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 \cite{7,29} have emerged as a promising approach for creating scalable and robust systems. Applications include speech recognition \cite{9}, image classification \cite{21}, game playing \cite{30}, and many others. It is now clear that software which 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 \cite{4} and airborne collision avoidance systems for unmanned aircraft (ACAS Xu) \cite{13}. DNNs are trained over a finite set of inputs and outputs and are expected to generalize, i.e. to behave correctly also 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 \cite{31}. 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 Appendix A). 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, 28], 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 [27, 28]).

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) [24]. 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 widely used, and it has been suggested that their piecewise linearity allows DNNs to generalize well to previously unseen inputs [6, 7, 11, 24]. 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 term 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 the aircraft should turn to avoid collision. We were able to successfully prove 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. We were additionally able to prove certain local adversarial robustness properties 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 [24]. 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 \( \text{ReLU}(x) = \max(0, x) \).

A DNN \( N \) can be defined formally by denoting the number of layers \( n \) and denoting the size of layer \( i \) (i.e., the number of its nodes) by \( s_i \). 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 \( s_i \), and its values are given by

\[
V_i = \text{ReLU}(W_i \cdot V_{i-1} + B_i),
\]

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 \( W_1, \ldots, W_n \) 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: \( x_4 \equiv x_1 \).

Satisfiability Modulo Theories. We present our algorithm as a theory solver in the context of satisfiability modulo theories (SMT). A theory is a pair \( T = (\Sigma, I) \) where \( \Sigma \) is a signature and \( I \) is a class of \( \Sigma \)-interpretations, the models of \( T \), that is closed under variable reassignment. A \( \Sigma \)-formula \( \varphi \) is \( T \)-satisfiable (resp., \( T \)-unsatisfiable) if it is satisfied by some (resp., no) interpretation in \( I \). 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 \( T \).

\footnote{Consistent with most treatments of SMT, we assume many-sorted first-order logic with equality as our underlying formalism (see, e.g., \( \PageIndex{2} \) for details).}
Given a theory $T$ with signature $\Sigma$, the DPLL($T$) architecture [25] provides a generic approach for determining the $T$-satisfiability of $\Sigma$-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. The splitting-on-demand framework [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 $T_R$. $T_R$ 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 $T_R$ 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 $\sum_{x_i \in \mathcal{X}} c_i x_i \triangleleft\سب\ d$, for $\triangleleft\سب\in\{=\سب<\سب\geq\سب\}$, 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 $T_R$-satisfiability of conjunctions of linear atoms.\footnote{There exist SMT-friendly extensions of simplex (see e.g. [15]) which can handle $T_R$-satisfiability of arbitrary literals, including strict inequalities and disequalities, but we omit these extensions here for simplicity (and without loss of generality).} Our algorithm extends simplex, and so we begin with an abstract calculus for the original algorithm (for a more thorough description see, e.g., [32]). 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 $\{\text{SAT}, \text{UNSAT}\}$ or a tuple $\langle B, T, l, u, \alpha \rangle$, where: $B \subseteq \mathcal{X}$ is a set of basic variables; $T$, the tableau, contains for each $x_i \in B$ an equation $x_i = \sum_{x_j \in B} c_{ij} x_j$; $l, u$ are mappings that assign each variable $x \in \mathcal{X}$ a lower and an upper bound, respectively; and $\alpha$, the assignment, maps each variable $x \in \mathcal{X}$ to a real value.

![Fig. 2: A small neural network.](image-url)
The initial configuration (and in particular the initial tableau $T_0$) is derived from a conjunction of input atoms as follows: for each atom $\sum_{x_i \in \chi} c_i x_i \geq d$, a new basic variable $b$ is introduced, the equation $b = \sum_{x_i \in \chi} c_i x_i$ is added to the tableau, and $d$ is added as a bound for $b$ (either upper, lower, or both, depending on $\geq$). 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 $B$) as a linear combination of non-basic variables (variables in $\chi \setminus B$). The rows of $T$ correspond to the variables in $B$ and its columns to those of $\chi \setminus B$. For $x_i \in B$ and $x_j \notin B$ we denote by $T_{i,j}$ the coefficient $c_j$ of $x_j$ in the equation $x_i = \sum_{x_j \notin B} c_j x_j$. 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 $x_i = \sum_{x_j \notin B} c_j x_k$ has been replaced by the equation $x_j = \frac{c_i}{c_j} - \sum_{x_k \notin B, k \neq j} \frac{c_i}{c_j} x_k$, 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 (and resulting expressions normalized to obtain the form required by the tableau). The variable assignment $\alpha$ is changed via update operations that are applied to non-basic variables: for $x_j \notin B$, an update$(\alpha,x_j,\delta)$ operation returns an updated assignment $\alpha'$ identical to $\alpha$, except that $\alpha'(x_j) = \alpha(x_j) + \delta$ and for every $x_i \in B$, we have $\alpha'(x_i) = \alpha(x_i) + \delta \cdot T_{i,j}$. To simplify later presentation we also denote:

