Towards Scalable Complete Verification of ReLU Neural Networks via Dependency-based Branching

Panagiotis Kouvaros and Alessio Lomuscio
Department of Computing, Imperial College London, UK
{p.kouvaros, a.lomuscio}@imperial.ac.uk

Abstract

We introduce an efficient method for the complete verification of ReLU-based feed-forward neural networks. The method implements branching on the ReLU states on the basis of a notion of dependency between the nodes. This results in dividing the original verification problem into a set of sub-problems whose MILP formulations require fewer integrality constraints. We evaluate the method on all of the ReLU-based fully connected networks from the first competition for neural network verification. The experimental results obtained show 145% performance gains over the present state-of-the-art in complete verification.

1 Introduction

While the accuracy of neural network classifiers has greatly improved in the past few years, concerns about their fragility and opacity have been raised. This hinders the application of machine learning in a variety of applications, including safety-critical systems, where error rates need to be shown to be small before deployment. The area of verification of neural networks has grown rapidly in the past 3 years in response of these concerns and aims to provide methods to verify automatically that a neural network meets its intended specifications. For example, robustness against adversarial attacks [Szegedy *et al.*, 2014], or local robustness, i.e., the invariance of an image classifier against small image perturbations, is a property that is often studied, among others, in this area. Results from the verification step can be used in a process of certification to demonstrate that the classifier is robust; alternatively, if counterexamples are found these can be used to improve the model.

A key difficulty in the area is scalability. Simply put, the present methods, while effective on small models, cannot presently analyse the networks used in vision and other complex tasks. This is particularly evident in complete verification, where, in contrast to incomplete verification, a definite answer to the verification problem needs to be given. The main drawback of complete verification is the state space explosion problem whereby the ReLU space, i.e., the search space generated by the possible states of the ReLU nodes, grows exponentially in the number of ReLU nodes. This challenge is well recognised and constitutes the key objective for further research. This paper makes a contribution in this direction. In particular, we employ a notion of dependency between ReLU nodes as a heuristic to construct a MILP-based, branching approach that divides the original verification problem into a set of sub-problems whose ReLU space is smaller than the original one. In contrast to previous work in complete verification, the branching method that we devise directly aims at the reduction of the ReLU space. As we show in experiments on the ReLU-based fully-connected networks from the first competition for neural network verification (VNN-COMP) [VNN-COMP, 2020], this leads to 145% performance gains over the present state-of-the-art in complete verification.

The rest of the paper is organised as follows. After discussing related work in Section 2, in Section 3 we introduce the basic notions for the verification of neural networks that we use later in the paper. In Section 4 we present a dependency-based branching method for the MILP formulation of the verification problem. In Section 5 we report an experimental comparison of the method to the state-of-the-art in complete verification using the networks from VNN-COMP.

2 Related Work

Formal verification of neural networks comprises complete and incomplete methods. Complete methods can in principle return a definite answer as to whether the verification property is satisfied, whereas incomplete methods may be unable to decide whether the property is satisfied. Complete methods are based on MILP formulations [Botoeva *et al.*, 2020; Bastani *et al.*, 2016; Lomuscio and Maganti, 2017; Cheng *et al.*, 2017; Fischetti and Jo, 2018; Tjeng *et al.*, 2019], SMT encodings [Ehlers, 2017; Katz *et al.*, 2017; Katz *et al.*, 2019], and input refinement [Wang *et al.*, 2018; P. Henriksen, 2020]. Incomplete methods are based on duality [Dvijotham *et al.*, 2018; Wong and Kolter, 2018], linear approximations [Tran *et al.*, 2020; Singh *et al.*, 2019; Weng *et al.*, 2018; Tjandraatmadja *et al.*, 2020] and semi-definite relaxations [Fazlyab *et al.*, 2020; Dathathri *et al.*, 2020]. While incomplete approaches differ, they all rely on approximations of the ReLU function. This often improves their scalability over complete methods but can also hinder their efficacy to solve the verification problem.

The present contribution uses and extends ideas from

MILP encodings [Tjeng *et al.*, 2019; Anderson *et al.*, 2019; Botoeva *et al.*, 2020] towards conquering scalability in complete verification. This is achieved by developing a novel divide-and-conquer method that branches on the states of the ReLU nodes. Branching methods of this kind were previously considered. Reluplex [Katz *et al.*, 2017] implements a Simplex-type method which branches on a ReLU node whenever several pivot operations fail to resolve conflicts in the constraints representing the verification problem. In SAT- and MILP-based neural network verification [Ehlers, 2017; Cheng *et al.*, 2017; Botoeva *et al.*, 2020], branching is delegated to the underlying SAT and MILP solvers, though MILP-solvers are often instructed to prioritise branching for ReLU nodes in the early layers of the network. Branching is also considered in [Wang *et al.*, 2018; Rössig and Petkovic, 2020], where the ReLU node to branch is selected on the basis the nodes' output gradients so as to tighten the output range of the network. A common theme among these works is that they do not target the reduction of the ReLU space, which is the key aim of the present contribution.

To accomplish this, our method exploits the structure of the networks on the basis of a notion of dependency between the ReLU nodes. Whilst the dependency analysis methods are extended from [Botoeva *et al.*, 2020], the latter work delegates branching to the MILP solver, thereby offering no mechanism for selecting the ReLU node to branch, which is a key consideration of the present method. Also, whereas the efficacy of the cited method in reducing the ReLU space relies on the analysis of callback cuts by the underlying MILP solver, our method directly aims at the construction of simpler MILP encodings by "eliminating" some of the binary variables. As we experimentally show this is often more effective.

The dependency-based branching methods that we develop are also related to conflict analysis in MILP [Achterberg, 2007], which generalises SAT infeasibility analysis, and look-ahead based SAT solvers, which prioritise branching for variables that optimise a heuristic value [Biere *et al.*, 2009]. However, these approaches are domain independent with no specificity in neural network verification. In particular, as we experimentally show by comparing with the state-of-the-art MILP-based verification tools, generic methods do not fully exploit the highly structured nature of neural networks and suffer from markedly lower scalability due to this fact.

