Proof Minimization in Neural Network Verification

Omri Isac[1(✉)], Idan Refaeli[1], Haoze Wu[2], Clark Barrett[3], and Guy Katz[1]

[1] The Hebrew University of Jerusalem, Jerusalem, Israel
omri.isac@mail.huji.ac.il
[2] Amherst College, Amherst, USA
[3] Stanford University, Stanford, USA

Abstract. The widespread adoption of deep neural networks (DNNs) requires efficient techniques for verifying their safety. DNN verifiers are complex tools, which might contain bugs that could compromise their soundness and undermine the reliability of the verification process. This concern can be mitigated using *proofs*: artifacts that are checkable by an external and reliable proof checker, and which attest to the correctness of the verification process. However, such proofs tend to be extremely large, limiting their use in many scenarios. In this work, we address this problem by minimizing proofs of unsatisfiability produced by DNN verifiers. We present algorithms that remove facts which were learned during the verification process, but which are unnecessary for the proof itself. Conceptually, our method analyzes the dependencies among facts used to deduce `UNSAT`, and removes facts that did not contribute. We then further minimize the proof by eliminating remaining unnecessary dependencies, using two alternative procedures. We implemented our algorithms on top of a proof producing DNN verifier, and evaluated them across several benchmarks. Our results show that our best-performing algorithm reduces proof size by 37%–82% and proof checking time by 30%–88%, while introducing a runtime overhead of 7%–20% to the verification process itself.

1 Introduction

Deep neural networks (DNNs) have made great strides in recent years, becoming the state-of-the-art solution for a variety of tasks in medicine [55], autonomous driving [12], natural language processing [53], and many other domains. However, DNNs lack structure that humans can readily interpret, rendering them opaque and potentially jeopardizing their trustworthiness [19]. One prominent example is the sensitivity of DNNs to small input perturbations, which can lead to significant and undesirable changes to the networks' outputs. This sensitivity can be exploited maliciously [64], possibly leading to catastrophic results.

The pervasiveness of DNNs, combined with their potential vulnerability, has made DNN verification a growing research field within the verification community [2,5,6,15,16,31,33,37,51,54,58,62,66,68,72]. Modern DNN verifiers

Y.-F. Chen et al. (Eds.): VMCAI 2026, LNCS 16417, pp. 99–124, 2026.
https://doi.org/10.1007/978-3-032-15700-3_6

typically use techniques such as SMT solving [1,8,44,52], abstract interpretation [31,33,51,62,69], LP solving [65], and combinations thereof. Many such verifiers are available [15,16], and some tools have even been applied to industrial case studies [3,30,48]. Given a DNN and a property over its input and outputs, DNN verifiers will typically attempt to either find an input to the network that violates the property or conclude that so such input exists.

Despite this success, one significant issue that the DNN verification community faces is related to the *reliability* of verifiers. One concern is that even state-of-the-art verifiers may contain implementation bugs. A second concern is that verifiers typically use floating-point arithmetic, which is crucial for scalability but could lead to rounding errors and numerical instability issues. These issues might be exploited to compromise a verifier's soundness [40,74], thus undermining user trust in these tools.

Due to their complexity, verifying the correctness of DNN verifiers themselves is typically infeasible. Instead, DNN verifiers can produce *proofs* for their verification results, as is commonly done in SAT [36] and SMT [9]. These proofs, which serve as witnesses of correctness of the verification result, constitute a mathematical object and can be checked by an external, trusted *proof checker* [25,26]. When a violation of the property is discovered, the returned counter-example may serve as a proof of correctness. However, proving the absence of a violation is less straightforward, due to the NP-hardness of the problem [39,44,57].

So far, only a handful of attempts have been made to enable DNN verifiers to produce proofs [38]. These approaches typically include eager bookkeeping of many intermediate lemmas deduced during verification [38]. Therefore, the resulting proofs tend to be extremely large, increasing the memory consumption of the verifier and harming the performance of proof checkers [25]. These problems limit the applicability of proof-producing verifiers, and call for further analysis to minimize proof sizes.

In this work, we address this problem by minimizing the proof objects constructed using the framework of Isac et al. [38]. Our proof reduction method involves two main steps. First, we apply a *dependency analysis* procedure that recursively examines the dependencies of each lemma on previously learned lemmas. This helps to identify and remove lemmas that were eagerly learned and stored during the verification process, but are not actually required in the final proof. Second, we apply two alternatives for *dependency minimization* procedures, which further refine the proof by removing unnecessary lemma dependencies, keeping only a minimal subset of dependencies for each lemma. This results in additional lemmas being classified as unused, leading to a smaller proof.

After the termination of the dependency analysis and minimization, the unused lemmas can be safely removed from the proof, without needing to be checked. Furthermore, our method allows lemma deletion on-the-fly, thus avoiding accumulation of unused lemmas and minimizing the memory consumption during the verification process. This, in turn, increases the scalability of proof producing DNN verifiers.

We evaluated our method across multiple benchmarks, using [38] as a baseline. We measured the effectiveness of our algorithms along two axes: the size

of the resulting proof, and the time overhead for applying our algorithms. The results for our best-performing algorithm indicate reduction of proof size of 37%–82% on average and of proof checking time by 30%–88% on average. The reduction incurs a overhead of 7%–20% to the verifier's runtime, where for some benchmarks the average verification time was slightly improved, possibly due to the use of more efficient data structures.

Our contributions are summarized as follows: (i) an algorithm for analyzing the dependencies of each lemma learned within proof objects, removing unused lemmas; (ii) two algorithms for further detecting a minimal subset of dependencies; and (iii) a demonstration of a substantial reduction of the proof size over several benchmarks, while adding a reasonable overhead in performance speed.

The rest of this paper is organized as follows: In Sect. 2 we provide the necessary background on DNNs and their verification, and on proof production. In Sect. 3 we explain our method to reduce the proof size. Section 4 is dedicated to evaluation of our method over several benchmarks, with respect to both proof size and verification speed. In Sect. 5 we discuss related work, and lastly, we conclude and outline ideas for future research in Sect. 6.

2 Background

2.1 Deep Neural Networks and Their Verification

Deep Neural Networks (DNNs) [32]**.** Formally, a DNN $\mathcal{N} : \mathbb{R}^n \to \mathbb{R}^m$ is a sequence of k layers $L_0, ..., L_{k-1}$ where each layer L_i consists of $s_i \in \mathbb{N}$ nodes $v_i^1, ..., v_i^{s_i}$ and s_i biases $b_i^j \in \mathbb{Q}$. Each directed edge in the DNN is of the form (v_{i-1}^l, v_i^j) and is labeled with a weight $w_{i,j,l} \in \mathbb{Q}$. The assignment to the nodes in the input layer is defined by $v_0^j = x_j$, where $\overline{x} \in \mathbb{R}^n$ is the input vector. The assignment for the j^{th} node in the $1 \leq i < k-1$ layer is computed as $v_i^j = f_i\left(\sum_{l=1}^{s_{i-1}} w_{i,j,l} \cdot v_{i-1}^l + b_i^j\right)$ for some activation function $f_i : \mathbb{R} \to \mathbb{R}$. Neurons in the output layer are computed similarly, where f_{k-1} is the identity function.

One of the most common activation functions is the *rectified linear unit* (ReLU), defined as $\mathrm{ReLU}(x) := \max(x, 0)$. The function has two linear phases: if the input is non-negative, the functions is *active*; otherwise, the function is *inactive*. For simplicity, we focus on DNNs with ReLU activations, though our work can be extended to support any piecewise-linear activation (e.g., *maxpool*).

Example 1. *A simple DNN with four layers appears in Fig. 1. For simplicity, all biases are set to* 0 *and are omitted. For input* $\langle 2, 1 \rangle$*, the node in the second layer evaluates to* $ReLU(2 \cdot 1 + 1 \cdot (-1)) = ReLU(1) = 1$*; the nodes in the third layer evaluate to* $ReLU(1 \cdot (-2)) = 0$ *and* $ReLU(1 \cdot 1) = 1$*; and the node in the output layer evaluates to* $0 + 1 \cdot 2 = 2$*.*

The DNN Verification Problem. Consider a DNN $\mathcal{N} : \mathbb{R}^n \to \mathbb{R}^m$ where $\overline{x} \in \mathbb{R}^n, \overline{y} \in \mathbb{R}^m$ denote the network's inputs and outputs. The *DNN verification problem* is the problem of deciding whether there exist $(\overline{x}, \overline{y}) \in \mathbb{R}^{n+m}$ such that

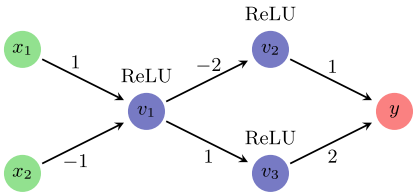

Fig. 1. A toy DNN with ReLU activations.

$(\mathcal{N}(\overline{x}) = \overline{y}) \wedge \varphi_I(\overline{x}) \wedge \varphi_O(\overline{y})$ for some property $\varphi_I \wedge \varphi_O$. If such $\overline{x}, \overline{y}$ exist, we say the problem is satisfiable, denoted `SAT`. Otherwise, we say that it is unsatisfiable, denoted `UNSAT`. In this work, we consider *quantifier-free linear properties*, i.e., formulas without quantifiers whose atoms are linear equations and inequalities.

