Neural Network Verification with Proof Production

Omri Isac[], Clark Barrett[†], Min Zhang[‡] and Guy Katz[]
[]The Hebrew University of Jerusalem, Jerusalem, Israel [†]Stanford University, Stanford, California, USA
[‡]East China Normal University, Shanghai, China
omri.isac@mail.huji.ac.il, barrett@cs.stanford.edu, zhangmin@sei.ecnu.edu.cn, guykatz@cs.huji.ac.il.

Abstract—Deep neural networks (DNNs) are increasingly being employed in safety-critical systems, and there is an urgent need to guarantee their correctness. Consequently, the verification community has devised multiple techniques and tools for verifying DNNs. When DNN verifiers discover an input that triggers an error, that is easy to confirm; but when they report that no error exists, there is no way to ensure that the verification tool itself is not flawed. As multiple errors have already been observed in DNN verification tools, this calls the applicability of DNN verification into question. In this work, we present a novel mechanism for enhancing Simplex-based DNN verifiers with proof production capabilities: the generation of an easy-to-check witness of unsatisfiability, which attests to the absence of errors. Our proof production is based on an efficient adaptation of the well-known Farkas' lemma, combined with mechanisms for handling piecewise-linear functions and numerical precision errors. As a proof of concept, we implemented our technique on top of the Marabou DNN verifier. Our evaluation on a safety-critical system for airborne collision avoidance shows that proof production succeeds in almost all cases and requires only minimal overhead.

I. Introduction

Machine learning techniques, and specifically deep neural networks (DNNs), have been achieving groundbreaking results in solving computationally difficult problems. Nowadays, DNNs are state-of-the-art tools for performing many safety-critical tasks in the domains of healthcare [29], aviation [45] and autonomous driving [19]. DNN training is performed by adjusting the parameters of a DNN to mimic a highly complex function over a large set of input-output examples (the training set) in an automated way that is mostly opaque to humans.

The Achilles heel of DNNs typically lies in generalizing their predictions from the finite training set to an infinite input domain. First, DNNs tend to produce unexpected results on inputs that are considerably different from those in the training set; and second, the input to the DNN might be perturbed by sensorial imperfections, or even by a malicious adversary, again resulting in unexpected and erroneous results. These weaknesses have already been observed in many modern DNNs [37], [64], and have even been demonstrated in the real world [30] — thus hindering the adoption of DNNs in safety-critical settings.

In order to bridge this gap, in recent years, the formal methods community has started devising techniques for DNN verification (e.g., [2], [11], [13], [31], [32], [40], [41], [53], [58], [61], [62], [66], [68], [73], among many others). Typically, DNN verification tools seek to prove that outputs from a given set of inputs are contained within a safe subspace of the output space, using various methods such as SMT solving [1], [16], [23], abstract interpretation [32], MILP solving [65], and combinations thereof. Notably, many modern approaches [50], [53], [55], [65] involve a search procedure, in which the verification problem is regarded as a set of constraints. Then, various input assignments to the DNN are considered in order to discover a counter-example that satisfies these constraints, or to prove that no such counter-example exists.

Verification tools are known to be as prone to errors as any other program [44], [72]. Moreover, the search procedures applied as part of DNN verification typically involve the repeated manipulation of a large number of floating-point equations; this can lead to rounding errors and numerical stability issues, which in turn could potentially compromise the verifier's soundness [12], [44]. When the verifier discovers a counter-example, this issue is perhaps less crucial, as the counter-example can be checked by evaluating the DNN; but when the verifier determines that no counter-example exists, this conclusion is typically not accompanied by a witness of its correctness.

In this work, we present a novel proof-production mechanism for a broad family of search-based DNN verification algorithms. Whenever the search procedure returns UNSAT (indicating that no counter-example exists), our mechanism produces a proof certificate that can be readily checked using simple, external checkers. The proof certificate is produced using a constructive version of Farkas' lemma, which guarantees the existence of a witness to the unsatisfiability of a set of linear equations — combined with additional constructs to support the non-linear components of a DNN, i.e., its piecewise-linear activation functions. We show how to instrument the verification algorithm in order to keep track of its search steps, and use that information to construct the proof with only a small overhead.

For evaluation purposes, we implemented our proof-production technique on top of the Marabou DNN verifier [50]. We then evaluated our technique on the ACAS Xu set of benchmarks for airborne collision avoidance [46], [48]. Our approach was able to produce proof certificates for the safety of various ACAS Xu properties with reasonable overhead (5.7% on average). Checking the proof certificates produced by our approach was usually considerably faster than dispatching the original verification query.

The main contribution of our paper is in proposing a proof-production mechanism for search-based DNN verifiers, which can substantially increase their reliability when de-

https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_9

termining unsatisfiability. However, it also lays a foundation for a conflict-driven clause learning (CDCL) [74] verification scheme for DNNs, which might significantly improve the performance of search-based procedures (see discussion in Sec. IX).

The rest of this paper is organized as follows. In Sec. II we provide relevant background on DNNs, formal verification, the Simplex algorithm, and on using Simplex for search-based DNN verification. In Sec. III, IV and V, we describe the proof-production mechanism for Simplex and its extension to DNN verification. Next, in Sec. VI, we briefly discuss complexity-theoretical aspects of the proof production. Sec. VII details our implementation of the technique and its evaluation. We then discuss related work in Sec. VIII and conclude with Sec. IX.

II. Background

Deep Neural Networks. Deep neural networks (DNNs) [36] are directed graphs, whose nodes (neurons) are organized into layers. Nodes in the first layer, called the input layer, are assigned values based on the input to the DNN; and then the values of nodes in each of the subsequent layers are computed as functions of the values assigned to neurons in the preceding layer. More specifically, each node value is computed by first applying an affine transformation to the values from the preceding layer and then applying a non-linear activation function to the result. The final (output) layer, which corresponds to the output of the network, is computed without applying an activation function.

One of the most common activation functions is the rectified linear unit (ReLU), which is defined as:

$$f(b) = \mathrm{ReLU}(b) = \begin{cases} b & b > 0 \\ 0 & \text{otherwise.} \end{cases}$$

When $b > 0$, we say that the ReLU is in the active phase; otherwise, we say it is in the inactive phase. For simplicity, we restrict our attention here to ReLUs, although our approach could be applied to other piecewise-linear functions (such as max pooling, absolute value, sign, etc.). Non piecewise-linear functions, such as as sigmoid or tanh, are left for future work.

Formally, a DNN $N : \mathbb{R}^m \to \mathbb{R}^k$, is a sequence of n layers $L_0, \ldots, L_{n-1}$ where each layer L_i consists of $s_i \in \mathbb{N}$ nodes, denoted $v_i^1, \ldots, v_i^{s_i}$. The assignment for the j^{th} node in the $1 \leq i < n-1$ layer is computed as

$$v_i^j = \mathrm{ReLU}\left(\sum_{l=1}^{s_{i-1}} w_{i,j,l} \cdot v_{i-1}^l + p_i^j\right)$$

and neurons in the output layer are computed as:

$$v_{n-1}^j = \sum_{l=1}^{s_{n-2}} w_{n-1,j,l} \cdot v_{n-2}^l + p_{n-1}^j$$

where $w_{i,j,l}$ and p_i^j are (respectively) the predetermined weights and biases of N. We set $s_0 = m$ and treat $v_0^1, \ldots, v_0^m$ as the input of N.

A simple DNN with four layers appears in Fig. 1. For simplicity, the p_i^j parameters are all set to zero and are ignored.

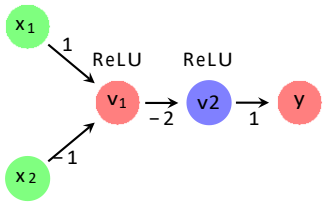

Fig. 1: A toy DNN.

For input $\langle 1, 2 \rangle$, the node in the second layer evaluates to $\mathrm{ReLU}(1 \cdot 1 + 2 \cdot (-1)) = \mathrm{ReLU}(-1) = 0$; the node in the third layer evaluates to $\mathrm{ReLU}(0 \cdot (-2)) = 0$; and the node in the fourth (output) layer evaluates to $0 \cdot 1 = 0$.