3 Background

The paper extends previous work in MILP formulations [Anderson *et al.*, 2019; Lomuscio and Maganti, 2017] and dependency analysis [Botoeva *et al.*, 2020] for the formal verification of neural networks. This section summarises basic concepts and fixes the notation used later in the paper.

Feed-forward ReLU networks. A *feed-forward neural network* (FFNN) is a vector-valued function $\mathbf{f}\colon \mathbb{R}^{s_0} \to \mathbb{R}^{s_L}$ that composes a sequence of $L \geq 1$ *layers*, $\mathbf{f}^{(1)}\colon \mathbb{R}^{s_0} \to \mathbb{R}^{s_1}, \ldots, \mathbf{f}^{(L)}\colon \mathbb{R}^{s_{L-1}} \to \mathbb{R}^{s_L}$. Each layer $\mathbf{f}^{(i)}$ is the composition of an affine transformation and a non-linear *activation* function. That is, $\mathbf{f}^{(i)}(\mathbf{x}^{(i-1)}) \triangleq \mathsf{act}^{(i)}(\mathbf{W}^{(i)}\mathbf{x}^{(i-1)} + \mathbf{b}^{(i)})$, where: $\mathbf{x}^{(0)}$ is the input to the network; $\mathbf{x}^{(i-1)}$ is the output of the $(i-1)$-th layer; $\mathsf{act}^{(i)}$ is the activation function of the i-th layer; $\mathbf{z}^{(i)} = \mathbf{W}^{(i)}\mathbf{x}^{(i-1)} + \mathbf{b}^{(i)}$ is the affine transformation, also called *pre-activation*, of the i-the layer for a weight matrix $\mathbf{W}^{(i)} \in \mathbb{R}^{s_i \times s_{i-1}}$ and a bias vector $\mathbf{b}^{(i)} \in \mathbb{R}^{s_i}$. A ReLU FFNN is a FFNN that contains only the Rectified Linear Unit (ReLU) activation function. The ReLU function is $\mathsf{ReLU}(\mathbf{z}^{(i)}) \triangleq \max(0, \mathbf{z}^{(i)})$ where the maximum function is applied element-wise on $\mathbf{z}^{(i)}$. Each j-th element of layer $\mathbf{f}^{(i)}$ is said to be the j-th *ReLU node*, or simply the j-th node, of the i-th layer; we sometimes write $n_j^{(i)}$ to refer to the node. A node $n_j^{(i)}$ is said to be in the *active state* if $\mathbf{z}_j^{(i)} \geq 0$ and in the *inactive state* if $\mathbf{z}_j^{(i)} < 0$.

Verification problem. Given a FFNN $\mathbf{f}$, a set of inputs $\mathcal{X} \subset \mathbb{R}^{s_0}$ and a set of outputs $\mathcal{Y} \subset \mathbb{R}^{s_L}$, the verification problem is to answer whether

$$\forall \mathbf{x} \in \mathcal{X}\colon \mathbf{f}(\mathbf{x}) \in \mathcal{Y}.$$

We write $(\mathbf{f}, \mathcal{X}, \mathcal{Y})$ to denote an instance of the verification problem. Typically, $\mathcal{X}$ and $\mathcal{Y}$ are finite sets of polyhedra [Anderson *et al.*, 2019; Wang *et al.*, 2018; Katz *et al.*, 2019; Tjeng *et al.*, 2019]. In particular, in this work $\mathcal{X}$ is a box given by lower and upper bounds on the inputs nodes, i.e., $\mathcal{X} = \left\{\mathbf{x}^{(0)} \mid \mathbf{l}_j^{(0)} \leq \mathbf{x}_j^{(0)} \leq \mathbf{u}_j^{(0)}, 1 \leq j \leq s_0, \mathbf{l}_j^{(0)} \leq \mathbf{u}_j^{(0)} \in \mathbb{R}\right\}$, and $\mathcal{Y}$ is a linear constraint on the network's outputs, i.e., $\mathcal{Y} = \left\{\mathbf{x}^{(L)} \mid \mathbf{c}^T\mathbf{x}^{(L)} + \mathbf{c}_0 > 0\right\}$. Among the various instantiations of the verification problem, the *local adversarial robustness problem* is one of the most well studied. The problem is to establish whether all images within a norm-ball of a given image are classified equivalently by the network. We here consider the l_∞ norm-ball. For a network $\mathbf{f}$, an image $\mathbf{x}$ with class label c and a perturbation (norm-ball) radius ϵ, the local adversarial robustness problem can be solved by solving a verification problem $(\mathbf{f}, \mathcal{X}, \mathcal{Y}_j)$ for each j with $1 \leq j \leq s_L$, $j \neq c$, $\mathcal{X} = \left\{\mathbf{x}^{(0)} \mid \mathbf{x} - \epsilon \leq \mathbf{x}^{(0)} \leq \mathbf{x} + \epsilon\right\}$ and $\mathcal{Y}_j = \left\{\mathbf{x}^{(L)} \mid \mathbf{x}_c^{(L)} - \mathbf{x}_j^{(L)} > 0\right\}$. The answer to the problem is positive iff every $(\mathbf{f}, \mathcal{X}, \mathcal{Y}_j)$ has a positive answer.

Bounds. Given a verification problem, we hereafter assume a lower bound $\mathbf{l}_j^{(i)}$ and an upper bound $\mathbf{u}_j^{(i)}$ for the pre-activation $\mathbf{z}_j^{(i)}$ of each node $n_j^{(i)}$. The bounds can be computed from $\mathcal{X}$ via bound propagation methods, see, e.g., [Singh *et al.*, 2019; Wang *et al.*, 2018; P. Henriksen, 2020]. A node $n_j^{(i)}$ is said to be *strictly active* if $\mathbf{l}_j^{(i)} \geq 0$ and *strictly inactive* if $\mathbf{u}_j^{(i)} \leq 0$. A *stable* node is a node that is either strictly active or strictly inactive. An *unstable* node is a node that is neither strictly active nor strictly inactive.

