# A Certified Proof Checker for Deep Neural Network Verification in Imandra

**Remi Desmartin**[1]
Heriot-Watt University, Edinburgh, UK

**Omri Isac**[1]
The Hebrew University of Jerusalem, Israel

**Grant Passmore**
Imandra Inc., Austin, TX, USA

**Ekaterina Komendantskaya**
Southampton University, UK
Heriot-Watt University, Edinburgh, UK

**Kathrin Stark**
Heriot-Watt University, Edinburgh, UK

**Guy Katz**
The Hebrew University of Jerusalem, Israel

## Abstract

Recent advances in the verification of deep neural networks (DNNs) have opened the way for a broader usage of DNN verification technology in many application areas, including safety-critical ones. However, DNN verifiers are themselves complex programs that have been shown to be susceptible to errors and numerical imprecision; this, in turn, has raised the question of trust in DNN verifiers. One prominent attempt to address this issue is enhancing DNN verifiers with the capability of producing certificates of their results that are subject to independent algorithmic checking. While formulations of Marabou certificate checking already exist on top of the state-of-the-art DNN verifier Marabou, they are implemented in C++, and that code itself raises the question of trust (e.g., in the precision of floating point calculations or guarantees for implementation soundness). Here, we present an alternative implementation of the Marabou certificate checking in Imandra – an industrial functional programming language and an interactive theorem prover (ITP) – that allows us to obtain full proof of certificate correctness. The significance of the result is two-fold. Firstly, it gives stronger independent guarantees for Marabou proofs. Secondly, it opens the way for the wider adoption of DNN verifiers in interactive theorem proving in the same way as many ITPs already incorporate SMT solvers.

**2012 ACM Subject Classification** Software and its engineering → Functional languages; Software and its engineering → Formal software verification; Computing methodologies → Neural networks; Theory of computation → Logic and verification

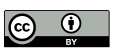
**Keywords and phrases** Neural Network Verification, Farkas Lemma, Proof Certification

**Digital Object Identifier** 10.4230/LIPIcs.ITP.2025.1

**Related Version** *Preprint*: `https://arxiv.org/abs/2405.10611`

**Supplementary Material** *Software (Source Code)*: `https://github.com/rdesmartin/imandra-marabou-proof-checking`

[1] Both authors contributed equally







**Funding** The work of Komendantskaya was partially supported by the EPSRC grant AISEC: AI Secure and Explainable by Construction (EP/T026960/1) and ARIA grant "Safeguarded AI". Desmartin acknowledges Imandra sponsorship for his PhD studies. The work of Isac and Katz was partially funded by 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.

## 1 Introduction

As part of the AI revolution in computer science, deep neural networks (DNNs) are becoming the state-of-the-art solution to many computational problems, including in safety-critical applications, e.g., in medicine [41], aviation [24] and autonomous vehicles [10]. DNNs are complex functions, with parameters *optimised* (or *trained*) to fit a large set of input-output examples. Therefore, DNNs are intrinsically opaque for humans and are known for their vulnerability to errors in the presence of distribution shifts [42]. This raises the question of the reliability of DNNs in safety-critical domains and calls for tools that guarantee their safety [15].

One family of such tools are *DNN verifiers*, which allow to either mathematically prove that a DNN complies with certain properties or provide a counterexample for a violation [14]. They are typically based on SMT-solving with bound tightening (Marabou [45]), interval bound propagation ($\alpha\beta$-CROWN [44]), or abstract interpretation (AI2 [25]). But implementing these verifiers is only a first step to full safety guarantees. Even complete DNN verifiers are prone to implementation bugs and numerical imprecision that, in turn, might compromise their soundness and can be maliciously exploited, as was shown in e.g. [15, 29, 46].

One could consider verifying DNN verifiers directly. However, as Wu et al. exemplify, a mature DNN verifier is a complex multi-platform library; its direct verification would be close to infeasible [45]. A similar problem was faced by the SMT solvers community [34, 6, 5, 18]. A solution was found: to produce (and then check) a *proof certificate* for each automatically generated proof, instead of checking the verifier in its entirety [33]. The software that checks the proof certificate is called the *proof checker*. Ideally, the proof checker should: a) be significantly simpler than the original verifier and b) yield strong guarantees of code correctness (i.e. be formally verified). For Marabou, the first half of this research agenda was accomplished by Isac et al. [27] with proof checking for Marabou being reduced to the application of the *Farkas lemma* [43], a well-known solvability theorem for a finite system of linear equations, along with a tree structure that reflects Marabou's verification procedure. However, the proof checker in [27] came with no formal guarantees of its own correct implementation.

In this paper, we address this problem by deploying the well-established principles of *proof-carrying code* [35, 31] and *self-certifying code* [33]. Namely, we implement formal proof production from Marabou certificates in the interactive theorem prover Imandra [37]. Imandra can implement – and prove properties of – programs written in a subset of OCaml, and thus avoids a potential discrepancy between implementation and formalisation. In Imandra, we obtain a certificate checker implementation that can be safely executed *and* proven correct in the same language. Imandra supports arbitrary precision real arithmetic, and thus does not suffer from floating point imprecision that haunts DNN verifiers. Finally, it strikes a good balance between interactive and automated proving: i.e., it features strong automation while admitting user interaction with the prover via tactics. We exhibit the role of the Imandra checker within the verification process in Figure 1.

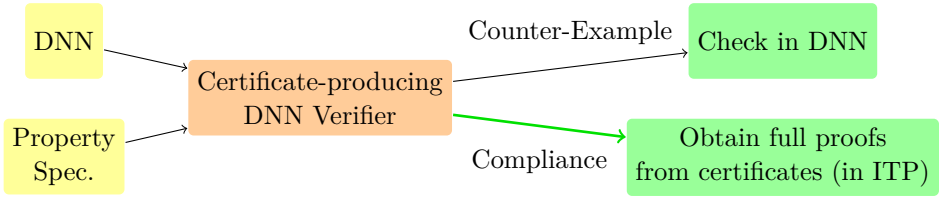



**Figure 1** Model of trust: a neural network (DNN) and a property specification verified by a certificate producing DNN Verifier (e.g. Marabou). The SAT/UNSAT certificate is then checked by a trusted program in both cases. This paper addresses verification of the `UNSAT` certificate, using the ITP Imandra.

**Contributions.** Reconstruction of full correctness proofs from Marabou certificates rests on two major results:

- Firstly, the Imandra implementation of an algorithm that reconstructs a proof of validity of Marabou certificates. The latter are given by *Marabou proof trees* with a DNN verification query at the root, nodes corresponding to partitions of the search space, and Farkas vectors at the leaves. We get for free that our implementation of the algorithm is free from floating point instability, unlike the C++ version in [27].
- Secondly, a formal proof of soundness of the algorithm proven in Imandra, that guarantees that if this algorithm returns `UNSAT` then indeed the given set of polynomial constraints at the root of the tree has no solution. This proof relies on a new formal proof of the *DNN variant* of the famous Farkas lemma in Imandra.

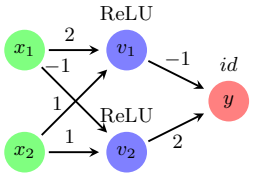
Finally, we demonstrate that our verified certificate checker can be executed in practice. We analyze the code complexity, run it on two verification benchmarks, and evaluate its performance speed against the original certificate checker and verifier [27]. Our results suggest an expected trade-off between the reliability and scalability of the proof checking process, with Imandra code taking about 4.56-4.76 times as long as original verification and checking time, on average.

The paper proceeds as follows. We start with explaining the essence of Marabou certificate production in Section 2 and introducing the Imandra proof of the Farkas lemma in Section 3. Section 4 introduces the new certificate checking algorithm in Imandra, and Section 5 proves its soundness. Finally, Section 6 is devoted to practical evaluation of the verified checker, and Section 7 concludes the paper.

# 2 Background

## 2.1 Deep Neural Networks as Graphs

A DNN is usually defined as a function $\mathcal{N} : \mathbb{R}^{k_I} \to \mathbb{R}^{k_O}$, with inputs $\mathbf{x_I} = x_1, \ldots, x_{k_I}$ and outputs $\mathbf{x_O} = y_1, \ldots, y_{k_O}$. We represent a DNN by a directed, acyclic, connected and weighted graph $(V, E)$, where $V$ is a finite set of vertices $\{v_1, \ldots, v_k\}$ and $E$ is a set of edges of the form $\{(v_i, v_j)\}$, equipped with a *weight function* $w : E \to \mathbb{R}$, and an *activation function* $a_v : \mathbb{R} \to \mathbb{R}$ for each node $v$. A vertex $v_i$ is called an *input vertex* if there is no $v_j$ s.t. $(v_j, v_i) \in E$ and an *output vertex* if there is no $v_j$ s.t. $(v_i, v_j) \in E$. The following picture shows an example of a DNN represented as a graph:




We assume that all input vertices are assigned a value, using $f(v_i) = r \in \mathbb{R}$. For all other vertices, we define the *weighted sum* ($z$) and *neuron* ($f$) functions as follows:

$$z(v_j) = \sum_{(v_i,v_j)\in E} w(v_i, v_j) \cdot f(v_i) \qquad\qquad f(v_j) = a_{v_j}(z(v_j)) \tag{1}$$

Note that in this paper, we will use only two activation functions: *rectified linear unit* (ReLU), defined as $\text{ReLU}(x) = \max(x, 0)$ and the identity function *id*. However, in principle, this work can be extended to support DNNs with any piecewise linear activation function.

To cohere with the existing literature, we will use $x$ and $y$ (with indices) to refer to input and output vertices and $\mathbf{x_I}$ and $\mathbf{x_O}$ to lists of input and output vertices. For brevity, we will call them simply *inputs* and *outputs*.

## 2.2 DNN Verification Queries

Given a variable vector $\mathbf{x}$ of size $k_I$, and $\mathbf{l}, \mathbf{u} \in \mathbb{R}^{k_I}$, a *bound property* $P(\mathbf{x})$ is defined by inequalities $l_j \leq x_j \leq u_j$ for each element $x_j$ of $\mathbf{x}$. Given a DNN $\mathcal{N}$, an *input bound property* $P(\mathbf{x_I})$ and a *linear arithmetic* output property $Q(\mathbf{x_O})$, such that $P(\mathbf{x_I}) \wedge Q(\mathcal{N}(\mathbf{x_I}))$. If such $\mathbf{x_I}$ exists, the verification problem $\langle \mathcal{N}, P, Q \rangle$ is *satisfiable* (`SAT`); otherwise, it is *unsatisfiable* (`UNSAT`). Typically, $P(\mathbf{x_I}) \wedge Q(\mathcal{N}(\mathbf{x_I}))$ represent an erroneous behavior; thus an input $\mathbf{x_I}$ and output $\mathcal{N}(\mathbf{x_I})$ that satisfy the query serve as a counterexample and `UNSAT` implies safe behavior. Although our approach supports any linear property for the output, we focus on bound properties for simplicity. This definition can be extended to more sophisticated properties of DNNs, that express constraints on the a DNN's internal variables [26].

