Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks

Rüdiger Ehlers
University of Bremen and DFKI GmbH, Bremen, Germany

We present an approach for the verification of feed-forward neural networks in which all nodes have a piece-wise linear activation function. Such networks are often used in deep learning and have been shown to be hard to verify for modern satisfiability modulo theory (SMT) and integer linear programming (ILP) solvers.

The starting point of our approach is the addition of a global linear approximation of the overall network behavior to the verification problem that helps with SMT-like reasoning over the network behavior. We present a specialized verification algorithm that employs this approximation in a search process in which it infers additional node phases for the non-linear nodes in the network from partial node phase assignments, similar to unit propagation in classical SAT solving. We also show how to infer additional conflict clauses and safe node fixtures from the results of the analysis steps performed during the search. The resulting approach is evaluated on collision avoidance and handwritten digit recognition case studies.

1 Introduction

Many tasks in computing are prohibitively difficult to formalize and thus hard to get right. A classical example is the recognition of digits from images. Formalizing what exactly distinguishes the digit *2* from a *7* is in a way that captures all common handwriting styles is so difficult that this task is normally left to the computer. A classical approach for doing so is to learn a *feed-forward neural network* from pre-classified example images. Since the advent of *deep learning* (see, e.g., [Sch15]), the artificial intelligence research community has learned a lot about engineering these networks, such that they nowadays achieve a very good classification precision and outperform human classifiers on some tasks, such as sketch recognition [YYS+15]. Even safety-critical applications such as obstacle detection in self-driving cars nowadays employ neural networks.

But if we do not have formal specifications, how can we assure the safety of such a system? The classical approach to tackle this problem is to construct *safety cases* [WK15]. In such a safety case, we characterize a set of environment conditions under which a certain output is desired and then test if the learned problem model ensures this output under all considered environment conditions. In a self-driving car scenario, we can define an abstract obstacle appearance model all of whose concretizations should be detected as obstacles. Likewise, in a character recognition application, we can define that all images that are *close* to a given example image (by some given metric) should be detected as the correct digit. The verification of safety cases somewhat deviates from the classical aim of formal methods to verify correct system behavior in all cases, but the latter is unrealistic due to the absence of a complete formal specification. Yet, having the means to test neural networks

on safety cases would help with certification and also provides valuable feedback to the system engineer.

Verifying formal properties of feed-forward neural networks is a challenging task. Pulina and Tacchella [PT10] present an approach for neurons with non-linear activation functions that only scales to small networks. In their work, they use networks with 6 nodes, which are far too few for most practical applications. They combine counterexample-triggered abstraction-refinement with *satisfiability modulo theory* (SMT) solving. Scheibler et al. [SWWB15] consider the bounded model checking problem for an inverse pendulum control scenario with non-linear system dynamics and non-linear neuron activation function, and despite employing the state-of-the-art SMT solver iSAT3 [SNM+16] and even extending this solver to deal better with the resulting problem instances, their experiments shows that the resulting verification problem is already challenging for neural networks with 26 nodes.

In *deep learning* [Sch15], many works use networks whose nodes have piece-wise linear activation functions. This choice has the advantage that they are more amenable to formal verification, for example using SMT solvers with the theory of linear real arithmetic, without the need to perform abstract interpretation. In such an approach, the solver chooses the *phases* of (some of) the nodes, and then applies a linear-programming-like sub-solver to check if there exist concrete real-valued inputs to the network such that all nodes have the selected phases. The node phases represent which part of the piece-wise linear activation functions are used for each node. It has been observed that the SMT instances stemming from such an encoding are very difficult to solve for modern SMT solvers, as they need to iterate through many such phase combinations before a problem instance is found to be satisfiable or unsatisfiable [KBD+17, PT12]. Due to the practical importance of verifying piecewise-linear feed-forward neural networks, this observation asks for a specialized approach for doing so.

Huang et al. [HKWW16] describe such an approach that is based on propagating constraints though the layers of a network. The constraints encode regions of the input space of each layer all of whose points lead to the same overall classification in the network. Their approach is partially based on discretization and focusses on robustness testing, i.e., determining the extent to which the input can be altered without changing the classification result. They do not support general verification properties. Katz et al. [KBD+17] provide an alternative approach that allows to check the input/output behavior of a neural network with linear and so-called *ReLU* nodes against convex specifications. Many modern network architectures employ these nodes. They present a modification of the *simplex algorithm* for solving linear programs that can also deal with the constraints imposed by ReLU nodes, and they show that their approach scales orders of magnitudes better than when applying the SMT solvers `MathSAT` or `Yices` on SMT instances generated from the verification problems.

Modern neural network architectures, especially those for image recognition, however often employ another type of neural network node that the approach by Katz et al. does not support: *MaxPool* nodes. They are used to determine the strongest signal from their input neurons, and they are crucial for *feature detection* in complex machine learning tasks. In order to support the verification of safety cases for machine learning applications that make use of this node type, it is thus important to have verification approaches that can operate on networks that have such nodes.

In this paper, we present an approach to verify neural networks with piece-wise linear activation functions against convex specifications. The approach supports all node types used in modern network network architectures that only employ piece-wise linear activation functions (such as MaxPool and ReLU nodes). The approach is based on combining satisfiability (SAT) solving and linear programming and employs a novel linear approximation of the overall network behavior. This approximation allows the approach to quickly rule out large search space parts for the node phases from being considered during the verification process. While the approximation can also be used as additional constraints in SMT solving and improves the computation times of the SMT

solver, we apply it in a customized solver that uses the *elastic filtering* algorithm from [CD91] for minimal infeasible linear constraint set finding in case of conflicts, and combine it with a specialized procedure for inferring implied node phases. Together, these components lead to much shorter verification times. We apply the approach on two cases studies, namely collision avoidance and character recognition, and report on experimental results. We also provide the resulting solver and the complete tool-chain to generate verifiable models using the Deep Learning framework `Caffe` [JSD+14] as open-source software.

