Harnessing Neuron Stability to Improve DNN Verification

HAI DUONG, George Mason University, USA
DONG XU, University of Virginia, USA
THANHVU NGUYEN, George Mason University, USA
MATTHEW B. DWYER, University of Virginia, USA

Deep Neural Networks (DNN) have emerged as an effective approach to tackling real-world problems. However, like human-written software, DNNs are susceptible to bugs and attacks. This has generated significant interest in developing effective and scalable DNN verification techniques and tools.

Recent developments in DNN verification have highlighted the potential of constraint-solving approaches that combine abstraction techniques with SAT solving. Abstraction approaches are effective at precisely encoding neuron behavior when it is linear, but they lead to overapproximation and combinatorial scaling when behavior is non-linear. SAT approaches in DNN verification have incorporated standard DPLL techniques, but have overlooked important optimizations found in modern SAT solvers that help them scale on industrial benchmarks.

In this paper, we present `VeriStable`, a novel extension of the recently proposed DPLL-based constraint DNN verification approach. `VeriStable` leverages the insight that while neuron behavior may be non-linear across the entire DNN input space, at intermediate states computed during verification many neurons may be constrained to have linear behavior – these neurons are stable. Efficiently detecting stable neurons reduces combinatorial complexity without compromising the precision of abstractions. Moreover, the structure of clauses arising in DNN verification problems shares important characteristics with industrial SAT benchmarks. We adapt and incorporate multi-threading and restart optimizations targeting those characteristics to further optimize DPLL-based DNN verification.

We evaluate the effectiveness of `VeriStable` across a range of challenging benchmarks including fully-connected feedforward networks (FNNs), convolutional neural networks (CNNs) and residual networks (ResNets) applied to the standard MNIST and CIFAR datasets. Preliminary results show that `VeriStable` is competitive and outperforms state-of-the-art DNN verification tools, including α-β-`CROWN` and `MN-BaB`, the first and second performers of the VNN-COMP, respectively.

CCS Concepts: • **Software and its engineering → Formal software verification**; • **Computing methodologies → Machine learning**.

Additional Key Words and Phrases: deep neural network verification, clause learning, abstraction, constraint solving, SAT/SMT solving

ACM Reference Format:
Hai Duong, Dong Xu, Thanhvu Nguyen, and Matthew B. Dwyer. 2024. Harnessing Neuron Stability to Improve DNN Verification. *Proc. ACM Softw. Eng.* 1, FSE, Article 39 (July 2024), 23 pages. https://doi.org/10.1145/3643765

Authors' addresses: Hai Duong, George Mason University, Fairfax, USA, hduong22@gmu.edu; Dong Xu, University of Virginia, Charlottesville, USA, dx3yy@virginia.edu; Thanhvu Nguyen, George Mason University, Fairfax, USA, tvn@gmu.edu; Matthew B. Dwyer, University of Virginia, Charlottesville, USA, matthewbdwyer@virginia.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 2994-970X/2024/7-ART39
https://doi.org/10.1145/3643765

1 INTRODUCTION

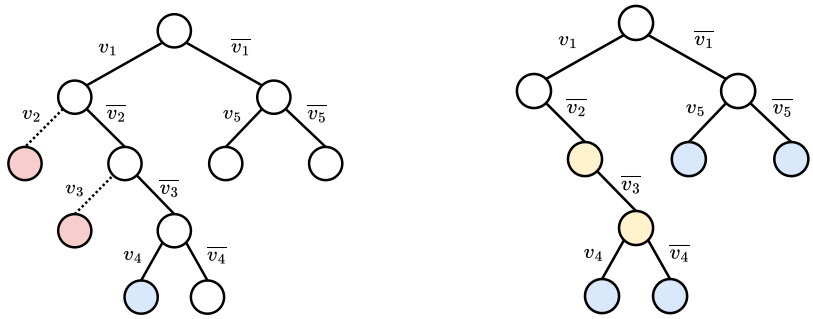
Increasingly deep neural networks (DNN) are being employed as components of mission-critical systems across a range of application domains, such as autonomous driving [Lee and Liu 2023; Shao et al. 2023], medicine [Bizjak et al. 2022; Morris et al. 2023], and infrastructure monitoring [Ewald et al. 2022; Ye et al. 2023]. DNNs require high levels of assurance in order to confidently deploy them in such systems.

As with traditional software, testing DNNs using rigorous coverage criteria [Dola et al. 2023; Kim et al. 2019; Sun et al. 2019; Zohdinasab et al. 2021] is necessary but not sufficient for critical deployments. To provide further evidence that DNN behavior meets expectations researchers have developed a range of techniques for verifying specifications formulated as pre/post-condition specifications that can be rendered in a canonical form [Shriver et al. 2021b]. Many dozens of DNN verifiers have been reported in the literature and a yearly competition has documented advances in the capabilities of such techniques [Bak et al. 2021; Brix et al. 2023; Müller et al. 2022].

Despite those advances, as with traditional software verification, DNN verification suffers from exponential worst-case complexity [Katz et al. 2017]. To understand why, consider the common case of DNNs with neurons using the rectified linear unit (ReLU) activation function [Goodfellow et al. 2016a]. The input of a neuron is defined as the weighted sum of the outputs of neurons preceding it in the computation graph, where the weights are the learned parameters of the DNN. The output of a neuron applies the ReLU function, $ReLu(x) = max(x, 0)$, to its input. This can be encoded as the disjunction of two partial linear functions – the zero and identity functions defined over negative and non-negative domains, respectively. When a neuron's input is positive, $x > 0$, the neuron is said to be *active*; otherwise, it is *inactive*. For a given input, running inference on a DNN causes each neuron to be either active or inactive. The vector of Boolean values representing each neuron's activation status is called an *activation pattern* for the input. In the worst-case, if the DNN has n neurons then there are 2^n activation patterns. Realistic DNNs, like ResNet [He et al. 2016], can have 10s of thousands of neurons making it extremely challenging to reason about the full space of activation patterns.

While this complexity seems daunting, history has shown that despite the worst-case exponential growth of verification problems, like propositional satisfiability (SAT) [Cook 1971], it is possible to solve very large problem instances with sophisticated algorithmic techniques [Biere et al. 2009]. Modern SAT solvers aim to determine if there exists an assignment of truth values to propositional variables that satisfies a given set of logical constraints. They are based on the classic Davis-Putnam-Logemann-Loveland (DPLL) algorithm [Davis et al. 1962] which searches the space of assignments by alternating between *deciding* how to extend a partial assignment – by choosing a variable and a truth value for it – and identify additional assignments that are *implied* by that decision. State-of-the-art solvers also incorporate a plethora of optimizations like Conflict-Driven Clause Learning (CDCL) to short-circuit later portions of the search [Zhang et al. 2001], heuristics to restart search with learned clauses [Biere 2008], and parallel exploration of variable assignments [Le Frioux et al. 2017]. Modern satisfiability modulo theory (SMT) solvers combine combine DPLL with theory-specific symbolic deduction methods that adapt and integrate with CDCL to form DPLL(T), where T stands for theory [Nieuwenhuis et al. 2006].

Most prior work on DNN verification either used SMT to discharge sub-problems formed by search of the space of activation patterns [Huang et al. 2017; Katz et al. 2017, 2022], applied forms of abstract interpretation to approximate the disjunctive neuron behavior [Bak 2021; Botoeva et al. 2020; Ferrari et al. 2022; Gehr et al. 2018; Henriksen and Lomuscio 2020; Singh et al. 2019a, 2018a,b, 2019b; Tjeng et al. 2017; Wang et al. 2018a, 2021; Xu et al. 2020c], or combined these approaches [Ehlers 2017; Katz et al. 2019].



(a) NeuralSAT (b) VeriStable

Fig. 1. The tree of activation patterns computed by NeuralSAT (left) and VeriStable (right) at corresponding points during a verification run.

The success of these methods inspired recent work that adapts DPLL(T) to DNN verification by incorporating an abstraction-based theory solver [Duong et al. 2024] to realize the NeuralSAT verifier. In NeuralSAT, propositional variables encode whether a neuron is active or inactive, and additional constraints encode the weighted sums for each neuron input. As illustrated on the left of Fig. 1, NeuralSAT searches the space of activation patterns for a DNN; here v_i and $\overline{v_i}$ denote that the ith neuron is active or inactive, respectively, and a path in the tree is a partial activation pattern. As we discuss in §2, NeuralSAT's contribution lies in combining DPLL(T) with a custom theory solver, that uses abstraction, to determine whether a partial activation pattern implies the specified property or implies conflict clauses that can prune subsequent search through CDCL.

In this paper, we further extend DPLL(T)-based DNN verification in two significant ways.

First, we propose a method for computing, from a partial activation pattern, a set of neurons that must be either active or inactive – such a neuron is said to be *stable*. Stable neurons eliminate the need for deciding their activation status later in the search and thereby lead to combinatorial reduction in the search. Unlike prior work [Chen et al. 2022; Li et al. 2022; Xiao et al. 2019; Zhang et al. 2019] which seeks to modify the network to create neurons that are stable for inputs described by the specification precondition, our approach (1) does not modify the network being verified and (2) detects neurons that are stable relative to subsets of the precondition. Our method can be thought of as *state-sensitive neuron stabilization*, where the state is a partial activation pattern encoding a subset of the precondition. Fig. 1 depicts how after v_1 is decided our method, VeriStable, stabilizes two neurons to be stable and inactive – shown in yellow – which eliminates the need to search their active branches – shown in red – as required by NeuralSAT. In this depiction, v_1 constitutes the state relative to which v_2 and v_3 are determined to be stable.

Second, we adapt parallelization techniques and restart heuristics from propositional SAT solvers to target the problem of DNN verification. Fig. 1 depicts how NeuralSAT's search frontier is a single state – shown in blue – and how VeriStable can expand a broader frontier and do so in parallel. As depicted, stabilization and parallelization are synergistic in that the former reduces the tree width which allows the latter to process a larger percentage of the tree.

While we developed these methods in the context of DPLL(T), these conceptual contributions are broadly applicable to any DNN verification approach that performs a search of the space of activation patterns and splits the search based on the activation status of neurons, such as [Bak 2021; Wang et al. 2021]. We implement the methods in VeriStable and demonstrate empirically that each of the methods it incorporates leads it to outperform NeuralSAT, that in combination all

of the methods lead to a 12-fold increase in the ability to solve verification problems, and that it establishes a new state-of-the-art in DNN verification compared with the top performers in the most recent DNN verifier competition [Müller et al. 2022].

