and **reluBaB** can be observed. Overall,



(a) Number of inputs

(b) Layer width

(c) Margin

(d) Network depth

Figure 8: Impact of the various parameters over the runtimes of the different solvers. The base network has 10 inputs and 4 layers of 25 hidden units, and the property to prove is true with a margin of 1000. Each of the plots corresponds to a variation of one of this parameters.

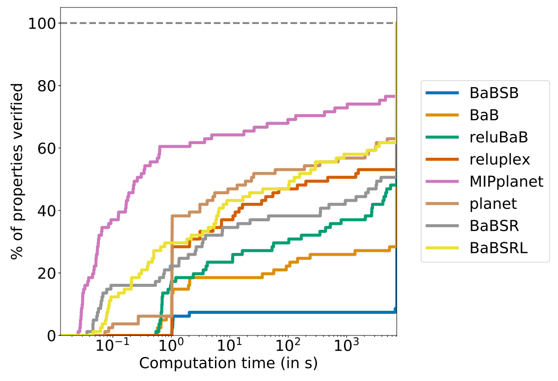


Figure 9: By design, the TwinStream data set consists of UNSAT properties only. All methods returned correct results for properties verified apart from **Reluplex**, which return SAT for several properties. These properties are treated as unsolved for these methods. **MIPplanet** outperforms all other methods on the TwinStream data set.

on the small networks with highly correlated layers like networks of Twinstream, **MIPplanet** is a definite winner. However, as we have observed previously, **MIPplanet** is likely to suffer from scalability issues. We expect **BaBSRL** might be a better option on large networks with highly correlated layers.

8. Conclusion

Through the lens of a unified Branch-and-Bound framework, we have identified the weakness of existing methods and proposed new methods to correct and improve on it. These new methods are effective by achieving considerate performance enhancements on our comprehensive data sets. However, there is still much room for improvements. We illustrate a few based on the BaB framework. In terms of bounding strategies, we observe tighter intermediate bounds could lead to faster convergence but they are expensive to obtain. Given the layer-wise structure of neural networks, exploiting the GPU computing power might be a potential way to compute tighter intermediate bounds cheaply. For branching strategies, methods discussed mainly rely on heuristics, which are likely to fail when problem changes. Recent studies have shown learning heuristics through graph neural networks might overcome the issue but they have high offline cost. Cheaper while effective strategies should be possible. Finally, BaB based methods require solving LPs, the main bottleneck hindering their development. Since LPs are mostly solved to decide whether a branching should be conducted on a (sub-)problem, learning to imitate LP decision could be a potential way to speed up the whole process by orders of magnitudes.

We encourage the development of new methods to address these issues and hope our inclusive and various data sets could facilitate the process by allowing comprehensive evaluations and comparison studies.

Acknowledgments

This work was supported by ERC grant ERC-2012-AdG 321162-HELIOS, EPSRC grant Seebibyte EP/M013774/1 and EPSRC/MURI grant EP/N019474/1. We would also like to acknowledge the Royal Academy of Engineering and FiveAI.

Appendix A. Planet Approximation

The feasible set of the Mixed Integer Programming formulation is given by the following set of equations. We assume that all $\mathbf{l_i}$ are negative and $\mathbf{u_i}$ are positive. In case this isn't true, it is possible to just update the bounds such that they are.

$$\mathbf{l_0} \leq \mathbf{x_0} \leq \mathbf{u_0} \tag{10a}$$
$$\mathbf{\hat{x}_{i+1}} = W_{i+i}\mathbf{x_i} + \mathbf{b_{i+i}} \qquad \forall i \in \{0,\ n-1\} \tag{10b}$$
$$\mathbf{x_i} \geq 0 \qquad \forall i \in \{1,\ n-1\} \tag{10c}$$
$$\mathbf{x_i} \geq \mathbf{\hat{x}_i} \qquad \forall i \in \{1,\ n-1\} \tag{10d}$$
$$\mathbf{x_i} \leq \mathbf{\hat{x}_i} - \mathbf{l_i} \cdot (1 - \boldsymbol{\delta_i}) \qquad \forall i \in \{1,\ n-1\} \tag{10e}$$
$$\mathbf{x_i} \leq \mathbf{u_i} \cdot \boldsymbol{\delta_i} \qquad \forall i \in \{1,\ n-1\} \tag{10f}$$
$$\boldsymbol{\delta_i} \in \{0,1\}^{h_i} \qquad \forall i \in \{1,\ n-1\} \tag{10g}$$
$$\hat{x}_n \leq 0. \tag{10h}$$