2 Preliminaries

Feed-Forward Neural Networks: We consider *multi-layer* (*Perceptron*) networks with *linear*, *ReLU*, and *MaxPool* nodes in this paper. Such networks are formally defined as directed acyclic weighted graphs $G = (V, E, W, B, T)$, where V is a set of nodes, $E \subset V \times V$ is a set of edges, $W : E \to \mathbb{R}$ assigns a weight to each edge of the network, $B : V \to \mathbb{R}$ assigns a *node bias* to each node, and T assigns a *type* to each node in the network from some set of available types $\mathcal{T} \in \{input, linear, ReLU, MaxPool\}$. Nodes without incoming edges are called *input nodes*, and we assume that $T(v) = input$ for every such node v. Vertices that have no outgoing edge are also called *output nodes*.

A feed-forward neural network with n input nodes and m output nodes represents a function $f : \mathbb{R}^n \to \mathbb{R}^m$. Given assignments $in : \{1, \ldots, n\} \to V$ and $out : \{1, \ldots, m\} \to V$ to the input and output nodes, and some input vector $(x_1, \ldots, x_n)$, we can define the network's behavior by a node value assignment function $a : V \to \mathbb{R}$ that is defined as follows:

- For every node v with $T(v) = input$, we set $a(v) = in^{-1}(v)$,
- For every node v with $T(v) = linear$, we set $a(v) = \sum_{v' \in V, (v', v) \in E} W((v, v')) \cdot a(v') + B(v)$.
- For every node v with $T(v) = ReLU$, we set $a(v) = \min(B(v) + \sum_{v' \in V, (v', v) \in E} W((v, v')) \cdot a(v'), 0)$.
- For every node v with $T(v) = MaxPool$, we set $a(v) = \max_{v' \in V, (v', v) \in E} a(v')$.

Function f's output for $(x_1, \ldots, x_n)$ is defined to be $(a(out(1)), \ldots, a(out(m)))$. Note that the weights of the edges leading to *MaxPool* nodes and their bias values are not used in the above definition. Given a node value assignment function $a : V \to \mathbb{R}$, we also simply call $a(v)$ the *value* of v. If for a ReLU node v, we have $s(v) < 0$ for $s(v) = B(v) + \sum_{v' \in V, (v', v) \in E} W((v, v')) \cdot a(v') < 0$, and hence $a(v) = 0$, we say that node n is in the ≤ 0 phase, and for $s(v) \geq 0$, and hence $a(v) \geq 0$, we say that it is in the ≥ 0 phase. If we have $s(v) = 0$, then it can be in either phase. For a *MaxPool* node v, we define it to be in phase $e \in E \cap (V \times \{v\})$ if $a(v) = a(v')$ for $e = (v, v')$. If multiple nodes with edges from v' have the same values, the node can have any of the respective phases.

Modern neural network architectures are *layered*, i.e., we have that every path from an input node to an output node has the same length. For the verification techniques given in this paper, it does however not matter whether the network is layered. Networks can also be used to *classify* inputs. In such a case, the network represents a function $f' : \mathbb{R}^n \to \{1, \ldots, m\}$ (for some numbering of the classes), and we define $f'(x_1, \ldots, x_n) = \arg\max_{i \in \{1, \ldots, m\}} y_i$ for $(y_1, \ldots, y_m) = f(x_1, \ldots, x_n)$.

We do not discuss here how neural networks are learned, but assume networks to be given with all their edge weights and node bias values. Frameworks such as `Caffe` [JSD+14] provide ready-to-use functionality for learning edge weights and bias values from databases of examples, i.e., tuples $(x_1, \ldots, x_n, y_1, \ldots, y_m)$ such that the want the network to induce a function f with $(x_1, \ldots, x_n) = (y_1, \ldots, y_m)$. Likewise, for classification problems, the databases consist of tuples $(x_1, \ldots, x_n, c)$ such that we want the network to induce a function f' with $f'(x_1, \ldots, x'_n) = c$. When using a neural network learning tool, the architecture of the network, i.e., everything except for the weights and node bias values, it defined up-front, and the framework automatically derives suitable edge weights

and node bias values. There are other node types (such as *Softmax* nodes) that are used during the learning process, but removed before the deployment of the trained network, and hence do not need to be considered in this work. Also, there are network layer types such that *convolutional layers* that have special special structures. From a verification point of view, these are are however just sets of linear nodes whose edges share some weights, and thus do not have to be treated differently.

Satisfiability Solvers: Satisfiability (SAT) solvers check if a Boolean formula has a satisfying assignment. The formula is normally required to be in conjunctive normal form, and thus consists of *clauses* that are connected by *conjunction*. Every clause is a disjuction of one of more *literals*, which are Boolean variables or their negation. A SAT solver operates by successively building a valuation of the Boolean variables and *backtracking* whenever a conflict of the current *partial valuation* and a clause has been found. To achieve a better performance, SAT solvers furthermore perform *unit propagation*, where the partial assignment is extended by literals that are the only remaining ones not yet violated by the partial valuation in some clause. Additionally, modern solvers perform *clause learning*, where clauses that are implied by the conjunction of some other clauses are lazily inferred during the search process, and select variables to branch on using a *branching heuristic*. Most modern solvers also perform *random restarts*. For more details on SAT solving, the interested reader is referred to [FM09].

Linear Programming: Given a set of linear inequalities over real-valued variables and a linear optimization function (which together are called a *linear program*), the linear programming problem is to find an assignment to the variables that minimizes the objective function and fulfills all constraints. Even though linear programming has been shown to have polynomial-time complexity, it has been observed that in practice [KS08], it is often faster to apply the *Simplex algorithm*, which is an exponential-time algorithm.

Satisfiability Modulo Theory Solving: SAT solvers only support Boolean variables. For problems that can be naturally represented as a Boolean combination of constraints over other variable types, Satisfiability Modulo Theory (SMT) solvers are normally applied instead. An SMT solver combines a SAT solver with specialized decision procedures for other theories (such as, e.g., the theory of linear arithmetic over real numbers).

