

Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks

Guy Katz, Clark Barrett, David Dill, Kyle Julian and Mykel Kochenderfer

Stanford University, USA

{guyk, clarkbarrett, dill, kjulian3, mykel}@stanford.edu



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 verified using existing methods.

1 Introduction

Artificial neural networks [7, 32] have emerged as a promising approach for creating scalable and robust systems. Applications include speech recognition [9], image classification [23], game playing [33], 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 (DNNs)*, 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 [4] and airborne collision avoidance systems for unmanned aircraft (ACAS Xu) [13]. 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 [34]. 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 Section I of the supplementary material [15]). DNN verification is experimentally beyond the reach of general-purpose tools such as *linear programming* (LP) solvers or existing *satisfiability modulo theories* (SMT) solvers [3, 10, 31], and thus far, dedicated tools have only been able to handle very small networks (e.g. a single hidden layer with only 10 to 20 hidden nodes [30, 31]).

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) [27]. 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 [23, 25], and it has been suggested that their piecewise linearity allows DNNs to generalize well to previously unseen inputs [6, 7, 11, 27]. Past efforts at verifying properties of DNNs with ReLUs have had to make significant simplifying assumptions [3, 10] — for instance, by considering only small input regions in which all ReLUs are fixed at either the active or inactive state [3], 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 [13]. 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 [3] 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 (i) present Reluplex, an SMT solver for a theory of linear real arithmetic with ReLU constraints; (ii) show how DNNs and properties of interest can be encoded as inputs to Reluplex; (iii) discuss several implementation details that are crucial to performance and scalability, such as the use of floating-point arithmetic, bound derivation for ReLU variables, and conflict analysis; and (iv) 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 existing techniques.

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

2 Background

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., [7] 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.

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* [7]. Here, we focus on the Rectified Linear Unit (ReLU) activation function [27]. 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 $2, \dots, n - 1$ are the hidden layers.

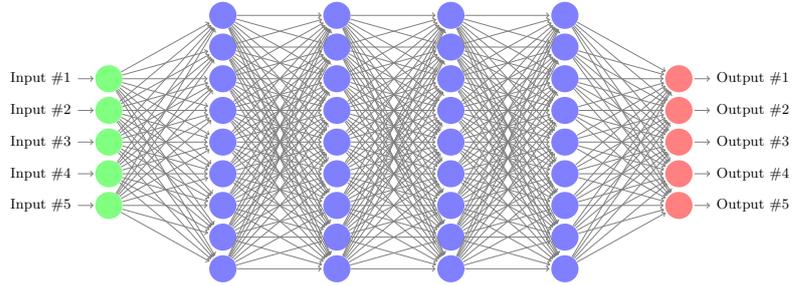


Fig. 1: A fully connected DNN with 5 input nodes (in green), 5 output nodes (in red), and 4 hidden layers containing a total of 36 hidden nodes (in blue).

The value of the j -th node of layer i is denoted $v_{i,j}$ and the column vector $[v_{i,1}, \dots, v_{i,s_i}]^T$ 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 , and its values are given by $V_i = \text{ReLU}(W_i V_{i-1} + B_i)$, with the ReLU function being applied element-wise. This rule is applied repeatedly for each layer until V_n is calculated. When the weight matrices W_1, \dots, W_n do not have any zero entries, the network is said to be *fully connected* (see Fig. 1 for an illustration).

Fig. 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, the network's output is always identical to its input: $v_{31} \equiv v_{11}$.

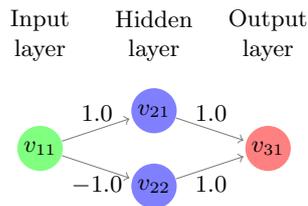


Fig. 2: A small neural network.

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 [28] 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* [1] extends DPLL(T) by allowing theory solvers to delegate case-splitting to the SAT engine in a generic and modular way. In Section 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.