Example 2. *Consider the DNN from Fig. 1, the input property*

$$\varphi_I(\overline{x}) := (1 \leq x_1) \wedge (1 \leq x_2) \wedge (x_1 \leq 2) \wedge (x_2 \leq 2)$$

and the output property

$$\varphi_O(y) := (y \leq -1).$$

This instance is `UNSAT`, as the output is a sum of two non-negative components with positive weights.

Linear Programming (LP) [24] and DNN Verification. In LP we seek an assignment to a set of variables that satisfies a set of linear constraints, while maximizing a given objective function. As is common in the context of DNN verification, we assume here that the objective function is constant and ignore it. Formally, let $V = [x_1, \ldots, x_n]^\intercal \in \mathbb{R}^n$ denote variables, $A \in M_{m \times n}(\mathbb{R})$ denote a constraint matrix and $l, u \in (\mathbb{R} \cup \{\pm\infty\})^n$ denote vectors of bounds. The LP problem is to decide the satisfiability of $(A \cdot V = 0) \wedge (l \leq V \leq u)$. Throughout the paper, we use $l(x_i)$ and $u(x_i)$, to refer to the lower and upper bounds (respectively) of $x_i \in V$, and denote $l \leq u$ to indicate that for each element i, $l(x_i) \leq u(x_i)$.

A DNN verification query can be encoded as a tuple $Q = \langle V, A, l, u, R \rangle$, where $\langle V, A, l, u \rangle$ constitutes a linear program [24,44] that represents the property and the affine transformations within $\mathcal{N}$, and where R is the set of ReLU activation constraints of the form $f_i = \text{ReLU}(b_i)$, for variables $b_i, f_i \in V$. In addition, a fresh auxiliary variable a_i is added for each ReLU constraint, representing the non-negative difference of its output and input. The variable is added to V, with an equation $a_i = f_i - b_i$ added to A, and the bounds $l(a_i) = 0$, $u(a_i) = u(f_i) - l(b_i)$ added to l, u respectively.

Example 3. *We demonstrate this encoding with the example presented in Fig. 1 and the property from Example 2. Let x_1, x_2 denote the network's inputs and y denote its output, and let b_i, f_i ($i \in \{1, 2, 3\}$) denote the input and output of*

neuron v_i, respectively. The affine transformations of the network are reduced to the equations:

$$\begin{aligned} x_1 - x_2 - b_1 &= 0 \\ 2f_1 + b_2 &= 0 \\ f_1 - b_3 &= 0 \\ f_2 + 2f_3 - y &= 0 \end{aligned}$$

accompanied by $f_i - b_i - a_i = 0$ for $i \in \{1, 2, 3\}$. The property is encoded using the inequalities: $(1 \leq x_1), (1 \leq x_2), (x_1 \leq 2), (x_2 \leq 2), (y \leq -1)$. We also add the inequalities $(0 \leq f_i)$ (for $i \in \{1, 2, 3\}$) to express the non-negativity of ReLUs, and $(0 \leq a_i)$ as described. The aforementioned constraints form the LP part of the query; and its piecewise-linear portion is comprised of the constraints $f_i = ReLU(b_i)$ for $i \in \{1, 2, 3\}$.

Using this encoding, the verification query can be dispatched through a series of invocations of an LP solver, combined with *case splitting* to handle the piecewise-linear constraints [44]. Schematically, the DNN verifier begins by invoking the LP solver over the linear part of the query. An `UNSAT` result of the LP solver implies the overall query is `UNSAT`. Otherwise, the LP solver finds an assignment that satisfies the linear part of the query. If this assignment satisfies the piecewise-linear part of the query, the DNN verifier may conclude `SAT`. If neither case applies, the DNN verifier splits the query into two subqueries, by deciding the phase of a single ReLU constraint $f_i = \text{ReLU}(b_i)$. One subquery is augmented with the bounds $b_i \geq 0$ and $a_i \leq 0$ corresponding to the active phase; the other is augmented with $b_i \leq 0$ and $f_i \leq 0$, corresponding to the inactive phase. Then, the LP solver is invoked again over each subquery, and the process is repeated. Note that the introduction of auxiliary variables is intended to reduce case splitting to bound updates, without needing to add new equations; empirically, this is known to improve performance [70].

This splitting approach creates an abstract tree structure, where each node represents a case split. If the LP solver deduces `UNSAT` for each leaf of the tree, then the original query is `UNSAT` as well. If not, then due to the completeness of LP solvers, it finds a satisfying assignment for one of the subqueries.

The search tree may be exponentially large in the number of piecewise linear constraints (i.e., the number of neurons). Therefore, it is common to use case splitting rarely, and apply algorithms to deduce tighter bounds of variables. These algorithms may conclude in advance that some neurons' phases are fixed, i.e., they are always active or inactive, avoiding the need to split on them [44].

2.2 Proof Production for DNN Verification

When DNN verification is regarded as a satisfiability problem, a satisfying assignment is an efficient proof of `SAT`: one can simply run it through the network and observe that it satisfies the given property. The `UNSAT` case is more complex, and requires producing a *proof certificate*. The first method for proof production

in DNN verification performed via LP solving and case splitting was introduced in [38]. These proof certificates consist of a *proof tree* that replicates the search tree created by the verifier, and is constructed during the verification process. Recall that for an UNSAT query, all leaves of the search tree represent UNSAT subqueries. Thus, each internal node of the proof tree represents a case split performed by the verifier, and each leaf corresponds to a subquery for which the verifier has deduced UNSAT. Therefore a *proof checker* for these proofs is required to traverse the tree, check its structural correctness (i.e., all splits are correct), and certify the unsatisfiability of each leaf.

In each leaf of the search tree, the verifier was able to conclude UNSAT using solely the available linear constraints. Thus, a proof for the leaf's unsatisfiability can be described as a vector, according to the Farkas lemma [21], which combines the linear constraints to derive an evident, easily-checkable, contradiction [38]. More formally:

Theorem 1. (Farkas Lemma Variant (From [38])). *Observe the constraints $A \cdot V = \bar{0}$ and $l \leq V \leq u$, where $A \in M_{m \times n}(\mathbb{R})$ and $l, V, u \in \mathbb{R}^n$. The constraints are UNSAT if and only if $\exists w \in \mathbb{R}^m$ such that for $w^\intercal \cdot A \cdot V := \sum_{i=1}^{n} c_i \cdot x_i$, we have that $\sum_{c_i>0} c_i \cdot u(x_i) + \sum_{c_i<0} c_i \cdot l(x_i) < 0$ whereas $w^\intercal \cdot \bar{0} = 0$. Thus, w is a proof of the constraints' unsatisfiability, and can be constructed during LP solving.*

Example 4. *We demonstrate this theorem with our example. Consider the query based on the DNN in Fig. 1 and the property in Example 2. Assume, for simplicity, that all lower and upper bounds not explicitly stated are -1 or 1, respectively. Further assume that both v_2, v_3 are inactive (i.e. $f_2 = f_3 = 0$ and $b_2, b_3 \leq 0$). The linear part of the query yields the LP instance:*

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

$$u = \begin{bmatrix} 2 & 2 & 1 & 0 & 0 & 1 & 0 & 0 & 2 & 1 & 1 & -1 \end{bmatrix}^\intercal$$
$$V = \begin{bmatrix} x_1 & x_2 & b_1 & b_2 & b_3 & f_1 & f_2 & f_3 & a_1 & a_2 & a_3 & y \end{bmatrix}^\intercal$$
$$l = \begin{bmatrix} 1 & 1 & -1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix}^\intercal$$

Note that in A the first four rows corresponds to the affine transformations of the DNN, and the latter three rows are introduced with the auxiliary variables. Consider the vector

$$w = \begin{bmatrix} 0 & 0 & -1 & -2 & 0 & 0 & 0 \end{bmatrix}^\intercal.$$

The product $w^\intercal \cdot A \cdot V$ yields the equation $-f_1 + b_3 - 2f_2 - 4f_3 + 2y = 0$. The left hand side of the equation is at most: $-l(f_1) + u(b_3) - 2 \cdot l(f_2) - 4 \cdot l(f_3) + 2 \cdot u(y) = -2 < 0$. Therefore, according to Theorem 1, w is a proof of UNSAT. Note that there may be multiple proofs of this result. An overall proof of UNSAT for the query consists of a proof tree, where each leaf has a proof vector proving UNSAT for the corresponding subquery.

Bound Tightening Lemmas. DNN verifiers often employ bound tightening procedures that deduce bounds using the non-linear activations, called *bound tightening lemmas*. Each bound tightening lemma consists of a ground bound and a learned bound, which replaces a bound currently in l or u. As opposed to deduction of bounds using linear equations, proving such lemmas generally cannot be captured by the Farkas vector described above, and require separate proofs [38]. For example, given $f = \mathrm{ReLU}(b)$ and present bounds $f \leq 5$ and $b \leq 7$, it is possible to deduce that $b \leq 5$. The lemma can be proven using a Farkas vector that proves $f \leq 5$, and another proof rule that captures this form of bound derivation.

