Parallelization Techniques for Verifying Neural Networks

Haoze Wu[1], Alex Ozdemir[1], Aleksandar Zeljić[1], Kyle Julian[1], Ahmed Irfan[1], Divya Gopinath[2],
Sadjad Fouladi[1], Guy Katz[5], Corina Pasareanu[3,4], and Clark Barrett[1]

[1]Stanford University, USA. [2]NASA Ames, KBR Inc. [3]NASA Ames, Moffett Field, CA.
[4]Carnegie Mellon University, USA. [5]The Hebrew University of Jerusalem, Israel.

Abstract—**Inspired by recent successes with parallel techniques for solving Boolean satisfiability, we investigate a set of strategies and heuristics for leveraging parallelism to improve the scalability of neural network verification. We present a general description of the partitioning algorithm, implemented within the Marabou framework, and discuss its parameters and heuristic choices. In particular, we explore two novel partitioning strategies, that partition the input space or the phases of the neuron activations, respectively. We introduce a branching heuristic and a direction heuristic that are based on the notion of polarity. We also introduce a highly parallel pre-processing algorithm for simplifying neural network verification problems. An extensive experimental evaluation shows the benefit of these techniques on both existing and new benchmarks. A preliminary experiment with ultra-scaling our algorithm using a large distributed cloud-based platform also shows promising results.**

I. INTRODUCTION

Recent breakthroughs in machine learning, specifically the rise of *deep neural networks (DNNs)* [1], have expanded the horizon of real-world problems that can be tackled effectively. Increasingly, complex systems are created using machine learning models [2] instead of using conventional engineering approaches. Machine learning models are trained on a set of (labeled) examples, using algorithms that allow the model to capture their properties and generalize them to unseen inputs. In practice, DNNs can significantly outperform hand-crafted systems, especially in fields where precise problem formulation is challenging, such as image classification [3], speech recognition [4] and game playing [5].

Despite their overall success, the black-box nature of DNNs calls into question their trustworthiness and hinders their application in safety-critical domains. These limitations are exacerbated by the fact that DNNs are known to be vulnerable to *adversarial perturbations*, small modifications to the inputs that lead to wrong responses from the network [6], and real-world attacks have already been carried out against safety-critical deployments of DNNs [7, 8]. One promising approach for addressing these concerns is the use of formal methods to certify and/or obtain rigorous guarantees about DNN behavior.

Early work in DNN formal verification [9, 10] focused on translating DNNs and their properties into formats supported by existing verification tools like general-purpose *Satisfiability Modulo Theories* (SMT) solvers (e.g., Z3 [11], CVC4 [12]).

However, this approach was limited to small toy networks (roughly tens of nodes).

More recently, a number of DNN-specific approaches and solvers, including Reluplex [13], ReluVal [14], Neurify [15], Planet [16], and Marabou [17], have been proposed and developed. These techniques scale to hundreds or a few thousand nodes. While a significant improvement, this is still several orders of magnitude fewer than the number of nodes present in many real-world applications. Scalability thus continues to be a challenge and the subject of active research.

Inspired by recent successes with parallelizing SAT solvers [18, 19], we propose a set of strategies and heuristics for leveraging parallelism to improve the scalability of neural network verification. The paper makes the following contributions: 1) We present a divide-and-conquer algorithm for neural network verification that is parameterized by different partition strategies and constraint solvers (Sec. III). 2) We describe two partitioning strategies for this algorithm (Sec. III-B): one that works by partitioning the input domain and a second one that performs case splitting based on the activation functions in the neural network. The first strategy was briefly mentioned in the Marabou tool paper [17]; we describe it in detail here. The second strategy is new. 3) We introduce the notion of *polarity* and use it to refine the partitioning (Sec. III-C); 4) We introduce a highly parallelizable pre-processing algorithm that significantly simplifies verification problems (Sec. III-D); 5) We show how polarity can additionally be used to speed up satisfiable queries (Sec. III-E); and 6) We implement the techniques in the Marabou framework and evaluate on existing and new neural network verification benchmarks from the aviation domain. We also perform an *ultra-scalability* experiment using cloud computing (Sec. IV). Our experiments show that the new and improved Marabou can outperform the previous version of Marabou as well as other state-of-the-art verification tools such as Neurify, especially on perception networks with a large number of inputs. We begin with preliminaries, review related work in Sec. V, and conclude in Sec. VI.