The level 0 of the Sherali-Adams hierarchy of relaxation Sherali and Adams (1994) doesn't include any additional constraints. Indeed, polynomials of degree 0 are simply constants and their multiplication with existing constraints followed by linearization therefore does not add any new constraints. As a result, the feasible domain given by the level 0 of the relaxation corresponds simply to the removal of the integrality constraints:

$$\mathbf{l_0} \leq \mathbf{x_0} \leq \mathbf{u_0} \tag{11a}$$
$$\mathbf{\hat{x}_{i+1}} = W_{i+i}\mathbf{x_i} + \mathbf{b_{i+i}} \qquad \forall i \in \{0,\ n-1\} \tag{11b}$$
$$\mathbf{x_i} \geq 0 \qquad \forall i \in \{1,\ n-1\} \tag{11c}$$
$$\mathbf{x_i} \geq \mathbf{\hat{x}_i} \qquad \forall i \in \{1,\ n-1\} \tag{11d}$$
$$\mathbf{x_i} \leq \mathbf{\hat{x}_i} - \mathbf{l_i} \cdot (1 - \boldsymbol{d_i}) \qquad \forall i \in \{1,\ n-1\} \tag{11e}$$
$$\mathbf{x_i} \leq \mathbf{u_i} \cdot \boldsymbol{d_i} \qquad \forall i \in \{1,\ n-1\} \tag{11f}$$
$$\underline{0 \leq \boldsymbol{d_i} \leq 1} \qquad \forall i \in \{1,\ n-1\} \tag{11g}$$
$$\hat{x}_n \leq 0. \tag{11h}$$

Combining the Equations 11e and 11f, looking at a single unit j in layer i, we obtain:

$$x_{i[j]} \leq \min\left(\hat{x}_{i[j]} - l_i(1 - d_{i[j]}), u_{i[j]}d_{i[j]}\right). \tag{12}$$

The function mapping $d_{i[j]}$ to an upperbound of $x_{i[j]}$ is a minimum of linear functions, which means that it is a concave function. As one of them is increasing and the other is decreasing, the maximum will be reached when they are both equals.

$$\begin{aligned} \hat{x}_{i[j]} - l_{i[j]}(1 - d^{\star}_{i[j]}) &= u_{i[j]}d^{\star}_{i[j]} \\ \Leftrightarrow \qquad d^{\star}_{i[j]} &= \frac{\hat{x}_{i[j]} - l_{i[j]}}{u_{i[j]} - l_{i[j]}}. \end{aligned} \tag{13}$$

Plugging this equation for $d^{\star}$ into Equation 12 gives that:

$$x_{i[j]} \leq u_{i[j]}\frac{\hat{x}_{i[j]} - l_{i[j]}}{u_{i[j]} - l_{i[j]}}, \tag{14}$$

which corresponds to the upper bound of $x_{i[j]}$ introduced for Planet (Ehlers, 2017a).

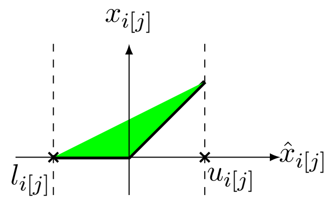


Figure 10: Feasible domain corresponding to the Planet relaxation for a single ReLU.

Appendix B. MaxPooling

For space reason, we only described the case of ReLU activation function in the main paper. We now present how to handle MaxPooling activation, either by converting them to the already handled case of ReLU activations or by introducing an explicit encoding of them when appropriate.

B.1 Mixed Integer Programming

Similarly to the encoding of ReLU constraints using binary variables and bounds on the inputs, it is possible to similarly encode MaxPooling constraints. The constraint

$$y = \max\left(x_1, \ldots, x_k\right) \tag{15}$$

can be replaced by

$$\begin{aligned}
& y \geq x_i && \forall i \in \{1 \ldots k\} && \text{(16a)}\\
& y \leq x_i + (u_{x_{1:k}} - l_{x_i})(1 - \delta_i) && \forall i \in \{1 \ldots k\} && \text{(16b)}\\
& \sum_{i \in \{1 \ldots k\}} \delta_i = 1 && && \text{(16c)}\\
& \delta_i \in \{0, 1\} && \forall i \in \{1 \ldots k\}. && \text{(16d)}
\end{aligned}$$

where $u_{x_{1:k}}$ is an upper-bound on all x_i for $i \in \{1 \ldots k\}$ and l_{x_i} is a lower bound on x_i.

