Towards a Certified Proof Checker for Deep Neural Network Verification

Remi Desmartin[1], Omri Isac[2](✉), Grant Passmore[3], Kathrin Stark[1], Ekaterina Komendantskaya[1], and Guy Katz[2]

[1] Heriot-Watt University, Edinburgh, UK
rhd2000@hw.ac.uk
[2] The Hebrew University of Jerusalem, Jerusalem, Israel
omri.isac@mail.huji.ac.il
[3] Imandra Inc., Austin, TX, USA

Abstract. Recent developments in deep neural networks (DNNs) have led to their adoption in safety-critical systems, which in turn has heightened the need for guaranteeing their safety. These safety properties of DNNs can be proven using tools developed by the verification community. However, these tools are themselves prone to implementation bugs and numerical stability problems, which make their reliability questionable. To overcome this, some verifiers produce proofs of their results which can be checked by a trusted checker. In this work, we present a novel implementation of a proof checker for DNN verification. It improves on existing implementations by offering numerical stability and greater verifiability. To achieve this, we leverage two key capabilities of Imandra, an industrial theorem prover: its support for exact real arithmetic and its formal verification infrastructure. So far, we have implemented a proof checker in Imandra, specified its correctness properties and started to verify the checker's compliance with them. Our ongoing work focuses on completing the formal verification of the checker and further optimising its performance.

Keywords: Deep Neural Network · Formal Verification · AI Safety

1 Introduction

Applications of deep neural networks (DNNs) have grown rapidly in recent years, as they are able to solve computationally hard problems. This has led to their wide use in safety-critical applications like medical imaging [33] or autonomous aircraft [19]. However, DNNs are hard to trust for safety-critical tasks, notably because small perturbations in their inputs – whether from faulty sensors or

R. Desmartin and O. Isac—Both authors contributed equally.
R. Desmartin—Funded by Imandra Inc.
E. Komendantskaya—Funded by EPSRC grant AISEC (EP/T026952/1) and NCSC grant "Neural Network Verification: in search of the missing spec.".

malicious adversarial attacks – may cause large variations of their outputs, leading to potentially catastrophic system failures [34]. To circumvent this issue, the verification community has developed techniques to guarantee DNN correctness using formal verification, employing mathematically rigorous techniques to analyse DNNs' possible behaviours in order to prove it safe and compliant e.g. [2,9,15,16,21,23,31,32,36]. Along with these DNN verifiers, the community holds the annual competition VNN-COMP [8] that led to the standardisation of formats [3].

Usually, DNN verifiers consider a trained DNN and prove input-output properties, e.g. that for inputs within a delimited region of the input space, the network's output will be in a safe set. Besides verifying DNNs at a component level, verification has the power to verify larger systems integrating DNNs. Integration of DNN verifiers in larger verification frameworks has been studied as well [10], and it requires the DNN verifiers to provide results that can be checked by the system-level verifier.

Unfortunately, DNN verifiers are susceptible to errors as any other program. One source of problems is floating-point arithmetic used for their internal calculations. While crucial for performance, floating-point arithmetic also leads to numerical instability and is known to compromise the soundness of DNN verifiers [18]. As the reliability of DNN verifiers becomes questionable, it is necessary to check that their results are not erroneous. When a DNN verifier concludes there exists a counterexample for a given property, this result can be easily checked by evaluating the counterexample over the network and ensuring the property's violation. However, when a verifier concludes that no counterexample exists, ensuring the correctness of this result becomes more complicated.

To overcome this, DNN verifiers may produce proofs for their results, allowing an external program to check their soundness. Producing proofs is a common practice [4,26], and was recently implemented on top of the Marabou DNN verifier [17,21]. Typically, proof checkers are simpler programs than the DNN verifiers, and hence much easier to inspect and verify. Moreover, while verifiers are usually implemented in performance-oriented languages such as C++, trusted proof checkers could be implemented in languages suitable for verification.