MILP formulation of the verification problem. The verification problem can be recast into a *mixed integer linear program* (MILP). A MILP is an optimisation problem whereby a linear objective function over real-valued and integer variables is sought to be minimised subject to a set of linear constraints on its variables, see, e.g., [Papadimitriou and Steiglitz, 1998]. The MILP formulation of a verification problem $(\mathbf{f}, \mathcal{X}, \mathcal{Y})$ is the following:

$$
\begin{aligned}
\min_{\mathbf{x}^{(0)},\ldots,\mathbf{x}^{(L)}} \quad & \mathbf{c}^T\mathbf{x}^{(L)} + \mathbf{c}_0 \\
\text{subject to} \quad & \mathbf{x}^{(0)} \in \mathcal{X}, && \text{(1a)}\\
& \mathbf{z}^{(i)} = \mathbf{W}^{(i)}\mathbf{x}^{(i-1)} + \mathbf{b}^{(i)}, && \text{(1b)}\\
& \mathbf{x}_j^{(i)} \geq \mathbf{z}_j^{(i)},\ \mathbf{x}_j^{(i)} \leq \mathbf{z}_j^{(i)} - \mathbf{l}_j^{(i)} \cdot (1-\delta_j^{(i)}), \\
& \mathbf{x}_j^{(i)} \leq \mathbf{u}_j^{(i)} \cdot \delta_j^{(i)},\ \mathbf{x}_j^{(i)} \geq 0,\ \delta_j^{(i)} \in \{0,1\}, && \text{(1c)}
\end{aligned}
$$

where $1 \leq i \leq L, 1 \leq j \leq s_i$. The constraints (1c) model the ReLU function. In particular, each $\delta_j^{(i)}$ is a binary variable such that $\delta_j^{(i)} = 0$ iff node $n_j^{(i)}$ is inactive and $\delta_j^{(i)} = 1$ iff node $n_j^{(i)}$ is active. We say that a MILP is satisfied if its optimal value is above zero. The verification problem has a positive answer iff its associated MILP is satisfied.

ReLU space. We focus on the branch-and-bound (B&B) method [Land and Doig, 2010] to solve the MILP (1). A key step of B&B is the recursive splitting of the feasible region, i.e., the set of admissible solutions, of a MILP into smaller regions by branching on the values of the binary variables until a feasible solution is found; see, e.g., [Morrison *et al.*, 2016] for a thorough description of B&B. The efficacy of B&B is therefore linked to the size of the search space generated by the binary variables. We refer to this space as the *ReLU space*; it is formally defined as the Cartesian product $\{0,1\}^{|\Delta|}$ of the ground terms of all the binary variables $\Delta \triangleq \left\{\delta_j^{(i)}\right\}_{i,j}$. The ReLU space induced by the MILP formulation (1) can be reduced by replacing the ReLU constraints for a node $n_j^{(i)}$ with $\mathbf{x}_j^{(i)} = \mathbf{z}_j^{(i)}$ if the node is strictly active and with $\mathbf{x}_j^{(i)} = 0$ if the node is strictly inactive [Tjeng *et al.*, 2019].

Dependency analysis. Dependency analysis [Botoeva *et al.*, 2020] is a method to further reduce the ReLU space that needs to be considered during B&B. In dependency analysis, the state s ($s \in \{active, inactive\}$) of a node $n_j^{(i)}$ *depends* on the state s' ($s' \in \{active, inactive\}$) of another node $n_r^{(q)}$ if whenever (i.e., for any network input in $\mathcal{X}$) the state of $n_r^{(q)}$ is s', the state of $n_j^{(i)}$ has to be s. The cited work identifies these dependencies during B&B, expresses them as MILP constraints and adds them to the MILP program being solved via callback cuts. Intuitively, each constraint determines the value of the binary variable associated with $n_j^{(i)}$ whenever $n_r^{(q)}$ is in state s' (as per its binary variable), thereby reducing the ReLU space.

4 Dependency-based Branching

This section introduces a dependency-based branching procedure for the MILP formulation of the verification problem defined above. The procedure recursively divides the verification problem into pairs of sub-problems. The two sub-problems are verification problems resulting from an originally unstable node being split and stabilised into the active and inactive state. Intuitively, the MILP formulations of the sub-problems are easier to solve as they contain fewer integrality constraints. Clearly, branching of this kind can be carried out exhaustively to generate sub-problems whose MILP formulations are linear programs that are generally easier to solve. However, the number of sub-problems grows exponentially in the number of unstable nodes, thereby rendering the exhaustive exploration of the ReLU space intractable for even small networks. In the light of this we develop a procedure that is parameterised on the *branching depth*, i.e., the depth of the tree obtained by recursively dividing the original problem. Once this depth is reached, all of the generated sub-problems are solved. Since the sub-problems are obtained via branching on the states of the ReLU nodes, the original problem is satisfied iff all of the sub-problems are satisfied. At the heart of the procedure is the selection of the ReLU node to branch. The key idea of this work is to exploit the dependency relations of the network to determine the ReLU node that will bring about the most significant reduction of the ReLU space. In particular, at each branching step, the ReLU node with the most nodes depended on it is selected for branching. Accounting for these dependencies, not only the ReLU node selected for branching is stabilised but also the nodes that depend on it. This leads to a linear encoding of the nodes, which in turn leads to a reduced ReLU space.

4.1 Branching Procedure

The overall verification method is outlined in Algorithm 1. The method relies on the branching procedure to divide the given verification problem into a set of sub-problems. It then encodes the sub-problems into MILPs which can be solved in parallel. Finally, it answers `yes` to the original verification problem iff all the MILPs are satisfied.

The branching procedure comprises three steps. In the first step, a *dependency graph* for the network in question is built. The graph expresses which node depends on which node. In the second step, the ReLU node that has the most depended nodes is identified and selected for branching. In the third step, the verification problem is divided into two sub-problems: one where the selected node is stabilised into the active state and one where it is stabilised into the inactive state. In each sub-problem, all the nodes that depend on the branching node are stabilised into the state prescribed by the corresponding dependency. The procedure recursively carries out these steps until the required branching depth is reached.

