DEEPSPLIT: An Efficient Splitting Method for Neural Network Verification via Indirect Effect Analysis

Patrick Henriksen , **Alessio Lomuscio**
Imperial College London
{patrick.henriksen18, a.lomuscio}@imperial.ac.uk

Abstract

We propose a novel, complete algorithm for the verification and analysis of feed-forward, ReLU-based neural networks. The algorithm, based on symbolic interval propagation, introduces a new method for determining split-nodes which evaluates the indirect effect that splitting has on the relaxations of successor nodes. We combine this with a new efficient linear-programming encoding of the splitting constraints to further improve the algorithm's performance. The resulting implementation, DEEPSPLIT, achieved speedups of around 1–2 orders of magnitude and 21–34% fewer timeouts when compared to the current SoA toolkits.

1 Introduction

AI-based, safety-critical applications, including autonomous driving, require reliable systems that are amenable to analysis leading to their certification. At present even the most complex neural classifiers suffer from misclassification rates that render the technology potentially unsafe. It therefore remains of importance to develop methods to formally verify the correctness of neural networks and identify unwanted features.

The area of *formal verification of neural networks* [Liu *et al.*, 2019] is concerned with the development of methods to ascertain whether a neural model is correct with respect to a given specification. For example, in image classification, we might need to establish whether for a given image a classifier is robust with respect to noise [Szegedy *et al.*, 2014] or geometric perturbations [Kouvaros and Lomuscio, 2018; Balunovic *et al.*, 2019]. In closed-loop systems with neural controllers, it is of interest to establish whether the system may violate some safety constraints expressed as temporal specifications [Akintunde *et al.*, 2018; Akintunde *et al.*, 2019; Tran *et al.*, 2020b; Akintunde *et al.*, 2020].

Given their noteworthy application domains, particular attention has been devoted to verifying high-dimensional, ReLU-based, feed-forward neural networks; this is also the aim of this paper. While a number of increasingly scalable methods have been proposed recently, the present state-of-the-art (SoA) cannot yet address the models used in practice. The key contribution in this paper is to define a procedure, within a symbolic interval propagation framework, for deciding which nodes to split based on the direct and indirect effect that the splitting has on the relaxations in subsequent nodes in the network. As we show, this leads to 1–2 orders of magnitude speedup over current SoA verifiers.

Related Work

Verification methods for neural networks can be divided into complete and incomplete methods. Complete methods are theoretically guaranteed to provide an answer to any verification query; incomplete methods may not reach a conclusion on a query but may be more scalable when a solution can be found. Both streams of work are considered important as they can complement each other.

Complete approaches can be further divided into Mixed Integer Linear Programming (MILP), reachable set and relaxation based approaches. MILP based approaches encode the verification problem into MILP solvers [Lomuscio and Maganti, 2017; Akintunde *et al.*, 2018; Anderson *et al.*, 2020; Botoeva *et al.*, 2020; Singh *et al.*, 2019b; Tjeng *et al.*, 2019]. Reachable-set based approaches [Tran *et al.*, 2020a; Tran *et al.*, 2020b; Bak *et al.*, 2020] verify properties by propagating exact symbolic representations of the reachable states through the network and reasoning over the resulting output states. Relaxation based methods [Henriksen and Lomuscio, 2020; Katz *et al.*, 2019; Rubies-Royo *et al.*, 2019; Wang *et al.*, 2018a; Bunel *et al.*, 2020; Singh *et al.*, 2019a] compute a linear relaxation of the network and iteratively refine it by splitting ReLU nodes and branching; an efficient relaxation can be obtained with Symbolic Interval Propagation (SIP) [Wang *et al.*, 2018b; Wang *et al.*, 2018a; Singh *et al.*, 2019c; Singh *et al.*, 2018; Henriksen and Lomuscio, 2020]. The work presented here also adopts the SIP-based approach, but differs from the SoA in that it (i) employs a novel procedure for splitting which is based on dependencies between ReLU relaxations, (ii) utilises a new succinct LP-encoding of the split-constraints and (iii) combines two different SIP algorithms to improve the linear relaxation. This results in significant speedups, as we report below.

[Lu and Kumar, 2020; Bak, 2020] also consider procedures for branching decisions. [Lu and Kumar, 2020] uses a graph-neural network for deciding on branching nodes; this approach was only shown to be able to verify relatively small networks with < 7000 ReLU nodes. In contrast we use inter-

mediate results from the SIP phase, leading to a more scalable approach. The procedure in [Bak, 2020] starts from a non-relaxed network and gradually increases the nodes to relax; in contrast, we start with a relaxed network and gradually decrease the abstraction through branching. In our experiments this results in a more scalable approach.

Node-dependency analysis is also present in [Botoeva *et al.*, 2020; Rubies-Royo *et al.*, 2019]. However, [Botoeva *et al.*, 2020] determines binary relations between nodes, while we estimate fractional dependencies between the nodes' relaxations. Dependencies are considered in [Rubies-Royo *et al.*, 2019] via the solution of a linear program encoding all the network's ReLUs; in common with other SIP-based approaches, we avoid encoding individual ReLUs for efficiency.

Incomplete verification algorithms [Singh *et al.*, 2019c; Singh *et al.*, 2018; Wang *et al.*, 2018b] differ from complete approaches in that they reason over a relaxed, or abstract network and do not implement a refinement strategy which guarantees that all overestimation is removed in a finite number of steps and are therefore not directly comparable to complete approaches. However, we show later that SoA complete approaches significantly outperformed a SoA incomplete approach on several benchmarks.

2 Preliminaries

In this section we introduce feed-forward neural networks, symbolic interval propagation and some related concepts.

We assume a *Feed-Forward Neural Network (FFNN)* to be composed of an input layer, one or more hidden layers and an output layer [Goodfellow *et al.*, 2016]. Each layer consists of multiple nodes and hidden layers are governed by non-linear activation functions $\sigma : \mathbb{R} \to \mathbb{R}$. We here consider networks with the $ReLU(z) = \max(0, z)$ activation function only. A layer's output is calculated by applying the layer's activation function element-wise to its input vector where the input vector is an affine transformation of the output from previous layers. So, a network with n input nodes and m output nodes is associated with a function $f : \mathbb{R}^n \to \mathbb{R}^m$ where f is the composition of the layer-by-layer transformations.