II. PRELIMINARIES

In this section, we briefly review neural networks and their formalization, as well as the Reluplex algorithm for verification of neural networks.
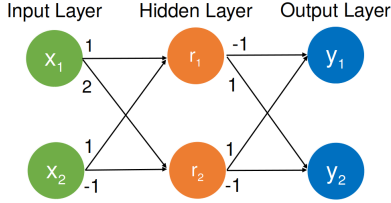
Fig. 1: A small feed-forward DNN $\mathcal{N}$.

A. Formalizing Neural Networks

Deep Neural Networks. A feed-forward *Deep Neural Network* (DNN) consists of a sequence of layers, including an input layer, an output layer, and one or more hidden layers in between. Each non-input layer comprises multiple *neurons*, whose values can be computed from the outputs of the preceding layer. Given an assignment of values to inputs, the output of the DNN can be computed by iteratively computing the values of neurons in each layer. Typically, a neuron's value is determined by computing an affine function of the outputs of the neurons in the previous layer and then applying a non-linear function, known as an *activation function*. A popular activation function is the Rectified Linear Unit (ReLU), defined as $ReLU(x) = max(0, x)$ (see [3, 20, 21]). In this paper, we focus on DNNs with ReLU activation functions; thus the output of each neuron is computed as $ReLU(w_1 \cdot v_1 + \ldots w_n \cdot v_n + b)$, where $v_1 \ldots v_n$ are the values of the previous layer's neurons, $w_1 \ldots w_n$ are the weight parameters, and b is a bias parameter associated with the neuron. A neuron is *active* or in the *active phase*, if its output is positive; otherwise, it is *inactive* or in the *inactive phase*.

Verification of Neural Networks. A neural network verification problem has two components: a neural network N, and a property P. P is often of the form $P_{in} \Rightarrow P_{out}$, where P_{in} is a formula over the inputs of N and P_{out} is a formula over the outputs of N. Typically, P_{in} defines an input region I, and P states that for each point in I, P_{out} holds for the output layer. Given a query like this, a verification tool tries to find a counter-example: an input point i in I, such that when applied to N, P_{out} is false over the resulting outputs. P holds only if such a counter-example does not exist.

The property to be verified may arise from the specific domain where the network is deployed. For instance, for networks that are used as controllers in an unmanned aircraft collision avoidance system (e.g., the ACAS Xu networks [13]), we would expect them to produce sensible advisories according to the location and the speed of the intruder planes in the vicinity. On the other hand, there are also properties that are generally desirable for a neural network. One such property is *local adversarial robustness* [22], which states that a small norm-bounded input perturbation should not cause major spikes in the network's output. More generally, a property may be an arbitrary formula over input values, output values, and values of hidden layers—such problems arise for example in the investigation of the neural networks' explainability [23], where one wants to check whether the activation of a certain

ReLU r implies a certain output behavior (e.g., the neural network always predicts a certain class). The verification of neural networks with ReLU functions is decidable and NP-Complete [13]. As with many other verification problems, scalability is a key challenge.

VNN Formulas. We introduce the notion of VNN (Verification of Neural Network) formulas to formalize Neural Network verification queries. Let $\mathcal{X}$ be a set of variables. A *linear constraint* is of the form $\sum_{x_i \in \mathcal{X}} a_i x_i \bowtie b$, where a_i, b are rational constants, and $\bowtie \in \{\leq, \geq, =\}$. A *ReLU constraint* is of the form $\text{ReLU}(x_i) = x_j$, where $x_i, x_j \in \mathcal{X}$.