Example 5. *Consider the query Q in Example 4. We can use the equation $b_2 = -2f_1$, which is equivalent to $2f_1 + b_2 = 0$, to deduce that $u(b_2) = -2 \cdot l(f_1) = 0$. Then, based on the ReLU constraint, we deduce that the neuron v_2 is inactive, i.e., that $u(f_2) = 0$. A lemma with a ground bound $u(b_2) = 0$, learned bound $u(f_2) = 0$ and a proof vector for generating $b_2 = -2f_1$ (the vector $\begin{bmatrix}0 & -1 & 0 & 0 & 0 & 0 & 0\end{bmatrix}^\intercal$) is learned at the root of the proof tree, without any case splits. Similarly, we can deduce that v_3 is inactive. Observe that $f_3 = \frac{1}{2}y - \frac{1}{2}f_2$, and thus $u(f_3) = \frac{1}{2}u(y) - \frac{1}{2}l(f_2) = -0.5$. Then a lemma with a ground bound $u(f_3) = -0.5$, learned bound $u(b_3) = 0$ and a proof vector $\begin{bmatrix}0 & 0 & 0 & -0.5 & 0 & 0 & 0\end{bmatrix}^\intercal$ is learned. Combined with Example 4, we get a proof of* `UNSAT` *for Q that consists of a single node with two lemmas, as illustrated in Fig. 2.*

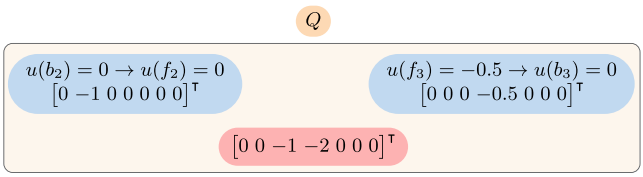

Fig. 2. A proof tree example with a single node (orange). Given the query Q, two lemmas (blue) are learned before deriving `UNSAT` (red). (Color figure online)

To avoid ambiguity in the definition of the bound vectors l, u, unless stated otherwise, l, u denote the tightmost bound vectors, while l_{input}, u_{input} denote the original bound vectors, given as inputs.

3 Proof Dependency Analysis and Minimization

In this section, we elaborate on our algorithms for proof minimization. Recall that each lemma and each leaf within the proof includes a proof vector; and that all these vectors have the same dimension, since the number of rows in A is constant. As proving `UNSAT` of all leaves in the proof tree is necessary for proving `UNSAT` for the whole query, proof minimization is achieved by reducing

the number of lemmas used within the proof. The minimization algorithms are applied upon deducing UNSAT on each proof tree leaf, thus enabling on-the-fly minimization and preventing the accumulation of unnecessary lemmas. We begin by describing our algorithm for dependency analysis in Sect. 3.1 and two algorithms for dependency minimization in Sect. 3.2. We conclude in Sect. 3.3, by introducing an additional minimization technique, which can be used in both algorithms, by explaining the procedure for removing the unused lemmas and by introducing engineering optimizations to improve performance speed.

Algorithm 1. *proofDeps():* Construct the dependency list of a proof vector

Input: A DNN verification subquery $Q = \langle V, A, l, u, R\rangle$, a list of learned lemmas Lem, and a proof of UNSAT w

Output: A list of lemmas sufficient to deduce UNSAT

// Let $\sum_{i=1}^{n} c_i \cdot x_i$ denote the linear combination $w^\intercal \cdot A \cdot V$

$Deps \leftarrow \emptyset$ ▷ An empty list of lemmas
for $i \in [n]$ **do**
  if $c_i > 0$ and $u(x_i)$ is learned by $\ell_i^u \in Lem$ **then**
    $Deps \leftarrow Deps \cup \ell_i^u$
  else if $c_i < 0$ and $l(x_i)$ is learned by $\ell_i^l \in Lem$ **then**
    $Deps \leftarrow Deps \cup \ell_i^l$
  end if
end for
for $lem \in Deps$ **do** ▷ Repeat recursively
  $lem.includeInProof \leftarrow true$
  $l', u' \leftarrow$ bounds with ID at most $lem.getID()$
  $w' \leftarrow lem.getProof()$
  $Deps' \leftarrow proofDeps(\langle V, A', l', u', R\rangle, Lem, w')$
  $Deps \leftarrow Deps \cup Deps'$
end for
return $Deps$

3.1 Analyzing Proof Dependencies

We now describe our algorithm for minimizing proof-trees by dependency analysis. Given a proof vector—either a proof of unsatisfiability for a search-tree leaf, or a proof for a bound tightening lemma—our analysis identifies which bound tightening lemmas are involved in its proof checking process. Our approach for analyzing the dependencies of a given proof vector w appears as Algorithm 1.

Conceptually, Algorithm 1 reconstructs the linear combination $w^\intercal \cdot A \cdot V$, denoted as $\sum_{i=1}^{n} c_i \cdot x_i$. Then, as the proof only uses the bounds $l(x_i)$ for variables with a negative coefficients, and $u(x_i)$ for variables with a positive coefficient, their corresponding lemmas form the *dependency list* for w. The dependency list provides a sufficient set of lemmas for proving the unsatisfiability of w—and this set can potentially be minimized. Then, the algorithm further analyses the

dependencies of the discovered lemmas, only after we know they are used in the proof. Recall that each lemma has its own proof vector, and thus this analysis is carried out through a recursive call on each lemma's proof.

To expedite the algorithm, we store for every bound $l(x_i), u(x_i)$, the lemma or split that was used to deduce it. In addition, to avoid cycles and ensure the completeness of our algorithms, each lemma is given a unique, chronological ID. Whenever a lemma with ID k is analyzed, only lemmas with ID $l < k$ are considered in the analysis.

Example 6. *Recall that in Example 4 and Example 5, after learning two lemmas we are able to derive* `UNSAT` *from a proof vector* $w = [0\ 0\ -1\ -2\ 0\ 0]^\intercal$. *Using this vector generates the equation* $-f_1 + b_3 - 2f_2 - 4f_3 + 2y = 0$, *which uses the bounds* $l(f_1), u(b_3), l(f_2), l(f_3), u(y)$ *to prove* `UNSAT`. *As* $u(b_3)$ *is deduced using one of the lemmas in Example 5, this lemma forms the dependency list of* w. *Then, the algorithm recursively generates the dependency list of the proof vector of the lemma in a similar manner. As shown in Example 5, that lemma is not dependent on* $u(f_2)$. *Thus, the lemma for deducing* $u(f_2)$ *is not actually used in the proof of* `UNSAT`, *and can then be removed from the proof tree. An illustration of this example appears in Fig. 3.*

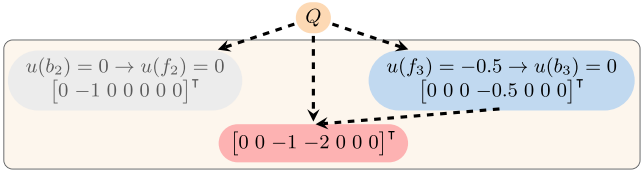

Fig. 3. Using Algorithm 1 over the proof vector used to derive `UNSAT` (red) shows that a single lemma (blue) is used during the proof process, and the other (gray) is not. All proof vectors use information in the root query Q (orange).(Color figure online)

3.2 Minimizing the Number of Proof Dependencies

We now elaborate on our two alternatives of algorithms for minimizing the dependency list of a proof vector. By doing so, we aim to increase the number of lemmas that do not appear in any dependency list, i.e., are unused. Such lemmas can then be omitted from the proof, further reducing its overall size.

Let w be a proof vector proving the unsatisfiability of a search-tree leaf. Our algorithm focuses on minimizing the number of bound tightening lemmas that participate in the refutation. Recall that a refutation is checked by computing the upper bound of $w^\intercal \cdot A \cdot V = \sum_{i=1}^{n} c_i \cdot x_i$, i.e.,

$$\Delta := \sum_{c_i > 0} c_i \cdot u(x_i) + \sum_{c_i < 0} c_i \cdot l(x_i)$$

and ensuring it is negative. The idea of our algorithm lies in the fact that Δ can be very far from zero. In this case, we can relax some of the bounds derived from the participating bound lemmas, without altering the sign of Δ, and thus maintain the proof's correctness. Analyzing the dependencies of a proof vector $\hat{w}$, used to prove a bound tightening lemmas, is performed similarly. In this case, Δ represents the difference between the bound it is used to prove and the actual bound achieved with $\hat{w}^\intercal \cdot A \cdot V$.

Proof Minimization. The first algorithm for proof minimization is greedy, and removes the maximal number of unnecessary dependencies for each proof vector separately. To do so, we first define for each lemma its *contribution to* w, which can be seen as its effect on Δ. More formally, given the input bound vectors l_{input}, u_{input}, and the bound vectors l, u with bounds derived by a set of bound lemmas and case splits. For each $c_j > 0$, the contribution of $u(x_j)$ to $w^\intercal \cdot A \cdot V$ is defined as:

$$cont(u_j, w) := c_j \cdot (u(x_j) - u_{input}(x_j))$$

For each $c_j < 0$, the contribution of $l(x_j)$ to $w^\intercal \cdot A \cdot V$ is:

$$cont(l_j, w) := c_j \cdot (l_{input}(x_j) - l(x_j))$$

Note that in all cases the contribution is non-positive, as $l_{input} \leq l \leq u \leq u_{input}$.