Functional programming languages (FPL), such as Haskell, OCaml and Lisp, are well-suited for this task, thanks to their deep relationship with logics employed by theorem provers. In fact, some FPLs, such as Agda [27], Coq [1], ACL2 [22], Isabelle [30] and Imandra [28] are also theorem provers in their own right. Implementing and then verifying a program in such a theorem prover allows to bridge the verification gap, i.e. minimise the discrepancies that can exist between the original (executable) program and its verified (abstract) model [7].

In this paper, we describe our ongoing work to design, implement and verify a *formally-verifiable and infinitely-precise proof checker* for DNN verifiers. We have implemented an adaptation of a checker of `UNSAT` proofs produced by the Marabou DNN verifier [17,21] to Imandra [28], a programming language with its own theorem prover that has been successfully used in fintech applications [29]. Three key features make Imandra a suitable tool: arbitrary precision ("exact") real arithmetic, efficient code extraction and the first-class integration of formal verification. Support for infinite precision real arithmetic prevents errors due to numerical instability in the proof checker. In the linear case, this corresponds to arbitrary precision rational arithmetic. In the nonlinear case, real computation in Imandra takes place over a canonical real closed field, the field of real algebraic numbers. The ability to extract verified Imandra code to native OCaml improves scalability as it can then benefit from the standard OCaml compiler's optimisations. Finally, with Imandra's integrated formal verification, we can directly analyse the correctness of the proof checker we implement. Capacities of Imandra in DNN verification have already been reported in [13].

Contributions. We improve on the previous implementation [17] in two ways: firstly, our checker can itself be formally verified by Imandra; and secondly, Imandra's infinite precision numbers eliminate the possibility of the usual floating point arithmetic errors. This increases the checker's reliability and overcomes a main barrier in integrating DNN verifiers in system-level checkers. Since reliability usually compromises scalability, our proof checker supports two checking modes: (i) one uses verified data structures at the expense of computation speed; (ii) the other accepts some parts of the proof without checking.

Our ongoing work is currently focused on formally verifying the proof checker. So far, we have managed to verify that our checker complies with linear algebra theorems, and we attempt to leverage these results to verify the proof checker as a whole in the future.

Paper Organisation. The rest of this paper is organised as follows. In Sect. 2 we provide relevant background on DNN verification and proof production. In Sect. 3 and Sect. 4 we respectively describe our proof checker, and our ongoing work towards formally verifying it using Imandra. In Sect. 5 we conclude our work, and describe our plans for completing our work and for the future.

An extended version of this paper is available in [12].

2 Background

2.1 DNN Verification

Throughout the paper, we focus on DNNs with $\mathrm{ReLU}(x) = max(0, x)$ activation functions, though all our work can be extended to DNNs using any piecewise-linear activation functions (e.g. *max pooling*). We refer the reader to the extended version of this paper for a formal definition of DNNs and activation functions [12]. An example of a DNN appears in Fig. 1.

The *DNN verification problem* is the decision problem of deciding whether for a given DNN $\mathcal{N} : \mathbb{R}^m \to \mathbb{R}^k$ and a property $P \subseteq \mathbb{R}^{m+k}$, there exists an input $x \in \mathbb{R}^m$ such that $\mathcal{N}(x) = y \wedge P(x, y)$. If such x exists, the verification query is *satisfiable* (`SAT`); otherwise it is *unsatisfiable* (`UNSAT`). Typically, P represents an erroneous behaviour, thus an input x satisfying the query serves as a counterexample and `UNSAT` indicates the network acts as expected.

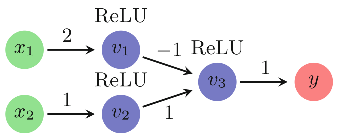

Fig. 1. A Simple DNN. The bias parameters are all set to zero and are ignored. Green denotes input nodes, blue hidden nodes, and red output nodes. (Color figure online)

Due to its linear and piecewise-linear structure, a DNN verification query can be reduced to an instance of Linear Programming (LP) [11], representing the affine functions of the DNN, and piecewise-linear constraints that represent the activation functions and the property. This reduction makes algorithms for solving LP instances, coupled with a *case-splitting* approach for handling the piecewise-linear constraints [5,20], a prime scheme for DNN verification, which we call *LP-based DNN verifiers*.

