Generating and Checking DNN Verification Proofs

Anonymous Author(s)
Affiliation
Address
email

Abstract

Deep Neural Networks (DNN) have emerged as an effective approach to implementing challenging sub-problems. They are increasingly being used as components in critical transportation, medical, and military systems. However, like human-written software, DNNs may have flaws that can lead to unsafe system performance. To confidently deploy DNNs in such systems, strong evidence is needed that they do not contain such flaws. This has led researchers to explore the adaptation and customization of software verification approaches to the problem of neural network verification (NNV). Many dozens of NNV tools have been developed in recent years and as a field these techniques have matured to the point where realistic networks can be analyzed to detect flaws and to prove conformance with specifications. NNV tools are highly-engineered and complex may harbor flaws that cause them to produce unsound results.

We identify commonalities in the algorithmic approaches taken by NNV tools to define a verifier independent proof format – activation pattern tree proofs (APTP) – and design an algorithm for checking those proofs that is proven correct and optimized to enable scalable checking. We demonstrate that existing verifiers can efficiently generate APTP proofs, and that an `APTPchecker` significantly outperforms prior work on a benchmark of 16 neural networks and 400 NNV problems, and that it is robust to variation in APTP proof structure arising from different NNV tools. `APTPchecker` is available at: https://anonymous.4open.science/r/APTPchecker-0482/.

1 Introduction

As deep neural networks (DNNs) become integral components of critical systems such as autonomous vehicles [1], medical decision-making [2], and robotics [3], it is imperative to rigorously verify their behavior. In recent years, the research community has developed a wide-range of algorithmic techniques to verify DNN properties and incorporated them into tools that now scale to realistic DNN models millions of neurons [4]. These advances have enabled verification of properties such as robustness to input perturbations and conformance to safety specifications [4, 5, 6, 7].

However, despite the progress in algorithmic advances, a fundamental question remains: "How can we *trust* the results produced by DNN verification tools?" Recent competitions such as VNN-COMP [4] have revealed correctness issues in multiple tools, including cases where a verifier incorrectly declared a property to be proven even when a counterexample exists. These errors are difficult to detect and debug due to the complexity of verifier implementations, which often exceed tens of thousands of lines of code and employ intricate optimization techniques, e.g., top of the line DNN verification tools such as $\alpha\beta$-`CROWN` [7] and `NeuralSAT` [8] have 20k SLOC implementations with complex algorithms that may harbor bugs. Without a mechanism to independently validate verification results, correctness of DNN verification tools cannot be assured and therefore posing a serious obstacle to deploying DNNs in safety-critical domains.

Submitted to 39th Conference on Neural Information Processing Systems (NeurIPS 2025). Do not distribute.

To address this, we propose *proof-producing DNN verification*: an approach in which verifiers emit a formal proof object that encodes the reasoning steps behind the verification result, and a separate, minimal proof checker certifies the proof's validity. This paradigm, long established in classical logic and SAT solving [9, 10, 11], brings transparency, auditability, and trust to the verification process.

More specifically, we analyze the broad class of "branch and bound" (BaB) DNN verification algorithms and reveal that they share two commonalities: (1) they refine the abstractions they use by performing case *splitting* to reason about the different phases of neuron activation, and (2) within cases they perform reasoning steps that can be formulated within the broad class of *mixed integer linear programming* (MILP) problems. Based on these insights, we show that BaB DNN verification naturally emit *activation pattern tree proofs* (`APTP`), which are a compact representation of the reasoning steps performed by the verifier (§3.1) . We also define a verifier independent `APTP` format that can be efficiently generated on-the-fly during DNN verification (§3.2). Finally, we present the `APTPchecker` algorithm along with a suite of optimizations to check `APTP` proofs (§4).

In addition to these foundational results, other contributions include: (1) extending two state-of-the-art DNN verifiers to efficiently emit `APTP` proofs; (2) implementing the small-footprint `APTPchecker` tool (800 SLOC); (3) evaluating the performance of `APTPchecker` on a benchmark of 400 verification problems involving 16 neural networks ranging up to several thousands of neurons and millions of parameters; (4) demonstrating that `APTPchecker` significantly outperforms previous DNN proof checking approaches; and (5) demonstrating that `APTP` and `APTPchecker` are robust to variation in the structure of proofs arising from different DNN verification algorithms.

2 Background

Deep Neural Network. A *neural network* [12] consists of an input layer, multiple hidden layers, and an output layer. The output of a DNN is obtained by progressively computing the values of neurons in each layer. More specifically, the value of a hidden neuron y is $ReLU(\sum_i^n w_i v_i + b)$, where b is the bias, ws are the weights of y, vs are the neurons of preceding layer, $\sum_i^n w_i v_i + b$ is the *affine transformation*, and $ReLU(x) = \max(x, 0)$ is the *activation function*. The Rectified Linear Unit (ReLU) is a representative of a broad class of *piece-wise linear* activation functions that could be supported by our approach. A ReLU neuron is said to be *active* if its input value is greater than zero and *inactive* otherwise.

DNN Verification. Given a DNN $\mathcal{N}$ and a property ϕ, the *DNN verification problem* asks if ϕ is a valid property of $\mathcal{N}$. In modern DNN verification, $\phi(x, y) := \phi_{in}(x) \Rightarrow \phi_{out}(y)$, where ϕ_{in} is a property over the inputs and ϕ_{out} is a property over the outputs of $\mathcal{N}$. This form of properties has been used to encode safety and security requirements of DNNs [13, 14].

DNN verification then can be formulated as checking the satisfiability of:

$$\alpha \wedge \phi_{in} \wedge \neg\phi_{out} \tag{1}$$

where α is the encoding of $\mathcal{N}$. A DNN verifier attempts to find a *counterexample* input to $\mathcal{N}$ that satisfies ϕ_{in} but violates ϕ_{out}. If Eq. 1 is unsatisfiable (e.g., no such counterexample exists), ϕ is a *valid* property of $\mathcal{N}$ and *invalid* otherwise.

For the widely-used ReLU activation problem, this problem becomes a search for *activation patterns*, i.e., boolean assignments representing activation status of neurons, that lead to satisfaction the formula in Eq. 1. Modern DNN verification techniques [7, 6, 8, 15, 16] all adopt this idea and search for satisfying assignments.

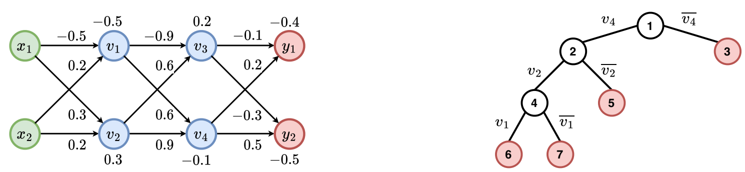
Related Work (more details in Apdx. D) Proof checking is a well-established area in constraint solving, particularly in SAT/SMT solving, with significant work on clausal proof generation and verification, such as DRAT for SAT solvers and various proof checkers like DRAT-trim and LRAT [9, 17, 11]. SMT solvers, such as Z3 and veriT, also produce proofs that can be reconstructed in proof assistants, and other solvers like MathSAT5, SMTInterpol, and CVC5 have similar capabilities [18, 19, 20, 21, 22]. However, DNN verification is a newer field, with limited research on proof checkers.