We now give a detailed description of the steps of the branching procedure. In our presentation, we will use the network from Figure 1 to exemplify each of the steps. We begin with the construction of the dependency graph. This is a directed graph whose vertices represent node-state pairs and whose edges represent dependencies between said pairs.

Definition 1 (Dependency graph)**.** *Given a verification problem* $(\mathbf{f}, \mathcal{X}, \mathcal{Y})$ *for which* $\mathbf{f}$ *comprises a set of unstable nodes* U*, the associated* dependency graph *of* $(\mathbf{f}, \mathcal{X}, \mathcal{Y})$ *is a directed graph* $\mathfrak{D} = (V, E)$*, where:*

- $V = U \times \{\mathbf{a}, \mathbf{i}\}$ *is the set of vertices, where* $\mathbf{a}$ *and* $\mathbf{i}$ *denote "active" and "inactive".*
- $E \subseteq V \times V$ *is the set of edges such that for every* $(n, s), (n', s') \in V$ *we have* $((n, s), (n', s')) \in E$ *iff*

Algorithm 1 The verification procedure.

```
1: procedure VERIFY((f, X, Y), bd)
2:     Input: verification problem (f, X, Y), branching depth bd
3:     Output: yes/no
4:     sub-problems ← branch((f, X, Y), bd)
5:     for P in sub-problems do
6:         milp ← encode(P)
7:         sub-result ← milp_solver(milp)
8:         if sub-result is not satisfied then return no
9:     return yes
10: procedure BRANCH((f, X, Y), bd)
11:     Input: verification problem (f, X, Y), branching depth bd
12:     Output: a set of verification sub-problems
13:     queue ← [(f, 0)], sub-problems ← []
14:     while queue is not empty do
15:         h, d ← pop top element of queue
16:         if d ≤ bd then
17:             D ← dependency_graph(h)
18:             n ← branching_node(D)
19:             h1(h2) ← stabilise h as per the active (inactive)
    dependency tree of n.
20:             add (h1, d + 1) ((h2, d + 1)) to queue
21:         else
22:             add (h, X, Y) to sub-problems
23:     return sub-problems
```

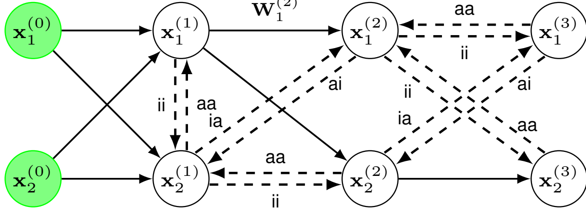

Figure 1: A ReLU FFNN. A dashed arrow from $\mathbf{x}_j^{(i)}$ to $\mathbf{x}_r^{(q)}$ with label ss' indicates that the s' state of node $n_r^{(q)}$ depends on the s state of node $n_j^{(i)}$.

the s'-state of node n' depends on the s-state of node n.

Figure 2 shows the dependency graph of the network in the running example.

The construction of the dependency graph relies on the method from [Botoeva *et al.*, 2020] to identify the dependencies between the nodes. The graph is built in time $\mathcal{O}(L \cdot S^2) \cdot S^2)$, where S is the maximum number of nodes in a layer, L is the number of layers, $\mathcal{O}(L \cdot S^2)$ is the bound on the number of edges in the graph, and $\mathcal{O}(S^2)$ is the cost of identifying a dependency [Botoeva *et al.*, 2020].

With the construction of the dependency graph, the ReLU node with the highest *dependency degree* is heuristically selected for branching. The dependency degree of a node expresses the total number of nodes that depend on it. Concretely, it equals the sum of the number of vertices in the *active* and *inactive dependency tree* of the node, where the active (inactive, respectively) dependency tree of the node is the spanning tree of the vertices reached during depth-first-search from the vertex associated with the active (inactive, respectively) state of the node in question. Formally, we have:

Definition 2 (Dependency degree)**.** *Given a dependency graph $\mathfrak{D}$, the dependency degree of a ReLU node n equals*

$$|\{(n', s') \mid (n', s') \textit{ is reachable from } (n, \mathbf{a}) \textit{ in } \mathfrak{D}\}| + |\{(n', s') \mid (n', s') \textit{ is reachable from } (n, \mathbf{i}) \textit{ in } \mathfrak{D}\}| .$$

Figure 2 highlights the active and inactive dependency trees of node $n_2^{(1)}$. From these we can compute the dependency degree of the node as 6.

The highest the dependency degree of the node selected for branching, the fewer the required binary variables to encode the resulting sub-problems.

Note that if the active (inactive, respectively) dependency tree of the selected node contains at least one pair of vertices of the form $(n, \mathbf{a})$ and $(n, \mathbf{i})$, then the sub-problem corresponding to the node being active (inactive, respectively) is discarded. This is because the MILP formulation of the sub-problem does not have any feasible solution.

The computation of the size of the dependency trees of the nodes takes time $\mathcal{O}(L^2 \cdot S^3)$ via DFS. So, accounting for the $\mathcal{O}((L \cdot S^2) \cdot S^2)$ cost of constructing the dependency graph, the identification of the node to split takes time $\mathcal{O}(L \cdot S^3 \cdot (L+S))$. Therefore, for a branching depth bd, the cost of the branching procedure is $\mathcal{O}(2^{bd} \cdot L \cdot S^3 \cdot (L + S))$.

The above concludes the description of the branching procedure. Next, we define the MILP encodings of the sub-problems generated by the procedure.

Definition 3 (MILP formulation)**.** *Given the dependency trees $\mathfrak{T}_1, \ldots, \mathfrak{T}_l$ of the ReLU nodes so far selected for branching, the MILP formulation of a sub-problem $(\mathbf{f}, \mathcal{X}, \mathcal{Y})$ is given as formulation* (1)*, but replacing the ReLU constraints* (1c) *of each node $n_j^{(i)}$ with either one of the following:*

(i) if either $\mathbf{l}_j^{(i)} \geq 0$ or $(n_j^{(i)}, \mathbf{a})$ is a vertex of some dependency tree, then $\mathbf{x}_j^{(i)} = \mathbf{z}_j^{(i)}$.