In this paper we consider verification problems defined by box constraints on the network's input and linear constraints on the output as defined in Definition 1. While much of the current body of work on neural network verification does not explicitly define the verification problem, the problem presented here is in line with the main focus of the field.

Definition 1. *A verification problem is a tuple $\langle f, \psi_{in}, \psi_{out} \rangle$ where $f : \mathbb{R}^n \to \mathbb{R}^m$ is a FFNN with ReLU activation functions for all hidden nodes, $\psi_{in} = \bigwedge_{k=0}^{m} l_k \leq \boldsymbol{x}_k \leq u_k$ is the input constraint, $\psi_{out} = \bigwedge_i (\bigvee_j \boldsymbol{a}_{i,j}^{\intercal} f(\boldsymbol{x}) + b_{i,j} < 0)$ is the output constraint, $l_k, u_k, b_{i,j} \in \mathbb{R}$ and $\boldsymbol{a}_{i,j} \in \mathbb{R}^m$. If ψ_{out} is satisfied for all $\boldsymbol{x}$ that satisfy ψ_{in}, then we say that the f is safe for the given constraints; otherwise, f is unsafe. An $\boldsymbol{x}$ that satisfies ψ_{in} and violates ψ_{out} is a counterexample.*

Verification problems are challenging to solve due to non-linear activation functions. A scalable approach to solve verification problems involves considering a linear relaxation of the FFNN [Wang *et al.*, 2018b; Wang *et al.*, 2018a]. The relaxed FFNN can be obtained by combining Symbolic Interval Propagation (SIP) with two-constraint linear relaxations of the activation functions. In the rest of this paper we follow the presentation of two-constraint linear relaxations from [Henriksen and Lomuscio, 2020] summarised below.

Definition 2 (Two-constraint relaxation)**.** *Let $z_l, z_u \in \mathbb{R}$ be concrete lower and upper bounds on the input of an activation function $\sigma : [z_l, z_u] \to \mathbb{R}$, where $[z_l, z_u] \subseteq \mathbb{R}$ denotes the closed real interval between z_l and z_u. A two-constraint linear relaxation is a tuple $\langle r_l, r_u \rangle$, where $r_l : [z_l, z_u] \to \mathbb{R}$ is a lower bounding linear function such that $r_l(z) \leq \sigma(z)$ and $r_u : [z_l, z_u] \to \mathbb{R}$ is an upper bounding linear function such that $r_u(z) \geq \sigma(z)$ for all $z \in [z_l, z_u]$.*

For ReLU functions where the input is bounded by lower and upper bounds $z_l, z_u \in \mathbb{R}$, the upper relaxation is normally defined by $r_u(z) = \frac{z_u}{z_u - z_l}(z - z_l)$ [Wang *et al.*, 2018a; Singh *et al.*, 2019c]. The formulation of the lower relaxation depends on the SIP algorithm used. Some approaches use $r_l(z) = \frac{z_u}{z_u - z_l} z$ [Wang *et al.*, 2018a]; others minimise the overestimation area by considering $r_l(z) = 0$ iff $z_u < |z_l|$ and $r_l(z) = z$ otherwise [Singh *et al.*, 2019c].

Relaxations are used in conjunction with SIP to calculate bounds on the network's output. Relevant to this work are two distinct SIP algorithms: Error-based Symbolic Interval Propagation (ESIP) [Wang *et al.*, 2018a; Henriksen and Lomuscio, 2020] and the algorithm here referred to as Reversed Symbolic Interval Propagation (RSIP) [Singh *et al.*, 2019c].

The ESIP algorithm propagates bounding equations and concrete errors layer-by-layer through the network. Activation functions are handled by propagating the terms through the lower relaxation and adding a new concrete error initialised as $\epsilon = \max_{z \in [z_l, z_u]}(r_u(z) - r_l(z))$ for each node; these errors are then propagated through the network with the equations. In the following $\epsilon_{n_1,n_2}^{l_1,l_2}$ denotes the error from a node in layer l_1 and position n_1 at a node in layer l_2 and position n_2 where $l_1 < l_2$; however, we sometimes use the simplified notation ϵ^n to mean the error from a node n when this does not cause confusion. For a node in layer l_2 and position n_2 with equation $q_{n_2}^{l_2}(\boldsymbol{x})$, the concrete lower and upper bounds $\boldsymbol{z}_{n_2,low}^{l_2}, \boldsymbol{z}_{n_2,up}^{l_2}$ (denoted as $\boldsymbol{y}_{n_2,low}$ and $\boldsymbol{y}_{n_2,up}$ at the output layer) are calculated as $\boldsymbol{z}_{n_2,low}^{l_2} = \min_{\boldsymbol{x}} q_{n_2}^{l_2}(\boldsymbol{x}) + \sum_{l_1=0}^{l_2-1} \sum_{n_1 | \epsilon_{n_1,n_2}^{l_1,l_2} < 0} \epsilon_{n_1,n_2}^{l_1,l_2}$ and $\boldsymbol{z}_{n_2,up}^{l_2} = \max_{\boldsymbol{x}} q_{n_2}^{l_2}(\boldsymbol{x}) + \sum_{l_1=0}^{l_2-1} \sum_{n_1 | \epsilon_{n_1,n_2}^{l_1,l_2} > 0} \epsilon_{n_1,n_2}^{l_1,l_2}$.

Different from ESIP, the RSIP algorithm calculates bounding equations by substituting variables layer-by-layer backwards through the network. For non-linear activation functions, the upper symbolic bounds are back-propagated by applying the upper relaxation on positive coefficients and lower relaxation for negative coefficients. Correspondingly, lower bounds are calculated by applying the lower relaxation to positive and upper relaxations to negative coefficients. RSIP often produces more succinct bounds than ESIP; however, in the next section we show that intermediate calculations from ESIP are particularly well suited to analyse the network.