The only existing proof checking work for DNNs focuses on `Marabou`, using Farkas's lemma and implemented in the Imandra framework [23, 24, 25]. In contrast, we introduces a more expressive proof format, `APTP`, and stronger proof checker, `APTPchecker`, specifically designed for neuron-splitting DNN verification, and significantly advancing previous methods.

Alg. 1. The $\texttt{BaB}_{\texttt{NV}}$ algorithm with proof generation.

input : DNN $\mathcal{N}$, property $\phi_{in} \Rightarrow \phi_{out}$
output : (unsat, proof) if property is valid, otherwise (sat, cex)

```
1  ActPatterns ← {∅} // initialize verification problems
2  proof ← { } // initialize proof tree
3  while ActPatterns do // main loop
4      σ_i ← Select(ActPatterns) // process problem i-th
5      if Deduce(N, φ_in, φ_out, σ_i) then
6          (cex, v_i) ← Decide(N, φ_in, φ_out, σ_i)
7          if cex then return (sat, cex) // found a valid counter-example
8          ActPatterns ← ActPatterns ∪ {σ_i ∧ v_i ; σ_i ∧ ¬v_i} // new activation patterns
9      else // detect a conflict
10         proof ← proof ∪ {σ_i} // build proof tree
11 return (unsat, proof)
```



(a) A simple DNN. (b) A proof tree.

Fig. 1: Example of verifying $(x_1, x_2) \in [-2.0, 2.0] \times [-1.0, 1, 0] \Rightarrow (y_1 > y_2)$.

3 Proof Generation for DNN Verification

All of the major DNN verification approaches including: $\alpha\beta$-`CROWN` [7], `NeuralSAT` [15], `MN-BaB` [6], `OVAL` [26], `nnenum` [16], and `Marabou` [27], share a common "branch and bound" (BaB) search structure: (i) (branch) split into smaller subproblems by using *neuron splitting*, which decides boolean values representing neuron activation status, and (ii) (bound) use abstraction and LP solving to approximate bounds on neuron values to determine the satisfiability of the partial activation pattern formed by the split. We leverage this commonality to bring proof generation capabilities with minimal overhead to existing DNN verification tools.

In this paper we focus on checking proofs of unsatisfiability (unsat). A counterexample, c, returned by a verifier is an input that is purported to violate the property. This constitutes a proof of satisfiability (sat) and can easily be checked by evaluating $\phi(c, N(c))$. In contrast, unsat proof, which explains why *no possible inputs* can violate the property, is inherently more complex to generate (§3), requires a more sophisticated encoding (§3.2), and an efficient checking algorithm (§4).

3.1 Neuron-Splitting DNN Verification

Alg. 1 illustrates $\texttt{BaB}_{\texttt{NV}}$, a reference architecture [28] for modern DNN verifiers based on the branch-and-bound (BaB) framework. $\texttt{BaB}_{\texttt{NV}}$ takes as input a ReLU-based DNN and a property of interest. It iteratively alternates between two core components: `Decide` (line 6), which performs neuron-splitting by assigning an activation status (active/inactive) to a neuron, and `Deduce` (line 5), which checks the feasibility of the current activation pattern and prunes infeasible branches.

Our key insight is that the BaB architecture of $\texttt{BaB}_{\texttt{NV}}$ naturally supports proof generation. To realize this, we augment $\texttt{BaB}_{\texttt{NV}}$ with a proof tree structure, stored in the `proof` variable (line 2). We also instrument $\texttt{BaB}_{\texttt{NV}}$ so that each branching decision made during the `Decide` step is explicitly recorded into this tree (line 10). Each node in the binary proof tree represents a neuron, and its left and right children correspond to the two possible activation decisions (active or inactive).

Example Fig. 1a illustrates a DNN and how $\texttt{BaB}_{\texttt{NV}}$ determines unsatisfiability (i.e., verifies the problem) and generates the unsat proof in Fig. 1b. First, $\texttt{BaB}_{\texttt{NV}}$ initializes the activation pattern set

`ActPatterns` with an empty activation pattern $\emptyset$. Then $\texttt{BaB}_{\texttt{NV}}$ enters a loop (line 3-line 10) to search for a satisfying assignment or a proof of unsatisfiability.

1st iteration: $\texttt{BaB}_{\texttt{NV}}$ selects the only available activation pattern $\emptyset \in$ `ActPatterns`, and calls `Deduce` to check the feasibility of the problem based on the current activation pattern. `Deduce` uses abstraction to approximate that from the input constraints the output values are feasible for the given network. Since `Deduce` cannot determine infeasibility, $\texttt{BaB}_{\texttt{NV}}$ invokes `Decide` to randomly select a neuron to split. Suppose it selects neuron v_4, which results in the original problem being divided into two independent subproblems: one where v_4 is active, and another where v_4 is inactive. $\texttt{BaB}_{\texttt{NV}}$ then adds v_4 and $\overline{v_4}$ to `ActPatterns`.

2nd iteration: $\texttt{BaB}_{\texttt{NV}}$ has two subproblems that can be processed in parallel. For the first subproblem with v_4, `Deduce` cannot decide infeasibility, so it selects v_2 to split. It then conjoins v_4 with v_2 and then with $\overline{v_2}$ and adds both conjuncts to `ActPatterns`. For the second subproblem with v_4 inactive (i.e., $\overline{v_4}$), `Deduce` determines that the problem is unsatisfiable and $\texttt{BaB}_{\texttt{NV}}$ saves $\overline{v_4}$ to the proof tree, as node 3, to indicate one unsatisfiable pattern, i.e., whenever the network has v_4 being inactive, the problem is unsatisfiable.

3rd iteration: $\texttt{BaB}_{\texttt{NV}}$ has two subproblems for $v_4 \wedge v_2$ and $v_4 \wedge \overline{v_2}$. For the first subproblem, `Deduce` cannot decide infeasibility, so it selects v_1 to split. It then conjoins v_1 and then $\overline{v_1}$ to the current activation pattern and adds them to `ActPatterns`. For the second one, `Deduce` determines that the problem is unsatisfiable and $\texttt{BaB}_{\texttt{NV}}$ saves the $v_4 \wedge \overline{v_2}$ to the proof tree, as node 5.

4th iteration: $\texttt{BaB}_{\texttt{NV}}$ has two subproblems for $v_4 \wedge v_2 \wedge v_1$ and $v_4 \wedge v_2 \wedge \overline{v_1}$. Both subproblems are determined to be unsatisfiable, and $\texttt{BaB}_{\texttt{NV}}$ saves them to the proof tree as nodes 6 and 7, respectively.

Finally, $\texttt{BaB}_{\texttt{NV}}$ has an empty `ActPatterns`, stops the search, and returns `unsat` and the proof tree.

The `APTP` proof tree The resulting proof tree has a specific structure. First, it is a binary tree where each parent node must have children for both activation status values of a neuron. Second, it is a *proof tree* that captures unsatisfiability reasoning, i.e., each leaf holds the constraint showing the activation pattern encoded from the root to this leaf results in unsatisfiability. The tree in Fig. 1b demonstrates this structure. Each white node corresponds to a branching node where $\texttt{BaB}_{\texttt{NV}}$ makes decisions to split neurons. The red leaves correspond to the unsatisfiable patterns that are saved to the proof tree. Note that a leaf node implies the unsatisfiability of the sub-tree rooted at the leaf, e.g., node 3 encodes the unsatisfiability of a set of 8 activation patterns.