The key contributions of the paper lie in:

- developing a novel approach that computes state-sensitive neuron stability to eliminate the need for neuron splitting in DNN verification;
- adaptation of advanced SAT optimizations into a DPLL(T)-based verification algorithm;
- evaluation results using a new challenging DNN verification benchmark, as well as existing benchmarks, that demonstrate a 12-fold performance improvement and that `VeriStable` establishes the state-of-the-art in DNN verifier performance; and
- release of an open source implementation of `VeriStable`[1] accepting verification problems in standard formats to promote the application of DNN verification and comparative evaluation.

2 BACKGROUND

2.1 The DNN verification problem

A *neural network* (**NN**) [Goodfellow et al. 2016b] consists of an input layer, multiple hidden layers, and an output layer. Each layer contains neurons connected to neurons in previous layers via predefined weights obtained through training with data. A *deep* neural network (**DNN**) is an NN with at least two hidden layers.

The output of a DNN is computed by iteratively calculating the values of neurons in each layer. Neurons in the input layer receive the input data. Neurons in the hidden layers compute their values through an *affine transformation* followed by an *activation function*, like the popular Rectified Linear Unit (ReLU) activation.

For ReLU activation, the value of a hidden neuron y is given by $ReLU(w_1v_1 + \ldots + w_nv_n + b)$, where b is the bias parameter for y, $w_i, \ldots, w_n$ are the weights of y, $v_1, \ldots, v_n$ are the neuron values from the preceding layer, $w_1v_1 + \cdots + w_nv_n + b$ represents the affine transformation, and $ReLU(x) = \max(x, 0)$ defines the ReLU activation. A ReLU-activated neuron is said to be *active* if its input value is greater than zero and *inactive* otherwise.

We note that ReLU DNNs are popular because they tend to be sparsely activated [Glorot et al. 2011] and $\max(x, 0)$ is efficient to compute which leads to efficient training and inference. Moreover, they avoid the vanishing gradient problem [Goodfellow et al. 2016a] which speeds training convergence, especially in deep networks. This makes ReLU networks an important class to target, but we note that `VeriStable` applies equally well to DNNs using other piecewise-linear activation functions, such as leaky ReLU [Maas et al. 2013] and parametric ReLU [He et al. 2015].

DNN Verification. Given a DNN N and a property ϕ, the *DNN verification problem* asks if ϕ is a valid property of N. Typically, ϕ is a formula of the form $\phi_{in} \Rightarrow \phi_{out}$, where ϕ_{in} is a property over the inputs of N and ϕ_{out} is a property over the outputs of N. This form of property has been used to encode safety and security requirements of DNNs, e.g., safety specifications to avoid collision in unmanned aircraft [Kochenderfer et al. 2012]. A DNN verifier attempts to find a *counterexample* input to N that satisfies ϕ_{in} but violates ϕ_{out}. If no such counterexample exists, ϕ is a valid property of N. Otherwise, ϕ is not valid and the counterexample can be used to retrain or debug the DNN [Huang et al. 2017].

Example. Fig. 2 shows a simple DNN with two inputs x_0, x_1, four hidden neurons $n_{00}, n_{01}, n_{10}, n_{11}$, and two outputs y_0, y_1. The weights of a neuron are shown on its incoming edges, and the bias is shown above or below each neuron. The outputs of the hidden neurons are computed by the affine

[1] https://github.com/dynaroars/neuralsat

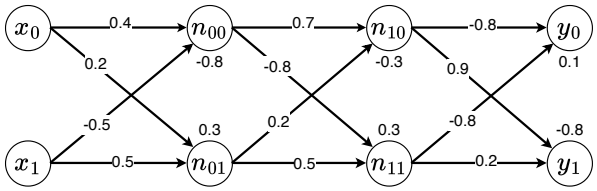


Fig. 2. An FNN with ReLU.

transformation and ReLU, e.g., $n_{00} = ReLU(0.4x_0 - 0.5x_1 - 0.8)$. The output neuron is computed with just the affine transformation, i.e., $y_0 = -0.8n_{10} - 0.8n_{11} + 0.1$. A valid property for this DNN is that the output is $y_0 > y_1$ for any inputs $x_0 \in [-2.0, 2.0], x_1 \in [-1.0, 1.0]$.

ReLU-based DNN verification is NP-Complete [Katz et al. 2017] and thus can be formulated as an SAT or SMT checking problem. Direct application of SMT solvers does not scale to the large and complex formulae encoding real-world, complex DNNs. While custom solvers, like `Planet` and `Reluplex`, retain the soundness, completeness, and termination of SMT and improve on the performance of a direct SMT encoding, they do not scale to handle realistic DNNs [Bak et al. 2021].

Abstraction. Applying techniques from abstract interpretation [Cousot and Cousot 1977], the abstraction-based DNN verifiers overapproximate nonlinear computations (e.g., ReLU) of the network using linear abstract domains such as intervals [Wang et al. 2018b] or polytopes [Singh et al. 2019b; Xu et al. 2020a]. This allows abstraction-based DNN verifiers to side-step the disjunctive splitting that is the performance bottleneck of constraint-based DNN verifiers.

2.2 DPLL(T)-based DNN Verification

While abstraction is crucial to the performance of DNN verification techniques, recent work on `NeuralSAT` [Duong et al. 2024] shows that combining it with the DPLL(T) approach of modern SMT solvers [Barrett et al. 2011; Kroening and Strichman 2016; Moura and Bjørner 2008] can further improve the scalability of DNN verification. Fig. 3 gives an overview of `NeuralSAT`, which consists of a theory solver (Deduce) and standard DPLL components (everything else).

`NeuralSAT` constructs a propositional formula representing neuron activation status (Boolean Abstraction) and searches for satisfying truth assignments while employing a DNN-specific theory solver to check feasibility with respect to DNN constraints and properties. The process integrates standard DPLL components, which include Deciding variable assignments, and performing Boolean constraint propagation (BCP), with DNN-specific theory solving (Deduce), which uses LP solving and the polytope abstraction to check the satisfiability of assignments with the property of interest. If satisfiability is confirmed, it continues with new assignments; otherwise, it analyzes and learns conflict clauses (Analyze Conflict) to backtrack. `NeuralSAT` continues it search until it either proves the property (`unsat`) or finds a total assignment (`sat`). In §4.1 we describe how these DPLL components are adapted and incorporated into `VeriStable`.

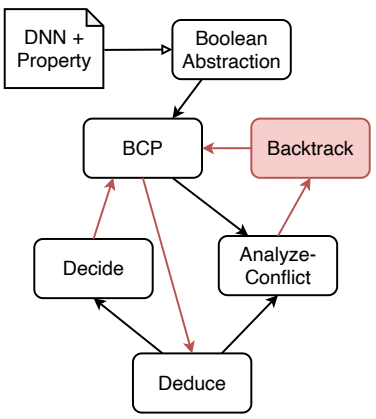


Fig. 3. `NeuralSAT` Architecture

2.3 Neuron Stability

A ReLU neuron is *stable* relative to a given specification when it is in either its active or inactive phase for all inputs satisfying the specification's precondition. Researchers have observed that stable neurons have the potential to improve verifier performance, since they tend to linearize the otherwise highly non-linear computation encoded in the NN. However, in prior work, this required modifying the NN. They have done this by increasing stability through a training objective [Xiao et al. 2019] or by identifying stable neurons and applying non-standard modifications that use strictly linear activation functions for those neurons [Chen et al. 2022; Zhang et al. 2019]. Importantly, this means that these techniques do not verify the original neural network.

We develop a method that can identify and exploit stable neurons while verifying the original network. Moreover, we observe that a neuron may be stable relative to a subset of a specification's pre-condition. Our method identifies when the verifier is analyzing such a subset which allows for a finer state-sensitive notion of neuron stability to be exploited. In §4.2.1 we define a method that encodes such subsets as partial activation patterns of neurons which allows neurons relative to that subset to be computed and subsequent verification to be more efficient.

3 OVERVIEW AND ILLUSTRATION

3.1 Overview

Fig. 4 gives an overview of `VeriStable` DPLL(T) approach. Compared to the `NeuralSAT` DPLL(T) algorithm in Fig. 3, `VeriStable` also consists of standard DPLL components `Decide`, `BCP`, and `AnalyzeConflict`, and the DNN-dedicated theory solver for `Deduce`. However, the `VeriStable` approach extends and significantly improves the performance of `NeuralSAT` in three main ways.

First, for theory solving, we leverage the concept of *neuron stability* to improve bound tightening and infer when neurons have linear behavior (Stabilize). This improves abstraction precision and eliminates the need to decide the activation status of neurons. Second, `VeriStable` employs a distributed search tree data structure to develop a parallel DPLL(T) approach. This allows `VeriStable` to leverage multicore processing and simultaneously analyze multiple possible assignments (Select). This replaces Backtrack from DPLL(T) because it considers multiple branches simultaneously (including the one that would be backtracked to if run sequentially). Finally, `VeriStable` adopts *restart heuristics* from modern SAT solving (e.g., PicoSAT [Biere 2008]) to escape local optima (Restart). As we will discuss later, restarting especially benefits "hard" DNN problems by enabling better clause learning and exploring different decision orderings. In combination these techniques allow `VeriStable` to solve 12 times more problems than `NeuralSAT` (§6).

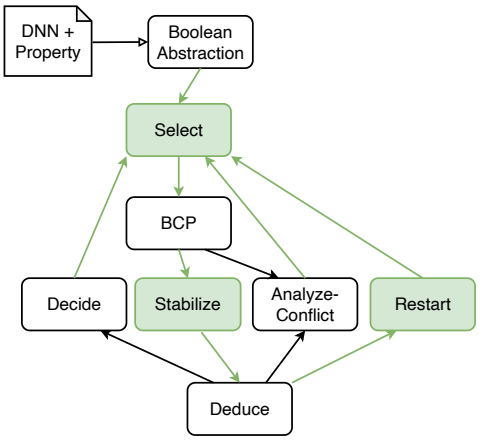


Fig. 4. `VeriStable` Architecture

3.2 Illustration

We illustrate how `VeriStable` operates to verify the specification

$$(x_0, x_1) \in [-2.0, 2.0] \times [-1.0, 1, 0] \Rightarrow y_0 > y_1$$

on the DNN in Fig. 2 by proving that the negation of the problem, $\alpha \wedge (-2.0 \leq x_0 \leq 2.0) \wedge (-1.0 \leq x_1 \leq 1.0) \wedge (y_0 \leq y_1)$, where α is a logical formula encoding the DNN, is unsatisfiable.