(ii) if either $\hat{\mathbf{u}}_{n_j^{(i)}}^{branch} \leq 0$ or $(n_j^{(i)}, \mathbf{i})$ is a non-root vertex of some dependency tree, then $\mathbf{x}_j^{(i)} = 0$.

(iii) if $(n_j^{(i)}, \mathbf{i})$ is the root of some dependency tree, then $\mathbf{z}_j^{(i)} \leq 0$, $\mathbf{x}_j^{(i)} = 0$.

(iv) if none of the above holds, then $\mathbf{x}_j^{(i)} \geq \mathbf{z}_j^{(i)}$, $\mathbf{x}_j^{(i)} \leq \mathbf{z}_j^{(i)} - \mathbf{l}_j^{(i)} \cdot (1 - \delta_j^{(i)})$, $\mathbf{x}_j^{(i)} \leq \mathbf{u}_j^{(i)} \cdot \delta_j^{(i)}$.

Clause (i) corresponds to the node being active, either because the lower bound of the node is above zero or because this is entailed by a dependency constraint. Analogously, clauses (ii) and (iii) correspond to the node being inactive. The two clauses distinguish between vertex $(n_j^{(i)}, \mathbf{i})$ being a root of a dependency tree and not. In the former case, the formulation includes the constraint $\mathbf{z}_j^{(i)} \leq 0$ so as to disallow for spurious solutions whereby $\mathbf{z}_j^{(i)} > 0$ and $\mathbf{x}_j^{(i)} = 0$. In

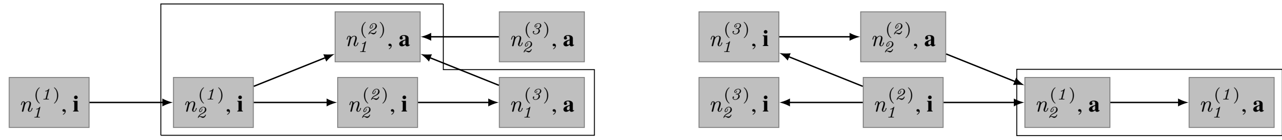

Figure 2: The dependency graph of the network from Figure 1. An edge from vertex (n, s) to vertex (n', s') represents that the s' state of node n' depends on the s state of node n. The rectangles highlight the active and inactive dependency trees of node $n_2^{(1)}$.

the latter case, said constraint is not required since $(n_j^{(i)}, \mathbf{i})$ is in the dependency tree of a node selected for branching; therefore, the pre-activation of $n_j^{(i)}$ is guaranteed to be below zero. Finally, clause (iv) corresponds to the standard MILP formulation for unstable nodes (see Section 3).

The above concludes the description of the verification procedure. The procedure is *sound*, i.e., it assesses a given network to be safe only when the network is safe. The procedure is also *complete*, i.e., it can in principle (given enough time) assess the safety of all networks that are safe.

Theorem 1 (Soundness and completeness)**.** *Given a verification problem* $(\mathbf{f}, \mathcal{X}, \mathcal{Y})$*, the procedure* $\texttt{VERIFY}((\mathbf{f}, \mathcal{X}, \mathcal{Y}))$ *from Algorithm 1 returns* yes *iff* $\forall \mathbf{x} \in \mathcal{X} \colon \mathbf{f}(\mathbf{x}) \in \mathcal{Y}$.

Proof. The proof is by straightforward induction on the branching depth. □

4.2 Optimisations

It is known that verifying neural networks is NP-complete [Katz *et al.*, 2017]. Constructing the dependency graph has only quadratic complexity. However, to improve the scalability of the approach, we devise two further optimisation methods.

Iterative branching. To reduce the overhead of the branching procedure, we consider a variant of the procedure where several splits are executed on the basis of the same dependency graph, as opposed to building a dependency graph for each split. Once there are no nodes suitable for branching (i.e., they have no dependencies with other nodes), the procedure re-computes the bounds for the nodes by taking into account the new bounds for the stabilised nodes. On the basis of these bounds it builds a new dependency graph which it uses to further divide the verification problem. While iterative branching does not guarantee that the ReLU node with the highest dependency degree is always selected for branching, it enhances the performance of the overall verification method as we experimentally show in Section 5.

Branch monitor. Whilst dependency-based branching improves the overall verification times (see Section 5), dependency-based branching for MILPs that are already easy to solve using B&B may be disadvantageous to solving the original problem without any (dependency-based) branching. A commonly used indicator of the hardness of MILP programs is the node throughput in B&B [Klotz and Newman, 2013]. In view of this, our verification procedure formulates the original verification problem as a MILP and uses the node throughput to determine whether to initiate branching or not. In particular, the procedure is parameterised with a threshold on the number of *B&B nodes*, i.e, the nodes in the B&B tree, to be explored by B&B before the branching procedure is initiated (from the root B&B node). The threshold takes high values for MILPs whose B&B nodes are easy to analyse and low values for MILPs whose B&B nodes are hard to analyse.