B.2 Reluplex

In the version introduced by (Katz et al., 2017a), there is no support for MaxPooling units. As the canonical representation we evaluate needs them, we provide a way of encoding a MaxPooling unit as a combination of Linear function and ReLUs.

To do so, we decompose the element-wise maximum into a series of pairwise maximum

$$\begin{aligned}
\max\left(x_j, x_2, x_3, x_4\right) = \max(\; & \max\left(x_1, x_2\right), \\
& \max\left(x_3, x_4\right))
\end{aligned} \tag{17}$$

and decompose the pairwise maximums as sum of ReLUs:

$$\max\left(x_1, x_2\right) = \max\left(x_1 - x_2,\ 0\right) + \max\left(x_2 - l_{x_2}, 0\right) + l_{x_2}, \tag{18}$$

where l_{x_2} is a pre-computed lower bound of the value that x_2 can take.

As a result, we have seen that an elementwise maximum such as a MaxPooling unit can be decomposed as a series of pairwise maximum, which can themselves be decomposed into a sum of ReLUs units. The only requirement is to be able to compute a lower bound on the input to the ReLU, for which the methods discussed in the paper can help.

B.3 Planet

As opposed to Reluplex, Planet Ehlers (2017a) directly supports MaxPooling units. The SMT solver driving the search can split either on ReLUs, by considering separately the case of the ReLU being passing or blocking. It also has the possibility on splitting on MaxPooling units, by treating separately each possible choice of units being the largest one.

For the computation of lower bounds, the constraint

$$y = \max\left(x_1, x_2, x_3, x_4\right) \tag{19}$$

is replaced by the set of constraints:

$$\begin{aligned} y &\geq \mathrm{x}_i \qquad \forall i \in \{1 \dots 4\} && \text{(20a)}\\ y &\leq \sum_i \left(x_i - l_{x_i}\right) + \max_i l_{x_i}, && \text{(20b)} \end{aligned}$$

where x_i are the inputs to the MaxPooling unit and l_{x_i} their lower bounds.

Appendix C. Mixed Integers Variants

C.1 Encoding

Several variants of encoding are available to use Mixed Integer Programming as a solver for Neural Network Verification. As a reminder, in the main paper we used the formulation of Tjeng and Tedrake (2019):

$$\begin{aligned} x_i = \max\left(\hat{\mathbf{x}}_\mathbf{i}, 0\right) \quad \Rightarrow \quad \boldsymbol{\delta_i} \in \{0,1\}^{h_i}, \quad & \mathbf{x_i} \geq 0, \qquad \mathbf{x_i} \leq \mathbf{u_i} \cdot \boldsymbol{\delta_i} && \text{(21a)}\\ & \mathbf{x_i} \geq \hat{\mathbf{x}}_\mathbf{i}, \qquad \mathbf{x_i} \leq \hat{\mathbf{x}}_\mathbf{i} - \mathbf{l_i} \cdot (1 - \boldsymbol{\delta_i}). && \text{(21b)} \end{aligned}$$

An alternative formulation is the one of Lomuscio and Maganti (2017) and Cheng et al. (2017a):

$$\begin{aligned} x_i = \max\left(\hat{\mathbf{x}}_\mathbf{i}, 0\right) \quad \Rightarrow \quad \boldsymbol{\delta_i} \in \{0,1\}^{h_i}, \quad & \mathbf{x_i} \geq 0, \qquad \mathbf{x_i} \leq \mathbf{M_i} \cdot \boldsymbol{\delta_i} && \text{(22a)}\\ & \mathbf{x_i} \geq \hat{\mathbf{x}}_\mathbf{i}, \qquad \mathbf{x_i} \leq \hat{\mathbf{x}}_\mathbf{i} - \mathbf{M_i} \cdot (1 - \boldsymbol{\delta_i}). && \text{(22b)} \end{aligned}$$

where $\mathbf{M_i} = \max\left(-\mathbf{l_i}, \mathbf{u_i}\right)$. This is fundamentally the same encoding but with a sligthly worse bounds that is used, as one of the side of the bounds isn't as tight as it could be.