3 Efficient Verification of Feed-forward Neural Networks

In this paper, we deal with the following verification problem:

Definition 1 *Given a feed-forward neural network G that implements a function $f : \mathbb{R}^n \to \mathbb{R}^m$, and a set of linear constraints ψ over the real-valued variables $V = \{x_1, \ldots, x_n, y_1, \ldots, y_m\}$, the neural net (NN) verification problem is to find a node value assignment function a for V that fulfils ψ over the input and output nodes of G and for which we have $f(x_1, \ldots, x_n) = (y_1, \ldots, y_m)$, or to conclude that no such node value assignment function exists.*

The restriction to conjunctions of linear properties in Definition 1 was done for simplicity. Verifying arbitrary Boolean combinations of linear properties can be fitted into Definition 1 by encoding them into the structure of the network itself, so that an additional output neuron y_{add} outputs a value ≥ 0 if and only if the property is fulfilled. In this case, ψ is then simply $y_{add} \geq 0$.

There are multiple ways to solve the neural network (NN) verification problem. The encoding of an NN verification problem to an SMT problem instance is straight-forward, but yields instances that are difficult to solve even for modern SMT solvers (as the experiments reported on in Section 4

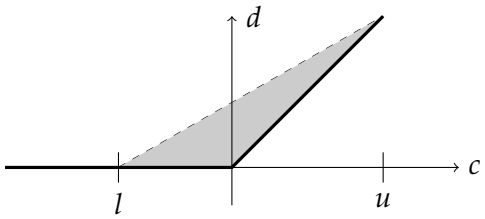


Figure 1: The activation function of a *ReLU* node, with a linear over-approximation drawn as filled area.

show). As an alternative, we present a new approach that combines 1) linear approximation of the overall NN behavior, 2) irreducible infeasible subset analysis for linear constraints based on elastic filtering [CD91], 3) inferring possible safe node phase choices from feasibility checking of partial node phase valuations, and 4) performing unit-propagation-like reasoning on node phases. We describe these ideas in this section, and present experimental results on a tool implementing them in the next section.

Starting point is the combination of a linear programming solver and a satisfiability solver. We let the satisfiability solver guide the search process. It determines the phases of the nodes and maintains a set of constraints over node phase combinations. On a technical level, we allocate the SAT variables $x_{(v,0)}$ and $x_{(v,1)}$ for every ReLU node v, and also reserve variables $x_{(v,e)}$ for every MaxPool node v and every edge e ending in v. The SAT solver performs unit propagation, clause learning, branching, and backtracking as usual, but whenever the solver is about to branch, we employ a linear programming solver to check a linear approximation of the network behavior (under the node phases already fixed) for feasibility. Whenever a conflict is detected, the SAT solver can then learn a conflict clause. Additionally, we infer implied node phases in the search process.

We describe the components of our approach in this section, and show how they are combined at the end of it.

3.1 Linear Approximation of Neural Network Value Assignment Functions

Let $G = (V, E, W, B, T)$ be a network representing a function $f : \mathbb{R}^n \to \mathbb{R}^m$. We want to build a system of linear constraints using V as the set of variables that closely approximates f, i.e., such that every node value assignment function a is a correct solution to the linear constraint system, and the constraints are as tight as possible. The main difficulty in building such a constraint system is that the *ReLU* and *MaxPool* nodes do not have linear input-output behavior (until their phases are fixed), so we have to approximate them linearly.

Figure 1 shows the activation function of a *ReLU* node, where we denote the weighted sum of the input signals to the node (and its bias) as variable c. The output of the node is denoted using the variable d. If we have upper and lower bounds $[l, u]$ of c, then we can approximate the relationship between c and d by the constraints $d \geq 0$, $d \geq c$, and $d \leq \frac{u \cdot (c-l)}{u-l}$, all of which are linear equations for constant u and l. This yields the set of allowed value combinations for c and d drawn as the filled area in Figure 1.

Obviously, this approach requires that we know upper and lower bounds on c. However, even though neural networks are defined as functions from $\mathbb{R}^n$, bounds on the input values are typically known. For example, in image processing networks, we know that the input neurons receive values from the range $[0, 1]$. In other networks, it is common to *normalize* the input values before learning the network, i.e., to scale them to the same interval or to $[-1, 1]$. This allows us to use classical *interval arithmetic* on the network to obtain basic lower and upper bounds $[l, u]$ on every node's values.

For the case of *MaxPool* nodes, we can approximate the behavior of the nodes linearly similarly to

the ReLU case, except that we do not need upper bounds for the nodes' values. Let $c_1, \ldots, c_k$ be the values of nodes with edges leading to the *MaxPool* node, $l_1, \ldots, k_k$ be their lower bounds, and d be the output value of the node. We instantiate the following linear constraints:

$$\bigwedge_{i \in \{1,\ldots,k\}} (d \geq c_i) \quad \wedge \quad (c_1 + \ldots + c_k \geq d + \sum_{i \in \{1,\ldots,k\}} l_i - \max_{i \in \{1,\ldots,k\}} l_i)$$

Note that these are the tightest linear constraints that can be given for the relationship between the values of the predecessor nodes of a *MaxPool* node and the node value of the *MaxPool* node itself.

After a linear program that approximates the behavior of the overall network has been built, we can use it to make all future approximations even tighter. To achieve this, we add the problem specification ψ as constraints and solve, for every variable $v \in V$, the resulting linear program while minimizing first for the objective functions $1 \cdot v$, and then doing the same for the objective function $-1 \cdot v$. This yields new tighter lower and upper bounds $[l, u]$ for each node (if the network has any ReLU nodes), which can be used to obtain a tighter linear program. Including the specification in the process allows us to derive tighter bounds than we would have found without the specification. The whole process can be repeated several times: whenever new upper and lower bounds have been obtained, they can be used to build a tighter linear network approximation, which in turn allows to obtain new tighter upper and lower bounds.

3.2 Search process and Infeasible Subset Finding

Given a phase fixture for all ReLU and MaxPool nodes in a network, checking if there exists a node value assignment function with these phases (and such that the verification constraint ψ is fulfilled) can be reduced to a linear programming problem. For this, we extend the linear program built with the approach from the previous subsection (with V as the variable set for the node values) by the following constraints:

- For every phase 0 selected for a ReLU node v, we add the constraints $v = 0$ and $\sum_{(v',v) \in E} W((v', v)) \cdot v' + B(v) \leq 0$.
- For every phase 1 selected for a ReLU node v, we add the constraint $v \geq \sum_{(v',v) \in E} W((v', v)) \cdot v' + B(v)$.
- For every phase (v', v) selected for a MaxPool node v, we add the constraint $v = v'$.