5 Evaluation

We evaluate the verification procedure introduced above and present a comparison with the state-of-the-art in complete methods. The procedure is implemented in `Venus2` [1], a Python toolkit that is based on `Venus` [Botoeva *et al.*, 2020]. The experimental comparisons focus on the leading and publicly available *complete* verification tools: `Eran` (deeppoly) [Singh *et al.*, 2019], `Marabou` [Katz *et al.*, 2019], `MIPVerify` [Tjeng *et al.*, 2019], `Neurify` [Wang *et al.*, 2018], `Nnenum` [S. Bak *et al.*, 2020], `Verinet` [P. Henriksen, 2020] and `Venus` [Botoeva *et al.*, 2020]. The comparisons are drawn on all of the benchmarks for fully connected ReLU FFNNs from VNN-COMP:

- *ACASXU* [Julian *et al.*, 2016] is a collection of 45 ReLU FFNNs which were developed as part of an airborne collision avoidance system to advise horizontal steering decisions for unmanned aircraft. Each network has 5 inputs, 300 ReLU nodes arranged in 6 layers with 50 neurons each and 5 outputs. We verify the networks against the safety specifications from [Katz *et al.*, 2017]. These include four properties that are checked on all of the 45 networks and 6 properties that are checked on a single network. We therefore consider a total of 186 verification problems.
- *MNIST* [LeCun *et al.*, 1998] is a dataset comprising images of hand-written digits 0-9, each formatted as 28x28x1-pixel grayscale image. We use three fully connected ReLU FFNNs trained on the dataset: MNIST2, MNIST4 and MNIST6. The networks comprise 2, 4 and 6 layers, respectively. Each layer of each of the networks has 256 ReLU nodes. We verify the networks against the local adversarial robustness property w.r.t 25 correctly classified images and perturbation radii of 0.02 and 0.05. We therefore consider a total of 150 verification problems.

All the experiments were carried out on an Intel Core i7-7700K (4 cores) equipped with 16GB RAM, running Linux kernel 4.15; the ablation experiment on the branching depth (see below) was carried out on an Intel Core i9-9900X (20 cores) equipped with 125GB RAM, running Linux kernel 5.3.

[1] `Venus2` is available at https://github.com/vas-group-imperial/venus2.

Model	Radius	Venus2		Eran		Marabou		MIPVerify		Neurify		Nnenum		Verinet		Venus	
		ver	t	ver	t	ver	t	ver	t	ver	t	ver	t	ver	t	ver	t
ACASXU	-	186	11.3	76	1065.0	149	497.9	125	807.1	179	98.9	186	1.8	-	-	184	123.3
MNIST2	0.02	25	0.3	25	0.5	24	151.0	25	0.3	23	283.9	25	1.7	25	50.1	25	0.6
	0.05	25	2.8	24	109.3	7	1430.8	25	2.9	20	395.2	25	19.1	22	216.0	25	5.5
MNIST4	0.02	25	4.3	25	18.6	21	325.3	25	20.9	23	213.7	24	96.5	20	360.1	24	83.6
	0.05	25	383.12	22	1688.0	0	1800.0	6	1407.0	1	1728.0	6	1267.8	15	1080.2	3	1603.1
MNIST6	0.02	25	120.9	18	504.4	7	325.3	19	362.5	18	512.7	21	293.0	16	648.1	9	1112.2
	0.05	25	605.7	0	1800.0	0	1800.0	0	1800.0	1	1728.0	6	1270.9	8	1237.9	3	1595.0
ALL		336	96.6	171	896.1	218	727.6	229	581.5	265	417.3	293	237.0	101	597.2	275	376.0

Table 1: Verification results for Venus2 and the state-of-the-art complete tools. The ver columns report the number of verification problems that each tool was able to solve and the t columns give the average time in seconds taken by each tool. Highlighted cells indicate the best performing tool for each of the benchmarks. Verinet was not benchmarked on ACASXU as it does not support the ACASXU specifications.

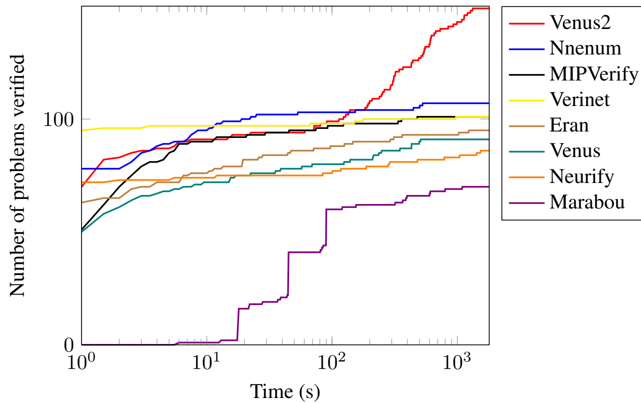

Figure 3: Number of verification problems solved as a function of time for the MNIST benchmarks.

Each verification problem was run for a timeout of 30 minutes. All MILP-based tools (Venus2, Eran, MIPVerify, Venus) rely on Gurobi 9.2 for the MILP backend. Venus2 was run with the branching depth set to 4 for MNIST2 and MNIST4 and to 7 for ACASXU and MNIST6. The B&B node threshold was set to 500 for ACASXU and MNIST6, to 5000 for MNIST4 and to 10000 for MNIST2.

Table 1 reports the experimental results [2]. Venus2 outperformed all of the tools by verifying more problems and by being faster on average. In particular, Venus2 solved 13% more problems and had a 145% faster average solve time than the second best performing tool Nnenum.

Venus2 was only outperformed on the ACASXU benchmark only by Nnenum. Also, as can be observed from Figure 3, Nnenum and Verinet were able to solve more MNIST verification problems under 100 seconds than Venus2 was able to solve within the same time. The latter two observations indicate an overhead in the construction of the dependency graphs and the MILP encodings for verification problems that the current state-of-the-art can solve in under 100 seconds. For harder problems Venus2 consistently outperforms all of the tools. This is particularly evident in MNIST4 and MNIST6 and the more challenging perturbation radius of 0.05. For these benchmarks significant branching on the ReLU nodes is required to solve the verification problems, as opposed to the 0.02 perturbation radius where the tighter bounds for the nodes can often be used to solve the verification problem without any branching.

We now proceed to evaluate the parameters and optimisations of Venus2 by documenting the variability of the tool's performance under different operational settings.

We begin with benchmarking Venus2 with iterative branching turned off. Table 2 reports the results obtained on the MNIST models and the 0.05 perturbation radius. Whilst iterative branching does not always select the ReLU node with the highest dependency degree for branching, the results show that the gains obtained from reducing the number of dependency graphs constructed outweigh the occasional non-optimal selection of the node to branch.

Still, the dependency-based branching heuristic is crucial to Venus2's performance: as Table 2 shows, the heuristic is consistently superior to random branching, e.g., it is 82.49% faster for the MNIST6 model and the 0.05 radius.