Verification computes an interval approximating each neuron including the outputs, $y_i \in [l_{y_i}, u_{y_i}]$. So if $l_{y_0} > u_{y_1}$ then the post-condition $\phi_{out} = y_0 > y_1$. In our explanation $p = l_{y_0} - u_{y_1}$ is the difference in those bounds. If $p > 0$ then the ϕ_{out} is *infeasible*, otherwise it *might be* feasible.

Boolean Abstraction. First, `VeriStable` creates the boolean variables $v_{00}, v_{01}, v_{10}, v_{11}$ to represent the activation status of the hidden neurons $n_{00}, n_{01}, n_{10}, n_{11}$, respectively. Next, `VeriStable` forms the initial clauses $\{c_0 : v_{00} \vee \overline{v_{00}}; c_1 : v_{01} \vee \overline{v_{01}}; c_2 : v_{10} \vee \overline{v_{10}}; c_3 : v_{11} \vee \overline{v_{11}}\}$, which indicate that these variables are either `true` (active) or `false` (inactive).

DPLL(T) Iterations. `VeriStable` now searches for satisfiable activation patterns, i.e., an assignment σ over these variables that satisfies the clauses *and* the constraints in the network that they represent.

Iteration 1: `VeriStable` starts with an empty assignment, σ_0. Next `VeriStable` performs abstraction to obtain the lower and upper bounds $n_{00} \in [-2.1, 0.5]$, $n_{01} \in [-0.6, 1.2]$, $n_{10} \in [-0.42, 0.16]$, and $n_{11} \in [-0.16, 0.9]$. Thus $p_{\sigma_0} = -0.1663$ and ϕ_{out} might be feasible. `VeriStable` then runs LP solving on these bounds and improves them: $n_{10} \in [-0.3, 0.09]$ and $n_{11} \in [0.0, 0.9]$. n_{11} has been determined to be *stable* in its active phase. The new bound is $p_{\sigma_0} = -0.0942$ and ϕ_{out} remains feasible.

Now `VeriStable` decides a variable and assigns it a value. This decision step performs *neuron splitting*, e.g., the decision $v_i \mapsto T$ means that neuron n_i has a non-negative ReLU value and therefore is active. Decisions may be inconsistent and require expensive backtracking. `VeriStable` reduces neuron splitting through neuron stabilization and parallel search. First, if a neuron, like n_{11}, is stable it does not need to be split. Second, `VeriStable` explores multiple decision branches in parallel.

After determining that n_{11} is stable, we only have to consider the other three neurons. Suppose that `VeriStable` chooses v_{01}. A sequential algorithm would decide a value for v_{01} and continue with that decision. Parallel `VeriStable` mitigates the possibility of wrong decisions by processing both decisions simultaneously. Specifically, `VeriStable` adds both assignments $\{\sigma_1 : v_{01}, \sigma_2 : \overline{v_{01}}\}$ to the set of assignments to be considered.

Iteration 2: `VeriStable` selects both σ_1 and σ_2 and runs them in parallel. As before `VeriStable` attempts to tighten bounds to determine stable neurons, but none can be found – our example is too simple. In deduction, for the $\sigma_1 : v_{01}$ branch, `VeriStable` generates a new constraint $n_{01} \in [0.0, 1.2]$, represented by $v_{01} \mapsto T$, corresponding to the active phase of $n_{01} \in [-0.6, 1.2]$. `VeriStable` then uses the new constraint to compute output bounds $p_{\sigma_1} = -0.0941$ so ϕ_{out} is feasible.

Similarly, for the $\sigma_2 : \overline{v_{01}}$ branch, `VeriStable` creates the constraint $n_{01} \in [-0.6, 0.0]$ and obtains $p_{\sigma_2} = 0.0001$ which indicates that ϕ_{out} is *not* feasible. This means that the decision $v_{01} \mapsto F$ is inconsistent and in a sequential DPLL(T) we would backtrack to try $v_{01} \mapsto T$. However, in `VeriStable` we are already processing this decision branch, σ_1, in parallel and therefore do not backtrack. Next, `VeriStable` analyses the infeasible assignment $\sigma_2 : \overline{v_{01}}$ and learns the new conflict clause $c_4 : v_{01}$.

For the $\sigma_1 : v_{01}$ branch, `VeriStable` determines feasibility and chooses v_{00} to split which adds variable assignments $\{\sigma_3 : v_{01} \wedge v_{00}, \sigma_4 : v_{01} \wedge \overline{v_{00}}\}$ to the set of assignments to be considered.

`VeriStable` continues for a few more iterations and determines that all activation patterns are infeasible and returns `UNSAT`, indicating the desired property is valid.

While simple, this example illustrates the benefits of stabilization and parallelization relative to a baseline DPLL(T)-based DNN verification approach. §4 details how both of these techniques work and describes how restarts can improve performance on challenging verification problems.

```
input  : DNN α, property φ_in ⇒ φ_out, parallel factors n and k
output : unsat if the property is valid and sat otherwise
1  clauses ← BooleanAbstraction(α)
2  while true do
3      assignments ← [(∅, ∅)] // initialize empty assignment and igraph
4      while true do // main DPLL loop
           // select n assignments (activation patterns) and corresponding igraphs
5          [(σ_1, igraph_1), ..., (σ_n, igraph_n)] ← Select(assignments, n)
           // process n assignments in parallel
6          parfor (σ_i, igraph_i) in [(σ_1, igraph_1), ..., (σ_n, igraph_n)] do
7              is_conflict ← true
8              if BCP(clauses, σ_i, igraph_i) then
9                  if StabilizeCondition() then // stabilize with condition
10                     Stabilize(α, φ_in, φ_out, σ_i, k) // stabilize k neurons
11                 if Deduce(σ_i, α, φ_in, φ_out) then
12                     (is_sat, v_i) ← Decide(α, φ_in, φ_out, σ_i) // decision heuristic
13                     if is_sat then
14                         return sat // consistent and complete assignment
15                     assignments ← assignments ∪ {(σ_i ∧ v_i, igraph_i) ; (σ_i ∧ ¬v_i, igraph_i)}
16                     is_conflict ← false // no conflict
17             if is_conflict then
18                 clauses ← clauses ∪ AnalyzeConflict(igraph_i) // learn conflict clauses
19         if length(assignments) ≡ 0 then // check unsat
20             return unsat // no more assignment to be processed
21         if Restart() then // check restart heuristic
22             break // restart occurs
```

Fig. 5. The VeriStable algorithm.

4 THE VERISTABLE APPROACH

Fig. 5 shows the VeriStable algorithm, which takes as input the formula α representing the ReLU-based DNN N and the formulae $\phi_{in} \Rightarrow \phi_{out}$ representing the property to be proved. Internally, VeriStable checks the satisfiability of the formula

$$\alpha \wedge \phi_{in} \wedge \overline{\phi_{out}}. \tag{1}$$

VeriStable returns unsat if the formula unsatisfiable, indicating that ϕ is a valid property of N, and sat if it is satisfiable, indicating the ϕ is not a valid property of N.

4.1 DPLL(T)-based DNN Verification

VeriStable uses a DPLL(T)-based algorithm to check unsatisfiability. The algorithm consists of a Boolean abstraction, standard DPLL components, and a theory solver (T-solver) that is specific to the verification of ReLU DNNs.

4.1.1 Boolean Representation. BooleanAbstraction(Fig. 5, line 1) encodes the DNN verification problem into a Boolean constraint to be solved. This step creates Boolean variables to represent the *activation status* of hidden neurons in the DNN. VeriStable also forms a set of initial clauses ensuring that each status variable is either T (active) or F (inactive).

4.1.2 DPLL search. `VeriStable` iteratively searches for an assignment satisfying the clauses. Throughout it maintains several state variables including: clauses, a set of *clauses* consisting of the initial activation clauses and learned conflict clauses; σ, a *truth assignment* mapping status variables to truth values which encodes a partial activation pattern; and *igraph*, an *implication graph* used for analyzing conflicts.

`Decide` (Fig. 5, line 12) chooses an unassigned variable and assigns it a random truth value. Assignments from `Decide` are essentially guesses that can be wrong which degrades performance. The purpose of `BCP`, `Deduce`, and `Stabilize` – which are discussed below – is to eliminate unassigned variables so that `Decide` has fewer choices.

`BooleanConstraintPropagation` or `BCP` (Fig. 5, line 8) detects *unit clauses*[2] from constraints representing the current assignment and clauses and infers values for variables in these clauses. For example, after the decision $a \mapsto F$, `BCP` determines that the clause $a \vee b$ becomes unit, and infers that $b \mapsto T$. Internally, `VeriStable` uses an *implication graph* [Barrett 2013] to represent the current assignment and the reason for each `BCP` implication.

`AnalyzeConflict` (Fig. 5, line 18) processes an implication graph with a conflict to learn a new *clause* that explains the conflict. The algorithm traverses the implication graph backward, starting from the conflicting node, while constructing a new clause through a series of resolution steps. `AnalyzeConflict` aims to obtain an *asserting* clause, which is a clause that will result a `BCP` implication. These are added to clauses so that they can block further searches from encountering an instance of the conflict.

These are standard components in DPLL-based algorithms including modern SAT/SMT solvers and `NeuralSAT`. As shown in Fig. 3, DPLL also has backtracking, which allows the algorithm to go back to an incorrect assignment decision and choose the correct one instead. However, as will be described in §4.2.2, the `VeriStable` parallel DPLL(T) does not require backtracking because it has optimistically considered both the correct and incorrect assignments simultaneously.

4.1.3 Theory Solver. `VeriStable`'s Theory or T-solver (Fig. 5, lines 9-16) consists of two parts: stabilization and deduction.