DNN Verification and Proofs. Given a DNN $N : \mathbb{R}^m \to \mathbb{R}^k$ and a property $P : \mathbb{R}^{m+k} \to \{T, F\}$, the DNN verification problem is to decide whether there exist $x \in \mathbb{R}^m$ and $y \in \mathbb{R}^k$ such that $(N(x) = y) \wedge P(x, y)$ holds. If such x and y exist, we say that the verification query $\langle N, P \rangle$ is satisfiable (SAT); and otherwise, we say that it is unsatisfiable (UNSAT). For example, given the toy DNN from Fig. 1, we can define a property $P : P(x, y) \Leftrightarrow (x \in [2, 3] \times [-1, 1]) \wedge (y \in [0.25, 0.5])$. Here, P expresses the existence of an input $x \in [2, 3] \times [-1, 1]$ that produces an output $y \in [0.25, 0.5]$. Later on, we will prove that no such x exists, i.e., the verification query $\langle N, P \rangle$ is UNSAT.

Typically, P represents the negation of a desired property, and so an input x which satisfies the query is a counter-example — whereas the query's unsatisfiability indicates that the property holds. In this work, we follow mainstream DNN verification research [53], [68] and focus on properties P that are a conjunction of linear lower- and upper-bound constraints on the neurons of x and y. It has been shown that even for such simple properties, and for DNNs that use only the ReLU activation function, the verification problem is NP-complete [48].

A proof is a mathematical object that certifies a mathematical statement. In case a DNN verification query is SAT, the input x for which P holds constitutes a proof of the query's satisfiability. Our goal here is to generate proofs also for the UNSAT case, which, to the best of our knowledge, is a feature that no DNN verifier currently supports [12].

Verifying DNNs via Linear Programming. Linear Programming (LP) [22] is the problem of optimizing a linear function over a given convex polytope. An LP instance over variables $V = [x_1, \ldots, x_n]^T \in \mathbb{R}^n$ contains an objective function $c \cdot V$ to be maximized, subject to the constraints $A \cdot V = b$ for some $A \in M_{m \times n}(\mathbb{R})$, $b \in \mathbb{R}^m$, and $l \leq V \leq u$ for some $l, u \in (\mathbb{R} \cup \{\pm\infty\})^n$. Throughout the paper, we use $l(x_i)$ and $u(x_i)$, to refer to the lower and upper bounds (respectively) of x_i. LP solving can also be used to check the satisfiability of constraints of the form $(A \cdot V = b) \wedge (l \leq V \leq u)$.

The Simplex algorithm [22] is a widely used technique for solving LP instances. It begins by creating a tableau, which is equivalent to the original set of equations $AV = b$.

Next, Simplex selects a certain subset of the variables, $B \subseteq \{x_1, \ldots, x_n\}$, to act as the basic variables; and the tableau is considered as representing each basic variable $x_i \in B$ as a linear combination of non-basic variables, $x_i = \sum_{j \notin B} c_j \cdot x_j$. We use $A_{i,j}$ to denote the coefficient of a variable x_j in the tableau row that corresponds to basic variable x_i. Apart from the tableau, Simplex also maintains a variable assignment that satisfies the equations of A, but which may temporarily violate the bound constraints $l \leq V \leq u$. The assignment for a variable x_i is denoted $\alpha(x_i)$.

After initialization, Simplex begins searching for an assignment that simultaneously satisfies both the tableau and bound constraints. This is done by manipulating the set B, each time swapping a basic and a non-basic variable. This alters the equations of A by adding multiples of equations to other equations, and allows the algorithm to explore new assignments. The algorithm can terminate with a SAT answer when a satisfying assignment is discovered or an UNSAT answer when: (i) a variable has contradicting bounds, i.e., $l(x_i) > u(x_i)$; or (ii) one of the tableau equations $x_i = \sum_{j \notin B} c_j \cdot x_j$ implies that x_i can never satisfy its bounds. The Simplex algorithm is sound, and is also complete if certain heuristics are used for selecting the manipulations of B [22]. A detailed calculus for the version of Simplex that we use appears in the extended version of this paper [42].

LP solving is particularly useful in the context of DNN verification, and is used by almost all modern tools (either natively [48], or by invoking external solvers such as GLPK [54] or Gurobi [39]). More specifically, a DNN verification query can be regarded as an LP instance with bounded variables that represents the property P and the affine transformations within N, combined with a set of piecewise-linear constraints that represent the activation functions. We demonstrate this with an example, and then explain how this formulation can be solved.

Recall the toy DNN from Fig. 1, and property P that is used for checking whether there exists an input x in the range $[2, 3] \times [-1, 1]$ for which N produces an output y in the range $[0.25, 0.5]$. We use b_1, f_1 to denote the input and output to node v_1; b_2, f_2 for the input and output of v_2; x_1 and x_2 to denote the network's inputs, and y to denote the network's output. The linear constraints of the network yield the linear equations $b_1 = x_1 - x_2$, $b_2 = -2f_1$, and $y = f_2$ (which we name e^1, e^2, and e^3, respectively). The restrictions on the network's input and output are translated to lower and upper bounds: $2 \leq x_1 \leq 3, -1 \leq x_2 \leq 1, 0.25 \leq y \leq 0.5$. The third equation implies that $0.25 \leq f_2 \leq 0.5$, which in turn implies that $b_2 \leq 0.5$. Assume we also restrict: $-0.5 \leq b_2, -0.5 \leq b_1 \leq 0.5, 0 \leq f_1 \leq 0.5$, . Together, these constraints give rise to the linear program that appears in Fig. 2. The remaining ReLU constraints, i.e. $f_i = \text{ReLU}(b_i)$ for $i \in \{1, 2\}$, exist alongside the LP instance. Together, query ϕ is equivalent to the DNN verification problem that we are trying to solve.

Using this formulation, the verification problem can be solved using Simplex, enhanced with a case-splitting approach for handling the ReLU constraints [17], [48]. Intuitively, we first invoke the LP solver on the LP portion of the query; and if it returns UNSAT, the whole query is UNSAT. Otherwise, if it finds a satisfying assignment, we check whether this assignment also satisfies the ReLU constraints. If it does, then the whole query is SAT. Otherwise, case splitting is applied in order to split the query into two different sub-queries, according to the two phases of the ReLU function.[1] Specifically, in one of the sub-queries, the LP query is adjusted to enforce the ReLU to be in the active phase: the equation $f = b$ is added, along with the bound $b \geq 0$. In the other sub-query, the inactive phase is enforced: $b \leq 0, 0 \leq f \leq 0$. This effectively reduces the ReLU constraint into linear constraints in each sub-query. This process is then repeated for each of the two sub-queries.

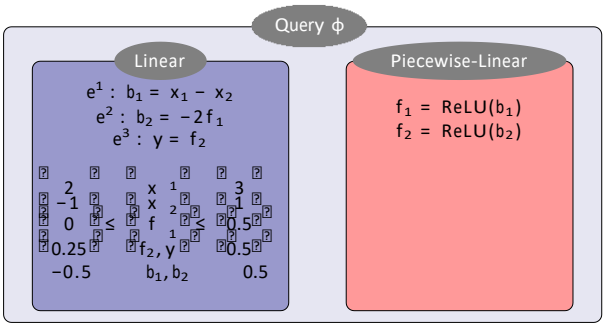

Fig. 2: An example of a DNN verification query ϕ, comprised of an LP instance and piecewise-linear constraints.

Case-splitting turns the verification procedure into a search tree [48], with nodes corresponding to the splits that were applied. The tree is constructed iteratively, with Simplex invoked on every node to try and derive UNSAT or find a true satisfying assignment. If Simplex is able to deduce that all leaves in the search tree are UNSAT, then so is the original query. Otherwise, it will eventually find a satisfying assignment that also satisfies the original query. This process is sound, and will always terminate if appropriate splitting strategies are used [22], [48]. Unfortunately, the size of the search tree can be exponential in the number of ReLU constraints; and so in order to keep the search tree small, case splitting is applied as little as possible, according to various heuristics that change from tool to tool [55], [62], [68]. In order to reduce the number of splits even further, verification algorithms apply clever deduction techniques for discovering tighter variable bounds, which may in turn rule out some of the splits a-priori. We also discuss this kind of deduction, which we refer to as dynamic bound tightening, in the following sections.