The bounds from the symbolic interval propagation can be used to calculate bounds for the linear equations in the verification problem's output constraint ψ_{out} [Singh *et al.*, 2018;

Singh *et al.*, 2019c]. Complete verification approaches [Wang *et al.*, 2018b; Wang *et al.*, 2018a; Henriksen and Lomuscio, 2020] combine SIP with a branch and bound phase and linear programming to constrain the ReLU nodes' input to > 0 and < 0 in separate branches to achieve completeness. Since the bounds produced by both SIP algorithms are exact whenever all ReLU nodes operate in one of their linear domains, the verification problem can be solved by exploring at most 2^N branches where N is the number of ReLU nodes in the network. In Section 4 we show how ESIP and RSIP can be combined into an efficient verification algorithm.

Due to the branch and bounds phases' exponential complexity, it is infeasible to branch on all ReLU nodes even for small networks. So, splitting the correct nodes is essential for an efficient implementation. In the next section we propose a novel method for identifying good splitting candidates thereby resulting in a an efficient verification procedure.

3 Splitting Score Functions

A key aspect differentiating all symbolic interval propagation methods lies in the choice of nodes to be split. Technically, splitting a ReLU node is realised by creating two separate branches in the resolution tree: in the first the node's input is constrained to negative values, in the second to positive ones. In the ESIP algorithm this operation has three distinct effects.

- **Direct effect.** The error cascading from the split node becomes 0 in both branches, thus improving the bounds of successor nodes.
- **Indirect effect**. Since the bounds for all nodes succeeding the split node improve, the corresponding linear relaxations are also improved which, in turn, tightens the bounds in all layers after those nodes.
- **Propagation effect**. After the split, the equations and errors are propagated through the ReLU function instead of the lower relaxation at the split node.

While the categorisation above is novel to this paper, the direct effect has previously been used to determine splitting candidates [Henriksen and Lomuscio, 2020] and the gradient-based methods from [Wang *et al.*, 2018b; Wang *et al.*, 2018a] are closely related to the direct and propagation effects. However, neither of the methods account for the indirect effect.

In the following we propose a novel method that estimates the effect a split has on the SIP-bounds relevant to the verification problem. The method considers both the direct and indirect effects, but not propagation effects as empirical evidence obtained suggests that they do not have a significant impact in ReLU networks.

The effect of splitting a node is estimated with a score function $s : N \to \mathbb{R}$ where N is the set of all hidden nodes in the FFNN. This score function is composed of the direct score $s_{dir} : N \to \mathbb{R}$ and the indirect score $s_{indir} : N \to \mathbb{R}$, so $s = s_{dir} + s_{indir}$. These functions are formalised below, but first we introduce some necessary notation and concepts.

In the rest of this section let $\langle f, \psi_{in}, \psi_{out} \rangle$ be the verification problem as defined in Definition 1 where $f : \mathbb{R}^n \to \mathbb{R}^m$ is an FFNN with l layers. Moreover, let $\epsilon^{l_1,l_2}_{n_1,n_2}$ denote the ESIP error from a node in layer l_1 and position n_1 at a node in layer l_2 and position n_2. Finally, let $\boldsymbol{y}_{n,low}, \boldsymbol{y}_{n,up}$ be the concrete lower and upper bounds for output node n as calculated by ESIP, so $\boldsymbol{y}_{n,low} \leq f(\boldsymbol{x})_n \leq \boldsymbol{y}_{n,up}$ for all $\boldsymbol{x}$ satisfying ψ_{in} where $f(\boldsymbol{x})_n \in \mathbb{R}$ is the n-th output of the neural network.

With this in mind, we now introduce a upper bound for the linear equations in ψ_{out} and motivate the score functions. Let $\boldsymbol{a}^\intercal f(\boldsymbol{x}) + b \leq 0$ be a linear constraint in ψ_{out} and notice that for all $\boldsymbol{x}$ satisfying ψ_{in} the equation $g_{up} = \sum_{n|(\boldsymbol{a}_n>0)} \boldsymbol{y}_{n,up}\boldsymbol{a}_n + \sum_{n|(\boldsymbol{a}_n<0)} \boldsymbol{y}_{n,low}\boldsymbol{a}_n + b$ is an upper bound for $\boldsymbol{a}^\intercal f(\boldsymbol{x}) + b$. Thus $g_{up} \leq 0$ is a sufficient condition for $\boldsymbol{a}^\intercal f(\boldsymbol{x}) + b \leq 0$ for all $\boldsymbol{x}$ satisfying ψ_{in}. The score function here proposed aims to identify splitting candidates that minimise the bound g_{up} by minimising $\boldsymbol{y}_{n,up}$ if $\boldsymbol{a}_n > 0$ and maximising $\boldsymbol{y}_{n,low}$ if $\boldsymbol{a}_n < 0$ for all n.

To estimate the direct effect, recall from Section 2 that negative errors are added to $\boldsymbol{y}_{n,low}$ and positive errors to $\boldsymbol{y}_{n,up}$ and that the error from a node $h^{l_1}_{n_1}$ in layer l_1 and position n_1 is removed after splitting the node. So, a good estimate of the reduction of g_{up} due to the direct effect from splitting $h^{l_1}_{n_1}$ is given by $\hat{s}_{dir}(h^{l_1}_{n_1}) = \sum_{n|(\text{sign}(\epsilon^{l_1,l}_{n_1,n})=\text{sign}(\boldsymbol{a}_n))} \epsilon^{l_1,l}_{n_1,n}\boldsymbol{a}_n$. where l is the network's output layer.