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

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

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 $\delta$) 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 $\alpha$ continues to satisfy the equations of $T$. Thus, if all
variables are within bounds then the Success rule can be applied, indicating that \( \alpha \) constitutes a satisfying assignment for the original problem.

Based on well-known properties of the simplex algorithm \cite{32}, it can be shown that the calculus is sound (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 \cite{32}. Variable selection strategies are also known to have a dramatic effect on performance \cite{32}. We also note that the version of simplex described above is usually referred to as phase 1 simplex, and is usually followed by a phase 2 in which the solution is optimized according to a cost function. However, as we are only considering satisfiability, phase 2 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 \cite{3,10}, this theoretical

\[
\begin{align*}
\text{Pivot}_1 & \quad x_i \in B, \quad \alpha(x_i) < l(x_i), \quad x_j \in \text{slack}^+(x_i) \\
T & := \text{pivot}(T, i, j), \quad B := B \cup \{x_j\} \setminus \{x_i\}
\end{align*}
\]

\[
\begin{align*}
\text{Pivot}_2 & \quad x_i \in B, \quad \alpha(x_i) > u(x_i), \quad x_j \in \text{slack}^-(x_i) \\
T & := \text{pivot}(T, i, j), \quad B := B \cup \{x_j\} \setminus \{x_i\}
\end{align*}
\]

\[
\begin{align*}
\text{Update} & \quad x_j \notin B, \quad \alpha(x_j) < l(x_j) \lor \alpha(x_j) > u(x_j), \quad l(x_j) \leq \alpha(x_j) + \delta \leq u(x_j) \\
\alpha & := \text{update}(\alpha, x_j, \delta)
\end{align*}
\]

\[
\begin{align*}
\text{Failure} & \quad x_i \in B, \quad (\alpha(x_i) < l(x_i) \land \text{slack}^+(x_i) = \emptyset) \lor (\alpha(x_i) > u(x_i) \land \text{slack}^-(x_i) = \emptyset) \\
\text{UNSAT}
\end{align*}
\]

\[
\begin{align*}
\text{Success} & \quad \forall x_i \in \mathcal{X}, \quad l(x_i) \leq \alpha(x_i) \leq u(x_i) \\
\text{SAT}
\end{align*}
\]
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 $\mathcal{T}_R$ to a theory $\mathcal{T}_{RR}$ of reals and ReLUs. $\mathcal{T}_{RR}$ is almost identical to $\mathcal{T}_R$, except that its signature additionally includes the binary predicate ReLU with the interpretation: 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 a conjunction of $\mathcal{T}_{RR}$-atoms. The main idea is to encode a single ReLU node $x$ as a pair of variables, $x^b$ and $x^f$, and then assert ReLU($x^b,x^f$). $x^b$, the backward-facing variable associated with $x$, is used to express the connection of $x$ to nodes from the preceding layer; whereas $x^f$, 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.](image)

**The Reluplex Procedure.** Reluplex is like simplex in that it iteratively looks for a feasible variable assignment. As with simplex, Reluplex allows variables to temporarily violate their bounds. 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 $(B, T, l, u, \alpha, R)$, where $B, T, l, u$ and $\alpha$ are as before, and $R \subseteq \mathcal{X} \times \mathcal{X}$ is the set of ReLU connections. The initial configuration for a conjunction of atoms is also obtained as before except that $(x, y) \in R$ iff ReLU($x,y$) is an atom. The simplex transition rules Pivot1, Pivot2 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\textsubscript{b} and Update\textsubscript{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\textsubscript{b} or Update\textsubscript{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 certain ReLU connections, guessing whether they are active (by setting $l(x_i) := 0$) or inactive (by setting $u(x_i) := 0$).