C.2 Obtaining Bounds

The formulation described in Equations 21 and 22 are dependant on obtaining lower and upper bounds for the value of the activation of the network.

C.2.1 Interval Analysis

One way to obtain them, mentionned in the paper, is the use of interval arithmetic (Hickey et al., 2001). If we have bounds $\mathbf{l_i}, \mathbf{u_i}$ for a vector $\mathbf{x_i}$, we can derive the bounds $\hat{\mathbf{l}}_{\mathbf{i+1}}, \hat{\mathbf{u}}_{\mathbf{i+1}}$ for a vector $\hat{\mathbf{x}}_{\mathbf{i+1}} = W_{i+1}\mathbf{x_i} + b_{i+1}$

$$\hat{l}_{i+1[j]} = \sum_k \left(W^{+}_{i+1[j,k]} l^{+}_{i[k]} + W^{-}_{i+1[j,k]} u^{+}_{i[k]} \right) + b_{i+1[j]} \tag{23a}$$

$$\hat{u}_{i+1[j]} = \sum_k \left(W^{+}_{i+1[j,k]} u^{+}_{i[k]} + W^{-}_{i+1[j,k]} l^{+}_{i[k]} \right) + b_{i+1[j]} \tag{23b}$$

with the notation $a^+ = \max(a, 0)$ and $a^- = \min(a, 0)$. Propagating the bounds through a ReLU activation is simply equivalent to applying the ReLU to the bounds.

C.2.2 Planet Linear approximation

An alternative way to obtain bounds is to use the relaxation of Planet. This is the methods that was employed in the paper: we build incrementally the network approximation, layer by layer. To obtain the bounds over an activation, we optimize its value subject to the constraints of the relaxation.

Given that this is a convex problem, we will achieve the optimum. Given that it is a relaxation, the optimum will be a valid bound for the activation (given that the feasible domain of the relaxation includes the feasible domains subject to the original constraints).

Once this value is obtained, we can use it to build the relaxation for the following layers. We can build the linear approximation for the whole network and extract the bounds for each activation to use in the encoding of the MIP. While obtaining the bounds in this manner is more expensive than simply doing interval analysis, the obtained bounds are better.

C.2 Objective Function

In the paper, we have formalised the verification problem as a satisfiability problem, equating the existence of a counterexample with the feasibility of the output of a (potentially modified) network being negative.

In practice, it is beneficial to not simply formulate it as a feasibility problem but as an optimization problem where the output of the network is explicitly minimized.

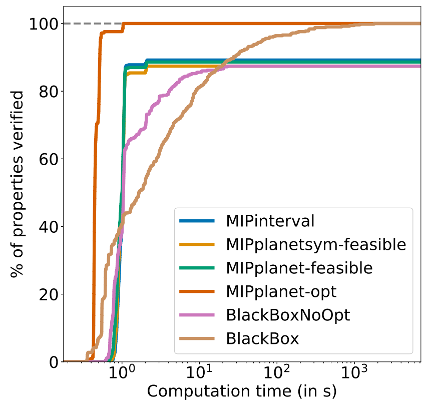
C.3 Comparison

We present here a comparison on CollisionDetection and ACAS of the different variants.

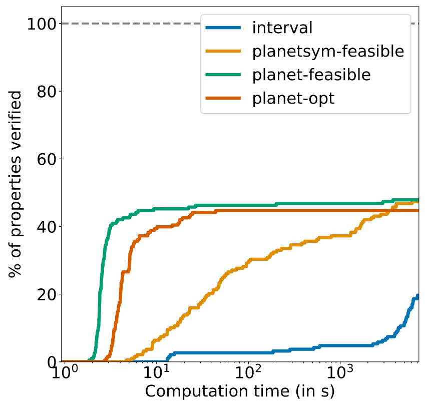
1. **Planet-feasible** uses the encoding of Equation 21, with bounds obtained based on the planet relaxation, and solve the problem simply as a satisfiability problem.

2. **Interval** is the same as **Planet-feasible**, except that the bounds used are obtained by interval analysis rather than with the Planet relaxation.

3. **Planet-symfeasible** is the same as **Planet-feasible**, except that the encoding is the one of Equation 22.



(a) CollisionDetection data set



(b) ACAS data set

Figure 11: Comparison between the different variants of MIP formulation for Neural Network verification.