We leverage this structure to store the proof in the `APTP` format (§3.2) and to check it using the `APTPchecker` algorithm (§4).

3.2 The `APTP` Proof Language

We have shown that the broad class of $\texttt{BaB}_{\texttt{NV}}$ DNN verification techniques can generate a binary tree that represents a proof of unsatisfiability (§3). We define a standard proof format for specifying DNN proofs, `APTP`, that is human-readable, compact, and can be efficiently generated by verification tools and processed by proof checkers. `APTP` is inspired by the SMTLIB format [29] used for SMT solving, which has also been adopted by the VNNLIB language [30] to specify DNN verification problems.

Fig. 2a presents the syntax of `APTP`. A proof consists of *declarations* and *assertions*. Declarations define input/output variables (real numbers) and hidden variables (with PWL activations like ReLU). Assertions encode preconditions over inputs and postconditions over outputs using logical formulas with comparisons and Boolean operators like `and` and `or`. More details on the syntax and semantics of `APTP` are available in (Apdx. A).

Example The proof in Fig. 2b corresponds to the proof tree in Fig. 1b. The statement `(and (< N_4 0))` corresponds to the rightmost path of the tree with $\overline{v_4}$ decision (leaf 3). The statement `(and (< N_2 0) (>= N_4 0))` corresponds to the path with $v_4 \wedge \overline{v_2}$ (leaf 5).

The `APTP` language is intentionally designed to (a) omit explicit weights and biases to reduce the size of the proof structure, and (b) explicitly encode a DNF structure to enable easy parallelization. The weights and biases of the DNN are already recorded in the ONNX format [31], which serves as a standard input to both verification tools and `APTP` checkers, like the one we describe in §4.

⟨*proof*⟩ ::= ⟨*declarations*⟩ ⟨*assertions*⟩
⟨*declarations*⟩ ::= ⟨*declaration*⟩ | ⟨*declaration*⟩ ⟨*declarations*⟩
⟨*declaration*⟩ ::= (**declare-const** ⟨*input-vars*⟩ **Real**)
| (**declare-const** ⟨*output-vars*⟩ **Real**)
| (**declare-pwl** ⟨*hidden-vars*⟩ ⟨*activation*⟩)
⟨*input-vars*⟩ ::= ⟨*input-var*⟩ | ⟨*input-var*⟩ ⟨*input-vars*⟩
⟨*output-vars*⟩ ::= ⟨*output-var*⟩ | ⟨*output-var*⟩ ⟨*output-vars*⟩
⟨*hidden-vars*⟩ ::= ⟨*hidden-var*⟩ | ⟨*hidden-var*⟩ ⟨*hidden-vars*⟩
⟨*activation*⟩ ::= ReLU | Leaky ReLU | . . .
⟨*assertions*⟩ ::= ⟨*assertion*⟩ | ⟨*assertion*⟩ ⟨*assertions*⟩
⟨*assertion*⟩ ::= (**assert** ⟨*formula*⟩)
⟨*formula*⟩ ::= (⟨*operator*⟩ ⟨*term*⟩ ⟨*term*⟩)
| (**and** ⟨*formula*⟩+) | (**or** ⟨*formula*⟩+)
⟨*term*⟩ ::= ⟨*input-var*⟩ | ⟨*output-var*⟩
| ⟨*hidden-var*⟩ | ⟨*constant*⟩
⟨*operator*⟩ ::= $<$ | $\leq$ | $>$ | $\geq$
⟨*input-var*⟩ ::= X_⟨*constant*⟩
⟨*output-var*⟩ ::= Y_⟨*constant*⟩
⟨*hidden-var*⟩ ::= N_⟨*constant*⟩
⟨*constant*⟩ ::= **Int** | **Real**

(a) The APTP proof language.

```
; Declare variables
(declare-const X_0 Real)
(declare-const X_1 Real)
(declare-const Y_0 Real)
(declare-const Y_1 Real)
(declare-pwl N_1 ReLU)
(declare-pwl N_2 ReLU)
(declare-pwl N_3 ReLU)
(declare-pwl N_4 ReLU)
; Input constraints
(assert (>= X_0 -2.0))
(assert (<= X_0  2.0))
(assert (>= X_1 -1.0))
(assert (<= X_1  1.0))
; Output constraints
(assert (<= Y_0 Y_1))
; Hidden constraints
(assert (or
  (and (<  N_4 0))
  (and (<  N_2 0)
       (>= N_4 0))
  (and (>= N_2 0)
       (>= N_1 0)
       (>= N_4 0))
  (and (>= N_2 0)
       (<  N_1 0)
       (>= N_4 0))))
```

(b) APTP example.

Fig. 2: The APTP format.

Alg. 2. APTPchecker algorithm.

input : DNN $\mathcal{N}$, property $\phi_{in} \Rightarrow \phi_{out}$, proof
output : certified if proof is valid, otherwise uncertified

1 **if** ¬ RepOK (proof) **then** RaiseError(*Invalid proof tree*)
2 model ← CreateStabilizedMILP($\mathcal{N}, \phi_{in}, \phi_{out}$) // initialize MILP model with inputs
3 **while** proof **do**
4   node ← Select(proof) // get node to check
5   model ← AddConstrs(model, node) // add corresponding constraints
6   **if** CheckFeasibility(model) **then**
7   **return** uncertified // cannot certify
8 **return** certified

4 Checking APTP Proofs

We introduce a proof checker, APTPchecker, that validates APTP proofs. The checker is verifier-independent and supports APTP proofs generated by different verification tools. It is also efficient and scales to handle large proof trees.

4.1 The Core APTPchecker Algorithm

The goal of APTPchecker is to verify that the APTP tree generated by a DNN verification tool is correct (i.e., the proof tree is a proof of unsatisfiability of the DNN verification problem). APTPchecker thus must verify that the constraint represented by each *leaf* node in the proof tree is unsatisfiable. To check each node, APTPchecker forms an MILP problem (§4.1.1) consisting of the constraint in Eq. 1 (the DNN, the input condition, and the negation of the output) with the constraints representing the activation pattern encoded by the tree path to the leaf node. APTPchecker then invokes an LP solver to check that the MILP problem is infeasible, which indicates unsatisfiability of the leaf node.

Core Algorithm Alg. 2 shows a minimal (core) APTPchecker algorithm, which takes as input a DNN $\mathcal{N}$, a property $\phi_{in} \Rightarrow \phi_{out}$, a proof tree proof, and returns certified if the proof tree is valid and uncertified otherwise. APTPchecker first checks the validity of the proof tree (line 2), i.e., the input must represent a proper APTP proof tree (§3.2). If the proof tree is invalid, APTPchecker raises an error. APTPchecker next creates a MILP model (line 2) representing the input. APTPchecker then enters a loop (line 3) that selects a (random) leaf node from the proof tree (line 4) and adds its MILP constraint to the model (line 5). It then checks the model using an LP solver to determine whether the leaf node is unsatisfiable. If the LP solver returns feasibility, APTPchecker returns uncertified, i.e., it cannot verify the input proof tree. APTPchecker continues until all leaf nodes are checked and returns certified, indicating the proof tree is valid.