Introducing such a splitting rule means that derivations are no longer linear. Using an appropriate generalization of derivations to derivation trees, we can show that Reluplex is sound and complete (see Appendix B for details). In practice, splitting can be managed by a SAT engine using the splitting-on-demand framework \cite{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 \cite{26}). However, as mentioned, this strategy does not scale. A better strategy is to try to fix broken ReLU pairs using the Update\textsubscript{b} and Update\textsubscript{f} rules first, splitting only when the number of updates to a specific ReLU pair exceeds some threshold. Intuitively, this strategy is likely to limit splits to only those ReLU pairs that are “problematic”, while still guaranteeing termination (see Appendix C). We explain our strategy in more detail 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 $x_1 \in [0, 1]$ and $x_4 \in [0.5, 1]$. As we know that the network outputs its input unchanged ($x_4 \equiv x_1$), we expect Reluplex to be able to derive SAT. The initial Reluplex configuration is obtained by introducing new
basic variables \(x_5, x_6, x_7\), and encoding the network with the equations:

\[
x_5 = -x_1 + x_2^b \\
x_6 = x_1 + x_3^b + x_4 \\
x_7 = -x_2^f - x_3^f + x_4
\]

The equations above form the initial tableau \(T_0\), and the initial set of basic variables is \(B = \{x_5, x_6, x_7\}\). The set of ReLU connections is \(R = \{(x_5^b, x_2^f), (x_3^b, x_4^f)\}\). 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:

\[
\begin{array}{c|ccccccccc}
\text{variable} & x_1 & x_2^b & x_2^f & x_3^b & x_3^f & x_4 & x_5 & x_6 & x_7 \\
\hline
\text{lower bound} & 0 & -\infty & 0 & -\infty & 0 & 0.5 & 0 & 0 & 0 \\
\text{assignment} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{upper bound} & 1 & \infty & \infty & \infty & 1 & 0 & 0 & 0 & 0
\end{array}
\]

Starting from this initial configuration, our search strategy is to first fix any out-of-bounds variables (using the original simplex rules). Variable \(x_4\) is non-basic and is out of bounds, so we perform an Update step and set it to 0.5. As a result, \(x_7\), which depends on \(x_4\), is also set to 0.5. \(x_7\) is now out of bounds and is also basic, so we pivot it with \(x_2^f\), and then update \(x_7\) back to 0. After these steps, the tableau consists of the equations:

\[
x_2^f = -x_3^f + x_4 - x_7 \\
x_5 = -x_1 + x_2^b \\
x_6 = x_1 + x_3^b \]

And the assignment is \(\alpha(x_2^f) = 0.5, \alpha(x_4) = 0.5,\) and \(\alpha(x) = 0\) for all other variables \(x\). At this point all variables are within their bounds, but the ReLuSuccess rule does not apply because \(\alpha(x_2^f) = 0.5 \neq 0 = \max(0, \alpha(x_2^f))\).

The next step is to fix the broken ReLU pair \((x_2^b, x_2^f)\). Since \(x_2^b\) is non-basic, we use Update, to increase its value by 0.5. The assignment becomes \(\alpha(x_2^b) = 0.5, \alpha(x_4) = 0.5, \alpha(x_5) = 0.5,\) and \(\alpha(x) = 0\) for all other variables \(x\). All ReLU constraints hold, but variable \(x_3\) is now out of bounds. This can be fixed by pivoting \(x_5\) with \(x_1\) and then updating it. The resulting tableau is:

\[
x_1 = x_2^b - x_5 \\
x_2^f = -x_3^f + x_4 - x_7 \\
x_6 = x_2^b + x_3^b - x_5
\]

Observe that because \(x_1\) is now basic, it was eliminated from the equation for \(x_6\) and replaced with \(x_2^b - x_5\). The non-zero assignments are now \(\alpha(x_1) = 0.5, \alpha(x_2^b) = 0.5, \alpha(x_4) = 0.5, \alpha(x_5) = 0.5\). Variable \(x_6\) is now too large, and so we have a final round of pivot-and-update: \(x_6\) is pivoted with \(x_3^b\) and then updated back to 0. The final tableau and assignments are:

\[
\begin{array}{c|ccccccccc}
\text{variable} & x_1 & x_2^b & x_2^f & x_3^b & x_3^f & x_4 & x_5 & x_6 & x_7 \\
\hline
\text{lower bound} & 0 & -\infty & 0 & -\infty & 0 & 0.5 & 0 & 0 & 0 \\
\text{assignment} & 0.5 & 0.5 & -0.5 & 0.5 & 0 & 0 & 0 & 0 & 0 \\
\text{upper bound} & 1 & \infty & \infty & \infty & 1 & 0 & 0 & 0 & 0
\end{array}
\]
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 floating point arithmetic, tighter bound derivation, conflict analysis and bound under-approximation.

Floating Point Arithmetic. SMT solvers typically use precise (as opposed to floating point) arithmetic to avoid roundoff errors and guarantee soundness. Unfortunately, precise arithmetic is very slow. 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 [32].