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 [5] 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., [35]). The rules of the calculus operate over data structures we call *configurations*. For a given set of variables $\mathcal{X} = \{x_1, \dots, x_n\}$, a simplex configuration is either one of the distinguished symbols $\{\text{SAT}, \text{UNSAT}\}$ 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

¹ Consistent with most treatments of SMT, we assume many-sorted first-order logic with equality as our underlying formalism (see, e.g., [2] for details).

² There exist SMT-friendly extensions of simplex (see e.g. [17]) 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).

$$\begin{array}{l}
\text{Pivot}_1 \quad \frac{x_i \in \mathcal{B}, \quad \alpha(x_i) < l(x_i), \quad x_j \in \text{slack}^+(x_i)}{T := \text{pivot}(T, i, j), \quad \mathcal{B} := \mathcal{B} \cup \{x_j\} \setminus \{x_i\}} \\
\text{Pivot}_2 \quad \frac{x_i \in \mathcal{B}, \quad \alpha(x_i) > u(x_i), \quad x_j \in \text{slack}^-(x_i)}{T := \text{pivot}(T, i, j), \quad \mathcal{B} := \mathcal{B} \cup \{x_j\} \setminus \{x_i\}} \\
\text{Update} \quad \frac{x_j \notin \mathcal{B}, \quad \alpha(x_j) < l(x_j) \vee \alpha(x_j) > u(x_j), \quad l(x_j) \leq \alpha(x_j) + \delta \leq u(x_j)}{\alpha := \text{update}(\alpha, x_j, \delta)} \\
\text{Failure} \quad \frac{x_i \in \mathcal{B}, \quad (\alpha(x_i) < l(x_i) \wedge \text{slack}^+(x_i) = \emptyset) \vee (\alpha(x_i) > u(x_i) \wedge \text{slack}^-(x_i) = \emptyset)}{\text{UNSAT}} \\
\text{Success} \quad \frac{\forall x_i \in \mathcal{X}. l(x_i) \leq \alpha(x_i) \leq u(x_i)}{\text{SAT}}
\end{array}$$

Fig. 3: Derivation rules for the abstract simplex algorithm.

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).

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 $x_i = \sum_{x_k \notin \mathcal{B}} c_k x_k$. 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 $\text{pivot}(T, i, j)$ 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 $\text{update}(\alpha, x_j, \delta)$ 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:

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

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 **Pivot**₁ or the **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 **Failure** rule applies and the problem is unsatisfiable. 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* [35] (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 [35]. Variable selection strategies are also known to have a dramatic effect on performance [35]. 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 Section 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 [3, 10], 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 Section 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: $\text{ReLU}(x, y)$ 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 $\text{ReLU}(v^b, v^f)$. 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 x 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.

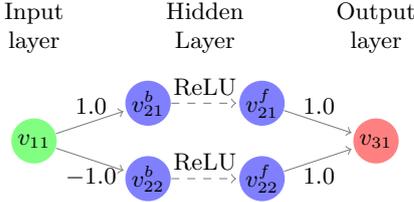


Fig. 4: The network from Fig. 2, with ReLU nodes split into backward- and forward-facing variables.

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 $\mathcal{X} = \{x_1, \dots, x_n\}$, a Reluplex configuration is either one of the distinguished symbols $\{\text{SAT}, \text{UNSAT}\}$ 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 and **Update** are included also in Reluplex, as they are designed to handle out-of-bounds violations. 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**