These definitions naturally give rise to Algorithm 2, which sorts the various lemmas according to their contributions to the proof, from small to large. Those lemmas with only a small contribution could potentially be omitted from the proof, without altering the sign of Δ. Attempting to remove lemmas in this order ensures the removal of the maximum number of lemmas from the dependency list, and consequently the minimality of the output, as we prove in Theorem 2. Algorithm 2 can be used within Algorithm 1, before any recursive application of the algorithm. By that, Algorithm 2 reduces the time overhead of Algorithm 1, by reducing the number of the latter's recursive calls.

Theorem 2. *Given* $\langle V, A, l, u, R \rangle$*, a proof vector* w*, a list of its lemma dependencies* D*, and the input bound vectors* l_{input}, u_{input}*, Algorithm 2 returns a dependency subset of Deps that is minimal in size.*

Proof. Assume towards contradiction that there exists D', which constitutes a dependency list of w, with $|D'| < |D|$. First, observe that if $D' \subsetneq D$, then there is some lemma $\ell \in D \setminus D'$ that can be removed from D when considered in line 6, this contradicts the definition of the algorithm.

Therefore, $D \setminus D'$ and $D' \setminus D$ are not empty. Let $\ell \in D \setminus D'$, with the minimal contribution δ, and let $\ell' \in D' \setminus D$ with the minimal contribution δ'. Since Algorithm 2 is greedy, we must have that $\delta' \leq \delta$, as otherwise the algorithm would have removed ℓ from D before removing ℓ'. This means that we can replace ℓ' with ℓ in D' without increasing the overall contribution removed from *Deps*. We can repeat the process until saturation, i.e., when $D' \subsetneq D$, which leads to a contradiction. □

Algorithm 2. *proofMin()*: Minimize the dependency list of a proof vector

Input: A DNN verification subquery $Q = \langle V, A, l, u, R \rangle$, a proof vector w, a list of its lemma dependencies *Deps*, and the input bound vectors l_{input}, u_{input}
Output: A minimial dependency list of w

// Let $\sum_{i=1}^{n} c_i \cdot x_i$ denote the linear combination $w^\intercal \cdot A \cdot V$
$\Delta \leftarrow \sum_{c_i > 0} c_i \cdot u(x_i) + \sum_{c_i < 0} c_i \cdot l(x_i)$
$Deps \leftarrow Deps.sortByContribution()$
$\delta \leftarrow 0$
for $dep \in Deps$ **do**
  $\delta += |cont(dep.getBoundType(), w)|$ ▷ Use $A, V, l, u, l_{input}, u_{input}$
  if $\delta \leq |\Delta|$ **then**
    $Deps.remove(dep)$
  else return $Deps$
  end if
end for
return $\emptyset$

Example 7 *In Example 6 we saw that the proof vector* $w = \begin{bmatrix} 0 & 0 & -1 & -2 & 0 & 0 \end{bmatrix}^\intercal$, *presented in Example 4, is dependent on only one of the two lemmas learned as in Example 5. In addition, the analysis in Example 4 shows that* $\Delta = -2$, *and that the proof vector* w *is dependent on the lemma used to deduce* $u(b_3) = 0$, *whose input bound is* $u(b_3) = 1$. *Therefore, the contribution of the lemma is* $1 \cdot (0 - 1) = -1$, *indicating this lemma can be removed from the dependency list as well. Observe that indeed, given the input bounds, we have:*

$$-l(f_1) + u_{input}(b_3) - 2 \cdot l(f_2) - 4 \cdot l(f_3) + 2 \cdot u(y) = -1 < 0$$

Together with Example 6, we conclude that both lemmas from Example 5 can be removed from the proof tree. An illustration for this example appears in Fig. 4.

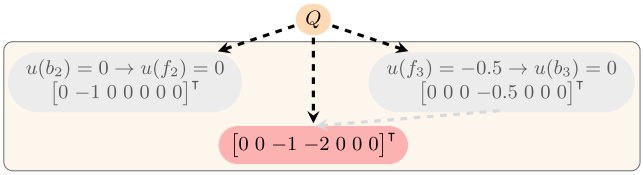

Fig. 4. Using Algorithm 2 over the proof vector used to derive `UNSAT` (red) shows that both lemmas (gray) are unnecessary for proving `UNSAT`.

Global Heuristic for Proof Minimization. Although Algorithm 2 guarantees achieving the minimum number of dependencies for each proof vector, this does not necessarily ensure achieving the global minimum across the entire proof tree. To incorporate global considerations, we adjust the order in which lemmas

are considered for removal, prioritizing those with more dependencies and whose elimination is likely to trigger the removal of additional downstream lemmas.

Specifically, before applying the minimization algorithm, we recursively apply the dependency analysis procedure to each lemma and compute its total number of dependencies, including those discovered through the recursive applications of the minimization procedure. The lemmas are then sorted in decreasing order by their overall dependency count (i.e., lemmas with more dependencies are considered for removal first). In case of ties, we fall back on the previous contribution score. We apply the analysis until saturation, meaning no further reduction of dependencies is possible. This ensures minimality of the subset of dependencies, though it does not guarantee the resulting subset is smallest. This can occur if Algorithm 3 diverges significantly from Algorithm 2, for example, when lemmas with high dependency counts contribute heavily. In such cases, the former algorithm may remove these lemmas first, while the latter will instead remove many more lemmas with a lower dependency count. To maintain the applicability of the algorithm during proof generation, we apply it each time an `UNSAT` leaf is detected, considering all lemmas learned globally up to that point.

This intuition is formalized in Algorithm 3, and it opens up possibility to explore additional heuristics for prioritizing lemmas for removal, e.g., by considering different combinations of each lemma's number of dependencies and its contribution.

Note that introducing global considerations for proof minimization involves dependency analysis for many lemmas, and thus might incur a relatively large time overhead.

3.3 Additional Improvements and Lemma Deletion

Further Reduction of Proof Size. Both minimization algorithms presented above consider the dependencies for each proof vector separately. However, it is possible that minimizing dependencies at the proof-vector level leads to redundant dependencies at the proof-tree level. Consider the case where two proof vectors w_1, w_2 depend on a lemma ℓ, and that the dependencies of w_1 are minimized before those of w_2. If the dependency minimization of w_1 indicates that ℓ is retained in the proof, then removing it from the dependencies of w_2 will not lead to the removal of ℓ from the overall proof, and will unnecessarily increase the contributions counter δ Algorithm 2 (line 5). Therefore, a straightforward improvement to both minimization algorithms is to consider removing dependencies only from lemmas that are not currently used in the proof. This approach allows us to skip lemmas that are already essential and would be retained regardless.

Algorithm 3. *globProofMin()*: Minimize the dependency list of a proof vector

Input: A DNN verification subquery $Q = \langle V, A, l, u, R \rangle$, a proof vector w, a list of its lemma dependencies $Deps$, the input bound vectors l_{input}, u_{input}, and the overall list of learned lemmas Lem

Output: A minimial dependency list of w

// Let $\sum_{i=1}^{n} c_i \cdot x_i$ denote the linear combination $w^\intercal \cdot A \cdot V$

$\Delta \leftarrow \sum_{c_i > 0} c_i \cdot u(x_i) + \sum_{c_i < 0} c_i \cdot l(x_i)$
$Deps \leftarrow Deps.sortByDepsLength(Q, Deps, Lem)$
$\delta \leftarrow 0$
$saturation \leftarrow false$
while $!saturation$ **do**
  $saturation \leftarrow true$
  for $dep \in Deps$ **do**
    $\delta += |cont(dep.getBoundType(), w)|$ ▷ Use $A, V, l, u, l_{input}, u_{input}$
    if $\delta \leq |\Delta|$ **then**
      $Deps.remove(dep)$
      $saturation \leftarrow false$
    else
      $\delta -= |cont(dep.getBoundType(), w)|$ ▷ Revert
    end if
  end for
end while
return $Deps$

sortByDepsLength$(Q, Deps, Lem)$:
for $\ell \in Deps$ **do**
  if $!\ell.wasAnalyzed$ **then**
    $\ell.score \leftarrow proofDeps(Q, Lem, \ell.getProof()).size()$
    $\ell.tieBreaker \leftarrow \ell.getContribution()$
    $\ell.wasAnalyzed \leftarrow true$
  end if
end for
return Deps.sortByScore()

Example 8 *Consider a lemma ℓ_1 with $\Delta_1 = -1$. Suppose that proof analysis of Algorithm 1 determines that ℓ_1 is dependent on ℓ_2, ℓ_3 with respective contributions of -0.5 and -0.6. Suppose also that other lemmas depend on ℓ_2, so it is retained in the proof, but none depend on l_3 yet. In this case, Algorithm 2 would first remove the dependency of ℓ_1 in ℓ_2 and then stop as $|-0.5| + |-0.6| > |-1|$. Therefore, this application of Algorithm 2 will keep both ℓ_2 and ℓ_3 in the proof. Alternatively, by ignoring ℓ_2 as it is already used in the proof, and removing the dependency on ℓ_3 instead, we may reduce the overall number of lemmas required for the proof.*

Avoiding Duplicate Computations. Recall that Algorithm 1 recursively iterates through lemmas and analyzes their dependencies. Since multiple lemmas can

depend on the same lemma, it is useful to avoid redundant or repeated analysis. To address this, we add a flag to each lemma indicating whether it has already been analyzed. This ensures that each lemma is analyzed at most once.