The widely used Simplex algorithm [11,14,20], is typically used by such verifiers. Based on the problem constraints, the algorithm initiates a matrix A called the *tableau*, a variable vector x and two *bound vectors* u, l such that $l \leq x \leq u$. The Simplex algorithm then attempts to find a solution to the system:

$$Ax = 0 \land l \leq x \leq u \tag{1}$$

or concludes that none exists. For clarity, we denote $u(x_i), l(x_i)$ as the upper and lower bounds of the variable x_i, instead of u_i, l_i.

Example 1. Consider the DNN in Fig. 1 and the property P that holds if and only if $(x_1, x_2) \in [-1, 1]^2 \land y \in [2, 3]$. We later show a proof of `UNSAT` for this query. We assign variables x_1, x_2, y to the input and output neurons. For all $i \in 1, 2, 3$ we assign a couple of variables f_i, b_i for the inputs and outputs of the neurons v_i, where $f_i = \text{ReLU}(b_i)$. We then get the linear constraints and bounds (where some bounds were arbitrarily fixed for simplicity):

$$b_1 = 2x_1, \;\; b_2 = x_2, \;\; b_3 = f_2 - f_1, \;\; y = f_3 \tag{2}$$

$$-1 \leq x_1, x_2, b_2 \leq 1, \; 0 \leq f_2 \leq 1, \; -2 \leq b_1, b_3 \leq 2, \; 0 \leq f_1, f_3 \leq 2, \; 2 \leq y \leq 3 \tag{3}$$

and the piecewise linear constraints: $\forall i \in 1, 2, 3 : f_i = \text{ReLU}(b_i)$

Then, an LP-based DNN verifier initiates the input for the Simplex algorithm:

$$A = \begin{bmatrix} 2 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{bmatrix} \quad \begin{aligned} u &= \begin{bmatrix} 1 & 1 & 2 & 1 & 2 & 2 & 1 & 2 & 3 \end{bmatrix}^\intercal \\ x &= \begin{bmatrix} x_1 & x_2 & b_1 & b_2 & b_3 & f_1 & f_2 & f_3 & y \end{bmatrix}^\intercal \\ l &= \begin{bmatrix} -1 & -1 & -2 & -1 & -2 & 0 & 0 & 0 & 2 \end{bmatrix}^\intercal \end{aligned}$$

In addition to the piecewise-linear constraints $\forall i \in 1, 2, 3 : f_i = \text{ReLU}(b_i)$.

One of the key tools used by the Simplex algorithm, and consequently by DNN verifiers, is dynamic bound tightening. This procedure allows deducing tighter bounds for each variable and is crucial for the solver's performance. For example, using the above equation $f_3 = y$ and the bound $u(y) = 2$, we can deduce $u(f_3) = 2$, and further use this bound to deduce other bounds as well. The piecewise-linear constraints introduce rules for tightening bounds as well, which we call *Theory-lemmas*. For instance, the output variable f_3 of the ReLU constraint of the above example is upper bounded by the input variable b_3, whose upper bound is 2. The list of supported lemmas appears in [12].

The case-splitting approach is used over the linear pieces of some piecewise-linear constraints, creating several sub-queries with each adding new information to the Simplex algorithm. For example, when performing a split over a constraint of the form $y = \text{ReLU}(x)$, two sub-queries are created. One is enhanced with $y = x \wedge x \geq 0$, and the other with $y = 0 \wedge x \leq 0$. The use of case-splitting also induces a tree structure for the verification algorithm, with nodes corresponding to the splits applied. On every node, the verifier attempts to conclude the satisfiability of the query based on its linear constraints. If it concludes an answer, then this node represents a leaf. In particular, a tree with all leaves corresponding to an `UNSAT` result of Simplex is a search tree of an `UNSAT` verification query.