`Deduce` (Fig. 5, line 11) checks the feasibility of the DNN constraints represented by the current propositional variable assignment. This component is shared with `NeuralSAT` and it leverages specific information from the DNN problem, including input and output properties, for efficient feasibility checking. Specifically, it obtains neuron bounds using the polytope abstraction[Xu et al. 2020a,c] and performs infeasibility checking to detect conflicts.

The second part of the theory solver, which is specific to `VeriStable`, implements stabilization and is described next.

4.2 Improvements in `VeriStable`

We now describe neuron stability, parallel search, and restart. In §6.1 and §6.2 we present ablation studies demonstrating the performance of these ideas individually and in combination.

4.2.1 Neuron Stability. The key idea in using neuron stability is that if we can determine that a neuron is stable, we can assign the exact truth value for the corresponding Boolean variable instead of having to guess. This has a similar effect as `BCP` – reducing mistaken assignments by `Decide` – but it operates at the theory level not the propositional Boolean level.

[2]A unit clause is a clause that has a single unassigned literal.

Stabilization involves the solution of a mixed integer linear program (MILP) system [Tjeng et al. 2019]:

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

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. The equations encode the semantics of a DNN as follows: (a) defines the affine transformation computing the pre-activation value for a neuron in terms of outputs in the preceding layer; (b) defines the inputs and outputs in terms of the adjacent hidden layers; (c) asserts that post-activation values are non-negative and no less than pre-activation values; (d) defines that the neuron activation status indicator variables that are either 0 or 1; and (e) defines 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) constraints 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 the semantics of $\hat{z}_j^{(i)} = max(z_j^{(i)}, 0)$.

Fig. 6 describes `Stabilize` solves this equation system. First, a MILP problem is created from the current assignment, the DNN, and the property of interest using formulation in Eq. 2. Note that the neuron lower ($l_j^{(i)}$) and upper bounds ($u_j^{(i)}$) can be quickly computed by polytope abstraction.

Next, it collects a list of all unassigned variables which are candidates being stabilized (line 2). In general, there are too many unassigned neurons, so `Stabilize` restricts consideration to k candidates. Because each neuron has a different impact on abstraction precision we prioritize the candidates. In `Stabilize`, neurons are prioritized based on their interval boundaries (line 3) with a preference for neurons with either lower or upper bounds that are closer to zero. The intuition is that neurons with bounds close to zero are more likely to become stable after tightening.

We then select the top-k (line 4) candidates and seek to further tighten their interval bounds. The order of optimizing bounds of select neurons is decided by its boundaries, e.g., if the lower bound is closer to zero than the upper bound then the lower bound would be optimized first. These optimization processes, i.e., `Maximize` (line 7 or line 13) and `Minimize` (line 9 or line 11), are performed by an external LP solver (e.g., Gurobi [Gurobi Optimization, LLC 2022]).

Note that the work in [Tjeng et al. 2019] uses the MILP system in Eq. 2 to encode the entire verification problem and thus is limited to the encodings of small networks that can be handled by an LP solver. In contrast, `VeriStable` creates this system based on the current assignment, which has significantly fewer constraints. Moreover, we only use the computed bounds of hidden neurons from this system, and thus even if it cannot be solved, `VeriStable` will still continue.

4.2.2 Parallelism. The DPLL(T) process in `VeriStable` is designed as a tree-search problem where each internal node encodes an *activation pattern* defined by the variable assignments from the root. To parallelize DPLL(T), we adopt a beam search-like strategy which combines distributed search from Distributed Tree Search (DTS) algorithm [Ferguson and Korf 1988] and Divide and Conquer (DNC) [Le Frioux et al. 2017] paradigms for splitting the search space into disjoint subspaces that can be solved independently. At every step of the search algorithm, we select up to n nodes of

input : DNN α, property $\phi_{in} \Rightarrow \phi_{out}$, current assignment σ, number of neurons for stabilization k
output : Tighten bounds for variables **not** in σ (unassigned variables)

```
1  model ← CreateMILP(α, φ_in, φ_out, σ) // create model (Eq. 2) with current assignment
2  [v_1, ..., v_m] ← GetUnassignedVariable(σ) // get all m current unassigned variables
3  [v'_1, ..., v'_m] ← Sort([v_1, ..., v_m]) // prioritize tightening order
4  [v'_1, ..., v'_k] ← Select([v'_1, ..., v'_m], k) // select top-k unassigned variables, k ≤ m
   // stabilize k neurons in parallel
5  parfor v_i in [v'_1, ..., v'_k] do
6      if (v_i.lower + v_i.upper) ≥ 0 then // lower is closer to 0 than upper, optimize lower first
7          Maximize(model, v_i.lower) // tighten lower bound of v_i
8          if v_i.lower < 0 then // still unstable
9              Minimize(model, v_i.upper) // tighten upper bound of v_i
10     else // upper is closer to 0 than lower, optimize upper first
11         Minimize(model, v_i.upper) // tighten upper bound of v_i
12         if v_i.upper > 0 then // still unstable
13             Maximize(model, v_i.lower) // tighten lower bound of v_i
```

Fig. 6. Stabilize

the DPLL(T) search tree to create a beam of width n. This splits (like DNC) the search into n subproblems that are independently processed. Each subproblem extends the tree by a depth of 1.

Our approach simplifies the more general DNC scheme since the n bodies of the **parfor** on line 6 of Fig. 5 are roughly load balanced. While this is a limited form of parallelism, it sidesteps one of the major roadblocks to DPLL parallelism – the need to efficiently synchronize across load-imbalanced subproblems [Le Frioux et al. 2017, 2019].

In addition to raw speedup due to multiprocessing, parallelism accelerates the sharing of information across search subspaces, in particular learned clause information for DPLL. In `VeriStable`, we only generate independent subproblems which eliminates the need to coordinate their solution. When all subproblems are complete, their conflicts are accumulated, Fig. 5 line 18, to inform the next round of search. As we show in §6, the engineering of this form of parallelism in DPLL(T) leads to substantial performance improvement.

4.2.3 Restart. As with any stochastic algorithm, `VeriStable` would perform poorly if it gets into a subspace of the search that does not quickly lead to a solution, e.g., due to choosing a bad sequence of neurons to split [De Palma et al. 2021; Ferrari et al. 2022; Wang et al. 2021]. This problem, which has been recognized in early SAT solving, motivates the introduction of restarting the search [Gomes et al. 1998] to avoid being stuck in such a *local optima*.

`VeriStable` uses a simple restart heuristic that triggers a restart when either the number of processed assignments (nodes) exceeds a pre-defined number or the number of remaining assignments that need be checked exceeds a pre-defined threshold. After a restart, `VeriStable` avoids using the same decision order of previous runs (i.e., it would use a different sequence of neuron splittings). It also resets all internal information except the learned conflict clauses, which are kept and reused as these are *facts* about the given constraint system. This allows a restarted search to quickly prune parts of the space of assignments. Although restarting may seem like an engineering aspect, it plays a crucial role in stochastic algorithms like `VeriStable` and helps reduce verification time for challenging problems as shown in §6.1.

Tab. 1. Benchmark instances. U: unsat, S: sat, ?: unknown.

Benchmarks	Networks		Per Network		Tasks	
	Type	Networks	Neurons	Parameters	Properties	Instances (U/S/?)
ACAS Xu	FNN	45	300	13305	10	139/47/0
MNISTFC	FNN	3	0.5-1.5K	269-532K	90	56/23/11
CIFAR2020	CNN	3	17-62K	2.1-2.5M	203	149/43/11
RESNET_A/B	CNN+ResNet	2	11K	354K	144	49/23/72
MNIST_GDVB	FNN	38	0.7-5.1K	0.2-3.0M	16	51/0/39
Total		**91**			**463**	**444/136**/133

4.3 VeriStable Implementation

VeriStable is written in Python, and uses Gurobi [Gurobi Optimization, LLC 2022] for LP solving and bounds tightening, and the LiRPA abstraction library [Xu et al. 2020a,c] for approximation. Currently, VeriStable supports feedforward (FNN), convolutional (CNN), and Residual Learning Architecture (ResNet) neural networks that use ReLU. VeriStable supports the standard specification formats ONNX [Bai et al. 2023] for neural networks and VNN-LIB [Tacchella et al. 2023] for properties. These formats are standard and are supported by state-of-the-art DNN verification tools, which enable comparative evaluation.

5 EXPERIMENTAL DESIGN

Our goals are to understand how incorporating stabilization and other DPLL(T) optimizations allows for scaling of DNN verification. We focus our evaluation on the following research questions:
RQ1 (§6.1): How does stabilization impact the performance of DPLL(T)-based DNN verification?
RQ2 (§6.2): How do VeriStable optimizations improve performance in isolation and combination?
RQ3 (§6.3): How does VeriStable compare to state-of-the-art DNN verifiers?

5.1 Benchmarks: DNN Verification Problems

To gain insights into the performance improvements of VeriStable we require benchmarks that force the algorithm to search a non-trivial portion of the space of activation patterns. It is well-known that SAT problems can be very easy to solve regardless of their size or whether they are satisfiable or unsatisfiable [Gent and Walsh 1994]. The same is true for DNN verification problems. The organizers of the first three DNN verifier competitions remark on the need for benchmarks that are "not so easy that every tool can solve all of them" in order to assess verifier performance [Brix et al. 2023].

To achieve this we leverage a systematic DNN verification problem generator GDVB [Xu et al. 2020b]. GDVB takes a seed neural network as input and systematically varies a number of architectural parameters, e.g., number of layers, and neurons per layer, to produce a set of DNNs. In this experiment, we begin with a single MNIST network with 3 layers, each with 1024 neurons and generate 38 different DNNs that cover combinations of parameter variations. We leverage the fact that local robustness properties are a pseudo-canonical form for pre-post condition specifications [Shriver et al. 2021b] and use GDVB to generate 16 properties with varying radii and center points. Next we run two state-of-the-art verifiers: α-β-CROWN and MN-BaB, for each of the $38 * 16 = 608$ combinations of DNN and property with a small timeout of 200 seconds. Any problem that could be solved within that timeout was removed from the benchmark as "too easy". This resulted in 90 verification problems that not only are more computationally challenging than

benchmarks used in other studies, e.g., [Müller et al. 2022], but also exhibit significant architectural diversity. We use this **MNIST_GDVB** benchmark for RQ1 and RQ2 to study the variation in performance on challenging problems.