Example For the APTP proof in Fig. 2b, we need to check that the four leaf nodes 3, 5, 6, and 7 of the proof tree in Fig. 1b are unsatisfiable. Assume APTPchecker first selects node 3, it forms the MILP problem for leaf node 3 by conjoining the constraint representing $0.6v_1 + 0.9v_2 - 0.1 \leq 0$ (i.e., $\overline{v_4}$) with the constraints in Eq. 1 representing the input ranges and the DNN with the objective of optimizing the output. APTPchecker then invokes an LP solver, which determines that this MILP is infeasible, i.e., leaf node 3 indeed leads to unsatisfiability. APTPchecker continues this process for the other three leaf nodes and returns certified as all leaf nodes are unsatisfiable.

Implementation and Validation APTPchecker is written in Python, and uses Gurobi [32] for LP solving. The core APTPchecker algorithm (Alg. 2) consists of 600 SLOC, while optimizations use an additional 200 SLOC. Currently, APTPchecker supports ReLU-based feed-forward (FNNs) and convolutional neural networks (CNNs). APTPchecker uses ONNX for neural networks and outputs APTP proofs. In addition, we used the CrossHair [33] symbolic execution tool to check the correctness of the core algorithm in APTPchecker. Specifically, CrossHair confirmed that key postconditions hold, e.g., that APTPchecker returns `certified` if and only if all leaf nodes in the proof tree are formally proven. While the verification is not exhaustive (CrossHair only explore program paths up to a certain depth), this increases confidence in the implementation's correctness up to certain depth. A detailed discussion is provided in Apdx. B.

4.1.1 MILP Formulation

APTPchecker formulates MILP problems [34] and checks for feasible solutions using off-the-shelf LP solving. Formally, the MILP problem is defined as:

$$\begin{aligned}
&(a)\quad z^{(i)} = W^{(i)}\hat{z}^{(i-i)} + b^{(i)}; &&(b)\quad y = z^{(L)}; x = \hat{z}^{(0)};\\
&(c)\quad \hat{z}_j^{(i)} \geq z_j^{(i)}; \hat{z}_j^{(i)} \geq 0; &&(d)\quad a_j^{(i)} \in \{0,1\};\\
&(e)\quad \hat{z}_j^{(i)} \leq a_j^{(i)} u_j^{(i)}; \hat{z}_j^{(i)} \leq z_j^{(i)} - l_j^{(i)}(1 - a_j^{(i)});
\end{aligned} \tag{2}$$

where x is input, y is output, and $z^{(i)}$, $\hat{z}^{(i)}$, $W^{(i)}$, and $b^{(i)}$ are the pre-activation, post-activation, weight, and bias vectors for layer i, respectively. This encodes precisely the semantics of a ReLU-based DNN: (a) the affine transformation computing the pre-activation value for a neuron; (b) the inputs and outputs in the DNN; (c) assertion that post-activation values are non-negative and no less than pre-activation values; (d) neuron activation status indicator variables that are either 0 or 1; and (e) constraints on the upper, $u_j^{(i)}$, and lower, $l_j^{(i)}$, bounds of the pre-activation value of the jth neuron in the ith layer. Deactivating a neuron, $a_j^{(i)} = 0$, simplifies the first of the (e) to $\hat{z}_j^{(i)} \leq 0$, and activating a neuron simplifies the second to $\hat{z}_j^{(i)} \leq z_j^{(i)}$, which is consistent with $\hat{z}_j^{(i)} = max(z_j^{(i)}, 0)$.

4.1.2 Correctness

Alg. 2 returns certified iff the input APTP proof tree is unsatisfiable. This proof tree encodes a disjunction of constraints, one per tree path, where each constraint represents an activation pattern of the network (the leaf node). Then each problem is reduced to a simple LP that exactly captures the semantics of the DNN for a specific activation pattern and thus, the algorithm introduces no approximations, i.e., it is sound and complete.

Note that this correctness argument assumes that the LP solver is correct—in practice multiple solvers could be used to guard against errors in that component. It is standard for proof checkers to assume the correctness of a small set of external tools, e.g., checkers that use theorem provers assume the correctness of the underlying prover [35].

Tab. 1: Benchmarks consist of a 8 neural networks comprised of varying numbers of CNN (C) and FNN (F) layers, neurons, and parameters, each paired with 25 properties to form UNSAT verification instances.

Name	Networks				Properties
	Num.	Layers	Neurons	Param.	Num.
CNN	8	1-2C;1F	320-3920	41K-180K	200
FNN	8	2-6F	64-3072	27K-1.7M	200

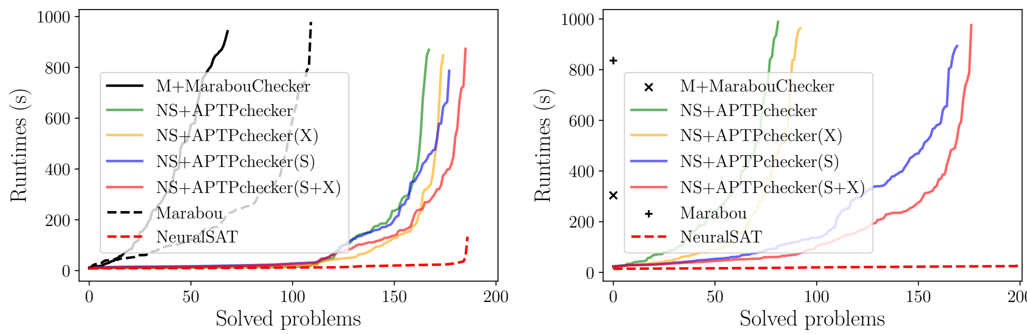


Fig. 3: Cactus plots for verifiers and proof checkers of FNN (left) and CNN (right) benchmarks.

4.1.3 Optimizations

Our `APTPchecker` implementation employs several optimizations to improve efficiency, especially for large proof trees. It uses *neuron stabilization* to identify stable neurons (either active or inactive) and replace disjunctive constraints with linear ones, and simplifying the MILP problem and reducing the work of the LP solver. Additionally, it employs *pruning of leaf nodes* and backtracking to check parent nodes only when necessary, reducing the number of LP problems to be solved. Finally, `APTPchecker` leverages the tree structure of `APTP` proof to *parallelize* the checking of leaf nodes, making the verification process scale better to large proof trees. Additional details on these optimizations are available in Apdx. C.

5 Evaluation

We evaluate our work using the following research questions: **RQ1** (§5.1): How does `APTPchecker` perform and compare prior work? **RQ2** (§5.2): How does proof checking performance vary with verification algorithms and optimizations?

Benchmarks We evaluate on UNSAT verification problems selected from the harder benchmark suite introduced in [8], which includes ACAS Xu, RESNET_A/B, CIFAR2020, MNISTFC, and MNIST_GDVB. As with prior work [5] we exclude ACAS Xu, which has networks with very low input dimensions and did not even need to use BaB on activation space to be solved. We also exclude RESNET, which are unsupported by the `APTPchecker`. This is a straightforward engineering limitation and there is no fundamental reason the checking algorithm is not applicable. From CIFAR2020, we selected CNN models with varied convolutional sizes and depths; from MNISTFC and MNIST_GDVB, we chose 8 FNNs of diverse sizes. For each network, we randomly sampled local robustness properties until we obtained 25 UNSAT instances, yielding 200 CNN and 200 FNN problems (400 total) as shown in Tab. 1.

