Efficient Verification of ReLU-Based Neural Networks via Dependency Analysis

Elena Botoeva, Panagiotis Kouvaros, Jan Kronqvist, Alessio Lomuscio, Ruth Misener
Department of Computing, Imperial College London, UK
{e.botoeva, p.kouvaros, j.kronqvist, a.lomuscio, r.misener}@imperial.ac.uk

Abstract

We introduce an efficient method for the verification of ReLU-based feed-forward neural networks. We derive an automated procedure that exploits dependency relations between the ReLU nodes, thereby pruning the search tree that needs to be considered by MILP-based formulations of the verification problem. We augment the resulting algorithm with methods for input domain splitting and symbolic interval propagation. We present Venus, the resulting verification toolkit, and evaluate it on the ACAS collision avoidance networks and models trained on the MNIST and CIFAR-10 datasets. The experimental results obtained indicate considerable gains over the present state-of-the-art tools.

1 Introduction

Artificial Intelligence (AI) methods are increasingly used in safety critical applications including, but not limited to, autonomous vehicles, avionics, and power generation. These domains typically require a certification aimed at establishing the safety of the application to be deployed.

Formal verification methods commonly used in software verification cannot address the validation of AI applications due to the inherently different components. In particular, AI applications increasingly utilise neural networks in key parts of their designs, most notably in perception and control modules. Due to this, the area of formal verification of neural networks has recently received considerable attention. Simply put, methods for assessing neural systems can provide the mathematical underpinning for safely deploying a wide number of AI applications.

The typical decision problem tackled by verification approaches is whether a neural network, or a closed-loop system in which neural networks are present, can output particular values, i.e., output reachability. Reachability is often studied in conjunction with local robustness properties, i.e., whether for a given input, e.g., an image, small alterations of this input can cause output variation, e.g., a different classification. The present state of the art (Liu et al. 2019) includes several ways of formulating this problem (see related work below); however, no method scales to the analysis of the neural networks presently used in industrial strength applications, including autonomous vehicles. Therefore, it remains of considerable importance to develop more scalable approaches. This is the aim of the present contribution.

This paper introduces a novel, MILP-based approach to verifying feed-forward ReLU-based neural networks. Rectified Linear Units (ReLUs) are the most commonly-used activation functions in vision and are the typical object of study in the above cited literature. This manuscript develops the concept of *dependency*. Two nodes in a neural network are in a dependency relation if there is a strict connection between their active or inactive state during the overall network computation. As we show, dependency can be exploited to improve the performance of a MILP formulation. Crucially, and differently from the related state-of-the-art, our dependency analysis is not aimed at reducing the number of variables in the verification problem, but rather at reducing the search space during a branch-and-bound approach to generate satisfiable assignments. This paper (i) develops effective methods exploiting these dependencies and (ii) integrates the methods into a larger implementation incorporating scalability-improving methods such as domain splitting. The resulting implementation generates speed-ups of at least one order of magnitude against competing methods.

The rest of the paper is organised as follows. After discussing related work below, Section 2 reports key concepts on neural networks and related verification approaches. Section 3 first presents the theoretical contribution on dependency analysis and then gives a dependencies-based verification algorithm. Section 4 presents a toolkit exploiting dependency analysis. Section 5 reports the experimental results obtained on the MNIST, CIFAR-10, and ACAS datasets and compares these against state-of-the-art implementations.

Related Work. Verification methods for neural networks can be partitioned into complete and incomplete ones. Complete methods can in principle return a definite answer as to whether the property in question is satisfied. Differently, incomplete methods may erroneously conclude that the network is not robust when it actually is or a certain output is reachable when it is actually not.

Incomplete methods include approaches based on duality (Dvijotham et al. 2018), abstract interpretation (Gehr et al. 2018), symbolic interval analysis (Zhang et al. 2018; Wang et al. 2018a) and semidefinite relaxations (Raghunathan, Steinhardt, and Liang 2018; Fazlyab, Morari, and Pappas 2019). While the techniques differ, they all overestimate the output of the network from a given input region in an attempt to draw a conclusion from this. While incomplete methods may be very efficient in some cases, they are not comparable to the ones here presented as they may not answer the verification problem due to false negatives.

Complete approaches can be divided into 3 main groups: (i) MILP-based (Bastani et al. 2016; Lomuscio and Maganti 2017; Cheng, Nührenberg, and Ruess 2017; Fischetti and Jo 2018; Bunel et al. 2018; Tjeng, Xiao, and Tedrake 2019) techniques that formulate the verification problem at hand as a mixed integer linear program; (ii) SMT-based (Ehlers 2017; Katz et al. 2017; 2019) techniques that encode the verification problem as the satisfiability modulo theory problem; (iii) techniques that use a combination of overestimation and refinement techniques to get a definite answer (Wang et al. 2018b; 2018a).

Closely related to this paper is some of the recent work, which has focused on conquering scalability and increasing precision in incomplete approaches. These include: (i) computing tight bounds using symbolic interval analysis (Wang et al. 2018a; Zhang et al. 2018); (ii) input splitting (Wang et al. 2018b; Katz et al. 2019; Rubies-Royo et al. 2019); (iii) optimised MILP formulations (Bunel et al. 2018; Tjeng, Xiao, and Tedrake 2019; Anderson et al. 2019). The work here presented uses MILP formulations to the verification problem combined with input splitting and symbolic interval analysis methods. However, differently from the work cited above, it uses novel heuristics based on dependency analysis to guide the search for feasible solutions.

2 Background

This section summarises and fixes the notation on some of the key notions used later in the paper.

Feed-forward neural networks. A feed-forward neural network (FFNN) is a directed acyclic graph whose nodes are structured in layers. The first layer is the *input layer*, also referred to as layer 0, the last layer is the *output layer*, also referred to as layer k, and every layer in-between is a *hidden layer*, also referred to as layer i, for $1 \leq i < k$. Every node other than an input node is connected to every node in the preceding layer. Each edge is associated with a weight, which is learned during the training phase. Given an input vector, the network computes a function by propagating the input through the network, where, at each step, a node's *output* results from applying an activation function to the *pre-activation* of the node, which is the weighted sum of the outputs of the nodes from the previous layer. Here, we only consider the *ReLU* activation function defined by $\mathsf{ReLU}(x) \triangleq \max(x, 0)$ for $x \in \mathbb{R}$.