III. Proof Production Overview

A Simplex-based verification process of a DNN is tree-shaped, and so we propose to generate a proof tree to match

[1] The approach is easily generalizable to any piecewise-linear constraint, by splitting the query according to the different linear pieces of the activation function.

it. Within the proof tree, internal nodes will correspond to case splits, whereas each leaf node will contain a proof of unsatisfiability based on all splits performed on the path between itself and the root. Thus, a proof tree constitutes a valid proof of unsatisfiability if each of its leaves contains a proof that demonstrates that all splits so far lead to a contradiction. The proof tree might also include proofs for lemmas, which are valid statements for the node in which they reside and its descendants (lemmas are needed for supporting bound tightening, as we discuss later).

As a simple, intuitive example, we depict in Fig. 3 a proof of unsatisfiability for the query ϕ from Fig. 2. The root of the proof tree represents the initial verification query, which is comprised of LP constraints and ReLU constraints. The fact that this node is not a leaf indicates that the Simplex-based verifier was unable to conclude UNSAT in this state, and needed to perform a case split on the ReLU node v_1. The left child of the root corresponds to the case where ReLU v_1 is inactive: the LP is augmented with additional constraints that represent the case split, i.e., $f_1 = 0$ and $b_1 \leq 0$. This new fact may now be used by the Simplex procedure, which is indeed able to obtain an UNSAT result. The node then contains a proof of this unsatisfiability: $\begin{bmatrix} -1 & 0 & 0 \end{bmatrix}$. This vector instructs the checker how to construct a linear combination of the current tableau's rows, in a way that leads to a bound contradiction, as we later explain in Sec. V.

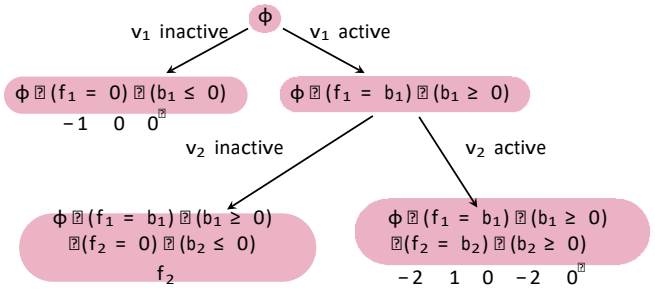

Fig. 3: A proof tree example.

In the right child of the root, which represents v_1's active phase, the constraints $f_1 = b_1$ and $b_1 \geq 0$ are added by the split. This node is not a leaf, because the verifier performed a second case split, this time on v_2. The left child represents v_2's inactive phase, and has the corresponding constraints $f_2 = 0$ and $b_2 \leq 0$. This child is a leaf, and is marked with f_2, indicating that f_2 is a variable whose bounds led to a contradiction. Specifically, $f_2 \geq 0.25$ from ϕ and $f_2 = 0$ from the case split are contradictory.

The last node (the rightmost leaf) represents v_2's active phase, and has the constraints $f_2 = b_2$ and $b_2 \geq 0$. Here, the node indicates that a contradiction can be reached from the current tableau, using the vector $\begin{bmatrix} -2 & 1 & 0 & -2 & 0 \end{bmatrix}$. In Sec. IV, we explain how this process works.

Because each leaf of the proof tree contains a proof of unsatisfiability, the tree itself proves that the original query is UNSAT. Note that many other proof trees may exist for the same query. In the following sections, we explain how to instrument a Simplex-based verifier in order to extract such proof trees from the solver execution.

IV. Simplex with Proofs

A. Producing proofs for LP

We now describe our approach for creating proof trees, beginning with leaf nodes. We start with the following lemma:

Lemma 1. If Simplex returns UNSAT, then there exists a variable with contradicting bounds; that is, there exists a variable $x_i \in V$ with lower and upper bounds $l(x_i)$ and $u(x_i)$, for which Simplex has discovered that $l(x_i) > u(x_i)$.

This lemma justifies our choice of using contradicting bounds as proofs of unsatisfiability in the leaves of the proof tree. The lemma follows directly from the derivation rules of Simplex. Specifically, there are only two ways to reach UNSAT: when the input problem already contains inconsistent bounds $l(x_i) > u(x_i)$, or when Simplex finds a tableau row $x_i = \sum_{j \notin B} c_j \cdot x_j$ that gives rise to such inconsistent bounds. The complete proof appears in the extended version of this paper [42].

We demonstrate this with an example, based on the query ϕ from Fig. 2. Suppose that, as part of its Simplex-based solution process, a DNN verifier performs two case splits, fixing the two ReLUs to their active states: $f_1 = b_1 \wedge b_1 \geq 0$ and $f_2 = b_2 \wedge b_2 \geq 0$. This gives rise to the following (slightly simplified) system of equations:

$$b_1 = x_1 - x_2 \quad b_2 = -2f_1 \quad y = f_2 \quad f_1 = b_1 \quad f_2 = b_2$$

Which corresponds to the tableau and variables

$$A = \begin{bmatrix} 1 & -1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & -2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & -1 & 0 \end{bmatrix} \quad V = \begin{bmatrix} x_1 \\ x_2 \\ b_1 \\ b_2 \\ f_1 \\ f_2 \\ y \end{bmatrix}$$

such that $AV = \bar{0}$, with the corresponding bound vectors:

$$l = \begin{bmatrix} 2 & -1 & 0 & 0 & 0 & 0.25 & 0.25 \end{bmatrix}$$

$$u = \begin{bmatrix} 3 & 1 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 \end{bmatrix}$$

Then, the Simplex solver iteratively alters the set of basic variables, which corresponds to multiplying various equations by scalars and summing them to obtain new equations. At some point, the equation $b_2 = -2x_1 + 2x_2$ is obtained (by computing $\begin{bmatrix} -2 & 1 & 0 & -2 & 0 \end{bmatrix} \cdot A \cdot V$), with a current assignment of $\alpha(V)^T = \begin{bmatrix} 2 & 1 & 1 & -2 & 1 & -2 & -2 \end{bmatrix}$.

At this point, the Simplex solver halts with an UNSAT notice. The reason is that b_2 is currently assigned the value -2, which is below its lower bound of 0, and so its value needs to be increased. However, the equation, combined with the fact that x_1 is pressed against its lower bound, while x_2 is

pressed against its upper bound, indicates that there is no slack remaining in order to increase the value of b_2 (this corresponds to the Failure$_1$ rule in the Simplex calculus described in the extended version of this paper [42]). The key point is that the same equation could be used in deducing a tighter bound for b_2:

$$b_2 \leq -2l(x_1) + 2u(x_2) = -2 \cdot 2 + 2 \cdot 1 = -2$$

and a contradiction could then be obtained based on the contradictory facts $0 = l(b_2) \leq b_2 \leq -2$. In other words, and as we formally prove in the extended version of this paper [42], any UNSAT answer returned by Simplex can be regarded as a case of conflicting lower and upper bounds.

Given Lemma 1, our goal is to instrument the Simplex procedure so that whenever it returns UNSAT, we are able to produce a proof which indicates that $l(x_i) > u(x_i)$ for some variable x_i. To this end, we introduce the following adaptation of Farkas' Lemma [67] to the Simplex setting, which states that a linear-sized proof of this fact exists.

Lemma 2. Given the constraints $A \cdot V = 0$ and $l \leq V \leq u$, where $A \in M_{m \times n}(R)$ and $l, V, u \in R^n$, exactly one of these two options holds:

1) The SAT case: $\exists V \in R^n$ such that $A \cdot V = 0$ and $l \leq V \leq u$.
2) The UNSAT case: $\exists w \in R^m$ such that for all $l \leq V \leq u$, $w^T \cdot A \cdot V < 0$, whereas $0 \cdot w = 0$. Thus, w is a proof of the constraints' unsatisfiability.