Unused Lemma Deletion. To reduce the memory consumption of the DNN verifier during the verification process, we propose to remove unused lemmas as early as possible. Since each lemma is applicable only in the search state S where it was deduced and in all descendant states of S, it becomes irrelevant once the sub-tree rooted at S has been fully explored. At that point, any lemma in S that was not involved in any dependency analysis can be safely deleted.

Supporting Additional Constraints. Although so far we focused on DNNs with ReLU constraints, our minimization algorithms can be applied to additional kinds of constraints in a straightforward manner. Therefore, assuming the underlying verifier supports proof production for additional kinds of piecewise-linear constraints, the minimization algorithms are readily applicable.

4 Implementation and Evaluation

To assess the effectiveness of our proposed proof minimization techniques, we conducted an empirical evaluation across six benchmark suites in neural network verification (ordered alphabetically): (i) the Acas-Xu drone collision avoidance networks [41]; (ii) the cersyve benchmark [71] for verification of neural safety certificate in control systems; (iii) a subset of the Cora benchmark [15,47], where disjuncts from the original queries where separated into different queries; (iv) MaxLeaky, a fresh benchmark that includes simple queries over a newly-trained DNN employing *Maxpool* and *LeakyRelu* [27] activation functions; (v) a robotic navigation system benchmark [4]; and, (vi) the SafeNLP benchmark [17,18]. These benchmarks were selected due to their relevance to safety-critical applications their diversity in size and activation functions, and their established use in previous work or at the DNN verification competition (VNNCOMP) [15,16]. Notably, we included all VNNCOMP benchmarks for which the proof-producing version of Marabou reasonably scales.

4.1 Experimental Setup and Implementation

Experiments were conducted on machines running Debian 12 Linux, each on a single CPU. We divided the benchmarks into two sets according to network sizes, with Acas-Xu and Cora consisting of larger DNNs, and the remaining benchmarks consisting of smaller DNNs. Memory and time configurations depended on category: for the larger benchamrks we used 16GB RAM and a 5-hour Timeout, and for the smaller ones—2GB RAM and a 2.5-hour Timeout.

Our implementation is based on the proof-producing version of the Marabou DNN verifier [38,46,70], into which we integrated our proof minimization algorithms and which we used as a baseline. The key metrics evaluated include (i) *Proof size:* the number of proof vectors (for both lemmas and leaves) in

the produced proofs of UNSAT queries; (ii) *Total Runtime:* the total runtime of the verification process, for all queries; and (iii) *Total Proof-Checking time:* the total proof-checking time for the UNSAT queries, using the native Marabou proof checker. Note that a partial proof object is constructed during verification in satisfiable queries as well (before concluding SAT); and thus, measuring verification time for all queries offers a more comprehensive comparison.

4.2 Evaluation

We begin by presenting a summary of the experimental results in Table 1, which reports the average proof size generated during proof production, the average total verification time, and the average total proof-checking time, for each benchmark and proof minimization technique. In this table, averaging was conducted with respect to queries solved by all methods. Furthermore, we provide a visual summary in Fig. 5, which compares the proof sizes of all UNSAT queries, and overall runtime across all queries. We evaluated three variants of our method: (i) *Dependency Analysis* (**Dep. Analysis**), as described in Algorithm 1; (ii) *Dependency Minimization* (**Dep. Min.**), which includes the implementation of Algorithm 2 on top of Algorithm 1; and (iii) *Global Dependency Minimization* (**Dep. Min. Glob.**), which replaces Algorithm 2 with Algorithm 3. For a deeper study of the effect on the total runtime, we consider an additional running configuration, **Min. Baseline**, which includes all data structures used for implementing our algorithms as exemplified in Sect. 3.1, but without actually executing Algorithm 1. By running this version, we are able to separate the running time effect of the changes in data structures, from the effect of the analysis and minimization algorithms themselves. All methods were implemented with the optimizations detailed in Sect. 3.3.

The results demonstrate a clear reduction in average proof size, achieving a reduction rate of 37%–82%. The incurred verification time overhead is 5%–24% for **Dep. Min.** and 7%–20% for **Dep. Min. Glob.**. In addition, the results indicate a reduction in proof checking time of up to 88%. We observe that this reduction is not necessarily correlated with the reduction rate of proof sizes; and we hypothesize that this stems from the fact that checking time of each lemma is dependent on its sparsity, and that our approach retains sparser lemmas in the proof tree. Comparing the minimization algorithms, the results show that Algorithm 1 is responsible for most of the removed lemmas, and both Algorithm 2 and Algorithm 3 improve it further with generally similar impact, except for the CORA benchmarks. However, both algorithms incur a small overhead comparing to using Algorithm 1 alone. Analyzing **Min. Baseline** indicates that changes in data structures contributed to a minor improvement in both verification time and proof-checking time, for most benchmarks. This has moderated the time overhead of the minimization algorithms. As expected, bookeeping more information slightly increased the number of overall MEMOUTs of **Min. Baseline** compared to **Baseline**. When applying the minimization algorithms however, the number of MEMOUTs decreases significantly, due to the reduction of proof size (see subsections below).

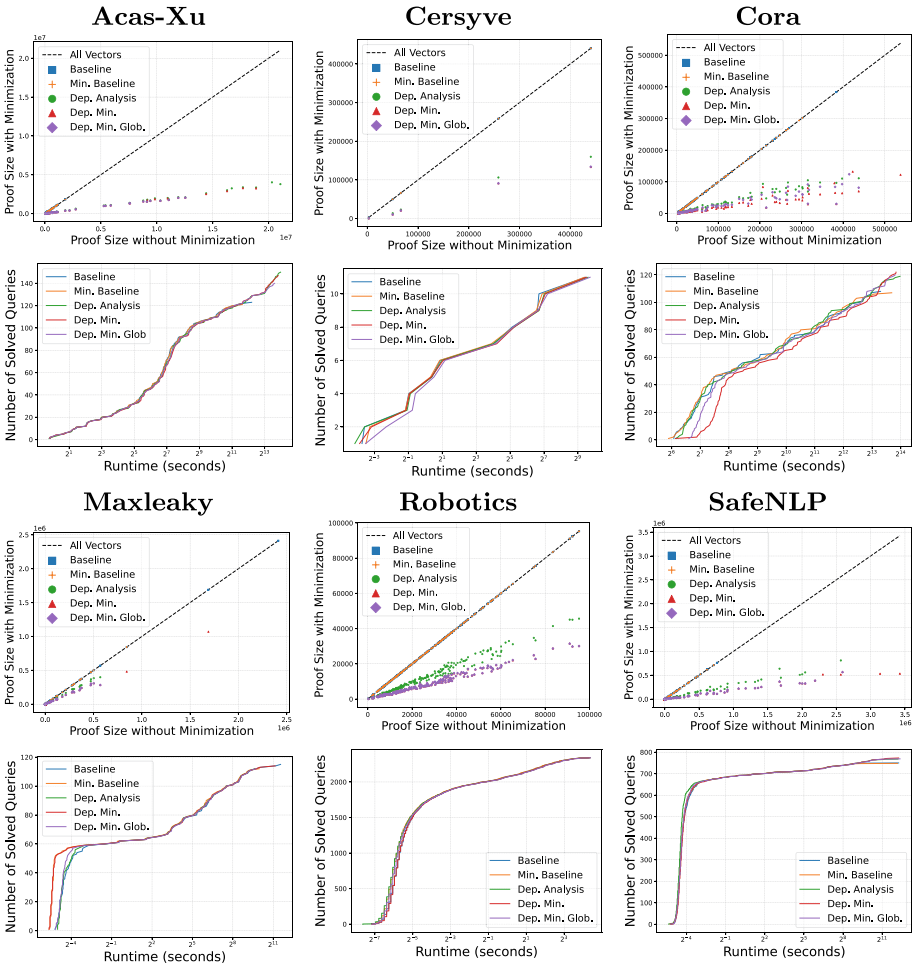

Fig. 5. Proof size and runtime comparison across all benchmarks and methods.

In the subsequent subsections, we provide a more detailed analysis of the results for each benchmark suite.

Acas-Xu Benchmark. The Acas-Xu suite consists of fully connected feedforward networks trained to support drone collision avoidance decision-making. We ran experiments on a set of 45 networks, each comprised of 6 hidden layers, each of which contained 50 neurons with ReLU activation functions. For each network we verified 4 different specifications [44]. This results in a total of 180 queries. The comparison in Table 1 was conducted on 82 `UNSAT` and 39 `SAT` queries solved by all methods. Notably, a single query was either solved with a failure during proof certification, or terminated due to memory out. This result is consistent

Table 1. Average proof size (number of leaves and lemmas), verification time for all queries and proof-checking time for UNSAT queries, using Marabou across benchmarks. Lower values indicate more compact proofs. Each non-baseline entry includes both reported numbers, and percentage of change with respect to baseline. Best results among our three algorithms are marked in **bold**.