If we only have a partial node phase selection, we add these constraints only for the fixed nodes. If the resulting linear program is infeasible, then we can discard all extensions to the partial valuation from consideration in the search process. This is done by adding a *conflict* clause that rules out the Boolean encoding of this partial node phase selection, so that even after restarts of the solver, the reason for infeasibility is retained.

However, the reasons for conflicts often involve relatively few nodes, so shorter conflict clauses can also be learned instead (which makes the search process more efficient). To achieve this, we employ *elastic filtering* [CD91]. In this approach, all of the constraints added due to node phase selection are weakened by *slack variables*, where there is one slack variable for each node. So, for example a constraint $\sum_{(v',v) \in E} W((v', v)) \cdot v' + B(v) \leq 0$ becomes $\sum_{(v',v) \in E} W((v', v)) \cdot v' + B(v) - s_v \leq 0$. When running the linear programming solver again with the task of minimizing a weighted sum of the slack variables, we get a ranking of the nodes by how much they contributed to the conflict, where some of them did not contribute at all (since their slack variable had a 0 value). We then fix the slack variable with the highest value to be 0, hence making the corresponding constraints strict, and repeat the search process until the resulting LP instance becomes infeasible. We then know that the node phase fixtures that were made strict during this process are together already infeasible,

and build conflict clauses that only contain them. We observed that these conflict clauses are much shorter than without applying elastic filtering.

Satisfiability modulo theory solvers typically employ cheaper procedures to compute *minimal infeasible subsets* of linear constraints, such as the one by Duterte and de Moura [DdM06], but the high number of constraints in the linear approximation of the network behavior that are independent of node phase selections seems to make the approach less well-suited, as our experiments with the SMT solver `Yices` that uses this approach suggest.

3.3 Implied Node Phase Inference during Partial Phase Fixture Checking

In the partial node fixture feasibility checking step from in Section 3.2, we employ a linear programming solver. However, except for the elastic filtering step, we did not employ an optimization function yet, as it was not needed for checking the feasibility of a partial node fixture.

For the common case that the partial node fixture *is* feasible (in the linear approximation), we define an optimization function that allows us to infer additional infeasible *and* feasible partial node fixtures when checking some other partial node fixture for feasibility. The feasible fixtures are cached so that if it or a partial fixture of it later evaluated, no linear programming has to be performed. Given a partial node fixture to the nodes $V' \subset V$, we use $-1 \cdot \sum_{v \in V \setminus V', T(v)=ReLU} v - \frac{1}{10} \sum_{v \in V \setminus V', T(v)=MaxPool} v$ as optimization function. This choice asks the linear programming solver to minimize the error for the ReLU nodes, i.e, the difference between $a(v)$ and $\min(\sum_{v' \in V,(v',v) \in E} W((v,v')) \cdot a(v') + B(v), 0)$ for every assignment a computed in the linear approximation of the network behavior and every ReLU-node v. While this choice only minimizes an approximation of the error sum of the nodes and thus does not guarantee that the resulting variable valuation denotes a valid node value assignment function, it often yields assignments in which a substantial number of nodes v *have* a tight value, i.e., have $a(v) = \min(\sum_{v' \in V,(v',v) \in E} W((v,v')) \cdot a(v') + B(v), 0)$.

If tight is the set of nodes with tight values, p is the partial SAT solver variable valuation that excodes the phase fixtures for the nodes V', and if p' is the (partial) valuation of the SAT variables that encodes the phases of the tight nodes, we can then cache that $p \cup p'$ is a partial assignment that is feasible in the linear approximation. So when the SAT solver adds literals from p' to the partial valuation, there is no need to let the linear programming solver run again.

At the same time, the valuation a (in the linear approximation) can be used to derive an additional clause for the SAT solver. Let $\mathsf{unfixed}$ be the ReLU nodes whose values are not fixed by p. If for any node $v \in \mathsf{unfixed}$, we have $a(v) > 0$, then we know by the choice of optimization function and the fact that we performed the analysis in a linear approximation of the network behavior, that some node in v needs to be in the ≥ 0 phase (under the partial valuation p). Thus, we can learn the additional clause $\{\neg v \mid v \in p\} \cup \{(v,1) \mapsto \textbf{true} \mid T(v) = ReLU, (v,0) \notin p\}$ for the SAT solver, provided that the values of the MaxPool nodes are valid, i.e, for all MaxPool nodes v we have $a(v) = a(v')$ for some $(v,v') \in E$. This last restriction is why we also included the MaxPool nodes in the optimization function above (but with lower weight).

3.4 Detecting Implied Phases

Whenever the SAT solver has fixed a new node phase, the selected phases together may imply other node phases. Take for example the net excerpt from Figure 2. There are two ReLU nodes, named r_1 and r_2, and one MaxPool node. Assume that during the initial analysis of the network (Section 3.1), it has been determined that the value of node r_1 is between 0.4 and 1.5, and the value of node r_2 is between 0.1 and 2.0. First of all, the SAT solver can unconditionally detect that node r_2 is in the ≥ 0 phase. Then, if at some point, the SAT solver decides that node r_1 should be in the ≤ 0 phase, this fixes the value of r_1 to 0. Since the flow out of r_2 has a lower bound > 0, we can then deduce that m's phase should be set to (r_2, m).

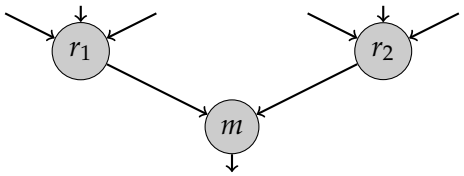


Figure 2: An example neural network part, used in Subsection 3.4.

Similar reasoning can also be performed for flow leading into a node. If we assume that the analysis of the initial linear approximation of the network's node functions yields that the outgoing flow of m needs to be between 0.5 and 0.7, and the phase of r_1 is chosen to be ≤ 0, then this implies that the phase of r_2 must be ≥ 0, as otherwise m would be unable to supply a flow of > 0.