For RQ3 we use five VNN-COMP'22 standard benchmarks in addition to MNIST_GDVB. These benchmarks, shown in Tab. 1, consist of 91 networks, spanning multiple types and architectures of layers, and 463 safety and robustness properties. The **Per Network** column gives the size of each network (**neurons** are the numbers of hidden neurons and **parameters** are the numbers of weights and biases). For example, each FNN in ACAS Xu has 5 inputs, 6 hidden layers (each with 50 neurons), 5 outputs, and thus has 300 neurons (6×50) and 13305 parameters ($5 \times 50 \times 50 + 2 \times 50 \times 5 + 6 \times 50 + 5$).

In total, we have 713 problem instances (an instance is the verification task of a property of a network). Among these instances, 444 are known to be `unsat` (U), 136 are `sat` (S), and 133 are unknown (?) because no existing verifiers, in this study or in VNN-COMP, can solve them. We exclude unknown instances from our study because they do not contribute to our evaluation or comparison to other tools.

The six benchmarks are as follows. **ACAS Xu** consists of 45 FNNs to issue turn advisories to aircrafts to avoid collisions. Each FNN has 5 inputs (speed, distance, etc). We use all 10 safety properties as specified in [Katz et al. 2017] and VNN-COMP'22, where properties 1–4 are used on 45 networks and properties 5–10 are used on a single network. **MNISTFC** consists of 3 FNNs for handwritten digit recognition and 30 robustness properties. Each FNN has 28x28 inputs representing a handwritten image. **CIFAR2020** has 3 CNNs for object detection and 203 robustness properties (each CNN has a set of different properties). Each network uses 3x32x32 RGB input images. For **RESNET_A/B**, each benchmark has only one network with the same architecture and 72 robustness properties. Each network uses 3x32x32 RGB input images.

5.2 Baselines: DNN Verifiers

For RQ1 we compare `VeriStable` to **`NeuralSAT`** [Duong et al. 2024] which is the only DPLL(T) DNN verifier available. `NeuralSAT` is recent and did not participate in VNN-COMP'22. However, it has been shown to have good performance for feedforward networks.

RQ2 compares different configurations of `VeriStable` to each other.

For RQ3, we selected four well-known DNN verifiers as baselines for comparison in addition to `NeuralSAT`. **`α-β-CROWN`** [Wang et al. 2021; Zhang et al. 2022] employs multiple abstractions and algorithms for efficient analysis, e.g., input splitting for networks with small input dimensions and parallel Branch-and-Bound [Bunel et al. 2020] (BaB) otherwise. **`MN-BaB`** [Ferrari et al. 2022], the successor of `ERAN` [Singh et al. 2019a,b], uses multiple abstractions and BaB. **`Marabou`** [Katz et al. 2022, 2019], the successor of the Reluplex work, is a simplex-based solver that employs a parallel Split-and-Conquer (SnC) [Wu et al. 2020] search and uses polytope abstraction [Singh et al. 2019b] and LP-based bound tightening. **`nnenum`** [Bak et al. 2020] combines optimizations such as parallel case splitting and multiple levels of abstractions, e.g., three types of zonotopes with imagestar/starset [Tran et al. 2019].

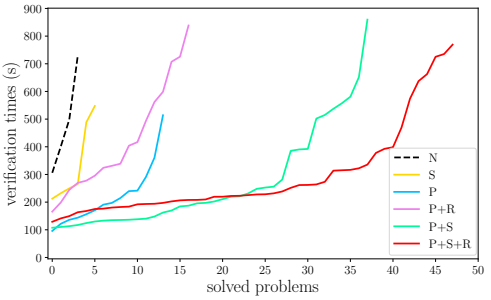
These four tools competed in VNN-COMP'22 [Müller et al. 2022] and were among the very top performers. For example, `α-β-CROWN` is the winner for MNISTFC and also the overall winner, `MN-BaB` ranked 3rd on MNISTFC and second overall, and `nnenum` was the only one that can solve all instances in ACAS Xu and was 4th overall. `Marabou` ranked 6th on MNISTFC and 7th overall.

5.3 Experimental Setup

Our 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. All tools use

Tool	Setting	#Solved	Avg. Time
NeuralSAT	-	4	867.35
VeriStable	S	6	833.25
	P	14	713.21
	P+R	17	741.00
	P+S	38	430.60
	P+S+R	48	330.46

(a) Problems solved and solve time (s).



(b) Sorted solved problems

Fig. 7. Performance of VeriStable with different optimization settings in comparison to NeuralSAT on the **MNIST_GDVB** benchmark with 900 second timeout, where: "N" is the base case (NeuralSAT), "P" enables Parallelism, "R" enables Restarts, and "S" enables Stabilization.

multiprocessing (even external tools/libraries including Gurobi, LiRPA, and Pytorch are multi-thread). α-β-CROWN, MN-BaB, NeuralSAT, and VeriStable leverage GPU processing for abstraction.

To conduct a fair evaluation, we reuse the benchmarks and installation/run-scripts available from VNN-COMP[3]. These scripts were tailored by the developers of each verifier to maximize performance on each benchmark. VNN-COMP uses varying runtimes for each problem instance ranging from 30 seconds to more than 20 minutes. The competition also uses several different Amazon AWS instances with different configurations (e.g., CPU, GPU, RAM) to run the tools. Thus, we experimented with timeouts on our machine and settled on 900 seconds per instance which allowed the verifiers to achieve similar scoring performance reported in VNN-COMP'22.

6 RESULTS AND ANALYSIS

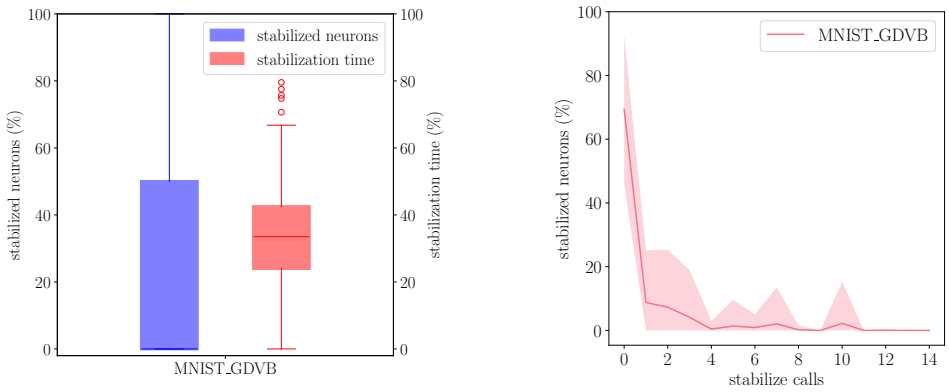
We discuss the metrics for each question, present experimental results, and interpret those results to answer the research questions.

6.1 RQ1: Benefit of Stabilization

We focus here on the benefit of stabilization on DPLL(T)-based DNN verification as implemented in NeuralSAT. We use the 51 challenging verification problems in the MNIST_GDVB benchmark to explore performance and measure the number of problems solved and the time to solve problems as metrics.

Fig. 7 presents data on NeuralSAT: the first row in the table on the left and the black dashed line in the plot on the right. The plot on the right shows the problems solved within the 900-second timeout for each technique sorted by runtime from fastest to slowest; problems that timeout are not shown on the plot. Enabling only stabilization in VeriStable yields the data indicated with an "S": the second row and yellow lines, respectively. We observe a 50% increase in the number of problems solved with stabilization. The average times show a modest reduction of about 4%, but since NeuralSAT or "S" solved just a few benchmarks the average is swamped by the time taken by problems that timeout – at 900 seconds. Comparing the dashed and yellow lines in Fig. 7b shows that for the solved problems "S" reduces verification time significantly, e.g., on the first problem from just over 300 seconds to just over 200 seconds. Stabilization alone improves performance, but it has a much more significant benefit in combination with other optimizations.

[3]https://github.com/ChristopherBrix/vnncomp2022_benchmarks



(a) Stabilization rate (per call) and stabilization time

(b) Stabilization rate over time

Fig. 8. Stabilization cost and effectiveness during verification.

We collected data to understand how frequently neurons could be stabilized and at what cost. Fig. 8a plots the percentage of neurons that are stabilized across the MNIST_GDVB benchmark, on the left axis, and the percentage of verification time taken up by stabilization, on the right axis. This aggregated data shows that stabilization can incur a non-trivial share of verification time, but as the data in Fig. 7 showed despite this overhead the overall verification time is reduced for solved problems.

We can also observe that while the mean number of stabilized neurons is low, the variance is quite high which indicates a degree of effectiveness in reducing the combinatorics in subsequent searches. We dug into the stabilization data further to try to understand this variance. Fig. 8b plots the mean – red line – and standard deviation – shaded region – of the number of stabilized neurons over time during verification; recall from line 9 of Fig. 5 that stabilization is selectively enabled during search. Stabilization is effective early in the search and less so as it progresses. This makes sense since line 3 in Fig. 6 prioritizes neurons for stabilization. This is desirable because it encourages stabilization at the beginning of the search which leads to a greater combinatorial reduction in the search and a consequent improvement in its scalability.

RQ1 Findings: Stabilization improves the number of problems solved and reduces verification time. It does so by trading overhead to compute stable neurons to linearize parts of the search of the space of activation patterns. Moreover, it pushes this linearization to the top of the search tree to yield greater combinatorial reduction.

6.2 RQ2: Optimization Ablation Study

We used the same benchmark as in RQ1, but here we focus primarily on the benefits and interactions among the optimizations in `VeriStable`. The bottom five rows in the table on the left of Fig. 7

We omit the use of restart on its own, since it is intended to function in concert with parallelization. Both “S” and “P” improve the number of problems solved and reduce cost relative to the `NeuralSAT` baseline, but parallelism yields greater improvements. When parallelism is combined with restart we see that the number of problems solved increases, but the average time increases slightly. The reason for this is that for the 3 additional benchmarks that could be solved the verification process had conducted a partial search of the space of activation patterns prior to restarts and the cost of that search is added to the cost of the successful post-restart search.

Tab. 2. A **Verifier**'s rank (#) is based on its VNN-COMP score (**S**) on a benchmark. For each benchmark, the number of problems verified (**V**) and falsified (**F**) are shown.