The score function $\hat{s}_{dir}(h^{l_1}_{n_1})$ therefore estimates how much closer we are to proving that a single linear constraint $\boldsymbol{a}^\intercal f(\boldsymbol{x}) + b \leq 0$ is satisfied for all $\boldsymbol{x}$ satisfying ψ_{in} after the split. However, the output constraint ψ_{out} as defined in Definition 1, is generally a conjunction of disjunctions of linear constraints. To estimate the cumulative effect, we sum the coefficients in each individual linear constraint $\boldsymbol{a}^\intercal_{i,j} f(\boldsymbol{x}) + b_{i,j} \leq 0$ in ψ_{out}, so $\hat{\boldsymbol{a}} = \sum_{i,j} \boldsymbol{a}_{i,j}$ and:

$$s_{dir}(h^{l_1}_{n_1}) = \sum_{n|(\text{sign}(\epsilon^{l_1,l}_{n_1,n})=\text{sign}(\hat{\boldsymbol{a}}_n))} \epsilon^{l_1,l}_{n_1,n}\hat{\boldsymbol{a}}_n \tag{1}$$

Section 4 covers the details of which constraints are used at different stages of the algorithm. Notice that the score function above computes the same value as that in [Henriksen and Lomuscio, 2020] in the case of local robustness for classification networks, but generalises it and covers all verification problems conforming to Definition 1. An example of such a problem can be found in the commonly used ACAS Xu benchmarks, where the task is to verify that an output node is never minimal; this is not supported in [Henriksen and Lomuscio, 2020], but does indeed conform to Definition 1.

In order to estimate the indirect effect, first notice that nodes in the last hidden layer $l-1$ do not have an indirect effect as there are no intermediate layers between layer $l-1$ and the output layer, so we take the score function to be $s(h^{l-1}_{n_2}) = s_{dir}(h^{l-1}_{n_2})$. However, indirect effects are present for layers preceding $l-1$.

Recall that $s(h^{l-1}_{n_2})$ estimates how much g_{up} is reduced by removing the error from node $h^{l-1}_{n_2}$. Thus, if splitting node $h^{l-2}_{n_1}$ removes an $\alpha^{l-2,l-1}_{n_1,n_2} \in [0,1]$ fraction of the error from $h^{l-1}_{n_2}$, then $s_{indir}(h^{l-2}_{n_1}) = \sum_{n_2} \alpha^{l-2,l-1}_{n_1,n_2} s(h^{l-1}_{n_2})$ is a good estimate of the indirect effects from splitting $h^{l-2}_{n_1}$.

We now estimate $\alpha^{l-2,l-1}_{n_1,n_2}$. Since the node $h^{l-1}_{n_2}$ is governed by the ReLU activation function, the error is completely removed if the lower bound $\boldsymbol{z}^{l-1}_{n_2,low}$ is larger than 0, or if the

upper bound $\boldsymbol{z}^{l-1}_{n_2,up}$ smaller than 0. Similar to the direct score function, we estimate that splitting node $h^{l-2}_{n_1}$ reduces the upper bound of node $h^{l-1}_{n_2}$ by $\epsilon^{l-2,l-1}_{n_1,n_2}$ if $\epsilon^{l-2,l-1}_{n_1,n_2} \geq 0$ (respectively, increases lower bound if $\epsilon^{l-2,l-1}_{n_1,n_2} \leq 0$). This leads us to the estimation $\hat{\alpha}^{l-2,l-1}_{n_1,n_2} = \epsilon^{l-2,l-1}_{n_1,n_2} / \boldsymbol{z}^{l-1}_{n_2,up}$ if $\epsilon^{l-2,l-1}_{n_1,n_2} \geq 0$ otherwise $\hat{\alpha}^{l-2,l-1}_{n_1,n_2} = \epsilon^{l-2,l-1}_{n_1,n_2} / \boldsymbol{z}^{l-1}_{n_2,low}$. Generalising this reasoning to arbitrary layers l_1 and l_2 with $l_1 < l_2$, we have $\hat{\alpha}^{l_1,l_2}_{n_1,n_2} = \epsilon^{l_1,l_2}_{n_1,n_2} / \boldsymbol{z}^{l_2}_{n_2,up}$ if $\epsilon^{l_1,l_2}_{n_1,n_2} \geq 0$ and $\epsilon^{l_1,l_2}_{n_1,n_2} / \boldsymbol{z}^{l_2}_{n_2,low}$ otherwise; which results in the following score function:

$$s_{indir}(h^{l_1}_{n_1}) = \sum_{l_2=l_1+1}^{l-1} \sum_{n_2} \hat{\alpha}^{l_1,l_2}_{n_1,n_2} s(h^{l_2}_{n_2}) \qquad (2)$$
$$s(h^{l_1}_{n_1}) = s_{dir}(h^{l_1}_{n_1}) + s_{indir}(h^{l_1}_{n_1})$$

The score function as presented above can also be used to estimate the effects of splitting input nodes. Input nodes do not have an error, thus $s_{dir}(h^0_{n_1}) = 0$; however, they do have an indirect effect that can be estimated with a small modification to $\alpha^{l_1,l_2}_{n_1,n_2}$. Calculating the lower and upper bounds at node $h^{l_2}_{n_2}$ requires minimising and maximising the linear equation $q^{l_2}_{n_2}(\boldsymbol{x})$, respectively (See Section 2). So, if $c^{l_2}_{n_1,n_2}$ is the coefficient for $\boldsymbol{x}_{n_1}$ in $q^{l_2}_{n_2}(\boldsymbol{x})$, then it is reasonable to assume that by splitting $\boldsymbol{x}_{n_1}$ at $(\max(\boldsymbol{x}_{n_1}) + \min(\boldsymbol{x}_{n_1}))/2$ the input interval of node $h^{l_2}_{n_2}$ is reduced by $c^{l_2}_{n_1,n_2}(\max(\boldsymbol{x}_{n_1}) - \min(\boldsymbol{x}_{n_1}))/2$ in each branch. Thus, we use the estimate $\hat{\alpha}^{0,l_2}_{n_1,n_2} = c^{l_2}_{n_1,n_2}(\max(\boldsymbol{x}_{n_1}) - \min(\boldsymbol{x}_{n_1}))/2(\boldsymbol{z}^{l_2}_{n_2,up} - \boldsymbol{z}^{l_2}_{n_2,low})$.

In the next section we present an algorithm that uses the functions here described to improve the scalability of the verification problem.

4 Verification Algorithm

We now present a verification algorithm that extends previous symbolic interval propagation-based approaches [Wang *et al.*, 2018a; Henriksen and Lomuscio, 2020] by introducing three novel contributions. Firstly, the method considers both direct and indirect effects of a split in order to locate the most influential split candidates. Secondly, when compared to the current SoA encoding for the adaptive splitting paradigm [Henriksen and Lomuscio, 2020], splits are encoded in a way that reduces the search space for counterexamples. Thirdly, we combine the relaxation from RSIP and ESIP to produce a precise relaxation that is also suitable for the novel splitting score-function and split-encoding.