Recall that all DNNs we consider 1.) come with affine (i.e., linear polynomial) weight functions and 2.) piecewise linear activation functions. As a consequence, any DNN verification problem can be reduced to piecewise linear constraints that represent the activation functions, accompanied with a Linear Programming (LP) [17] instance, which represents the affine functions and the input/output properties. One prominent approach for solving the DNN verification constraints is using algorithms for solving LP, coupled with a *case-splitting* approach for handling the piecewise linear constraints [7, 30].

One example of such a DNN verification algorithm is the Reluplex [30] algorithm, which extends the Simplex algorithm [17, 21] for solving LP problems. Based on a DNN verification problem $\langle \mathcal{N}, P, Q \rangle$, the algorithm initiates:

- a variable vector $\mathbf{x}$ containing variables $\mathbf{x_I}$ and $\mathbf{x_O}$, and fresh variables $x_{z(v_i)}$ and $x_{f(v_i)}$ whose values represent weighted sum and neuron functions $z(v_i)$ and $f(v_i)$ for all $v_i \in V$ which are neither an input nor an output. For each node $v_i \in V$ with the ReLU activation function, the algorithm initiates an additional fresh *auxiliary variable* $x_{aux_i}$, to represent the non-negative difference $f(v_i) - z(v_i)$.
- two *bound vectors* $\mathbf{u}, \mathbf{l}$, giving upper and lower bounds to each element in $\mathbf{x}$. The values for $\mathbf{l}$ and $\mathbf{u}$ are generated as follows: values of $\mathbf{x_I}$ and $\mathbf{x_O}$ are given directly by $P(\mathbf{x_I})$ and $Q(\mathbf{x_O})$. The remaining values of $\mathbf{l}$ and $\mathbf{u}$ are computed by propagating forward the lower and upper bounds of $\mathbf{x_I}$ through $\mathcal{N}$ using equations (1) and (2). For variables $x_{f(v_i)}, x_{z(v_i)}, x_{aux_i}$, we denote their lower and upper bounds $\mathbf{l}_{f(v_i)}, \mathbf{l}_{z(v_i)}, \mathbf{l}_{aux_i}$ and $\mathbf{u}_{f(v_i)}, \mathbf{u}_{z(v_i)}, \mathbf{u}_{aux_i}$, respectively.

- a matrix $A$, called *tableau*, which represents the equations given in (1), with additional *auxiliary* equations. It is computed by defining a system of equations of the form:

$$\sum_{(v_i,v_j)\in E} w(v_i,v_j)\cdot x_{f(v_i)} - x_{z(v_i)} = 0 \qquad x_{f(v_i)} - x_{z(v_j)} - x_{aux_i} = 0 \tag{2}$$

  one for each non-input node $v_i$. In $A$, we record these coefficients as follows. Let $A \in \mathbb{R}^{m\times n}$, where $n$ is the size of $\mathbf{x}$ and $m$ is the number of equations we generated as in (2). Each entry $(k, j)$ in $A$ contains the coefficient of the $j^{th}$ variable in $\mathbf{x}$ in the $k^{th}$ equation. We then obtain $A \cdot \mathbf{x} = \mathbf{0}$, where $\cdot$ is the dot product, and $\mathbf{0}$ is the vector of $m$ zeros.
- a set $C$ of *ReLU constraints* for variables in $\mathbf{x}$, given by $x_{f(v_i)} = ReLU(x_{z(v_i)})$ for all non-input nodes $v_i$, such that $f(v_i) = ReLU(z(v_i))$ (as per equation (2)). We call the variables $x_{f(v_i)}$, $x_{z(v_i)}$ and $x_{aux_i}$ the *participating variables* of a constraint. We can join all $c \in C$ in a conjunction: $\bigwedge c$, and denote the resulting formula by $\mathcal{C}(\mathbf{x})$. Intuitively, $\mathcal{C}(\mathbf{x})$ holds if the variables in $\mathbf{x}$ satisfy the constraints in $C$.

The tuple $\langle A, \mathbf{u}, \mathbf{l}, C\rangle$ is a *DNN verification query*, and the tuple $\langle A, \mathbf{u}, \mathbf{l}\rangle$ is a *linear query*. We use the notation $\mathbf{x}[\mathbf{x}_i/a]$ for substituting the $i^{th}$ element in a vector $\mathbf{x}$ by the value $a$.

▶ **Example 1** (DNN Verification Query). Consider the DNN in Section 2.1, the input bound property $P$ that holds if and only if $(x_1, x_2) \in [0, 1]^2$ and the output bound property $Q$ that holds if and only if $y \in [4, 5]$. We first obtain $\mathbf{x}$:

$\mathbf{x} = \begin{bmatrix} x_1 & x_2 & x_{z(v_1)} & x_{z(v_2)} & x_{f(v_1)} & x_{f(v_2)} & x_{aux_1} & x_{aux_2} & y \end{bmatrix}^\intercal$ Next, $\mathbf{l}$ and $\mathbf{u}$ are computed as follows. Firstly, $P$ and $Q$ already give the bounds for $x_1$, $x_2$, $y$. For the rest, we propagate the input bounds. We get:

$$0 \le x_1, x_2 \le 1,\ 0 \le x_{z(v_1)}, x_{f(v_1)} \le 3,\ -1 \le x_{z(v_2)} \le 1,\ 0 \le x_{f(v_2)} \le 1,$$
$$0 \le x_{aux_1} \le 3,\ 0 \le x_{aux_2} \le 2,\ 4 \le y \le 5 \tag{3}$$

It only takes to assemble these values into vectors $\mathbf{l}$ and $\mathbf{u}$:

$$\mathbf{u} = \begin{bmatrix} 1 & 1 & 3 & 1 & 3 & 1 & 3 & 2 & 5 \end{bmatrix}^\intercal$$
$$\mathbf{l} = \begin{bmatrix} 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 4 \end{bmatrix}^\intercal$$

To obtain $A$, we first get all equations as in (2): $2x_1 + x_2 - x_{z(v_1)} = 0$, $x_2 - x_1 - x_{z(v_2)} = 0$, $2x_{f(v_2)} - x_{f(v_1)} - y = 0$, $x_{f(v_1)} - x_{z(v_1)} - x_{aux_1} = 0$, $x_{f(v_2)} - x_{z(v_2)} - x_{aux_2} = 0$.
The first equation suggests we have coefficients: 2 for $x_1$, 1 for $x_2$ and $-1$ for $z(v_1)$. Because this equation does not feature any other variables in $\mathbf{x}$, we record zero coefficients for all other variables; which gives the first row in the matrix $A$. We continue in the same manner for the remaining equations:

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

Finally, the set of ReLU constraints is given by: $\{x_{f(v_1)} = \text{ReLU}(x_{z(v_1)}), x_{f(v_2)} = \text{ReLU}(x_{z(v_2)})\}$.

▶ **Definition 2** (Solution to DNN verification query). *Given a vector $\mathbf{s} \in \mathbb{R}^n$, variable assignment $\mathbf{x} = \mathbf{s}$ is a* solution of the DNN verification query $\langle A, \mathbf{u}, \mathbf{l}, C\rangle$ *if:*

$$(A \cdot \mathbf{x} = \mathbf{0} \wedge \mathbf{l} \le \mathbf{x} \le \mathbf{u} \wedge \mathcal{C}(\mathbf{x}))[\mathbf{x}/\mathbf{s}] \tag{4}$$

*We define the predicate* `is_solution`$(\langle A, \mathbf{u}, \mathbf{l}, C\rangle, \mathbf{s})$ *that returns* true *if $\mathbf{s}$ is a solution to $\langle A, \mathbf{u}, \mathbf{l}, C\rangle$. In case $C$ is empty, we will write* `is_solution`$(\langle A, \mathbf{u}, \mathbf{l}\rangle, \mathbf{s})$.

## 2.3 DNN Verification in Marabou

In order to find a solution for a DNN verification query, or conclude there is none, DNN verifiers such as Marabou [45] rely on algorithms such as Simplex [17] for solving the linear part of the query (i.e. $\langle A, \mathbf{u}, \mathbf{l} \rangle$), and enhance it with a splitting approach for handling the piecewise linear constraints (i.e. ReLU) in $C$. Conceptually, the DNN verifier invokes the linear solver, which either returns a solution to the linear part of the query or concludes there is none. If no solution exists, the DNN verifier concludes the whole query is `UNSAT`. If a solution is provided, then the verifier checks whether it satisfies the remaining ReLU constraints. If so, the verifier concludes `SAT`. If not, case splitting is applied, and the process repeats for every case. For constraints of the form $f = \text{ReLU}(z)$, a case-split divides the query into two subqueries: one enhances the linear part of the query with $f = 0 \wedge u_z = 0$ and the other with $aux = 0 \wedge l_z = 0$. Adding $aux = 0$ is equivalent to adding $x_f = x_v$, thus by adding the auxiliary variable, case splitting is performed by updating the bounds ($\mathbf{u}_{aux} = \mathbf{l}_{aux} = 0$) without adding new equations. Besides splitting over a piecewise linear constraint, splitting can be performed over a single variable $x$ whose value can either be less or equal than some constant $c \in \mathbb{R}$ or greater or equal than the constant; i.e., one subquery is enhanced with $u_x = c$ and the other with $l_x = c$. This induces a tree structure, with nodes corresponding to the splits. If the verifier answers a subquery, then its node represents a leaf, as no further splits are performed. A tree with all leaf subqueries are `UNSAT` corresponds to an `UNSAT` query, and inversely a tree with a single `SAT` leaf corresponds to a `SAT` query. Note that in every such `UNSAT` leaf, unsatisfiability is deduced using the linear part of the query.

Also, note that such a DNN verification scheme heavily relies on a solver for linear equations, which often manipulates the matrix $A$, as well as many optimizations and heuristics for splitting strategy. A proof checker avoids implementing any of those operations.

# 3 Farkas Lemma for DNN Verification

Recall that the idea of a proof checker assumes that we can obtain a proof witness that the checker can certify independently. Since a solution to a DNN verification query is a satisfiability problem, a satisfying assignment serves as a straightforward (and independently checkable) proof witness of `SAT`. However, providing a proof witness for the `UNSAT` case was an open question, based on the NP-hardness of the problem [40]. The novelty of Isac et al.'s result [27] was in showing that the Farkas lemma [43] can be reformulated constructively in terms of the DNN verification problem, and that the vectors in its construction can be then used to witness the `UNSAT` proof. We will call this specialized constructive form of the Farkas lemma the *DNN Farkas lemma* (cf. § 3.1).

To use the DNN Farkas lemma as part of the verified checker, we need to prove its correctness in Imandra. Indeed, there exist *Mathematical Components* [1] proofs of the lemma in its original form [2, 39]. However, those formalisations rely strongly on comprehensive MathComp matrix libraries. Other ITP attempts [11, 8, 36] have suggested ways to prove the Farkas lemma directly in terms of systems of linear polynomial equations (abbreviated *systems of l.p.e.* for short), obtaining the original Farkas lemma as a corollary (cf. Figure 2). We will call this form of the Farkas lemma *the polynomial Farkas lemma*.