Benchmark	Avg. Proof Size [±% of Baseline] Avg. Verification Time (Sec.) [±% of Baseline] Avg. Checking Time (Sec.) [±% of Baseline]				
	Baseline[38]	Min. Baseline	Dep. Analysis	Dep. Min.	Dep. Min. Glob.
Acas-Xu	302021.14 265.98 66.37	302021.14 0% 269.42 +1.2% 46.58 -29.8%	59231.07 -80.4% 299.96 +12.8% 8.67 -86.9%	**52890.17** **-82.5%** 294.58 +10.7% **7.78** **-88.3%**	53144.84 -82.4% **288.42** **+8.4%** 8.15 -87.7%
cersyve	163592.4 86.99 13.77	163592.4 0% 84.84 -2.5% 13.66 -0.8%	60764.6 -62.9% 96.90 +11.4% 3.14 -77.2%	**51283.6** **-68.6%** **93.02** **+6.9%** **2.62** **-81%**	51492.6 -68.5% 104.57 +20.2% 2.77 -79.9%
Cora	49202.05 1277.88 1633.47	49202.05 0% 1231.5 -3.6% 1572.66 -3.7%	15199.58 -69.1% **1262.87** **-1.2%** 455.88 -72.1%	**8819.75** **-82.1%** 1583.02 +23.9% **266.17** **-83.7%**	11549.88 -76.5% 1405.65 +10% 365.32 -77.6%
MaxLeaky	55342.88 60.85 2.67	55342.88 0% 56.34 -7.4% 2.63 -1.5%	42224.96 -23.7% **59.01** **-3%** 2.07 -22.5%	34852.52 -37% 59.7 -1.9% **1.89** **-29.2%**	**34847.12** **-37%** 60.01 -1.4% **1.89** **-29.2%**
Robotics	25549.65 0.55 0.32	25549.65 0% 0.53 -3.6% 0.31 -3.1%	11981.7 -53.1% **0.57** **+3.6%** 0.15 -53.1%	**8016.64** **-68.6%** 0.58 +5.4% **0.11** **-65.6%**	8026.99 -68.6% 0.59 +7.3% **0.11** **-65.6%**
SafeNLP	55066.71 14.69 5.89	55066.71 0% 14.27 -2.8% 7.78 +32%	17135.91 -68.9% **14.73** **+0.3%** 1.59 -73%	10118.98 -81.6% 14.94 +1.7% **0.96** **-83.7%**	**10062.95** **-81.7%** 15.88 +8.1% 0.97 -83.5%

with the original evaluation of the Acas-Xu benchmark in [38], and we hypothesize that for this particular verification instance, the proof-producing Marabou is numerically unstable.

To assess robustness under resource constraints, we also report the number of queries that were successfully solved within the 5-hour timeout and 16GB memory limit. Table 2 summarizes the number of solved queries for each method.

As shown, all three minimization methods outperform the baseline, solving more queries by avoiding memory blowup during proof construction.

Table 2. Number of ACAS-XU `SAT` queries solved, `UNSAT` queries solved, timed out, and failed due to memory exhaustion, out of 180 queries.

Method	#SAT	#UNSAT	#Timeout	#Memout
Baseline	41	82	0	57
Min. Baseline	39	82	0	59
Dep. Analysis	44	107	29	0
Dep. Min.	44	104	21	11
Dep. Min. Glob.	44	96	20	20

cersyve Benchmark. The CERSYVE suite consists of fully connected feedforward networks serving as neural certificates for control systems. We ran experiments on a set of 12 networks, with 65 to 198 ReLU neurons. For each network we verified a single property, as in the VNNCOMP 2025 benchmark [67]. This results in a total of 12 queries, out of which 6 are `UNSAT`. For this benchmark, all methods successfully verified 11 queries, whereas a single `UNSAT` query fails due to memory restrictions when tested on all methods.

Cora Benchmark. The CORA benchmark consists of local adversarial robustness queries for image classifiers trained on several datasets. For our experiments, we used a subset of these queries on a feedforward DNN with seven layers of 250 ReLU neurons each, trained on the MNIST dataset. To extend the benchmark and improve scalability, each query was further divided into nine sub-queries, each corresponding to a single possible misclassification option. In total, this yields 189 queries. The comparison in Table 1 was conducted on 104 `UNSAT` queries solved by all methods.

In Table 3, we report the number of instances verified under a time limit of 5 h and a memory limit of 16GB. As previously observed, the analysis and min-

Table 3. Number of CORA `SAT` queries solved, `UNSAT` queries solved, timed out, and failed due to memory exhaustion, out of 189 queries.

Method	#SAT	#UNSAT	#Timeout	#Memout
Baseline	1	107	81	0
Min. Baseline	1	106	81	1
Dep. Analysis	1	118	70	0
Dep. Min.	0	122	67	0
Dep. Min. Glob.	1	119	69	0

imization algorithms increase the total number of solved queries, this time primarily due to improvements in overall running time. Since the reduction in proof-checking time is more significant than the reduction in verification time (as shown in Table 1), we conclude that the overall improvement is mainly attributable to the reduced time of proof-checking.

MaxLeaky Benchmark. The MaxLeaky benchmark includes a single DNN trained to classify wine types based on chemical composition [23]. To test our approach over a DNN which employs various activation functions, we trained a fresh DNN with 64 ReLU, 32 LeakyReLU, and 16 Maxpool neurons, to classify the dataset. We then generated a single adversarial robustness query for 133 train and 44 test data points. This results in a total of 177 queries. The comparison in Table 1 was conducted on 99 `UNSAT` and 12 `SAT` queries solved by all methods.

Table 4 includes the number of instances verified under the limitation of 2.5-hour time and 2GB memory. The results indicate a minor reduction in the number of solved queries when introducing the minimization algorithms. This result is unique, and is likely a result of the relatively moderate reduction of proof size (see Table 1), combined with the small overhead introduced when keeping additional information for each lemma (see Sect. 3.1).

Table 4. Number of MaxLeaky `SAT` queries solved, `UNSAT` queries solved, timed out, and failed due to memory exhaustion, out of 177 queries.

Method	#`SAT`	#`UNSAT`	#Timeout	#Memout
Baseline	12	163	1	1
Min. Baseline	12	160	0	5
Dep. Analysis	12	160	1	4
Dep. Min.	12	162	0	3
Dep. Min. Glob.	12	160	0	5

Robotics Navigation Benchmark. The robotic navigation benchmark [4] involves DNN controllers for obstacle avoidance. We tested our approach on 2340 queries from this benchmark, where 2126 of them are `SAT` queries, and 214 are `UNSAT`. Each query includes a DNN with the ReLU activation only. The DNN is comprised of an input layer with 9 neurons, two hidden layers with 16 neurons each, and an output layer of 3 neurons. For this benchmark, all methods were able to verify all queries and produce proofs in the time and memory limitations.

SafeNLP Benchmark. The safeNLP benchmark [17,18] examines robustness to word- and sentence-level perturbations in the embedding space of several sentences over two fully-connected DNNs. Both DNNs have the same architecture, which is comprised of an input layer of 30 nodes, a single hidden layer

of 128 nodes with ReLU activation functions, and an output layer of 2 nodes. The benchmark consists of 1080 queries in total. The comparison in Table 1 was conducted on 149 `UNSAT` and 599 `SAT` queries solved by all methods.

Table 5 reports the number of queries successfully verified within the resource limits (2.5 h and 2GB RAM). Here again, our minimization methods demonstrate improved performance, reducing memory exhaustion and increasing the number of successful verification queries compared to the baseline.

Table 5. Number of safeNLP `SAT` queries solved, `UNSAT` queries solved, timed out, and failed due to memory exhaustion, out of 1080 queries.

Method	#`SAT`	#`UNSAT`	#Timeout	#Memout
Baseline	599	196	13	272
Min. Baseline	599	194	7	280
Dep. Analysis	599	214	36	231
Dep. Min.	599	219	54	208
Dep. Min. Glob.	599	215	42	224

5 Related Work

Producing proofs as witnesses of correctness is a common practice in both SMT and SAT solving. In both communities, the proof size tends to be significant, limiting their applicability [7,9,36,63]. Our work builds on concepts inspired by *lazy proofs* [45], where proof objects are constructed gradually, using only necessary lemmas. In DNN verification, proof production has been studied only recently, and in a handful of works [38,61]. The authors of [61] consider a symbolic approach for increasing the reliability of DNN verifiers, which reduces the problem to an additional SMT query. Our approach is then not directly applicable to [61].

The study of dependencies analysis of `UNSAT` results has been explored in various contexts [14,35,42,43,49] often using the name ***`UNSAT`*** *core*. To the best of our knowledge, ours is the first to investigate this in the context of DNN verification. In DNN verification, previous studies have attempted to analyze dependencies between the phases of neurons as part of the DNN verification process [13]. As fixing neuron phases could be captured by proof lemmas, it would be interesting to study the synergies between that approach and ours.