$$\begin{array}{l}
\text{Update}_b \frac{x_i \notin \mathcal{B}, \langle x_i, x_j \rangle \in R, \alpha(x_j) \neq \max(0, \alpha(x_i)), \alpha(x_j) \geq 0}{\alpha := \text{update}(\alpha, x_i, \alpha(x_j) - \alpha(x_i))} \\
\text{Update}_f \frac{x_j \notin \mathcal{B}, \langle x_i, x_j \rangle \in R, \alpha(x_j) \neq \max(0, \alpha(x_i))}{\alpha := \text{update}(\alpha, x_j, \max(0, \alpha(x_i)) - \alpha(x_j))} \\
\text{PivotForRelu} \frac{x_i \in \mathcal{B}, \exists x_l. \langle x_i, x_l \rangle \in R \vee \langle x_l, x_i \rangle \in R, x_j \notin \mathcal{B}, T_{i,j} \neq 0}{T := \text{pivot}(T, i, j), \mathcal{B} := \mathcal{B} \cup \{x_j\} \setminus \{x_i\}} \\
\text{ReluSplit} \frac{\langle x_i, x_j \rangle \in R, l(x_i) < 0, u(x_i) > 0}{u(x_i) := 0 \quad l(x_i) := 0} \\
\text{ReluSuccess} \frac{\forall x \in \mathcal{X}. l(x) \leq \alpha(x) \leq u(x), \forall \langle x^b, x^f \rangle \in R. \alpha(x^f) = \max(0, \alpha(x^b))}{\text{SAT}}
\end{array}$$

Fig. 5: Additional derivation rules for the abstract Reluplex algorithm.

rule is used for splitting on certain ReLU connections, guessing whether they are active (by setting $l(x_i) := 0$) or inactive (by setting $u(x_i) := 0$).

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 Section II of the supplementary material [15]). In practice, splitting can be managed by a SAT engine with splitting-on-demand [1]. The naïve approach mentioned at the beginning of this section can be simulated by applying the ReluSplit rule eagerly until it no longer applies and then solving each derived sub-problem separately (this reduction trivially guarantees termination just as do branch-and-cut techniques in mixed integer solvers [29]). 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 (see Section III of the supplementary material [15]). Additional details appear in Section 6.

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 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:

variable	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:

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

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:

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

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:

$v_{11} = v_{21}^b - a_1$	variable	v_{11}	v_{21}^b	v_{21}^f	v_{22}^b	v_{22}^f	v_{31}	a_1	a_2	a_3
$v_{22}^b = -v_{21}^b + a_1 + a_2$	lower bound	0	$-\infty$	0	$-\infty$	0	0.5	0	0	0
$v_{21}^f = -v_{22}^f + v_{31} - a_3$	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 three techniques that significantly boost the performance of Reluplex: use of tighter bound derivation, conflict analysis and floating point arithmetic. A fourth technique, under-approximation, is discussed in Section IV of the supplementary material [15].

Tighter Bound Derivation. The simplex and Reluplex procedures naturally lend themselves to deriving tighter variable bounds as the search progresses [17]. 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:

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

The derived bounds can later be used to derive additional, tighter bounds.

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.

Bound tightening operations incur overhead, and simplex implementations often use them sparsely [17]. 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 Section 6.

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 [26].

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. 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 [35].

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

number of Pivot steps we would measure the accumulated roundoff error; and (ii) 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:

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

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 Section 8).

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 [24]. Recent work has focused on creating a new system, known as *Airborne Collision Avoidance System X (ACAS X)* [19, 20]. 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 [19, 20, 22].

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 [13]. 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 [13]. 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 [13]. 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 [21], 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.

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 [20]: (i) ρ : Distance from ownship to intruder; (ii) θ : Angle to intruder relative to ownship heading direction; (iii) ψ : Heading angle of intruder relative to ownship heading direction; (iv) v_{own} : Speed of ownship; (v) v_{int} : Speed of intruder; (vi) τ : Time until loss of vertical separation; and (vii) 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 $1.5^\circ/\text{s}$ and $3.0^\circ/\text{s}$, respectively.

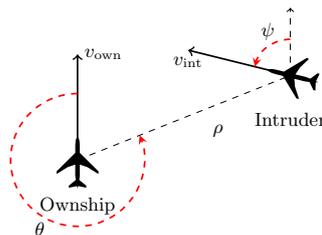


Fig. 6: Geometry for ACAS Xu Horizontal Logic Table

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.

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: (i) The system does not give unnecessary turning advisories; (ii) Alerting regions are uniform and do not contain inconsistent alerts; and (iii) Strong alerts do not appear for high τ values.