Baselines The only prior DNN proof checker [36] focuses on the `Marabou` verifier. In contrast, we adapted two verifiers: $\alpha\beta$-`CROWN` and `NeuralSAT`, to generate `APTP` proofs. RQ1 is on proof checking performance, so we compare `Marabou` and its proof checker with `APTPchecker` using `NeuralSAT`. RQ2 compares optimized vs. unoptimized `APTPchecker`. RQ3 evaluates how `APTPchecker` accommodates proofs generated by different verification algorithm variants.

Metrics To assess performance we use the two common metrics in the verification community: time to solve and number of problems solved. We record time to verify, generate, and check proofs, using

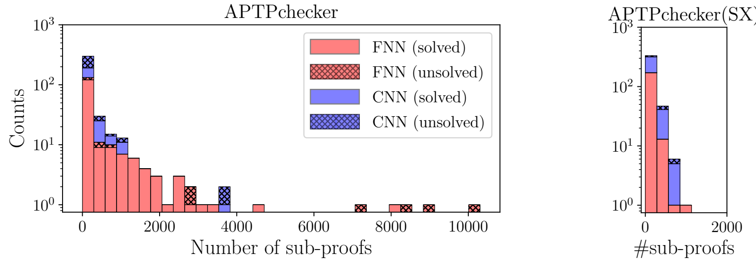


Fig. 4: Number of sub-proofs per problem with (right) and without (left) `APTPchecker` optimizations.

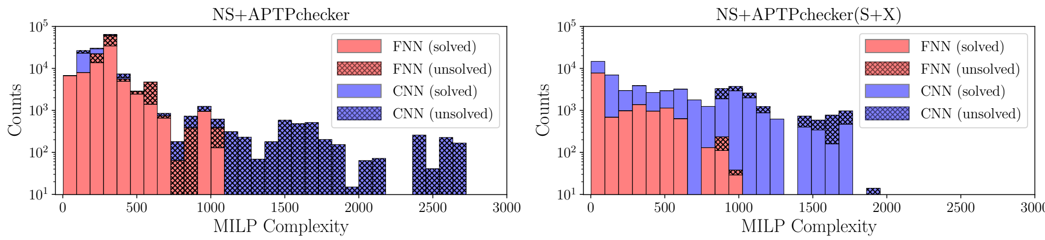


Fig. 5: Number of constraints per problem with (right) and without (left) `APTPchecker` optimizations.

a 1000s timeout. A problem is "solved" if all steps complete within time. We report cactus plots (Fig. 3), comparing runtime and problem count. We also measure proof size (number of sub-proofs) and MILP complexity as the number of neurons that do *not* have a fixed value, i.e., the number of unstable neurons.

Setup All experiments were run on a Linux machine with an AMD Threadripper 64-core 4.2GHZ CPU, 128GB RAM, and an NVIDIA GeForce RTX 4090 GPU with 24 GB VRAM.

5.1 RQ1 : Proof Checking Performance

Fig. 3 presents data on the performance of `APTPchecker` relative to both an underlying verifier, `NeuralSAT`, and prior work on neural network proof checking, `Marabou`'s proof checker. In cactus plots like this, lines that extend further on the x-axis are better – more problems solved – and lines that are lower are better – faster solve times. Another way to view these is to pick a point on the x-axis where the plots for two techniques are defined and think of the areas under the two curves as the "total cost" to solve that number of problems. The dashed lines show the performance of the verifier and the solid lines show the performance of the verifier, proof generation, and the proof checker. Several configurations of `APTPchecker` are shown, but in this RQ we draw the readers attention to the plots for the `APTPchecker`(S+X) configurations; the rest are discussed in detail below.

The cactus plot for the FNN benchmark (left) shows that `Marabou` and its checker are able to solve 69 problems or 35% of the benchmark, whereas `APTPchecker` can solve 186 or 93%. For the CNN benchmark (left) `Marabou` and its checker can solve a single benchmark, whereas `APTPchecker` can solve 177 problems or 89%. In total, `APTPchecker` solved 363 problems or 91%, whereas `Marabou` solved 70 problems or 18% of all instances.

The shape of these cactus plots indicates a high-degree of variability in the cost of proof checking relative to verification. From Alg. 2 it is clear that both the number of leaves in the tree structure, line 4, and the complexity of the model to be checked, line 6, are factors that contribute to the cost of proof checking. To explore those factors we plot their variation across the benchmarks when running `APTPchecker`.

Fig. 4 (left) plots a histogram of the number of sub-proofs solved per verification problem, i.e., the number of nodes of the proof tree. When interpreting these plots, understand that the y-axis log scale means that vertical distances have a different meaning as you move upward in the plot. While the vast majority of the verification problems have proof trees of fewer then 2000 leaves, but 17 of them have larger trees up to a maximum of more than 10000 leaves. Note also that even among the smaller

Tab. 2: Proof statistics for best verifier configurations.

Verifier	**Num. Sub-Proofs** Mean	Median	**MILP Complexity** Mean	Median
`NeuralSAT(S)`	95	36	601	545
$\alpha\beta$-`CROWN`	230	180	414	179

sized proof trees, there are some problems that cannot be solved. This is due to complexity of solving the MILP constraints at the leaves of those proof trees.

Fig. 5 (left) plots a histogram of the number of occurrences of MILP problems of a given complexity across the benchmarks. Here again we see a spread in data, but unlike with the number of sub-proofs the CNN benchmarks seem to have consistently larger constraints and there is a clear bias among the unsolved problems towards larger constraint size. To optimize proof checking, we must address both of these sources of complexity.

5.2 RQ2 : Proof Checking and Verifier Optimizations

Fig. 6 shows cactus plots for two configurations of `NeuralSAT` and $\alpha\beta$-`CROWN` generated proofs across the benchmarks. The performance of the verifiers (dashed lines) differ across configurations and they are able to verify between 337 and 400 problems. For both of the verifiers and configurations, `APTPchecker` is able to check between 93.7% and 99.4% of the proofs that are generated. This demonstrates that the `APTP` is able to encode proofs generated by differing neural network verification algorithms, and that `APTPchecker` can check them.

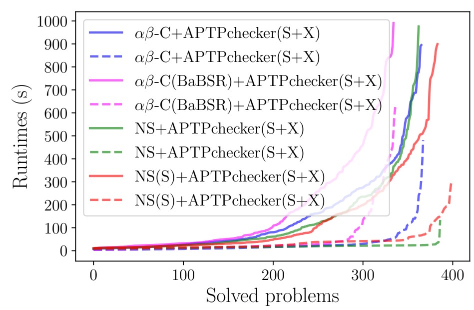


Fig. 6: `APTPchecker` runtimes.

We analyzed both the number of sub-proofs and MILP complexity for the proofs generated by the two best performing verifier configurations. These values follow a skewed distribution, so we report the mean and median in Tab. 2. The proof structures vary between verifiers: `NeuralSAT` produces smaller proof trees, but with more complex MILP problems. In contrast, $\alpha\beta$-`CROWN` generates significantly larger proof trees, but with simpler MILP problems. This variation suggests directions for future work, such as enabling `NeuralSAT` to generate larger proof trees with simpler MILPs for better parallelization, or adopting fast verification during development and switching to proof-friendly strategies once all properties are verified.

