Formal Methods in System Design
https://doi.org/10.1007/s10703-021-00363-7

Reluplex: a calculus for reasoning about deep neural networks

Guy Katz[1,2] · Clark Barrett[1] · David L. Dill[1] · Kyle Julian[3] · Mykel J. Kochenderfer[3]

Received: 10 September 2018 / Accepted: 10 February 2021

Abstract
Deep neural networks have emerged as a widely used and effective means for tackling complex, real-world problems. However, a major obstacle in applying them to safety-critical systems is the great difficulty in providing formal guarantees about their behavior. We present a novel, scalable, and efficient technique for verifying properties of deep neural networks (or providing counter-examples). The technique is based on the simplex method, extended to handle the non-convex *Rectified Linear Unit* (*ReLU*) activation function, which is a crucial ingredient in many modern neural networks. The verification procedure tackles neural networks as a whole, without making any simplifying assumptions. We evaluated our technique on a prototype deep neural network implementation of the next-generation airborne collision avoidance system for unmanned aircraft (ACAS Xu). Results show that our technique can successfully prove properties of networks that are an order of magnitude larger than the largest networks that could be verified previously.

Keywords Neural networks · Verification · Satisfiability modulo theories

This is an extended version of the paper *Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks* that appeared at CAV 2017.

✉ Guy Katz
guykatz@cs.huji.ac.il

✉ Clark Barrett
barrett@cs.stanford.edu

David L. Dill
dill@cs.stanford.edu

Kyle Julian
kjulian3@stanford.edu

Mykel J. Kochenderfer
mykel@stanford.edu

[1] Department of Computer Science, Stanford University, Stanford, USA

[2] Department of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel

[3] Department of Aeronautics and Astronautics, Stanford University, Stanford, USA

Published online: 01 July 2021

1 Introduction

Artificial neural networks [22,62] have emerged as a promising approach for creating scalable and robust systems. Applications include speech recognition [26], image classification [46], game playing [64], and many others. It is now clear that software that may be extremely difficult for humans to implement can instead be created by training *deep neural networks* (*DNN*s), and that the performance of these DNNs is often comparable to, or even surpasses, the performance of manually crafted software. DNNs are becoming widespread, and this trend is likely to continue and intensify.

Great effort is now being put into using DNNs as controllers for safety-critical systems such as autonomous vehicles [6] and airborne collision avoidance systems for unmanned aircraft (ACAS Xu) [32]. DNNs are trained over a finite set of inputs and outputs and are expected to *generalize*, i.e. to behave correctly for previously-unseen inputs. However, it has been observed that DNNs can react in unexpected and incorrect ways to even slight perturbations of their inputs [69]. This unexpected behavior of DNNs is likely to result in unsafe systems, or restrict the usage of DNNs in safety-critical applications. Hence, there is an urgent need for methods that can provide formal guarantees about DNN behavior. Unfortunately, manual reasoning about large DNNs is impossible, as their structure renders them incomprehensible to humans. Automatic verification techniques are thus sorely needed, but here, the state of the art is a severely limiting factor.

Verifying DNNs is a difficult problem. DNNs are large, non-linear, and non-convex, and verifying even simple properties about them is an NP-complete problem (see "Verifying properties in DNNs with ReLUs is NP-complete" section of the Appendix). DNN verification is experimentally beyond the reach of general-purpose tools such as *linear programming* (*LP*) solvers or existing *satisfiability modulo theories* (*SMT*) solvers [4,27,60], and, prior to this work, dedicated tools have only been able to handle very small networks (e.g. a single hidden layer with only 10 to 20 hidden nodes [59,60]).

The difficulty in proving properties about DNNs is caused by the presence of *activation functions*. A DNN is comprised of a set of layers of nodes, and the value of each node is determined by computing a linear combination of values from nodes in the preceding layer and then applying an activation function to the result. These activation functions are non-linear and render the problem non-convex. We focus here on DNNs with a specific kind of activation function, called a *Rectified Linear Unit* (*ReLU*) [56]. When the ReLU function is applied to a node with a positive value, it returns the value unchanged (the *active* case), but when the value is negative, the ReLU function returns 0 (the *inactive* case). ReLUs are very widely used [46,52], and it has been suggested that their piecewise linearity allows DNNs to generalize well to previously unseen inputs [19,22,30,56]. Past efforts at verifying properties of DNNs with ReLUs have had to make significant simplifying assumptions [4,27]—for instance, by considering only small input regions in which all ReLUs are fixed at either the active or inactive state [4], hence making the problem convex but at the cost of being able to verify only an approximation of the desired property.

We propose a novel, scalable, and efficient algorithm for verifying properties of DNNs with ReLUs. We address the issue of the activation functions head-on, by extending the simplex algorithm—a standard algorithm for solving LP instances—to support ReLU constraints. This is achieved by leveraging the piecewise linear nature of ReLUs and attempting to gradually satisfy the constraints that they impose as the algorithm searches for a feasible solution. We call the algorithm *Reluplex*, for "ReLU with Simplex".

The problem's NP-completeness means that we must expect the worst-case performance of the algorithm to be poor. However, as is often the case with SAT and SMT solvers, the performance in practice can be quite reasonable; in particular, our experiments show that during the search for a solution, many of the ReLUs can be ignored or even discarded altogether, reducing the search space by an order of magnitude or more. Occasionally, Reluplex will still need to *split* on a specific ReLU constraint—i.e., guess that it is either active or inactive, and possibly backtrack later if the choice leads to a contradiction.

We evaluated Reluplex on a family of 45 real-world DNNs, developed as an early prototype for the next-generation airborne collision avoidance system for unmanned aircraft ACAS Xu [32]. These fully connected DNNs have 8 layers and 300 ReLU nodes each, and are intended to be run onboard aircraft. They take in sensor data indicating the speed and present course of the aircraft (the *ownship*) and that of any nearby intruder aircraft, and issue appropriate navigation advisories. These advisories indicate whether the aircraft is clear-of-conflict, in which case the present course can be maintained, or whether it should turn to avoid collision. We successfully proved several properties of these networks, e.g. that a clear-of-conflict advisory will always be issued if the intruder is sufficiently far away or that it will never be issued if the intruder is sufficiently close and on a collision course with the ownship. Additionally, we were able to prove certain *robustness* properties [4] of the networks, meaning that small adversarial perturbations do not change the advisories produced for certain inputs.

Our contributions can be summarized as follows. We (1) present Reluplex, an SMT solver for a theory of linear real arithmetic with ReLU constraints; (2) show how DNNs and properties of interest can be encoded as inputs to Reluplex; (3) discuss several implementation details that are crucial to performance and scalability, such as the use of floating-point arithmetic, bound derivation for ReLU variables, conflict analysis, and under-approximation techniques; and (4) conduct a thorough evaluation on the DNN implementation of the prototype ACAS Xu system, demonstrating the ability of Reluplex to scale to DNNs that are an order of magnitude larger than those that can be analyzed using previously proposed techniques.

The rest of the paper is organized as follows. We begin with some background on DNNs, SMT, and simplex in Sect. 2. The abstract Reluplex algorithm is described in Sect. 3, with key implementation details highlighted in Sect. 4. We then describe the ACAS Xu system and its prototype DNN implementation that we used as a case-study in Sect. 5, followed by experimental results in Sect. 6. Related work is discussed in Sect. 7, and we conclude in Sect. 8.

2 Background

2.1 Neural networks

Deep neural networks (DNNs) are comprised of an input layer, an output layer, and multiple hidden layers in between. A layer is comprised of multiple nodes, each connected to nodes from the preceding layer using a predetermined set of weights (see Fig. 1). Weight selection is crucial, and is performed during a *training* phase (see, e.g., [22] for an overview). By assigning values to inputs and then feeding them forward through the network, values for each layer can be computed from the values of the previous layer, finally resulting in values for the outputs.

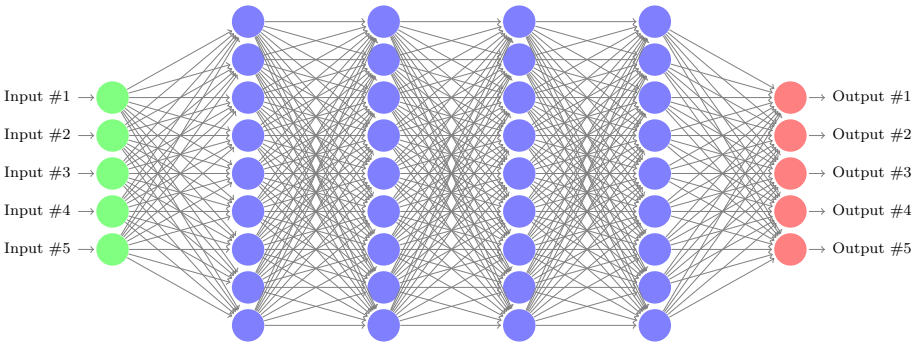
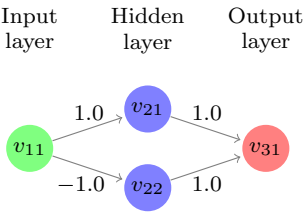


Fig. 1 A fully connected DNN with 5 input nodes (in green), 5 output nodes (in red), and 4 hidden layers containing a total of 36 hidden nodes (in blue)

Fig. 2 A small neural network



The value of each hidden node in the network is determined by calculating a linear combination of node values from the previous layer, and then applying a non-linear *activation function* [22]. Here, we focus on the Rectified Linear Unit (ReLU) activation function [56]. When a ReLU activation function is applied to a node, that node's value is calculated as the maximum of the linear combination of nodes from the previous layer and 0. We can thus regard ReLUs as the function $\text{ReLU}(x) = \max(0, x)$.

Formally, for a DNN N, we use n to denote the number of layers and s_i to denote the size of layer i (i.e., the number of its nodes). Layer 1 is the input layer, layer n is the output layer, and layers $2, \ldots, n-1$ are the hidden layers. The value of the j-th node of layer i is denoted $v_{i,j}$ and the column vector $[v_{i,1}, \ldots, v_{i,s_i}]^T$ is denoted V_i. Evaluating N entails calculating V_n for a given assignment V_1 of the input layer. This is performed by propagating the input values through the network using predefined weights and biases, and applying the activation functions—ReLUs, in our case. Each layer $2 \leq i \leq n$ has a weight matrix W_i of size $s_i \times s_{i-1}$ and a bias vector B_i of size s_i. For $2 \leq i < n$, V_i is given by $V_i = \text{ReLU}(W_i V_{i-1} + B_i)$, with the ReLU function being applied element-wise. The last layer's values are computed in a similar way, but without applying ReLUs: $V_n = W_n V_{n-1} + B_n$. These rules are applied repeatedly for each layer, until V_n is calculated. When the weight matrices $W_1, \ldots W_n$ do not have any zero entries, the network is said to be *fully connected* (see Fig. 1 for an illustration).

Figure 2 depicts a small network that we will use as a running example. The network has one input node, one output node and a single hidden layer with two nodes. The bias vectors are set to 0 and are ignored, and the weights are shown for each edge. The ReLU function is applied to each of the hidden nodes. It is possible to show that, due to the effect of the ReLUs, for non-negative input values the network's output is always identical to its input: $v_{31} \equiv v_{11}$.

2.2 Satisfiability modulo theories

We present our algorithm as a theory solver in the context of satisfiability modulo theories (SMT).[1] A *theory* is a pair $T = (\Sigma, \mathbf{I})$ where Σ is a signature and $\mathbf{I}$ is a class of Σ-interpretations, the *models* of T, that is closed under variable reassignment. A Σ-formula φ is T*-satisfiable* (resp., T*-unsatisfiable*) if it is satisfied by some (resp., no) interpretation in $\mathbf{I}$. In this paper, we consider only *quantifier-free* formulas. The SMT problem is the problem of determining the T-satisfiability of a formula for a given theory T.

Given a theory T with signature Σ, the DPLL(T) architecture [57] provides a generic approach for determining the T-satisfiability of Σ-formulas. In DPLL(T), a Boolean satisfiability (SAT) engine operates on a Boolean abstraction of the formula, performing Boolean propagation, case-splitting, and Boolean conflict resolution. The SAT engine is coupled with a dedicated *theory solver*, which checks the T-satisfiability of the decisions made by the SAT engine. *Splitting-on-demand* [2] extends DPLL(T) by allowing theory solvers to delegate case-splitting to the SAT engine in a generic and modular way. In Sect. 3, we present our algorithm as a deductive calculus (with splitting rules) operating on conjunctions of literals. The DPLL(T) and splitting-on-demand mechanisms can then be used to obtain a full decision procedure for arbitrary formulas.