Verifier	ACAS Xu				MNISTFC				CIFAR2020				RESNET_A/B				MNIST_GDVB				Overall			
	#	S	V	F	#	S	V	F	#	S	V	F	#	S	V	F	#	S	V	F	#	S	V	F
VeriStable	**1**	**1437**	**139**	**47**	2	573	55	**23**	**1**	**1533**	**149**	**43**	**1**	**513**	**49**	**23**	**1**	**480**	**48**	0	**1**	**4536**	**440**	**136**
α-β-CROWN	3	1436	**139**	46	**1**	**582**	**56**	22	2	1522	148	42	**1**	**513**	**49**	**23**	2	400	40	0	2	4453	432	133
NeuralSAT	5	1417	137	**47**	4	383	36	**23**	4	1522	148	42	3	483	46	**23**	4	40	4	0	3	3845	371	135
MN-BaB	6	1097	105	**47**	5	370	36	10	3	1486	145	36	4	363	34	**23**	3	200	20	0	4	3516	340	116
nnenum	**1**	**1437**	**139**	**47**	3	403	39	13	5	518	50	18	-	-	-	-	-	-	-	-	5	2358	228	78
Marabou	4	1426	138	46	6	370	35	20	-	-	-	-	-	-	-	-	-	-	-	-	6	1796	173	66

Perhaps most noteworthy is the data on parallelism in combination with stabilization. We see a significant jump in the number of solved problems relative to both "S" and "P" – a 6.3 fold and 2.7 fold increase, respectively. As illustrated in Fig. 1 this combination is synergistic because stabilization creates a *narrower* tree within which the parallel *beam* can make more rapid progress. Adding in restart yields the best performance in terms of both problems solved – 12 fold increase NeuralSAT – and solve time – 2.6 fold decrease.

The plot on the right of Fig. 7 shows the trend in verification solve times for each optimization combination across the benchmarks. One can observe that adding more optimizations improves performance both by the fact that the plots are lower and extend further to the right. For example, extending "P" to "P+S" shows lower solve times for the first 17 problems – the one's "P" could solve – and that 38 of the 51 benchmark problems are solved. Extending "P+S" to the full set of optimizations exhibits what appears to be a degradation in performance for the first 23 problems solved and this is likely due to the fact that, as explained above, restart forces some re-exploration of the search. However, the benefit of restart shows in the ability to significantly reduce verification time for 25 of the 48 problems solved by "P+S+R".

RQ2 Findings: Each of the VeriStable optimizations improves on the performance of the baseline DPLL(T)-based DNN verifier. Moreover, combinations of the optimizations appear to operate synergistically to increase performance beyond their additive benefits. When running VeriStable, enabling all optimizations appears to be the best choice.

6.3 RQ3: Comparison with state-of-the-art DNN verifiers

In this section, we evaluate VeriStable relative to a set of 5 baseline DNN verifiers across a broader benchmark that reflects the problems used in VNN-COMP [Müller et al. 2022]. For metrics, we adopt the scoring system proposed for VNN-COMP 2023 which seeks to balance the relative difficulty of verifying a problem versus falsifying it and to account for the possibility that verifiers report erroneous results. More specifically, for each benchmark instance, a verifier scores 10 points if it correctly verifies an instance, 1 point if it correctly falsifies an instance, 0 points if it cannot solve (e.g., times out, has errors, or returns unknown), and -150 points if it gives incorrect results[4]. This scoring emphasizes a technique's ability to correctly verify problems[5].

Tab. 2 shows the results of all six tools. Since the magnitude of the score is not easily interpreted, since it depends on the size of the benchmark, we report the **Rank** of each tool using the VNN-COMP score for each benchmark as well as the overall rank. Tools that do not work on a benchmark

[4]We note that all of the verifiers in our study gave correct results on the considered benchmarks.
[5]We dropped the extra 2 bonus points for the fastest verifiers in the VNN-COMP'22 scoring system because VNN-COMP has removed this time bonus as they found it did not make a difference in scoring

are not shown under that benchmark (e.g., Marabou reports errors for all CIFAR2020 problems, nnenum and Marabou cannot solve any instances of MNIST_GDVB). The last two columns break down the number of problems each verifier was able to **Verify** or **Falsify**.

On 5 of the 6 benchmarks, and overall, VeriStable ranks at the top, tying with other verifiers on the ACAS Xu and RESNET benchmarks. Recall that these benchmarks vary significantly in the number of neurons and parameters, with the ACAS Xu models being modestly sized and the CIFAR models being the largest, and VeriStable is the best on both ends of the scale spectrum. VeriStable ranks second on the MNISTFC benchmark to α-β-CROWN both solve the same number of problems, but α-β-CROWN verifies a problem that VeriStable does not leading to its higher score. The MNIST_GDVB benchmark varies in size from being comparable to the smallest MNISTFC network to larger than the largest MNISTFC network. Still, a key distinguishing feature of the benchmark is the filtration of *easy* problems. Whereas MNISTFC includes 23 problems that can be falsified, MNIST_GDVB has none, yet VeriStable performs better on these harder problems.

While not a factor in our evaluation, we note that several baseline verifiers require hyperparameter tuning. For example, the run-script of α-β-CROWN for VNN-COMP customizes 10 parameters per *each* benchmark to optimize its performance[6]. In contrast, when run with all optimizations enabled, which we recommend based on RQ2's findings, VeriStable has two parameters: the degree of parallelism, n, and the number of neurons to attempt to stabilize, k. In these experiments, we fixed these at $k = 64$ and $n = 4000$ for all benchmarks, which we believe is evidence that developers can more easily apply VeriStable to new benchmarks while achieving good performance.

RQ3 Findings: VeriStable ranks at the top of a set of baseline DNN verifiers that were shown to be the best performers in a recent DNN verification competition [Müller et al. 2022]. It performs well on smaller problems like ACAS Xu, where techniques with sophisticated abstract domains like nnenum work well. It performs well on larger problems like CIFAR2020, where techniques like nnenum fail to solve problems and even highly optimized abstraction-based methods like α-β-CROWN fall short. It performs well on challenging problems like MNIST_GDVB, forcing verifiers to analyze the combinatorially sized space of activation patterns to verify problems.

7 THREATS TO VALIDITY

Regarding threats to internal validity, we built off the existing code base of NeuralSAT, thereby leveraging that team's efforts to validate their implementation. We used assertions in almost every function of our implementation to check the correctness properties flowing from our algorithms, e.g., that lower and upper bounds are properly ordered. Those assertions were enabled during our rigorous testing process that ran all of the VNN-COMP benchmarks through our implementation, where we confirmed the expected results.

We selected VNN-COMP benchmarks to promote comparability and enhance external validity. Those benchmarks were developed by other researchers to express verification problems for neural networks, e.g., ACAS Xu is a collision avoidance prediction network for small aerial drones. Based on our own experience and the experience of the VNN-COMP organizers, who found that some of the VNN-COMP benchmarks were *too easy*, we developed a new benchmark, MNIST_GDVB. That benchmark was developed using an approach that guarantees a form of systematic diversity across the networks and specifications that comprise the benchmark. We plan to continue to push for the development of benchmarks that reflect the challenges of DNN verification, but in this work, we believe our benchmarks are broader and more challenging than prior work.

[6]For MNIST_GDVB, the default configuration of α-β-CROWN performed poorly, so we adopted the configuration used for MNISTFC which gave good results for MNIST_GDVB.

Regarding construct validity, we used standard metrics, like number of problems solved and VNN-COMP score, that have been widely used [Müller et al. 2022; Xu et al. 2020b]. This makes comparing our results to prior work easier and allows researchers familiar with the metrics to interpret our results easily. Moreover, the metrics lead to a natural interpretation that permits answering the research questions, e.g., a verifier that solves more problems has better performance.

8 DISCUSSION

In addition to scalability, which is the focus of `VeriStable`, there are two other common challenges in DNN verification: identifying valuable correctness properties of DNNs and developing a formal notation to encode them.

Property Specifications. This paper focuses on improving the verification of specifications formulated using sets of half-space polytopes – each specified as the conjunction of cutting planes – where one set defines the pre-condition, ϕ_{in}, and another the post-condition, ϕ_{out} (§2.1). While it does not address the pragmatics of expressing meaningful domain-specific specifications, we note that this is an active area of work [Geng et al. 2023; Toledo et al. 2023]. For example, the work in [Toledo et al. 2023] allows one to define domain-specific masking and transformation operations to localize perturbations to a region within an input image and within a range of values that remain on the data distribution. For example, the color of vehicles in a scene does not impact the predicted steering angle for an end-to-end driving model. We note, however, that such properties are amenable to verification with tools like `VeriStable`.

This paper leverages the fact that an arbitrary half-space polytope specification can be expressed as a local robustness property [Shriver et al. 2021b]. This means that we can evaluate verification scalability improvements by only considering local robustness specifications and these results will be informative about the verifier performance on a much broader class of specifications, like those in [Toledo et al. 2023].

Specification Format. As mentioned, half-space polytope allows for a general class of specifications to be checked, but for high-dimensional input spaces it can be inconvenient to write specifications using notations like the standard VNN-LIB format [Tacchella et al. 2023]. For example, a well-studied class of specifications expresses local robustness properties of the form: $\forall x : \forall p \in [0, \epsilon] : N(x) = N(x \pm p)$. Expressing such a specification for MNIST requires choosing an input image, x, and a maximum perturbation, p, as defined by the robustness radius, ϵ. In VNN-LIB such a specification would would be more than 1500 lines long, since each dimension of the 784 input must be constrained from above and below. Moreover, a separate specification must be produced for each input image and radius.

To address these pragmatic challenges, Shriver et al. developed the DNNV toolkit [Shriver et al. 2021a], which consists of a parametric Python-embedded DSL to express such specifications concisely, e.g., just 10 lines of code for the aforementioned MNIST specification[7]. Moreover, DNNV allows specifications to be written in a form that is independent of model input dimension, as above, and translated to VNN-LIB for verification with `VeriStable`.

9 RELATED WORK

Research on DNN verification is extensive and continuously expanding. This section provides an overview of established techniques and their accompanying tool implementations.