We denote by s_i the number of nodes in layer i. We use $n_{i,q}$ to refer to the q-th node of layer i. Given an input $\mathbf{x}$ to the network, we write $\hat{\mathbf{x}}_{i,q}$ ($\mathbf{x}_{i,q}$, respectively) for the pre-activation (output, respectively) of the node $n_{i,q}$. For a set of inputs over a bounded domain, every node is associated with lower and upper pre-activation and activation bounds. These can be derived in a number of ways discussed below. We write $\hat{\mathbf{l}}_{i,q}$ and $\hat{\mathbf{u}}_{i,q}$ ($\mathbf{l}_{i,q}$ and $\mathbf{u}_{i,q}$, respectively) for the pre-activation's (output's, respectively) lower and upper bounds. Similarly, $\hat{\mathbf{x}}_i$ and $\mathbf{x}_i$ refer to the vector of pre-activations and outputs of layer i over domains $[\hat{\mathbf{l}}_i, \hat{\mathbf{u}}_i]$ and $[\mathbf{l}_i, \mathbf{u}_i]$, respectively, where $\mathbf{x}_0 = \mathbf{x}$, and $\mathbf{l}_0$ and $\mathbf{u}_0$ are the input lower and upper bounds. We write W_i and b_i to refer to the weight matrix and bias vector of layer i, $i \geq 1$, respectively.

Given an input $\mathbf{x}$ and for $i \geq 1$, the output $\mathbf{x}_i$ of layer i is computed from $\mathbf{x}_{i-1}$ by applying the function $f_i \colon \mathbb{R}^{s_{i-1}} \to \mathbb{R}^{s_i}$, which is defined as $f_i(\mathbf{x}_{i-1}) \triangleq \mathsf{ReLU}(W_i \mathbf{x}_{i-1} + b_i) = \mathsf{ReLU}(\hat{\mathbf{x}}_i)$, where the ReLU function is applied element-wise. Given the above, a neural network of $k+1$ layers, is defined as a function $f \colon \mathbb{R}^{s_0} \to \mathbb{R}^{s_k}$, corresponding to the composition of the functions f_i computed by each layer i, i.e., $f(\mathbf{x}) \triangleq f_k(\ldots f_1(\mathbf{x}) \ldots)$.

A ReLU node $n_{i,q}$ can be in one of two states. It is in the *strictly active* state (or, is strictly active), denoted $\mathsf{st}(n_{i,q}) = \top$ if $\hat{\mathbf{l}}_{i,q} \geq 0$. It is in the *strictly inactive* state (or, is strictly inactive), denoted $\mathsf{st}(n_{i,q}) = \bot$, if $\hat{\mathbf{u}}_{i,q} \leq 0$. A *stable* node is a node that is either strictly active or strictly inactive. Otherwise, the node is said to be *unstable*, denoted $\mathsf{st}(n_{i,q}) = ?$.

Verification problem. Given a network $f \colon \mathbb{R}^{s_0} \to \mathbb{R}^{s_k}$ and a specification $(\mathcal{X}_0, \mathcal{X}_k) \subseteq \mathbb{R}^{s_0} \times \mathbb{R}^{s_k}$, the verification problem determines whether $\forall \mathbf{x}_0 \in \mathcal{X}_0 \colon \mathbf{x}_k \in \mathcal{X}_k$.

To enable the MILP representation, we hereafter assume that $\mathcal{X}_0$ is an intersection of finite sets of polyhedra. The *local robustness* and *reachability* problems are instantiations of the verification problem. The local robustness problem establishes if the network's output is unaffected by small perturbations of a given input $\mathbf{x}'$. In the case of image classifiers, local robustness checks if all images within a norm-ball of $\mathbf{x}'$ are classified equivalently. The problem can be represented by setting $\mathcal{X}_0 = \{\mathbf{x} \in \mathbb{R}^{s_0} \mid \|\mathbf{x} - \mathbf{x}'\|_p \leq \epsilon\}$, for some $\epsilon \geq 0$ and norm p, and $\mathcal{X}_k = \{\mathbf{x}_k \in \mathbb{R}^{s_k} \mid \forall i \neq c \colon (\mathbf{x}_k)_i < (\mathbf{x}_k)_c\}$, where $(\mathbf{x}_k)_j$ is the j-th component of $\mathbf{x}_k$ and c is the class of $\mathbf{x}'$.

The reachability problem establishes if there exists an admissible input in a given set $\mathcal{I}$ for which the network computes a given output $\mathbf{y}$. The reachability problem is not directly expressible as the verification problem defined above as it includes an existential quantification over the inputs. The dual problem can, however, be represented by taking $\mathcal{X}_0 = \mathcal{I}$ and $\mathcal{X}_k = \mathbb{R}^{s_k} \setminus \{\mathbf{y}\}$. Therefore, the answer to the reachability problem is the complement of the answer to the verification problem encoded as the dual above.

MILP formulation. The verification problem admits a *precise* representation as a Mixed Integer Linear Program (MILP) by means of the "big-M" encoding (Akintunde et al. 2018). Specifically, the corresponding MILP program is feasible iff the answer to the verification problem is *no*. Assuming the pre-activation bounds of the nodes have already been calculated (see below), the MILP encoding of a node $n_{i,q}$ depends on its state. If the node is strictly active, then it can be encoded by $\mathbf{x}_{i,q} = \hat{\mathbf{x}}_{i,q}$. If the node is strictly inactive, then it can be encoded by $\mathbf{x}_{i,q} = 0$. Otherwise, the encoding of the node is given by:

$$\begin{aligned} \mathbf{x}_{i,q} &\geq 0, & \mathbf{x}_{i,q} &\geq \hat{\mathbf{x}}_{i,q}, \\ \mathbf{x}_{i,q} &\leq \hat{\mathbf{u}}_{i,q} \cdot \delta_{i,q}, & \mathbf{x}_{i,q} &\leq \hat{\mathbf{x}}_{i,q} - \hat{\mathbf{l}}_{i,q} \cdot (1 - \delta_{i,q}), \end{aligned}$$

where $\delta_{i,q}$ is a binary variable such that $\delta_{i,q} = 0$ iff $\mathbf{x}_{i,q} = 0$ and $\delta_{i,q} = 1$ iff $\mathbf{x}_{i,j} = \hat{\mathbf{x}}_{i,q}$.

For an MILP program comprising a set Δ of binary variables, a (partial) *configuration* is a (partial) function $h : \Delta \rightarrow \{0, 1\}$ that assigns to each variable (some of the variables) a value from $\{0, 1\}$. The set of all possible partial configurations is said to be the program's *configuration space*.

A leading approach for solving MILP programs is the branch-and-bound method. In branch-and-bound, each integrality constraint $\delta_{i,j} \in \{0, 1\}$ is relaxed to a linear constraint $\delta_{i,j} \in [0, 1]$, thereby defining a linear program which can be solved in polynomial-time (Karmarkar 1984). Integrality is iteratively enforced by dividing the search domain into sub-regions excluding fractional solutions. In the context of neural networks, the efficacy of branch-and-bound depends on (i) the number of binary variables, i.e., the number of unstable nodes, and (ii) the tightness of the linear relaxations, i.e., the tightness of the pre-activation bounds.