To provide increased confidence that any roundoff error remained within an acceptable range, we also added the following safeguards: (i) After a specific number of Pivot steps, we measure the current accumulated roundoff error. (ii) If the error exceeds a threshold $M$, we 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 how much these values differ from the current assignment $\alpha(x_i)$. We define the cumulative roundoff error as:

$$\sum_{x_i \in B_0} |\alpha(x_i) - \sum_{x_j \notin B_0} T_{0,j} \cdot \alpha(x_j)|$$

$T$ 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).
Tighter Bound Derivation. The simplex and Reluplex procedures naturally lend themselves to deriving tighter variable bounds as the search progress \[15\]. Consider a basic variable \(x_i \in \mathcal{B}\) and let 
\[
\text{pos}(x_i) = \{x_j \in \mathcal{B} | T_{i,j} > 0\} \quad \text{and} \quad 
\text{neg}(x_i) = \{x_j \notin \mathcal{B} | T_{i,j} < 0\}.
\]
Throughout the execution, the following rules can be used to derive tighter bounds for \(x_i\), regardless of the current assignment:

\[
\text{deriveLowerBound}\quad \text{if } x_i \in \mathcal{B}, \quad l(x_i) < \sum_{x_j \in \text{pos}(x_i)} T_{i,j} \cdot l(x_j) + \sum_{x_j \in \text{neg}(x_i)} T_{i,j} \cdot u(x_j)
\]

\[
\text{deriveUpperBound}\quad \text{if } x_i \in \mathcal{B}, \quad u(x_i) > \sum_{x_j \in \text{pos}(x_i)} T_{i,j} \cdot u(x_j) + \sum_{x_j \in \text{neg}(x_i)} T_{i,j} \cdot l(x_j)
\]

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 \[15\]. 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 \textsc{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 \[23\].

Bound Under-Approximation. The Reluplex procedure can be enhanced with under-approximations 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 \( \epsilon \). 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 \(^{22}\). Recent work has focused on creating a new system, known as Airborne Collision Avoidance System X (ACAS X) \(^{17,18}\). 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 \(^{17,18,20}\).

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 finite and 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 \(^{19}\), 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 \(^{18}\).
(i) $\rho$: Distance from ownship to intruder. (ii) $\theta$: Angle to intruder relative to ownship heading direction. (iii) $\psi$: Heading angle of intruder relative to ownship heading direction. (iv) $v_{own}$: Speed of ownship. (v) $v_{int}$: Speed of intruder. (vi) $\tau$: Time until loss of vertical separation. (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°/s and 3.0°/s, respectively.

The array of 45 DNNs was produced by discretizing $\tau$ 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 network’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. (iii) Strong alerts do not appear for high $\tau$ values.

## 6 Evaluation

We used a proof-of-concept implementation to evaluate the ability of Reluplex to check realistic properties on the 45 ACAS Xu DNNs. Our implementation
Our search strategy was to repeatedly fix any out-of-bounds violations first, and only then correct any violated ReLU constraints (which could introduce 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 the Update_b or Update_f rules or pivoted via the PivotForRelu rule, and split only when this number reached 5 (a number empirically determined to work well). We also implemented conflict analysis and back-jumping. Bound underapproximation was implemented, but was not used for the evaluation discussed in this section. 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 the error exceeded $10^{-6}$. Most of the 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. 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 $\phi_1, \ldots, \phi_8$.

---

*www.gnu.org/software/glpk/

*For the final version of the paper we will include a link to our implementation online, together with all of the experiments described in this section.
each of the form $x \geq c$ for a fixed output variable $x$ and a constant $c$. The results are depicted in Table 1. 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 also to their use of precise arithmetic. Gurobi did solve 3 of the 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 Appendix D 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 Appendix E. Table 2 depicts for each property the number of tested networks (specified as part of the property), the test results, and the test 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 $\phi_1$ 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 $\phi_2$ 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 $\phi_3$ and $\phi_4$ deal with situations where the intruder is directly ahead and is either moving towards the ownship ($\phi_3$) or away from the ownship but at a lower speed ($\phi_4$), and state that the DNN will never issue a COC advisory.