Also, whilst parallelisation definitely contributes to Venus2's performance, the simplified MILP encodings from dependency-based branching is an additional contributing factor. In particular, even though Venus2's performance degrades to half its speed when run on a single thread (see Table 2), it still outperforms the baseline tool Venus when run on multiple threads by verifying markedly more images.

Finally, we evaluate the sensitivity of Venus2 to the B&B node threshold and the branching depth parameters. For the study of the former, we use the MNIST6 model and the 0.02 radius since the benchmark contains a mixture of easy and hard verification problems. For the study of the latter, we use the MNIST4 model and the 0.05 radius since the benchmark mainly comprises hard problems for which Venus2 initiates dependency-based branching.

Figure 4 summarises the results. We observe that Venus2

[2] Note that the comparative performance of the tools is not always in agreement with the one reported in VNN-COMP. This is because the participants in the competition used different machines with different specifications for the execution of the experiments.

Ablation test	MNIST2 (0.05)				MNIST4 (0.05)				MNIST6 (0.05)			
	#branch	ver	t	↑	#branch	ver	t	↑	#branch	ver	t	↑
Iterative Branching Off	2	25	6.63	136.78 %	23	24	471.52	23.07%	24	17	1096.9	81.09 %
Random Branching		25	4.27	52.49%		23	712.08	48%		18	1105.38	82.49%
Single-threaded Execution		25	6.04	115.71%		24	501.32	4.40%		21	1154.35	90.58%

Table 2: Ablation experiments for iterative branching turned off instead of on, random selection of the branching node instead of dependency-based selection, and single-threaded execution instead of multi-threaded execution. The *#branch* columns report the number of verification problems for which dependency-based branching was initiated, the *ver* columns indicate the number of verification problems solved for each model, the t columns give the average time in seconds taken for each model, and the ↑ columns show the increase in time from the corresponding results in Table 1.

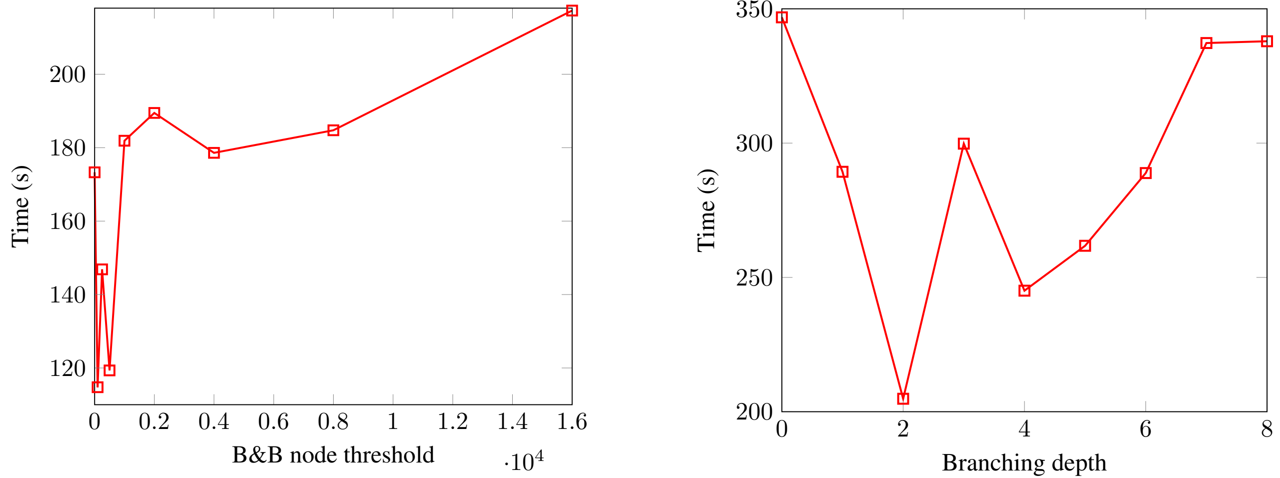

Figure 4: Sensitivity of `Venus2` to the B&B node threshold and the branching depth parameters. Left: average runtime of `Venus2` on MNIST6 and the 0.02 perturbation radius as a function of the B&B node threshold. Right: average runtime of `Venus2` on MNIST4 and the 0.05 perturbation radius as a function of the branching depth.

performs best for intermediate values of the B&B node threshold. Intuitively, the branch monitor aims at initiating dependency-based branching only for problems that are hard to solve via B&B. Under this light, low values for the parameter are problematic as they induce the overhead of dependency-based branching for problems that can easily be solved via B&B. Analogously, high values are also problematic as they delay the execution of dependency-based branching for problems that are harder to solve.

We similarly observe that `Venus2` performs best for intermediate values of the branching depth parameter. Intuitively, intermediate values for the parameter are more effective as dependency-based branching is executed w.r.t ReLU nodes with high dependency degrees, as opposed to lower values which debilitate the effective execution of dependency-based branching and larger values which encourage branching on nodes with lower dependency degrees.

6 Conclusions

In this paper we introduced a novel verification method for ReLU-based neural networks. The key idea upon which the method is developed is that branching on nodes whose stabilisation forces the stabilisation of other nodes enables the analysis of the verification problem under stronger MILP formulations. Doing this requires the analysis of the dependency relations of the ReLU nodes, so that the nodes whose splitting induces the greatest reduction of the ReLU space can be heuristically determined. The experimental results showed 145% performance gains for the fully connected networks from the first competition for neural network verification, thereby indicating a step towards more scalable complete neural network verification.

Acknowledgements

This work is partly funded by DARPA under the Assured Autonomy programme (FA8750-18-C-0095). A. Lomuscio is supported by a Royal Academy of Engineering Chair in Emerging Technologies.

References

[Achterberg, 2007] T. Achterberg. Conflict analysis in mixed integer programming. *Discrete Optimization*, 4(1):4–20, 2007.

[Anderson *et al.*, 2019] R. Anderson, J. Huchette, C. Tjandraatmadja, and J. Vielma. Strong mixed-integer programming formulations for trained neural networks. In *IPCO19*, pages 27–42, 2019.