6 Conclusion and Future Work

We introduce a proof format `APTP` which can express proofs generated by state-of-the-art DNN verifiers. To check proofs in this format, we design the `APTPchecker` algorithm and prove it, and its optimizations, correct. We believe these contributions enable the community to take a critical step toward *certifiable and reliable neural network verification*, closing the gap between practical verification algorithms and the assurance required for deployment in real-world AI systems.

Limitations `APTPchecker` can be made more scalable. Better parallel checking strategies couple apply stabilization optimization to sub-proof trees instead of the whole proof tree, potentially identifying more stable neurons. Moreover, `APTP`'s tree structure naturally lends itself to incremental solving, and while solvers like Gurobi do not support this, new frameworks are emerging to support optimized solving of related MILP problems [37].

Potential Negative Societal Impact The research line on DNN verification can exploited to find issues in DNNs and this work, which aims to improve DNN verification, indirectly supports that. However, DNN verification, and therefore this work, also helps to ensure that DNNs are safe and secure for deployment in critical applications.

References

[1] Nuro.ai, "Nuro Driver," 2024. https://www.nuro.ai/technology.

[2] B. Kovatchev, A. Castillo, E. Pryor, L. L. Kollar, C. L. Barnett, M. D. DeBoer, and S. A. a. Brown, "Neural-net artificial pancreas: A randomized crossover trial of a first-in-class automated insulin delivery algorithm," *Diabetes Technology & Therapeutics*, vol. 26, no. 6, pp. 375–382, 2024.

[3] D. Hanover, A. Loquercio, L. Bauersfeld, A. Romero, R. Penicka, Y. Song, G. Cioffi, E. Kaufmann, and D. Scaramuzza, "Autonomous drone racing: A survey," *IEEE Transactions on Robotics*, 2024.

[4] C. Brix, S. Bak, C. Liu, and T. T. Johnson, "The Fourth International Verification of Neural Networks Competition (VNN-COMP 2023): Summary and Results," 2023.

[5] H. Zhang, S. Wang, K. Xu, L. Li, B. Li, S. Jana, C.-J. Hsieh, and J. Z. Kolter, "General cutting planes for bound-propagation-based neural network verification," *arXiv preprint arXiv:2208.05740*, 2022.

[6] C. Ferrari, M. N. Mueller, N. Jovanović, and M. Vechev, "Complete Verification via Multi-Neuron Relaxation Guided Branch-and-Bound," in *International Conference on Learning Representations*, 2022.

[7] S. Wang, H. Zhang, K. Xu, X. Lin, S. Jana, C.-J. Hsieh, and J. Z. Kolter, "Beta-crown: Efficient bound propagation with per-neuron split constraints for neural network robustness verification," *Advances in Neural Information Processing Systems*, vol. 34, pp. 29909–29921, 2021.

[8] H. Duong, D. Xu, T. Nguyen, and M. B. Dwyer, "Harnessing neuron stability to improve dnn verification," *Proc. ACM Softw. Eng.*, vol. 1, jul 2024.

[9] N. Wetzler, M. J. Heule, and W. A. Hunt Jr, "Drat-trim: Efficient checking and trimming using expressive clausal proofs," in *International Conference on Theory and Applications of Satisfiability Testing*, pp. 422–429, Springer, 2014.

[10] F. Pollitt, M. Fleury, and A. Biere, "Faster lrat checking than solving with cadical," in *26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023)*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.

[11] P. Lammich, "Efficient verified (un) sat certificate checking," *Journal of Automated Reasoning*, vol. 64, no. 3, pp. 513–532, 2020.

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

[13] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Towards proving the adversarial robustness of deep neural networks," *Proc. 1st Workshop on Formal Verification of Autonomous Vehicles (FVAV), pp. 19-26*, 2017.

[14] M. J. Kochenderfer, J. E. Holland, and J. P. Chryssanthacopoulos, "Next-generation airborne collision avoidance system," tech. rep., Massachusetts Institute of Technology-Lincoln Laboratory Lexington United States, 2012.

[15] H. Duong, T. Nguyen, and M. Dwyer, "A DPLL(T) Framework for Verifying Deep Neural Networks," *arXiv preprint arXiv:2307.10266*, 2024.

[16] S. Bak, "nnenum: Verification of relu neural networks with optimized abstraction refinement," in *NASA Formal Methods Symposium*, pp. 19–36, Springer, 2021.

[17] L. Cruz-Filipe, M. J. Heule, W. A. Hunt, M. Kaufmann, and P. Schneider-Kamp, "Efficient certified rat verification," in *Automated Deduction–CADE 26: 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6–11, 2017, Proceedings*, pp. 220–236, Springer, 2017.

[18] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, Springer, 2008.

[19] T. Bouton, D. Caminha B. de Oliveira, D. Déharbe, and P. Fontaine, "verit: an open, trustable and efficient smt-solver," in *International Conference on Automated Deduction*, pp. 151–156, Springer, 2009.

[20] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani, "The mathsat5 smt solver," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 93–107, Springer, 2013.

[21] J. Christ, J. Hoenicke, and A. Nutz, "Smtinterpol: An interpolating smt solver," in *International SPIN Workshop on Model Checking of Software*, pp. 248–254, Springer, 2012.

[22] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, "cvc5: A versatile and industrial-strength smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems* (D. Fisman and G. Rosu, eds.), (Cham), pp. 415–442, Springer International Publishing, 2022.

[23] O. Isac, C. Barrett, M. Zhang, and G. Katz, "Neural network verification with proof production," *Proc. 22nd Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, 2022.

[24] R. Desmartin, O. Isac, G. Passmore, K. Stark, G. Katz, and E. Komendantskaya, "Towards a certified proof checker for deep neural network verification," 2024.

[25] G. Passmore, S. Cruanes, D. Ignatovich, D. Aitken, M. Bray, E. Kagan, K. Kanishev, E. Maclean, and N. Mometto, "The imandra automated reasoning system (system description)," in *Automated Reasoning: 10th International Joint Conference, IJCAR 2020, Paris, France, July 1–4, 2020, Proceedings, Part II 10*, pp. 464–471, Springer, 2020.

[26] OVAL-group, "OVAL - Branch-and-Bound-based Neural Network Verification," 2023. https://github.com/oval-group/oval-bab.

[27] G. Katz, D. A. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljić, *et al.*, "The marabou framework for verification and analysis of deep neural networks," in *International Conference on Computer Aided Verification*, pp. 443–452, Springer, 2019.

[28] E. Y. Nakagawa, M. Guessi, J. C. Maldonado, D. Feitosa, and F. Oquendo, "Consolidating a process for the design, representation, and evaluation of reference architectures," in *2014 IEEE/IFIP Conference on Software Architecture*, pp. 143–152, IEEE, 2014.

[29] C. Barrett, A. Stump, C. Tinelli, *et al.*, "The smt-lib standard: Version 2.0," in *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, England)*, vol. 13, p. 14, 2010.