The procedure is outlined in Algorithm 1. It takes as input an FFNN f, input constraints ψ_{in} and output constraints ψ_{out}; the output is the result of the verification problem. The main loop consists of three phases.

The *symbolic interval propagation phase (lines 5 to 6)* of the algorithm uses RSIP and ESIP (see Section 2) to calculate a linear relaxation for each term $\hat{f}_{i,j}(\boldsymbol{x}) = \boldsymbol{a}^{\intercal}_{i,j} f(\boldsymbol{x}) + b_{i,j}$ in the output constraint $\psi_{out} = \bigwedge_i (\bigvee_j \boldsymbol{a}^{\intercal}_{i,j} f(\boldsymbol{x}) + b_{i,j} < 0)$. Notice that $\hat{f}_{i,j} : \mathbb{R}^n \to \mathbb{R}$ is an affine transformation of $f(\boldsymbol{x})$ so $\hat{f}_{i,j}$ is an FFNN in itself, and that the output constraint of the verification problem can be written as $\psi_{out} = \bigwedge_i (\bigvee_j \hat{f}_{i,j}(\boldsymbol{x}) < 0)$.

Algorithm 1 Verification Algorithm

1: $f, \psi_{in}, \psi_{out} \leftarrow$ FFNN, inputConstraint, outputConstraint
2: queue $\leftarrow$ [Branch(splitConstraints=None, bounds=None)]
3: **while** (not queue.isEmpty) **do**
4:     foundCandidateCex, branch $\leftarrow$ $False$, queue.pop()
5:     rsip $\leftarrow$ RSIP(f, ψ_{in}, branch)
6:     esip $\leftarrow$ ESIP(f, ψ_{in}, branch, rsip.bounds)
7:     **for** each clause ψ^j_{out} in ψ_{out} **do**
8:         candidateCex $\leftarrow$ LPSolver(esip, rsip, ψ_{in}, ψ^j_{out}, ψ_{split})
9:         **if** (candidateCex is not None) **then**
10:             foundCandidateCex = $True$
11:             cex $\leftarrow$ LocalSearch(candidateCex, f, ψ_{in})
12:             **if** (cex is not None) **then** return (UNSAFE, cex)
13:     **if** (foundCandidateCex) **then**
14:         splitNode $\leftarrow$ argmax(splitScore(f, ψ_{in}, ψ_{out}, esip)))
15:         queue.add(Branch(branch, splitNode ≥ 0, esip.bounds))
16:         queue.add(Branch(branch, splitNode ≤ 0, esip.bounds))
17: **return** SAFE

For each FFNN $\hat{f}_{i,j}$, the algorithm first runs RSIP, and then ESIP augmented with the bounds from RSIP. During ESIP, calculated bounds are compared with the bounds from RSIP on the fly and the tightest bounds are used to relax the ReLU nodes, resulting in a better relaxation than what can be achieved by ESIP alone.

Recall that RSIP produces linear bounds $y^{rsip}_{low}(\boldsymbol{x})_{i,j}$ and $y^{rsip}_{up}(\boldsymbol{x})_{i,j}$ such that $y^{rsip}_{low}(\boldsymbol{x})_{i,j} \leq \hat{f}_{i,j}(\boldsymbol{x}) \leq y^{rsip}_{up}(\boldsymbol{x})_{i,j}$ for all $\boldsymbol{x}$ satisfying ψ_{out}. In contrast, ESIP produces a linear equation $q(\boldsymbol{x})_{i,j}$ and concrete errors $\epsilon^n_{i,j}$ at the output node, such that $q(\boldsymbol{x})_{i,j} + \sum_{n | (\epsilon^n_{i,j} < 0)} \epsilon^n_{i,j} \leq \hat{f}_{i,j}(\boldsymbol{x}) \leq q(\boldsymbol{x})_{i,j} + \sum_{n | (\epsilon^n_{i,j} > 0)} \epsilon^n_{i,j}$ for all $\boldsymbol{x}$ satisfying ψ_{out}. Note that we here take $\epsilon^n_{i,j}$ to mean the errors at the output node of network $\hat{f}_{i,j}$ where n is an enumeration of all ReLU nodes in the network.

The *search phase (lines 7–12)* uses an LP-satisfiability call and a gradient descent-based local search in an attempt to locate a counterexample as first introduced in [Henriksen and Lomuscio, 2020]; however, our work differs in that we introduce a generalised version supporting any output constraint ψ_{out}, not just local robustness for classification problems.

A counterexample is an input $\boldsymbol{x}$ satisfying ψ_{in} that violates at least one of the clauses ψ^i_{out} from ψ_{out} or, equivalently, satisfies $\neg\psi^i_{out} = (\bigwedge_j \hat{f}_{i,j}(\boldsymbol{x}) \geq 0)$. Since $\hat{f}_{i,j}(\boldsymbol{x})$ is not linear, the relaxations from the previous phase are used to encode necessary conditions for $\neg\psi^i_{out}$.

The satisfiability call is performed for each clause ψ^i_{out} with the constraints $\{\bigwedge_j y^{rsip}_{up}(\boldsymbol{x})_{i,j} \geq 0, \bigwedge_j y^{esip}(\boldsymbol{x}, \boldsymbol{\sigma})_{i,j} \geq 0, \psi_{in}, \psi_{split}, (\boldsymbol{\sigma}_k \in [0,1])_{\forall k}\}$. Here, $y^{rsip}_{up}(\boldsymbol{x})_{i,j}$ is the upper bound produced by RSIP, ψ_{split} are the split constraints covered later in this section, and $y^{esip}(\boldsymbol{x}, \boldsymbol{\sigma})_{i,j} = q(\boldsymbol{x}) + \sum_n \boldsymbol{\sigma}^n \epsilon^n_{i,j}$ is a novel encoding of the ESIP bound introducing new auxiliary variables $\boldsymbol{\sigma}^n \in [0,1]$. Indeed, $\max_{\boldsymbol{\sigma}} y^{esip}_{i,j}(\boldsymbol{x}, \boldsymbol{\sigma})$ produces the same upper bound as in [Henriksen and Lomuscio, 2020]; however, we clarify below that $\boldsymbol{\sigma}$ can be further constrained during

splitting thus resulting in tighter bounds.