Calculating bounds. Interval arithmetic derives pre-activation bounds by propagating the interval of the input domain through the network. However, the resulting bounds are often over-approximated as the method neglects dependencies between the input nodes, and propagating the over-approximated bounds leads to larger over-approximations following each layer. To enable tighter approximations, rather than propagating concrete intervals, approaches based on symbolic interval analysis (Wang et al. 2018a) define linear equations for the lower and upper bounds which are built from variables expressing the inputs of the network. To tackle the non-linearity of the ReLU function, propagating the equations involves their linear relaxation (Wang et al. 2018a). Lastly, even tighter bounds can be obtained by splitting the input domains into several sub-domains and solving the verification problem for each sub-domain (Wang et al. 2018b; 2018a; Katz et al. 2019).

3 Dependency Analysis

As discussed in the previous section, a major impediment to the scalable verification of ReLU-based FFNNs is the configuration space generated by the piecewise linearity of the ReLU nodes. Several approaches have been put forward for reducing the number of non-linearities that need to be considered for solving the verification problem. In particular, techniques that split the input domain have been shown effective in stabilising the ReLU nodes, thereby generating easier verification problems whose solutions can be combined to decide the original problem in a more efficient manner. Still, since the number of splits that need to be carried out grows exponentially in the number of splits, networks with high input dimensionality remain hard to tackle. To overcome this, we introduce a technique that exploits what we define below as the *network's dependency relation* to reduce the configuration space that needs to be considered in solving a verification problem. Informally, the network's dependency relation can be used to stabilise a ReLU node on the basis of an assumed stable state of another node. Formally it is defined as follows.

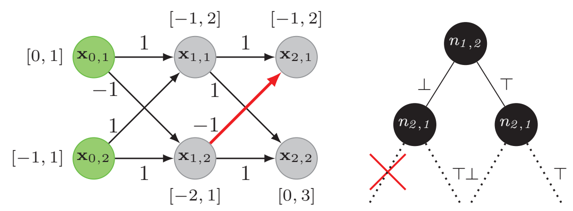

Figure 1: **Left:** Feedforward neural network exhibiting the dependency "if $n_{1,2}$ is inactive, then $n_{2,1}$ is inactive". **Right:** Depiction of the configuration space reduction induced by the dependency.

Definition 1 (Dependency relation)**.** *Given a neural network f that comprises a set of unstable nodes U, the* dependency relation for U, $\mathcal{D}_f \subseteq U \times U$ *is the set of all pairs $(n_{i,q}, n_{j,r})$ such that* $\mathsf{st}(n_{i,q}) \neq ? \implies \mathsf{st}(n_{j,r}) \neq ?$.

A node $n_{j,r}$ depends on a node $n_{i,q}$ if whenever $n_{i,q}$ is either strictly active or inactive, then $n_{j,r}$ has to be either strictly active or inactive. It follows that the configuration space generated by a branch-and-bound method can be reduced by stabilising $n_{j,r}$ whenever $n_{i,q}$ becomes stable. In particular, for a network with n unstable nodes, there are 2^{n-2} configurations that violate a given dependency; therefore, each dependency provides a means to reduce the configuration space by a factor of $1/4$.

Example 1. *Consider the network shown in the left part of Figure 1. In the figure each interval next to a node denotes the pre-activation bounds of the node. Note that nodes $n_{1,2}$ and $n_{2,1}$ are unstable. Assume that a branch-and-bound method branches on node $n_{1,2}$, thereby splitting the optimisation problem into two sub-problems: one where $n_{1,2}$ is strictly active and one where $n_{1,2}$ is strictly inactive. Consider the latter sub-problem. We have that $\mathbf{l}_{1,2} = 0$ and $\mathbf{u}_{1,2} = 0$. Therefore, $\hat{\mathbf{l}}_{2,1} = 1 \cdot 0 + -1 \cdot 0 = 0$ and $\hat{\mathbf{u}}_{2,1} = 1 \cdot 2 - 1 \cdot 0 = 2$. Hence, $n_{2,1}$ is strictly active, and consequently, $(n_{i,j}, n_{q,r}) \in \mathcal{D}_f$. The right part of Figure 1 depicts the configuration space satisfying said dependency.*

We now proceed to derive a procedure for computing a network's dependency relations. To ease the presentation, we express dependency relations as unions of four disjoint sets $\mathcal{D}_f = \bigcup_{z,z' \in \{\top,\perp\}} \mathcal{D}_f^{z,z'}$, where each $\mathcal{D}_f^{z,z'}$ comprises dependencies pertaining to the ReLU states z and z', i.e., $\mathcal{D}_f^{z,z'} \triangleq \{(n_{i,q}, n_{j,r}) \mid \mathsf{st}(n_{i,q}) = z \implies \mathsf{st}(n_{j,r}) = z'\}$. Also, we distinguish between inter- and intra-layer dependencies, which require a different algorithmic treatment. We begin by studying dependencies in the same layer.

Intra-layer dependencies. A dependency $(n_{i,q}, n_{j,r})$ is said to be an *intra-layer dependency* if $i = j$. To compute the dependency relation, given a pair of nodes, we compute the lower and upper bounds of a node under the assumption that the pre-activation of the other is zero, and use the bounds to determine the dependencies.

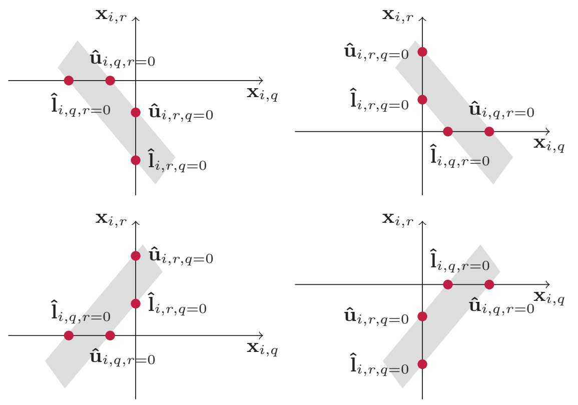
$\mathbf{x}_{i,r}$
$\hat{\mathbf{u}}_{i,q,r=0}$
$\mathbf{x}_{i,q}$
$\hat{\mathbf{l}}_{i,q,r=0}$
$\hat{\mathbf{u}}_{i,r,q=0}$
$\hat{\mathbf{l}}_{i,r,q=0}$

Figure 2: The types of intra-layer dependencies. **Top-left:** if $n_{i,q}$ is active, then $n_{i,r}$ is inactive. **Top-right:** if $n_{i,q}$ is inactive, then $n_{i,r}$ is active. **Bottom-left:** if $n_{i,q}$ is active, then $n_{i,r}$ is active. **Bottom-right:** If $n_{i,q}$ is inactive, then $n_{i,r}$ is inactive.