Properties $\phi_5$ through $\phi_{10}$ each involve a single network, and check for consistent behavior in a specific input region. $\phi_5$ states that if the intruder is near and
| φ   | Networks | Result | Time  | Stack | Splits |
|-----|----------|--------|-------|-------|--------|
| φ₁  |          | 38     | 278403| 45    | 1090082|
|     |          |        |       |       | 7      |
| φ₂  |          | 35     | 82419 | 44    | 284515 |
| φ₃  |          | 42     | 28156 | 22    | 52080  |
| φ₄  |          | 42     | 12475 | 21    | 23940  |
| φ₅  |          | 1      | 19355 | 46    | 58914  |
| φ₆  |          | 1      | 180288| 50    | 548496 |
| φ₇  |          | 1      | 19944 | 49    | 88520  |
| φ₈  |          | 1      | 40102 | 69    | 116697 |
| φ₉  |          | 1      | 99634 | 48    | 227002 |
| φ₁₀ |          | 1      | 19944 | 49    | 88520  |

Table 2: Verifying properties of the ACAS Xu networks.

approaching from the left, the network advises “strong right”; and φ₉ checks that 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. Property φ₇, 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 φ₈ 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 counterexample, 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 local adversarial robustness properties. DNNs have been shown to be susceptible to adversarial inputs [31], correctly classified inputs that an adversary slightly perturbs, leading to their misclassification by the network. Local 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 define a notion of local robustness that can be checked using Reluplex. We say that a network is $\delta$-locally-robust at input point $x$ if for every $x'$ such that $\|x - x'\|_\infty \leq \delta$, the network assigns the same label to $x$ and $x'$. In the case of the ACAS Xu DNNs, this means that the same output has the lowest score for both $x$ and $x'$. Reluplex can be used to prove the absence of adversarial inputs for a given $x$ and a given $\delta$. Table 3 depicts the results of local adversarial robustness checks on one of the ACAS Xu networks. We tested combinations of 5 arbitrary points and 5 values of $\delta$. 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 \( \delta \), Reluplex can thus be used for approximating the optimal \( \delta \) value up to a desired precision: for example, for point 4 the optimal \( \delta \) 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 \textit{global adversarial robustness}, and which can also be solved by Reluplex. Whereas local adversarial robustness is measured for a specific \( x \), 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 \( x_1 \) and \( x_2 \), respectively. We can then check whether \( \| x_1 - x_2 \|_{\infty} \leq \delta \) 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 \( \epsilon \)-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.

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

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

7 Related Work

In [27], 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. Due to the use of approximations, spurious counter-examples are often found and are used to trigger a refinement of the approximation. The authors highlight the difficulty in scaling-up this technique, and are able to tackle only small networks with at most 20 hidden nodes [28].

The authors of [3] propose a technique for finding local adversarial examples in DNNs with ReLUs. Given an input point \( x \), 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 \( x \), in which
all ReLUs are fixed at the active or inactive state, making the problem convex. Thus, it is unclear how to address an $x$ 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 $x$, the authors attempt to prove consistent labeling in a neighborhood of $x$ 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. As the DNNs of [13] were trained on a similar lookup table, the technique of [12] can serve to increase the confidence that the training data was correct. Reluplex can then be used to verify various properties on the DNNs which serve as the actual implementation.

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 non-convex ReLU functions 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.

The main direction that we plan to pursue in the future is increasing the technique’s scalability. Apart from making engineering improvements to our proof-of-concept tool, we plan to explore better strategies and heuristics for the application of the Reluplex rules, and to apply more 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 using floating-point arithmetic to first solve the problem and then replaying the solution using precise arithmetic [16], or by producing externally-checkable proofs of correctness [14].

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, February 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](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. 13th 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](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](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, 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.

15. T. King. *Effective Algorithms for the Satisfiability of Quantifier-Free Formulas Over Linear Real and Integer Arithmetic*. PhD Thesis, 2014.

16. 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.

17. M. Kochenderfer. *Decision Making Under Uncertainty: Theory and Application*, chapter Optimized Airborne Collision Avoidance, pages 259–276. MIT, 2015.

18. M. J. Kochenderfer and J. P. Chryssanthacopoulos. Robust airborne collision avoidance through dynamic programming. Project Report ATC-371, Massachusetts Institute of Technology, Lincoln Laboratory, 2011.