We follow the polynomial approach and prove the polynomial Farkas lemma in Imandra (`UNSAT` case only, cf. Theorem 5), obtaining its specialization to the case that refers to the parameters of the DNN verification problem, referred to as the *DNN Polynomial Farkas lemma* (cf. Theorem 6). In this shape, the lemma relates the existence of witnesses of a certain form to the unsatisfiability of a system of l.p.e. constructed from a DNN verification

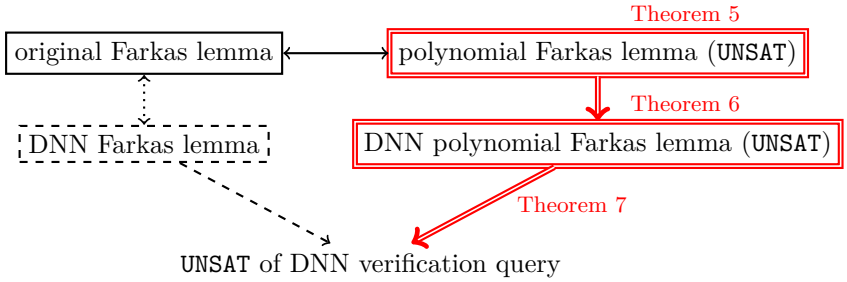



**Figure 2** Soundness and known relations between different versions of the Farkas lemma. Boxes represent lemmas that are proven sound. An arrow from $A$ to $B$ states that the soundness of $A$ implies the soundness of $B$. Dotted lines show conjectures; dashed lines show results that were proven manually; solid lines show those formalized in an ITP. Single black lines indicates previously known results, double red lines refer to results presented here. We indicate the lemmas that correspond to each result.

query. To ensure that the checker is sound, it remains to link this unsatisfiability to the unsatisfiability of the DNN verification query (Theorem 7). These results are shown in red double lines in Figure 2.

## 3.1 The DNN Farkas Lemma

To produce proofs of UNSAT for linear queries of the form $\langle A, \mathbf{u}, \mathbf{l}\rangle$, Isac et al. [27] prove a constructive variant of the Farkas lemma that defines proof witnesses of UNSAT:

▶ **Theorem 3** (DNN Farkas lemma [27]). *Let $A \in \mathbb{R}^{m\times n}$, $\mathbf{x}$ a vector of $n$ variables and $\mathbf{l}, \mathbf{u} \in \mathbb{R}^n$, such that $A \cdot \mathbf{x} = \mathbf{0}$ and $\mathbf{l} \leq \mathbf{x} \leq \mathbf{u}$. Then exactly one of these two options holds:*

1. *SAT: There exists a solution $\mathbf{s} \in \mathbb{R}^n$ such that $(A \cdot \mathbf{x} = \mathbf{0} \wedge \mathbf{l} \leq \mathbf{x} \leq \mathbf{u})[\mathbf{x}/\mathbf{s}]$ is true.*
2. *UNSAT: There exists a* contradiction vector *$w \in \mathbb{R}^m$ such that for all $\mathbf{x}$, if $\mathbf{l} \leq \mathbf{x} \leq \mathbf{u}$ we have $w^\intercal \cdot A \cdot \mathbf{x} < 0$. As $w \cdot \mathbf{0} = 0$, $w$ is a proof of the constraints' unsatisfiability.*

*Moreover, in the UNSAT case, the contradiction vector can be constructed during the execution of the Simplex algorithm.*

Note that, given a vector $s$ or $w$, a proof checker can immediately conclude the satisfiability of $\langle A, \mathbf{u}, \mathbf{l}\rangle$, while constructing $s$ or $w$ often requires the DNN verifier to apply more sophisticated procedures [27]. We will prove a polynomial version of this result in Imandra.

## 3.2 Polynomial Farkas Lemma

Given *coefficients* $\overline{\alpha} = \alpha_0, \ldots, \alpha_n \in \mathbb{R}^{n+1}$ and a vector of variables $\mathbf{x} = (x_1, ..., x_n)$, a *real linear polynomial* $p^{\overline{\alpha}}(\mathbf{x})$ of size $n$ is defined as follows: $p^{\overline{\alpha}}(\mathbf{x}) := \alpha_0 + \sum_{i=1}^{n} \alpha_i x_i$.

Given $\mathbf{x}$, one can form different polynomials $p^{\overline{\alpha}_1}(\mathbf{x}), \ldots, p^{\overline{\alpha}_m}(\mathbf{x})$. To simplify the notation, we write $p_1(\mathbf{x}), \ldots, p_m(\mathbf{x})$ to denote $m$ arbitrary polynomials when the concrete choice of $\overline{\alpha}_i$s is unimportant. We will use $q_1(\mathbf{x}), \ldots, q_m(\mathbf{x})$ to denote a potentially different list of polynomials. A *linear polynomial equation* is an equation of the form $p(\mathbf{x}) \bowtie 0$, where $\bowtie$ is either $=$ or $\geq$. The following formula defines a *system of linear polynomial equations* $\mathcal{S}(\mathbf{x}) := (\bigwedge_{i=1}^{K} (p_i(\mathbf{x}) = 0)) \wedge (\bigwedge_{j=1}^{M} (q_j(\mathbf{x}) \geq 0))$ where $K, M$ are arbitrary natural numbers. We say that $\mathcal{S}(\mathbf{x})$ is *satisfied by a vector* $\mathbf{s} \in \mathbb{R}^n$ if $\mathcal{S}(\mathbf{x})[\mathbf{x}/\mathbf{s}]$ is true. In Imandra, we use the

```
theorem farkas_unsat (s: system)
    (x: var_vect) (c: certificate) =
  well_formed s x && check_cert s c
  ==>
  eval_system s x = false
[@@by [%use cert_is_neg s c x]
   @> [%use solution_is_not_neg s c x]
   @> auto] [@@fc]
```

**Theorem 6.** (Polynomial Farkas Lemma (`UNSAT` case)). **If** $\exists \mathbb{I}, \mathbb{C}. \mathbb{I}(\mathbf{x}) + \mathbb{C}(\mathbf{x}) < 0$ **then** $\neg(\exists \mathbf{s}. \texttt{eval\_system}(S(\mathbf{x})), \mathbf{s})$

**Figure 3** Left: Imandra code for the Polynomial Farkas lemma. The keywords `use` and `auto` are theorem proving tactics of Imandra; the `fc` annotation instructs Imandra to automatically make use of the lemma as a *forward chaining* rule. Note the explicit use of lemmas `cert_is_neg` and `solution_is_not_neg` in the tactic script. Right: pseudocode formulation of the same theorem. In pseudocode, we assume all elements are well-formed, i.e. the system of l.p.e. is valid w.r.t. $\mathbf{x}$. In Imandra, we assert this with the predicate `well_formed`, and the function `check_cert` checks whether the provided coefficients build witnesses $\mathbb{I}, \mathbb{C}$ s.t. $\mathbb{I} + \mathbb{C} < 0$. Then `eval_system` checks if the assignment $\mathbf{x}$ is a solution to the system of l.p.e.

predicate $\texttt{eval\_system}(\mathcal{S}(\mathbf{x}), \mathbf{s})$, which returns *true* when the vector $\mathbf{s}$ is a solution to $\mathcal{S}(\mathbf{x})$. We say the system of l.p.e. $\mathcal{S}(\mathbf{x})$ *has a solution*, denoted `SAT`, if there exists an $\mathbf{s}$ such that $\mathcal{S}(\mathbf{x})[\mathbf{x}/\mathbf{s}]$ is true. Otherwise, $\mathcal{S}(\mathbf{x})$ is `UNSAT`.

**Operations on Polynomials.** A polynomial $p^{\overline{\alpha}}(\mathbf{x})$ with coefficients $\overline{\alpha} = \alpha_0, \ldots, \alpha_n$ *scaled by a real constant* $c$ is defined as the polynomial over $\mathbf{x}$ with coefficients $c\alpha_0, \ldots, c\alpha_n$, and noted $c \cdot p(\mathbf{x})$. The *addition of two polynomials* $p^{\overline{\alpha}}(\mathbf{x})$ and $p^{\overline{\beta}}(\mathbf{x})$ with coefficients $\overline{\alpha} = \alpha_0, \ldots, \alpha_n$ and $\overline{\beta} = \beta_0, \ldots, \beta_n$ respectively, denoted as $p^{\overline{\alpha}} + p^{\overline{\beta}}(\mathbf{x})$, is a polynomial with coefficients $\alpha_0 + \beta_0, \ldots, \alpha_n + \beta_n$. The *linear combination of polynomials* $p_1(\mathbf{x}), \ldots, p_N(\mathbf{x})$ *with coefficient vector* $\mathbf{c} = c_1, \ldots, c_N \in \mathbb{R}^N$ is defined as the sum of polynomials $p_1(\mathbf{x}), \ldots, p_N(\mathbf{x})$ scaled by the coefficients in $\mathbf{c}$: $\sum_{i=1}^{N} c_i p_i(\mathbf{x})$.

We define the conversion function from the DNN verification query form to linear polynomial equations and inequalities.

▶ **Definition 4.** *Tableau-to-polynomial conversion. Given a DNN verification query* $\langle A, \mathbf{u}, \mathbf{l} \rangle$ *and variable vector* $\mathbf{x}$*, the conversion function* $\mathcal{S}^{A,\mathbf{u},\mathbf{l}}$ *is defined as follows:*

$$\mathcal{S}^{A,\mathbf{u},\mathbf{l}}(\mathbf{x}) := \bigwedge_{i=0}^{m} p^{\alpha(i)}(\mathbf{x}) = 0 \wedge \bigwedge_{j=0}^{n} q^{\beta(\mathbf{u},j)}(\mathbf{x}) \geq 0 \wedge \bigwedge_{k=0}^{n} q^{\gamma(\mathbf{l},k)}(\mathbf{x}) \geq 0, \text{ where} \tag{5}$$

- $p^{\alpha(i)}$ *is a polynomial whose coefficients are the values in the* $i^{th}$ *row of the matrix* $A$ *(and constant element is* $0$*):* $p^{\alpha(i)}(\mathbf{x}) := p^{0,a_{i,1},\ldots,a_{i,n}}(\mathbf{x})$*. It encodes the linear polynomial equations as defined in equation (2).*
- *For all* $j \in [0, n]$*,* $q^{\beta(\mathbf{u},j)}(\mathbf{x}) := \mathbf{u}_j - x_j$
- *For all* $k \in [0, n]$*,* $q^{\gamma(\mathbf{l},k)}(\mathbf{x}) := x_k - \mathbf{l}_k$

*Note that the coefficients for polynomials* $q^{\beta(\mathbf{u},j)}$ *and* $q^{\gamma(\mathbf{l},k)}(\mathbf{x})$ *are sparse, with* $0$ *values at all indices except for the constant and at the index of* $x_i$*.*