Constraint-based approaches, e.g., `Reluplex` [Katz et al. 2017], and its successor `Marabou` [Katz et al. 2022, 2019], `DLV` [Huang et al. 2017], `Planet` [Ehlers 2017], and `MIPVerify` [Tjeng et al. 2019]

[7] https://github.com/dlshriver/dnnv

encode the problem as a constraint-solving task. These techniques transform DNN verification into a constraint problem, solvable using tools like SMT solvers (Planet, DLV) or SAT-based approach with custom simplex and MILP solvers (Reluplex, Marabou). **Abstraction-based approaches**, e.g., AI^2 [Gehr et al. 2018], ERAN [Müller et al. 2021; Singh et al. 2018a, 2019b] (DeepZ, RefineZono, DeepPoly, K-ReLU), MN-BaB [Ferrari et al. 2022], Reluval [Wang et al. 2018b], Neurify [Wang et al. 2018a], VeriNet [Henriksen and Lomuscio 2020], NNV [Tran et al. 2021], nnenum [Bak 2021; Bak et al. 2020], CROWN [Zhang et al. 2018], and α-β-CROWN [Wang et al. 2021], leverage abstract domains to tackle scalability. These techniques employ various abstract domains, such as intervals, zonotopes, polytopes, and starsets/imagestars, to improve scalability. To address spurious counterexamples due to overapproximations, these methods often iterate to check counterexamples and refine abstractions. NeuralSAT [Duong et al. 2024] integrates DPLL search with abstraction-based theory solving. OVAL [OVAL-group 2023] and DNNV [Shriver et al. 2021a] serve as platforms that integrate multiple existing DNN verification tools. VeriStable extends the NeuralSAT DPLL(T) approach with neuron stabilization, parallel search, and restart.

Common **abstract domains** used in DNN verification include intervals [Wang et al. 2018b], zonotopes [Singh et al. 2018a], polytopes [Singh et al. 2019b; Xu et al. 2020a,c], and starsets/imagestars [Bak et al. 2020; Tran et al. 2021]. Notably, top verifiers like ERAN, MN-BaB, and nnenum leverage multiple abstract domains to enhance their effectiveness. For instance, ERAN combines zonotopes and polytopes, while nnenum incorporates polytopes, zonotopes, and imagestars. Currently, VeriStable employs polytope abstraction for bound tightening but can also use other abstract domains.

Heuristics and **optimizations** play a crucial role in the efficiency of SAT solving. Modern SAT/SMT solvers [Barrett et al. 2011; Kroening and Strichman 2016; Moura and Bjørner 2008], for instance, benefit from strategies such as VSIDS and DLIS for decision (branching), random restart, and clause shortening or deletion to optimize memory utilization and avoid local maxima. Specifically for DNNs, **neuron stability** serves as a hidden metric for assessing the linearity of neurons with piece-wise linear activation functions. In practice, researchers employ heuristics to apply neuron stability to ReLU during the training of neural networks. For example, the RS Loss approach [Xiao et al. 2019; Xu et al. 2024] incorporates regularization techniques to train more stable weights. The linearity grafting technique [Chen et al. 2022] directly replaces ReLU activation functions with linear ones to achieve stability. Both unstructured and structured DNN pruning [Zhangheng et al. 2022] can also help network stabilization. We introduce DPLL(T) parallel search and the concept of neuron stabilization to improve the scalability of DNN verification.

10 CONCLUSION AND FUTURE WORK

As the need for formal analysis increases when more neural networks are being deployed in safety-critical areas, the DNN verification field has received great attention in recent years. In this work, we introduce VeriStable, a ReLU-based DNN verification tool that integrates an advanced DPLL(T) search technique in SAT solving with the concept of neuron stability to significantly reduce the search space of DNN verification. Our evaluation confirms the effectiveness of VeriStable, which establishes a new state-of-the-art in DNN verifications compared to the performances in the recent DNN verification competition.

We have many opportunities to further improve the performance of VeriStable. For example, we plan to extend the capability of neuron stabilization in other non-DPLL-based DNN verification techniques and explore new decision heuristics for DPLL(T)-based tools. Moreover, by using DPLL with implication graphs, VeriStable inherits a native mechanism to verify its own results, e.g., using these graphs and conflicting clauses to obtain resolution graphs/proofs and UNSAT cores as proofs of unsatisfiability [Asín et al. 2008; Kroening and Strichman 2016; Zhang and Malik 2003].

11 DATA AVAILABILITY

`VeriStable` is available at: https://github.com/dynaroars/neuralsat

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful comments. This material is based in part upon work supported by the National Science Foundation under grant numbers 1900676, 2019239, 2129824, 2200621, 2217071, 2238133, and 2319131, and by an Amazon Research Award.

REFERENCES

Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. 2008. Efficient generation of unsatisfiability proofs and cores in SAT. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 16–30.

Junjie Bai, Fang Lu, and Ke Zhang. 2023. ONNX Open neural network exchange. https://www.onnx.ai/

Stanley Bak. 2021. nnenum: Verification of relu neural networks with optimized abstraction refinement. In *NASA Formal Methods Symposium*. Springer, 19–36.

Stanley Bak, Changliu Liu, and Taylor Johnson. 2021. The Second International verification of Neural Networks Competition (VNN-COMP 2021): Summary and Results. *arXiv preprint arXiv:2109.00498* (2021).

Stanley Bak, Hoang-Dung Tran, Kerianne Hobbs, and Taylor T Johnson. 2020. Improved geometric path enumeration for verifying relu neural networks. In *International Conference on Computer Aided Verification*. Springer, 66–96.

Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. Cvc4. In *International Conference on Computer Aided Verification*. Springer, 171–177.

Clark W Barrett. 2013. Decision Procedures: An Algorithmic Point of View. *J. Autom. Reason.* 51, 4 (2013), 453–456.

Armin Biere. 2008. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation* 4, 2-4 (2008), 75–97.

Armin Biere, Marijn Heule, and Hans van Maaren. 2009. *Handbook of satisfiability*. Vol. 185. IOS press.

Žiga Bizjak, June Ho Choi, Wonhyoung Park, and Žiga Špiclin. 2022. Deep Learning Based Modality-Independent Intracranial Aneurysm Detection. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*. Springer, 760–769.

Elena Botoeva, Panagiotis Kouvaros, Jan Kronqvist, Alessio Lomuscio, and Ruth Misener. 2020. Efficient verification of relu-based neural networks via dependency analysis. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 3291–3299.

Christopher Brix, Mark Niklas Müller, Stanley Bak, Taylor T Johnson, and Changliu Liu. 2023. First three years of the international verification of neural networks competition (VNN-COMP). *International Journal on Software Tools for Technology Transfer* (2023), 1–11.

Rudy Bunel, P Mudigonda, Ilker Turkaslan, P Torr, Jingyue Lu, and Pushmeet Kohli. 2020. Branch and bound for piecewise linear neural network verification. *Journal of Machine Learning Research* 21, 2020 (2020).

Tianlong Chen, Huan Zhang, Zhenyu Zhang, Shiyu Chang, Sijia Liu, Pin-Yu Chen, and Zhangyang Wang. 2022. Linearity grafting: Relaxed neuron pruning helps certifiable robustness. In *International Conference on Machine Learning*. PMLR, 3760–3772.

Stephen A Cook. 1971. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*. 151–158.

Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 238–252.

Martin Davis, George Logemann, and Donald Loveland. 1962. A machine program for theorem-proving. *Commun. ACM* 5, 7 (1962), 394–397.

Alessandro De Palma, Rudy Bunel, Alban Desmaison, Krishnamurthy Dvijotham, Pushmeet Kohli, Philip HS Torr, and M Pawan Kumar. 2021. Improved branch and bound for neural network verification via lagrangian decomposition. *arXiv preprint arXiv:2104.06718* (2021).

Swaroopa Dola, Matthew B Dwyer, and Mary Lou Soffa. 2023. Input Distribution Coverage: Measuring Feature Interaction Adequacy in Neural Network Testing. *ACM Transactions on Software Engineering and Methodology* 32, 3 (2023), 1–48.

Hai Duong, ThanhVu Nguyen, and Matthew Dwyer. 2024. A DPLL(T) Framework for Verifying Deep Neural Networks. *arXiv preprint arXiv:2307.10266* (2024).

Ruediger Ehlers. 2017. Formal verification of piece-wise linear feed-forward neural networks. In *International Symposium on Automated Technology for Verification and Analysis*. Springer, 269–286.

Vincentius Ewald, Ramanan Sridaran Venkat, Aadhik Asokkumar, Rinze Benedictus, Christian Boller, and Roger M Groves. 2022. Perception modelling by invariant representation of deep learning for automated structural diagnostic in aircraft maintenance: A study case using DeepSHM. *Mechanical Systems and Signal Processing* 165 (2022), 108153.

Chris Ferguson and Richard E Korf. 1988. Distributed Tree Search and Its Application to Alpha-Beta Pruning.. In *AAAI*, Vol. 88. 128–132.

Claudio Ferrari, Mark Niklas Mueller, Nikola Jovanović, and Martin Vechev. 2022. Complete Verification via Multi-Neuron Relaxation Guided Branch-and-Bound. In *International Conference on Learning Representations*.

Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. 2018. Ai2: Safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE symposium on security and privacy (SP)*. IEEE, 3–18.

Chuqin Geng, Nham Le, Xiaojie Xu, Zhaoyue Wang, Arie Gurfinkel, and Xujie Si. 2023. Towards reliable neural specifications. In *International Conference on Machine Learning*. PMLR, 11196–11212.

Ian P Gent and Toby Walsh. 1994. The SAT phase transition. In *ECAI*, Vol. 94. PITMAN, 105–109.

Xavier Glorot, Antoine Bordes, and Yoshua Bengio. 2011. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, 315–323.

Carla P Gomes, Bart Selman, Henry Kautz, et al. 1998. Boosting combinatorial search through randomization. *AAAI/IAAI* 98 (1998), 431–437.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016a. *Deep learning*. MIT press.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016b. *Deep Learning*. MIT Press. https://www.deeplearningbook.org

Gurobi Optimization, LLC. 2022. Gurobi Optimizer Reference Manual. https://www.gurobi.com

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*. 1026–1034.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 770–778.

Patrick Henriksen and Alessio Lomuscio. 2020. Efficient neural network verification via adaptive refinement and adversarial search. In *ECAI 2020*. IOS Press, 2513–2520.

Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. 2017. Safety verification of deep neural networks. In *International conference on computer aided verification*. Springer, 3–29.

Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. 2017. Reluplex: An efficient SMT solver for verifying deep neural networks. In *International Conference on Computer Aided Verification*. Springer, 97–117.

Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. 2022. Reluplex: a calculus for reasoning about deep neural networks. *Formal Methods in System Design* 60, 1 (2022), 87–116.

Guy Katz, Derek A Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljić, et al. 2019. The marabou framework for verification and analysis of deep neural networks. In *International Conference on Computer Aided Verification*. Springer, 443–452.

Jinhan Kim, Robert Feldt, and Shin Yoo. 2019. Guiding deep learning system testing using surprise adequacy. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1039–1049.

Mykel J Kochenderfer, Jessica E Holland, and James P Chryssanthacopoulos. 2012. *Next-generation airborne collision avoidance system*. Technical Report. Massachusetts Institute of Technology-Lincoln Laboratory Lexington United States.

Daniel Kroening and Ofer Strichman. 2016. *Decision procedures*. Springer.

Ludovic Le Frioux, Souheib Baarir, Julien Sopena, and Fabrice Kordon. 2017. PaInleSS: a framework for parallel SAT solving. In *Theory and Applications of Satisfiability Testing–SAT 2017: 20th International Conference, Melbourne, VIC, Australia, August 28–September 1, 2017, Proceedings 20*. Springer, 233–250.

Ludovic Le Frioux, Souheib Baarir, Julien Sopena, and Fabrice Kordon. 2019. Modular and efficient divide-and-conquer SAT solver on top of the painless framework. In *Tools and Algorithms for the Construction and Analysis of Systems: 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings, Part I 25*. Springer, 135–151.

Der-Hau Lee and Jinn-Liang Liu. 2023. End-to-end deep learning of lane detection and path prediction for real-time autonomous driving. *Signal, Image and Video Processing* 17, 1 (2023), 199–205.

Zhangheng Li, Tianlong Chen, Linyi Li, Bo Li, and Zhangyang Wang. 2022. Can pruning improve certified robustness of neural networks? *arXiv preprint arXiv:2206.07311* (2022).

Andrew L Maas, Awni Y Hannun, Andrew Y Ng, et al. 2013. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, Vol. 30. Atlanta, GA, 3.

Miranda X Morris, Aashish Rajesh, Malke Asaad, Abbas Hassan, Rakan Saadoun, and Charles E Butler. 2023. Deep learning applications in surgery: Current uses and future directions. *The American Surgeon* 89, 1 (2023), 36–42.

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

Christoph Müller, François Serre, Gagandeep Singh, Markus Püschel, and Martin Vechev. 2021. Scaling polyhedral neural network verification on gpus. *Proceedings of Machine Learning and Systems* 3 (2021), 733–746.

Mark Niklas Müller, Christopher Brix, Stanley Bak, Changliu Liu, and Taylor T Johnson. 2022. The Third International Verification of Neural Networks Competition (VNN-COMP 2022): Summary and Results. *arXiv preprint arXiv:2212.10376* (2022).

Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. 2006. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL (T). *Journal of the ACM (JACM)* 53, 6 (2006), 937–977.

OVAL-group. 2023. OVAL - Branch-and-Bound-based Neural Network Verification. https://github.com/oval-group/oval-bab.

Hao Shao, Letian Wang, Ruobing Chen, Hongsheng Li, and Yu Liu. 2023. Safety-enhanced autonomous driving using interpretable sensor fusion transformer. In *Conference on Robot Learning*. PMLR, 726–737.

David Shriver, Sebastian Elbaum, and Matthew B Dwyer. 2021a. DNNV: A framework for deep neural network verification. In *International Conference on Computer Aided Verification*. Springer, 137–150.

David Shriver, Sebastian Elbaum, and Matthew B Dwyer. 2021b. Reducing dnn properties to enable falsification with adversarial attacks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 275–287.

Gagandeep Singh, Rupanshu Ganvir, Markus Püschel, and Martin Vechev. 2019a. Beyond the single neuron convex barrier for neural network certification. *Advances in Neural Information Processing Systems* 32 (2019).

Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin Vechev. 2018a. Fast and effective robustness certification. *Advances in neural information processing systems* 31 (2018).

Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. 2018b. Boosting robustness certification of neural networks. In *International Conference on Learning Representations*.

Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. 2019b. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–30.

Youcheng Sun, Xiaowei Huang, Daniel Kroening, James Sharp, Matthew Hill, and Rob Ashmore. 2019. Structural test coverage criteria for deep neural networks. *ACM Transactions on Embedded Computing Systems (TECS)* 18, 5s (2019), 1–23.

Armando Tacchella, Luca Pulina, Dario Guidotti, and Stefano Demarchi. 2023. The international benchmarks standard for the Verification of Neural Networks. https://www.vnnlib.org/

Vincent Tjeng, Kai Xiao, and Russ Tedrake. 2017. Evaluating robustness of neural networks with mixed integer programming. *arXiv preprint arXiv:1711.07356* (2017).

Vincent Tjeng, Kai Y Xiao, and Russ Tedrake. 2019. Evaluating Robustness of Neural Networks with Mixed Integer Programming. In *International Conference on Learning Representations*.

Felipe Toledo, David Shriver, Sebastian Elbaum, and Matthew B Dwyer. 2023. Deeper Notions of Correctness in Image-Based DNNs: Lifting Properties from Pixel to Entities. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2122–2126.

Hoang-Dung Tran, Diago Manzanas Lopez, Patrick Musau, Xiaodong Yang, Luan Viet Nguyen, Weiming Xiang, and Taylor T Johnson. 2019. Star-based reachability analysis of deep neural networks. In *International symposium on formal methods*. Springer, 670–686.

Hoang-Dung Tran, Neelanjana Pal, Patrick Musau, Diego Manzanas Lopez, Nathaniel Hamilton, Xiaodong Yang, Stanley Bak, and Taylor T Johnson. 2021. Robustness verification of semantic segmentation neural networks using relaxed reachability. In *International Conference on Computer Aided Verification*. Springer, 263–286.

Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. 2018a. Efficient formal safety analysis of neural networks. *Advances in Neural Information Processing Systems* 31 (2018).

Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. 2018b. Formal security analysis of neural networks using symbolic intervals. In *27th USENIX Security Symposium (USENIX Security 18)*. 1599–1614.

Shiqi Wang, Huan Zhang, Kaidi Xu, Xue Lin, Suman Jana, Cho-Jui Hsieh, and J Zico Kolter. 2021. Beta-crown: Efficient bound propagation with per-neuron split constraints for neural network robustness verification. *Advances in Neural Information Processing Systems* 34 (2021), 29909–29921.

Haoze Wu, Alex Ozdemir, Aleksandar Zeljic, Kyle Julian, Ahmed Irfan, Divya Gopinath, Sadjad Fouladi, Guy Katz, Corina Pasareanu, and Clark Barrett. 2020. Parallelization techniques for verifying neural networks, Vol. 1. TU Wien Academic Press, 128–137.

Kai Yuanqing Xiao, Vincent Tjeng, Nur Muhammad (Mahi) Shafiullah, and Aleksander Madry. 2019. Training for Faster Adversarial Robustness Verification via Inducing ReLU Stability. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=BJfIVjAcKm

Dong Xu, Nusrat Jahan Mozumder, Hai Duong, and Matthew Dwyer. 2024. Training for Verification: Increasing Neuron Stability to Scale DNN Verification. In *Tools and Algorithms for the Construction and Analysis of Systems - 30th International*

Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS. Springer, to appear.

Dong Xu, David Shriver, Matthew B Dwyer, and Sebastian Elbaum. 2020b. Systematic generation of diverse benchmarks for dnn verification. In *International Conference on Computer Aided Verification*. Springer, 97–121.

Kaidi Xu, Zhouxing Shi, Huan Zhang, Yihan Wang, Kai-Wei Chang, Minlie Huang, Bhavya Kailkhura, Xue Lin, and Cho-Jui Hsieh. 2020a. Automatic perturbation analysis for scalable certified robustness and beyond. *Advances in Neural Information Processing Systems* 33 (2020), 1129–1141.

Kaidi Xu, Huan Zhang, Shiqi Wang, Yihan Wang, Suman Jana, Xue Lin, and Cho-Jui Hsieh. 2020c. Fast and complete: Enabling complete neural network verification with rapid and massively parallel incomplete verifiers. *arXiv preprint arXiv:2011.13824* (2020).

Xijun Ye, Peirong Wu, Airong Liu, Xiaoyu Zhan, Zeyu Wang, and Yinghao Zhao. 2023. A Deep Learning-based Method for Automatic Abnormal Data Detection: Case Study for Bridge Structural Health Monitoring. *International Journal of Structural Stability and Dynamics* (2023), 2350131.

Huan Zhang, Hongge Chen, Chaowei Xiao, Sven Gowal, Robert Stanforth, Bo Li, Duane Boning, and Cho-Jui Hsieh. 2019. Towards stable and efficient training of verifiably robust neural networks. *arXiv preprint arXiv:1906.06316* (2019).

Huan Zhang, Shiqi Wang, Kaidi Xu, Linyi Li, Bo Li, Suman Jana, Cho-Jui Hsieh, and J Zico Kolter. 2022. General cutting planes for bound-propagation-based neural network verification. *arXiv preprint arXiv:2208.05740* (2022).

Huan Zhang, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh, and Luca Daniel. 2018. Efficient neural network robustness certification with general activation functions. *Advances in neural information processing systems* 31 (2018).

Lintao Zhang, Conor F Madigan, Matthew H Moskewicz, and Sharad Malik. 2001. Efficient conflict driven learning in a boolean satisfiability solver. In *IEEE/ACM International Conference on Computer Aided Design. ICCAD 2001. IEEE/ACM Digest of Technical Papers (Cat. No. 01CH37281)*. IEEE, 279–285.

Lintao Zhang and Sharad Malik. 2003. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *2003 Design, Automation and Test in Europe Conference and Exhibition*. IEEE, 880–885.

LI Zhangheng, Tianlong Chen, Linyi Li, Bo Li, and Zhangyang Wang. 2022. Can Pruning Improve Certified Robustness of Neural Networks? *Transactions on Machine Learning Research* (2022).

Tahereh Zohdinasab, Vincenzo Riccio, Alessio Gambi, and Paolo Tonella. 2021. Deephyperion: exploring the feature space of deep learning-based systems through illumination search. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 79–90.

Received 2023-09-29; accepted 2024-01-23