19. M. J. Kochenderfer, M. W. M. Edwards, L. P. Espindle, J. K. Kuchar, and J. D. Griffith. Airspace encounter models for estimating collision risk. *AIAA Journal on Guidance, Control, and Dynamics*, 33(2):487–499, 2010.
20. M. J. Kochenderfer, J. E. Holland, and J. P. Chryssanthacopoulos. Next generation airborne collision avoidance system. *Lincoln Laboratory Journal*, 19(1):17–33, 2012.
21. A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.
22. J. K. Kuchar and A. C. Drumm. The traffic alert and collision avoidance system. *Lincoln Laboratory Journal*, 16(2):277–296, 2007.
23. J. Marques-Silva and K. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.
24. 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.
25. 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.
26. 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.
27. 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.
28. L. Pulina and A. Tacchella. Challenging SMT Solvers to Verify Neural Networks. *AI Communications*, 25(2):117–135, 2012.
29. M. Riesenhuber and P. Tomaso. Hierarchical Models of Object Recognition in Cortex. *Nature Neuroscience*, 2(11):1019–1025, 1999.
30. 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.
31. 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](http://arxiv.org/abs/1312.6199)
32. R. Vanderbei. *Linear Programming: Foundations and Extensions*. Springer, 1996.
Appendix

A Verifying Properties in DNNs with ReLUs is NP-Complete

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

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

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

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 \( \psi \) can be transformed into a DNN with ReLUs \( N \) and a property \( \varphi \), such that \( \varphi \) is satisfiable on \( N \) if and only if \( \psi \) is satisfiable.

Let \( \psi = C_1 \land C_2 \land \ldots \land C_n \) 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 \( q_1^i \lor q_2^i \lor q_3^i \) 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 \( \psi \), 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

\[ y_i = 1 - \max(0, 1 - \sum_{j=1}^{3} q_j^i) \]

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 \( q_j^i \equiv \neg x_j \), before feeding the literal into the disjunction gadget we first use a negation gadget:

\[ x_j \rightarrow \neg x_j \rightarrow q_j^i \]

This gadget simply calculates \( 1 - x_j \), and then we continue as before.

The last part of the construction involves a conjunction gadget:

\[ y_1 \rightarrow 1 \rightarrow \ldots \rightarrow y \rightarrow 1 \rightarrow y_n \]

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 $a : X \rightarrow \{0, 1\}$ satisfies the input and output constraints on the network if and only if it also satisfies the original $\psi$, 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 $\varphi_1(x)$ 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 $x_i \in [0, \epsilon]$ or $x_i \in [1 - \epsilon, 1]$. 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 $[0, \epsilon]$. It is straightforward to show that this holds for $x \in [0, 1]$ if and only if $x \in [0, \epsilon]$ or $x \in [1 - \epsilon, 1]$.

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 $[0, 3 \cdot \epsilon]$ if all inputs were in the range $[0, \epsilon]$, or in the range $[1 - \epsilon, 1]$ if at least one input was in the range $[1 - \epsilon, 1]$.

If every input clause has at least one node in the range $[1 - \epsilon, 1]$ then all $y_i$ nodes will be in the range $[1 - \epsilon, 1]$, and consequently $y$ will be in the range $[n(1 - \epsilon), n]$. However, if at least one clause does not have a node in the range $[1 - \epsilon, 1]$ then $y$ will be smaller than $n(1 - \epsilon)$ (for $\epsilon < \frac{1}{n+3}$). Thus, by requiring that $y \in [n(1 - \epsilon), n]$, the input and output constraints will be satisfiable on the network if and only if $\psi$ is satisfiable; and the satisfying assignment can be constructed by treating every $x_i \in [0, \epsilon]$ as 0 and every $x_i \in [1 - \epsilon, 1]$ as 1. \qed
B 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 $\phi$ is a conjunction of atoms, we say that $D$ is a derivation from $\phi$ if the initial tree in $D$ contains the configuration initialized from $\phi$. A calculus is sound if, whenever a derivation $D$ from $\phi$ is either a refutation or a witness, $\phi$ is correspondingly unsatisfiable or satisfiable, respectively. A calculus is complete if there always exists either a refutation or a witness starting from any $\phi$.

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

Lemma 1. Let $D$ denote a derivation starting from a derivation tree $D_0$ with a single node $s_0 = \langle B_0, T_0, l_0, u_0, \alpha_0, R_0 \rangle$. Then, for every derivation tree $D_i$ appearing in $D$, and for each node $s = \langle B, T, l, u, \alpha, R \rangle$ appearing in $D_i$ (except for the distinguished nodes SAT and UNSAT), the following properties hold:

(i) an assignment satisfies $T_0$ if and only if it satisfies $T$; and
(ii) the assignment $\alpha$ satisfies $T$ (i.e., $\alpha$ satisfies all equations in $T$).