Moreover, these vectors can be constructed during the run of the Simplex algorithm.

This Lemma is actually a corollary of Theorem 3, which we introduce later. For a complete proof, see the extended version of this paper [42].

In our previous, UNSAT example, one possible vector is $w = \begin{pmatrix} -2 & 1 & 0 & -2 & 0 \end{pmatrix}^T$. Indeed, $w \cdot A \cdot V = 0$ gives us the equation $-2x_1 + 2x_2 - b_2 = 0$. Given the lower and upper bounds for the participating variables, the largest value that the left-hand side of the equation can obtain is:

$$-2l(x_1) + 2u(x_2) - l(b_2) = -2 \cdot 2 + 2 \cdot 1 - 0 = -2 < 0$$

Therefore, no variable assignment within the stated bounds can satisfy the equation, indicating that the constraints are UNSAT.

Given Lemma 2, all that remains is to instrument the Simplex solver in order to produce the proof vector w on the fly, whenever a contradiction is detected. In case a trivial contradiction $l(x_i) > u(x_i)$ is given as part of the input query for some variable x_i, we simply return "x_i" as the proof (we later discuss also how to handle this case in the presence of dynamic bound tightenings). Otherwise, a non-trivial contradiction is detected as a result of an equation $e \equiv x_i = \sum_{j \notin B} c_j \cdot x_j$, which contradicts one of the input bounds of x_i. In this case, no assignment can satisfy the equivalent equation $\sum_{j \notin B} c_j \cdot x_j - x_i = 0$. Since the Simplex algorithm applies only linear operations to the input tableau, e is given by a linear combination of the original tableau rows. Let $coef(e)$ denote the Farkas vector of the equation e, i.e., the column vector such that $coef(e)^T \cdot A = e$, and which proves unsatisfiability in this case. Our framework simply keeps track, for each row of the tableau, of its coefficient vector; and if that row leads to a contradiction, the vector is returned.

B. Supporting dynamic bound tightening

So far, we have only considered Simplex executions that do not perform any bound tightening steps; i.e., derive UNSAT by finding a contradiction to the original bounds. However, in practice, modern DNN solvers perform a great deal of dynamic bound tightening, and so this needs to be reflected in the proof.

We use the term ground bounds to refer to variable bounds that are part of the LP being solved, whether they were introduced by the original input, or by successive case splits, as we will explain in Sec. V. This is opposed to dynamic bounds, which are bounds introduced on the fly, via bound tightening. The ground bounds, denoted $l, u \in R^n$, are used in explaining dynamic bounds, denoted $l', u' \in R^n$, via Farkas vectors.

For simplicity, we consider here a simple and popular version of bound tightening, called interval propagation [25], [48]. Given an equation $x_i = \sum_{j \notin B} c_j \cdot x_j$ and current bounds $l'(x)$ and $u'(x)$ for each of the variables (whether these are the ground bounds or dynamically tightened bounds themselves), a new upper bound for x_i can be derived:

$$u'(x_i) := \sum_{x_j \notin B,\ c_j > 0} c_j \cdot u'(x_j) + \sum_{x_j \notin B,\ c_j < 0} c_j \cdot l'(x_j) \quad (1)$$

(provided that the new bound is tighter, i.e., smaller, than the current upper bound for x_i). A symmetrical version exists for discovering lower bounds.

A naive approach for handling bound tightening is to store, each time a new bound is discovered, a separate proof that justifies it; for example, a Farkas vector for deriving the equation that was used in the bound tightening. However, a Simplex execution can include many thousands of bound tightenings — and so doing this would strain resources. Even worse, many of the intermediate bound tightenings might not even participate in deriving the final contradiction, and so storing them would be a waste.

In order to circumvent this issue, we propose a scheme in which we store, for each variable in the query, a single column vector that justifies its current lower bound, and another for its current upper bound. Whenever a tighter bound is dynamically discovered, the corresponding vector is updated; and even if other, previously discovered dynamic bounds were used in the derivation, the vector that we store indicates how the same bound can be derived using the ground bounds. Thus, the proof of the tightened bounds remains compact, regardless of the number of derived bounds; specifically, it requires only $O(n \cdot m)$ space overall. Formally, we have the following result:

Theorem 3. Let $A \cdot V = 0$ such that $l \leq V \leq u$ be an LP instance, where $A \in M_{m \times n}(R)$ and $l, V, u \in R^n$.

Let $u', l' \in \mathbb{R}^n$ represent dynamically tightened bounds of V. Then $\forall i \in [n]\ \exists f_u(x_i), f_l(x_i) \in \mathbb{R}^m$ such that $f_u(x_i)^T \cdot A$ and $f_l(x_i)^T \cdot A$ can be used to efficiently compute $u'(x_i), l'(x_i)$ from l and u. Moreover, vectors $f_u(x_i)$ and $f_l(x_i)$ can be constructed during the run of the Simplex algorithm.

When a Simplex procedure with bound tightening reaches an UNSAT answer, it has discovered a variable x_i with $l'(x_i) > u'(x_i)$. The theorem guarantees that in this case we have two column vectors, $f_u(x_i)$ and $f_l(x_i)$, which explain how $u'(x_i)$ and $l'(x_i)$ were discovered. We refer to these vectors as the Farkas vectors of the upper and lower bounds of x_i, respectively. Because $u'(x_i) - l'(x_i)$ is negative, the column vector $w = f_u(x_i) - f_l(x_i)$ creates a tableau row which is always negative, making $w \in \mathbb{R}^m$ a proof of unsatisfiability. The formal, constructive proof of the theorem appears in the extended version of this paper [42].

In order to maintain $f_u(x_i)$ and $f_l(x_i)$ during the execution of Simplex, whenever a tigher upper bound is tightened using Eq. 1, we update the matching Farkas vector:

$$f_u(x_i) := \sum_{j=i,\, c_j > 0} c_j \cdot f_u(x_j) + \sum_{j=i,\, c_j < 0} c_j \cdot f_l(x_j) + \text{coef}(e),$$

where e is the linear equation used for tightening, and $\text{coef}(e)$ is the column vector such that $\text{coef}(e)^T \cdot A = e$. The lower bound case is symmetrical. To demonstrate the procedure, consider again the verification query from Fig. 2. Assume the phases of v_1, v_2 have both been set to active, and that consequently two new equations have been added: $e^4 : f_1 = b_1$, $e^5 : f_2 = b_2$. In this example, we have five linear equations, so we initialize a zero vector of size five for each of the variable bounds. Now, suppose Simplex tightens the lower bound of b_1 using the first equation e^1:

$$l'(b_1) := l(x_1) - u(x_2) = 2 - 1 = 1$$

and thus we update

$$\begin{aligned} f_l(b_1) &:= f_l(x) - f_u(y) + \text{coef}(e^1) \\ &= \begin{bmatrix}0 & 0 & 0 & 0 & 0\end{bmatrix}^T + \begin{bmatrix}0 & 0 & 0 & 0 & 0\end{bmatrix}^T \\ &+ \begin{bmatrix}1 & 0 & 0 & 0 & 0\end{bmatrix}^T \\ &= \begin{bmatrix}1 & 0 & 0 & 0 & 0\end{bmatrix}^T \end{aligned}$$

since all f_l and f_u vectors have been initialized to $\bar{0}$ and $\text{coef}(e) = \begin{bmatrix}1 & 0 & 0 & 0 & 0\end{bmatrix}^T$ — which indicates that e^1 is simply the first row of the tableau.

We can now tighten bounds again, using the fourth row $f_1 = b_1$, and get $l'(f_1) := l'(b_1) = 1$. We update $f_l(f_1)$:

$$\begin{aligned} f_l(f_1) &:= f_l(b_1) + \text{coef}(e^4) \\ &= \begin{bmatrix}1 & 0 & 0 & 0 & 0\end{bmatrix}^T + \begin{bmatrix}0 & 0 & 0 & 1 & 0\end{bmatrix}^T \\ &= \begin{bmatrix}1 & 0 & 0 & 1 & 0\end{bmatrix}^T \end{aligned}$$

To see that the Farkas vector can indeed explain the dynamically tightened bound, observe that the combination $\begin{bmatrix}1 & 0 & 0 & 1 & 0\end{bmatrix}^T$ of tableau rows gives the equation $f_1 = x_1 - x_2$. We can then tighten the lower bound of f_1, using the ground bounds: $l'(f_1) := l(x_1) - u(x_2) = 2 - 1 = 1$. This bound matches the one that we had discovered dynamically, though we derived it using ground bounds only.