4. **Planet-opt** is the same as **Planet-feasible**, except that the problem is solved as an optimization problem. The MIP solver attempt to find the global minimum of the output of the network. Using Gurobi's callback, if a feasible solution is found with a negative value, the optimization is interrupted and the current solution is returned. This corresponds to the version that is reported in the main paper.

The comparison also includes two variants of **BlackBox**: **BlackBox** and **BlackBoxNoOpt**. Similarly to **Planet-opt**, **BlackBox** attempts to do global optimization and interrupt the search when a feasible solution with negative value is found. **BlackBoxNoOpt** works like the other MIP encoding by simply encoding the problem as satisfiability.

The first observation that can be made is that when we look at the CollisionDetection data set in Figure 11a, only **Planet-opt** and **BlackBox** solves the data set to 100% accuracy. The reason why the other methods don't reach it is not because of timeout but because they return spurious counterexamples. As they encode only satisfiability problem, they terminate as soon as they identify a solution with a value of zero. Due to the large constants involved in the big-M, those solutions are sometimes not actually valid counterexamples. This is a significant advantage to encoding the problem as optimization problems versus simply as satisfiability problems.

The other results that we can observe is the impact of the quality of the bounds when the networks get deeper, and the problem becomes therefore more complex, such as in the ACAS data set. **Interval** has the worst bounds and is much slower than the other methods. **Planetsym-feasible**, with its slightly worse bounds, performs worse than **Planet-feasible** and **Planet-opt**.

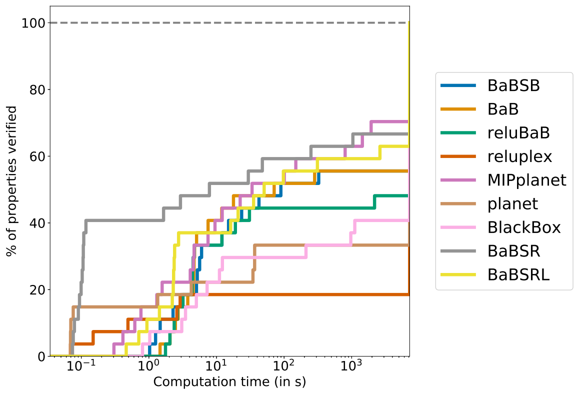


Figure 12: Proportion of properties verifiable for varying time budgets depending on the methods employed. Overall, **BaBSR** performed the best on easy properties but worse than **MIPplanet** on difficult properties, which is consistent to what is observed on properties of the reduced Robust network.

Appendix D. PCAMNIST Details

PCAMNIST is a novel data set that we introduce to get a better understanding of what factors influence the performance of various methods. It is generated in a way to give control over different architecture parameters. The networks takes k features as input, corresponding to the first k eigenvectors of a Principal Component Analysis decomposition of the digits from the MNIST data set. We also vary the depth (number of layers), width (number of hidden unit in each layer) of the networks. We train a different network for each combination of parameters on the task of predicting the parity of the presented digit. This results in the accuracies reported in Table 4.

The properties that we attempt to verify are whether there exists an input for which the score assigned to the odd class is greater than the score of the even class plus a large confidence. By tweaking the value of the confidence in the properties, we can make the property either true or false, and we can choose by how much is it true. This gives us the possibility of tweaking the "margin", which represent a good measure of difficulty of a network.

In addition to the impact of each factors separately as was shown in the main paper, we can also look at it as a generic data set and plot the cactus plots like for the other data sets. This can be found in Figure 12.

Appendix E. TwinStream Details

The networks contain two separate streams, where each of the streams has the same architecture, weights, and inputs. The final layer of the network computes the difference between the outputs of the two streams, and adds a positive bias term, which we will refer to as the margin, denoted as m. As a result, the output is always equal to the value of the final bias. We give explicit formulations of weights and biases of the TwinStream networks. Given a

Network Parameter			**Accuracy**	
Nb inputs	**Width**	**Depth**	**Train**	**Test**
5	25	4	88.18%	87.3%
10	25	4	97.42%	96.09%
25	25	4	99.87%	98.69%
100	25	4	100%	98.77%
500	25	4	100%	98.84%
784	25	4	100%	98.64%
10	10	4	96.34%	95.75%
10	15	4	96.31%	95.81%
10	25	4	97.42%	96.09%
10	50	4	97.35%	96.0%
10	100	4	97.72%	95.75%
10	25	2	96.45%	95.71%
10	25	3	96.98%	96.05%
10	25	4	97.42%	96.09%
10	25	5	96.78%	95.9%
10	25	6	95.48%	95.2%
10	25	7	96.81%	96.07%

Table 4: Accuracies of the network trained for the PCAMNIST data set.