Proof. The proof is by induction on $i$. For $i = 0$, the claim holds trivially (recall that $\alpha_0$ 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 $T$. Because $s$ appears in $D_i$, we know by the induction hypothesis that an assignment satisfies $T_0$ if it satisfies $T$ and that $\alpha$ satisfies $T$. Let $s'$ be a new node (not a distinguished node SAT or UNSAT) with tableau $T'$ and assignment $\alpha'$, introduced by the rule $d$. Note that $d$ cannot be ReluSuccess or Failure as these introduce only distinguished nodes, and that if $d$ is deriveLowerBound, deriveUpperBound, or ReluSplit, 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. From this observation, 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 $\delta$. By definition of the update operation, because $\alpha$ satisfied the equations of $T$, $\alpha'$ continues to satisfy these equations and so (because $T' = T$) $\alpha'$ also satisfies $T'$.

\hfill $\Box$
Lemma 2. Let $D$ denote a derivation starting from a derivation tree $D_0$ with a single node $s_0 = (B_0, T_0, l_0, u_0, \alpha_0, R_0)$. If there exists an assignment $\alpha^*$ (not necessarily $\alpha_0$) such that $\alpha^*$ satisfies $T_0$ 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 $D$ at least one of these two properties holds:

(i) $D_i$ has a SAT leaf.
(ii) $D_i$ has a leaf $s = (B, T, l, u, \alpha, R)$ (that is not a distinguished node SAT or UNSAT) such that $l(x_i) \leq \alpha^*(x_i) \leq u(x_i)$ for all $i$.

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 $s$ that is either a SAT leaf or that satisfies property (ii). If $s \neq s$, then $s$ also appears in $D_{i+1}$, and the claim holds. We will show that the claim also holds when $s = s$. 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 $s = (B, T, l, u, \alpha, R)$.

If $d$ is ReluSuccess, $D_{i+1}$ has a SAT leaf and property (i) holds. Suppose $d$ is Pivot1, Pivot2, PivotForRelu, Update, Update2 or UpdateF. In any of these cases, node $s$ has a single child in $D_{i+1}$, which we denote $s' = (B', T', l', u', \alpha', R')$. By definition of these derivation rules, $l'(x_j) = l(x_j)$ and $u'(x_j) = u(x_j)$ for all $j$. 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 lower bound for $x_i$ is 0, and a state $s^−$ in which the upper bound for $x_i$ is 0. All other lower and upper bounds in $s^+$ and $s^−$ are identical to those of $s$. It is straightforward to see that if $\alpha^*(x_i) \geq 0$ then property (ii) holds for $s^+$, and if $\alpha^*(x_i) \leq 0$ then property (ii) holds for $s^−$. 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 $s' = (B', T', l', u', \alpha', R')$. 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 $\alpha^*$ satisfies $T_0$, it follows from Lemma 1 that it satisfies $T$. 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 only remaining case is when $d$ is the Failure rule. We explain why this case is impossible. Suppose towards contradiction that in node $s$ the Failure rule is applicable to 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 Lemma 1 we know that $\alpha^*$ satisfies $T$. Consequently, there must be a variable $x_k$ such that $(T_1, k > 0 \land \alpha(x_k) < \alpha^*(x_k))$, or $(T_1, k < 0 \land \alpha(x_k) >$
\( \alpha^*(x_i) \)). But because all variables under \( \alpha^* \) are within their bounds, this means that \( \text{slack}^+(x_i) \neq \emptyset \), which is contradictory to the fact that the \text{Failure} rule was applicable in \( s \).

**Lemma 3.** Let \( D \) denote a derivation starting from a derivation tree \( D_0 \) with a single node \( s_0 = (B_0, T_0, l_0, u_0, \alpha_0, R_0) \). Then, for every derivation tree \( D_i \) appearing in \( D \), and for each node \( s = (B, T, l, u, \alpha, R) \) appearing in \( D_i \) (except for the distinguished nodes \text{SAT} and \text{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 \text{ReluSuccess} and \text{Failure}) changes the set \( R \). Property (ii) follows from the fact that the only rules (except for \text{ReluSuccess} and \text{Failure}) that update lower and upper variable bounds are \text{deriveLowerBound} and \text{deriveUpperBound}, respectively, 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 \( D \) denote a witness for \( \phi \). By definition, the final tree \( D_0 \) in \( D \) has a \text{SAT} leaf. Let \( s_0 = (B_0, T_0, l_0, u_0, \alpha_0, R_0) \) denote the initial state of \( D_0 \) and let \( s = (B, T, l, u, \alpha, R) \) denote a state in \( D \) in which the \text{ReluSuccess} rule was applied (i.e., a predecessor of a \text{SAT} leaf).

By Lemma 1, \( \alpha \) satisfies \( T_0 \). Also, by the guard conditions of the \text{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, \( \alpha \) satisfies every linear inequality in \( \phi \).