Both cases can be detected without analyzing the linear approximation of the network. Rather, we can just propagate the lower and upper bounds on the nodes' outgoing flows through the network and detect implied phases. Doing so can be done in time linear in the size of the network, which is considerably faster than making an LP solver call. This allows the detection of implied phases to be applied in a way similar to classical unit propagation in SAT solving: whenever a decision has been made by the solver, we run implied phase detection to extend the partial valuation of the SAT solver by implied choices (which allows to make the linear approximation tighter for the following partial node fixture feasibility checks).

3.5 Overview of the Integrated Solver

To conclude this section, let us discuss how the techniques presented in it are combined. Algorithm 1 shows the overall approach. In the first step, upper and lower bounds for all nodes' values are computed. The solver then prepares an empty partial valuation to the SAT variables, and an empty list *extra* in which additional clauses generated by the LP instance analysis steps proposed in this section are stored. The SAT instance is initialized with clauses that enforce that every ReLU node and every MaxPool node has exactly one phase selected (using a *one-hot encoding*).

In the main loop of the algorithm, the first step is to perform most steps of SAT solving, such as unit propagation, conflict detection & analysis, and others. We assume that the partial valuation is always labelled by *decision levels* so that backtracking can also be performed whenever needed. Furthermore, additional clauses from *extra* are mixed to the SAT instance ψ. This is done on a step-by-step basis, as the additional clauses may trigger unit propagation and even conflicts, which need to be dealt with eagerly. After all clauses from *extra* have been mixed into ψ, and possibly the partial valuation p has been extended by implied literals, in line 11, the approach presented in Sect. 3.4 is applied. If it returns new implied literals (in the form of additional clauses), they are taken care of by the SAT solving steps in line 10 next. This is because the clauses already in ψ may lead to unit propagation on the newly inferred literals, which makes sense to check as every additional literal makes the linear approximation of the network behavior tighter (and can lead to additional implied literals being detected). Only when all node phases have been inferred, p is checked for feasibility in the linear approximation (line 13).

There are two different outcomes of this check: if the LP instance is infeasible, a new conflict clause is generated, and hence the condition in line 14 is not satisfied. The algorithm then continues in line 10 in this case. Otherwise, the branching step of the SAT solver is executed. If p is already a complete valuation, we know at this point that the instance is *satisfiable*, as then the CheckForFeasibility function just executed operated on an LP problem that is not approximate, but rather captures the precise behavior of the network. Otherwise, p is extended by a decision to set a variable b to **true** (for some variable chosen by the SAT solver's variable selection heuristics). Whenever this happens, we employ a plain SAT solver for checking if the partial valuation can be extended to one that satisfies

Algorithm 1 Top-level view onto the neural network verification algorithm.

1: **function** VERIFYNN(V, E, T, B, W)
2: $(\overrightarrow{min}, \overrightarrow{max}) \leftarrow$ ComputeInitialBounds(V, E, T, B, W) ▷ Section 3.1
3: $(\overrightarrow{min}, \overrightarrow{max}) \leftarrow$ RefineBounds($V, E, T, B, W, \overrightarrow{min}, \overrightarrow{max}$) ▷ Section 3.1
4: $p \leftarrow \emptyset$, $extra \leftarrow \emptyset$
5: $\psi \leftarrow \bigwedge_{v \in V, T(v)=MaxPool} \bigvee_{v' \in V,(v',v) \in E} x_{v,(v',v)}$
6: $\psi \leftarrow \psi \wedge \bigwedge_{v \in V, T(v)=MaxPool, v', v'' \in V, v' \neq v'', (v',v) \in E, (v'',v) \in E} (\neg x_{v,(v',v)} \vee \neg x_{v,(v'',v)})$
7: $\psi \leftarrow \psi \wedge \bigwedge_{v \in V, T(v)=ReLU} (x_{v,0} \vee x_{v,1}) \wedge (\neg x_{v,0} \vee \neg x_{v,1})$
8: **while** ψ has a satisfying assignment **do**
9: **while** *extra* is non-empty **do**
10: Perform unit propagation, conflict detection, backtracking, and clause learning for p on ψ, while moving the clauses from *extra* to ψ one-by-one.
11: $extra \leftarrow$ InferNodePhases($V, E, T, B, W, p, \overrightarrow{min}, \overrightarrow{max}$) ▷ Section 3.4
12: **if** $extra = \emptyset$ **then**
13: $extra \leftarrow$ CheckForFeasibility($V, E, T, B, W, p, \overrightarrow{min}, \overrightarrow{max}$) ▷ Section 3.2 and 3.3
14: **if** $p \models c$ for all clauses $c \in extra$ **then**
15: **if** p is a complete assignment to all variables **then**
16: **return Satisfiable**
17: Add a new variable assignment $b \mapsto$ **true** for $b \in B$ to p.
18: **if** p cannot be extended to a satisfying valuation to ψ **then**
19: $p = p \setminus \{b \mapsto \textbf{true}\} \cup \{b \mapsto \textbf{false}\}$
20: **return Unsatisfiable**

ψ. This not being the case may not be detected by unit propagation in line 10 and hence it makes sense to do an eager SAT check. In case of conflict, the choice of b's value is inverted, and in any case, the algorithm continues with the search.

4 Experiments

We implemented the approach presented in the preceding section in a tool called `Planet`. It is written in C++ and bases on the linear programming toolkit `GLPK` 4.61[1] and the SAT solver `Minisat 2.2.0` [ES03]. While we use `GLPK` as it is, we modified the main search procedure of `Minisat` to implement Algorithm 1. We repeat the initial approximation tightening process from Section 3.1 until the cumulative changes in $\overrightarrow{min}$ and $\overrightarrow{max}$ fall below 1.0. We also abort the process if 5000 node approximation updates have been performed (to not spend too much time in the process for very large nets), provided that the bounds for every node have been updated at least three times.