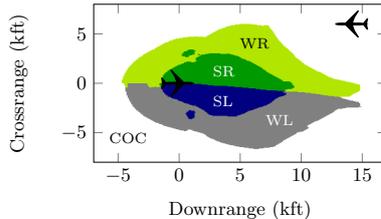


Fig. 7: Advisories for a head-on encounter with $a_{\text{prev}} = \text{COC}$, $\tau = 0\text{s}$.

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: (i) A simplex engine for providing core functionality such as tableau representation and pivot and update operations; (ii) A Reluplex engine for driving the search and performing bound derivation, ReLU pivots and ReLU updates; and (iii) A simple SMT core for providing splitting-on-demand services. For the simplex engine we used the GLPK open-source LP solver³ 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 [14].

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 round-off error (due to the use of double-precision floating point arithmetic) after every

³ www.gnu.org/software/glpk/

5000 Pivot steps, and restored the tableau if 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 $\varphi_1, \dots, \varphi_8$, 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 Section V of the supplementary material [15] for the SMT and LP encodings that we used.

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

Next, we used Reluplex to test a set of 10 quantitative properties ϕ_1, \dots, ϕ_{10} . The properties, described below, are formally defined in Section VI of the supplementary material [15]. 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 (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

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

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 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 [34]: 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 [8]. There exist approaches for finding adversarial inputs [3, 8], 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

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

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

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 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

In [30], the authors propose an approach for verifying properties of neural networks with sigmoid activation functions. They replace the activation functions with piecewise linear approximations thereof, 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 [31].

The authors of [3] 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 a recent paper [10], 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.

Finally, in [12], the authors employ hybrid techniques to analyze an ACAS X controller given in lookup-table form, seeking to identify *safe input regions* in which collisions cannot occur. It will be interesting to combine our technique with that of [12], in order to verify that following the advisories provided by the DNNs indeed leads to collision avoidance.

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 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 [18], or by producing externally-checkable correctness proofs [16]. Finally, we plan to extend our approach to handle DNNs with additional kinds of layers. We speculate that the mechanism we applied to ReLUs can be applied to other piecewise linear layers, such as max-pooling layers.

Acknowledgements. We thank Neal Suchy from the Federal Aviation Administration, Lindsey Kuper from Intel and Tim King from Google for their valuable comments and support. This work was partially supported by a grant from Intel.

References

1. C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Splitting On Demand in SAT Modulo Theories. In *Proc. 13th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, pages 512–526, 2006.
2. C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. Satisfiability Modulo Theories. In A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 26, pages 825–885. IOS Press, 2009.
3. O. Bastani, Y. Ioannou, L. Lampropoulos, D. Vytiniotis, A. Nori, and A. Criminisi. Measuring Neural Net Robustness with Constraints. In *Proc. 30th Conf. on Neural Information Processing Systems (NIPS)*, 2016.
4. M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba. End to End Learning for Self-Driving Cars, 2016. Technical Report. <http://arxiv.org/abs/1604.07316>.
5. G. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963.
6. X. Glorot, A. Bordes, and Y. Bengio. Deep Sparse Rectifier Neural Networks. In *Proc. 14th Int. Conf. on Artificial Intelligence and Statistics (AISTATS)*, pages 315–323, 2011.
7. I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.
8. I. Goodfellow, J. Shlens, and C. Szegedy. Explaining and Harnessing Adversarial Examples, 2014. Technical Report. <http://arxiv.org/abs/1412.6572>.
9. G. Hinton, L. Deng, D. Yu, G. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath, and B. Kingsbury. Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.
10. X. Huang, M. Kwiatkowska, S. Wang, and M. Wu. Safety Verification of Deep Neural Networks, 2016. Technical Report. <http://arxiv.org/abs/1610.06940>.
11. K. Jarrett, K. Kavukcuoglu, and Y. LeCun. What is the Best Multi-Stage Architecture for Object Recognition? In *Proc. 12th IEEE Int. Conf. on Computer Vision (ICCV)*, pages 2146–2153, 2009.
12. J.-B. Jeannin, K. Ghorbal, Y. Kouskoulas, R. Gardner, A. Schmidt, E. Zawadzki, and A. Platzer. A Formally Verified Hybrid System for the Next-Generation Airborne Collision Avoidance System. In *Proc. 21st Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 21–36, 2015.
13. K. Julian, J. Lopez, J. Brush, M. Owen, and M. Kochenderfer. Policy Compression for Aircraft Collision Avoidance Systems. In *Proc. 35th Digital Avionics Systems Conf. (DASC)*, pages 1–10, 2016.
14. G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Reluplex, 2017. <https://github.com/guykatzz/ReluplexCav2017>.
15. G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. Supplementary Material, 2017. <https://arxiv.org/abs/1702.01135>.