2.2 Proof Production for DNN Verification

Proof production for `SAT` is straightforward using a satisfying assignment. On the other hand, when a query is `UNSAT`, the verification algorithm induces a search tree, where each leaf corresponds to an `UNSAT` result of the Simplex algorithm for that particular leaf. Thus, a proof of `UNSAT` is comprised of a matching proof tree where each leaf contains a proof of the matching Simplex `UNSAT` result. Proving `UNSAT` results of Simplex is based on a constructive version of the Farkas Lemma [35], which identifies the proof for `UNSAT` LP instances. Formally, it was proven [17] that:

Theorem 1. *Let $A \in M_{m\times n}(\mathbb{R})$ and $l, x, u \in \mathbb{R}^n$, such that $A \cdot x = 0$ and $l \leq x \leq u$, exactly one of these two options holds:*

1. *The `SAT` case: $\exists x \in \mathbb{R}^n$ such that $A \cdot x = 0$ and $l \leq x \leq u$.*
2. *The `UNSAT` case: $\exists w \in \mathbb{R}^m$ such that for all $l \leq x \leq u$, $w^\intercal \cdot A \cdot x < 0$, whereas $0 \cdot w = 0$. Thus, w is a proof of the constraints' unsatisfiability.*

Moreover, these vectors can be constructed while executing the Simplex algorithm.

To construct the proof vectors, two column vectors are assigned to each variable x_i, denoted $f_u(x_i), f_l(x_i)$, which are updated during bound tightening. These vectors are used to prove the tightest upper and lower bounds of x_i deduced during the bound tightenings performed by Simplex, based on u, l and A. Constructing the proof vector of Theorem 1 case *2.* allows the proof checker to check the unsatisfiability of the query immediately, without repeating Simplex procedure. This mechanism was designed and implemented [17], on top of the Marabou DNN verifier [21].

Supporting the complete tree structure of the verification algorithm is done by constructing the proof tree in a similar manner to the search tree—every split performed in the search directly creates a similar split in the proof tree, with updates to the equations and bounds introduced by the split. Proving theory lemmas is done by keeping details about the bound that invoked the lemma together with a Farkas vector proving its deduction and the newly learned bound, and adding them to the corresponding proof tree node.

3 The Imandra Proof Checker

Our proof checker is designed to check proofs produced by the Marabou DNN verifier [21], to the best of our knowledge the only proof producing DNN verifier. When given a Marabou proof of `UNSAT` as a JSON [6] file, the proof checker reconstructs the proof tree using datatypes encoded in Imandra.

The proof tree consists of two different node types—a proof node and a proof leaf. Both node types contain a list of lemmas and a corresponding split. In addition, a node contains a list of its children, and a leaf contains a contradiction vector, as constructed by Theorem 1. This enables the checker to check the proof tree structure at the type level. The proof checker also initiates a matrix A called a *tableau*, vectors of upper and lower bounds u, l and a list of piecewise-linear constraints (see Sect. 2.1).

The checking process consists of traversing the proof tree. For each node, the checker begins by locally updating u, l and A according to the split, and optionally checking the correctness of all lemmas. Lemma checking is similar to checking contradictions, as shown in Example 2 below (see [12] for details).

If the node checked is not a leaf, then the checker will check that all its children's splits correspond to some piecewise-linear constraint of the problem i.e. one child has a split of the form $y = x \wedge x \geq 0$ and the other of the form $y = 0 \wedge x \leq 0$ for some constraint $y = \text{ReLU}(x)$. If the checker certifies the node, it will recursively check all its children, passing changes to u, l and A to them.

When checking a leaf, the checker checks that the contradiction vector w implies `UNSAT`, as stated in Theorem 1. As implied from the theorem, the checker will first create the row vector $w^\intercal \cdot A$, and will compute the upper bound of its underlying linear combination of variables $w^\intercal \cdot A \cdot x$. The checker concludes by asserting this upper bound is negative.