[30] A. Tacchella, L. Pulina, D. Guidotti, and S. Demarchi, "The international benchmarks standard for the Verification of Neural Networks," 2023.

[31] J. Bai, F. Lu, and K. Zhang, "ONNX Open neural network exchange."

[32] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2022.

[33] CrossHair, "An analysis tool for Python that blurs the line between testing and type systems. ," 2025. https://github.com/pschanely/CrossHair.

[34] V. Tjeng, K. Y. Xiao, and R. Tedrake, "Evaluating robustness of neural networks with mixed integer programming," in *International Conference on Learning Representations*, 2019.

[35] P. Lammich, "Grat: a formally verified (un) sat proof checker," *SAT COMPETITION 2023*, p. 82, 2023.

[36] R. Desmartin, O. Isac, G. Passmore, K. Stark, E. Komendantskaya, and G. Katz, "Towards a certified proof checker for deep neural network verification," in *International Symposium on Logic-Based Program Synthesis and Transformation*, pp. 198–209, Springer, 2023.

[37] The JuMP Projet, "JuMP," 2024.

[38] M. R. Garey and D. S. Johnson, *Computers and intractability*, vol. 174. Freeman San Francisco, 1979.

[39] C. Barrett, L. De Moura, and P. Fontaine, "Proofs in satisfiability modulo theories," *All about proofs, Proofs for all*, vol. 55, no. 1, pp. 23–44, 2015.

[40] S. Conchon, A. Mebsout, and F. Zaïdi, "Certificates for parameterized model checking," in *FM 2015: Formal Methods: 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings 20*, pp. 126–142, Springer, 2015.

[41] A. Griggio, M. Roveri, and S. Tonetta, "Certifying proofs for sat-based model checking," *Formal Methods in System Design*, vol. 57, no. 2, pp. 178–210, 2021.

[42] A. Van Gelder, “Producing and verifying extremely large propositional refutations: Have your cake and eat it too,” *Annals of Mathematics and Artificial Intelligence*, vol. 65, pp. 329–372, 2012.

[43] E. Goldberg and Y. Novikov, “Verification of proofs of unsatisfiability for cnf formulas,” in *2003 Design, Automation and Test in Europe Conference and Exhibition*, pp. 886–891, IEEE, 2003.

[44] M. J. Heule, W. A. Hunt, and N. Wetzler, “Trimming while checking clausal proofs,” in *2013 Formal Methods in Computer-Aided Design*, pp. 181–188, IEEE, 2013.

[45] M. J. Heule, “The drat format and drat-trim checker,” *arXiv preprint arXiv:1610.06229*, 2016.

[46] H. Barbosa, A. Reynolds, G. Kremer, H. Lachnitt, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Viswanathan, S. Viteri, *et al.*, “Flexible proof production in an industrial-strength smt solver,” in *International Joint Conference on Automated Reasoning*, pp. 15–35, Springer International Publishing Cham, 2022.

[47] L. De Moura and N. Bjørner, “Satisfiability modulo theories: introduction and applications,” *Communications of the ACM*, vol. 54, no. 9, pp. 69–77, 2011.

[48] G. C. Necula, *Compiling with proofs*. Carnegie Mellon University, 1998.

[49] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner, “A modular integration of sat/smt solvers to coq through proof witnesses,” in *International Conference on Certified Programs and Proofs*, pp. 135–150, Springer, 2011.

[50] S. Böhme, A. C. Fox, T. Sewell, and T. Weber, “Reconstruction of z3’s bit-vector proofs in hol4 and isabelle/hol,” in *Certified Programs and Proofs: First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings 1*, pp. 183–198, Springer, 2011.

[51] S. Böhme and T. Weber, “Fast lcf-style proof reconstruction for z3,” in *Interactive Theorem Proving: First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings 1*, pp. 179–194, Springer, 2010.

[52] H.-J. Schurr, M. Fleury, and M. Desharnais, “Reliable reconstruction of fine-grained proofs in a proof assistant.,” in *CADE*, vol. 28, pp. 450–467, 2021.

[53] S. Kan, A. W. Lin, P. Rümmer, and M. Schrader, “Certistr: a certified string solver,” in *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pp. 210–224, 2022.

[54] B. Andreotti, H. Lachnitt, and H. Barbosa, “Carcara: An efficient proof checker and elaborator for smt proofs in the alethe format,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 367–386, Springer, 2023.

[55] H.-J. Schurr, M. Fleury, H. Barbosa, and P. Fontaine, “Alethe: Towards a generic smt proof format,” *arXiv preprint arXiv:2107.02354*, 2021.

[56] R. J. Vanderbei, “Linear programming: foundations and extensions,” *Journal of the Operational Research Society*, vol. 49, no. 1, pp. 94–94, 1998.

A Syntax and Grammar of APTP

Fig. 2a in §3.2 outlines the APTP syntax and grammar, represented as production rules. A proof is composed of *declarations* and *assertions*. Declarations define the variables and their types within the proof. Specifically, *input variables* (prefixed with X) and *output variables* (prefixed with Y) are declared as real numbers, representing the inputs and outputs of the network. Additionally, *hidden variables* are declared with specific piece-wise linear (PWL) activation functions, such as ReLU. These hidden variables correspond to the internal nodes of the neural network that process the input data through various activation functions.

Assertions are logical statements that specify the conditions or properties that must hold within the proof. Assertions over input variables are *preconditions* and those over output variables are *post-conditions*. Each assertion is composed of a *formula*, which can involve terms and logical operators. Formulas include simple comparisons between terms (e.g., less than, greater than) or more complex logical combinations using `and` and `or` operators. The terms used in these formulas can be variables or constants.

The `declare-*` statements declare input, output, and hidden variables, while the `assert` statements specify the constraints on these variables (i.e., the pre and postcondition of the desired property). The hidden constraints represent the activation patterns of the hidden neurons in the network (i.e., the proof tree). Each `and` statement represents a tree path that represents an activation pattern.

B Correctness of APTPchecker Implementation

We were able to verify the implementation of the core APTPchecker algorithm (§4.1) using the CrossHair [33] symbolic execution for upto certain thresholds (e.g., timeout per condition `per_condition_timeout=10`).

To perform such analysis, we need to create a simplified version of APTPchecker[1] including: (1) No optimization – remove all optimizations in Apdx. C; (2) Assume that Gurobi (MIP) is correct, therefore, the condition indicating whether MIP is correct or not must be made up (e.g., $sum(n) \geq 0$ – summation of all literals (e.g., variable and branch condition) in a leaf node); and (3) Add pre- and post-conditions to the main function. This make APTPchecker codebase minimal with just about 100 LoC. In particular, some pre- and post- conditions are listed in Listing 1.

```
"""
pre: isinstance(proof, list)
pre: all(isinstance(p, list) for p in proof)
post: _ in {CERTIFIED, UNCERTIFIED}
post: (_ == UNCERTIFIED) == (any(sum(n) < 0 for n in proof))
post: (_ == CERTIFIED) == (all(sum(n) >= 0 for n in proof))
"""
```