If the calls are unsatisfiable for all clauses, then the network is safe by Theorem 1. However, if an assignment $\boldsymbol{x}^i$ is found for at least one clause ψ^i_{out}, then this assignment is a potential counterexample. The candidate $\boldsymbol{x}^i$ is tested by evaluating $f(\boldsymbol{x}^i)$; if the result does not satisfy ψ_{out}, then $\boldsymbol{x}^i$ is a valid counterexample and the network is unsafe; otherwise, the algorithm proceeds to the local search.

The *local search phase (lines 11 to 12)* performs a gradient descent with the loss function $L(\boldsymbol{x}) = -\sum_j \boldsymbol{a}^\intercal_{i,j} f(\boldsymbol{x})$ for each potential counterexample $\boldsymbol{x}^i$ and corresponding clause $\psi^i_{out} = \bigvee_j \boldsymbol{a}^\intercal_{i,j} f(\boldsymbol{x}) + b_{i,j} \leq 0$. The gradient descent initialises at $\boldsymbol{x}^0 = \boldsymbol{x}^i$ and after each step the resulting $\boldsymbol{x}^k$ is clipped to the input box $\psi_{\boldsymbol{x}}$ and checked to see whether it is a valid counterexample. If a counterexample is found, then the network is unsafe under the given constraints; otherwise, the branch and bound phase is initiated.

The *branch and bound phase (lines 13 to 16)* starts by determining the next split with the method described in Section 3. A ReLU-split is performed by constraining the split node's input to ≤ 0 and ≥ 0 in a lower and upper branch, respectively. During ESIP and RSIP the ReLU constraints are handled trivially by only considering the corresponding linear parts of the ReLU; however, enforcing constraints in the LP-solver is somewhat more involved.

In the LP-solver, splitting a ReLU node with ESIP-errors ϵ^n and equation $q(\boldsymbol{x})$ is handled by adding the constraint $(z(\boldsymbol{x}, \boldsymbol{\sigma}) = q(\boldsymbol{x}) + \sum_n \boldsymbol{\sigma}^n \epsilon^n) \leq 0$ in the lower branch (upper branch ≥ 0, respectively) where $\boldsymbol{\sigma}^n \in [0, 1]$ are the auxiliary variables introduced in the search-phase. This is in contrast to previous algorithms that either split hierarchically from the first layer [Wang *et al.*, 2018a], so split nodes have no errors, or use a worst-case bound $(z(\boldsymbol{x}) = q(\boldsymbol{x}) + \sum_{n|\epsilon^n>0} \epsilon^n) \leq 0$ [Henriksen and Lomuscio, 2020]. The encoding here presented is always at least as succinct as the one in the latter reference, but it is usually smaller as the $\boldsymbol{\sigma}$ variables encapsulate dependencies between the constraints.

Splitting an input node $\boldsymbol{x}_k$ is performed by adjusting the split node's lower and upper bounds to $(\min(\boldsymbol{x}_k) + \max(\boldsymbol{x}_k))/2$ in ψ_{in} in the upper and lower branch, respectively. To ensure that the algorithm eventually splits all ReLUs, and thus is complete from Theorem 1, we only consider input nodes for splitting at branching-depths up to $d \in \mathbb{N}$ where d is an adjustable hyper-parameter.

After adding the new branch constraints, the main loop restarts with a few optimisations. Any clause ψ^i_{out} that was proven not to have a counterexample in the previous branch cannot admit a counterexample in the new, stronger constrained branch and is disregarded. Moreover, the bounds of ESIP and RSIP only change for layers after the layer containing the split node and are thus only recalculated for those layers. Finally, we perform a low-cost relaxation utilising only ESIP in the first branch; RSIP is applied in later branches.

We complete this section by showing that Algorithm 1, as presented, is sound and complete.

Theorem 1 (Soundness and Completeness)**.** *Let $\langle f, \psi_{in}, \psi_{out}\rangle$ be a verification problem as defined in Definition 1. Algorithm 1 returns "unsafe" iff there exists an $\boldsymbol{x}$ satisfying ψ_{in} that violates ψ_{out}. It returns "safe" iff ψ_{out} is satisfied for all $\boldsymbol{x}$ satisfying ψ_{in}*

Proof sketch. This result follows from the fact that (i) the linear program is sound in the sense that all counterexamples are satisfiable solutions to the program and (ii) the algorithm is guaranteed to terminate after splitting all ReLU nodes and exploring the corresponding branches.

5 Implementation and Experimental Results

We implemented the algorithm proposed in Section 4 in a Python toolkit DEEPSPLIT [Henriksen and Lomuscio, 2021].

For benchmarking we used three MNIST fully-connected, two CIFAR10 convolutional and two MNIST convolutional networks from the Verification of Neural Networks Competition (VNN-COMP) [VNN20, 2020]. The convolutional nets were originally published in [Balunovic and Vechev, 2020]), while the FC nets were introduced in the VNN-COMP.

The performance of DEEPSPLIT was evaluated against all CPU-based toolkits that finished in the top 3 in at least one of the categories (VERINET [Henriksen and Lomuscio, 2020], VENUS [Botoeva *et al.*, 2020], NNENUM [BAK, 2020], ERAN [Singh *et al.*, 2018; Singh *et al.*, 2019c]), as well as the MARABOU toolkit [Katz *et al.*, 2019]. The remaining SIP-based toolkits were not used as [Henriksen and Lomuscio, 2020] outperformed [Wang *et al.*, 2018a] and [Wang *et al.*, 2018a] outperformed [Wang *et al.*, 2018b] in experiments.