The checker then concludes that the proof tree represents a correct proof if and only if all nodes passed the checking process.

Example 2. Consider the simple proof in Fig. 2. The root contains a single lemma and each leaf contains a contradiction vector, which means the verifier performed a single split. In addition, the proof object contains the tableau A, the bound vectors u, l, and the ReLU constraints as presented in Example 1.

The proof checker begins by checking the lemma of the root. It does so by creating the linear combination $\begin{bmatrix}0 \; 0 \; 1 \; 0\end{bmatrix}^\intercal \cdot A \cdot x = -b_3 - f_1 + f_2$. As the lemma is invoked by the upper bound of b_3, the checker uses the equivalent equation $b_3 = f_2 - f_1$, which gives the upper bound $u(b_3) = u(f_2) - l(f_1) = 1$. We can indeed deduce the bound $u(f_3) = 1$ based on the constraint $f_3 = \text{ReLU}(b_3)$, so the lemma proof is correct. Then, the checker certifies that the splits $f_3 = 0 \wedge b_3 \leq 0$

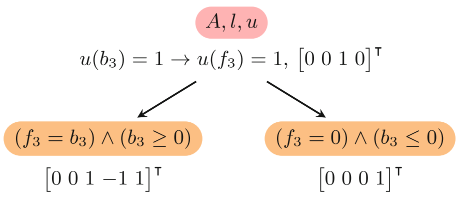
A, l, u
$u(b_3) = 1 \rightarrow u(f_3) = 1,\ \begin{bmatrix}0 & 0 & 1 & 0\end{bmatrix}^\intercal$
$(f_3 = b_3) \wedge (b_3 \geq 0)$
$\begin{bmatrix}0 & 0 & 1 & -1 & 1\end{bmatrix}^\intercal$
$(f_3 = 0) \wedge (b_3 \leq 0)$
$\begin{bmatrix}0 & 0 & 0 & 1\end{bmatrix}^\intercal$

Fig. 2. A proof tree example.

and $f_3 = b_3 \wedge b_3 \geq 0$ correspond to the two splits of $f_3 = \mathrm{ReLU}(b_3)$. The checker then begins checking the left leaf. It starts by updating $l(b_3) = 0$ and adding the equation $f_3 = b_3$ as the row $\begin{bmatrix}0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 & 0\end{bmatrix}$ to A. Then, the checker checks the contradiction vector by computing $\begin{bmatrix}0 & 0 & 1 & -1 & 1\end{bmatrix}^\intercal \cdot A \cdot x = -f_1 + f_2 - y$. The upper bound of this combination is $-l(f_1) + u(f_2) - l(y) = -1$ which is negative, thus proving `UNSAT` for the leaf according to Theorem 1. Checking the right leaf is done similarly. After checking all nodes, the checker asserts the proof tree indeed proves `UNSAT` for the whole query.

Implementation in Imandra, OCaml Extraction and Evaluation. Porting the proof checker from C++ to Imandra necessitates taking into account the trade-off between scalability and computation.

The choice of data structures for common objects – like vectors – is essential in the balance between scalability and efficiency [13]. In this work, we experiment with two different implementations for vectors: native OCaml lists, and sparse vectors using Imandra's built-in `Map` data type, based on binary search trees. The latter has better performance but the former makes it easier to verify, so for now our verification efforts focus on the native list implementation (for further discussion of data structure choice, see [12]).

Imandra's logic includes theories for arbitrary precision integer and real arithmetic, which for integers and linear (rational) computations over reals are implemented using OCaml's `Zarith` library [24]. Zarith and GMP, its underlying library, are not verified but trusted. Nonlinear real computations are handled with on-demand field extensions via constructive real closures [25]. As a result, the Imandra implementation of the checker supports arbitrary precision real arithmetic with no overhead.