Formally, for a pair of nodes $n_{i,q}$, $n_{i,r}$, we define $\hat{\mathbf{x}}_{i,q,r=0}$ as the set of pre-activations of $n_{i,q}$ when the pre-activation of $n_{i,r}$ is zero:

$$\hat{\mathbf{x}}_{i,q,r=0} \triangleq \{(W_i)_q \cdot \mathbf{x}_{i-1} + (b_i)_q \mid (W_i)_r \mathbf{x}_{i-1} + (b_i)_r = 0\}.$$

Geometrically, this can be viewed as the intersection of the plane generated by the pre-activations of $n_{i,q}$ and $n_{i,r}$ with $\hat{\mathbf{x}}_{i,r} = 0$. Note that said intersection always exists as both $n_{i,q}$ and $n_{i,r}$ are unstable; therefore, there is an input for which their pre-activations equal zero. On the basis of the lower and upper bounds of $\hat{\mathbf{x}}_{i,q,r=0}$ and $\hat{\mathbf{x}}_{i,r,q=0}$, which can be computed as standard using interval arithmetic, the following lemma identifies the intra-layer dependencies (see Figure 2).

Lemma 1. *For a neural network f and a pair of unstable nodes $(n_{i,q}, n_{i,r})$, the following hold:*

1. $(n_{i,q}, n_{i,r}) \in \mathcal{D}_f^{\top,\bot}$ *iff* $\hat{\mathbf{u}}_{i,q,r=0} < 0$ *and* $\hat{\mathbf{u}}_{i,r,q=0} < 0$.
2. $(n_{i,q}, n_{i,r}) \in \mathcal{D}_f^{\bot,\top}$ *iff* $\hat{\mathbf{l}}_{i,q,r=0} > 0$ *and* $\hat{\mathbf{l}}_{i,r,q=0} > 0$.
3. $(n_{i,q}, n_{i,r}) \in \mathcal{D}_f^{\top,\top}$ *iff* $\hat{\mathbf{u}}_{i,q,r=0} < 0$ *and* $\hat{\mathbf{l}}_{i,r,q=0} > 0$.
4. $(n_{i,q}, n_{i,r}) \in \mathcal{D}_f^{\bot,\bot}$ *iff* $\hat{\mathbf{l}}_{i,q,r=0} > 0$ *and* $\hat{\mathbf{u}}_{i,r,q=0} < 0$.

Lemma 1 gives a procedure for identifying intra-layer dependencies by computing the right hand side of each of the above clauses for every pair of unstable nodes in a layer. Dependencies between layers require a different treatment.

Consecutive-layer and inter-layer dependencies. A dependency $(n_{i,q}, n_{j,r})$ is said to be an *inter-layer dependency* if $j \neq i$. A special case of inter-layer dependencies are those defined by $j = i+1$, which we call *consecutive-layer dependencies*. As we show below the latter are sufficient to obtain the smallest possible configuration space.

Lemma 2. *Let S be the subset of a given network's configuration space that satisfies all consecutive-layer dependencies. Then, every configuration in S satisfies any inter-layer dependency $(n_{i,q}, n_{j,r}) \in \mathcal{D}_f^{z,z'}$ with $j - i > 1$.*

It follows that consecutive-layer dependencies are sufficient to determine the minimum configuration space. To obtain a procedure for calculating consecutive-layer dependencies we introduce the result below.

Lemma 3. *For a neural network f and a pair of unstable nodes $n_{i,q}$, $n_{j,r}$, for $j = i+1$, the following hold:*

1. $(n_{i,q}, n_{j,r}) \in \mathcal{D}_f^{\bot,\bot} \Leftrightarrow \hat{\mathbf{u}}_{j,r} - (W_j)_{r,q} \cdot \mathbf{u}_{i,q} \leq 0.$
2. $(n_{i,q}, n_{j,r}) \in \mathcal{D}_f^{\bot,\top} \Leftrightarrow \hat{\mathbf{l}}_{j,r} - (W_j)_{r,q} \cdot \mathbf{u}_{i,q} \geq 0.$
3. $\mathcal{D}_f^{\top,\bot} = \emptyset$
4. $\mathcal{D}_f^{\top,\top} = \emptyset$

Lemma 3 gives a procedure for identifying consecutive-layer dependencies by checking the right hand side of clauses (1) and (2) for every pair of unstable nodes in consecutive layers.

Dependency analyser. Given the above, we now put forward a procedure using the identification of dependencies to reduce the configuration space. The procedure runs in conjunction with a MILP solver, where it builds a new constraint for each dependency which it adds to the program being analysed by the solver. This is performed at runtime during the branch-and-bound procedure, as the computation of the dependencies is consistent with the current, partial configuration of ReLU nodes being considered by the MILP solver. This allows for the identification of dependencies whilst several nodes have already been stabilised, as opposed to offline methods where most nodes would typically be unstable, thereby hindering the existence of dependencies, as it is rarer for a node to cause a state change on another.

Consider a partial configuration h being considered by the MILP solver. To determine its validity, the solver either extends it to a complete one that satisfies all constraints or to a partial one that violates at least one of the constraints. The dependency analysis procedure put forward here reduces the number of extensions of h that need to be evaluated. Algorithm 1 enumerates the steps of the procedure. First, it stabilises the ReLU nodes as per h and re-computes the bounds for the ones being unstable under h. On the basis of the bounds, it determines the dependencies as per Lemmas 3 and 1. These are then expressed as constraints, referred to as *dependency cuts*, which are added at runtime to the MILP program. The dependency cuts are defined as follows.

Definition 2 (Dependency cuts)**.** *For a partial configuration $h \colon \Delta \to [0,1]$, the associated* dependency cut $cut_{d,h}$ *of a dependency $d \triangleq (n_{i,q}, n_{j,r}) \in \mathcal{D}_f^{z,z'}$ is a MILP constraint defined as follows:*

$$cut_{d,h} \triangleq \gamma_{j,r}(z') \leq \sum_{h(\delta)=0} \delta + \sum_{h(\delta)=1} 1 - \delta + \gamma_{i,q}(z),$$

where $\gamma_{i,q}(z)$ equals $\delta_{i,q}$ if $z = \bot$ and $1 - \delta_{i,q}$ if $z = \top$.