All numerical computations are performed with `double` precision, and we did not use any compensation for numerical imprecision in the code apart from using a fixed safety margin $\epsilon = 0.0001$ for detecting node assignment values $a(v)$ to be greater or smaller than other node assignment values $a(v')$, i.e., we actually check if $a(v) \leq a(v') - \epsilon$ to conclude $a(v) \leq a(v')$, whenever such a comparison is made in the verification algorithm steps described in Sect. 3.2 and Sect. 3.3. Since the neural networks learned using the `Caffe` [JSD+14] deep learning framework (which we employ for our experiments in this paper) tend not to degenerate in the node weights, this is sufficient for the experimental evaluation in this paper. Also, we did not observe any differences in the verification results between the SMT solver `Yices` [Dut14] on the SMT instances that we computed from the verification problems and the results computed by our tool. The tool is available under GPLv3

[1] GNU Linear Programming Kit, `http://www.gnu.org/software/glpk/glpk.html`

license and can be obtained from `https://github.com/progirep/planet` along with all scripts & configuration files needed to learn the neural networks used in our experiments with the `Caffe` framework and to translate them to input files for our tool.

All computation times given in the following were obtained on a computer with Intel Core i5-4200U 1.60 GHz CPU and 8 GB of RAM running an x64 version of GNU/Linux. We do not report memory usage, as it was always < 1 GB. All tools run with a single computation thread.

4.1 Collision Avoidance

As a first example, we consider the problem of predicting collisions between two vehicles that follow curved paths at different speeds. We learned a neural network that processes tuples (x, y, s, d, c_1, c_2) and classifies them into whether they represent a colliding or non-colliding case. In such a tuple,

- the x and y components represent the relative distances of the vehicles in their workspace in the X- and Y-dimensions,
- the speed of the second vehicle is s,
- the starting direction of the second vehicle is d, and
- the rotation speed values of the two vehicles are c_1 and c_2.

The data is given in normalized (scaled) form to the neural network learner, so that all tuple components are between 0 and 1 (or between -1 and 1 for c_1 and c_2). We wrote a tool that generates a few random tuples (within some intervals of possible values) along with whether they represent a colliding or non-colliding case, as determined by simulation. The vehicles are circle-shaped, and we defined a safety margin and only consider tuples for which either the safety margins around the vehicles never overlap, or the vehicles themselves collide. So when only the safety margins overlap, this represents a "don't care" case for the learner. The tool also visualizes the cases, and we show two example traces in Figure 3. The tool ensures that the number of colliding cases and non-colliding ones are the same in the case list given to the NN learner (by discarding tuples whenever needed). We generated 3000 tuples in total as input for `Planet`.

We defined a neural network architecture that consists of 40 linear nodes in the first layer, followed by a layer of MapPool nodes, each having 4 input edges, followed by a layer of 19 ReLU nodes, and 2 ReLU nodes for the output layer. Since `Caffe` employs randomization to initialize the node weights, the accuracy of the computed network is not constant. In 86 out of 100 tries, we were able to learn a network with an accuracy of 100%, i.e., that classifies all example tuples correctly.

We want to find out the *safety margin* around the tuples, i.e., a the highest value of $\epsilon > 0$ such that for every tuple (x, y, s, d, c_1, c_2) that is classified to $b \in \{colliding, notCol\text{-}liding\}$, we have that all other tuples $(x \pm \epsilon, y \pm \epsilon, s \pm \epsilon, d \pm \epsilon, c_1 \pm \epsilon, c_2 \pm \epsilon)$ are classified to b by the network as well. We perform this check for the first 100 tuples in the list, use *bisection search* to test this for $\epsilon \in [0, 0.05]$, and abort the search process if ϵ has been determined with a precision of 0.002.

We obtained 500 NN verification problem instances from this safety margin exploration process. Figure 4 shows the distribution of the computation times of our tool on the problem instances, with a timeout of 1 hour. For comparison, we show the computation times of the SMT solver `Yices 2.5.2` and the (I)LP solver `Gurobi 7.02` on the problem instances. The SMT solver `z3` was observed to perform much worse than `Yices` on the verification problems, and is thus not shown. The choice of these comparison solvers was rooted in the fact that they performed best for verifying networks without MaxPool nodes in [KBD+17]. We also give computation times for `Gurobi` and `Yices` after adding additional linear approximation constraints obtained with the approach in Section 3.1. The computation times include the time to obtain them with our tool.

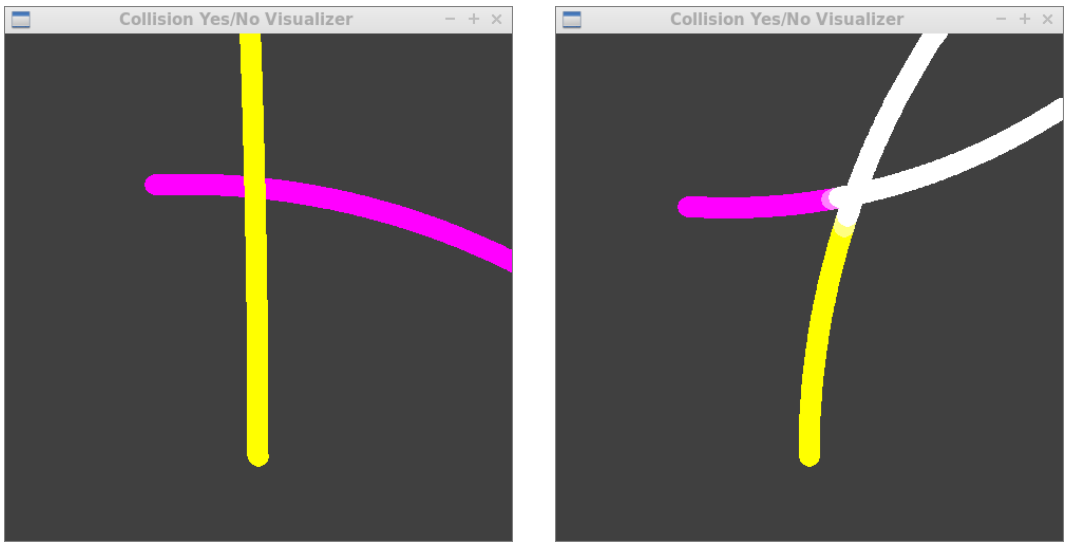


Figure 3: Two pairs of vehicle trajectories, where the first one is non-colliding, and the second one is colliding. The lower vehicle starts in roughly in north direction, whereas the other one starts roughly in east direction. The first one is non-colliding as the to vehicles pass through the trajectory intersection point at different times.