Executing code within Imandra's reasoning environment is helpful during the implementation and verification process, but is not optimised for performance. To that end, `imandra-extract` is a facility to extract native OCaml code that can be compiled – and optimised – with standard OCaml compilers. The extracted code retains Imandra's semantics, meaning that it still uses infinite precision real arithmetic. An initial comparison of the execution time for checking the same proofs from the ACAS-Xu benchmark [21] in the C++ implementation and in the extracted OCaml code with native lists shows that our implementation is about 150 times slower than the original implementation but stays within a reasonable time, i.e. less than 40 min for all the examples ran (see Table 1). Further optimisations and a comprehensive benchmark are ongoing work.

Table 1. Comparison of the execution speed for checking Marabou proofs for verification tasks from the ACAS Xu benchmark.

ACAS-	C++ [17]	Imandra (native lists)		Imandra (sparse vectors)	
Xu tasks	Full (s)	Partial (s)	Full (s)	Partial (s)	Full (s)
N(2, 9) p3	5.130	167.078	878.075	15.125	4784.866
N(2, 9) p4	5.658	206.675	1019.770	11.208	8817.575
N(3, 7) p3	10.557	299.608	1493.763	24.979	1638.844
N(5, 7) p3	2.568	58.288	311.096	50.365	12276.323
N(5, 9) p3	15.116	424.816	2210.472	30.611	6265.039

4 Specification of the Proof Checker's Correctness

We aim to verify the two main checks performed by the proof checker when traversing the proof tree (see Sect. 3): contradictions and theory lemmas.

Contradictions Checking. We want to verify that our proof checker identifies correctly when a contradiction vector is a valid proof of `UNSAT`, thus satisfying Theorem 1 (case 2). Formally, the specification can be given as:

For any contradiction vector w, tableau A, bounds u, l, and a bounded input $l \leq x \leq u$, if the upper bound of $w^T \cdot A \cdot x$ is negative, then x cannot satisfy the constraints $A \cdot x = 0 \wedge l \leq x \leq u$. The Imandra implementation of this specification is given in Listing 1.1.

```
theorem contra_correct x contra tableau u_bounds l_bounds =
  is_bounded x u_bounds l_bounds
  && check_contradiction contra tableau u_bounds l_bounds
   ==> not (null_product tableau x)
```

Listing 1.1. *High-level theorem formalising correctness of contradiction checking. The function* `check_contradiction` *is a key component of the proof checker which should return* `true` *iff the linear combination of the tableau and contradiction vectors has a negative upper bound.*

Theory Lemmas. We aim to prove that each theory lemma within the proof corresponds to a known theory lemma (see [12] for further details).

Proving the specification necessitates guiding Imandra by providing supporting lemmas, in our case properties of linear algebra. After proving these intermediary lemmas, Imandra's proof automation can apply them automatically, or we can manually specify which lemma to apply.

So far we have defined and proved that our checker is coherent with known properties of linear algebra (e.g. Listing 1.2). Our current work focuses on building on top of these lemmas to fully prove the checker's correctness.

```
lemma dot_product_coeff x y c =
  dot_product x (list_mult y c) = c *. dot_product x y
[@@auto]

lemma dot_product_coeff_eq x y c =
  dot_product x y = 0. ==> dot_product x (list_mult y c) = 0.
[@@auto][@@apply dot_product_coeff x y c]
```

Listing 1.2. *Definition of lemmas proved in Imandra;* `dot_product_coeff`*, which defines the homogeneity of the dot-product, is used to prove the second lemma.*

5 Discussion and Future Work

We have implemented a checking algorithm for proofs generated by a DNN verifier in the functional programming language of Imandra, enabling the checking algorithm to be infinitely precise and formally verifiable by Imandra's prover.

Compared to previous work, our implementation presents two new guarantees: it avoids numerical instability by using arbitrary precision real numbers instead of floating-point numbers; and its correctness can be formally verified as it is implemented in a theorem prover. The arbitrary precision linear real arithmetic library, GMP, is standard but it is not itself formally verified.