In linear programming, the well-established concept of *Irreducible Infeasible Systems* [20] is defined as a minimal set of constraints that maintain unsatisfiablity. The method to detect IISs from [20] has been used in DNN verifiers [29], and is implemented in the Gurobi linear optimizer [34] often used in DNN verifiers. Although IIS detection algorithms typically rely on repeated LP solver invocations, which can be computationally expensive, it presents a compelling direction for future exploration in this work's context.

6 Conclusion and Future Work

In this work, we developed methods to reduce the size of `UNSAT` proofs generated by DNN verifiers, as presented in [38]. These proofs can be checked by a trusted, external proof checker in order to improve reliability of the DNN verifier. Importantly, DNN verifiers currently assume ideal mathematical implementation of the DNN itself, and modeling any specific DNN implementations remains a challenge for the DNN verification community in general [22]. Our goal here is to improve memory consumption in proof-producing verifiers and reduce proof checking time, thereby enhancing the practicality of reliable DNN verification.

To this end, we designed an algorithm that analyzes, for each lemma in the proof, its dependencies on previously derived lemmas. This enables us to eliminate lemmas that are eagerly learned but are ultimately irrelevant for proving `UNSAT`. In addition, we designed an algorithm to minimize the number of such dependencies, with two variants. All algorithms have been implemented on top of the proof-producing version of the Marabou DNN verifier [70].

Our results show that Algorithm 1 reduces proof size by an order of magnitude across multiple benchmarks. The dependency minimization algorithms further reduces proof size, though more moderately. When comparing the two variants of the minimization algorithm, both versions achieve similar results on almost all benchmarks. Moreover, the performance overhead introduced by our methods is reasonable, especially when considering improvements in proof checking time. Notably, the time overhead from the dependency analysis is reduced due to the use of some alternative data structures.

Future Work. Moving forward, we intend to continue this work along several paths. First, we aim to explore different variants of proof minimization. In particular, our current algorithms consider a fixed proof vector w, which we could try to alter in order to reduce the number of dependencies even further. Additionally, it would be interesting to explore additional alternatives to the minimization algorithm, attempting to remove dependencies according to other heuristics, e.g., by their stage of deduction. Furthermore, it would also be interesting to consider the sparsity of the proof vectors and prioritize removing dense vectors, as they (heuristically) depend on many other bound lemmas.

Second, we aim to extend the proof checker as in [25], to support checking proofs of bound tightening lemmas. We will then evaluate the reduction of proof-checking time for this version of the checker as well.

A third direction for future research is extending our work to support additional subroutines commonly used in DNN verifiers, particularly abstract interpretation. This integration is inherently more complex, as it requires proving linear relaxations of non-linear constraints. However, once this support is established, we believe our proof minimization approach and technique could also be useful in this context.

Fourth, our work's analysis method paves the way for a *conflict analysis* method, as a part of *Conflict-Driven Clause Learning* [10,59,60] scheme, which is commonly used in SAT and SMT solving [8,11]. Conceptually, in search-based

solving, CDCL identifies subspaces of the search space which are "similar" to subspaces already traversed, and which were shown not to contain any satisfying assignment. This similarity guarantees that the new subspaces also do not contain any such assignment, and they can be safely skipped by the verifier. A core component in CDCL is the generation of *conflict clauses*, a representation of an `UNSAT` subspaces, that guide the solver not to search in other subspaces that are also guaranteed to be `UNSAT`. Naturally, the algorithm for deriving conflict clauses is a key to a successful CDCL framework. Our work can serve as a basis for the derivation of potentially useful conflict clauses for DNN verifiers—leading to a significant improvement in their performance. Although several DNN verifiers employ CDCL or similar algorithms [28,29,50,73], to the best of our knowledge, none of these use `UNSAT` proofs for deriving conflict clauses.

Acknowledgments. The work of Isac, Refaeli and Katz was supported by the Binational Science Foundation (grant numbers 2020250 and 2021769), the Israeli Science Foundation (grant number 558/24) and the European Union (ERC, VeriDeL, 101112713). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

The work of Barrett was supported in part by the Binational Science Foundation (grant number 2020250), the National Science Foundation (grant numbers 1814369 and 2211505), and the Stanford Center for AI Safety.

Data Availability Statement. The implementation and scripts used for the experiments in Sect. 4 are publicly available [56].

References

1. Abraham, E., Kremer, G.: SMT solving for arithmetic theories: theory and tool support. In: Proceedings of the 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), pp. 1–8 (2017)
2. Akintunde, M., Kevorchian, A., Lomuscio, A., Pirovano, E.: Verification of RNN-based neural agent-environment systems. In: Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI), pp. 197–210 (2019)
3. Amir, G., Freund, Z., Katz, G., Mandelbaum, E., Refaeli, I.: veriFIRE: verifying an industrial, learning-based wildfire detection system. In: Proceedings of the 25th International Symposium on Formal Methods (FM), pp. 648–656 (2023)
4. Amir, G., et al.: Verifying learning-based robotic navigation systems. In: Proceedings of the 29th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 607–627 (2023)
5. Avni, G., Bloem, R., Chatterjee, K., Henzinger, T., Konighofer, B., Pranger, S.: Run-time optimization for learned controllers through quantitative games. In: Proceedings of the 31st International Conference on Computer Aided Verification (CAV), pp. 630–649 (2019)
6. Baluta, T., Shen, S., Shinde, S., Meel, K., Saxena, P.: Quantitative verification of neural networks and its security applications. In: Proceedings of the 26th ACM Conference on Computer and Communication Security (CCS) (2019)

7. Barbosa, H., et al.: Generating and Exploiting Automated Reasoning Proof Certificates. Commun. ACM **66**(10), 86–95 (2023)
8. Barrett, C., Tinelli, C.: Satisfiability Modulo Theories. In: Clarke, E., Henzinger, T., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 305–343. Springer International Publishing (2018)
9. Barrett, C., de Moura, L., Fontaine, P.: Proofs in satisfiability modulo theories. All about Proofs, Proofs for All **55**(1), 23–44 (2015)
10. Bayardo Jr, R., Schrag, R.: Using CSP look-back techniques to solve real-world SAT Instances. In: Proceedings of the 14th National Conference on Artificial Intelligence (AAAI), pp. 203–208 (1997)
11. Biere, A., Faller, T., Fazekas, K., Fleury, M., Froleyks, N., Pollitt, F.: CaDiCaL 2.0. In: Proceedings of the 36th International Conference on Computer Aided Verification (CAV), pp. 133–152 (2024)
12. Bojarski, M., et al.: End to end learning for self-driving cars (2016). technical Report. http://arxiv.org/abs/1604.07316
13. Botoeva, E., Kouvaros, P., Kronqvist, J., Lomuscio, A., Misener, R.: Efficient verification of relu-based neural networks via dependency analysis. In: Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI), pp. 3291–3299 (2020)
14. Bradley, A., Manna, Z.: Property-directed incremental invariant generation. Formal Aspects Comput. **20**, 379–405 (2008)
15. Brix, C., Bak, S., Johnson, T., Wu, H.: The Fifth International Verification of Neural Networks Competition (VNN-COMP 2024): Summary and Results (2024). technical Report. http://arxiv.org/abs/2412.19985
16. Brix, C., Müller, M., Bak, S., Johnson, T., Liu, C.: First three years of the international verification of neural networks competition (VNN-COMP). Int. J. Softw. Tools Technol. Transfer 1–11 (2023)
17. Casadio, M., et al.: ANTONIO: towards a systematic method of generating NLP benchmarks for verification. In: Proc. 6th Workshop on Formal Methods for ML-Enabled Autonomous Systems (FoMLAS), vol. 16, pp. 59–70 (2023)
18. Casadio, M., et al.: NLP verification: towards a general methodology for certifying robustness. Europ. J. Appl. Math. 1–58 (2025)
19. Cheng, C.H., et al.: Neural networks for safety-critical applications — challenges, experiments and perspectives. In: Proceedings of the International Conference on Design, Automation and Test in Europe (DATE), pp. 1005–1006. IEEE (2018)
20. Chinneck, J., Dravnieks, E.: Locating minimal infeasible constraint sets in linear programs. ORSA J. Comput. **3**(2), 157–168 (1991)
21. Chvátal, V.: Linear Programming. W. H, Freeman and Company (1983)
22. Cordeiro, L.C., et al.: Neural Network Verification is a Programming Language Challenge. In: Proceedings of the 34th European Symposium on Programming (ESOP), pp. 206–235 (2025)
23. Cortez, P., Cerdeira, A., Almeida, F., Matos, T., Reis, J.: Modeling Wine Preferences by Data Mining from Physicochemical Properties. Decis. Support Syst. **47**(4), 547–553 (2009)
24. Dantzig, G.: Linear Programming and Extensions. Princeton University Press (1963)
25. Desmartin, R., Isac, O., Komendantskaya, E., Stark, K., Passmore, G., Katz, G.: a certified proof checker for deep neural network verification in imandra. In: Proceedings of the 16th International Conference on Interactive Theorem Proving (ITP), pp. 1–21 (2025)