2.3 Linear real arithmetic and simplex

In the context of DNNs, a particularly relevant theory is that of real arithmetic, which we denote as $\mathcal{T}_{\mathbb{R}}$. $\mathcal{T}_{\mathbb{R}}$ consists of the signature containing all rational number constants and the symbols $\{+, -, \cdot, \leq, \geq\}$, paired with the standard model of the real numbers. We focus on *linear* formulas: formulas over $\mathcal{T}_{\mathbb{R}}$ with the additional restriction that the multiplication symbol $\cdot$ can only appear if at least one of its operands is a rational constant. Linear atoms can always be rewritten into the form $\sum_{x_i \in \mathcal{X}} c_i x_i \bowtie d$, for $\bowtie \in \{=, \leq, \geq\}$, where $\mathcal{X}$ is a set of variables and c_i, d are rational constants.

The simplex method [10] is a standard and highly efficient decision procedure for determining the $\mathcal{T}_{\mathbb{R}}$-satisfiability of conjunctions of linear atoms.[2] Our algorithm extends simplex, and so we begin with an abstract calculus for the original algorithm (for a more thorough description see, e.g., [71]). The rules of the calculus operate over data structures we call *configurations*. For a given set of variables $\mathcal{X} = \{x_1, \ldots, x_n\}$, a simplex configuration is either one of the distinguished symbols $\{\mathtt{SAT}, \mathtt{UNSAT}\}$ or a tuple $\langle \mathcal{B}, T, l, u, \alpha \rangle$, where: $\mathcal{B} \subseteq \mathcal{X}$ is a set of basic variables; T, the *tableau*, contains for each $x_i \in \mathcal{B}$ an equation $x_i = \sum_{x_j \notin \mathcal{B}} c_j x_j$; l, u are mappings that assign each variable $x \in \mathcal{X}$ a lower and an upper bound, respectively; and α, the *assignment*, maps each variable $x \in \mathcal{X}$ to a real value. The initial configuration (and in particular the initial tableau T_0) is derived from a conjunction of input atoms as follows: for each atom $\sum_{x_i \in \mathcal{X}} c_i x_i \bowtie d$, a new basic variable b is introduced, the equation $b = \sum_{x_i \in \mathcal{X}} c_i x_i$ is added to the tableau, and d is added as a bound for b (either upper, lower, or both, depending on $\bowtie$). The initial assignment is set to 0 for all variables, ensuring that all tableau equations hold (though variable bounds may be violated).

[1] Consistent with most treatments of SMT, we assume many-sorted first-order logic with equality as our underlying formalism (see, e.g., [3] for details).

[2] There exist SMT-friendly extensions of simplex (see e.g. [39]) which can handle $\mathcal{T}_{\mathbb{R}}$-satisfiability of arbitrary literals, including strict inequalities and disequalities, but we omit these extensions here for simplicity (and without loss of generality).

$$\textsf{Pivot}_1 \quad \frac{x_i \in \mathcal{B}, \quad \alpha(x_i) < l(x_i), \quad x_j \in \text{slack}^+(x_i)}{T := pivot(T, i, j), \quad \mathcal{B} := \mathcal{B} \cup \{x_j\} \setminus \{x_i\}}$$

$$\textsf{Pivot}_2 \quad \frac{x_i \in \mathcal{B}, \quad \alpha(x_i) > u(x_i), \quad x_j \in \text{slack}^-(x_i)}{T := pivot(T, i, j), \quad \mathcal{B} := \mathcal{B} \cup \{x_j\} \setminus \{x_i\}}$$

$$\textsf{Update} \quad \frac{x_j \notin \mathcal{B}, \quad \alpha(x_j) < l(x_j) \vee \alpha(x_j) > u(x_j), \quad l(x_j) \leq \alpha(x_j) + \delta \leq u(x_j)}{\alpha := update(\alpha, x_j, \delta)}$$

$$\textsf{Failure}_1 \quad \frac{x_i \in \mathcal{B}, \quad (\alpha(x_i) < l(x_i) \ \wedge \ \text{slack}^+(x_i) = \emptyset) \vee (\alpha(x_i) > u(x_i) \ \wedge \ \text{slack}^-(x_i) = \emptyset)}{\texttt{UNSAT}}$$

$$\textsf{Failure}_2 \quad \frac{l(x_i) > u(x_i)}{\texttt{UNSAT}}$$

$$\textsf{Success} \quad \frac{\forall x_i \in \mathcal{X}.\ l(x_i) \leq \alpha(x_i) \leq u(x_i)}{\texttt{SAT}}$$

Fig. 3 Derivation rules for the abstract simplex algorithm

The tableau T can be regarded as a matrix expressing each of the basic variables (variables in $\mathcal{B}$) as a linear combination of non-basic variables (variables in $\mathcal{X} \setminus \mathcal{B}$). The rows of T correspond to the variables in $\mathcal{B}$ and its columns to those of $\mathcal{X} \setminus \mathcal{B}$. For $x_i \in \mathcal{B}$ and $x_j \notin \mathcal{B}$ we denote by $T_{i,j}$ the coefficient c_j of x_j in the equation $x_i = \sum_{x_j \notin \mathcal{B}} c_j x_j$. The tableau is changed via pivoting: the switching of a basic variable x_i (the *leaving* variable) with a non-basic variable x_j (the *entering* variable) for which $T_{i,j} \neq 0$. A $pivot(T, i, j)$ operation returns a new tableau in which the equation $x_i = \sum_{x_k \notin \mathcal{B}} c_k x_k$ has been replaced by the equation $x_j = \frac{x_i}{c_j} - \sum_{x_k \notin \mathcal{B}, k \neq j} \frac{c_k}{c_j} x_k$, and in which every occurrence of x_j in each of the other equations has been replaced by the right-hand side of the new equation (the resulting expressions are also normalized to retain the tableau form). The variable assignment α is changed via *update* operations that are applied to non-basic variables: for $x_j \notin \mathcal{B}$, an $update(\alpha, x_j, \delta)$ operation returns an updated assignment α' identical to α, except that $\alpha'(x_j) = \alpha(x_j) + \delta$ and for every $x_i \in \mathcal{B}$, we have $\alpha'(x_i) = \alpha(x_i) + \delta \cdot T_{i,j}$. To simplify later presentation we also denote:

$$\text{slack}^+(x_i) = \{x_j \notin \mathcal{B} \mid (T_{i,j} > 0 \wedge \alpha(x_j) < u(x_j)) \vee (T_{i,j} < 0 \wedge \alpha(x_j) > l(x_j))\}$$
$$\text{slack}^-(x_i) = \{x_j \notin \mathcal{B} \mid (T_{i,j} < 0 \wedge \alpha(x_j) < u(x_j)) \vee (T_{i,j} > 0 \wedge \alpha(x_j) > l(x_j))\}$$

The rules of the simplex calculus are provided in Fig. 3 in *guarded assignment form*. A rule applies to a configuration S if all of the rule's premises hold for S. A rule's conclusion describes how each component of S is changed, if at all. When S' is the result of applying a rule to S, we say that S derives S'. A sequence of configurations S_i where each S_i derives S_{i+1} is called a *derivation*.

The Update rule (with appropriate values of δ) is used to enforce that non-basic variables satisfy their bounds. Basic variables cannot be directly updated. Instead, if a basic variable x_i is too small or too great, either the $\textsf{Pivot}_1$ or the $\textsf{Pivot}_2$ rule is applied, respectively, to pivot it with a non-basic variable x_j. This makes x_i non-basic so that its assignment can be adjusted using the Update rule. Pivoting is only allowed when x_j affords *slack*, that is, the assignment for x_j can be adjusted to bring x_i closer to its bound without violating its own bound. Of course, once pivoting occurs and the Update rule is used to bring x_i within its bounds, other variables (such as the now basic x_j) may be sent outside their bounds, in

which case they must be corrected in a later iteration. If a basic variable is out of bounds, but none of the non-basic variables affords it any slack, then the $\mathsf{Failure}_1$ rule applies and the problem is unsatisfiable. The problem can also be determined to be unsatisfiable (using the $\mathsf{Failure}_2$ rule) if there exists a variable whose lower bound is strictly greater than its upper bound. Because the tableau is only changed by scaling and adding rows, the set of variable assignments that satisfy its equations is always kept identical to that of T_0. Also, the *update* operation guarantees that α continues to satisfy the equations of T. Thus, if all variables are within bounds then the Success rule can be applied, indicating that α constitutes a satisfying assignment for the original problem.

It is well-known that the simplex calculus is *sound* [71] (i.e. if a derivation ends in SAT or UNSAT, then the original problem is satisfiable or unsatisfiable, respectively) and *complete* (there always exists a derivation ending in either SAT or UNSAT from any starting configuration). Termination can be guaranteed if certain strategies are used in applying the transition rules—in particular in picking the leaving and entering variables when multiple options exist [71]. Variable selection strategies are also known to have a dramatic effect on performance [71]. We note that the version of simplex described above is usually referred to as *phase one* simplex, and is usually followed by a *phase two* in which the solution is optimized according to a cost function. However, as we are only considering satisfiability, phase two is not required.

3 From simplex to Reluplex

The simplex algorithm described in Sect. 2 is an efficient means for solving problems that can be encoded as a conjunction of atoms. Unfortunately, while the weights, biases, and certain properties of DNNs can be encoded this way, the non-linear ReLU functions cannot.

When a theory solver operates within an SMT solver, input atoms can be embedded in arbitrary Boolean structure. A naïve approach is then to encode ReLUs using disjunctions, which is possible because ReLUs are piecewise linear. However, this encoding requires the SAT engine within the SMT solver to enumerate the different cases. In the worst case, for a DNN with n ReLU nodes, the solver ends up splitting the problem into 2^n sub-problems, each of which is a conjunction of atoms. As observed by us and others [4,27], this theoretical worst-case behavior is also seen in practice, and hence this approach is practical only for very small networks. A similar phenomenon occurs when encoding DNNs as mixed integer problems (see Sect. 6).

We take a different route and extend the theory $\mathcal{T}_{\mathbb{R}}$ to a theory $\mathcal{T}_{\mathbb{R}R}$ of reals and ReLUs. $\mathcal{T}_{\mathbb{R}R}$ is almost identical to $\mathcal{T}_{\mathbb{R}}$, except that its signature additionally includes the binary predicate ReLU with the interpretation: $\mathrm{ReLU}(x, y)$ iff $y = \max(0, x)$. Formulas are then assumed to contain atoms that are either linear inequalities or applications of the ReLU predicate to linear terms.

DNNs and their (linear) properties can be directly encoded as conjunctions of $\mathcal{T}_{\mathbb{R}R}$-atoms. The main idea is to encode a single ReLU node v as a *pair* of variables, v^b and v^f, and then assert $\mathrm{ReLU}(v^b, v^f)$. v^b, the *backward-facing* variable, is used to express the connection of v to nodes from the preceding layer; whereas v^f, the *forward-facing* variable, is used for the connections of v to the following layer (see Fig. 4). The rest of this section is devoted to presenting an efficient algorithm, Reluplex, for deciding the satisfiability of a conjunction of such atoms.

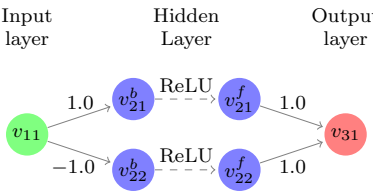


Fig. 4 The network from Fig. 2, with ReLU nodes split into backward- and forward-facing variables

3.1 The Reluplex procedure

As with simplex, Reluplex allows variables to temporarily violate their bounds as it iteratively looks for a feasible variable assignment. However, Reluplex also allows variables that are members of ReLU pairs to temporarily violate the ReLU semantics. Then, as it iterates, Reluplex repeatedly picks variables that are either out of bounds or that violate a ReLU, and corrects them using Pivot and Update operations.

For a given set of variables $\mathcal{X} = \{x_1, \ldots, x_n\}$, a Reluplex configuration is either one of the distinguished symbols {SAT, UNSAT} or a tuple $\langle \mathcal{B}, T, l, u, \alpha, R \rangle$, where $\mathcal{B}, T, l, u$ and α are as before, and $R \subset \mathcal{X} \times \mathcal{X}$ is the set of ReLU connections. The initial configuration for a conjunction of atoms is also obtained as before except that $\langle x, y \rangle \in R$ iff $\text{ReLU}(x, y)$ is an atom. The simplex transition rules Pivot_1, Pivot_2, Update, Failure_1, and Failure_2 are included also in Reluplex. We replace the Success rule with the ReluSuccess rule and add rules for handling ReLU violations, as depicted in Fig. 5. The Update_b and Update_f rules allow a broken ReLU connection to be corrected by updating the backward- or forward-facing variables, respectively, provided that these variables are non-basic. The PivotForRelu rule allows a basic variable appearing in a ReLU to be pivoted so that either Update_b or Update_f can be applied (this is needed to make progress when both variables in a ReLU are basic and their assignments do not satisfy the ReLU semantics). The ReluSplit rule is used for splitting on ReLU connections, guessing whether they are inactive (by enforcing that $u(x_i) \leq 0$, $l(x_j) \geq 0$ and $u(x_j) \leq 0$), or active (by enforcing that $l(x_i) \geq 0$). When we guess that a ReLU is active, we also apply the *addEq* operation, which adds to the tableau a new equation, $x_j = x_i$, in order to enforce that the ReLU is satisfied in the active phase. This operation uses a fresh variable, b, as the basic variable for the equation, i.e. $b = x_j - x_i$, and b is then added to $\mathcal{B}$. We assume, without loss of generality, that these fresh variables, one for each ReLU, exist in $\mathcal{X}$, have 0 for lower and upper bounds, and are not used in any other equation. Further, the assignment of b is set to $\alpha(x_j) - \alpha(x_i)$, to make sure that the new equation is satisfied by the current assignment. (In case x_j is basic, i.e. $x_j = \sum_{x_k \notin \mathcal{B}} c_k x_k$, we substitute x_j with $x_j = \sum_{x_k \notin \mathcal{B}} c_k x_k$ in the new equation; and likewise for x_i).