One limitation of our work is the discrepancy between the initial verified model and the model encoded in the checked proofs: training and verification frameworks use floating point numbers; Marabou uses overapproximation to mitigate the numerical instability, and rounds the values during the proof serialisation; the proof checker then uses exact real arithmetic to reason about the weights. Ultimately though, if the checker validates a proof, it means that the encoded model satisfies the property and can be extracted and deployed.

As expected, adding safety guarantees comes at a cost of performance, but the extraction of native OCaml minimises the overhead compared to the unverified C++ implementation. Furthermore, using an FPL checker to check proofs produced by a DNN verifier is a first step towards integrating DNN verification into the verification of larger systems with DNN-enabled components.

Our immediate future work is to continue the verification of the proof checker. In addition, we intend to identify cases where the existing checker implementation fails (e.g. due to numerical instability) and ours correctly checks the proof. Investigating further optimisations is also a promising direction by implementing better performance data structures, such as AVL trees.

Acknowledgements. We thank the reviewers for their valuable comments and suggestions, which greatly helped us to improve our manuscript.

References

1. The Coq Proof Assistant (1984). https://coq.inria.fr
2. Bak, S.: Nnenum: verification of ReLU neural networks with optimized abstraction refinement. In: Proceedings of 13th International Symposium NASA Formal Methods (NFM), pp. 19–36 (2021)
3. Barrett, C., Katz, G., Guidotti, D., Pulina, L., Narodytska, N., Tacchella, A.: The Verification of Neural Networks Library (VNN-LIB) (2019). https://www.vnnlib.org/
4. Barrett, C., de Moura, L., Fontaine, P.: Proofs in satisfiability modulo theories. In: All About Proofs, Proofs for All, vol. 55, no. 1, pp. 23–44 (2015)
5. Bastani, O., Ioannou, Y., Lampropoulos, L., Vytiniotis, D., Nori, A., Criminisi, A.: Measuring neural net robustness with constraints. In: Proceedings of 30th Conference on Neural Information Processing Systems (NeurIPS) (2016)
6. Bray, T.: The JavaScript Object Notation (JSON) Data Interchange Format (2014). https://www.rfc-editor.org/info/rfc7159
7. Breitner, J., et al.: Ready, set, verify! applying Hs-to-Coq to real-world Haskell code. J. Funct. Program. **31**, e5 (2021)
8. Brix, C., Müller, M.N., Bak, S., Johnson, T.T., Liu, C.: First Three Years of the International Verification of Neural Networks Competition (VNN-COMP). Technical report (2023). http://arxiv.org/abs/2301.05815
9. Brix, C., Noll, T.: Debona: Decoupled Boundary Network Analysis for Tighter Bounds and Faster Adversarial Robustness Proofs. Technical report (2020). http://arxiv.org/abs/2006.09040
10. Daggitt, M.L., Kokke, W., Atkey, R., Arnaboldi, L., Komendantskaya, E.: Vehicle: Interfacing Neural Network Verifiers with Interactive Theorem Provers. Technical report (2022). http://arxiv.org/abs/2202.05207
11. Dantzig, G.: Linear Programming and Extensions. Princeton University Press, Princeton (1963)
12. Desmartin, R., Isac, O., Passmore, G., Stark, K., Katz, G., Komendantskaya, E.: Towards a Certified Proof Checker for Deep Neural Network Verification. Technical report (2023). http://arxiv.org/abs/2307.06299
13. Desmartin, R., Passmore, G.O., Komendantskaya, E.: Neural networks in imandra: matrix representation as a verification choice. In: Proceedings of 5th International Workshop of Software Verification and Formal Methods for ML-Enabled Autonomous Systems (FoMLAS) and 15th International Workshop on Numerical Software Verification (NSV), pp. 78–95 (2022)
14. Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006). https://doi.org/10.1007/11817963_11
15. Ferrari, C., Mueller, M.N., Jovanović, N., Vechev, M.: Complete verification via multi-neuron relaxation guided branch-and-bound. In: Proceedings of 10th International Conference on Learning Representations (ICLR) (2022)
16. Henriksen, P., Lomuscio, A.: DEEPSPLIT: an efficient splitting method for neural network verification via indirect effect analysis. In: Proceedings of 30th International Joint Conference on Artificial Intelligence (IJCAI), pp. 2549–2555 (2021)
17. Isac, O., Barrett, C., Zhang, M., Katz, G.: Neural network verification with proof production. In: Proceedings 22nd International Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 38–48 (2022)