Finally, we observe that by the conditions of the \text{ReluSuccess} rule, \( \alpha \) satisfies all ReLU constraints of \( s \). From property (i) of Lemma 3, it follows that \( \alpha \) also satisfies the ReLU constraints of \( s_0 \), which are precisely the ReLU constraints in \( \phi \). We conclude that \( \alpha \) satisfies every constraint in \( \phi \), and hence \( \phi \) is satisfiable, as needed.

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

**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 \( \phi \) within a finite number of steps. As mentioned in Section 3, one such strategy involves two steps: (i) Eagerly apply the \text{ReluSplit} rule until it no longer applies; and (ii) For
every leaf of the resulting derivation tree, apply the simplex rules Pivot$_1$, Pivot$_2$, Update, and Failure, 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 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 \cite{32}. One such widely used strategy is Bland’s Rule \cite{32}. 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 to completion, any assignment that satisfies the variable bounds in $s$ also satisfies the ReLU constraints in $s$.

It follows that for every $\phi$ we can produce a witness or a refutation, as needed.

\hfill $\square$

C A Reluplex Strategy that Guarantees Termination

As discussed in Section 6, our strategy for applying the Reluplex rules was to repeatedly fix any out-of-bounds violations first (using the original simplex rules), and only afterwards to correct any violated ReLU constraints using the Update$_b$, Update$_f$ and PivotForRelu rules. If correcting a violated ReLU constraint introduced new out-of-bounds violations, these were again fixed using the simplex rules, and so on.

As mentioned above, 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 become assigned to values within their bounds, or the Failure rule is applicable (and is applied) \cite{32}. By using such a strategy for fixing out-of-bounds violations, and by splitting on a ReLU pair whenever the Update$_b$, Update$_f$ or PivotForRelu rules are applied to it more some fixed number of times, termination is guaranteed.

D 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:

\[(\text{assert } (= y (\text{ite } (>= x 0) x 0)))\]

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:

$$
b_{on} + b_{off} = 1
$$
$$
y \geq 0
$$
$$
x - y - M*b_{off} \leq 0
$$
$$
x - y + M*b_{off} \geq 0
$$
$$
y - M*b_{on} \leq 0
$$
$$
x - M*b_{on} \leq 0
$$

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 $\infty$; 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.

E Formal Definitions for Properties $\phi_1, \ldots, \phi_{10}$

The units for the ACAS Xu DNNs’ inputs are:

- $\rho$: feet.
- $\theta, \psi$: radians.
- $v_{\text{own}}, v_{\text{int}}$: feet per second.
- $\tau$: seconds.

$\theta$ and $\psi$ are measured counter clockwise, and are always in the range $[-\pi, \pi]$.

In line with the discussion in Section 5, the family of 45 ACAS Xu DNNs are indexed according to the previous action $a_{\text{prev}}$ and time until loss of vertical separation $\tau$. The possible values are for these two indices are:

1. $a_{\text{prev}}$: [Clear-of-Conflict, weak left, weak right, strong left, strong right].
2. $\tau$: [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 $a_{\text{prev}}$ and $y$-th value of $\tau$. For example, $N_{2,3}$ is the network trained for the case where $a_{\text{prev}} = \text{weak left}$ 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 $\phi_1$.
- 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 $\phi_2$.
- 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 $\phi_3$.
- 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: $1500 \leq \rho \leq 1800$, $-0.06 \leq \theta \leq 0.06$, $\psi \geq 3.10$, $v_{own} \geq 980$, $v_{int} \geq 960$.
- Desired output property: the score for COC is not the minimal score.

Property $\phi_4$.
- 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: $1500 \leq \rho \leq 1800$, $-0.06 \leq \theta \leq 0.06$, $\psi = 0$, $v_{own} \geq 1000$, $700 \leq v_{int} \leq 800$.
- Desired output property: the score for COC is not the minimal score.

Property $\phi_5$.
- Description: If the intruder is near and approaching from the left, the network advises “strong right”.
- Tested on: $N_{1,1}$.
- Input constraints: $250 \leq \rho \leq 400$, $0.2 \leq \theta \leq 0.4$, $-3.141592 \leq \psi \leq -3.141592 + 0.005$, $100 \leq v_{own} \leq 400$, $0 \leq v_{int} \leq 400$.
- Desired output property: the score for “strong right” is the minimal score.
Property $\phi_6$.

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

Property $\phi_7$.

- 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 $\phi_8$.

- 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 $\phi_9$.

- 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 $\phi_{10}$.

- 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.