A dependency cut derived from a configuration h is satisfied by an extension of h iff the extended configuration satisfies the corresponding dependency; it follows that each dependency cut removes from the search space all configurations extending h that do not satisfy the dependency. Additionally, for any configuration that does not extend h, the cut

Algorithm 1 The dependency analysis procedure.

```
1:  procedure DEPENDENCY ANALYSIS(milp, $h$)
2:      Input: MILP milp, partial configuration $h$.
3:      for each $h(\delta_{i,q}) = 0$ do
4:          $\mathbf{l}_{i,q} \leftarrow 0$, $\mathbf{u}_{i,q} \leftarrow 0$
5:      Compute remaining bounds (Section 2).
6:      Compute $\mathcal{D}_f$ (Lemmas 3 and 1).
7:      Add $\left\{ cut_{d,h} \mid d \in \mathcal{D}_f \right\}$ to milp (Definition 2)
```

Algorithm 2 The verification procedure.

```
1:  procedure VERIFY($N$, $(\mathcal{X}_0, \mathcal{X}_k)$)
2:      Input: network $N$, specification $(\mathcal{X}_0, \mathcal{X}_k)$
3:      Output: YES/NO
4:      sub-problems $\leftarrow$ split($N, (\mathcal{X}_0, \mathcal{X}_k)$)
5:      result $\leftarrow$ YES
6:      for $P$ in sub-problems do
7:          milp $\leftarrow$ encode($P$)
8:          sub-result $\leftarrow$ milp_solver(milp)
9:          if sub-result is feasible then
10:             result $\leftarrow$ NO
11:             break
12:     return result
```

is trivially satisfied, thereby not altering the search space for those configurations. The former is shown by clause (1) and the latter is proved by clause (2) of the following theorem.

Theorem 1. *Let h be a partial configuration and $d \in \mathcal{D}_f^{z,z'}$ a dependency. Then, the following hold:*

1. *For every h' with $h \subseteq h'$, $h' \models cut_{d,h}$ iff $h' \models d$.*
2. *For every h' with $h \nsubseteq h'$, $h' \models cut_{d,h}$.*

The above concludes the description of the dependency analysis procedure. The procedure runs in time $\mathcal{O}(k \cdot s^2)$, where k is the number of layers and s is the layers' maximal size. Clearly running this procedure has a cost. In the next section we will experimentally evaluate how frequent these calls should be. Also, note that since our dependency framework is a function of the bounds of the ReLU nodes, the procedure can further be optimised by using domain splitting and symbolic interval propagation methods, since these lead to tighter intervals for the ReLU nodes.

In the next section we show that all these factors combined improve the scalability of formal verification of neural networks over the state-of-the-art.

4 The Venus Verification Tool

In this section we introduce Venus (Venus 2019), a verification toolkit that implements the dependency analysis procedure and augments it with symbolic interval arithmetic and domain splitting techniques. While methods on domain splitting divide the input domain into sub-domains, thereby tightening the nodes' bound intervals, methods on symbolic interval arithmetic enable the efficient and tight approximation of the latter; therefore, by Lemmas 1 and 3, both methods promote the existence of dependencies.

Algorithm 3 The splitting procedure.

```
1:  procedure SPLIT($P$)
2:      Input: verification problem $P = (N, (\mathcal{X}_0, \mathcal{X}_k))$
3:      Output: a set of verification sub-problems.
4:      $d \leftarrow 1$                                  ▷ Splitting depth
5:      tosplit $\leftarrow$ [($d$, $\mathcal{X}_0$)]
6:      sub-problems $\leftarrow$ []
7:      while tosplit not empty do
8:          $d$, $R$ $\leftarrow$ pop top element of tosplit
9:          $R_1, R_2 \leftarrow$ best_split($R$)
10:         if worth_split($R$, $R_1$, $R_2$, $d$) then
11:             add $(d+1, R_1)$, $(d+1, R_2)$ to tosplit
12:         else
13:             add $(N, (R, \mathcal{X}_k))$ to sub-problems
14:     return sub-problems
```

The verification procedure upon which Venus is based is outlined in Algorithm 2. The procedure follows a divide-and-conquer approach whereby it recursively splits the input domain until certain heuristic criteria are met and solves the verification sub-problems associated with each sub-domain. Each sub-problem is encoded as a MILP program. These can be analysed in parallel. A MILP program is feasible iff the answer to its associated verification problem is "no". By the definition of the verification problem (Section 2), the answer to the original problem is "no" iff there is at least one sub-problem whose answer is "no". So, as soon as one of the sub-problems is found to be feasible, the procedure terminates without analysing the remaining MILP programs.

Venus uses the "big-M" encoding for the verification problems, and strengthens the linear relaxations by adding *dependency cuts* and *"ideal cuts"* (Anderson et al. 2019) to the MILP programs. The cuts are added at runtime through solver callbacks. Although the cuts strengthen the relaxation, they add complexity to the sub-problems within the solver. Therefore, the addition of a large number of cuts can slow down the solver. Following this, cuts are only added in a fraction of all solver callbacks.

Splitting procedure. The splitting procedure is outlined by Algorithm 3. The procedure recursively splits the input domain by selecting at each step one of the input dimensions and dividing its range in half.

The dimension is heuristically selected on the basis of what we call the *stability-ratio*, the ratio of stable to total number of nodes for a given network and input domain (Line 9). In particular, for each input dimension, we bisect the input domain along the dimension, compute the stability-ratio for each of the two resulting sub-domains and record the average stability-ratio. Then, the dimension along which to split is selected as the one that maximises the recorded averages, or, equivalently, as the one that achieves (on average) the greatest reduction of the configuration space of the induced sub-problems.

Clearly, the number of splits that need to be performed in order to obtain (significantly) simpler sub-problems grows in the number of dimensions. As a result, since the number of sub-problems grows exponentially in the number of splits, the number of sub-problems that need to considered grows exponentially in the number of dimensions. So, whereas verification problems for networks with low input dimensionality can effectively be divided into a number of small sub-problems that are easier to solve, problems for networks with high input dimensionality render such partitions intractable. As reported in the next section, a key advantage of Venus over related tools lies in its ability to solve both low and high input dimensionalities. While domain splitting is very effective for low input dimensionalities, MILP solvers in conjunction with dependency analysers are very powerful for high input dimensionalities. Venus combines the two approaches by considering a heuristic criterion that terminates splitting and signals the employment of an MILP solver (Line 10). The criterion expresses an estimation of the difficulty of the verification problem before splitting versus its difficulty after splitting.

The estimation of the difficulty of a problem p at splitting depth d that we consider is defined by

$$score(p, d) = \frac{stability\_ratio(p) - stability\_ratio(P)}{d^{\frac{1}{m}}},$$

where P is the original verification problem and m is the *splitting parameter*. The larger the score the less difficult the verification problem is estimated to be. The score rewards the improvement of the stability ratio with respect to the original problem and penalises large splitting depths. The splitting parameter controls the degree of "discount" to the splitting depth penalty, where higher values of m signify larger discount. So, following the above discussion, in the case of problems over networks with low input dimensionality, the splitting parameter should be kept high so as to favour splitting. Differently, for problems over networks with high input dimensionality, the splitting parameter should be kept low in order to discourage splitting.