Definition 1. *A VNN formula ϕ is a conjunction of linear constraints and ReLU constraints.*

A feed-forward neural network can be encoded as a VNN formula as follows. Each ReLU r is represented by introducing a pair of input/output variables r_b, r_f and then adding a ReLU constraint $\text{ReLU}(r_b) = r_f$. We refer to r_b as the *backward-facing variable*, and it is used to connect r to the preceding layer. r_f is called the *forward-facing variable* and is used to connect r to the next layer. The weighted sums are encoded as linear constraints.

In general, a property could be any formula P over the variables used to represent $\mathcal{N}$. To check whether P holds on $\mathcal{N}$, we simply conjoin the representation of $\mathcal{N}$ with the negation of P and use a constraint solver to check for satisfiability. P holds iff the constraint is unsatisfiable.

Note that a solver for VNN formulas can solve a property P only if the negation of P is also a VNN formula. We assume this is the case in this paper, but more general properties can be handled by decomposing $\neg P$ into a disjunction of VNN formulas and checking each separately (or, equivalently, using a DPLL(T) approach [24]). This works as long as the atomic constraints are linear. Non-linear constraints (other than ReLU) are beyond the scope of this paper.

B. The Reluplex Procedure

The Reluplex procedure [13] is a sound, complete and terminating algorithm that decides the satisfiability of a VNN formula. The procedure extends the Simplex algorithm—a standard efficient decision procedure for conjunctions of linear constraints—to handle ReLU constraints. At a high level, the algorithm iteratively searches for an assignment that satisfies all the linear constraints, but treats the ReLU constraints lazily in the hope that many of them will be irrelevant for proving the property. There are two ways to fix a violated ReLU constraint r: 1) *repair the assignment* by updating the assignment to forward-facing r_f or backward-facing variable r_b to satisfy r, or 2) *case split* by considering separate cases for each phase of r (adding the appropriate constraint in each case) In both cases, the search continues using the Simplex algorithm, in the first with a soft correction via assignment update and in the second by adding hard constraints to the linear problem. Lazy handling of ReLUs is achieved by the threshold parameter t — the number of times a ReLU is repaired before the algorithm performs a case split. In [13], this parameter was set to 20,
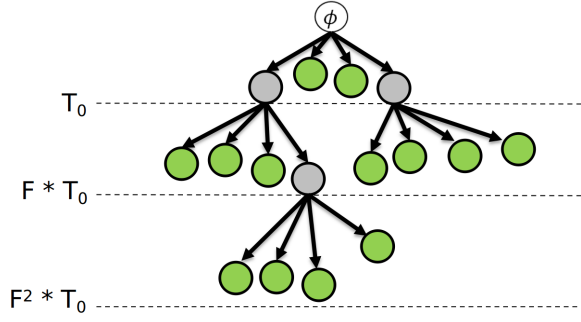
Fig. 2: An execution of the D&C algorithm.

Algorithm 1 Divide-and-Conquer

Input: query ϕ, initial partition size N_0, initial timeout T_0, partition size N, timeout factor F
Output: SAT/UNSAT
for $\psi \in \textbf{partition}(\phi, N_0)$ **do**
  $Q.\,\text{enqueue}(\langle \psi, T_0 \rangle)$
while $Q.\,\text{notEmpty}()$ **do**
  $\langle \phi', t \rangle \leftarrow Q.\,\text{dequeue}()$
  $result \leftarrow \textbf{solve}(\phi', t)$
  if $result = \text{SAT}$ **then**
    return SAT
  else if $result = \text{TIMEOUT}$ **then**
    for $\psi \in \textbf{partition}(\phi', N)$ **do**
      $Q.\,\text{enqueue}(\langle \psi, t \cdot F \rangle)$
return UNSAT

---

but even more eager splitting is beneficial in some cases. The Reluplex algorithm also uses bound propagation to fix ReLUs to one phase whenever possible.