26. Desmartin, R., Isac, O., Passmore, G., Stark, K., Komendantskaya, E., Katz, G.: Towards a certified proof checker for deep neural network verification. In: Proceedings of the 33rd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR), pp. 198–209 (2023)
27. Dubey, S., Singh, S., Chaudhuri, B.: Activation functions in deep learning: a comprehensive survey and benchmark. Neurocomputing (2022)
28. Duong, H., Nguyen, T., Dwyer, M.: A DPLL(T) Framework for Verifying Deep Neural Networks (2023), technical Report. http://arxiv.org/abs/2307.10266
29. Ehlers, R.: Formal verification of piece-wise linear feed-forward neural networks. In: Proceedings of the 15th International Symposium on Automated Technology for Verification and Analysis (ATVA), pp. 269–286 (2017)
30. Elboher, Y., et al.: Robustness assessment of a runway object classifier for safe aircraft taxiing. In: Proceedings of the 43rd International Digital Avionics Systems Conference (DASC) (2024)
31. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, E., Chaudhuri, S., Vechev, M.: AI2: safety and robustness certification of neural networks with abstract interpretation. In: Proceedings of the 39th IEEE Symposium on Security and Privacy (S&P) (2018)
32. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press (2016)
33. Goubault, E., Palumby, S., Putot, S., Rustenholz, L., Sankaranarayanan, S.: Static analysis of ReLU neural networks with tropical polyhedra. In: Proceedings of the 28th International Static Analysis Symposium (SAS), pp. 166–190 (2021)
34. The Gurobi Optimizer. https://www.gurobi.com/
35. Guthmann, O., Strichman, O., Trostanetski, A.: Minimal unsatisfiable core extraction for SMT. In: In Proceedings 16th International Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 57–64 (2016)
36. Heule, M.J., Biere, A.: Proofs for satisfiability problems. All about Proofs, Proofs for all **55**(1), 1–22 (2015)
37. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: Proceedings of the 29th International Conference on Computer Aided Verification (CAV), pp. 3–29 (2017)
38. Isac, O., Barrett, C., Zhang, M., Katz, G.: Neural network verification with proof production. In: Proceedings of the 22nd International Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 38–48 (2022)
39. Isac, O., Zohar, Y., Barrett, C., Katz, G.: DNN verification, reachability, and the exponential function problem. In: Proceedings of the 34th International Conference on Concurrency Theory (CONCUR) (2023)
40. Jia, K., Rinard, M.: Exploiting Verified Neural Networks via Floating Point Numerical Error. In: Proc. 28th Int. Static Analysis Symposium (SAS). pp. 191–205 (2021)
41. Julian, K., Kochenderfer, M., Owen, M.: Deep Neural Network Compression for Aircraft Collision Avoidance Systems. J. Guid. Control. Dyn. **42**(3), 598–608 (2019)
42. Junker, U.: Quickxplain: conflict detection for arbitrary constraint propagation algorithms. In: Proceedings of the Workshop on Modelling and Solving Problems with Constraints (2001)
43. Junker, U.: Quickxplain: preferred explanations and relaxations for over-constrained problems. In: Proceedings of the 19th National Conference on Artificial Intelligence (AAAI), pp. 167–172 (2004)
44. Katz, G., Barrett, C., Dill, D., Julian, K., Kochenderfer, M.: Reluplex: a Calculus for Reasoning about Deep Neural Networks. Formal Methods in System Design (FMSD) (2021)

45. Katz, G., Barrett, C., Tinelli, C., Reynolds, A., Hadarean, L.: Lazy Proofs for DPLL(T)-based SMT solvers. In: Proceedings of the 16th International Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 93–100 (2016)
46. Katz, G., et al.: The marabou framework for verification and analysis of deep neural networks. In: Proceedings of the 31st International Conference on Computer Aided Verification (CAV), pp. 443–452 (2019)
47. Koller, L., Ladner, T., Althoff, M.: Set-Based Training for Neural Network Verification (2024), technical Report. http://arxiv.org/abs/2401.14961
48. Kouvaros, P., Leofante, F., Edwards, B., Chung, C., Margineantu, D., Lomuscio, A.: Verification of semantic key point detection for aircraft pose estimation. In: Proceedings of the 20th International Conference on Principles of Knowledge Representation and Reasoning (KR), pp. 757–762 (2023)
49. Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. J. Autom. Reason. **40**, 1–33 (2008)
50. Liu, Z., Yang, P., Zhang, L., Huang, X.: DeepCDCL: A CDCL-based neural network verification framework. In: Proceedings of the 18th International Symposium on Theoretical Aspects of Software Engineering (TASE), pp. 343–355 (2024)
51. Lyu, Z., Ko, C.Y., Kong, Z., Wong, N., Lin, D., Daniel, L.: Fastened crown: tightened neural network robustness certificates. In: Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI), pp. 5037–5044 (2020)
52. de Moura, L., Bjørner, N.: Satisfiability modulo theories: introduction and applications. Commun. ACM **54**(9), 69–77 (2011)
53. OpenAI: ChatGPT. https://chatgpt.com
54. Pulina, L., Tacchella, A.: An Abstraction-Refinement Approach to Verification of Artificial Neural Networks. In: Proceedings of the 22nd International Conference on Computer Aided Verification (CAV), pp. 243–257 (2010)
55. Rajpurkar, P., Chen, E., Banerjee, O., Topol, E.J.: AI in health and medicine. Nat. Med. **28**(1), 31–38 (2022)
56. Refaeli, I., Isac, O.: Proof Minimization in Neural Network Verification (Artifact) (2025). https://doi.org/10.5281/ZENODO.17169557, https://zenodo.org/doi/10.5281/zenodo.17169557
57. Sälzer, M., Lange, M.: Reachability Is NP-complete even for the simplest neural networks. In: Proceedings of the 15th International Conference on Reachability Problems (RP), pp. 149–164 (2021)
58. Sankaranarayanan, S., Dutta, S., Mover, S.: Reaching out towards fully verified autonomous systems. In: Proceedings of the 13th International Conference on Reachability Problems (RP), pp. 22–32 (2019)
59. Silva, J., Sakallah, K.: GRASP-a new search algorithm for satisfiability. In: Proceedings of the 15th International Conference on Computer Aided Design (ICCAD), pp. 220–227 (1996)
60. Silva, J.M., Sakallah, K.A.: GRASP: a search algorithm for propositional satisfiability. IEEE Trans. Comput. **48**(5), 506–521 (1999)
61. Singh, A., Sarita, Y.C., Mendis, C., Singh, G.: Automated verification of soundness of DNN Certifiers. Proc. ACM Programm. Lang. **9** (2025)
62. Singh, G., Gehr, T., Puschel, M., Vechev, M.: An abstract domain for certifying neural networks. In: Proceedings of the 46th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL) (2019)
63. Sörensson, N., Biere, A.: Minimizing learned clauses. In: Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT), pp. 237–243. Springer (2009)

64. Szegedy, C., et al.: Intriguing Properties of Neural Networks (2013), technical Report. http://arxiv.org/abs/1312.6199
65. Tjeng, V., Xiao, K., Tedrake, R.: Evaluating Robustness of Neural Networks with Mixed Integer Programming (2017), technical Report. http://arxiv.org/abs/1711.07356
66. Tran, H.D., Bak, S., Xiang, W., Johnson, T.: Verification of deep convolutional neural networks using imagestars. In: Proceedings of the 32nd International Conference on Computer Aided Verification (CAV), pp. 18–42 (2020)
67. The VNNCOMP 2025 Benchmarks Repository. https://github.com/VNN-COMP/vnncomp2025_benchmarks (2025)
68. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Formal security analysis of neural networks using symbolic intervals. In: Proceedings of the 27th USENIX Security Symposium, pp. 1599–1614 (2018)
69. Wang, S., et al.: Beta-crown: efficient bound propagation with per-neuron split constraints for neural network robustness verification. In: Proceedings of the 35th Conference on Neural Information Processing Systems (NeurIPS (2021)
70. Wu, H., et al.: Marabou 2.0: a versatile formal analyzer of neural networks. In: Proceedings of the 36th International Conference on Computer Aided Verification (CAV) (2024)
71. Yang, Y., Hu, H., Wei, T., Li, S.E., Liu, C.: Scalable synthesis of formally verified neural value function for Hamilton-Jacobi reachability analysis. J. Artif. Intell. Res. **83** (2025)
72. Zhang, H., Shinn, M., Gupta, A., Gurfinkel, A., Le, N., Narodytska, N.: Verification of recurrent neural networks for cognitive tasks via reachability analysis. In: Proceedings of the 24th European Conference on Artificial Intelligence (ECAI), pp. 1690–1697 (2020)
73. Zhou, D., Brix, C., Hanasusanto, G.A., Zhang, H.: Scalable neural network verification with branch-and-bound inferred cutting planes. In: Proceedings of the 38th Conferenc on Neural Information Processing Systems (NeurIPS) (2024)
74. Zombori, D., Bánhelyi, B., Csendes, T., Megyeri, I., Jelasity, M.: Fooling a complete neural network verifier. In: Proceedings of the 9th International Conference on Learning Representations (ICLR) (2021)