Given a problem p at splitting depth d, and the sub-problems p_1 and p_2 resulting from splitting the chosen dimension of the input domain of p, the splitting is carried out only if the score of (p, d) is less than the average of the scores of $(p_1, d + 1)$ and $(p_2, d + 1)$. In cases where excessive splitting is still observed, a cut-off stability-ratio is used above which the splitting process terminates independently of the aforementioned scores.

Implementation. The architecture of Venus is shown in Figure 3. The toolkit comprises the following components: (i) the *Splitter* performing domain splitting and adding the derived sub-problems to the jobs queue; and (ii) the *Worker* reading sub-problems from the jobs queue, solving them by calling an MILP-solver and dependency analyser ensemble, and recording the verification results to the results queue. Venus aggregates the results from the workers and reports the combined verification result as per Algorithm 2. Both the *Splitter* and *Worker* follow a parallelisation scheme whereby several splitters and workers carry out the domain splitting and the MILP analysis in parallel. Venus is implemented in Python 3.7 and relies on Gurobi 8.1 for the MILP backend.

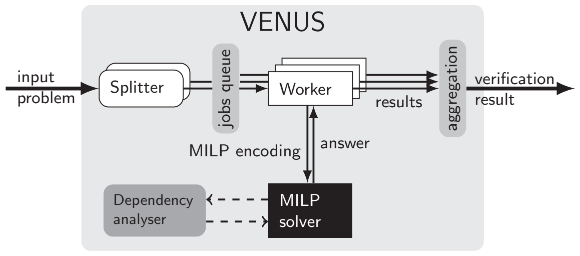

Figure 3: The architecture of Venus.

5 Experimental Results and Evaluation

In this section we evaluate Venus on a number of widely used benchmarks and compare it against the state-of-the-art neural-network verification engines.

For the comparisons we restrict our attention to *complete* methods; while these are often less scalable than incomplete ones, they provide full guarantees on the correctness of their outputs, which is a key objective here. At present, the leading complete verification tools are Marabou (Katz et al. 2019) and Neurify (Wang et al. 2018a). To assess the improvement of Venus over plain MILP-based verification, we additionally compare Venus against NSVerify (Akintunde et al. 2018). We used the most commonly used benchmarks in the context of FFNNs verification:

ACAS Xu (Julian et al. 2016) comprises 45 ReLU-based FFNNs, which were developed as part of an airborne collision avoidance system to advise horizontal steering decisions for unmanned aircrafts. We considered the specifications reported in (Katz et al. 2017). Each was tested on all 45 networks, thereby giving rise (for a total of 10 specifications) to 172 verification problems. For the experiments, Venus was run with 2 splitters, 4 workers, the stability-ratio cutoff set to 0.7, the depth discount set to 20 and with the dependency analyser turned off; Neurify was run with `MAX_THREAD` set to 2; Marabou was run with the parameters reported in (Katz et al. 2019).

MNIST (LeCun, Cortes, and Burges 1998) is a dataset comprising images of hand-written digits 0-9, each formatted as a 28x28x1-pixel grayscale image. We used MNIST to train a FFNN with 2 hidden layers, each comprising 512 neurons. We verified the network against local robustness for a perturbation radius of 0.05 on 100 randomly selected images. For the experiments, we ran Venus with 2 splitters, 2 workers, the stability-ratio set to 0.4, the depth discount set to 4, and the dependency analyser turned on; Neurify was run with `MAX_THREAD` set to 1; Marabou was run with the parameters reported in (Katz et al. 2019).

CIFAR-10 (Krizhevsky, Nair, and Hinton 2014) is a dataset comprising images of objects from 10 different classes (airplanes, cars, birds, cats, etc.). Each image is formatted as a 32x32x3-pixel colour image. We used CIFAR-10 to train a FFNN with 3 hidden layers, the first comprising 1024 neurons, and the second and third comprising 512 neurons. We verified the network against local robustness for a perturbation radius of 0.01 on 100 randomly selected images. We ran all tools with the same parameters as for MNIST.

	MNIST (100 verification queries)					**CIFAR-10** (100 verification queries)					**ACAS XU** (172 verification queries)				
	n_s	t_{all}	t_{solved}	av_s	$\frac{t_{all}}{t_{all}^{\text{Venus}}}$	n_s	t_{all}	t_{solved}	av_s	$\frac{t_{all}}{t_{all}^{\text{Venus}}}$	n_s	t_{all}	t_{solved}	av_s	$\frac{t_{all}}{t_{all}^{\text{Venus}}}$
Venus	100	5,953.46	573.38	9.10	–	100	857.11	560.04	7.36	–	170	19,642.57	5,527.76	36.36	–
Marabou	0	86,400.00	–	–	14.51	0	86,400.00	–	–	100.80	156	140,916.96	75,747.78	498.3	7.17
Neurify	65	126,007.24	7.00	0.11	21.17	76	87,178.46	778.46	10.24	101.71	167	23,628.75	2,555.38	16.81	1.2
NSVerify	95	26,906.81	2,515.15	39.9	4.52	100	6,898.64	3,460.41	45.53	8.04	6	86,400.00	–	–	30.77

Table 1: Experimental results obtained when running Venus, Marabou, Neurify and NSVerify.

All experiments were carried out on an Intel Core i7-7700K (4 cores) equipped with 16GB RAM, running Ubuntu 18.04. Each verification query had a *local* timeout of 1 hour. The sum of verification queries associated with each benchmark had a *global* timeout of 24 hours. Table 1 reports the experimental results. For each of the tools and benchmarks, the table gives the number n_s of verification queries that were solved, the overall time t_{all} taken for all queries, the overall time t_{solved} and the average time av_s taken for the queries that three best performing tools were able to solve, and the ratio between overall time taken t_{all} by a tool over that of Venus, t_{all}^{Venus}.

The results obtained on MNIST show that Venus was the most performing of the toolkits, both in terms of the overall verification time and the number of verification queries solved. Neurify was not able to analyse 35 of the queries because of excessive memory consumption. For these cases, we considered Neurify as having timed out. Marabou did not solve any of the queries under local and global timeouts. NSVerify performed better than both Neurify and Marabou. Venus was found 4.52 times faster than NSVerify and 21 times faster than Neurify. For CIFAR-10 the difference between Venus and the other tools was greater, suggesting that the higher the dimensionality and complexity of the model, the bigger the difference.

Venus's performance was also found superior on ACAS XU, both in terms of the overall verification time and the number of queries solved. Neurify was the fastest tool w.r.t. the number of queries that all the tools could solve. Marabou solved a comparable number of queries to Venus and Neurify but was slower than both of them. NSVerify solved only 6 queries within the local and global timeouts.