N-layer stream with weights $W_1^s, \dots, W_N^s$ and biases $b_1^s, \dots, b_{N-1}^s$, the corresponding N-layer TwinStream network consists of following weights $W_1, \dots, W_N$ and biases $b_1, \dots, b_N$:

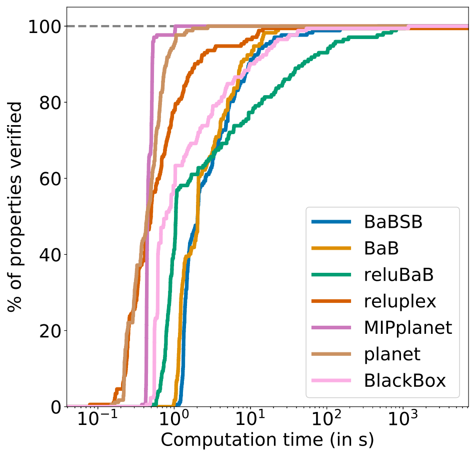
$$W_1 = \begin{bmatrix} W_1^s \\ W_1^s \end{bmatrix}, \qquad W_i = \begin{bmatrix} W_i^s & \mathbf{0} \\ \mathbf{0} & W_i^s \end{bmatrix} \quad \text{for } i \in \{2, \dots, N-1\}, \qquad W_N = \begin{bmatrix} W_N^s \\ -W_N^s \end{bmatrix}.$$

$$b_i = \begin{bmatrix} b_i^s \\ b_i^s \end{bmatrix} \quad \text{for } i \in \{2, \dots, N-1\}, \qquad b_N = m.$$

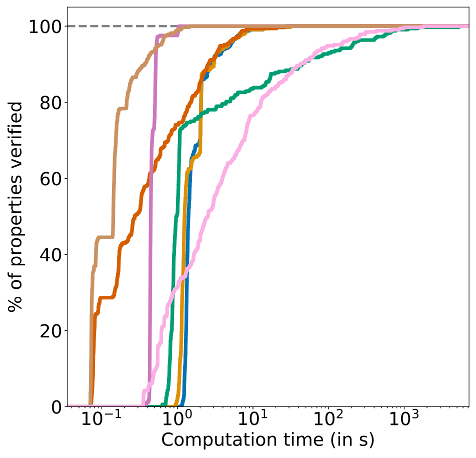
On each of those networks, we attempt to prove the true property that the output of the network is positive. We generate streams with random weights using Glorot initialisation Glorot and Bengio (2010). Various TwinStream Networks are constructed by varying the depth, number of hidden units in each of the stream, number of inputs, and the value of the margin. Note that as opposed to the other two data sets, the weights aren't the result of an optimization process and therefore may not be representative of real use-cases.

Appendix F. Additional Performance Details

Given that there is a difference in the way verification works for SAT problems vs. UNSAT problems, we report also comparison results on the subset of data sorted by decision type.

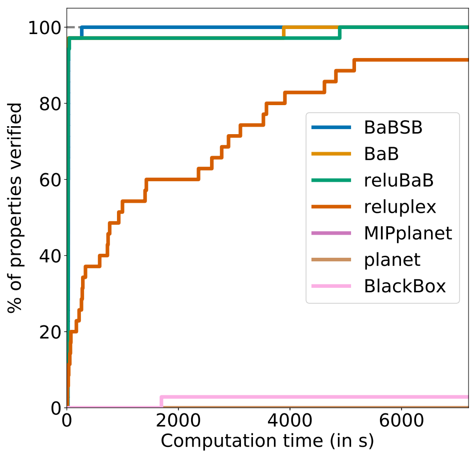


(a) On SAT properties

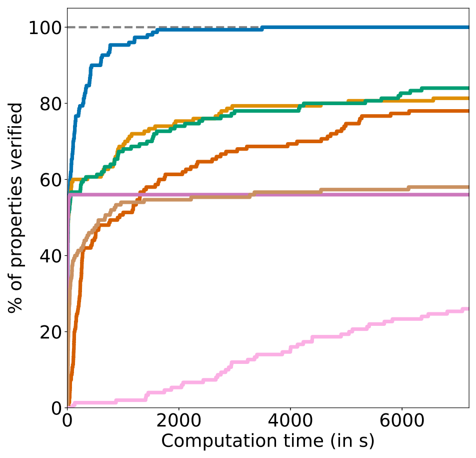


(b) On UNSAT properties

Figure 13: Proportion of properties verifiable by different methods under varying time budgets on the **CollisionDetection** data set. We can identify that all the errors that **BlackBox** makes are on SAT properties, as it returns incorrect counterexamples.