Introducing splitting means that derivations are no longer linear. Using the notion of derivation trees, we can show that Reluplex is sound and complete (see "The Reluplex calculus is sound and complete" section of the Appendix). In practice, splitting can be managed by a SAT engine with splitting-on-demand [2]. The naïve approach mentioned at the beginning of this section can be simulated by applying the ReluSplit rule eagerly, once for each ReLU pair, and then solving each derived sub-problem separately (this reduction trivially guarantees termination just as do branch-and-cut techniques in mixed integer solvers [58]). However, a more scalable strategy is to try to fix broken ReLU pairs using the Update_b and Update_f rules first, and split only when the number of updates to a specific ReLU pair exceeds some threshold. Intuitively, this is likely to limit splits to "problematic" ReLU pairs, while still guaranteeing termination. Specifically, there exist well-known strategies for applying the simplex rules in a way that guarantees that within a finite number of steps, either all variables

$$\textsf{Update}_b \quad \frac{x_i \notin \mathcal{B}, \quad \langle x_i, x_j \rangle \in R, \quad \alpha(x_j) \neq \max(0, \alpha(x_i)), \quad \alpha(x_j) \geq 0}{\alpha := update(\alpha, x_i, \alpha(x_j) - \alpha(x_i))}$$

$$\textsf{Update}_f \quad \frac{x_j \notin \mathcal{B}, \quad \langle x_i, x_j \rangle \in R, \quad \alpha(x_j) \neq \max(0, \alpha(x_i))}{\alpha := update(\alpha, x_j, \max(0, \alpha(x_i)) - \alpha(x_j))}$$

$$\textsf{PivotForRelu} \quad \frac{x_i \in \mathcal{B}, \quad \exists x_l.\ \langle x_i, x_l \rangle \in R \vee \langle x_l, x_i \rangle \in R, \quad x_j \notin \mathcal{B}, \quad T_{i,j} \neq 0}{T := pivot(T, i, j), \quad \mathcal{B} := \mathcal{B} \cup \{x_j\} \setminus \{x_i\}}$$

$$\textsf{ReluSplit} \quad \frac{\langle x_i, x_j \rangle \in R}{\begin{array}{ll} u(x_i) := \min(u(x_i), 0), & l(x_i) := \max(l(x_i), 0), \\ l(x_j) := \max(l(x_j), 0), & \langle \mathcal{B}, T, \alpha \rangle := addEq(x_j = x_i) \\ u(x_j) := \min(u(x_j), 0) & \end{array}}$$

$$\textsf{ReluSuccess} \quad \frac{\forall x \in \mathcal{X}.\ l(x) \leq \alpha(x) \leq u(x), \quad \forall \langle x^b, x^f \rangle \in R.\ \alpha(x^f) = \max(0, \alpha(x^b))}{\texttt{SAT}}$$

Fig. 5 Additional derivation rules for the abstract Reluplex algorithm

become assigned to values within their bounds, or the $\textsf{Failure}_1$ or $\textsf{Failure}_2$ rules are applicable (and are applied) [71]. By repeatedly using such a strategy for fixing out-of-bounds violations, and by splitting on a ReLU pair whenever the $\textsf{Update}_b$, $\textsf{Update}_f$ or PivotForRelu rules are applied to it more than some fixed number of times, termination is guaranteed.

3.2 Example

To illustrate the use of the derivation rules, we use Reluplex to solve a simple example. Consider the network in Fig. 4, and suppose we wish to check whether it is possible to satisfy $v_{11} \in [0, 1]$ and $v_{31} \in [0.5, 1]$. As we know that for non-negative inputs the network outputs its input unchanged ($v_{31} \equiv v_{11}$), we expect Reluplex to be able to derive SAT. The initial Reluplex configuration is obtained by introducing new basic variables a_1, a_2, a_3, and encoding the network with the equations:

$$a_1 = -v_{11} + v_{21}^b \quad a_2 = v_{11} + v_{22}^b \quad a_3 = -v_{21}^f - v_{22}^f + v_{31}$$

The equations above form the initial tableau T_0, and the initial set of basic variables is $\mathcal{B} = \{a_1, a_2, a_3\}$. The set of ReLU connections is $R = \{\langle v_{21}^b, v_{21}^f \rangle, \langle v_{22}^b, v_{22}^f \rangle\}$. The initial assignment of all variables is set to 0. The lower and upper bounds of the basic variables are set to 0, in order to enforce the equalities that they represent. The bounds for the input and output variables are set according to the problem at hand; and the hidden variables are unbounded, except that forward-facing variables are, by definition, non-negative:

	v_{11}	v_{21}^b	v_{21}^f	v_{22}^b	v_{22}^f	v_{31}	a_1	a_2	a_3
Lower bound	0	$-\infty$	0	$-\infty$	0	0.5	0	0	0
Assignment	0	0	0	0	0	0	0	0	0
Upper bound	1	∞	∞	∞	∞	1	0	0	0

Starting from this initial configuration, our search strategy is to first fix any out-of-bounds variables. Variable v_{31} is non-basic and is out of bounds, so we perform an Update step and set it to 0.5. As a result, a_3, which depends on v_{31}, is also set to 0.5. a_3 is now basic and out of bounds, so we pivot it with v_{21}^f, and then update a_3 back to 0. The tableau now consists of the equations:

$$a_1 = -v_{11} + v_{21}^b \qquad a_2 = v_{11} + v_{22}^b \qquad v_{21}^f = -v_{22}^f + v_{31} - a_3$$

And the assignment is $\alpha(v_{21}^f) = 0.5$, $\alpha(v_{31}) = 0.5$, and $\alpha(v) = 0$ for all other variables v. At this point, all variables are within their bounds, but the ReluSuccess rule does not apply because $\alpha(v_{21}^f) = 0.5 \neq 0 = \max(0, \alpha(v_{21}^b))$.

The next step is to fix the broken ReLU pair $\langle v_{21}^b, v_{21}^f \rangle$. Since v_{21}^b is non-basic, we use Update_b to increase its value by 0.5. The assignment becomes $\alpha(v_{21}^b) = 0.5$, $\alpha(v_{21}^f) = 0.5$, $\alpha(v_{31}) = 0.5$, $\alpha(a_1) = 0.5$, and $\alpha(v) = 0$ for all other variables v. All ReLU constraints hold, but a_1 is now out of bounds. This is fixed by pivoting a_1 with v_{11} and then updating it. The resulting tableau is:

$$v_{11} = v_{21}^b - a_1 \qquad a_2 = v_{21}^b + v_{22}^b - a_1 \qquad v_{21}^f = -v_{22}^f + v_{31} - a_3$$

Observe that because v_{11} is now basic, it was eliminated from the equation for a_2 and replaced with $v_{21}^b - a_1$. The non-zero assignments are now $\alpha(v_{11}) = 0.5, \alpha(v_{21}^b) = 0.5, \alpha(v_{21}^f) = 0.5$, $\alpha(v_{31}) = 0.5$, $\alpha(a_2) = 0.5$. Variable a_2 is now too large, and so we have a final round of pivot-and-update: a_2 is pivoted with v_{22}^b and then updated back to 0. The final tableau and assignments are:

$$v_{11} = v_{21}^b - a_1 \qquad v_{22}^b = -v_{21}^b + a_1 + a_2 \qquad v_{21}^f = -v_{22}^f + v_{31} - a_3$$

	v_{11}	v_{21}^b	v_{21}^f	v_{22}^b	v_{22}^f	v_{31}	a_1	a_2	a_3
Lower bound	0	$-\infty$	0	$-\infty$	0	0.5	0	0	0
Assignment	0.5	0.5	0.5	−0.5	0	0.5	0	0	0
Upper bound	1	∞	∞	∞	∞	1	0	0	0

and the algorithm halts with the feasible solution it has found. A key observation is that we did not ever split on any of the ReLU connections. Instead, it was sufficient to simply use updates to adjust the ReLU variables as needed.

4 Efficiently implementing Reluplex

We next discuss four techniques that significantly boost the performance of Reluplex: use of tighter bound derivation, conflict analysis, floating point arithmetic, and under-approximations.

4.1 Tighter bound derivation

The simplex and Reluplex procedures naturally lend themselves to deriving tighter variable bounds as the search progresses [39]. Consider a basic variable $x_i \in \mathcal{B}$ and let $\text{pos}(x_i) = \{x_j \notin$

$\mathcal{B} \mid T_{i,j} > 0\}$ and $\text{neg}(x_i) = \{x_j \notin \mathcal{B} \mid T_{i,j} < 0\}$. Throughout the execution, the following rules can be used to derive tighter bounds for x_i, regardless of the current assignment:

$$\textsf{deriveLowerBound} \quad \frac{x_i \in \mathcal{B},\ \ l(x_i) < \sum_{x_j \in \text{pos}(x_i)} T_{i,j} \cdot l(x_j) + \sum_{x_j \in \text{neg}(x_i)} T_{i,j} \cdot u(x_j)}{l(x_i) := \sum_{x_j \in \text{pos}(x_i)} T_{i,j} \cdot l(x_j) + \sum_{x_j \in \text{neg}(x_i)} T_{i,j} \cdot u(x_j)}$$

$$\textsf{deriveUpperBound} \quad \frac{x_i \in \mathcal{B},\ \ u(x_i) > \sum_{x_j \in \text{pos}(x_i)} T_{i,j} \cdot u(x_j) + \sum_{x_j \in \text{neg}(x_i)} T_{i,j} \cdot l(x_j)}{u(x_i) := \sum_{x_j \in \text{pos}(x_i)} T_{i,j} \cdot u(x_j) + \sum_{x_j \in \text{neg}(x_i)} T_{i,j} \cdot l(x_j)}$$

The derived bounds can later be used to derive additional, tighter bounds. Similar approaches have proven useful for linear arithmetic theory solvers [11].

When tighter bounds are derived for ReLU variables, these variables can sometimes be eliminated, i.e., fixed to the active or inactive state, without splitting. For a ReLU pair $x^f = \text{ReLU}(x^b)$, discovering that either $l(x^b)$ or $l(x^f)$ is strictly positive means that in any feasible solution this ReLU connection will be active. Similarly, discovering that $u(x^b) < 0$ implies inactivity. In these cases, the ReLU constraint should be replaced by linear constraints similar to those introduced by the ReluSplit rule.

Bound tightening operations incur overhead, and simplex implementations often use them sparsely [39]. In Reluplex, however, the benefits of eliminating ReLUs justify the cost. The actual amount of bound tightening to perform can be determined heuristically; we describe the heuristic that we used in Sect. 6.

4.2 Derived bounds and conflict analysis

Bound derivation can lead to situations where we learn that $l(x) > u(x)$ for some variable x. Such contradictions allow Reluplex to immediately undo a previous split (or answer UNSAT if no previous splits exist). However, in many cases more than just the previous split can be undone. For example, if we have performed 8 nested splits so far, it may be that the conflicting bounds for x are the direct result of split number 5 but have only just been discovered. In this case we can immediately undo splits number 8, 7, and 6. This is a particular case of *conflict analysis*, which is a standard technique in SAT and SMT solvers [53].

4.3 Floating point arithmetic

SMT solvers typically use precise (as opposed to floating point) arithmetic to avoid roundoff errors and guarantee soundness. Unfortunately, precise computation is usually at least an order of magnitude slower than its floating point equivalent, and so efforts have been made to leverage floating point arithmetic in solvers (e.g., [17,55]). Invoking Reluplex on a large DNN can require millions of pivot operations, each of which involves the multiplication and division of rational numbers, potentially with large numerators or denominators—making the use of floating point arithmetic important for scalability.

There are standard techniques for keeping the roundoff error small when implementing simplex using floating point, which we incorporated into our implementation. For example, one important practice is trying to avoid Pivot operations involving the inversion of extremely small numbers [71].