For all toolkits, we used the latest release of the implementations and ran them with the same settings as in the VNN-COMP for all toolkits with the exception of MARABOU and ERAN. MARABOU did not compete in VNN-COMP and we were unable to find the actual codebase that was used for ERAN in VNN-COMP; thus, we used the default settings for both. All toolkits are complete, except ERAN, which was used both in its complete and incomplete setting. However, note that incomplete and complete algorithms are not directly comparable, as discussed in the introduction. For the largest CIFAR10 network we reduced the ERAN incomplete LP and MILP timeouts per call to 1 second and the k-ReLU nodes to 2 due to runtimes of more than one hour per query.

While the results here are mostly in line with the VNN-COMP, note that each toolkit was run under different hardware in VNN-COMP, often with different results; in contrast, we here use the same experimental setup for all toolkits. Moreover, according to [VNN20, 2020], ERAN was augmented for the VNN-COMP with a PGD attack; this was not considered here for the reason above. Also, we limited the benchmarks to CPU-based toolkits. Because of this, we did not benchmark OVAL as it makes considerable use of GPUs.

The verification problem in all benchmarks concerns establishing whether the classification remains correct for all input perturbations within some l_∞-ball. We used the same inputs and perturbation radii as in the VNN-COMP (0.02 and 0.05 for the MNIST FC networks and 0.1, 0.3, 2/255 and 8/255 for the two MNIST and CIFAR10 convolutional networks, respectively). For the FC networks we also used an additional intermediate perturbation radius of 0.03, which had a more balanced number of safe and unsafe cases. All benchmarks

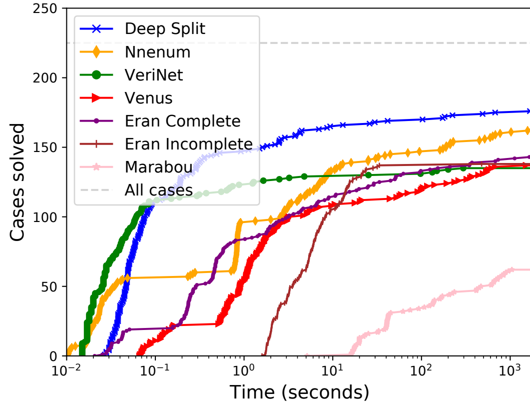

Figure 1: MNIST-FC NNs: cases solved as a function of time.

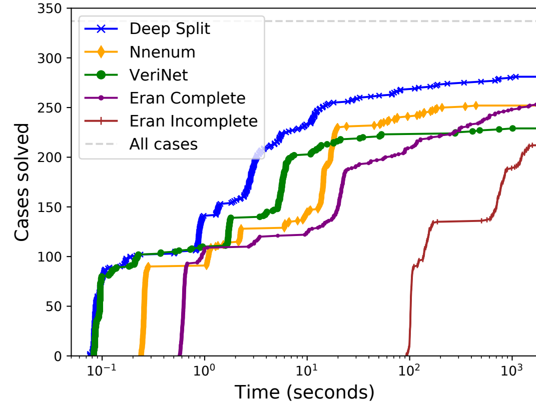

Figure 2: Convolutional NNs: cases solved as a function of time.

used a 1800 second timeout and were performed on a workstation with an Intel Core i9-10920X 3.5 GHz 12-core CPU, 128 GB Ram and Fedora 31 with Linux kernel 5.4.20.

Speedups reported below are calculated as follows: For toolkits A and B with n_A and n_B successfully verified cases, the speedup of toolkit A is $t^A_{n_A}/t^B_{n_A}$ if $n_A < n_B$ and $t^A_{n_B}/t^B_{n_B}$ otherwise; here $t^A_{n_A}, t^B_{n_A}$ is the total amount of time toolkit A and B used to verify their n_A fastest verified cases. Thus we compare the cases where each toolkit performed the best.

In the following, we present the results of the experiments.

Fully-connected networks (Figure 1). Compared to the two most performing toolkits, DEEPSPLIT achieved speedups of 90 and 157 times (VERINET, NNENUM) and had 21% and 41% fewer timeouts (NNENUM, ERAN-COMPLETE). DEEPSPLIT was also the most performing considering safe and unsafe cases in isolation; DEEPSPLIT verified 109 safe and 68 unsafe cases, while the second most performing toolkits verified 107 safe (NNENUM) and 66 unsafe cases (VERINET).

Convolutional networks (Figure 2). Compared to the two most performing toolkits, DEEPSPLIT achieved speedups of 7.4 and 12.3 times (NNENUM, VERINET) and had 34% and 35% fewer timeouts (ERAN-COMPLETE, NNENUM). ERAN-COMPLETE verified 251 safe cases while allowing long runtimes, whereas DEEPSPLIT only found 240 cases; however, ERAN-COMPLETE was among the slowest toolkits in general and only identified 3 unsafe cases, while DEEPSPLIT found 42. VERINET performed well on unsafe cases, also verifying 42; however, it only identified 188 safe-cases.

Ablation tests (Table 1). To determine the impact of the individual novel techniques proposed in this paper, we performed ablation experiments on the two-layer MNIST network. The results are reported in Table 1 for the following settings from top to bottom row: (i) all techniques enabled (ii) indirect score disabled (iii) RSIP disabled, only ESIP was used, (iv) worst-case encoding of split-constraints were used in the LP-Solver as discussed in Section 4. Each contribution resulted in significant overall gains. This is due to the fact that each method contributed to significantly reducing the number of branches that had to be explored during verification.

MNIST 256x2	n_s	$t(s)$	t/t_{All}	*Branches*
All	75	31.6	1	8643
Direct effect	75	429.9	13.6	174040
No RSIP	72	5568.3	176.2	1470569
Worst-case LP-encoding	75	184.73	5.85	82966

Table 1: **Ablation**. Columns report the number of cases solved, time used, speedup and mean number of branches explored, respectively.

6 Conclusions

SoA methods for the verification of neural networks at present do not scale to the network sizes that are commonly used in applications such as computer vision.

In this paper we proposed a complete symbolic interval propagation-based verification algorithm which extends previous approaches with several contributions. We introduced a novel technique to determine promising split candidates, succinct LP-encodings of the split-constraints and a relaxation technique combining two methods for symbolic interval propagation. The experiments demonstrate that these improvements lead to a speedup of 1-2 orders of magnitude over the current SoA for most networks.