Figure 4 gives a graphical representation of the total number of verification queries that each tool could verify as a function of time. In summary, Venus solved most verification instances after approx. 15 secs. Also, to the best of our knowledge, Venus is the only tool that can seemingly analyse both low-dimensional and high-dimensional networks, outperforming the state-of-the-art tools for each class, often by more than one order of magnitude. The only aspect we found Venus to be less performing was counterexample generation where Neurify was the fastest tool.

The experiments also suggest that the verification of networks with low input-dimensionality is particularly amenable to domain splitting, as domain splitting techniques act as effective configuration-space minimisers. As a result, branch-and-bound methods that combine domain splitting are advantageous over ones that do not, as indicated by the outperformance of Venus over NSVerify on ACAS.

Ablation test	n_s	av_s	t_{all}	t_{solved}
Big-M formulation	98	38.15	13,555.49	3,700.62
Splitting	98	42.58	11,409.96	4,129.83
Ideal formulation	100	36.02	6,777.95	3,460.17
Splitting+Ideal	100	36.75	8,277.85	3,565.02
Inter Deps	99	30.06	8,561.21	2,916.23
Intra Deps	98	33.03	10,426.82	3,203.54
Inter+Intra Deps	98	25.81	9,729.07	2,503.41
All methods enabled	100	26.52	5,953.46	2,572.90

Table 2: Ablation experiments for MNIST. The average is calculated for the images that are verified in all cases. Similarly, t_{solved} is calculated for the images verified in all cases.

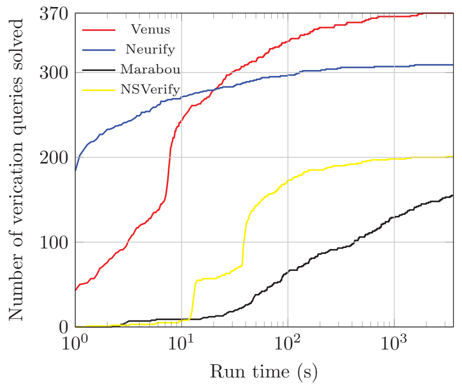

Figure 4: Number of verification queries that Venus, Neurify, Marabou and NSVerify could solve as a function of time.

Differently, for high-dimensional input domains, the experiments suggest that domain splitting methods do not significantly reduce the configuration space, as indicated by the degraded performance of Neurify and Marabou on MNIST and CIFAR-10. In contrast, techniques that directly target the reduction of the configuration space exhibit high efficacy over high-dimensional inputs, as exemplified by Venus and NSVerify. Venus is more effective than NSVerify by considering FFNN-specific configuration-space reductions. This suggests that MILP solvers are not necessarily best used as black boxes, but application-specific considerations can help to improve their effectiveness.

Indeed, the performance gains exhibited by Venus over NSVerify on MNIST and CIFAR-10 are a consequence of combining dependency analysis and ideal formulations. This

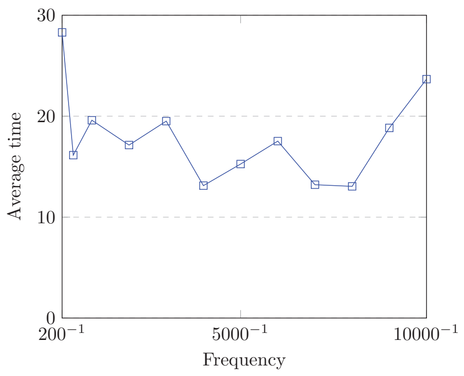

Figure 5: Average runtime of Venus on MNIST as a function of callback frequency.

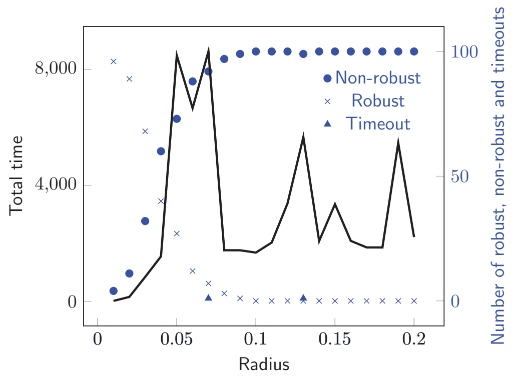

Figure 6: Total runtime of Venus on MNIST as a function of perturbation radius.

is evidenced by separately evaluating Venus on MNIST for different combinations of the techniques that the tool implements. Table 2 reports ablation experiments to analyse this in detail. The results confirm that domain splitting is not effective for high dimensional inputs. They also show that ideal formulations and dependency analysis improve on pure big-M formulations not only when they are jointly utilised but also when they are independently employed. In the latter case, ideal formulations enabled the verification of two images that could not be verified by dependency analysis, whereas dependency analysis led to better average and total time for verifying all images that could be verified in all cases. In the latter case the results suggest that the combination of dependency analysis and ideal formulations is preferable in terms of all performance metrics than either technique considered in isolation.

As discussed in Section 3, running the dependency analysis procedure has a cost. As a result, the above performance gains can only be obtained after determining how often these calls should be. Figure 5 gives the average runtime of Venus on 100 MNIST images as a function of the frequency with which the dependency analysis procedure is called from a Gurobi callback. High and low frequencies degrade the performance of Venus, whereas frequencies in the range $[0.00014-0.00025]$ balance out the cost of computing dependencies and the reduction of the configuration space enabled by their computation.

We conclude this section by studying the performance of Venus as a function of the perturbation radius for which the robustness of MNIST is established. Intuitively, small perturbation radiuses pertain to easy verification problems on the one hand, as the bounds for the nodes are tighter, and to hard problems on the other hand, as the network is more likely to be robust w.r.t. the corresponding perturbed images. Similarly, large perturbation radiuses pertain to hard verification problems on the one hand, as the bounds for the nodes are looser, and to easy problems on the other hand, as the network is less likely to be robust w.r.t. the corresponding perturbed images. Figure 6 reports Venus's total running time for verifying 100 MNIST images and for perturbation radiuses that range from 0.01 to 0.2. The figure also shows the number of images for which Venus has timed out and for which the network was found non-robust and robust. The figure shows that Venus is consistently efficient for all perturbation radiuses (with an average verification time per image of less than 90 seconds). The figure also indicates that the performance of Venus is mostly degraded for perturbation radiuses within the range $[0.05, 0.07]$. Notably, these radiuses result in verification problems that do not permit for sufficiently tight bounds for the nodes whilst not exhibiting sufficiently adversarial regions.

6 Conclusions

As we argued in the introduction, the deployment of learning methods based on neural networks in safety critical AI applications urgently requires verification and validation methods. A growing area of research is concerned with the development of formal verification methods for neural networks with particular emphasis to ReLU-based deep networks used in vision and control. While progress in this area has been rapid, the present state-of-the-art still falls short of the capabilities required to verify industry-strength models. It is unlikely that this scalability issue will be solved in the immediate future; but there is a need for novel methods to gradually conquer larger and larger networks.