To provide increased confidence that any roundoff error remained within an acceptable range, we also added the following safeguards: (1) After a certain number of Pivot steps we

would measure the accumulated roundoff error; and (2) If the error exceeded a threshold M, we would *restore* the coefficients of the current tableau T using the initial tableau T_0.

Cumulative roundoff error can be measured by plugging the current assignment values for the non-basic variables into the equations of the initial tableau T_0, using them to calculate the values for every basic variable x_i, and then measuring by how much these values differ from the current assignment $\alpha(x_i)$. We define the cumulative roundoff error as:

$$\sum_{x_i \in \mathcal{B}_0} \left| \alpha(x_i) - \sum_{x_j \notin \mathcal{B}_0} T_{0_{i,j}} \cdot \alpha(x_j) \right|$$

T is restored by starting from T_0 and performing a short series of Pivot steps that result in the same set of basic variables as in T. In general, the shortest sequence of pivot steps to transform T_0 to T is much shorter than the series of steps that was followed by Reluplex—and hence, although it is also performed using floating point arithmetic, it incurs a smaller roundoff error.

The tableau restoration technique serves to increase our confidence in the algorithm's results when using floating point arithmetic, but it does not guarantee soundness. Providing true soundness when using floating point arithmetic remains a future goal (see Sect. 8).

4.4 Under-approximations

Under-approximation can be integrated into the Reluplex algorithm in a straightforward manner. Consider a variable x with lower and upper bounds $l(x)$ and $u(x)$, respectively. Since we are searching for feasible solutions for which $x \in [l(x), u(x)]$, an under-approximation can be obtained by restricting this range, and only considering feasible solutions for which $x \in [l(x) + \epsilon, u(x) - \epsilon]$ for some small $\epsilon > 0$.

Applying under-approximations can be particularly useful when it effectively eliminates a ReLU constraint (consequently reducing the potential number of case splits needed). Specifically, observe a ReLU pair $x^f = \text{ReLU}(x^b)$ for which we have $l(x^b) \geq -\epsilon$ for a very small positive ϵ. We can under-approximate this range and instead set $l(x^b) = 0$; and, as previously discussed, we can then fix the ReLU pair to the active state. Symmetrical measures can be employed when learning a very small upper bound for x^f, in this case leading to the ReLU pair being fixed in the inactive state.

Any feasible solution that is found using this kind of under-approximation will be a feasible solution for the original problem. However, if we determine that the under-approximated problem is infeasible, the original may yet be feasible.

5 Case study: the ACAS Xu system

Airborne collision avoidance systems are critical for ensuring the safe operation of aircraft. The *Traffic Alert and Collision Avoidance System* (*TCAS*) was developed in response to midair collisions between commercial aircraft, and is currently mandated on all large commercial aircraft worldwide [47]. Recent work has focused on creating a new system, known as *Airborne Collision Avoidance System X* (*ACAS X*) [41,42]. This system adopts an approach that involves solving a partially observable Markov decision process to optimize the alerting logic and further reduce the probability of midair collisions, while minimizing unnecessary alerts [41,42,44].

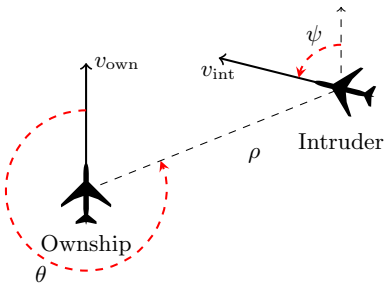


Fig. 6 Geometry for ACAS Xu horizontal logic table

The unmanned variant of ACAS X, known as ACAS Xu, produces horizontal maneuver advisories. So far, development of ACAS Xu has focused on using a large lookup table that maps sensor measurements to advisories [32]. However, this table requires over 2GB of memory. There is concern about the memory requirements for certified avionics hardware. To overcome this challenge, a DNN representation was explored as a potential replacement for the table [32]. Initial results show a dramatic reduction in memory requirements without compromising safety. In fact, due to its continuous nature, the DNN approach can sometimes outperform the discrete lookup table [32]. Recently, in order to reduce lookup time, the DNN approach was improved further, and the single DNN was replaced by an array of 45 DNNs. As a result, the original 2GB table can now be substituted with efficient DNNs that require less than 3MB of memory.

A DNN implementation of ACAS Xu presents new certification challenges. Proving that a set of inputs cannot produce an erroneous alert is paramount for certifying the system for use in safety-critical settings. Previous certification methodologies included exhaustively testing the system in 1.5 million simulated encounters [43], but this is insufficient for proving that faulty behaviors do not exist within the continuous DNNs. This highlights the need for verifying DNNs and makes the ACAS Xu DNNs prime candidates on which to apply Reluplex.

5.1 Network functionality

The ACAS Xu system maps input variables to action advisories. Each advisory is assigned a score, with the lowest score corresponding to the best action. The input state is composed of seven dimensions (shown in Fig. 6) which represent information determined from sensor measurements [42]: (1) ρ: Distance from ownship to intruder; (2) θ: Angle to intruder relative to ownship heading direction; (3) ψ: Heading angle of intruder relative to ownship heading direction; (4) v_{own}: Speed of ownship; (5) v_{int}: Speed of intruder; (6) τ: Time until loss of vertical separation; and (7) a_{prev}: Previous advisory. There are five outputs which represent the different horizontal advisories that can be given to the ownship: Clear-of-Conflict (COC), weak right, strong right, weak left, or strong left. Weak and strong mean heading rates of 1.5 $\circ$ /s and 3.0 $\circ$ /s, respectively.

The array of 45 DNNs was produced by discretizing τ and a_{prev}, and producing a network for each discretized combination. Each of these networks thus has five inputs (one for each of the other dimensions) and five outputs. The DNNs are fully connected, use ReLU activation functions, and have 6 hidden layers with a total of 300 ReLU nodes each.

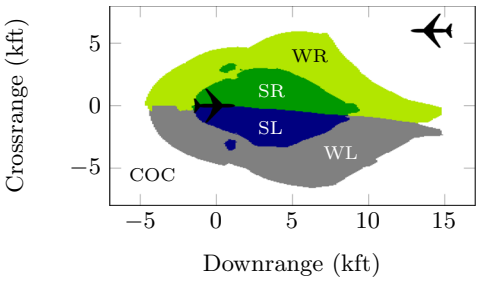


Fig. 7 Advisories for a head-on encounter with $a_{\text{prev}} = \text{COC}$, $\tau = 0$ s. Ranges are measured in kilofeet (kft)

5.2 Network properties

It is desirable to verify that the ACAS Xu networks assign correct scores to the output advisories in various input domains. Fig. 7 illustrates this kind of property by showing a top-down view of a head-on encounter scenario, in which each pixel is colored to represent the best action if the intruder were at that location. We expect the DNN's advisories to be consistent in each of these regions; however, Fig. 7 was generated from a finite set of input samples, and there may exist other inputs for which a wrong advisory is produced, possibly leading to collision. Therefore, we used Reluplex to prove properties from the following categories on the DNNs: (1) The system does not give unnecessary turning advisories; (2) Alerting regions are uniform and do not contain inconsistent alerts; and (3) Strong alerts do not appear for high τ values.

6 Evaluation

We used a proof-of-concept implementation of Reluplex to check realistic properties on the 45 ACAS Xu DNNs. Our implementation consists of three main logical components: (1) a simplex engine for providing core functionality such as tableau representation and pivot and update operations; (2) a Reluplex engine for driving the search and performing bound derivation, ReLU pivots and ReLU updates; and (3) a simple SMT core for providing splitting-on-demand services. For the simplex engine we used the GLPK open-source LP solver[3] with some modifications, for instance in order to allow the Reluplex core to perform bound tightening on tableau equations calculated by GLPK. Our implementation, together with the experiments described in this section, is available online [33].

Our search strategy was to repeatedly fix any out-of-bounds violations first, and only then correct any violated ReLU constraints (possibly introducing new out-of-bounds violations). We performed bound tightening on the entering variable after every pivot operation, and performed a more thorough bound tightening on all the equations in the tableau once every few thousand pivot steps. Tighter bound derivation proved extremely useful, and we often observed that after splitting on about 10% of the ReLU variables it led to the elimination of all remaining ReLUs. We counted the number of times a ReLU pair was fixed via Update_b or Update_f or pivoted via PivotForRelu, and split only when this number reached 5 (a number empirically determined to work well). We also implemented conflict analysis and back-jumping. Finally, we checked the accumulated roundoff error (due to the use of double-precision floating point arithmetic) after every 5000 Pivot steps, and restored the tableau if

[3] www.gnu.org/software/glpk/.

Table 1 Comparison to SMT and LP solvers. Entries indicate solution time (in seconds)

	φ_1	φ_2	φ_3	φ_4	φ_5	φ_6	φ_7	φ_8
CVC4	–	–	–	–	–	–	–	–
Z3	–	–	–	–	–	–	–	–
Yices	1	37	–	–	–	–	–	–
MathSat	2040	9780	–	–	–	–	–	–
Gurobi	1	1	1	–	–	–	–	–
Reluplex	8	2	7	7	93	4	7	9

the error exceeded 10^{-6}. Most experiments described below required two tableau restorations or fewer.

We began by comparing our implementation of Reluplex to state-of-the-art solvers: the CVC4, Z3, Yices and MathSat SMT solvers and the Gurobi LP solver (see Table 1). We ran all solvers with a 4 hour timeout on 2 of the ACAS Xu networks (selected arbitrarily), trying to solve for 8 simple satisfiable properties $\varphi_1, \ldots, \varphi_8$, each of the form $x \geq c$ for a fixed output variable x and a constant c. The SMT solvers generally performed poorly, with only Yices and MathSat successfully solving two instances each. We attribute the results to these solvers' lack of direct support for encoding ReLUs, and to their use of precise arithmetic. Gurobi solved 3 instances quickly, but timed out on all the rest. Its logs indicated that whenever Gurobi could solve the problem without case-splitting, it did so quickly; but whenever the problem required case-splitting, Gurobi would time out. Reluplex was able to solve all 8 instances. See "Encoding ReLUs for SMT and LP solvers" section of the Appendix for the SMT and LP encodings that we used.

Next, we used Reluplex to test a set of 10 quantitative properties $\phi_1, \ldots, \phi_{10}$. The properties, described below, are formally defined in Formal definitions for properties $\Phi_1, \ldots, \Phi_{10}$ section of the Appendix. Table 2 depicts for each property the number of tested networks (specified as part of the property), the test results and the total duration (in seconds). The *Stack* and *Splits* columns list the maximal depth of nested case-splits reached (out of a maximal depth of 300; averaged over the tested networks) and the total number of case-splits performed, respectively. For each property, we looked for an input that would violate it; thus, an `UNSAT` result indicates that a property holds, and a `SAT` result indicates that it does not hold. In the `SAT` case, the satisfying assignment is an example of an input that violates the property.

Property ϕ_1 states that if the intruder is distant and is significantly slower than the ownship, the score of a COC advisory will always be below a certain fixed threshold (recall that the best action has the lowest score). Property ϕ_2 states that under similar conditions, the score for COC can never be maximal, meaning that it can never be the worst action to take. This property was discovered not to hold for 35 networks, but this was later determined to be acceptable behavior: the DNNs have a strong bias for producing the same advisory they had previously produced, and this can result in advisories other than COC even for far-away intruders if the previous advisory was also something other than COC. Properties ϕ_3 and ϕ_4 deal with situations where the intruder is directly ahead of the ownship, and state that the DNNs will never issue a COC advisory.

Properties ϕ_5 through ϕ_{10} each involve a single network, and check for consistent behavior in a specific input region. For example, ϕ_5 states that if the intruder is near and approaching from the left, the network advises "strong right". Property ϕ_7, on which we timed out, states that when the vertical separation is large the network will never advise a strong turn. The

Table 2 Verifying properties of the ACAS Xu networks

	Networks	Result	Time	Stack	Splits
ϕ_1	41	UNSAT	394517	47	1522384
	4	TIMEOUT			
ϕ_2	1	UNSAT	463	55	88388
	35	SAT	82419	44	284515
ϕ_3	42	UNSAT	28156	22	52080
ϕ_4	42	UNSAT	12475	21	23940
ϕ_5	1	UNSAT	19355	46	58914
ϕ_6	1	UNSAT	180288	50	548496
ϕ_7	1	TIMEOUT			
ϕ_8	1	SAT	40102	69	116697
ϕ_9	1	UNSAT	99634	48	227002
ϕ_{10}	1	UNSAT	19944	49	88520

large input domain and the particular network proved difficult to verify. Property ϕ_8 states that for a large vertical separation and a previous "weak left" advisory, the network will either output COC or continue advising "weak left". Here, we were able to find a counter-example, exposing an input on which the DNN was inconsistent with the lookup table. This confirmed the existence of a discrepancy that had also been seen in simulations, and which will be addressed by retraining the DNN. We observe that for all properties, the maximal depth of nested splits was always well below the total number of ReLU nodes, 300, illustrating the fact that Reluplex did not split on many of them. Also, the total number of case-splits indicates that large portions of the search space were pruned.