▶ **Theorem 5** (Polynomial Farkas Lemma)**.** *A system of l.p.e.* $\mathcal{S}(\mathbf{x}) := \bigwedge_{i=1}^{K} (p_i(\mathbf{x}) = 0) \wedge \bigwedge_{j=1}^{M} (q_j(\mathbf{x}) \geq 0)$ *has a solution iff there does not exist:* ***1.*** *a linear combination* $\mathbb{I}(\mathbf{x})$ *of* $p_1(\mathbf{x}), \ldots, p_K(\mathbf{x})$ *and* ***2.*** *a linear combination* $\mathbb{C}(\mathbf{x})$ *of* $q_1(\mathbf{x}), \ldots, q_M(\mathbf{x})$ *with non-negative coefficients, such that* $\mathbb{I}(\mathbf{x}) + \mathbb{C}(\mathbf{x}) < 0$*. We call* $\mathbb{I}, \mathbb{C}$ witnesses *of* `UNSAT` *for* $\mathcal{S}(\mathbf{x})$*.*

```
Goal:
((not
 ((eval_system es x) && (well_formed es x)
    && es <> [])
  ==>
  (eval_poly (mk_cert_poly cs es) x) >=. 0.0)
 || (not
 ((well_formed es x) && (check_cert es cs))
  ==>
  (eval_poly (mk_cert_poly cs es) x) <. 0.0)
 || ((well_formed es x) && (check_cert es cs)
     && es <> [])
   ==>
  (eval_system es x) = false)
.
Enter waterfall with goal:
fun (cs : real list) (es : expr list)
    (x : real list)
-> ((not
 ((eval_system es x) && (well_formed es x)
     &&  es <> [])
  ==>
  (eval_poly (mk_cert_poly cs es) x) >=. 0.0)
 || (not
 ((well_formed es x) && (check_cert es cs))
  ==>
  (eval_poly (mk_cert_poly cs es) x) <. 0.0)
 || ((well_formed es x) && (check_cert es cs)
     && es <> [])
  ==>
  (eval_system es x) = false)
      1 nontautological subgoal.
Subgoal 1:
H0.((eval_system es x) && (well_formed es x) &&
     es <> [])
    ==>
    (eval_poly (mk_cert_poly cs es) x) >=. 0.0
H1.((well_formed es x) && (check_cert es cs))
    ==>
    (eval_poly (mk_cert_poly cs es) x) <. 0.0
|-----------------------------------------
 ((well_formed es x) && (check_cert es cs) &&
    es <> [])
 ==>
 (eval_system es x) = false

  But simplification reduces this to true,
     using the forward-chaining rules
     eval_const_neg and scale_empty_invariant
     and times_neg and times_neg_2 and times_pd
      and times_psd
```

**Figure 4** Proof generated by Imandra for Theorem 5 (Figure 3). It is resolved by simplification and using auxiliary lemmas. Although the application of lemmas `cert_is_neg` and `solution_is_not_neg` (see Table 1) does not appear explicitly, the auxiliary lemmas shown in the proof trace stem from it.

**Proof.** We prove that witnesses $\mathbb{I}(\mathbf{x})$, $\mathbb{C}(\mathbf{x})$ and a solution cannot exist simultaneously.

1. Assume there exists $c = (c_1, \ldots, c_K) \in \mathbb{R}^K$ and $d = (d_1, \ldots, d_M) \in \mathbb{R}^M$, where $d_j \geq 0$ for all $d_j \in d$, such that $\mathbb{I}(\mathbf{x}) + \mathbb{C}(\mathbf{x}) = \sum_{i=1}^{K} c_i \cdot p_i(\mathbf{x}) + \sum_{j=1}^{M} d_j \cdot q_j(\mathbf{x}) < 0$.
2. Furthermore, assume that the system of l.p.e. $\mathcal{S}(\mathbf{x})$ has a solution $\mathbf{s}$. Then, for $\mathcal{S}(\mathbf{x})[\mathbf{x}/\mathbf{s}]$ and for all $i \in [1, K], p_i(\mathbf{s}) = 0$, so we have $\sum_{i=1}^{K} c_i \cdot p_i(\mathbf{s}) = 0$. Similarly, it implies that for all $j \in [1, M]$, $q_j(\mathbf{s}) \geq 0$, so we have $\sum_{j=1}^{M} d_j \cdot q_j(\mathbf{s}) \geq 0$. This means that $\sum_{i=1}^{K} c_i \cdot p_i(\mathbf{s}) + \sum_{j=1}^{M} d_j \cdot q_j(\mathbf{s}) \geq 0$.

   Since we postulated that this sum is negative, this leads to a contradiction. ◀

The Imandra code covering one direction of Theorem 5 is given in Figure 3. It is the only direction needed to prove soundness of our implementation, and corresponds to the top double box in Figure 2. In Imandra, we define a valid certificate as a linear combination of the tableau rows which gives a polynomial with all coefficients equal to 0 and a negative constant. The proof then proceeds by applying two lemmas: `cert_is_neg` ("a valid certificate always evaluates to a negative value") and `solution_is_not_neg` ("evaluating a certificate with a solution to the system evaluates to a non-negative value"). Table 1 shows formal statements of these lemmas. The tactic `auto` resolves the contradiction in the theorem's conclusion. We require that the system and variable vectors have matching dimensions, and use the predicate `well_formed` to assert that. Figure 4 gives the interested reader a glimpse of the completed proof produced by Imandra in response to user prompt given in Figure 3.

Generally, the user interacts with Imandra by supplying a set of automation tactics and possibly providing missing auxiliary lemmas. The keyword `waterfall` refers to a famous Boyer-Moore inductive proof automation approach [12] that is deployed in ACL2 and Imandra [37].

**Table 1** Summary of main lemmas used in proofs of Farkas lemma and Soundness. The notation in this table follows the one given in Theorem 3 and Definition 4: $A$ is a generic tableau in $\mathbb{R}^{n\times m}$; $\mathbf{x}, \mathbf{l}, \mathbf{u}$ are vectors in $\mathbb{R}^n$; $\mathcal{S}^{A,\mathbf{u},\mathbf{l}}$ is the corresponding system of l.p.e; $\mathbf{l}^L, \mathbf{u}^L, \mathbf{l}^R, \mathbf{u}^R$ are updated bounds computed by `update_bounds`.

| Sec. | Lemma Name | Mathematical Statement |
|---|---|---|
| § 3 | `cert_is_neg` | If $\mathbb{I}(\mathbf{x}) + \mathbb{C}(\mathbf{x})$ is a polynomial with 0 coefficients and a negative constant, then $\forall \mathbf{s}. \mathbb{I}(\mathbf{x}) + \mathbb{C}(\mathbf{x})[\mathbf{x}/\mathbf{s}] < 0$ |
| | `solution_is_not_neg` | $\forall \mathbf{s}$ s.t. $\mathbf{s}$ is a solution to $S$. $\mathbb{I}(\mathbf{x}) + \mathbb{C}(\mathbf{x})[\mathbf{x}/\mathbf{s}] \geq 0$ |
| | `tableau_reduction` | If $\exists \mathbf{s}$ s.t. $A.\mathbf{s} = \vec{0}$, then $\mathbf{s}$ satisfies the equalities in $\mathcal{S}^{A,\mathbf{u},\mathbf{l}}(\mathbf{x})$ |
| | `bound_reduction` | If $\exists \mathbf{s}$ bounded by $\mathbf{l}$ and $\mathbf{u}$, then $\mathbf{s}$ satisfies the inequalities in $\mathcal{S}^{A,\mathbf{u},\mathbf{l}}(\mathbf{x})$ |
| § 4 | `relu_split` | $\forall x_f, x_b, x_{aux} \in \mathbb{R}$ such that $x_f = \text{ReLU}(x_b)$ and $x_{aux} = x_f - x_b$, either $(x_b \geq 0 \wedge x_{aux} = 0)$ or $(x_b < 0 \wedge x_f = 0)$ hold |
| | `get_set_nth` | $\forall \mathbf{l} \in \mathbb{R}^n, k \in \mathbb{R}, i \in [0, n-1]. \mathbf{l}[\mathbf{l}_i/k]_i = k$ |
| | `ith_set_nth` | $\forall \mathbf{l} \in \mathbb{R}^n, k \in \mathbb{R}, i, j \in [0, n-1]$ s.t. $i \neq j. \mathbf{l}[\mathbf{l}_i/k]_j = \mathbf{l}_j$ |
| § 5 | `well_formed_preservation` | If $\langle A, \mathbf{u}, \mathbf{l}\rangle$ well-formed w.r.t. variable vector $\mathbf{x}$, then $\langle A, \mathbf{l}^L, \mathbf{u}^L\rangle$ and $\langle A, \mathbf{l}^R, \mathbf{u}^R\rangle$ are also well-formed w.r.t. $\mathbf{x}$. |

It dynamically composes tactics that generate induction schemes, perform simplification, rewriting, generalization and forward-chaining reasoning during proof search. Together with proofs of auxiliary lemmas the proof of Theorem 5 is nearly 200 lines long.

## 3.3 Farkas Lemma for DNN Proof Checking

Marabou produces proofs of `UNSAT` in the form given in Theorem 3, but the Imandra implementation uses the formulation of the Farkas lemma given in Theorem 5. We need to prove that Theorem 5 specializes to systems of l.p.e that encode DNN verification queries:

▶ **Theorem 6** (DNN Polynomial Farkas Lemma). $\forall A, \mathbf{l}, \mathbf{u}, C$ ***If*** $\exists \mathbb{I}, \mathbb{C}. \mathbb{I}(\mathbf{x}) + \mathbb{C}(\mathbf{x}) < 0$ ***then*** $\neg(\exists \mathbf{s}. \mathit{eval\_system}(S^{A,\mathbf{l},\mathbf{u}}(\mathbf{x}), \mathbf{s}))$.

To deduce the `UNSAT` of a DNN verification query from the `UNSAT` of its equivalent system of l.p.e., we prove the following Lemma (represented by the lower red double arrow in Figure 2).

▶ **Theorem 7** (Sound Application of DNN Polynomial Farkas Lemma). ***If*** $\neg(\exists \mathbf{s}. \mathit{eval\_system}(\mathcal{S}^{A,\mathbf{u},\mathbf{l}}(\mathbf{x})), \mathbf{s})$ ***then*** $\neg(\exists \mathbf{s}. \mathit{is\_solution}(\langle A, \mathbf{u}, \mathbf{l}\rangle, \mathbf{s})$.

It is straightforward to see that, by Definition 4, polynomials in $\mathcal{S}^{A,\mathbf{u},\mathbf{l}}$ have a one-to-one correspondence to constraints in $\langle A, \mathbf{u}, \mathbf{l}\rangle$. However proving this in Imandra was not trivial. We consider the equations from the tableau rows and the inequalities from the bounds separately. For the equations, `tableau_reduction` is proven by Imandra with minimal guidance by induction on the tableau length. On the other hand, `bound_reduction` necessitated linking index-based computations (the bound polynomial for the $i^{th}$ bound is constructed as a polynomial with a single non-zero coefficient at index $i$) and recursion-based computation (e.g. for checking whether a vector is bounded by $\mathbf{l}$ and $\mathbf{u}$).