V. DNN Verification with Proofs

A. Producing a proof-tree

We now discuss how to leverage the results of Sec. IV in order to produce the entire proof tree for an UNSAT DNN verification query. Recall that the main challenge lies in accounting for the piecewise-linear constraints, which affect the solving process by introducing case-splits.

Each case split performed by the solver introduces a branching in the proof tree — with a new child node for each of the linear phases of the constraint being split on — and introduces new equations and bounds. In the case of ReLU, one child node represents the active branch, through the equation $f = b$ and bound $b \geq 0$; and another represents the inactive branch, with $b \leq 0$ and $0 \leq f \leq 0$. These new bounds become the ground bounds for this node: their Farkas vectors are reset to zero, and all subsequent Farkas vectors refer to these new bounds (as opposed to the ground bounds of the parent node). A new node inherits any previously-discovered dynamic bounds, as well as the Farkas vectors that explain them, from its parent; these vectors remain valid, as ground bounds only become tighter as a result of splitting (see the extended version of this paper [42]).

For example, let us return to the query from Fig. 2 and the proof tree from Fig. 3. Initially, the solver decides to split on v_1. This adds two new children to the proof tree. In the first child, representing the inactive case, we update the ground bounds $u(b_1) := 0$, $u(f_1) := 0$, and reset the corresponding Farkas vectors $f_u(b_1)$ and $f_u(f_1)$ to $\bar{0}$. Now, Simplex can tighten the lower bound of b_1 using the first equation e^1:

$$l'(b_1) := l(x_1) - u(x_2) = 2 - 1 = 1$$

resulting in the the updated $f_l(b_1) = \begin{bmatrix}1 & 0 & 0\end{bmatrix}^T$, as shown in Sec. IV, where we use vectors of size three since in this search state we have three equations. Observe this bound contradicts the upper ground bound of b_1, represented by the zero vector. We can then use the vector

$$f_u(b_1) - f_l(b_1) = \bar{0} - \begin{bmatrix}1 & 0 & 0\end{bmatrix}^T = \begin{bmatrix}-1 & 0 & 0\end{bmatrix}^T$$

as a proof for contradiction. Indeed, the matrix A', which is obtained using the first three rows and columns of A as defined in Sec. III, corresponds to the tableau before adding any new equations. Observe that $\begin{bmatrix}-1 & 0 & 0\end{bmatrix}^T \cdot A' \cdot V = 0$ gives the equation $-x_1 + x_2 + b_1 = 0$. Given the current ground bounds, the largest value of the left-hand side is:

$$-l(x_1) + u(x_2) + u(b_1) = -2 + 1 + 0 = -1$$

which is negative, meaning that no variable assignment within these bounds can satisfy the equation. This indicates that the proof node representing v_1's inactive phase is UNSAT.

In the second child, representing v_1's active case, we update the ground bound $l(b_1) := 0$ and the Farkas vector $f_l(b_1) := \bar{0}$.

We also add the equation $e^4 : f_1 = b_1$. Next, the solver performs another split on v_2, adding two new children to the tree. In the first one (representing the inactive case) we update the ground bounds $u(b_2) := 0$, $u(f_2) := 0$, and reset the corresponding Farkas vectors $f_u(b_2)$ and $f_u(f_2)$ to $\vec{0}$. In this node, we have a contradiction already in the ground bounds, since $u(f_2) := 0$ but $l(f_2) := 0.25$. The contradiction in this case is comprised of a symbol for f_2.

We are left with proving UNSAT for the last child, representing the case where both ReLU nodes v_1, v_2 are active. For this node of the proof tree, we update the ground bound $l(b_2) := 0$ and Farkas vector $f_l(b_2) := \vec{0}$, and add the equation $e^5 : f_2 = b_2$. Recall that previously, we learned the tighter bound $l'(f_1) = 1$. With the same procedure as described in Sec. IV, we can update $f_l(f_1) = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 \end{bmatrix}$. Now, we can use $e^2 : b_2 = -2f_1$ to tighten $u'(b_2) := -2l'(f_1) = -2$, and consequently update the Farkas vector:

$$\begin{aligned} f_u(b_2) &= -2 \cdot f_l(f_1) + \text{coef}(e^2) \\ &= -2 \cdot \begin{bmatrix} 1 & 0 & 0 & 1 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \end{bmatrix} \\ &= \begin{bmatrix} -2 & 1 & 0 & -2 & 0 \end{bmatrix} \end{aligned}$$

The bound $u'(b_2) = -2$, explained by $\begin{bmatrix} -2 & 1 & 0 & -2 & 0 \end{bmatrix}$ contradicts the ground bound $l(b_2) = 0$ explained by the zero vector. Therefore, we get the vector

$$\begin{bmatrix} -2 & 1 & 0 & -2 & 0 \end{bmatrix} - 0 = \begin{bmatrix} -2 & 1 & 0 & -2 & 0 \end{bmatrix}$$

as the proof of contradiction for this node.

B. Bound tightenings from piecewise-linear constraints

Modern solvers often use sophisticated methods [25], [50], [62] to tighten variable bounds using the piecewise-linear constraints. For example, if $f = \text{ReLU}(b)$, then in particular $b \leq f$, and so $u(b) \leq u(f)$. Thus, if initially $u(b) = u(f) = 7$ and it is later discovered that $u'(f) = 5$, we can deduce that also $u'(b) = 5$. We show here how such tightening can be supported by our proof framework, focusing on some ReLU tightening rules as specified in the extended version of this paper [42]. Supporting additional rules should be similar.

We distinguish between two kinds of ReLU bound tightenings. The first are tightenings that can be explained via a Farkas vector; these are handled the same way as bounds discovered using interval propagation. The second, more complex tightenings are those that cannot be explained using an equation (and thus a Farkas vector). Instead, we treat these bound tightenings as lemmas, which are added to the proof node along with their respective proofs; and the bounds that they tighten are introduced as ground bounds, to be used in constructing future Farkas vectors. The proof for a lemma consists of Farkas vectors explaining any current bounds that were used in deducing it; as well as an indication of the tightening rule that was used. The list of allowed tightening rules must be agreed upon beforehand and provided to the checker; in the extended version of this paper [42], we present the tightening rules for ReLUs that we currently support.

For example, if $f = \text{ReLU}(b)$ and $u'(f) = 5$ causes a bound tightening $u'(b) = 5$, then this new bound $u'(b) = 5$ is stored as a lemma. Its proof consists of the Farkas vector $f_u(f)$ which explains why $u'(f) = 5$, and an indication of the deduction rule that was used (in this case, $u'(b) \leq u'(f)$).

VI. Proof Checking and Numerical Stability

Checking the validity of a proof tree is straightforward. First, the checker must read the initial query and confirm that it is consistent with the LP and piecewise-linear constraints stored at the root of the tree. Next, the checker begins a depth-first traversal of the proof tree. Whenever it reaches a new inner node, it must confirm that that node's children correspond to the linear phases of a piecewise-linear constraint present in the query. Further, the checker must maintain a list of current equations and lower and upper bounds, and whenever a new node is visited — update these lists (i.e., add equations and tighten bounds as needed), to reflect the LP stored in that node. Additionally, the checker must confirm the validity of lemmas that appear in the node — specifically, to confirm that they adhere to one of the permitted derivation rules. Finally, when a leaf node is visited, the checker must confirm that the Farkas vector stored therein does indeed lead to a contradiction when applied to the current LP — by ensuring that the linear combination of rows created by the Farkas vector leads to a matrix row $\sum c_j \cdot x_j = 0$, such that for any assignment of the variables, the left-hand side will have a negative value.