Another class of properties that we tested is *adversarial robustness* properties. DNNs have been shown to be susceptible to adversarial inputs [69]: correctly classified inputs that an adversary slightly perturbs, leading to their misclassification by the network. Adversarial robustness is thus a safety consideration, and adversarial inputs can be used to train the network further, making it more robust [23]. There exist approaches for finding adversarial inputs [4,23], but the ability to verify their absence is limited.

We say that a network is δ-*locally-robust* at input point $\mathbf{x}$ if for every $\mathbf{x'}$ such that $\|\mathbf{x} - \mathbf{x'}\|_\infty \leq \delta$, the network assigns the same label to $\mathbf{x}$ and $\mathbf{x'}$. In the case of the ACAS Xu DNNs, this means that the same output has the lowest score for both $\mathbf{x}$ and $\mathbf{x'}$. Reluplex can be used to prove local robustness for a given $\mathbf{x}$ and δ, as depicted in Table 3. We used one of the ACAS Xu networks, and tested combinations of 5 arbitrary points and 5 values of δ. SAT results show that Reluplex found an adversarial input within the prescribed neighborhood, and UNSAT results indicate that no such inputs exist. Using binary search on values of δ, Reluplex can thus be used for approximating the optimal δ value up to a desired precision: for example, for point 4 the optimal δ is between 0.025 and 0.05. It is expected that different input points will have different local robustness, and the acceptable thresholds will thus need to be set individually.

Finally, we mention an additional variant of adversarial robustness which we term *global adversarial robustness*, and which can also be solved by Reluplex. Whereas local adversarial robustness is measured for a specific $\mathbf{x}$, global adversarial robustness applies to all inputs simultaneously. This is expressed by encoding two side-by-side copies of the DNN in question, N_1 and N_2, operating on separate input variables $\mathbf{x_1}$ and $\mathbf{x_2}$, respectively, such that $\mathbf{x_2}$

Table 3 Local adversarial robustness tests. All times are in seconds

δ		Point 1	Point 2	Point 3	Point 4	Point 5
0.1	Result	SAT	UNSAT	UNSAT	SAT	UNSAT
	Time	135	5880	863	2	14560
0.075	Result	SAT	UNSAT	UNSAT	SAT	UNSAT
	Time	239	1167	436	977	4344
0.05	Result	SAT	UNSAT	UNSAT	SAT	UNSAT
	Time	24	285	99	1168	1331
0.025	Result	UNSAT	UNSAT	UNSAT	UNSAT	UNSAT
	Time	609	57	53	656	221
0.01	Result	UNSAT	UNSAT	UNSAT	UNSAT	UNSAT
	Time	57	5	1	7	6
Total Time		1064	7394	1452	2810	20462

represents an adversarial perturbation of $\mathbf{x_1}$. We can then check whether $\|\mathbf{x_1} - \mathbf{x_2}\|_\infty \leq \delta$ implies that the two copies of the DNN produce similar outputs. Formally, we require that if N_1 and N_2 assign output a values p_1 and p_2 respectively, then $|p_1 - p_2| \leq \epsilon$. If this holds for every output, we say that the network is ϵ-globally-robust. Global adversarial robustness is harder to prove than the local variant, because encoding two copies of the network results in twice as many ReLU nodes and because the problem is not restricted to a small input domain. We were able to prove global adversarial robustness only on small networks; improving the scalability of this technique is left for future work.

7 Related work

The state of the art prior to this work was limited to verifying fairly small networks. For example, in [59] the authors propose an approach for verifying properties of neural networks with sigmoid activation functions. They replace the activation functions with piecewise linear approximations, and then invoke black-box SMT solvers. When spurious counter-examples are found, the approximation is refined. The authors highlight the difficulty in scaling up this technique and are able to tackle only small networks with at most 20 hidden nodes [60].

Other prior work focused on trading away soundness for better scalability. For example, the authors of [4] propose a technique for finding local adversarial examples in DNNs with ReLUs. Given an input point $\mathbf{x}$, they encode the problem as a linear program and invoke a black-box LP solver. The activation function issue is circumvented by considering a sufficiently small neighborhood of $\mathbf{x}$, in which all ReLUs are fixed at the active or inactive state, making the problem convex. Thus, it is unclear how to address an $\mathbf{x}$ for which one or more ReLUs are on the boundary between active and inactive states. In contrast, Reluplex can be used on input domains for which ReLUs can have more than one possible state. In another paper [27], the authors propose a method for proving the local adversarial robustness of DNNs. For a specific input point $\mathbf{x}$, the authors attempt to prove consistent labeling in a neighborhood of $\mathbf{x}$ by means of discretization: they reduce the infinite neighborhood into a finite set of points, and check that the labeling of these points is consistent. This process is then propagated through the network, layer by layer. While the technique is general in the sense that it is not tailored for a specific activation function, the discretization process

means that any UNSAT result only holds modulo the assumption that the finite sets correctly represent their infinite domains. In contrast, our technique can guarantee that there are no irregularities hiding between the discrete points.

Since this work first appeared [34], there have been several interesting developments in the field, and additional verification approaches have been proposed [50]: constraint-solving based approaches with various heuristics (e.g., [7,13,15,51,70,75]), approaches that analyze DNNs as continuous functions (e.g., [54,63]), and approaches based on abstract interpretation (e.g., [18,65,66,72]). Other researchers have focused on verifying cyber-physical systems that incorporate DNNs as controllers (e.g., [5,12,28,68,74]) and on using verification to explain the behaviors of DNNs to humans (e.g., [9,31]). Yet another line of work attempted to train DNNs that are correct by construction (e.g., [14,25,45,49,61]).

In addition to these advances, the Reluplex algorithm itself has been implemented as part of the open-source Marabou framework [37] and has been extended in various ways. These include: (i) the integration of abstraction-refinement methods [16] and parallelization methods [73] to significantly improve the scalability of the algorithm; the addition of phase two simplex into the algorithm, to solve qualitative questions about neural networks [67]; support for binarized neural networks [1]; and the leveraging of automated invariant inference techniques to verify recurrent neural networks [29]. Reluplex has also been used to verify the adversarial robustness of networks in various contexts [8,24,35,48], to simplify neural networks through the removal of neurons and edges in ways that do not harm their accuracy [20], to verify computer network protocols [38], and to make provably minimal modifications to DNNs in order to remove unwanted behaviors [21].

8 Conclusion and next steps

We presented a novel decision algorithm for solving queries on deep neural networks with ReLU activation functions. The technique is based on extending the simplex algorithm to support the non-convex ReLUs in a way that allows their inputs and outputs to be temporarily inconsistent and then fixed as the algorithm progresses. To guarantee termination, some ReLU connections may need to be split upon—but in many cases this is not required, resulting in an efficient solution. Our success in verifying properties of the ACAS Xu prototype networks indicates that the technique holds much potential for verifying real-world DNNs.

In the future, we plan to increase the technique's scalability. Apart from making engineering improvements to our implementation, we plan to explore better strategies for the application of the Reluplex rules, and to employ advanced conflict analysis techniques for reducing the amount of case-splitting required. Another direction is to provide better soundness guarantees without harming performance, for example by replaying floating-point solutions using precise arithmetic [40], or by producing externally-checkable correctness proofs [36]. Finally, we plan to extend our approach to handle DNNs with additional kinds of layers; specifically, layers that involve activation functions that are not piecewise linear.

Acknowledgements We thank Neal Suchy, Lindsey Kuper, Tim King, Tom Zelazny, and Kishor Jothimurugan for their valuable comments and support. This work was partially supported by a grant from the Intel Corporation.

Appendix

Verifying properties in DNNs with ReLUs is NP-complete

Let N be a DNN with ReLUs and let φ denote a property that is a conjunction of linear constraints on the inputs $\mathbf{x}$ and outputs $\mathbf{y}$ of N, i.e. $\varphi = \varphi_1(\mathbf{x}) \wedge \varphi_2(\mathbf{y})$. We say that φ *is satisfiable on* N if there exists an assignment α for the variables $\mathbf{x}$ and $\mathbf{y}$ such that $\alpha(\mathbf{y})$ is the result of propagating $\alpha(\mathbf{x})$ through N and α satisfies φ.

Claim The problem of determining whether φ is satisfiable on N for a given DNN N and a property φ is NP-complete.

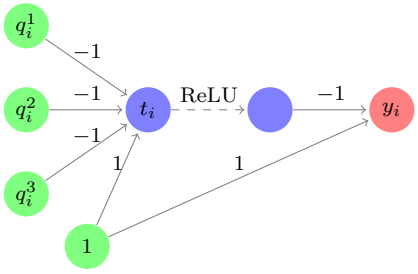
Proof We first show that the problem is in NP. A satisfiability witness is simply an assignment $\alpha(\mathbf{x})$ for the input variables $\mathbf{x}$. This witness can be checked by feeding the values for the input variables forward through the network, obtaining the assignment $\alpha(\mathbf{y})$ for the output values, and checking whether $\varphi_1(\mathbf{x}) \wedge \varphi_2(\mathbf{y})$ holds under the assignment α.

Next, we show that the problem is NP-hard, using a reduction from the 3-SAT problem. We will show how any 3-SAT formula ψ can be transformed into a DNN with ReLUs N and a property φ, such that φ is satisfiable on N if and only if ψ is satisfiable.

Let $\psi = C_1 \wedge C_2 \wedge \ldots \wedge C_n$ denote a 3-SAT formula over variable set $X = \{x_1, \ldots, x_k\}$, i.e. each C_i is a disjunction of three literals $q_i^1 \vee q_i^2 \vee q_i^3$ where the q's are variables from X or their negations. The question is to determine whether there exists an assignment $a : X \to \{0, 1\}$ that satisfies ψ, i.e. that satisfies all the clauses simultaneously.

For simplicity, we first show the construction assuming that the input nodes take the discrete values 0 or 1. Later we will explain how this limitation can be relaxed, so that the only limitation on the input nodes is that they be in the range $[0, 1]$.

We begin by introducing the *disjunction gadget* which, given nodes $q_1, q_2, q_3 \in \{0, 1\}$, outputs a node y_i that is 1 if $q_1 + q_2 + q_3 \geq 1$ and 0 otherwise. The gadget is shown below for the case that the q_i literals are all variables (i.e. not negations of variables):

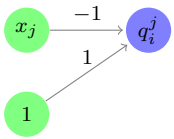


The disjunction gadget can be regarded as calculating the expression

$$y_i = 1 - \max\left(0, 1 - \sum_{j=1}^{3} q_i^j\right)$$

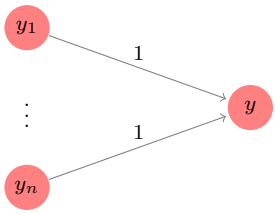
If there is at least one input variable set to 1, y_i will be equal to 1. If all inputs are 0, y_i will be equal to 0. The crux of this gadget is that the ReLU operator allows us to guarantee that even if there are multiple inputs set to 1, the output y_i will still be precisely 1.

In order to handle any negative literals $q_i^j \equiv \neg x_j$, before feeding the literal into the disjunction gadget we first use a *negation gadget*:



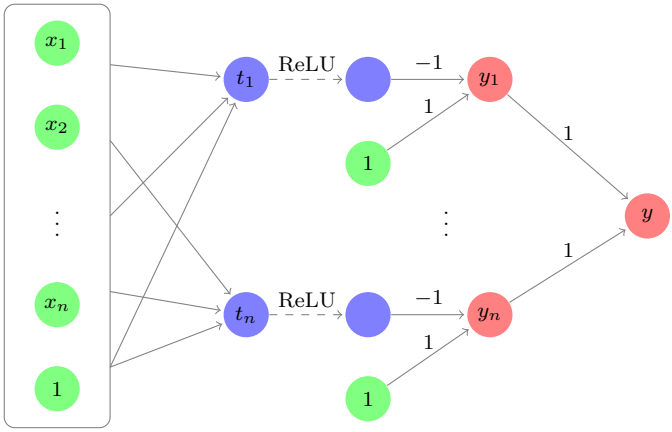
This gadget simply calculates $1 - x_j$, and then we continue as before.

The last part of the construction involves a *conjunction gadget*:



Assuming all nodes $y_1, \ldots, y_n$ are in the domain $\{0, 1\}$, we require that node y be in the range $[n, n]$. Clearly this holds only if $y_i = 1$ for all i.