Assembling Theorems 5 and 7, we get that a witness of `UNSAT` for a system of l.p.e. constructed from a DNN verification query is sufficient to guarantee that the query is `UNSAT`.

# 4 A DNN Certificate Checker in Imandra

Recall that Marabou search induces tree-like structures. The proof production in [27] constructs a *proof tree* based on the search trace of `UNSAT` queries, serving as a witness. We now introduce an alternative algorithm that checks these proof trees and certifies that the

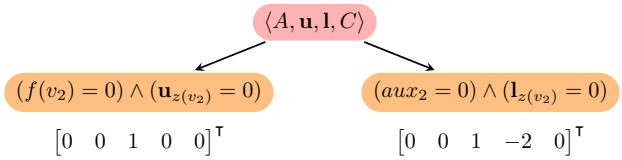
```
type proofTree =
  | Leaf of real list
  | Node of Split * proofTree * proofTree

type Split = Relu of int * int * int
  | SingleVar of int * real
```

**Figure 5** Marabou proof tree as an inductive data type in Imandra.

corresponding DNN verification query is `UNSAT`. If an error is detected, it can identify the specific parts of the proof that contributed to the failure. These correspond to search states where Marabou failed to produce a correct proof.

▶ **Example 8** (Marabou Proof Tree Construction)**.** Below is a graphical representation of a proof tree proving that the query presented in Example 1 is `UNSAT` and witnessing that during execution, Marabou performed a single split over the ReLU constraint derived from $v_2$. The proof contains the tableau $A$, the bound vectors $\mathbf{u}, \mathbf{l}$, and the ReLU constraints $C$:




Each leaf contains a contradiction vector for their respective subquery (cf. Theorem 3).

A proof tree `t`, represented by an inductive type `proofTree`, can be either a leaf `Leaf w` or a non-leaf node `Node s tl tr`. A leaf `Leaf w` corresponds to an `UNSAT` result of the linear part of a subquery; leaves hence always contain a contradiction vector `w : real list`. A non-leaf node `Node s tl tr` corresponds to a split `s: Split` and corresponding sub-proof-trees `tl` and `tr`. We consider two kinds of splits: `s: Split` can be either a case split over a ReLU constraint, `Relu b f aux`, where `b`, `f`, and `aux` are the participating variables in the corresponding ReLU constraint (i.e. the DNN verification query includes $\mathbf{x}_f = \text{ReLU}(\mathbf{x}_b) \wedge \mathbf{x}_f - \mathbf{x}_b - \mathbf{x}_{aux} = 0$), or a single-variable split `SingleVar xi k`, performed on a variable $\mathbf{x}_i$ with a split on value $k$.

This type guarantees some structural properties but not the full certification of the proof trees. For example, for `Leaf w`, `w` has to be checked to be a contradiction vector for the corresponding subquery to construct a witness of `UNSAT` (Theorem 5). During the parsing of the Marabou proof, we hence check that each ReLU split corresponds to one of the constraints in the original DNN verification query via the recursive `check_tree` algorithm (Algorithm 1).

Marabou uses splits to compute tighter upper and lower bound vectors, later used to prove `UNSAT` in the proof tree's leaves. The function `update_bounds u l s` performs this bound tightening and yields updated lower and upper bounds for both subtrees:

```
let update_bounds (lbs: real list) (ubs: real list) (split: Split):
        ((real list * real list) * (real list * real list)) =
    match split with
    | SingleSplit (i,k) -> ((lbs,set_nth ubs i k), (set_nth lbs i k,ubs))
    | ReluSplit (b, f, aux) ->
        let lbs_l = set_nth lbs f 0. in
        let ubs_l = set_nth (set_nth ubs b 0.) f 0. in
        let lbs_r = set_nth (set_nth lbs b 0.) aux 0. in
        let ubs_r = set_nth ubs aux 0. in
        (lbs_l, ubs_l), (lbs_r, ubs_r)
```

In the case of a split on a single variable $i$ with value $k$, the updated bounds are $\mathbf{l}, \mathbf{u}[\mathbf{u}_i/k]$ and $\mathbf{l}[\mathbf{l}_i/k], \mathbf{u}$. In the case of a split on a ReLU constraint of the form $f = \text{ReLU}(b) \wedge aux = f - b$, the updated bounds are $\mathbf{l}[\mathbf{l}_b/0, \mathbf{l}_{aux}/0]$ and $\mathbf{u}[\mathbf{u}_{aux}/0]$ for one child and $\mathbf{l}[\mathbf{l}_f = 0]$ and

**Algorithm 1** Proof tree checking algorithm (`check_tree`).

```
1: Inputs: A, u, l, C, T
2: Outputs: {true, false}
3: if T is a leaf, with list of reals w then
4:     I, C := mk_certificate(A, u, l, w)
5:     if I(x) + C(x) < 0 then
6:         return true
7:     else
8:         return false
9: if T is a non-leaf node, containing Split, T^L, T^R then
10:    ⟨l^L, u^L, l^R, u^R⟩ := update_bounds(u, l, Split)
11:    if check_tree(A, l^L, u^L, C, T^L)
         ∧ check_tree(A, l^R, u^R, C, T^R) then
12:        return true
13:    else
14:        return false
```

$\mathbf{u}[\mathbf{u}_b/0, \mathbf{u}_f/0]$ for the other. This corresponds to the case analysis of lemma `relu_split` (see Table 1). We call the first case *inactive phase* and the second case *active phase* of the ReLU constraint. Note that the order of the phases is fixed during the tree construction; the fixed order is followed in the bound updating implementation (see lines 4, 12).

The function `check_tree`$(A, \mathbf{u}, \mathbf{l}, C, \mathtt{t})$, traverses a proof tree `t` and given a DNN verification query $\langle A, \mathbf{u}, \mathbf{l}, C\rangle$ ensures that for all the leaves it is able to build witnesses of `UNSAT` in the sense of Theorem 6. It is given in pseudocode (Algorithm 1) and Imandra code (Figure 6). For a leaf node containing a list of reals $w$, the procedure `mk_certificate` computes witness candidates $\mathbb{I}(\mathbf{x}), \mathbb{C}(\mathbf{x})$ as in Theorem 6, then the algorithm checks whether $\mathbb{I}(\mathbf{x}) + \mathbb{C}(\mathbf{x}) < 0$ (`mk_certificate` in Figure 6, l. 6). If this check passes, by Theorem 5, $\mathbb{I}(\mathbf{x}), \mathbb{C}(\mathbf{x})$ are witnesses of `UNSAT` for the system of l.p.e. $\mathcal{S}^{A,\mathbf{u},\mathbf{l}}(\mathbf{x})$. By Theorem 7, $\langle A, \mathbf{u}, \mathbf{l}\rangle$ is `UNSAT` as well. For a non-leaf node with data $Split, \mathcal{T}^L, \mathcal{T}^R$:

- the procedure `update_bounds` computes updated versions of $\mathbf{u}, \mathbf{l}$ corresponding to the two phases of the split as described in the previous section (`update_bounds`, lines 10-11).
- recursive calls to `check_tree`, using the new bounds that correspond to each sub-tree, check the sub-trees rooted in $\mathcal{T}^L$ and $\mathcal{T}^R$ (lines 13-14).

A tree $\mathcal{T}$ is *certified* if all checks pass. The function `check_tree`$(A, \mathbf{u}, \mathbf{l}, C, \mathcal{T})$ implements Algorithm 1 almost verbatim, with the notable difference in the use of the `check_split` function (Figure 6, line 9). It re-iterates the check that ReLU splits indeed correspond to a known ReLU constraint of the verification query.

The following example shows the execution of this algorithm.

▶ **Example 9** (Checking the Proof Tree of Example 8)**.** To certify the proof tree, the algorithm begins by checking the root node by certifying that the bound vectors of the two children correspond to the splits $\mathbf{u}_{f(v_2)} = 0 \wedge \mathbf{l}_{f(v_2)} = 0 \wedge \mathbf{u}_{z(v_2)} = 0$ and $\mathbf{u}_{aux_2} = 0 \wedge \mathbf{l}_{aux_2} = 0 \wedge \mathbf{l}_{z(v_2)} = 0$, i.e., to the two splits of the constraint $x_{f(v_2)} = \mathrm{ReLU}(x_{z(v_2)})$. Then, it recursively checks the leaves. It starts by updating $\mathbf{u}_{z(v_2)} = 0$, $\mathbf{l}_{f(v_2)} = 0$, and $\mathbf{u}_{f(v_2)} = 0$ and certifying the contradiction vector of the leaf: it constructs the system of l.p.e. with equations representing the tableau $A$: $2x_1 + x_2 - x_{z(v_1)} = 0$, $x_2 - x_1 - x_{z(v_2)} = 0$, $2x_{f(v_2)} - x_{f(v_1)} - y = 0$, $x_{f(v_1)} - x_{z(v_1)} - x_{aux_1} = 0$, $x_{f(v_2)} - x_{z(v_2)} - x_{aux_2} = 0$, and the inequalities representing the bounds $\mathbf{l}, \mathbf{u}$. For simplicity, we consider only the inequalities $0 - x_{f(v_2)} \geq 0, x_{f(v_1)} - 0 \geq 0$ and $y - 4 \geq 0$. Then, it constructs the certificate $\mathbb{I} = \begin{bmatrix}0 & 0 & 1 & 0 & 0\end{bmatrix}^\intercal \cdot A \cdot x = 2x_{f(v_2)} - x_{f(v_1)} - y = 0$, $\mathbb{C} = 2(-x_{f(v_1)}) + x_{f(v_1)} + y - 4$. Lastly, it checks that indeed $\mathbb{I} + \mathbb{C} = 2x_{f(v_2)} - x_{f(v_1)} - y - 2x_{f(v_1)} + x_{f(v_1)} + y - 4 = -4 < 0$.

```
let rec check_node (tableau: expr list) (upper_bounds: real list)
  (lower_bounds: real list) (constraints: Constraint.t list)
  (proof_node: Proof_tree.t): bool =
    match proof_node with
    | Proof_tree.Leaf (contradiction, bound_lemmas) ->
        check_contradiction contradiction tableau upper_bounds
        lower_bounds
    | Proof_tree.Node (split, bound_lemmas, left, right) ->
        let valid_split = check_split split constraints in
        let (lb_left, ub_left), (lb_right, ub_right) = update_bounds
            lower_bounds upper_bounds split in
        let valid_children =
            (check_node tableau ub_left lb_left constraints left) &&
            (check_node tableau ub_right lb_right constraints right) in
        valid_split && valid_children
```

**Figure 6** Imandra implementation of `check_tree`.

The right leaf is checked similarly. Since all nodes pass the checks, the algorithm returns *true*, and the certification of this proof tree is complete.