It can be observed that the computation times of `Gurobi` and `Yices` are too long for practical verification, except if the linear approximation constraints from our approach in this paper are added to the SMT and ILP instances to help the solvers. While `Yices` is then still slower than our approach, `Gurobi` actually becomes a bit faster in most cases, which is not surprising, given that it is a highly optimized commercial product that employs many sophisticated heuristics under-the-hood, whereas we use the less optimized `GLPK` linear programming framework. `Planet` spends most time on LP solving.

4.2 MNIST Digit Recognition

As a second case study, we consider handwritten digit recognition. This is a classical problem in machine learning, and the MNIST dataset [LC09] is the most commonly used benchmark for comparing different machine learning approaches. The `Caffe` framework comes with some example architectures, and we use a simplified version of Caffe's version of the `lenet` network [LBBH98] for our experiments. The `Caffe` version differs from the original network in that is has piecewise linear node activation functions.

Figure 5 (a)-(b) shows some example digits from the MNIST dataset. All images are in gray-scale and have 28×28 pixels. Our simplified network uses the following layers:

- One input layer with 28×28 nodes,
- one convolutional network layer with $3 \times 13 \times 13$ nodes, where every node has 16 incoming edges,
- one pooling layer with $3 \times 4 \times 4$ nodes, where each node has 16 incoming edges,
- one ReLU layer with 8 nodes, and
- one ReLU output layer with 10 nodes

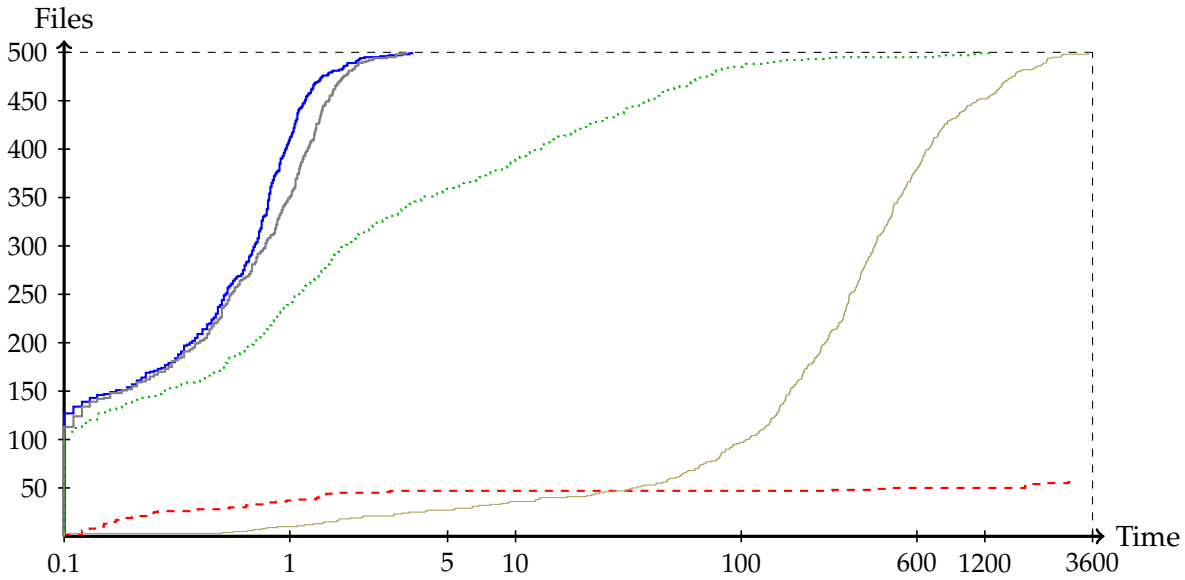


Figure 4: Cactus plot of the solver time comparison for the 500 vehicle collision benchmarks. Time is given in seconds (on a log-scale), and the lines, from bottom right to top left, represent `Gurobi` without linear approximation (dashed), `Yices` without linear approximation (solid), `Yices` with linear approximation (dotted), `Planet` (solid), and `Gurobi` with linear approximation (solid).

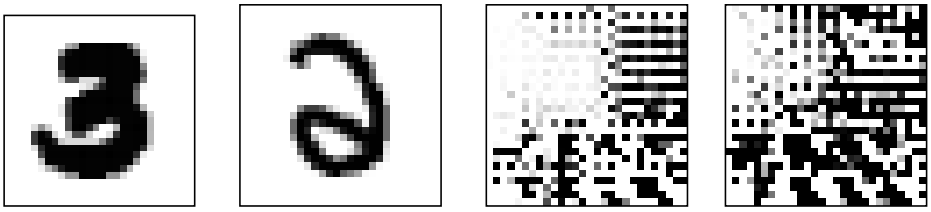
The ReLU layers are fully connected. Overall, the network has 1341 nodes, the search space for the node phases is of size $16^{3 \cdot 4 \cdot 4} \cdot 2^8 \cdot 2^{10} = 2^{162}$, and the network has 9344 edges.

We used this architecture to learn a network from the 100000 training images of the dataset, and the resulting network has an accuracy of 95.05% on a separate testing dataset. Note that an accuracy of 100% cannot be expected from any machine learning technique, as the dataset also contains digits that are even hardly identifiable for humans (as shown in Figure 5(b)).

We performed a few tests with the resulting network. First we wanted to see an input image that is classified strongly as a 2. More formally, we wanted to obtain an input image $(x_{1,1}, \ldots, x_{28,28})$ for which the network outputs a vector $(y_0, \ldots, y_9)$ for which $y_2 \geq y_i + \delta$ for all $i \in \{0, 1, 3, 4, 5, 6, 7, 8, 9\}$ for a large value of δ. We found that for values of $\delta = 20$ and $\delta = 30$, such images can be found in 4 minutes 25 seconds and 32 minutes 35 seconds, respectively. The two images are shown in Figure 5(c) and Figure 5(d). For $\delta = 50$, no such image can be found (4 minutes 41 seconds of computation time), but for $\delta = 35$ on the other hand, `Planet` times out after 4 hours. `Gurobi` (with the added linear approximation constraints) could not find a solution in this time frame, either.