In this paper we introduced the concept of dependency analysis which we developed in the context of a MILP-based verification method. The method further benefits from input splitting and symbolic interval propagation. We derived algorithms based on the resulting theory and reported the results obtained with Venus, a novel tool for the verification of neural networks. As we demonstrated experimentally on three different, widely used benchmarks, Venus could solve more verification queries than the present state-of-the art tool based on complete methods. Venus is also the fastest tool to verify the correctness of a network; in some cases Neurify proved to be faster in finding counterexamples.

In future work we intend to apply Venus to the verification of more complex specifications for neural networks including transformational robustness (Kouvaros and Lomuscio 2018).

Acknowledgements. This research was partly funded by DARPA under the Assured Autonomy program, the Royal

Academy of Engineering (via a Chair in Emerging Technologies), the Royal Society (NIF\R1\182194), and ESPRC (grant EP/P016871/1).

References

Akintunde, M. E.; Lomuscio, A.; Maganti, L.; and Pirovano, E. 2018. Reachability analysis for neural agent-environment systems. In *Proceedings of the 16th International Conference on Principles of Knowledge Representation and Reasoning (KR18)*, 184–193. AAAI Press.

Anderson, R.; Huchette, J.; Tjandraatmadja, C.; and Vielma, J. 2019. Strong mixed-integer programming formulations for trained neural networks. In *Integer Programming and Combinatorial Optimization (IPCO19)*, Lecture Notes in Computer Science, 27–42. Springer.

Bastani, O.; Ioannou, Y.; Lampropoulos, L.; Vytiniotis, D.; Nori, A. V.; and Criminisi, A. 2016. Measuring neural net robustness with constraints. In *Proceedings of the 30th International Conference on Neural Information Processing Systems (NIPS16)*, 2613–2621.

Bunel, R. R.; Turkaslan, I.; Torr, P.; Kohli, P.; and Mudigonda, P. K. 2018. A unified view of piecewise linear neural network verification. In *Proceedings of the 31st Annual Conference on Neural Information Processing Systems (NeurIPS18)*. Curran Associates, Inc. 4790–4799.

Cheng, C.-H.; Nührenberg, G.; and Ruess, H. 2017. Maximum resilience of artificial neural networks. In *Automated Technology for Verification and Analysis*, 251–268. Springer International Publishing.

Dvijotham, K.; Stanforth, R.; Gowal, S.; Mann, T.; and Kohli, P. 2018. A dual approach to scalable verification of deep networks. In *Proceedings of the 34th Annual Conference on Uncertainty in Artificial Intelligence (UAI18)*, 162–171. AUAI Press.

Ehlers, R. 2017. Formal verification of piece-wise linear feed-forward neural networks. In *Proceedings of the 15th International Symposium on Automated Technology for Verification and Analysis (ATVA17)*, volume 10482 of *Lecture Notes in Computer Science*, 269–286. Springer.

Fazlyab, M.; Morari, M.; and Pappas, G. J. 2019. Safety verification and robustness analysis of neural networks via quadratic constraints and semidefinite programming. *arXiv preprint arXiv:1903.01287*.

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

Gehr, T.; Mirman, M.; Drachsler-Cohen, D.; Tsankov, P.; Chaudhuri, S.; and Vechev, M. 2018. AI^2: Safety and robustness certification of neural networks with abstract interpretation. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (S&P18)*, 948–963.

Julian, K.; Lopez, J.; Brush, J.; Owen, M.; and Kochenderfer, M. 2016. Policy compression for aircraft collision avoidance systems. In *Proceedings of the 35th Digital Avionics Systems Conference (DASC16)*, 1–10.

Karmarkar, N. 1984. A new polynomial-time algorithm for linear programming. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing (STOC84)*, 302–311. ACM.

Katz, G.; Barrett, C. W.; Dill, D. L.; Julian, K.; and Kochenderfer, M. J. 2017. Reluplex: An efficient SMT solver for verifying deep neural networks. In *Proceedings of the 29th International Conference on Computer Aided Verification (CAV17)*, volume 10426 of *Lecture Notes in Computer Science*, 97–117. Springer.

Katz, G.; Huang, D. A.; Ibeling, D.; Julian, K.; Lazarus, C.; Lim, R.; Shah, P.; Thakoor, S.; Wu, H.; Zeljic, A.; Dill, D. L.; Kochenderfer, M. J.; and Barrett, C. W. 2019. The marabou framework for verification and analysis of deep neural networks. In *Proceedings of the 31st International Conference on Computer Aided Verification (CAV19)*, 443–452.

Kouvaros, P., and Lomuscio, A. 2018. Formal verification of cnn-based perception systems. *arXiv preprint arXiv:1811.11373*.

Krizhevsky, A.; Nair, V.; and Hinton, G. 2014. The CIFAR-10 dataset. http://www.cs.toronto.edu/kriz/cifar.html.

LeCun, Y.; Cortes, C.; and Burges, C. J. 1998. The MNIST database of handwritten digits.

Liu, C.; Arnon, T.; Lazarus, C.; Barrett, C.; and Kochenderfer, M. 2019. Algorithms for verifying deep neural networks. *CoRR* abs/1903.06758.

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

Raghunathan, A.; Steinhardt, J.; and Liang, P. S. 2018. Semidefinite relaxations for certifying robustness to adversarial examples. In *Proceedings of 31st Annual Conference on Neural Information Processing Systems (NeurIPS18)*, 10900–10910.

Rubies-Royo, V.; Calandra, R.; Stipanovic, D. M.; and Tomlin, C. 2019. Fast neural network verification via shadow prices. In *Proceedings of the 36th International Conference on Machine Learning (ICML19)*.

Tjeng, V.; Xiao, K. Y.; and Tedrake, R. 2019. Evaluating robustness of neural networks with mixed integer programming. In *Proceedings of the 7th International Conference on Learning Representations (ICLR19)*.

Venus. 2019. https://vas.doc.ic.ac.uk/software/neural.

Wang, S.; Pei, K.; Whitehouse, J.; Yang, J.; and Jana, S. 2018a. Efficient formal safety analysis of neural networks. In *Proceedings of the 31st Annual Conference on Neural Information Processing Systems 2018 (NeurIPS18)*, 6369–6379.

Wang, S.; Pei, K.; Whitehouse, J.; Yang, J.; and Jana, S. 2018b. Formal security analysis of neural networks using symbolic intervals. In *Proceedings of the 27th USENIX Security Symposium, (USENIX18)*, 1599–1614.

Zhang, H.; Weng, T.; Chen, P.; Hsieh, C.; and Daniel, L. 2018. Efficient neural network robustness certification with general activation functions. In *Proceedings of the 31st Annual Conference on Neural Information Processing Systems 2018 (NeurIPS2018)*, 4944–4953. Curran Associates, Inc.