Finally, in order to check whether all clauses $C_1, \ldots, C_n$ are simultaneously satisfied, we construct a disjunction gadget for each of the clauses (using negation gadgets for their inputs as needed), and combine them using a conjunction gadget:



where the input variables are mapped to each t_i node according to the definition of clause C_i. As we discussed before, node y_i will be equal to 1 if clause C_i is satisfied, and will be 0 otherwise. Therefore, node y will be in the range $[n, n]$ if and only if all clauses are simultaneously satisfied. Consequently, an input assignment $a : X \to \{0, 1\}$ satisfies the input and output constraints on the network if and only if it also satisfies the original ψ, as needed.

The construction above is based on the assumption that we can require that the input nodes take values in the discrete set $\{0, 1\}$, which does not fit our assumption that $\varphi_1(\mathbf{x})$ is a conjunction of linear constraints. We show now how this requirement can be relaxed.

Let $\epsilon > 0$ be a very small number. We set the input range for each variable x_i to be $[0, 1]$, but we will ensure that any feasible solution has $x_i \in [0, \frac{\epsilon}{2}]$ or $x_i \in [1 - \frac{\epsilon}{2}, 1]$. We do this

by adding to the network for each x_i an auxiliary gadget that uses ReLU nodes to compute the expression

$$\max(0, \epsilon - x) + \max(0, x - 1 + \epsilon),$$

and requiring that the output node of this gadget be in the range $[\frac{\epsilon}{2}, \epsilon]$. It is straightforward to show that for $x \in [0, 1]$, this holds if and only if $x \in [0, \frac{\epsilon}{2}]$ or $x \in [1 - \frac{\epsilon}{2}, 1]$.

The disjunction gadgets in our construction then change accordingly. The y_i nodes at the end of each gadget will no longer take just the discrete values $\{0, 1\}$, but instead be in the range $[0, 3 \cdot \frac{\epsilon}{2}]$ if all inputs were in the range $[0, \frac{\epsilon}{2}]$, or in the range $[1 - \frac{\epsilon}{2}, 1]$ if at least one input was in the range $[1 - \frac{\epsilon}{2}, 1]$.

If every input clause has at least one node in the range $[1 - \frac{\epsilon}{2}, 1]$ then all y_i nodes will be in the range $[1 - \frac{\epsilon}{2}, 1]$, and consequently y will be in the range $[n(1 - \frac{\epsilon}{2}), n]$. However, if at least one clause does not have a node in the range $[1 - \frac{\epsilon}{2}, 1]$ then y will be smaller than $n(1 - \frac{\epsilon}{2})$ (for $\epsilon < \frac{2}{n+3}$). Thus, by requiring that $y \in [n(1 - \frac{\epsilon}{2}), n]$, the input and output constraints will be satisfiable on the network if and only if ψ is satisfiable; and the satisfying assignment can be constructed by treating every $x_i \in [0, \frac{\epsilon}{2}]$ as 0 and every $x_i \in [1 - \frac{\epsilon}{2}, 1]$ as 1. □

The Reluplex calculus is sound and complete

We define a *derivation tree* as a tree where each node is a configuration whose children (if any) are obtained by applying to it one of the derivation rules. A derivation tree D *derives* a derivation tree D' if D' is obtained from D by applying exactly one derivation rule to one of D's leaves. A *derivation* is a sequence D_i of derivation trees such that D_0 has only a single node and each D_i derives D_{i+1}. A *refutation* is a derivation ending in a tree, all of whose leaves are UNSAT. A *witness* is a derivation ending in a tree, at least one of whose leaves is SAT. If ϕ is a conjunction of atoms, we say that $\mathcal{D}$ is a derivation from ϕ if the initial tree in $\mathcal{D}$ contains the configuration initialized from ϕ. A calculus is *sound* if, whenever a derivation $\mathcal{D}$ from ϕ is either a refutation or a witness, ϕ is correspondingly unsatisfiable or satisfiable, respectively. A calculus is *complete* if there always exists either a refutation or a witness starting from any ϕ.

In order to prove that the Reluplex calculus is sound, we first prove the following lemmas:

Lemma 1 *Let $\mathcal{D}$ denote a derivation starting from a derivation tree D_0 with a single node $s_0 = \langle \mathcal{B}_0, T_0, l_0, u_0, \alpha_0, R_0 \rangle$. Then, for every derivation tree D_i appearing in $\mathcal{D}$, and for each node $s = \langle \mathcal{B}, T, l, u, \alpha, R \rangle$ appearing in D_i (except for the distinguished nodes SAT and UNSAT), the following properties hold:*

(i) if an assignment satisfies T, then it also satisfies T_0; and
(ii) the assignment α satisfies T (i.e., α satisfies all equations in T).

Proof The proof is by induction on i. For $i = 0$, the claim holds trivially (recall that α_0 assigns every variable to 0). Now, suppose the claim holds for some i and consider D_{i+1}. D_{i+1} is equivalent to D_i except for the addition of one or more nodes added by the application of a single derivation rule d to some node s with tableau T and assignment α. Because s appears in D_i, we know by the induction hypothesis that an assignment that satisfies T also satisfies T_0, and that α satisfies T. Let s' be a new node (not a distinguished node SAT or UNSAT) with tableau T' and assignment α', introduced by the rule d. Note that d cannot be ReluSuccess, $\mathsf{Failure}_1$, or $\mathsf{Failure}_2$, as these introduce only distinguished nodes; note also that

if d is deriveLowerBound or deriveUpperBound then both the tableau and the assignment are unchanged, so both properties are trivially preserved.

Suppose d is Pivot_1, Pivot_2 or PivotForRelu. For any of these rules, $\alpha' = \alpha$ and $T' = pivot(T, i, j)$ for some i and j. Observe that by definition of the *pivot* operation, the equations of T logically entail those of T' and vice versa, and so they are satisfied by exactly the same assignments. Alternatively, suppose d is ReluSplit. For the child node corresponding to the inactive case ($u(x_i) := \min(u(x_i), 0)$, $l(x_j) := \max(l(x_j), 0)$ and $u(x_j) := \min(u(x_j), 0)$), the tableau and assignment are unchanged. For the active case ($l(x_i) := \max(l(x_i), 0)$), the tableau and assignment are changed by the *addEq* operation. This operation adds a single equation with a fresh variable as its left hand side, and then extends the assignment to assign this fresh variable a value that satisfies the new equation; the assignments of all other variables are unchanged. From these observations, both properties follow easily.

The remaining cases are when d is Update, Update_b or Update_f. For these rules, $T' = T$, from which property (i) follows trivially. For property (ii), we first note that $\alpha' = update(\alpha, x_i, \delta)$ for some i and δ. By definition of the *update* operation, because α satisfied the equations of T, α' continues to satisfy these equations and so (because $T' = T$) α' also satisfies T'. □

Lemma 2 *Let $\mathcal{D}$ denote a derivation starting from a derivation tree D_0 with a single node $s_0 = \langle \mathcal{B}_0, T_0, l_0, u_0, \alpha_0, R_0 \rangle$. If there exists an assignment α^* (not necessarily α_0) such that α^* satisfies T_0, for every pair $\langle x_i, x_j \rangle \in R$ it holds that $\alpha^*(x_j) = \max(0, \alpha^*(x_i))$, and $l_0(x_i) \leq \alpha^*(x_i) \leq u_0(x_i)$ for all i, then for each derivation tree D_i appearing in $\mathcal{D}$ at least one of these two properties holds:*

(i) D_i has a `SAT` *leaf.*
(ii) D_i has a leaf $s = \langle \mathcal{B}, T, l, u, \alpha, R \rangle$ (that is not a distinguished node `SAT` *or* `UNSAT`*) such that $l(x_i) \leq \alpha^*(x_i) \leq u(x_i)$ for all i, and α^* satisfies T.*

Proof The proof is again by induction on i. For $i = 0$, property (ii) holds trivially. Now, suppose the claim holds for some i and consider D_{i+1}. D_{i+1} is equivalent to D_i except for the addition of one or more nodes added by the application of a single derivation rule d to a leaf s of D_i.

Due to the induction hypothesis, we know that D_i has a leaf $\bar{s}$ that is either a `SAT` leaf or that satisfies property (ii). If $\bar{s} \neq s$, then $\bar{s}$ also appears as a leaf in D_{i+1}, and the claim holds. We will show that the claim also holds when $\bar{s} = s$. Because none of the derivation rules can be applied to a `SAT` or `UNSAT` node, we know that node s is not a distinguished `SAT` or `UNSAT` node, and we denote $s = \langle \mathcal{B}, T, l, u, \alpha, R \rangle$.

If d is ReluSuccess, D_{i+1} has a `SAT` leaf and property (i) holds. Suppose d is Pivot_1, Pivot_2, PivotForRelu, Update, Update_b or Update_f. In any of these cases, node s has a single child in D_{i+1}, which we denote $s' = \langle \mathcal{B}', T', l', u', \alpha', R' \rangle$. By definition of these derivation rules, $l'(x_j) = l(x_j)$ and $u'(x_j) = u(x_j)$ for all j. Further, T' is either identical or equivalent to T. Because node s satisfies property (ii), we get that s' is a leaf that satisfies property (ii), as needed.

Suppose that d is ReluSplit, applied to a pair $\langle x_i, x_j \rangle \in R$. Node s has two children in D_{i+1}: a state s^- in which the upper bounds for x_i and x_j have been decreased to 0 if they were previously positive, and the lower bound for x_j has been increased to 0 if it was previously negative; and a state s^+ in which the lower bound for x_i has been increased to 0 if it was previously negative, and the tableau has been extended to include the equation $x_j = x_i$. It is straightforward to see that if $\alpha^*(x_i) \leq 0$, then property (ii) holds for s^-; and that if $\alpha^*(x_i) \geq 0$, then property (ii) holds for s^+. In particular, in the latter case, $\alpha^*(x_j) = \max(0, \alpha^*(x_i))$

combined with $\alpha^*(x_i) \geq 0$ implies that the new equation in T, namely $x_j = x_i$, is satisfied (we assume without loss of generality that α^* assigns 0 to all variables introduced by *addEq*). Either way, D_{i+1} has a leaf for which property (ii) holds, as needed.

Next, consider the case where d is deriveLowerBound (the deriveUpperBound case is symmetrical and is omitted). Node s has a single child in D_{i+1}, which we denote $s' = \langle \mathcal{B}', T', l', u', \alpha', R' \rangle$. Because the deriveLowerBound and deriveUpperBound rules cannot be applied to the distinguished SAT node, property (ii) must hold for s. Let x_i denote the variable to which deriveLowerBound was applied. By definition, $l'(x_i) \geq l(x_i)$, and all other variable bounds are unchanged between s and s'. Thus, it suffices to show that $\alpha^*(x_i) \geq l'(x_i)$. Because property (ii) holds for s, α^* satisfies T; and by the induction hypothesis, $l(x_j) \leq \alpha^*(x_j) \leq u(x_j)$ for all j. The fact that $\alpha^*(x_i) \geq l'(x_i)$ then follows directly from the guard condition of deriveLowerBound.

The remaining two cases are when d is the $\mathsf{Failure}_1$ or $\mathsf{Failure}_2$ rule. Because these rules are not applicable to the distinguished SAT node, it follows that property (ii) holds for s. Suppose towards contradiction that in this case, the $\mathsf{Failure}_1$ rule is applicable to some variable x_i, and suppose (without loss of generality) that $\alpha(x_i) < l(x_i)$. By the inductive hypothesis, we know that $l(x_j) \leq \alpha^*(x_j) \leq u(x_j)$ for all j, and by property (ii) we know that α^* satisfies T. Consequently, there must be a variable x_k such that $(T_{i,k} > 0 \wedge \alpha(x_k) < \alpha^*(x_k))$, or $(T_{i,k} < 0 \wedge \alpha(x_k) > \alpha^*(x_k))$. But because all variables under α^* are within their bounds, it follows that $slack^+(x_i) \neq \emptyset$, which is contradictory to the fact that the $\mathsf{Failure}_1$ rule was applicable in s. Next, suppose towards contradiction that the $\mathsf{Failure}_2$ rule is applicable to some variable x_i, i.e. that $l(x_i) > u(x_i)$. This immediately contradicts the fact that $l(x_i) \leq \alpha^*(x_i) \leq u(x_i)$. The claim follows. □

Lemma 3 *Let $\mathcal{D}$ denote a derivation starting from a derivation tree D_0 with a single node $s_0 = \langle \mathcal{B}_0, T_0, l_0, u_0, \alpha_0, R_0 \rangle$. Then, for every derivation tree D_i appearing in $\mathcal{D}$, and for each node $s = \langle \mathcal{B}, T, l, u, \alpha, R \rangle$ appearing in D_i (except for the distinguished nodes SAT and UNSAT), the following properties hold:*

(i) $R = R_0$; and
(ii) $l(x_i) \geq l_0(x_i)$ and $u(x_i) \leq u_0(x_i)$ for all i.