In future work we plan to investigate how similar techniques can be used to improve the performance for networks with Sigmoid and Tanh activation functions.

Acknowledgements

This work is partly funded by the UKRI Centre for Doctoral Training in Safe and Trusted Artificial Intelligence (grant number EP/S023356/1). Alessio Lomuscio is supported by a Royal Academy of Engineering Chair in Emerging Technologies.

References

[Akintunde *et al.*, 2018] M. E. Akintunde, A. Lomuscio, L. Maganti, and E. Pirovano. Reachability analysis for neural agent-environment systems. In *KR18*, pages 184–193. AAAI Press, 2018.

[Akintunde *et al.*, 2019] M. E. Akintunde, A. Kevorchian, A. Lomuscio, and E. Pirovano. Verification of RNN-based neural agent-environment systems. In *AAAI19*, pages 6006–6013. AAAI Press, 2019.

[Akintunde *et al.*, 2020] M. E. Akintunde, E. Botoeva, P. Kouvaros, and A. Lomuscio. Verifying strategic abilities of neural-symbolic multi-agent systems. In *KR20*, pages 22–32. AAAI Press, 2020.

[Anderson *et al.*, 2020] R. Anderson, J. Huchette, W. Ma, C. Tjandraatmadja, and J.P. Vielma. Strong mixed-integer programming formulations for trained neural networks. In *Integer Programming and Combinatorial Optimization*, volume 11480 of *LNCS*, pages 27–42. Springer, 2020.

[Bak *et al.*, 2020] S. Bak, H.-D. Tran, K. Hobbs, and T.T. Johnson. Improved geometric path enumeration for verifying relu neural networks. In *CAV20*, volume 12224 of *LNCS*, pages 66–96. Springer, 2020.

[Bak, 2020] S. Bak. Execution-guided overapproximation (ego) for improving scalability of neural network verification. In *3rd International Workshop on Verification of Neural Networks*, 2020.

[Balunovic and Vechev, 2020] M. Balunovic and M. Vechev. Adversarial training and provable defenses: Bridging the gap. In *ICLR20*. OpenReview.net, 2020.

[Balunovic *et al.*, 2019] M. Balunovic, M. Baader, G. Singh, T. Gehr, and M. Vechev. Certifying geometric robustness of neural networks. In *NeurIPS19*, pages 15313–15323. Curran Associates, Inc., 2019.

[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*, pages 3291–3299. AAAI Press, 2020.

[Bunel *et al.*, 2020] R. Bunel, J. Lu, I. Turkaslan, P.H.S. Torr, P. Kohli, and M.P. Kumar. Branch and bound for piecewise linear neural network verification. *JMLR20*, 21(42):1–39, 2020.

[Goodfellow *et al.*, 2016] A. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT press Cambridge, 2016.

[Henriksen and Lomuscio, 2020] P. Henriksen and A. Lomuscio. Efficient neural network verification via adaptive refinement and adversarial search. In *ECAI20*, pages 2513–2520. IOS Press, 2020.

[Henriksen and Lomuscio, 2021] P. Henriksen and A. Lomuscio. DEEPSPLIT toolkit. https://vas.doc.ic.ac.uk/software/neural/, 2021. Release scheduled for: 2021-08-21.

[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*, volume 11561 of *LNCS*, pages 443–452. Springer, 2019.

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

[Liu *et al.*, 2019] C. Liu, T. Arnon, C. Lazaru, C. Barrett, and M.J. Kochenderfer. Algorithms for verifying deep neural networks. *arXiv preprint arXiv:1903.06758*, 2019.

[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.

[Lu and Kumar, 2020] J. Lu and M.P. Kumar. Neural network branching for neural network verification. In *ICLR20*. OpenReview.net, 2020.

[Rubies-Royo *et al.*, 2019] V. Rubies-Royo, R. Calandra, D. M. Stipanovic, and C. Tomlin. Fast neural network verification via shadow prices. *arXiv:1902.07247*, 2019.

[Singh *et al.*, 2018] G. Singh, T. Gehr, M. Mirman, M. Püschel, and M. Vechev. Fast and effective robustness certification. In *NeurIPS18*, pages 10802–10813. Curran Associates, Inc., 2018.

[Singh *et al.*, 2019a] G. Singh, R. Ganvir, M. Püschel, and M. Vechev. Beyond the single neuron convex barrier for neural network certification. In *NeurIPS19*, pages 15098–15109. Curran Associates, Inc., 2019.

[Singh *et al.*, 2019b] G. Singh, T. Gehr, M. Püschel, and M. Vechev. Boosting robustness certification of neural networks. In *ICLR19*. OpenReview.net, 2019.

[Singh *et al.*, 2019c] G. Singh, T. Gehr, M. Püschel, and P. Vechev. An abstract domain for certifying neural networks. In *ACM on Programming Languages*, volume 3, pages 1–30. ACM Press, 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.

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

[Tran *et al.*, 2020a] H.D. Tran, S. Bak, W. Xiang, and T.T. Johnson. Verification of deep convolutional neural networks using imagestars. In *CAV20*, volume 12224 of *LNCS*, pages 18–42. Springer, 2020.

[Tran *et al.*, 2020b] H.D. Tran, X. Yang, D.M. Lopez, P. Musau, L.V. Nguyen, W. Xiang, S. Bak, and T.T. Johnson. Nnv: The neural network verification tool for deep neural networks and learning-enabled cyber-physical systems. In *CAV20*, volume 12224 of *LNCS*, pages 3–17. Springer, 2020.

[VNN20, 2020] VNN20. Verification of neural networks competition. https://sites.google.com/view/vnn20/vnncomp, 2020. Accessed: 2020-12-01.

[Wang *et al.*, 2018a] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Efficient formal safety analysis of neural networks. In *NeurIPS18*, pages 6367–6377. Curran Associates, Inc., 2018.

[Wang *et al.*, 2018b] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Formal security analysis of neural networks using symbolic intervals. In *USENIX18*, pages 1599–1614. USENIX Association, 2018.