The process of checking a proof certificate is thus much simpler than verifying a DNN using modern approaches, as it consists primarily of traversing a tree and computing linear combinations of the tableau's columns. Furthermore, the proof checking process does not require using division for its arithmetic computations, thus making the checking program more stable arithmetically [44]. Consequently, we propose to treat the checker as a trusted code-base, as is commonly done [15], [49].

Complexity and Proof Size. Proving that a DNN verification query is SAT (by providing a satisfying assignment) is significantly easier than discovering an UNSAT witness using our technique. Indeed, this is not surprising; recall that the DNN verification problem is NP-complete, and that yes-instances of NP problems have polynomial-size witnesses (i.e., polynomial-size proofs). Discovering a way to similarly produce polynomial proofs for no-instances of DNN verification is equivalent to proving that NP = coNP, which is a major open problem [8] and might, of course, be impossible.

Numerical Stability. Recall that enhancing DNN verifiers with proof production is needed in part because they might produce incorrect UNSAT results due to numerical instability. When this happens, the proof checking will fail when checking a proof leaf, and the user will receive warning. There are, however, cases where the query is UNSAT, but only the proof produced by the verifier is flawed. To recover from these cases

and correct the proof, we propose to use an external SMT solver to re-solve the query stored in the leaf in question.

SMT solvers typically use sound arithmetic (as opposed to DNN verifiers), and so their conclusions are generally more reliable. Further, if a proof-producing SMT solver is used, the proof that it produces could be plugged into the larger proof tree, instead of the incorrect proof previously discovered. Although using SMT solvers to directly verify DNNs has been shown to be highly ineffective [48], [59], in our evaluation we observed that leaves typically represented problems that were significantly simpler than the original query, and could be solved efficiently by the SMT solver.

VII. Implementation and Evaluation

Implementation. For evaluation purposes, we instrumented the Marabou DNN verifier [50], [69] with proof production capabilities. Marabou is a state-of-the-art DNN verifier, which uses a native Simplex solver, and combines it with other modern techniques — such as abstraction and abstract interpretation [26], [27], [57], [62], [68], [71], advanced splitting heuristics [70], DNN optimization [63], and support for varied activation functions [6]. Additionally, Marabou has been applied to a variety of verification-based tasks, such as verifying recurrent networks [43] and DRL-based systems [3], [5], [28], [51], network repair [34], [60], network simplification [33], [52], and ensemble selection [4].

As part of our enhancements to Marabou's Simplex core, we added a mechanism that stores, for each variable, the current Farkas vectors that explain its bounds. These vectors are updated with each Simplex iteration in which the tableau is altered. Additionally, we instrumented some of Marabou's Simplex bound propagation mechanisms — specifically, those that perform interval-based bound tightening on individual rows [25], to record for each tighter bound the Farkas vector that justifies it. Thus, whenever the Simplex core declares UNSAT as a result of conflicting bounds, the proof infrastructure is able to collect all relevant components for creating the certificate for that particular leaf in the proof tree. Due to time restrictions, we were not able to instrument all of Marabou's many bound propagation components; this is ongoing work, and our experiments described below were run with yet-unsupported components turned off. The only exception is Marabou's preprocessing component, which is not supported, but is run before proof production starts.

In order to keep track of Marabou's tree-like search, we instrumented Marabou's SmtCore class, which is in charge of case splitting and backtracking [50]. Whenever a case-split was performed, the corresponding equations and bounds were added to the proof tree as ground truths; and whenever a previous split was popped, our data structures would backtrack as well, returning to the previous ground bounds.

In addition to the instrumentation of Marabou, we also wrote a simple proof checker that receives a query and a proof artifact — and then checks, based on this artifact, that the query is indeed UNSAT. That checker also interfaces with the cvc5 SMT solver [14] for attempting recovery from numerical instability errors.

Evaluation. We used our proof-producing version of Marabou to solve queries on the ACAS-Xu family of benchmarks for airborne collision avoidance [45]. We argue that the safety-critical nature of this system makes it a prime candidate for proof production. Our set of benchmarks was thus comprised of 45 networks and 4 properties to test on each, producing a total of 180 verification queries. Marabou returned an UNSAT result on 113 of these queries, and so we focus on them. In the future, we intend to evaluate our proof-production mechanism on other benchmarks as well.

We set out to evaluate our proof production mechanism along 3 axes: (i) correctness: how often was the checker able to verify the proof artifact, and how often did Marabou (prob-ably due to numerical instability issues) produce incorrect proofs?; (ii) overhead: by how much did Marabou's runtime increase due to the added overhead of proof production?; and (iii) checking time: how long did it take to check the produced proofs? Below we address each of these questions.

Correctness. Over 1.46 million proof-tree leaves were created and checked as part of our experiments. Of these, proof checking failed for only 77 leaves, meaning that the Farkas vector written in the proof-tree leaf did not allow the proof checker to deduce a contradiction. Out of the 113 queries checked, 97 had all their proof-tree leaves checked successfully. As for the rest, typically only a tiny number of leaves would fail per query, but we did identify a single query where a significant number of proofs failed to check (see Fig. 4). We speculate that this query had some intrinsic numerical issues encoded into it (e.g., equations with very small coefficients [20]).

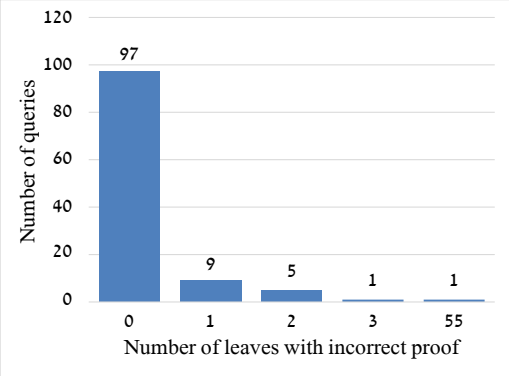

Fig. 4: Number of queries per number of leaves with incorrect proofs.

Next, when we encoded each of the 77 leaves as a query to the cvc5 SMT solver [14], it was able to show that all queries were indeed UNSAT, in under 20 seconds per query. From this we learn that although some of the proof certificates produced by Marabou were incorrect, the ultimate UNSAT result was correct. Further, it is interesting to note how quickly each of the queries could be solved. This gives rise to an

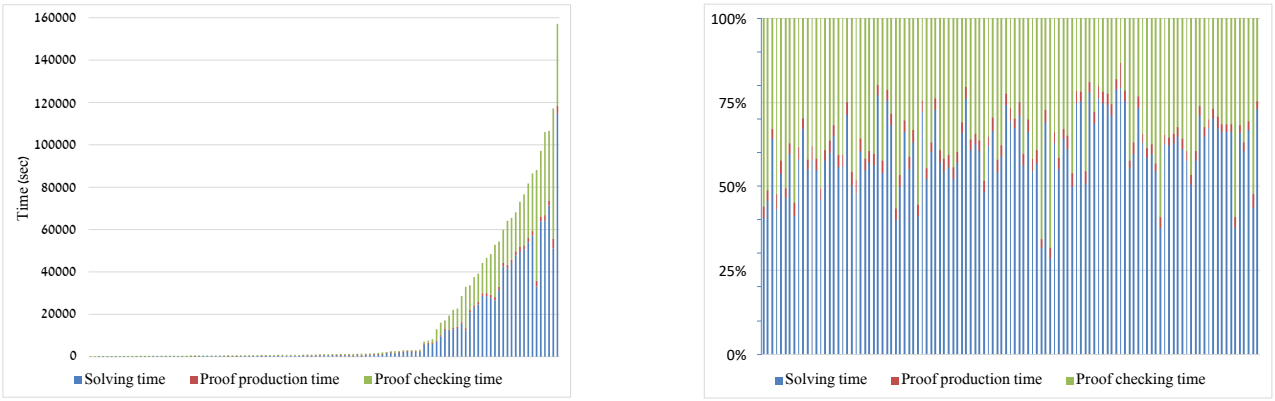

Fig. 5: Proof production and checking time comparison — absolute (left) and relative (right)

interesting verification strategy: use modern DNN verifiers to do the "heavy-lifting", and then use more precise SMT solvers specifically on small components of the query that proved difficult to solve accurately.