16. G. Katz, C. Barrett, C. Tinelli, A. Reynolds, and L. Hadarean. Lazy Proofs for DPLL(T)-Based SMT Solvers. In *Proc. 16th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 93–100, 2016.
17. T. King. *Effective Algorithms for the Satisfiability of Quantifier-Free Formulas Over Linear Real and Integer Arithmetic*. PhD Thesis, 2014.
18. T. King, C. Barret, and C. Tinelli. Leveraging Linear and Mixed Integer Programming for SMT. In *Proc. 14th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 139–146, 2014.
19. M. Kochenderfer. *Decision Making Under Uncertainty: Theory and Application*, chapter Optimized Airborne Collision Avoidance, pages 259–276. MIT, 2015.
20. M. Kochenderfer and J. Chryssanthacopoulos. Robust Airborne Collision Avoidance through Dynamic Programming. Project Report ATC-371, Massachusetts Institute of Technology, Lincoln Laboratory, 2011.
21. M. Kochenderfer, M. Edwards, L. Espindle, J. Kuchar, and J. Griffith. Airspace Encounter Models for Estimating Collision Risk. *AIAA Journal on Guidance, Control, and Dynamics*, 33(2):487–499, 2010.
22. M. Kochenderfer, J. Holland, and J. Chryssanthacopoulos. Next Generation Airborne Collision Avoidance System. *Lincoln Laboratory Journal*, 19(1):17–33, 2012.
23. A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.
24. J. Kuchar and A. Drumm. The Traffic Alert and Collision Avoidance System. *Lincoln Laboratory Journal*, 16(2):277–296, 2007.
25. A. Maas, A. Hannun, and A. Ng. Rectifier Nonlinearities improve Neural Network Acoustic Models. In *Proc. 30th Int. Conf. on Machine Learning (ICML)*, 2013.
26. J. Marques-Silva and K. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.
27. V. Nair and G. Hinton. Rectified Linear Units Improve Restricted Boltzmann Machines. In *Proc. 27th Int. Conf. on Machine Learning (ICML)*, pages 807–814, 2010.
28. R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM (JACM)*, 53(6):937–977, 2006.
29. M. Padberg and G. Rinaldi. A Branch-And-Cut Algorithm for the Resolution of Large-Scale Symmetric Traveling Salesman Problems. *IEEE Transactions on Computers*, 33(1):60–100, 1991.
30. L. Pulina and A. Tacchella. An Abstraction-Refinement Approach to Verification of Artificial Neural Networks. In *Proc. 22nd Int. Conf. on Computer Aided Verification (CAV)*, pages 243–257, 2010.
31. L. Pulina and A. Tacchella. Challenging SMT Solvers to Verify Neural Networks. *AI Communications*, 25(2):117–135, 2012.
32. M. Riesenhuber and P. Tomaso. Hierarchical Models of Object Recognition in Cortex. *Nature Neuroscience*, 2(11):1019–1025, 1999.
33. D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, and S. Dieleman. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529(7587):484–489, 2016.
34. C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing Properties of Neural Networks, 2013. Technical Report. <http://arxiv.org/abs/1312.6199>.
35. R. Vanderbei. *Linear Programming: Foundations and Extensions*. Springer, 1996.