## 5 Proof of Soundness

This section presents the proof of soundness of the algorithm introduced in the previous section: given a DNN verification query $\langle A, \mathbf{u}, \mathbf{l}, C\rangle$ and the corresponding tree witness of UNSAT $\mathcal{T}$, if `check_tree`$(A, \mathbf{u}, \mathbf{l}, C, \mathcal{T})$ returns *true*, then there should exist no satisfying assignment for $\langle A, \mathbf{u}, \mathbf{l}, C\rangle$.

The proof follows a custom induction scheme on the structure of `check_tree`. We will first prove the base case, when the proof tree is a leaf (Lemma 10), then the induction step, when it is a node with children (Lemma 13). The proof for leaves is straightforward thanks to the Farkas lemma. The case for non-leaf nodes is trickier, as it involves proving that splits fully cover the DNN verification query solution space. This can be done by proving that both single variable splits and ReLU splits are covering (Lemmas 12 and 13 respectively).

We first prove that `check_tree` is correct for proof trees leaves:

▶ **Lemma 10** (Leaf checking). ***If*** $\mathcal{T}$ *is a leaf* ***and*** `check_tree`$(A, \mathbf{u}, \mathbf{l}, C, \mathcal{T})$ *returns true* ***then*** $\neg(\exists \mathbf{s}.$ `eval_system`$(\mathcal{S}^{A,\mathbf{u},\mathbf{l}}(\mathbf{x}), \mathbf{s}))$.

**Proof.** By the definition of `check_tree` and Theorems 5 and 7. ◀

For the inductive case, we first prove that tightening bounds according to splits is covering. We state the definition of boundedness as a conjunction instead of a universally quantified index to avoid instantiating an index and to allow better automation in Imandra.

▶ **Definition 11** (Bound vectors). *Let* $\mathbf{u}, \mathbf{l}, \mathbf{x} \in \mathbb{R}^n$. *We say that* $\mathbf{x}$ *is bounded by* $\mathbf{l}$ *and* $\mathbf{u}$ *(and we write* $\mathbf{l} \leq \mathbf{x} \leq \mathbf{u}$*) if* $\bigwedge_{i=0}^{n-1} \mathbf{l}_i \leq \mathbf{x}_i \leq \mathbf{u}_i$.

For single variable splits, the following lemma is proven (we omit the proof as it is simple):

▶ **Lemma 12** (Single variable splits are covering). *Let* $\mathbf{u}, \mathbf{l}, \mathbf{x} \in \mathbb{R}^n$*;* $i \in [0, n-1]$*;* $k \in \mathbb{R}$*;* $\mathbf{l}^L := \mathbf{l}$, $\mathbf{u}^L := \mathbf{u}[\mathbf{u}_i/k]$, $\mathbf{l}^R := \mathbf{l}[\mathbf{l}_i/k]$, $\mathbf{u}^R := \mathbf{u}$.
***If*** $\mathbf{l} \leq \mathbf{x} \leq \mathbf{u}$ ***then*** $\mathbf{l}^L \leq \mathbf{x} \leq \mathbf{u}^L \vee \mathbf{l}^R \leq \mathbf{x} \leq \mathbf{u}^R$.

For ReLU splits, recall that for each split with indices $b, f, aux$, the DNN verification query includes $\mathbf{x}_f = \text{ReLU}(\mathbf{x}_b) \wedge \mathbf{x}_f - \mathbf{x}_b - \mathbf{x}_{aux} = 0$.

▶ **Lemma 13** (ReLU splits are covering). *Let* $\mathbf{u}, \mathbf{l}, \mathbf{x} \in \mathbb{R}^n$*;* $f, b, aux \in [0, n-1]$ *such that* $f \neq b, f \neq aux, b \neq aux$*;* $\mathbf{l}^L := \mathbf{l}[\mathbf{l}_f/0]$*,* $\mathbf{u}^L := \mathbf{u}[\mathbf{u}_b/0, \mathbf{u}_f/0]$*,* $\mathbf{l}^R := \mathbf{l}[\mathbf{l}_{aux}/0]$*,* $\mathbf{u}^R := \mathbf{u}[\mathbf{u}_b/0, \mathbf{u}_{aux}/0]$.

***If*** $\mathbf{l} \leq \mathbf{x} \leq \mathbf{u} \wedge \mathbf{x}_f = ReLU(\mathbf{x}_b) \wedge \mathbf{x}_f - \mathbf{x}_b - \mathbf{x}_{aux} = 0$ ***then*** $\mathbf{l}^L \leq \mathbf{x} \leq \mathbf{u}^L \vee \mathbf{l}^R \leq \mathbf{x} \leq \mathbf{u}^R$.

**Proof.** Assuming that $\mathbf{l} \leq \mathbf{x} \leq \mathbf{u} \wedge \mathbf{x}_f = \mathrm{ReLU}(\mathbf{x}_b) \wedge \mathbf{x}_f - \mathbf{x}_b - \mathbf{x}_{aux} = 0$ holds, we consider the cases given by `relu_split`, see Table 1 for statements of auxiliary lemmas mentioned below:

1. Case 1: $\mathbf{x}_b < 0 \wedge \mathbf{x}_f = 0$. We prove that for all indices $i \in [0, n-1]$, $\mathbf{l}[\mathbf{l}_f/0]_i \leq \mathbf{x}_i \leq \mathbf{u}[\mathbf{u}_b/0, \mathbf{u}_f/0]_i$, by considering cases according to the value of $i$.
   a. if $i = b$:
   $$\mathbf{l}[\mathbf{l}_f/0]_i \overset{f\neq b, \texttt{ith\_set\_nth}}{=} \mathbf{l}_i \overset{ass., Def.11}{\leq} \mathbf{x}_i \overset{i=b}{=} \mathbf{x}_b \overset{ass., \texttt{leq\_lt}}{\leq} 0 \overset{i=b, \texttt{get\_set\_nth}}{=} \mathbf{u}[\mathbf{u}_b/0, \mathbf{u}_f/0]_i$$
   b. if $i = f$:
   $$\mathbf{l}[\mathbf{l}_f/0]_i \overset{i=f, \texttt{get\_set\_nth}}{=} 0 \overset{ass.}{\leq} \mathbf{x}_i \overset{i=f}{=} \mathbf{x}_f \overset{ass.}{\leq} 0 \overset{i=f, \texttt{get\_set\_nth}}{=} \mathbf{u}[\mathbf{u}_b/0, \mathbf{u}_f/0]_i$$
   c. if $i \neq b \wedge i \neq f$:
   $$\mathbf{l}[\mathbf{l}_f/0]_i \overset{i\neq f, \texttt{ith\_set\_nth}}{=} \mathbf{l}_i \overset{ass., Def.11}{\leq} \mathbf{x}_i \overset{ass., Def.11}{\leq} \mathbf{u}_i \overset{i\neq f \wedge i \neq b, \texttt{ith\_set\_nth}}{=} \mathbf{u}[\mathbf{u}_b/0, \mathbf{u}_f/0]_i$$
2. Case 2: $\mathbf{x}_b \geq 0 \wedge \mathbf{x}_{aux} = 0$. We prove that for all indices $j \in [0, n-1]$, $\mathbf{l}[\mathbf{l}_b/0, \mathbf{l}_{aux}/0]_j \leq \mathbf{x}_j \leq \mathbf{u}[\mathbf{u}_{aux}/0]_i$, we do a similar case analysis on the value of $j$. The proof proceeds similarly to Case 1. ◀

Lemma 14 states that if the query corresponding to a parent node in the proof tree has a satisfying assignment, then this assignment will satisfy one of the child node's queries.

▶ **Lemma 14** (Splits are covering). *Let* $\mathbf{u}, \mathbf{l}, \mathbf{x} \in \mathbb{R}^n$*; Split a split,* $\mathbf{l}^L, \mathbf{u}^L$*,* $\mathbf{l}^R, \mathbf{u}^R$ *are the bounds computed by* `update_bounds`$(\mathbf{u}, \mathbf{l}, Split)$.

***If*** *there exists* $\mathbf{s}$ *that satisfies* $\langle A, \mathbf{u}, \mathbf{l}, C\rangle$ ***then*** $\mathbf{s}$ *satisfies* $\langle A, \mathbf{u}^L, \mathbf{l}^L, C\rangle$ ***or*** $\mathbf{s}$ *satisfies* $\langle A, \mathbf{u}^R, \mathbf{l}^R, C\rangle$.

**Proof.** Since the bounds are the only parts of the children's queries that differ from their parent's query, proving Lemma 14 only requires to prove that the updated bounds are covering. This follows from Lemmas 12 and 13. ◀

Lemmas 10 and 14 now allow us to prove the overall soundness of `check_tree`.

▶ **Theorem 15** (Algorithm 1 is sound). ***If*** `check_tree`$(A, \mathbf{u}, \mathbf{l}, C, \mathcal{T})$ ***returns true, then*** $\neg(\exists \mathbf{s}.$ `is_solution`$(\langle A, \mathbf{l}, \mathbf{u}, C\rangle, \mathbf{s}))$.

**Proof.** We proceed by a functional induction scheme based on `check_tree`'s definition.

**Base case.** $\mathcal{T}$ is a leaf. By Lemma 10.

**Induction step.** Let *Split* be a split, $\mathcal{T}^L$ and $\mathcal{T}^R$ be two proof trees and $\mathbf{u}^L, \mathbf{l}^L, \mathbf{u}^R, \mathbf{l}^R$ be the bounds obtained from `update_bounds`$(\mathbf{l}, \mathbf{u}, Split)$. Our induction hypothesis states that `check_tree` is sound for `check_tree`$(A, \mathbf{l}^L, \mathbf{u}^L, C, \mathcal{T}^L)$ and `check_tree`$(A, \mathbf{l}^R, \mathbf{u}^R, C, \mathcal{T}^R)$. We now need to prove that `check_tree` is sound for `check_tree`$(A, \mathbf{l}, \mathbf{u}, C, \mathcal{T})$, where $\mathcal{T}$ is the proof tree with children $\mathcal{T}^L$ and $\mathcal{T}^R$ and split *Split*.