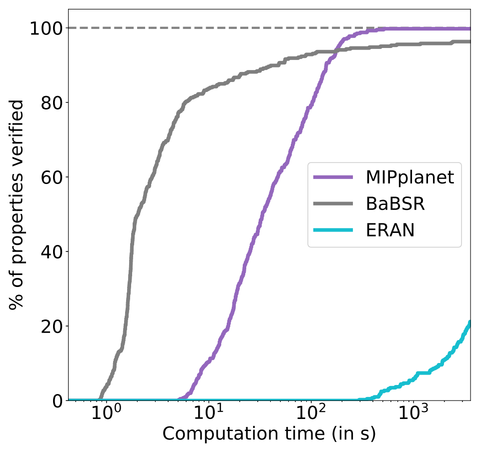


(a) On SAT properties

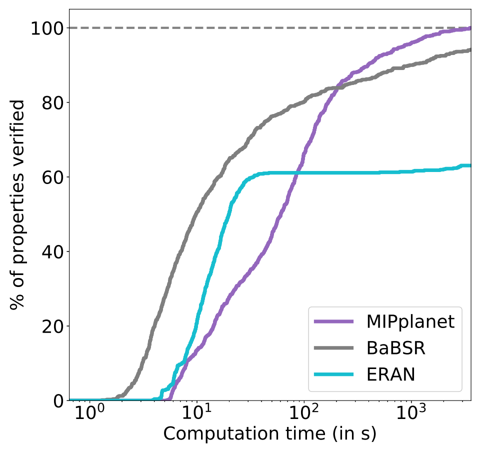


(b) On UNSAT properties

Figure 14: Proportion of properties verifiable by different methods under varying time budgets on the **ACAS** data set. We observe that **planet** doesn't succeed in solving any of the SAT properties, while our proposed methods are extremely efficient at it, even if there remains some properties that they can't solve.

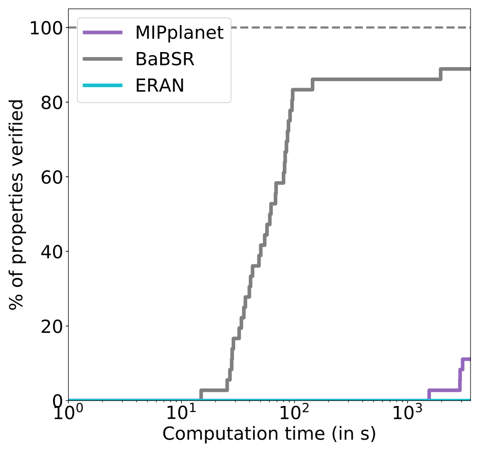


(a) On SAT properties

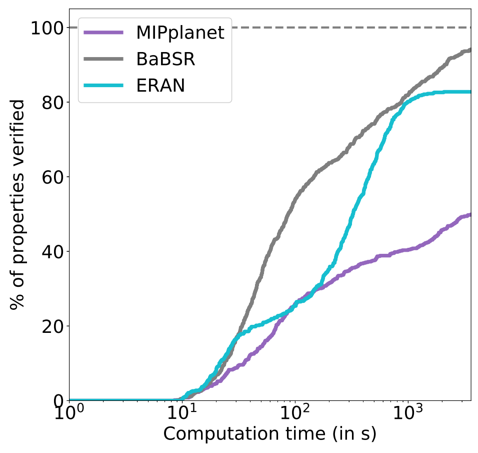


(b) On UNSAT properties

Figure 15: Proportion of properties verifiable by different methods under varying time budgets on the **reduced Robust network** data set. Similar performances can be observed on both SAT and UNSAT properties. In terms of the total number of properties solved, **MIPplanet** slightly outperforms **BaBSR** on challenging UNSAT problems. However, on simple problems, **BaBSR** are much more time efficient than **MIPplanet**.