[Bastani *et al.*, 2016] O. Bastani, Y. Ioannou, L. Lampropoulos, D. Vytiniotis, A. V. Nori, and A. Criminisi. Measuring neural net robustness with constraints. In *NeurIPS16*, pages 2613–2621, 2016.

[Biere *et al.*, 2009] A. Biere, M Heule, and H. Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[Botoeva *et al.*, 2020] E. Botoeva, P. Kouvaros, J. Kronqvist, A. Lomuscio, and R. Misener. Efficient verification of neural networks via dependency analysis. In *AAAI20*, 2020.

[Cheng *et al.*, 2017] C. Cheng, G. Nührenberg, and H. Ruess. Maximum resilience of artificial neural networks. In *ATVA17*, pages 251–268. Springer, 2017.

[Dathathri *et al.*, 2020] S. Dathathri, K. Dvijotham, A. Kurakin, A. Raghunathan, J. Uesato, R. Bunel, S. Shankar, J. Steinhardt, I. Goodfellow, P. Liang, and P. Kohli. Enabling certification of verification-agnostic networks via memory-efficient semidefinite programming. In *NeurIPS20*, 2020.

[Dvijotham *et al.*, 2018] K. Dvijotham, R. Stanforth, S. Gowal, T. Mann, and P. Kohli. A dual approach to scalable verification of deep networks. In *UAI*, volume 1, page 2, 2018.

[Ehlers, 2017] R. Ehlers. Formal verification of piece-wise linear feed-forward neural networks. In *ATVA17*, volume 10482, pages 269–286. Springer, 2017.

[Fazlyab *et al.*, 2020] M. Fazlyab, M. Morari, and G. J. Pappas. Safety verification and robustness analysis of neural networks via quadratic constraints and semidefinite programming. *IEEE Transactions on Automatic Control*, 2020.

[Fischetti and Jo, 2018] M. Fischetti and J. Jo. Deep neural networks and mixed integer linear optimization. *Constraints*, pages 1–14, 2018.

[Julian *et al.*, 2016] K. Julian, J. Lopez, J. Brush, M. Owen, and M. Kochenderfer. Policy compression for aircraft collision avoidance systems. In *DASC16*, pages 1–10, 2016.

[Katz *et al.*, 2017] G. Katz, C. W. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In *CAV17*, pages 97–117, 2017.

[Katz *et al.*, 2019] G. Katz, D. A. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljic, D. L. Dill, M. J. Kochenderfer, and C. W. Barrett. The marabou framework for verification and analysis of deep neural networks. In *CAV19*, pages 443–452, 2019.

[Klotz and Newman, 2013] E. Klotz and A. Newman. Practical guidelines for solving difficult mixed integer linear programs. *Surveys in Operations Research and Management Science*, 18(1-2):18–32, 2013.

[Land and Doig, 2010] A. Land and A. Doig. An automatic method for solving discrete programming problems. In *50 Years of Integer Programming 1958-2008*, pages 105–132. Springer, 2010.

[LeCun *et al.*, 1998] Y. LeCun, C. Cortes, and C. J. Burges. The mnist database of handwritten digits, 1998.

[Lomuscio and Maganti, 2017] A. Lomuscio and L. Maganti. An approach to reachability analysis for feed-forward relu neural networks. *CoRR*, abs/1706.07351, 2017.

[Morrison *et al.*, 2016] D. Morrison, S. Jacobson, J. Sauppe, and E. Sewell. Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning. *Discrete Optimization*, 19:79–102, 2016.

[P. Henriksen, 2020] A. Lomuscio P. Henriksen. Efficient neural network verification via adaptive refinement and adversarial search. In *ECAI20*, 2020.

[Papadimitriou and Steiglitz, 1998] C. Papadimitriou and K. Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998.

[Rössig and Petkovic, 2020] A. Rössig and M. Petkovic. Advances in verification of relu neural networks. *Journal of Global Optimization*, pages 1–44, 2020.

[S. Bak *et al.*, 2020] H. Tran S. Bak, K. Hobbs, and T. Johnson. Improved geometric path enumeration for verifying relu neural networks. In *CAV20*, pages 66–96, 2020.

[Singh *et al.*, 2019] G. Singh, T. Gehr, M. Püschel, and P. Vechev. An abstract domain for certifying neural networks. *POPL19*, 3(POPL):1–30, 2019.

[Szegedy *et al.*, 2014] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. In *ICLR14*, 2014.

[Tjandraatmadja *et al.*, 2020] C. Tjandraatmadja, R. Anderson, J. Huchette, W. Ma, K. Patel, and J. Vielma. The convex relaxation barrier, revisited: Tightened single-neuron relaxations for neural network verification. In *NeurIPS20*, 2020.

[Tjeng *et al.*, 2019] V. Tjeng, K. Y. Xiao, and R. Tedrake. Evaluating robustness of neural networks with mixed integer programming. In *ICLR19*, 2019.

[Tran *et al.*, 2020] H. Tran, S. Bak, W. Xiang, and T. Johnson. Verification of deep convolutional neural networks using imagestars. In *International Conference on Computer Aided Verification*, pages 18–42. Springer, 2020.

[VNN-COMP, 2020] VNN-COMP. Verification of neural networks competition (vnn-comp20). https://sites.google.com/view/vnn20/vnncomp, 2020. Accessed: 2021-05-29.

[Wang *et al.*, 2018] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Efficient formal safety analysis of neural networks. In *NeurIPS18*, pages 6369–6379, 2018.

[Weng *et al.*, 2018] L. Weng, G. Zhang, H. Chen, Z. Song, C. Hsieh, L. Daniel, D. Boning, and I. Dhillon. Towards fast computation of certified robustness for relu networks. In *ICML18*, pages 5276–5285, 2018.

[Wong and Kolter, 2018] E. Wong and J. Kolter. Provable defenses against adversarial examples via the convex outer adversarial polytope. In *ICML18*, pages 5286–5295, 2018.