Proof Property (i) follows from the fact that none of the derivation rules (except for ReluSuccess, $\mathsf{Failure}_1$, and $\mathsf{Failure}_2$) changes the set R. Property (ii) follows from the fact that the only rules (except for ReluSuccess, $\mathsf{Failure}_1$, and $\mathsf{Failure}_2$) that update lower and upper variable bounds are deriveLowerBound, deriveUpperBound, and ReluSplit, and that these rules can only increase lower bounds or decrease upper bounds.

We are now ready to prove that the Reluplex calculus is sound and complete.

Claim The Reluplex calculus is sound.

Proof We begin with the satisfiable case. Let $\mathcal{D}$ denote a witness for ϕ. By definition, the final tree D in $\mathcal{D}$ has a SAT leaf. Let $s_0 = \langle \mathcal{B}_0, T_0, l_0, u_0, \alpha_0, R_0 \rangle$ denote the initial state of D_0, and let $s = \langle \mathcal{B}, T, l, u, \alpha, R \rangle$ denote a state in D to which the ReluSuccess rule was applied (i.e., a predecessor of a SAT leaf).

By Lemma 1, α satisfies T_0. Also, by the guard conditions of the ReluSuccess rule, $l(x_i) \leq \alpha(x_i) \leq u(x_i)$ for all i. By property (ii) of Lemma 3, this implies that $l_0(x_i) \leq \alpha(x_i) \leq u_0(x_i)$ for all i. Consequently, α satisfies every linear inequality in ϕ. Finally, we observe that by the conditions of the ReluSuccess rule, α satisfies all the ReLU constraints of s. From property (i) of Lemma 3, it follows that α also satisfies the ReLU constraints of s_0, which are precisely

the ReLU constraints in ϕ. We conclude that α satisfies every constraint in ϕ, and hence ϕ is satisfiable, as needed.

For the unsatisfiable case, it suffices to show that if ϕ is satisfiable then there cannot exist a refutation for it. This is a direct result of Lemma 2: if ϕ is satisfiable, then there exists an assignment α^* that satisfies the initial tableau T_0, and for which all variables are within bounds and all ReLU constraints are satisfied. Hence, Lemma 2 implies that any derivation tree in any derivation $\mathcal{D}$ from ϕ must have a leaf that is not the distinguished UNSAT leaf. It follows that there cannot exist a refutation for ϕ. □

Claim The Reluplex calculus is complete.

Proof Having shown that the Reluplex calculus is sound, it suffices to show a strategy for deriving a witness or a refutation for every ϕ within a finite number of steps. As mentioned in Sect. 3, one such strategy involves two steps: (i) Eagerly apply the ReluSplit rule, once for each ReLU in R; and (ii) For every leaf of the resulting derivation tree, apply the simplex rules Pivot_1, Pivot_2, Update, Failure_1, and Failure_2, and the Reluplex rule ReluSuccess, in a way that guarantees a SAT or an UNSAT configuration is reached within a finite number of steps.

Let D denote the derivation tree obtained after step (i). In every leaf s of D, all ReLU connections have been eliminated, meaning that the variable bounds and equations force each ReLU connection to be either active or inactive. This means that every such s can be regarded as a pure simplex problem, and that any solution to that simplex problem is guaranteed to satisfy also the ReLU constraints in s.

The existence of a terminating simplex strategy for deciding the satisfiability of each leaf of D follows from the completeness of the simplex calculus [71]. One such widely used strategy is *Bland's Rule* [71]. We observe that although the simplex Success rule does not exist in Reluplex, it can be directly substituted with the ReluSuccess rule. This is so because, having applied the ReluSplit rule on each of the ReLUs, any assignment that satisfies the variable bounds in s also satisfies the ReLU constraints in s.

It follows that for every ϕ, we can produce a witness or a refutation, as needed. □

Encoding ReLUs for SMT and LP solvers

We demonstrate the encoding of ReLU nodes that we used for the evaluation conducted using SMT and LP solvers. Let $y = \text{ReLU}(x)$. In the SMTLIB format, used by all SMT solvers that we tested, ReLUs were encoded using an if-then-else construct:

```
(assert (= y (ite (>= x 0) x 0)))
```

In LP format this was encoded using mixed integer programming. Using Gurobi's built-in Boolean type, we defined for every ReLU connection a pair of Boolean variables, b_{on} and b_{off}, and used them to encode the two possible states of the connection. Taking M to be a very large positive constant, we used the following assertions:

$b_{on} + b_{off} = 1$
y >= 0
x - y - M*b_{off} <= 0
x - y + M*b_{off} >= 0
y - M*b_{on} <= 0
x - M*b_{on} <= 0

When $b_{on} = 1$ and $b_{off} = 0$, the ReLU connection is in the active state; and otherwise, when $b_{on} = 0$ and $b_{off} = 1$, it is in the inactive state.

In the active case, because $b_{off} = 0$ the third and fourth equations imply that $x = y$ (observe that y is always non-negative). M is very large, and can be regarded as ∞; hence, because $b_{on} = 1$, the last two equations merely imply that $x, y \leq \infty$, and so pose no restriction on the solution.

In the inactive case, $b_{on} = 0$, and so the last two equations force $y = 0$ and $x \leq 0$. In this case $b_{off} = 1$ and so the third and fourth equations pose no restriction on the solution.

Formal definitions for properties $\phi_1,\ldots,\phi_{10}$

The units for the ACAS Xu DNNs' inputs are:

- ρ: feet.
- θ, ψ: radians.
- v_{own}, v_{int}: feet per second.
- τ: seconds.

θ and ψ are measured counter clockwise, and are always in the range $[-\pi, \pi]$. In line with the discussion in Sect. 5, the family of 45 ACAS Xu DNNs are indexed according to the previous action a_{prev} and time until loss of vertical separation τ. The possible values for these two indices are:

1. a_{prev}: [Clear-of-Conflict, weak left, weak right, strong left, strong right].
2. τ: [0, 1, 5, 10, 20, 40, 60, 80, 100].

We use $N_{x,y}$ to denote the network trained for the x-th value of a_{prev} and y-th value of τ. For example, $N_{2,3}$ is the network trained for the case where a_{prev} = weak left and $\tau = 5$. Using this notation, we now give the formal definition of each of the properties $\phi_1, \ldots, \phi_{10}$ that we tested.

Property ϕ_1

- Description: If the intruder is distant and is significantly slower than the ownship, the score of a COC advisory will always be below a certain fixed threshold.
- Tested on: all 45 networks.
- Input constraints: $\rho \geq 55947.691$, $v_{own} \geq 1145$, $v_{int} \leq 60$.
- Desired output property: the score for COC is at most 1500.

Property ϕ_2

- Description: If the intruder is distant and is significantly slower than the ownship, the score of a COC advisory will never be maximal.
- Tested on: $N_{x,y}$ for all $x \geq 2$ and for all y.
- Input constraints: $\rho \geq 55947.691$, $v_{own} \geq 1145$, $v_{int} \leq 60$.
- Desired output property: the score for COC is not the maximal score.

Property ϕ_3

- Description: If the intruder is directly ahead and is moving towards the ownship, the score for COC will not be minimal.
- Tested on: all networks except $N_{1,7}$, $N_{1,8}$, and $N_{1,9}$.
- Input constraints: $1500 \leq \rho \leq 1800$, $-0.06 \leq \theta \leq 0.06$, $\psi \geq 3.10$, $v_{\text{own}} \geq 980$, $v_{\text{int}} \geq 960$.
- Desired output property: the score for COC is not the minimal score.

Property ϕ_4

- Description: If the intruder is directly ahead and is moving away from the ownship but at a lower speed than that of the ownship, the score for COC will not be minimal.
- Tested on: all networks except $N_{1,7}$, $N_{1,8}$, and $N_{1,9}$.
- Input constraints: $1500 \leq \rho \leq 1800$, $-0.06 \leq \theta \leq 0.06$, $\psi = 0$, $v_{\text{own}} \geq 1000$, $700 \leq v_{\text{int}} \leq 800$.
- Desired output property: the score for COC is not the minimal score.

Property ϕ_5

- Description: If the intruder is near and approaching from the left, the network advises "strong right".
- Tested on: $N_{1,1}$.
- Input constraints: $250 \leq \rho \leq 400$, $0.2 \leq \theta \leq 0.4$, $-3.141592 \leq \psi \leq -3.141592 + 0.005$, $100 \leq v_{\text{own}} \leq 400$, $0 \leq v_{\text{int}} \leq 400$.
- Desired output property: the score for "strong right" is the minimal score.

Property ϕ_6

- Description: If the intruder is sufficiently far away, the network advises COC.
- Tested on: $N_{1,1}$.
- Input constraints: $12000 \leq \rho \leq 62000$, $(0.7 \leq \theta \leq 3.141592) \vee (-3.141592 \leq \theta \leq -0.7)$, $-3.141592 \leq \psi \leq -3.141592 + 0.005$, $100 \leq v_{\text{own}} \leq 1200$, $0 \leq v_{\text{int}} \leq 1200$.
- Desired output property: the score for COC is the minimal score.

Property ϕ_7

- Description: If vertical separation is large, the network will never advise a strong turn.
- Tested on: $N_{1,9}$.
- Input constraints: $0 \leq \rho \leq 60760$, $-3.141592 \leq \theta \leq 3.141592$, $-3.141592 \leq \psi \leq 3.141592$, $100 \leq v_{\text{own}} \leq 1200$, $0 \leq v_{\text{int}} \leq 1200$.
- Desired output property: the scores for "strong right" and "strong left" are never the minimal scores.

Property ϕ_8

- Description: For a large vertical separation and a previous "weak left" advisory, the network will either output COC or continue advising "weak left".
- Tested on: $N_{2,9}$.
- Input constraints: $0 \leq \rho \leq 60760$, $-3.141592 \leq \theta \leq -0.75 \cdot 3.141592$, $-0.1 \leq \psi \leq 0.1$, $600 \leq v_{\text{own}} \leq 1200$, $600 \leq v_{\text{int}} \leq 1200$.
- Desired output property: the score for "weak left" is minimal or the score for COC is minimal.

Property ϕ_9

- Description: Even if the previous advisory was "weak right", the presence of a nearby intruder will cause the network to output a "strong left" advisory instead.
- Tested on: $N_{3,3}$.
- Input constraints: $2000 \leq \rho \leq 7000$, $-0.4 \leq \theta \leq -0.14$, $-3.141592 \leq \psi \leq -3.141592 + 0.01$, $100 \leq v_{\text{own}} \leq 150$, $0 \leq v_{\text{int}} \leq 150$.
- Desired output property: the score for "strong left" is minimal.

Property ϕ_{10}

- Description: For a far away intruder, the network advises COC.
- Tested on: $N_{4,5}$.
- Input constraints: $36000 \leq \rho \leq 60760$, $0.7 \leq \theta \leq 3.141592$, $-3.141592 \leq \psi \leq -3.141592 + 0.01$, $900 \leq v_{\text{own}} \leq 1200$, $600 \leq v_{\text{int}} \leq 1200$.
- Desired output property: the score for COC is minimal.