Listing 1: Pre- and Post- Conditions for CrossHair

CrossHair outputs are shown in Listing 2.

```
attempt_call() Postcondition confirmed.
analyze_calltree() Path tree stats {CONFIRMED:58}
analyze_calltree() Iter complete. Worst status found so far: UNKNOWN
analyze_calltree() Exceeded condition timeout, stopping
analyze_calltree() Aborted calltree search with UNKNOWN and 0 messages. Number of
    iterations: 58
analyze_class() Analyzing class ProofReturnStatus
condition_parser() Using parsers: (AnalysisKind.PEP316, AnalysisKind.icontract,
    AnalysisKind.deal)
analyze_class() Analyzing class ProofTree
condition_parser() Using parsers: (AnalysisKind.PEP316, AnalysisKind.icontract,
    AnalysisKind.deal)
analyze_function() Analyzing mip_worker
condition_parser() Using parsers: (AnalysisKind.PEP316, AnalysisKind.icontract,
    AnalysisKind.deal)
```

Listing 2: CrossHair traces

[1] https://anonymous.4open.science/r/APTPchecker-Symex-7865/checker/checker_testable.py

C Optimizations

While the core `APTPchecker` algorithm in Alg. 2 is minimal, it can be inefficient. `APTPchecker` employs several optimizations to improve its efficiency. These are crucial for checking large proof trees generated for challenging problems.

Neuron Stabilization A primary challenge in DNN analysis is the presence of large numbers of piece-wise linear constraints (e.g., ReLU) which generate a large number of branches and yield large proof trees. In the MILP formulation, this creates many disjunctions which are hard to solve. To reduce the number of disjunctions, `APTPchecker` uses *neuron stabilization* [8] to determine neurons that are *stable*, either active or inactive, for all inputs defined by the property pre-condition. For all stable neurons, the disjunctive ReLU constraint is replaced with a linear constraint that represents the neuron's value. This simplifies the MILP problem.

`APTPchecker` traverses the DNN and computes stable neurons. It initializes the MILP model with input constraints and then iterates over each layer of the network. Next, for each layer, it creates constraints depending on the layer type. Moreover, it uses approximation to estimate bounds of neuron values to determine neuron stability. Next, it filters unstable neurons and attempts to make them stable by optimizing either their lower or upper bounds.

Pruning Leaf Nodes `APTPchecker` uses a backtracking mechanism to check the parent node only when the child nodes are infeasible. Specifically, if it determines unsatisfiability of leaf l, it will check the parent p of l. If p is unsatisfiable it immediately removes the children of p (more specifically the sibling of l). Next it backtracks to the parent of p and repeats until meeting a stopping criteria. This optimization reduces the number of LP problems that need to be solved, making the proof checking process more efficient.

Parallelization `APTPchecker` leverages the structure of `APTP` proof tree to parallelize the checking of leaf nodes. Each tree path is an independent sub-proof and partitions of the tree allow checker to leverage multiprocessing to check large proof trees efficiently.

C.1 Proof Checking Optimizations

The performance cactus plots Fig. 3 present an ablation study of the pruning (X) and stabilization (S) optimizations of `APTPchecker`. The trend across both benchmarks is consistent with pruning (yellow) and stabilization (blue) improving the number of problems solved by 5% and 36%, respectively, over the unoptimized `APTPchecker` (green). The combination of optimizations (red) improves the number of solved problems by 46%, which is more than the sum of their individual improvements demonstrating that the methods create opportunities for one another for further optimization.

The Fig. 4 (right) and Fig. 5 (right) explore the impact of the S and X optimizations on the number of sub-proofs and MILP complexity. Across the benchmarks optimizations reduce the number of sub-proofs is to less than 1000 and MILP complexity to less than 2000. The reduction in sub-proofs directly contributes to the increase in performance of `APTPchecker`, but the reduction in MILP complexity is more subtle. Integer programming, and thus MILP, is known to be NP-Hard in general [38]. The stabilization optimization addresses this complexity by calculating sets of variables that are forced to take on specific values based on other constraints in the MILP problem. For each such variable, the constraints associated with it is effectively eliminated. We can observe this in comparing the left and right of Fig. 5 where we see both constraints of higher complexity eliminated and the peak of the constraint distribution shifted downward from 400 to 100 constraints.

D Related Work

Proof checking has been widely-recognized in the field of constraint solving such as SAT/SMT solving. (e.g., [39, 40, 41]). There is extensive literature on clausal proof generation and checking for SAT solvers [42, 43, 44, 9, 45]. Most modern SAT solvers can produce resolution-based proofs in standard formats (e.g., DRAT [9]), which can be independently checked by proof checkers, e.g., by efficient, untrusted programs such as `DRAT-trim` [9] or by certified, slower programs that work on extended formats such as LRAT [17] and GRAT [11].

SMT proof checkers [46, 47, 48] share the same purpose of checking unsatisfiability proofs, but they are more complex than SAT proof checkers due to the richer languages and theories of SMT formulas (e.g., theory of strings). Two significant proof-producing state-of-the-art SMT solvers are z3 [18] and veriT [19] that both can have their proofs successfully reconstructed in proof assistants [49, 50, 51, 52]. Other proof-producing SMT solvers are MathSAT5 [20] and SMTInterpol [21], CVC5 [22] and CertiStr [53]. Recently, a high-performance stand-alone checker Carcara [54] for the Alethe [55] proof format was also introduced.

Compared to SAT/SMT, DNN verification is a relatively new field, and the development of proof checkers for DNN verifiers is few. To the best of our knowledge, there is only one line of work [23, 24] that is explicitly for the `Marabou`. This work uses Farkas's lemma [56] for checking and is implemented in the Imandra [25] that can produce verifiable code. Our work generalizes to neuron-splitting based DNN verification and introduces a new, more expressive proof format, `APTP`, that can be adopted by other DNN verifiers. Our proof checker, `APTPchecker`, is also significantly more capable (§5.1).

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: Our claims match our theoretical and empirical results. `APTPchecker` solved more problems than SoTA checkers across benchmarks.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: The limitations are explicitly mentioned in §6.

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [Yes]

Justification: The correctness arguments have been provided in §4.1.2 and Apdx. B.

Guidelines:

- The answer NA means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: The model architectures and configurations used are provided in §5.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: The data and code with instructions to reproduce the results have been uploaded to an anonymized repo.

Guidelines:

- The answer NA means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (https://nips.cc/public/guides/CodeSubmissionPolicy) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (https://nips.cc/public/guides/CodeSubmissionPolicy) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyper-parameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

   Justification: Experimental details have been provided in §5.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [NA]

   Justification: Results are deterministic on the benchmarks, no random process involved, and no error bars need to be provided.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).

- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Detailed hardware resources are provided in §5.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics https://neurips.cc/public/EthicsGuidelines?

   Answer: [Yes]

   Justification: We have acknowledged the NeurIPS Code of Ethics.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: They have been discussed in §6.

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.
    - If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.

- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: No data or models are released.

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: We cited both $\alpha\beta$-`CROWN`, `NeuralSAT`, `Marabou` and its checker, and libraries used for experimental evaluation.

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, paperswithcode.com/datasets has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.

- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [NA]

    Justification: No new assets are introduced in this paper.

    Guidelines:

    - The answer NA means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [NA]

    Justification: The paper paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: The paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

Answer: [NA]

Justification: LLM was only used for revising writing and suggesting words.

Guidelines:

- The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy (https://neurips.cc/Conferences/2025/LLM) for what should or should not be described.