Then, we are interested in how much noise can be added to images before they are not categorized correctly anymore. We start with the digit given in Figure 5(a), which is correctly categorized by the learned network as digit 3. We ask whether there is another image that is categorized as a 4, but for which each pixel has values that are within an absolute range of ±8% of color intensity of the original image's pixels, where we keep the pixels the same that are at most three pixels away from the boundaries. To determine that this is not the case, `planet` requires 1 minutes 46.8 seconds. For a range of ±0.12, `planet` times out after four hours. The output of `planet` shows that long conflict clauses are learned in the process, which suggests that we applied it to a difficult verification problem.

Let us now consider an error model that captures noise that is likely to occur in practice (e.g., due to stains on scanned paper). It excludes sharp noise edges such as the ones in Figure 5(b). Instead of restricting the amplitude of noise, we restrict the noise value differences in adjacent pixels to be



(a) '3' digit from the MNIST dataset

(b) '2' digit from the MNIST dataset

(c) Image classified as digit 2 with $\delta = 20$

(d) Image classified as digit 2 with $\delta = 30$

Figure 5: Example digit images from Section 4.2

≤ 0.05 (i.e., 5% of color density). This constraint essentially states that the noise must pass through a linearized low-pass filter unmodified. We still exclude the pixels from the image boundaries from being modified. Our tool concludes in 9 minutes 2.4 seconds that the network never misclassifies the image from Figure 5(a) as a 4 under this noise model. Since the model allows many pixels to have large deviations, we can see that including a linear noise model can improve the computation time of `planet`.

5 Conclusion

In this paper, we presented a new approach for the verification of feed-forward neural networks with piece-wise linear activation functions. Our main idea was to generate a linear approximation of the overall network behavior that can be added to SMT or ILP instances which encode neural network verification problems, and to use the approximation in a specialized approach that features multiple additional techniques geared towards neural network verification, which are grouped around a SAT solver for choosing the node phases in the network. We considered two case studies from different application domains. The approach allows arbitrary convex verification conditions, and we used them to define a noise model for testing the robustness of a network for recognizing handwritten digits.

We make the approach presented in this paper available as open-source software in the hope that it fosters the co-development of neural network verification tools and neural network architectures that are easier to verify. While our approach is limited to network types in which all components have piece-wise linear activation functions, they are often used in modern network architectures anyway. But even if more advanced activation functions such as *exponential linear units* [CUH15] shall be used in learning, they can still be applied to learn an initial model, which is then linearly approximated with ReLU nodes and fine-tuned by an additional learning process. The final model is then easier to verify. Such a modification of the network architecture during the learning process is not commonly applied in the artificial intelligence community yet, but while verification becomes more practical, this may change in the future.

Despite the improvement in neural network verification performance reported in this paper, there is still a lot to be done on the verification side: we currently do not employ specialized heuristics for node phase branching selection, and while our approach increases the scalablity of neural network verification substantially, we observed it to still be quite fragile and prone to timeouts for difficult verification properties (as we saw in the MNIST example). Also, we had to simplify the LeNet architecture for digit recognition in our experiments, as the original net is so large that even obtaining a lower bound for a single variable in the network (which we do for all network nodes before starting the actual solution process as explained in Section 3.1) takes more than 30 minutes

otherwise, even though this only means solving a single linear program. While the approach by Huang et al. [HKWW16] does not suffer from this limitation, it cannot handle general verification properties, which we believe to be important. We plan to work on tackling the network size limitation of the approach presented in this paper in the future.

Acknowledgements

This work was partially funded by the Institutional Strategy of the University of Bremen, funded by the German Excellence Initiative.

References

[CD91] John W. Chinneck and Erik W. Dravnieks. Locating minimal infeasible constraint sets in linear programs. *INFORMS Journal on Computing*, 3(2):157–168, 1991.

[CUH15] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (ELUs). *CoRR*, abs/1511.07289, 2015.

[DdM06] Bruno Dutertre and Leonardo Mendonça de Moura. A fast linear-arithmetic solver for DPLL(T). In *18th International Conference on Computer Aided Verification (CAV)*, pages 81–94, 2006.

[Dut14] Bruno Dutertre. Yices 2.2. In *26th International Conference on Computer Aided Verification (CAV)*, pages 737–744, 2014.

[ES03] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, pages 502–518, 2003.

[FM09] John Franco and John Martin. *A History of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 1, pages 3–74. IOS Press, February 2009.

[HKWW16] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety verification of deep neural networks. *CoRR*, abs/1610.06940, 2016.

[JSD+14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[KBD+17] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. *CoRR*, abs/1702.01135, 2017.

[KS08] Daniel Kroening and Ofer Strichman. *Decision Procedures – An Algorithmic Point of View*. Springer, 2008.

[LBBH98] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998.

[LC09] Yann Lecun and Corinna Cortes. The MNIST database of handwritten digits. 2009.

[PT10] Luca Pulina and Armando Tacchella. An abstraction-refinement approach to verification of artificial neural networks. In *Computer Aided Verification (CAV)*, pages 243–257, 2010.

[PT12] Luca Pulina and Armando Tacchella. Challenging SMT solvers to verify neural networks. *AI Commun.*, 25(2):117–135, 2012.

[Sch15] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[SNM+16] Karsten Scheibler, Felix Neubauer, Ahmed Mahdi, Martin Fränzle, Tino Teige, Tom Bienmüller, Detlef Fehrer, and Bernd Becker. Accurate ICP-based floating-point reasoning. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 177–184, 2016.

[SWWB15] Karsten Scheibler, Leonore Winterer, Ralf Wimmer, and Bernd Becker. Towards verification of artificial neural networks. In *MBMV Workshop 2015, Chemnitz, Germany*, pages 30–40, 2015.

[WK15] Michael Wagner and Philip Koopman. *A Philosophy for Developing Trust in Self-driving Cars*, chapter 5.1, pages 163–171. Springer International Publishing, 2015.

[YYS+15] Qian Yu, Yongxin Yang, Yi-Zhe Song, Tao Xiang, and Timothy M. Hospedales. Sketch-a-net that beats humans. In *British Machine Vision Conference (BMVC)*, pages 7.1–7.12, 2015.