Assuming that `check_tree`$(A, \mathbf{l}, \mathbf{u}, C, \mathcal{T})$ holds, by definition `check_tree`$(A, \mathbf{l}^L, \mathbf{u}^L, C, \mathcal{T}^L)$ and `check_tree`$(A, \mathbf{l}^R, \mathbf{u}^R, C, \mathcal{T}^R)$ also hold. By the induction hypothesis, this means that $\neg(\exists$`s.is_solution`$(\langle A, \mathbf{l}^L, \mathbf{u}^L, C\rangle, \mathbf{s}))$ and $\neg(\exists$`s.is_solution`$(\langle A, \mathbf{l}^R, \mathbf{u}^R, C\rangle, \mathbf{s}))$

By Lemma 14, we conclude that $\neg(\exists$`s.is_solution`$(\langle A, \mathbf{l}, \mathbf{u}, C\rangle, \mathbf{s}))$. ◀

```
theorem check_node_soundness (tableau: real list list) (upper_bounds: real list)
    (lower_bounds: real list) (constraints: Constraint.t list) (tree: Proof_tree.t)
    (x: real list) =
    valid_proof tableau upper_bounds lower_bounds constraints tree
    && well_formed_vector tableau x
    ==> unsat tableau upper_bounds lower_bounds constraints x
[@@by [%expand "valid_proof"]
   @> [%expand "well_formed_vector"]
   @> induction ()
      @>>| [%use check_node_soundness_full tableau upper_bounds lower_bounds constraints tree x]
           @> [%use check_node_parent_imply_check_node_children (mk_eq_constraints tableau)
                 upper_bounds lower_bounds constraints tree]
           @> [%use well_formed_preservation tableau upper_bounds lower_bounds (split_of_node tree)]
           @> auto]
[@@disable List.length, well_formed_tableau_bounds, check_node, mk_eq_constraints, unsat, set_nth,
     update_bounds_from_split]
```

**Figure 7** Proof of Theorem 10 in Imandra: `valid_proof` calls `check_tree`, in addition to some structural checks (e.g. on the tableau dimensions).

Note that the proof assumes that all the query elements are well-formed w.r.t the variable vector, i.e. the tableau, bounds and variable vector dimensions match, and the proof-tree splits correspond to the query constraints. In the implementation, we need to prove that these properties are preserved throughout the inductive steps, with lemmas such as `well_formed_preservation` (see Table 1). Even though we need to give such indications, the custom inductive scheme is derived automatically by Imandra: the user tactic interaction with Imandra is shown in Figure 7.

## 6 Evaluation

In this section, we evaluate the new implementation of across two orthogonal axes: first, the code complexity of its main modules; and second, the performance speed, compared to the Marabou C++ implementation.

**Code Complexity.** The overview of the entire formalisation is given in Table 2. In addition to counting L.O.C., we also count the number of auxiliary lemmas per proof, as they are the main mode of user interaction with Imandra proof search. We note that, due to the clever proof production offered by the Waterfall method in Imandra [12, 37], the size of the human-written code (as exemplified in Figure 3, 7) is much smaller than the actual length of the corresponding proof (as shown in Figure 4).

**Performance Speed.** As scalability of DNN verifiers is a major factor within the DNN verification community [13, 14], any implementation of algorithms should be considered with respect to its performance. We used the proof producing version of Marabou to solve queries from two families of benchmarks: (1) *collision avoidance* (coav), which verifies a DNN with 137 ReLU neurons attempting to predict collisions of two vehicles that follow curved paths at different speed [22]; and (2) *robotics navigation* [3] with properties of a neural robot controller with 32 ReLU neurons. We chose these benchmarks because they provide a large dataset of `UNSAT` DNN verification queries that are solvable in a short time. We have disabled some optimizations within Marabou – proofs of bound tightenings that are derived based on the ReLU constraints, and thus are not proven directly by using the Farkas lemma [27].

Overall, Marabou has solved all queries and successfully generated `UNSAT` proofs for 293 coav and 213 robotics queries. For some queries, proofs were not generated due to early `UNSAT` deduction during preprocessing. We evaluated the results of 276 coav and 180 robotics queries, which have proof size smaller than 5MB. All `UNSAT` proofs were checked by the native Marabou checker and by Imandra, and both checkers have certified all queries. For

**Table 2** Summary of the entire formalisation. The Table reads as follows: a result $A$ is proven in module `A_mod`, which is $N$ lines long, calls $M$ auxiliary lemmas, and depends on libraries as listed.

| Result | Module name | L.O.C | Aux. Lem. | Library Dependencies (accumulating) |
|---|---|---|---|---|
| Poly Farkas lemma (**Theorem 5**) | farkas.iml | 194 | 21 | Imandra Standard Libraries: Real, List, Polynomials |
| Sound application of DNN polynomial Farkas lemma (**Theorem 7**) | well_formed_reduction.iml<br>bound_reduction.iml<br>tableau_reduction.iml | 41<br>359<br>356 | 13 | farkas.iml, certificate.iml, arithmetic.iml, util.iml, tightening.iml, constraint.iml, proof_tree.iml, checker.iml, bound_reduct_g.iml, mk_bound_poly.iml |
| Soundness of leaf checking (**Lemma 10**) | leaf_soundness.iml | 145 | 40 | sat.iml, split.iml bound_reduction.iml, well_formed_reduction.iml, tableau_reduction.iml |
| Single variable splits are covering (**Lemma 12**) | single_var_split_soundness.iml | 81 | 10 | |
| ReLU splits are covering (**Lemma 14**) | relu_split_soundness.iml<br>relu_case_1_bounded.iml<br>relu_case_2_bounded.iml | 113<br>337<br>338 | 18 | relu.iml |
| Soundness of node checking | node_soundness.iml | 78 | 19 | relu_split_soundness.iml, single_var_split_soundness.iml |
| Soundness (**Theorem 15**) | checker_soundness.iml | 71 | 146 | leaf_soundness.iml, node_soundness.iml |
| **Total:** | | **2113** | **267** | |

each query, we measured the time it took Marabou to deduce `UNSAT` (Verification time) and the time it took the native Marabou and Imandra to check the proofs (Native and Imandra checking time, respectively). Our evaluation is depicted in Figure 8. For the coav and robotics benchmarks, on average, Marabou solved the queries in 5.40 and 5.62 seconds, respectively; while its native checker checked the proofs in average of 0.003, 0.03 seconds. Imandra required on average 24.64 and 26.72 seconds, suggesting that, regardless of network size, Imandra requires time proportionate to the verification time of Marabou. This hypothesis needs to be further investigated. Furthermore, even though Imandra's checking time is considerably slower than Marabou's, its effect on the overall verification process is of factor $4.56\times - 4.76\times$ slowdown.

Although performance differences are expected due to the use of more precise arithmetic, our results suggest a crucial trade-off between scalability and reliability. We believe that some performance difference is due to the use of verification-oriented data structures such as lists, and thus using optimized structures such as maps may improve speed significantly as shown in [19], especially in the presence of sparse vectors in DNN Verification queries.

## 7 Conclusions, Future and Related Work

The nascent field of DNN verifiers faces several challenges, and certification of DNN verifiers themselves is widely recognized as one of them [15, 45]. Recent work of [27] laid theoretical foundations for the certification of the DNN verifier Marabou, by connecting its certificate

producing version with the well-known Farkas lemma. In this paper, we take inspiration from the *proof-carrying code* [35, 31] and *self-certifying code* [33] tradition and propose a framework that implements, executes, and verifies a checker for certificates produced by Marabou, *within the same programming language*, thus obtaining high assurance levels. Moreover, thanks to Imandra's native use of infinite precision reals, we avoid the problem of floating point imprecision in the checker. Evaluating our checker suggests a trade-off between reliability and scalability. We hypothesize that the differences originate primarily due to the use of infinite-precision arithmetic and the choice of verification-oriented data structures. We leave a more detailed analysis for future work.

## 7.1 Related Work

We hope to encourage further collaboration between verification and programming language communities. Notable is the recent success of industrial ITPs, such as Imandra [38] and F* [4, 32]. Such languages are particularly suitable for AI verification, thanks to their automation and other modern features [20]. Our work opens the opportunity for future integration of DNN verifiers in these provers and ITPs more generally.

**Imandra vs other Provers.** In principle, this work could be replicated in other proof assistants such as Isabelle/HOL, ACL2, Lean, PVS or Rocq, and it could be interesting to do so. Anecdotally, we find that Imandra's high level of proof automation, lemma discovery features, integrated bounded and unbounded verification, and efficient model execution make it an ideal environment for developing verified tools such as our verified proof checker.

**Proof-evidence production for SMT solvers.** is a known problem [34, 6, 5, 18], and we build on some experience in this domain. However, no SMT solver has been verified in an ITP as far as we are aware, the closest work in this direction comes from ITPs that integrate SMT solvers for proof automation and can verify the proof evidence [9, 23].

**Existing formalisations of the Farkas Lemma.** Although our proof of the Farkas lemma takes into consideration the previous experience in other ITPs [8, 11, 36], it had to include substantial modifications to cover the DNN case and in particular Imandra Theorem 7 is original relative to the cited papers. Moreover, we prove Farkas lemma in a polynomial form (in Section 3). As opposed to the original matrix-based Farkas lemma [43], the polynomial version yields easier automation in Imandra.

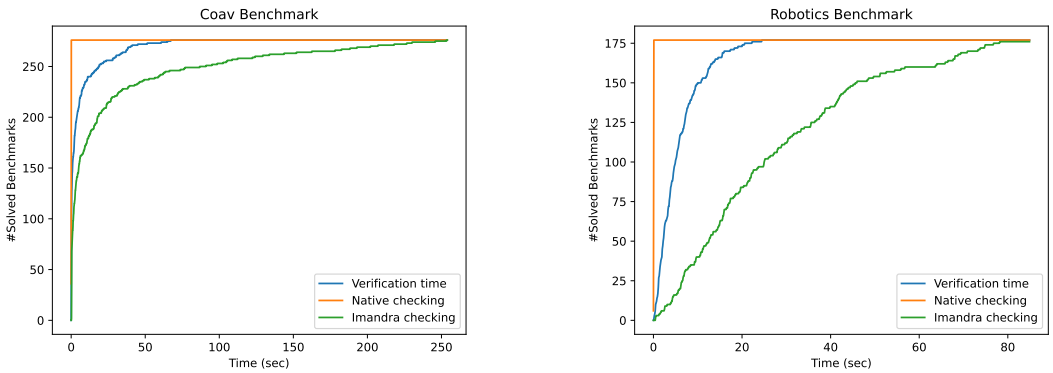



**Figure 8** Comparison of verification, native checking and Imandra checking time. Each point $(x, y)$ represents the number $x$ of verified/checked instances in $y$ seconds.

## 7.2 Future Work

There are a number of directions in which we plan to extend this work.

**From Farkas vectors to specifications.** We plan to lift the soundness of the checker (Theorem 15) to the level of DNN verification queries, and, via a certified compilation procedure, to higher-order specification languages such as Vehicle [16]. This would extend formal verification to encompass the left-hand-side blocks of Figure 1. Thus, this paper can be seen as a first step towards developing methods of *proof-carrying neuro-symbolic code* [31].

**Optimisation of Proof Checking.** One of Marabou's key techniques for scaling to larger verification problems is the use of *theory lemmas* for dynamic bound tightening. These lemmas characterize the connections between variables that participate in a ReLU constraint, and are not proven directly by using the Farkas lemma. This feature is supported by the proof production in [27]. Although it is possible to run DNN verification queries – and certify their proofs – without support of these theory lemmas, they are key to scaling to larger verification queries. We implemented them in Imandra, but certifying their soundness would be the next logical step to fully support proofs generated by Marabou. While supporting DNNs with other piecewise-linear activations, such as *maxpool* and *sign*, is relatively striaghtforward, non-linear activations are more challenging: their verification relies on over-approximations, which Marabou does not currently generate proofs for. This is also grounded in theoretical results [28].