In this paper, we explore heuristic choices behind the two options to handle violated ReLU constraints. In the case of assignment repair, the question is which variable assignment, r_f or r_b, to modify (often both are possible). We refer to the strategy used to make this decision as the *direction heuristic*, and we discuss direction heuristics, especially in the context of parallel solving in Sec. III-E. For case splitting, the question is which ReLU constraint to choose. We refer to the strategy used for making this decision as the *branching heuristic*. We explore branching heuristics and their application to parallelizing the algorithm in Sec. III-B and Sec. III-C.

III. D&C: PARALLELIZING THE RELUPLEX PROCEDURE

In this section, we present a parallel algorithm called *Divide-and-Conquer* (or simply D&C) for solving VNN formulas, using the Reluplex procedure and an iterative-deepening strategy. We discuss two partitioning strategies: input interval splitting and ReLU case splitting.

Remark. A divide-and-conquer approach with an input-splitting strategy was described in the Marabou tool paper [17], albeit briefly and informally. We provide here a more general divide-and-conquer framework, which includes new techniques and heuristics, described in detail below.

A. The D&C algorithm

The D&C algorithm partitions an input problem into several sub-problems (that are ideally easier to solve) and tries to solve each sub-problem within a given time budget. If solving a problem exceeds the time budget, that problem is further partitioned and the resulting sub-problems are allocated an increased time budget. Fig. 2 shows solving of problem ϕ as a tree, where the root of the tree denotes the original problem. Sub-problems that exceed their allotted time budget are partitioned, becoming inner nodes, and leaves are sub-problems solved within their time budget. A formula ϕ is satisfiable if some leaf is satisfiable. If the partitioning is *exhaustive*, that is: $\phi \coloneqq \bigvee_{\phi_i \in \textbf{partition}(\phi, n)} \phi_i$, for any $n > 1$, then ϕ is unsatisfiable iff all the leaves are unsatisfiable.

The pseudo-code of the D&C algorithm is shown in Algorithm 1, which can be seen as a framework parameterized by

the partitioning heuristic and the underlying solver. Details of these parameters are abstracted away within the **partition** and **solve** functions respectively and will be discussed in subsequent sections. The D&C algorithm takes as input the VNN formula ϕ and the following parameters: initial number of partitions N_0, initial timeout T_0, number of partitions N, and the timeout factor F. During solving, D&C maintains a queue Q of $\langle$query, timeout$\rangle$ pairs, which is initialized with the partition $N_0 \coloneqq \langle \phi, T_0 \rangle$. While the queue is not empty, the next pair $\langle \phi', t \rangle$ is retrieved from it, and the query ϕ' is solved with time budget t. If ϕ' is satisfiable, then the original query ϕ is satisfiable, and SAT is returned. If ϕ' times out, **partition**(ϕ', N) creates N sub-problems of ϕ', each of which is enqueued with an increased time budget $t \cdot F$. If the sub-problem ϕ' is unsatisfiable, no special action needs to be taken. If Q becomes empty, the original query is unsatisfiable and the algorithm returns UNSAT. Note that the main loop of the algorithm naturally lends itself to parallelization, since the **solve** calls are mutually independent and query-timeout pairs can be asynchronously enqueued and dequeued.

We state without proof the following result, which is a well-known property of such algorithms.

Theorem 1. *The Divide-and-Conquer(ϕ, N_0, T_0, N, F) algorithm is sound and complete if the following holds: 1) the* **solve** *function is sound and complete; and 2) the* **partition** *function is exhaustive.*

In addition, with modest assumptions on **solve** and **partition**, and with $F > 1$, the algorithm can be shown to be terminating. In particular, it is terminating for the instantiations we consider below. The D&C algorithm can be tailored to the available computing resources (e.g., number of processors) by specifying the number of initial splits N_0. The other three search parameters of D&C specify the dynamic behavior of the algorithm, e.g. if T_0 and F are small, or if N is large, then new sub-queries are created frequently, which entails a more aggressive D&C strategy (and vice versa). Notice that we can completely discard the dynamic aspect of D&C by setting the initial timeout to be ∞.