18. Jia, K., Rinard, M.: Exploiting verified neural networks via floating point numerical error. In: Drăgoi, C., Mukherjee, S., Namjoshi, K. (eds.) SAS 2021. LNCS, vol. 12913, pp. 191–205. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-88806-0_9
19. Julian, K., Kochenderfer, M., Owen, M.: Deep neural network compression for aircraft collision avoidance systems. J. Guid. Control. Dyn. **42**(3), 598–608 (2019)
20. Katz, G., Barrett, C., Dill, D., Julian, K., Kochenderfer, M.: Reluplex: a calculus for reasoning about deep neural networks. Form. Methods Syst. Des. (FMSD) **60**(1), 87–116 (2021)
21. Katz, G., et al.: The marabou framework for verification and analysis of deep neural networks. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 443–452. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_26
22. Kaufmann, M., Moore, J.S.: ACL2: an industrial strength version of Nqthm. In: Proceedings of 11th Conference on Computer Assurance (COMPASS), pp. 23–34 (1996)
23. Khedr, H., Ferlez, J., Shoukry, Y.: PEREGRiNN: penalized-relaxation greedy neural network verifier. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12759, pp. 287–300. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_13
24. Miné, A., Leroy, X., Cuoq, P., Troestler, C.: The Zarith Library (2023). https://github.com/ocaml/Zarith
25. de Moura, L., Passmore, G.O.: Computation in real closed infinitesimal and transcendental extensions of the rationals. In: Bonacina, M.P. (ed.) CADE 2013. LNCS (LNAI), vol. 7898, pp. 178–192. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38574-2_12
26. Necula, G.: Compiling with Proofs. Carnegie Mellon University (1998)
27. Norell, U.: Dependently typed programming in Agda. In: Proceedings of 4th International Workshop on Types in Language Design and Implementation (TLDI), pp. 1–2 (2009)
28. Passmore, G., et al.: The imandra automated reasoning system (system description). In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020. LNCS (LNAI), vol. 12167, pp. 464–471. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51054-1_30
29. Passmore, G.O.: Some lessons learned in the industrialization of formal methods for financial algorithms. In: Huisman, M., Păsăreanu, C., Zhan, N. (eds.) FM 2021. LNCS, vol. 13047, pp. 717–721. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-90870-6_39
30. Paulson, L.C.: Isabelle: A Generic Theorem Prover. Springer, Heidelberg (1994). https://doi.org/10.1007/BFb0030541
31. Prabhakar, P., Afzal, Z.R.: Abstraction based output range analysis for neural networks. In: Proceedings of 32nd International Conference on Neural Information Processing Systems (NeurIPS), pp. 15762–15772 (2019)
32. Smith, J., Allen, J., Swaminathan, V., Zhang, Z.: Refutation-Based Adversarial Robustness Verification of Deep Neural Networks (2021)
33. Suzuki, K.: Overview of deep learning in medical imaging. Radiol. Phys. Technol. **10**(3), 257–273 (2017)

34. Szegedy, C., et al.: Intriguing Properties of Neural Networks. Technical report (2013). http://arxiv.org/abs/1312.6199
35. Vanderbei, R.: Linear programming: foundations and extensions. J. Oper. Res. Soc. (1996)
36. Wang, S., et al.: Beta-CROWN: efficient bound propagation with per-neuron split constraints for neural network robustness verification. Adv. Neural. Inf. Process. Syst. **34**, 29909–29921 (2021)