**Verification of cyber-physical systems with DNN components.** is another possible application for presented results. For this, DNN verifiers need to interface with languages that can express the system dynamics and/or probabilistic safety properties [16]. The presented proof production will ensure that any such integration is sound.

## References


1 Reynald Affeldt, Yves Bertot, Cyril Cohen, Marie Kerjean, Assia Mahboubi, Damien Rouhling, Pierre Roux, Kazuhiko Sakaguchi, Zachary Stone, Pierre-Yves Strub, et al. MathComp-Analysis: Mathematical Components Compliant Analysis Library. `https://math-comp.github.io/`, 2022.

2 Xavier Allamigeon and Ricardo D. Katz. A Formalization of Convex Polyhedra Based on the Simplex Method. *Journal of Automated Reasoning*, 63(2):323–345, 2019. `doi:10.1007/S10817-018-9477-1`.

3 Guy Amir, Davide Corsi, Raz Yerushalmi, Luca Marzari, David Harel, Alessandro Farinelli, and Guy Katz. Verifying Learning-Based Robotic Navigation Systems. In *Proc. 29th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 607–627, 2023. `doi:10.1007/978-3-031-30823-9_31`.

4 Cezar-Constantin Andrici, Stefan Ciobaca, Catalin Hritcu, Guido Martínez, Exequiel Rivas, Éric Tanter, and Théo Winterhalter. Securing verified IO programs against unverified code in F*. *Proc. 51st Symposium on Principles of Programming Languages (POPL)*, pages 2226–2259, 2024. `doi:10.1145/3632916`.

5 Haniel Barbosa, Andrew Reynolds, Gereon Kremer, Hanna Lachnitt, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Arjun Viswanathan, Scott Viteri, Yoni Zohar, Cesare Tinelli, and Clark Barrett. Flexible Proof Production in an Industrial-Strength SMT Solver. In *Proc. 11th Int. Joint Conference on Automated Reasoning (IJCAR)*, pages 15–35, 2022. `doi:10.1007/978-3-031-10769-6_3`.

6 Clark Barrett, Leonardo de Moura, and Pascal Fontaine. Proofs in Satisfiability Modulo Theories. *All about Proofs, Proofs for All*, 55(1):23–44, 2015.

7 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 (NeurIPS)*, 2016.

8 Frédéric Besson, Pierre-Emmanuel Cornilleau, and David Pichardie. Modular SMT Proofs for Fast Reflexive Checking inside Coq. In *Proc. 38th Symposium on Principles of Programming Languages (POPL)*, pages 151–166, 2011. doi:10.1007/978-3-642-25379-9_13.

9 Sascha Böhme and Tjark Weber. Fast LCF-Style Proof Reconstruction for Z3. In *Proc. 1st Int. Conf. on Interactive Theorem Proving (ITP)*, pages 179–194, 2010. doi:10.1007/978-3-642-14052-5_14.

10 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. arXiv:1604.07316.

11 Ralph Bottesch, Max W. Haslbeck, and René Thiemann. Farkas' Lemma and Motzkin's Transposition Theorem. *Archive of Formal Proofs*, 2019.

12 Robert S. Boyer and J. Strother Moore. *A Computational Logic*. ACM Monograph Series. Academic Press, New York, 1979.

13 Christopher Brix, Stanley Bak, Taylor T Johnson, and Haoze Wu. The Fifth International Verification of Neural Networks Competition (VNN-COMP 2024): Summary and Results, 2024. Technical report. http://arxiv.org/abs/2412.19985. doi:10.48550/arXiv.2412.19985.

14 Christopher Brix, Mark Niklas Müller, Stanley Bak, Taylor T Johnson, and Changliu Liu. First Three Years of the International Verification of Neural Networks Competition (VNN-COMP), 2023. Technical report. http://arxiv.org/abs/2301.05815. doi:10.48550/arXiv.2301.05815.

15 L. Cordeiro, M. Daggitt, J. Girard, O. Isac, T. Johnson, G. Katz, E. Komendantskaya, E. Manino, A. Sinkarovs, and H. Wu. Neural Network Verification is a Programming Language Challenge. In *Proc. 34th European Symposium on Programming (ESOP)*, 2025.

16 Matthew L. Daggitt, Wen Kokke, Robert Atkey, Ekaterina Komendantskaya, Natalia Slusarz, and Luca Arnaboldi. Vehicle: Bridging the Embedding Gap in the Verification of Neuro-Symbolic Programs. In *Proc. 10th Int. Conf. on Formal Structures for Computation and Deductionn, (FSCD)*, 2025.

17 G. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963.

18 L. de Moura and N. Bjørner. Satisfiability Modulo Theories: Introduction and Applications. *Communications of the ACM*, 54(9):69–77, 2011. doi:10.1145/1995376.1995394.

19 Remi Desmartin, Grant O. Passmore, and Ekaterina Komendantskaya. Neural Networks in Imandra: Matrix Representation as a Verification Choice. In *Proc. 5th Int. Workshop of Software Verification and Formal Methods for ML-Enabled Autonomous Systems (FoMLAS) and 15th Int. Workshop on Numerical Software Verification (NSV)*, pages 78–95, 2022. doi:10.1007/978-3-031-21222-2_6.

20 Remi Desmartin, Grant O. Passmore, Ekaterina Komendantskaya, and Matthew Daggit. CheckINN: Wide Range Neural Network Verification in Imandra. In *Proc. 24th Int. Symposium on Principles and Practice of Declarative Programming (PPDP)*, pages 3:1–3:14, 2022. doi:10.1145/3551357.3551372.

21 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.

22 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.

23 Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark Barrett. SMTCoq: A Plug-in for Integrating SMT Solvers into Coq. In *Proc. 29th Int. Conf. Computer Aided Verification (CAV 2017)*, pages 126–133, 2017. doi:10.1007/978-3-319-63390-9_7.

24 Yizhak Elboher, Raya Elsaleh, Omri Isac, Mélanie Ducoffe, Audrey Galametz, Guillaume Povéda, Ryma Boumazouza, Noémie Cohen, and Guy Katz. Robustness Assessment of a Runway Object Classifier for Safe Aircraft Taxiing. In *Proc. 43rd Int Digital Avionics Systems Conf. (DASC)*, pages 1–6, 2024.

25 Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *Proc. 39th IEEE Symposium on Security and Privacy (SP)*, pages 3–18, 2018. doi:10.1109/SP.2018.00058.

26 Divya Gopinath, Luca Lungeanu, Ravi Mangal, Corina Pasareanu, Siqi Xie, and Huafeng Yu. Feature-Guided Analysis of Neural Networks. In *Proc. 26th Int. Conf. on Fundamental Approaches to Software Engineering (FASE)*, pages 133–142, 2023. doi:10.1007/978-3-031-30826-0_7.

27 O. Isac, C. Barrett, M. Zhang, and G. Katz. Neural Network Verification with Proof Production. In *Proc. 22nd Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 38–48, 2022.

28 Omri Isac, Yoni Zohar, Clark Barrett, and Guy Katz. DNN Verification, Reachability, and the Exponential Function Problem. In *Proc. 34th Int. Conf. on Concurrency Theory (CONCUR)*, 2023.

29 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.

30 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.

31 Ekaterina Komendantskaya. Proof-Carrying Neuro-Symbolic Code. In *Computability in Europe (CiE)*, 2025.

32 Denis Merigoux, Nicolas Chataing, and Jonathan Protzenko. Catala: A Programming Language for the Law. *Proceedings of the ACM on Programming Languages*, 5:77:1–77:29, 2021. doi:10.1145/3473582.

33 Kedar S. Namjoshi and Lenore D. Zuck. Program Correctness through Self-Certification. *Communications of the ACM*, pages 74–84, 2025. doi:10.1145/3689624.

34 G. Necula. *Compiling with Proofs*. Carnegie Mellon University, 1998.

35 George Necula. Proof-carrying code. In *Proc. 24th Symposium on Principles of Programming Languages (POPL)*, pages 106–119, 1997. doi:10.1145/263699.263712.

36 Grant Passmore. ACL2 Proofs of Nonlinear Inequalities with Imandra. *Electronic Proceedings in Theoretical Computer Science*, 393:151–160, 2023. doi:10.4204/EPTCS.393.12.

37 Grant Passmore, Simon Cruanes, Denis Ignatovich, Dave Aitken, Matt Bray, Elijah Kagan, Kostya Kanishev, Ewen Maclean, and Nicola Mometto. The Imandra Automated Reasoning System (System Description). In *Proc. 10th Int. Joint Conf. Automated Reasoning (IJCAR)*, pages 464–471, 2020. doi:10.1007/978-3-030-51054-1_30.

38 Grant Olney Passmore. Some Lessons Learned in the Industrialization of Formal Methods for Financial Algorithms. In *Proc. 24th Int. Symposium on Formal Methods (FM)*, pages 717–721, 2021. doi:10.1007/978-3-030-90870-6_39.

39 Kazuhiko Sakaguchi. vass. https://github.com/pi8027/vass, 2025.

40 Marco Sälzer and Martin Lange. Reachability Is NP-Complete Even for the Simplest Neural Networks. In *Proc. 15th Int. Conf. on Reachability Problems (RP)*, pages 149–164, 2021. doi:10.1007/978-3-030-89716-1_10.

41 Kenji Suzuki. Overview of Deep Learning in Medical Imaging. *Radiological Physics and Technology*, 10(3):257–273, 2017.

42 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.

43 R. Vanderbei. Linear Programming: Foundations and Extensions. *Journal of the Operational Research Society*, 1996.

**44** Shiqi Wang, Huan Zhang, Kaidi Xu, Xue Lin, Suman Jana, Cho-Jui Hsieh, and J Zico Kolter. Beta-CROWN: Efficient Bound Propagation with Per-neuron Split Constraints for Neural Network Robustness Verification. *Advances in Neural Information Processing Systems*, 34:29909–29921, 2021. URL: `https://proceedings.neurips.cc/paper/2021/hash/fac7fead96dafceaf80c1daffeae82a4-Abstract.html`.

**45** H. Wu, O. Isac, A. Zeljić, T. Tagomori, M. Daggitt, W. Kokke, I. Refaeli, G. Amir, K. Julian, S. Bassan, P. Huang, O. Lahav, M. Wu, M. Zhang, E. Komendantskaya, G. Katz, and C. Barrett. Marabou 2.0: A Versatile Formal Analyzer of Neural Networks. In *Proc. 36th Int. Conf. on Computer Aided Verification (CAV)*, pages 249–264, 2024.

**46** Dániel Zombori, Balázs Bánhelyi, Tibor Csendes, István Megyeri, and Márk Jelasity. Fooling a Complete Neural Network Verifier. In *Proc. 9th Int. Conf. on Learning Representations (ICLR)*, 2021.