(a) On SAT properties



(b) On UNSAT properties

Figure 16: Proportion of properties verifiable by different methods under varying time budgets on the **Robust Network** data set. **BaBSR** outperforms **MIPplanet** significantly in both cases. The huge performance gap on SAT properties indicates that Branch-and-Bound is an effective algorithm for finding counterexamples on large networks.

References

Michael Akintunde, Alessio Lomuscio, Lalit Maganti, and Edoardo Pirovano. Reachability analysis for neural agent-environment systems. *Sixteenth International Conference on Principles of Knowledge Representation and Reasoning*, 2018.

Ross Anderson, Joey Huchette, Christian Tjandraatmadja, and Juan Pablo Vielma. Strong mixed-integer programming formulations for trained neural networks. *Proceedings of IPCO 2019*, 2019.

Clark Barrett, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Splitting on demand in sat modulo theories. *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, 2006.

Osbert Bastani, Yani Ioannou, Leonidas Lampropoulos, Dimitrios Vytiniotis, Aditya Nori, and Antonio Criminisi. Measuring neural net robustness with constraints. *Conference on Neural Information Processing Systems*, 2016.

John N Buxton and Brian Randell. *Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee.* NATO Science Committee, 1970.

Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. *2017 IEEE Symposium on Security and Privacy (SP)*, 2017.

Chih-Hong Cheng, Georg Nührenberg, and Harald Ruess. Maximum resilience of artificial neural networks. *Automated Technology for Verification and Analysis*, 2017a.

Chih-Hong Cheng, Georg Nührenberg, and Harald Ruess. Verification of binarized neural networks. *arXiv:1710.03107*, 2017b.

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

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

Ruediger Ehlers. Planet. `https://github.com/progirep/planet`, 2017b.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. *AISTATS*, 2010.

Matthias Hein and Maksym Andriushchenko. Formal guarantees on the robustness of a classifier against adversarial manipulation. *Conference on Neural Information Processing Systems*, 2017.

Timothy Hickey, Qun Ju, and Maarten H Van Emden. Interval arithmetic: From principles to implementation. *Journal of the ACM (JACM)*, 2001.

Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety verification of deep neural networks. *International Conference on Computer Aided Verification*, 2017.

Guy Katz, Clark Barrett, David Dill, Kyle Julian, and Mykel Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks. *CAV*, 2017a.

Guy Katz, Clark Barrett, David Dill, Kyle Julian, and Mykel Kochenderfer. Reluplex. `https://github.com/guykatzz/ReluplexCav2017`, 2017b.

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.

Ailsa H. Land and Alison G. Doig. An automatic method for solving discrete programming problems. *Econometrica 28.3*, 1960.

Alessio Lomuscio and Lalit Maganti. An approach to reachability analysis for feed-forward relu neural networks. *arXiv:1706.07351*, 2017.

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

João P Marques-Silva and Karem A Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 1999.

D. R. Morrison, S. H. Jacobson, J. J. Sauppe, and E. C Sewell. Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning. *Discrete Optimization 19*, 2016.

Nina Narodytska, Shiva Prasad Kasiviswanathan, Leonid Ryzhyk, Mooly Sagiv, and Toby Walsh. Verifying properties of binarized deep neural networks. *arXiv:1709.06662*, 2017.

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

Hanif D Sherali and Warren P Adams. A hierarchy of relaxations and convex hull characterizations for mixed-integer zero—one programming problems. *Discrete Applied Mathematics*, 1994.

Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin Vechev. Fast and effective robustness certification. *Conference on Neural Information Processing Systems*, 2018.

Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. An abstract domain for certifiying neural networks. *Proc. ACM Program. Lang. 3*, 2019a.

Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. Boosting robustness cerfication of neural networks. *International Conference on Learning Representations*, 2019b.

Vincent Tjeng and Russ Tedrake. Verifying neural networks with mixed integer programming. *International Conference on Learning Representations*, 2019.

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)*, pages 1599–1614, 2018b.

Stefan Webb, Tom Rainforth, Yee Whye Teh, and M. Pawan Kumar. A statistical approach to assessing neural network robustness. *International Conference on Learning Representations*, 2019.

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.

Weiming Xiang, Hoang-Dung Tran, and Taylor T Johnson. Output reachable set estimation and verification for multi-layer neural networks. *arXiv:1708.03322*, 2017.

Radosiaw R Zakrzewski. Verification of a trained neural network accuracy. *IJCNN*, 2001.