Overhead and Checking Time. In Fig. 5, we compare the running time of vanilla Marabou, the overhead incurred by our proof-production extension to Marabou, and the checking time of the resulting proof certificates. We can see that the overhead of proof production time is relatively small for all queries (an average overhead of 5.7%), while the certification time is non-negligible, but shorter than the time it takes to solve the queries by a factor of 66.5% on average.

VIII. Related Work

The importance of proof production in verifiers has been repeatedly recognized, for example by the SAT, SMT, and model-checking communities (e.g., [15], [21], [38]). Although the risks posed by numerical imprecision within DNN verifiers have been raised repeatedly [12], [44], [48], [47], we are unaware of any existing proof-producing DNN verifiers.

Proof production for various Simplex variants has been studied previously [56]. In [24], Dutertre and de Moura study a Simplex variant similar to ours, but without explicit support for dynamic bound tightening. Techniques for producing Farkas vectors have also been studied [10], but again without support for dynamic bound tightening, which is crucial in DNN verification. Other uses of Farkas vectors, specifically in the context of interpolants, have also been explored [9], [18].

Other frameworks for proof production for machine learning have also been proposed [7], [35]; but these frameworks are interactive, unlike the automated mechanism we present here.

IX. Conclusion and Future work

We presented a novel framework for producing proofs of unsatisfiability for Simplex-based DNN verifiers. Our framework constructs a proof tree that contains lemma proofs in internal nodes and unsatisfiability proofs in each leaf. The certificates of unsatisfiability that we provide can increase the reliability of DNN verification, particularly when floating-point arithmetic (which is susceptible to numerical instability) is used.

We plan to continue this work along two orthogonal paths: (i) extend our mechanism to support additional steps performed in modern verifiers, such as preprocessing and additional abstract interpretation steps [53], [62]; and (ii) use our infrastructure to allow learning succinct conflict clauses. During search, the Farkas vectors produced by our approach could be used to generate conflict clauses on-the-fly. Intuitively, conflict clauses guide the verification algorithm to avoid any future search for a satisfying assignment within subspaces of the search space already proven to be UNSAT. Such clauses are a key component in modern SAT and SMT solvers, and are the main component of CDCL algorithms [74] — and could significantly curtail the search space traversed by DNN verifiers and improve their scalability.

Acknowledgments. This work was supported by the Israel Science Foundation (grant number 683/18), the ISF-NSFC joint research program (grant numbers 3420/21 and 62161146001), the Binational Science Foundation (grant numbers 2017662 and 2020250), and the National Science Foundation (grant number 1814369).

References

[1] E. Ábrahám and G. Kremer. SMT Solving for Arithmetic Theories: Theory and Tool Support. In Proc. 19th Int. Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), pages 1–8, 2017.

[2] M. Akintunde, A. Kevorchian, A. Lomuscio, and E. Pirovano. Verification of RNN-Based Neural Agent-Environment Systems. In Proc. 33rd AAAI Conf. on Artificial Intelligence (AAAI), pages 197–210, 2019.

[3] G. Amir, D. Corsi, R. Yerushalmi, L. Marzari, D. Harel, A. Farinelli, and G. Katz. Verifying Learning-Based Robotic Navigation Systems, 2022. Technical Report. https://arxiv.org/abs/2205.13536.

[4] G. Amir, G. Katz, and M. Schapira. Verification-Aided Deep Ensemble Selection. In Proc. 22nd Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD), 2022.

[5] G. Amir, M. Schapira, and G. Katz. Towards Scalable Verification of Deep Reinforcement Learning. In Proc. 21st Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD), pages 193–203, 2021.

[6] G. Amir, H. Wu, C. Barrett, and G. Katz. An SMT-Based Approach for Verifying Binarized Neural Networks. In Proc. 27th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pages 203–222, 2021.

[7] C. Anil, G. Zhang, A. Wu, and R. Grosse. Learning to Give Checkable Answers with Prover-Verifier Games, 2021. Technical Report. https://arxiv.org/abs/2108.12099.

[8] S. Arora and B. Barak. Computational Complexity: A Modern Approach. Cambridge University Press, 2009.

[9] S. Asadi, M. Blicha, A. Hyvarinen, G. Fedyukovich, and N. Sharygina. Farkas-Based Tree Interpolation. In Proc. 27th Int. Static Analysis Symposium (SAS), pages 357–379, 2020.

[10] D. Avis and B. Kaluzny. Solving Inequalities and Proving Farkas's Lemma Made Easy. The American Mathematical Monthly, 111(2):152–157, 2004.

[11] G. Avni, R. Bloem, K. Chatterjee, T. Henzinger, B. Konighofer, and S. Pranger. Run-Time Optimization for Learned Controllers through Quantitative Games. In Proc. 31st Int. Conf. on Computer Aided Verification (CAV), pages 630–649, 2019.

[12] S. Bak, C. Liu, and T. Johnson. The Second International Verification of Neural Networks Competition (VNN-COMP 2021): Summary and Results, 2021. Technical Report. http://arxiv.org/abs/2109.00498.

[13] T. Baluta, S. Shen, S. Shinde, K. Meel, and P. Saxena. Quantitative Verification of Neural Networks And its Security Applications. In Proc. ACM SIGSAC Conf. on Computer and Communications Security (CCS), pages 1249–1264, 2019.

[14] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar. cvc5: A Versatile and Industrial-Strength SMT Solver. In Proc. 28th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pages 415–442, 2022.

[15] C. Barrett, L. de Moura, and P. Fontaine. Proofs in Satisfiability Modulo Theories. All about Proofs, Proofs for All, 55(1):23–44, 2015.

[16] C. Barrett and C. Tinelli. Satisfiability Modulo Theories. In Handbook of Model Checking, pages 305–343. Springer, 2018.

[17] 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.

[18] M. Blicha, A. Hyvärinen, J. Kofroň, and N. Sharygina. Decomposing Farkas Interpolants. In Proc. 25th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pages 3–20, 2019.

[19] 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.

[20] V. Chvátal. Linear Programming. W. H. Freeman and Company, 1983.

[21] S. Conchon, A. Mebsout, and F. Zaïdi. Certificates for Parameterized Model Checking. In Proc. 20th Int. Symposium on Formal Methods (FM), pages 126–142, 2015.

[22] G. Dantzig. Linear Programming and Extensions. Princeton University Press, 1963.

[23] L. de Moura and N. Bjørner. Satisfiability Modulo Theories: Introduction and Applications. Communications of the ACM, 54(9):69–77, 2011.

[24] B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In Proc. 18th Int. Conf. on Computer Aided Verification (CAV), pages 81–94, 2006.

[25] R. Ehlers. Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks. In Proc. 15th Int. Symp. on Automated Technology for Verification and Analysis (ATVA), pages 269–286, 2017.

[26] Y. Elboher, E. Cohen, and G. Katz. Neural Network Verification using Residual Reasoning. In Proc. 20th Int. Conf. on Software Engineering and Formal Methods (SEFM), 2022.

[27] Y. Elboher, J. Gottschlich, and G. Katz. An Abstraction-Based Framework for Neural Network Verification. In Proc. 32nd Int. Conf. on Computer Aided Verification (CAV), pages 43–65, 2020.

[28] T. Eliyahu, Y. Kazak, G. Katz, and M. Schapira. Verifying Learning-Augmented Systems. In Proc. Conf. of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM), pages 305–318, 2021.

[29] A. Esteva, A. Robicquet, B. Ramsundar, V. Kuleshov, M. DePristo, K. Chou, C. Cui, G. Corrado, S. Thrun, and J. Dean. A Guide to Deep Learning in Healthcare. Nature medicine, 25(1):24–29, 2019.

[30] K. Eykholt, I. Evtimov, E. Fernandes, B. Li, A. Rahmati, C. Xiao, A. Prakash, T. Kohno, and D. Song. Robust Physical-World Attacks on Deep Learning Visual Classification. In Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR), pages 1625–1634, 2018.