References

1. Amir G, Wu H, Barrett C, Katz G (2020) An SMT-based approach for verifying binarized neural networks. Technical Report. arXiv:2011.02948
2. Barrett C, Nieuwenhuis R, Oliveras A, Tinelli C (2006) Splitting on demand in SAT modulo theories. In: Proceedings of 13th international conference on logic for programming, artificial intelligence, and reasoning (LPAR), pp 512–526
3. Barrett C, Sebastiani R, Seshia S, Tinelli C (2009) Satisfiability modulo theories. In: Biere A, Heule MJH, van Maaren H, Walsh T (eds) Handbook of satisfiability. Frontiers in Artificial Intelligence and Applications, chapter 26, vol 185. IOS Press, New York, pp 825–885
4. Bastani O, Ioannou Y, Lampropoulos L, Vytiniotis D, Nori A, Criminisi A (2016) Measuring neural net robustness with constraints. In: Proceedings of 30th conference on neural information processing systems (NIPS)
5. Bastani O, Pu Y, Solar-Lezama A (2018) Verifiable reinforcement learning via policy extraction. In: Proceedings of 32nd conference on neural information processing systems (NeurIPS)
6. Bojarski M, Del Testa D, Dworakowski D, Firner B, Flepp B, Goyal P, Jackel L, Monfort M, Muller U, Zhang J, Zhang X, Zhao J, Zieba K (2016) End to end learning for self-driving cars. Technical Report. arXiv:1604.07316
7. Bunel R, Turkaslan I, Torr P, Kohli P, Kumar M (2017) Piecewise linear neural network verification: a comparative study. Technical Report. arXiv:1711.00455v1
8. Carlini N, Katz G, Barrett C, Dill D (2017) Provably Minimally-distorted adversarial examples. Technical Report. arXiv:1709.10207
9. Choi A, Shi W, Shih A, Darwiche A (2019) Compiling neural networks into tractable boolean circuits. In: Proceedings of 1st AAAI spring symposium on verification of neural networks (VNN)

10. Dantzig G (1963) Linear programming and extensions. Princeton University Press, Princeton
11. Dutertre B, de Moura L (2006) A fast linear-arithmetic solver for DPLL(T). In: Proceedings of 18th international conference on computer aided verification (CAV), pp 81–94
12. Dutta S, Chen X, Sankaranarayanan S (2019) Reachability analysis for neural feedback systems using regressive polynomial rule inference. In: Proceedings of 22nd ACM international conference on hybrid systems: computation and control (HSCC)
13. Dutta S, Jha S, Sanakaranarayanan S, Tiwari A (2018) Output range analysis for deep neural networks. In: Proceedings of 10th NASA formal methods symposium (NFM), pp 121–138
14. Dvijotham K, Stanforth R, Gowal S, Mann T, Kohli P (2018) A dual approach to scalable verification of deep networks. In: Proceedings of conference on uncertainty in artificial intelligence (UAI), pp 550–559
15. Ehlers R (2017) Formal verification of piece-wise linear feed-forward neural networks. In: Proceedings of 15th international symposium on automated technology for verification and analysis (ATVA), pp 269–286
16. Elboher Y, Gottschlich J, Katz G (2020) An abstraction-based framework for neural network verification. In: Proceedings of 32nd international conference on computer aided verification (CAV), pp 43–65
17. Faure G, Nieuwenhuis R, Oliveras A, Rodríguez-Carbonell E (2008) SAT modulo the theory of linear arithmetic: exact, inexact and commercial solvers. In: Proceedings of 11th international conference on theory and applications of satisfiability testing (SAT), pp 77–90
18. Gehr T, Mirman M, Drachsler-Cohen D, Tsankov E, Chaudhuri S, Vechev M (2018) AI2: safety and robustness certification of neural networks with abstract interpretation. In: Proceedings of 39th IEEE symposium on security and privacy (S&P)
19. Glorot X, Bordes A, Bengio Y (2011) Deep sparse rectifier neural networks. In: Proceedings of 14th international conference on artificial intelligence and statistics (AISTATS), pp 315–323
20. Gokulanathan S, Feldsher A, Malca A, Barrett C, Katz G (2020) Simplifying neural networks using formal verification. In: Proceedings of 12th NASA formal methods symposium (NFM), pp 85–93
21. Goldberger B, Adi Y, Keshet J, Katz G (2020) Minimal modifications of deep neural networks using verification. In: Proceedings of 23rd international conference on logic for programming, artificial intelligence and reasoning (LPAR), pp 260–278
22. Goodfellow I, Bengio Y, Courville A (2016) Deep learning. MIT Press, Cambridge
23. Goodfellow I, Shlens J, Szegedy C (2014) Explaining and harnessing adversarial examples. Technical Report. arXiv:1412.6572
24. Gopinath D, Katz G, Păsăreanu C, Barrett C (2018) DeepSafe: a data-driven approach for assessing robustness of neural networks. In: Proceedings of 16th international symposium on automated technology for verification and analysis (ATVA), pp 3–19
25. Gowal S, Dvijotham K, Stanforth R, Bunel R, Qin C, Uesato J, Mann T, Kohli P (2018) On the effectiveness of interval bound propagation for training verifiably robust models. Technical Report. arXiv:1810.12715
26. Hinton G, Deng L, Yu D, Dahl G, Mohamed A, Jaitly N, Senior A, Vanhoucke V, Nguyen P, Sainath T, Kingsbury B (2012) Deep neural networks for acoustic modeling in speech recognition: the shared views of four research groups. IEEE Signal Process Mag 29(6):82–97
27. Huang X, Kwiatkowska M, Wang S, Wu M (2016) Safety verification of deep neural networks. Technical Report. arXiv:1610.06940
28. Ivanov R, Weimer J, Alur R, Pappas G, Lee I (2019) Verisig: verifying safety properties of hybrid systems with neural network controllers. In: Proceedings of 22nd ACM international conference on hybrid systems: computation and control (HSCC)
29. Jacoby Y, Barrett C, Katz G (2020) Verifying recurrent neural networks using invariant inference. In: Proceedings of 18th international symposium on automated technology for verification and analysis (ATVA), pp 57–74
30. Jarrett K, Kavukcuoglu K, LeCun Y (2009) What is the best multi-stage architecture for object recognition? In: Proceedings of 12th IEEE international conference on computer vision (ICCV), pp 2146–2153
31. Jha S (2019) Logic extraction for explainable AI. In: Proceedings of 2nd workshop on formal methods for ML-enabled autonomous systems (FoMLAS)
32. Julian K, Kochenderfer M, Owen M (2019) Deep neural network compression for aircraft collision avoidance systems. J Guid Control Dyn 42(3):598–608
33. Katz G, Barrett C, Dill D, Julian K, Kochenderfer M (2017) Reluplex. https://github.com/guykatzz/ReluplexCav2017
34. Katz G, Barrett C, Dill D, Julian K, Kochenderfer M (2017) Reluplex: an efficient SMT solver for verifying deep neural networks. In: Proceedings of 29th international conference on computer aided verification (CAV), pp 97–117
35. Katz G, Barrett C, Dill D, Julian K, Kochenderfer M (2017) Towards proving the adversarial robustness of deep neural networks. In: Proceedings of 1st workshop on formal verification of autonomous vehicles (FVAV), pp 19–26

36. Katz G, Barrett C, Tinelli C, Reynolds A, Hadarean L (2016) Lazy proofs for DPLL(T)-based SMT solvers. In: Proceedings of 16th international conference on formal methods in computer-aided design (FMCAD), pp 93–100
37. Katz G, Huang D, Ibeling D, Julian K, Lazarus C, Lim R, Shah P, Thakoor S, Wu H, Zeljić A, Dill D, Kochenderfer M, Barrett C (2019) The Marabou framework for verification and analysis of deep neural networks. In: Proceedings of 31st international conference on computer aided verification (CAV), pp 443–452
38. Kazak Y, Barrett C, Katz G, Schapira M (2019) Verifying deep-RL-driven systems. In: Proceedings of 1st ACM SIGCOMM workshop on network meets AI and ML (NetAI), pp 83–89
39. King T (2014) Effective algorithms for the satisfiability of quantifier-free formulas over linear real and integer arithmetic. PhD Thesis
40. King T, Barret C, Tinelli C (2014) Leveraging linear and mixed integer programming for SMT. In: Proceedings of 14th international conference on formal methods in computer-aided design (FMCAD), pp 139–146
41. Kochenderfer M (2015) Decision making under uncertainty: theory and application. In: Optimized airborne collision avoidance, chapter. MIT, pp 259–276
42. Kochenderfer M, Chryssanthacopoulos J (2011) Robust airborne collision avoidance through dynamic programming. Project Report ATC-371, Massachusetts Institute of Technology, Lincoln Laboratory
43. Kochenderfer M, Edwards M, Espindle L, Kuchar J, Griffith J (2010) Airspace encounter models for estimating collision risk. AIAA J Guid Control Dyn 33(2):487–499
44. Kochenderfer M, Holland J, Chryssanthacopoulos J (2012) Next generation airborne collision avoidance system. Lincoln Lab J 19(1):17–33
45. Kolter J, Wong E (2018) Provable defenses against adversarial examples via the convex outer adversarial polytope. In: Proceedings of 16th IEEE international conference on machine learning and applications (ICML)
46. Krizhevsky A, Sutskever I, Hinton G (2012) Imagenet classification with deep convolutional neural networks. In: Advances in neural information processing systems, pp 1097–1105
47. Kuchar J, Drumm A (2007) The traffic alert and collision avoidance system. Lincoln Lab J 16(2):277–296
48. Kuper L, Katz G, Gottschlich J, Julian K, Barrett C, Kochenderfer M (2018) Toward scalable verification for safety-critical deep networks. Technical Report. arXiv:1801.05950
49. Lin X, Zhu H, Samanta R, Jagannathan S (2019) ART: abstraction refinement-guided training for provably correct neural networks. Technical Report. arXiv:1907.10662
50. Liu C, Arnon T, Lazarus C, Strong C, Barrett C, Kochenderfer M (2020) Algorithms for verifying deep neural networks. Found Trends Optim 4
51. Lomuscio A, Maganti L(2017) An approach to reachability analysis for feed-forward ReLU neural networks. Technical Report. arXiv:1706.07351
52. Maas A, Hannun A, Ng A (2013) Rectifier nonlinearities improve neural network acoustic models. In: Proceedings of 30th international conference on machine learning (ICML)
53. Marques-Silva J, Sakallah K (1999) GRASP: a search algorithm for propositional satisfiability. IEEE Trans Comput 48(5):506–521
54. Matthias H, Andriushchenko M (2017) Formal guarantees on the robustness of a classifier against adversarial manipulation. In: Proceedings of 31st conference on neural information processing systems (NeurIPS)
55. Monniaux D (2009) On using floating-point computations to help an exact linear arithmetic decision procedure. In: Proceedings of 21st international conference on computer aided verification (CAV), pp 570–583
56. Nair V, Hinton G (2010) Rectified linear units improve restricted Boltzmann machines. In: Proceedings of 27th international conference on machine learning (ICML), pp 807–814
57. Nieuwenhuis R, Oliveras A, Tinelli C (2006) Solving SAT and SAT modulo theories: from an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). J ACM (JACM) 53(6):937–977
58. Padberg M, Rinaldi G (1991) A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. SIAM Rev 33(1):60–100
59. Pulina L, Tacchella A (2010) An abstraction-refinement approach to verification of artificial neural networks. In: Proceedings of 22nd international conference on computer aided verification (CAV), pp 243–257
60. Pulina L, Tacchella A (2012) Challenging SMT solvers to verify neural networks. AI Commun 25(2):117–135
61. Raghunathan A, Steinhardt J, Liang P (2018) Certified defenses against adversarial examples. In: Proceedings of 6th international conference on learning representations (ICLR)
62. Riesenhuber M, Tomaso P (1999) Hierarchical models of object recognition in cortex. Nat Neurosci 2(11):1019–1025

63. Ruan W, Huang X, Kwiatkowska M (2018) Reachability analysis of deep neural networks with provable guarantees. In: Proceedings of 27th international joint conference on artificial intelligence (IJCAI)
64. Silver D, Huang A, Maddison C, Guez A, Sifre L, Van Den Driessche G, Schrittwieser J, Antonoglou I, Panneershelvam V, Lanctot M, Dieleman S (2016) Mastering the game of go with deep neural networks and tree search. Nature 529(7587):484–489
65. Singh G, Gehr T, Mirman M, Puschel M, Vechev M (2018) Fast and effective robustness certification. In: Proceedings of 32nd conference on neural information processing systems (NeurIPS)
66. Singh G, Gehr T, Puschel M, Vechev M (2019) An abstract domain for certifying neural networks. In: Proceedings of 6th ACM SIGPLAN symposium on principles of programming languages (POPL)
67. Strong C, Wu H, Zeljić A, Julian K, Katz G, Barrett C, Kochenderfer M (2020) Global optimization of objective functions represented by ReLU networks. Technical Report. arXiv:2010.03258
68. Sun X, K H, Shoukry Y (2019) Formal verification of neural network controlled autonomous systems. In: Proceedings of 22nd ACM international conference on hybrid systems: computation and control (HSCC)
69. Szegedy C, Zaremba W, Sutskever I, Bruna J, Erhan D, Goodfellow I, Fergus R (2013) Intriguing properties of neural networks. Technical Report. arXiv:1312.6199
70. Tjeng V, Xiao K, Tedrake R (2017) Evaluating robustness of neural networks with mixed integer programming. Technical Report. arXiv:1711.07356
71. Vanderbei R (1996) Linear programming: foundations and extensions. Springer, Berlin
72. Wang S, Pei K, Whitehouse J, Yang J, Jana S (2018) Formal security analysis of neural networks using symbolic intervals. In: Proceedings of 27th USENIX security symposium
73. Wu H, Ozdemir A, Zeljić A, Irfan A, Julian K, Gopinath D, Fouladi S, Katz G, Păsăreanu C, Barrett C (2020) Parallelization techniques for verifying neural networks. In: Proceedings of 20th international conference on formal methods in computer-aided design (FMCAD), pp 128–137
74. Xiang W, Johnson T (2018) Reachability analysis and safety verification for neural network control systems. Technical Report. arXiv:1805.09944
75. Xiang W, Tran H-D, Johnson T (2018) Output reachable set estimation and verification for multilayer neural networks. IEEE Trans Neural Netw Learn Syst (TNNLS) 99:1–7