[31] D. Fremont, J. Chiu, D. Margineantu, D. Osipychev, and S. Seshia. Formal Analysis and Redesign of a Neural Network-Based Aircraft Taxiing System with VERIFAI. In Proc. 32nd Int. Conf. on Computer Aided Verification (CAV), pages 122–134, 2020.

[32] T. Gehr, M. Mirman, D. Drachsler-Cohen, E. Tsankov, S. Chaudhuri, and M. Vechev. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In Proc. 39th IEEE Symposium on Security and Privacy (S&P), pages 3–18, 2018.

[33] S. Gokulanathan, A. Feldsher, A. Malca, C. Barrett, and G. Katz. Simplifying Neural Networks using Formal Verification. In Proc. 12th NASA Formal Methods Symposium (NFM), pages 85–93, 2020.

[34] B. Goldberger, Y. Adi, J. Keshet, and G. Katz. Minimal Modifications of Deep Neural Networks using Verification. In Proc. 23rd Int. Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR), pages 260–278, 2020.

[35] S. Goldwasser, G. Rothblum, J. Shafer, and A. Yehudayoff. Interactive Proofs for Verifying Machine Learning. In Proc. 12th Innovations in Theoretical Computer Science Conf. (ITCS), 2021.

[36] I. Goodfellow, Y. Bengio, and A. Courville. Deep Learning. MIT Press, 2016.

[37] I. Goodfellow, J. Shlens, and C. Szegedy. Explaining and Harnessing Adversarial Examples, 2014. Technical Report. http://arxiv.org/abs/1412.6572.

[38] A. Griggio, M. Roveri, and S. Tonetta. Certifying Proofs for SAT-Based Model Checking. Formal Methods in System Design, 57(2):178–210, 2021.

[39] The Gurobi Optimizer. https://www.gurobi.com/.

[40] P. Henriksen and A. Lomuscio. Efficient Neural Network Verification via Adaptive Refinement and Adversarial Search. In Proc. 24th European Conf. on Artificial Intelligence (ECAI), pages 2513–2520, 2020.

[41] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu. Safety Verification of Deep Neural Networks. In Proc. 29th Int. Conf. on Computer Aided Verification (CAV), pages 3–29, 2017.

[42] O. Isac, C. Barrett, M. Zhang, and G. Katz. Neural Network Verification with Proof Production, 2022. Technical Report. https://arxiv.org/abs/2206.00512.

[43] Y. Jacoby, C. Barrett, and G. Katz. Verifying Recurrent Neural Networks using Invariant Inference. In Proc. 18th Int. Symposium on Automated Technology for Verification and Analysis (ATVA), pages 57–74, 2020.

[44] K. Jia and M. Rinard. Exploiting Verified Neural Networks via Floating Point Numerical Error. In Proc. 28th Int. Static Analysis Symposium (SAS), pages 191–205, 2021.

[45] K. Julian, M. Kochenderfer, and M. Owen. Deep Neural Network Compression for Aircraft Collision Avoidance Systems. Journal of Guidance, Control, and Dynamics, 42(3):598–608, 2019.

[46] 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.

[47] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In Proc. 29th Int. Conf. on Computer Aided Verification (CAV), pages 97–117, 2017.

[48] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Reluplex: a Calculus for Reasoning about Deep Neural Networks. Formal Methods in System Design (FMSD), 2021.

[49] 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.

[50] G. Katz, D. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljić, D. Dill, M. Kochenderfer, and C. Barrett. The Marabou Framework for Verification and Analysis of Deep Neural Networks. In Proc. 31st Int. Conf. on Computer Aided Verification (CAV), pages 443–452, 2019.

[51] Y. Kazak, C. Barrett, G. Katz, and M. Schapira. Verifying Deep-RL-Driven Systems. In Proc. 1st ACM SIGCOMM Workshop on Network Meets AI & ML (NetAI), pages 83–89, 2019.

[52] O. Lahav and G. Katz. Pruning and Slicing Neural Networks using Formal Verification. In Proc. 21st Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD), pages 183–192, 2021.

[53] Z. Lyu, C.-Y. Ko, Z. Kong, N. Wong, D. Lin, and L. Daniel. Fastened Crown: Tightened Neural Network Robustness Certificates. In Proc.

34th AAAI Conf. on Artificial Intelligence (AAAI), pages 5037–5044, 2020.

[54] A. Makhorin. GLPK (GNU Linear Programming Kit). https://www.gnu.org/s/glpk/glpk.html.

[55] M. Müller, G. Makarchuk, G. Singh, M. Püschel, and M. Vechev. PRIMA: General and Precise Neural Network Certification via Scalable Convex Hull Approximations. In Proc. 49th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL), 2022.

[56] G. Necula. Compiling with Proofs. Carnegie Mellon University, 1998.

[57] M. Ostrovsky, C. Barrett, and G. Katz. An Abstraction-Refinement Approach to Verifying Convolutional Neural Networks. In Proc. 20th. Int. Symposium on Automated Technology for Verification and Analysis (ATVA), 2022.

[58] 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.

[59] L. Pulina and A. Tacchella. Challenging SMT Solvers to Verify Neural Networks. AI Communications, 25(2):117–135, 2012.

[60] I. Refaeli and G. Katz. Minimal Multi-Layer Modifications of Deep Neural Networks. In Proc. 5th Workshop on Formal Methods for ML-Enabled Autonomous Systems (FoMLAS), 2022.

[61] S. Sankaranarayanan, S. Dutta, and S. Mover. Reaching Out Towards Fully Verified Autonomous Systems. In Proc. 13th Int. Conf. on Reachability Problems (RP), pages 22–32, 2019.

[62] G. Singh, T. Gehr, M. Püschel, and M. Vechev. An Abstract Domain for Certifying Neural Networks. In Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL), pages 1–30, 2019.

[63] C. Strong, H. Wu, A. Zeljic,´ K. Julian, G. Katz, C. Barrett, and M. Kochenderfer. Global Optimization of Objective Functions Represented by ReLU Networks. Journal of Machine Learning, pages 1–28, 2021.

[64] 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.

[65] V. Tjeng, K. Xiao, and R. Tedrake. Evaluating Robustness of Neural Networks with Mixed Integer Programming, 2017. Technical Report. http://arxiv.org/abs/1711.07356.

[66] H.-D. Tran, S. Bak, W. Xiang, and T. Johnson. Verification of Deep Convolutional Neural Networks Using ImageStars. In Proc. 32nd Int. Conf. on Computer Aided Verification (CAV), pages 18–42, 2020.

[67] R. Vanderbei. Linear Programming: Foundations and Extensions. Springer, Berlin, 1996.

[68] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Formal Security Analysis of Neural Networks using Symbolic Intervals. In Proc. 27th USENIX Security Symposium, pages 1599–1614, 2018.

[69] H. Wu, A. Ozdemir, A. Zeljic,´ A. Irfan, K. Julian, D. Gopinath, S. Fouladi, G. Katz, C. Pasareanu, and C. Barrett. Parallelization Techniques for Verifying Neural Networks. In Proc. 20th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD), pages 128–137, 2020.

[70] H. Wu, A. Zeljić, K. Katz, and C. Barrett. Efficient Neural Network Analysis with Sum-of-Infeasibilities. In Proc. 28th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pages 143–163, 2022.

[71] T. Zelazny, H. Wu, C. Barrett, and G. Katz. On Reducing Over-Approximation Errors for Neural Network Verification. In Proc. 22nd Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD), 2022.

[72] C. Zhang, T. Su, Y. Yan, F. Zhang, G. Pu, and Z. Su. Finding and Understanding Bugs in Software Model Checkers. In Proc. 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pages 673–773, 2019.

[73] H. Zhang, M. Shinn, A. Gupta, A. Gurfinkel, N. Le, and N. Narodytska. Verification of Recurrent Neural Networks for Cognitive Tasks via Reachability Analysis. In Proc. 24th European Conf. on Artificial Intelligence (ECAI), pages 1690–1697, 2020.

[74] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In Proc. IEEE/ACM Int. Conf. on Computer Aided Design (ICCAD), pages 279–285, 2001.