A potential downside of the algorithm is that each call to **solve** that times out is essentially wasted time, overhead above

3

and beyond the useful work needed to solve the problem. Fortunately, as the following theorem shows, the number of wasted calls is bounded.

Theorem 2. *When Algorithm 1 runs on an unsatisfiable formula with $N \leq N_0$, the fraction of calls to **solve** that time out is less than $\frac{1}{N}$.*

Proof. Consider first the case when $N = N_0$. We can view D&C's UNSAT proof as constructing an N-ary tree, as shown in Fig. 2. The ℓ leaf nodes are calls to **solve** that do not time out. The t non-leaves are calls to **solve** that do time out. Since this is a tree, the total number of nodes n is one more than the number of edges. Since each query that times out has an edge to each of its N sub-queries, the number of edges is Nt. Thus we have $n = Nt + 1$ which can be rearranged to show the fraction of queries that time out: $\frac{t}{n} = \frac{1 - 1/n}{N} < \frac{1}{N}$. If $N < N_0$, then let $k = N_0 - N$. The number of nodes is then $n = Nt + k + 1$, and the result follows as before. $\square$

B. Partitioning Strategies

A partitioning strategy specifies how to decompose a VNN formula to produce (hopefully easier) sub-problems.

A ReLU is *fixed* when the bounds on the backward-facing or forward-facing variable either imply that the ReLU is active or imply that the ReLU is inactive. Fixing as many ReLUs as possible reduces the complexity of the resulting problem.

With these concepts in mind, we present two strategies: 1) *input-based partitioning* creates case splits over the ranges of input variables, relying on bound propagation to fix ReLUs, whereas 2) *ReLU-based partitioning* creates case splits that fix the phase of ReLUs directly. Both strategies are exhaustive, ensuring soundness and completeness of the D&C algorithm (by Theorem 1). The *branching heuristic* which determines the choice of input variable, respectively ReLU, on which to split, can have a significant impact on performance. The branching heuristic should keep the total runtime of the sub-problems low as well as achieve a good *balance* between them. To illustrate, suppose the sub-problems created by splitting ReLU_1 take 10 and 300 seconds to solve, whereas those created by splitting ReLU_2 take 150 and 160 seconds to solve. Though the total solving time is the same, the more balanced split, on ReLU_2, results in shorter wall-clock time (given two parallel workers).

If most splits led to easier and balanced sub-formulas, then D&C would perform well, even without a carefully-designed branching heuristic. However, we have observed that this is not the case for many possible splits: the time taken to solve one (or both!) of the sub-problems generated by such splits is comparable to that required by the original formula (or even worse). Therefore, an effective branching heuristic is crucial. We describe two such heuristics below.

Input-based Partitioning. This simple partitioning strategy performs case splits over the range of an input variable. As an example, consider a VNN formula $\phi := \phi' \wedge (-2 \leq x_1 \leq 1) \wedge (-2 \leq x_2 \leq 2)$, where x_1 and x_2 are the two input variables of a neural network encoded by ϕ'. Suppose we call **partition**$(\phi, 2)$ using the input-splitting strategy. The choice

is between splitting on the range of x_1 or the range of x_2. If we choose x_1, the result is two sub-formulas, ϕ_1 and ϕ_2, where: $\phi_1 := \phi' \wedge (-\mathbf{2} \leq \mathbf{x_1} < -\mathbf{0.5}) \wedge (-2 \leq x_2 \leq 2)$ and $\phi_2 := \phi' \wedge (-\mathbf{0.5} \leq \mathbf{x_1} \leq \mathbf{1}) \wedge (-2 \leq x_2 \leq 2)$. An obvious heuristic is to choose the input with largest range. A more complex heuristic was introduced in [17]. It samples the network repeatedly, which requires considerable overhead. In fact, both of these heuristics perform reasonably well on benchmarks with only a few inputs (the ACAS Xu benchmarks, for example). Unfortunately, regardless of the heuristic used, this strategy suffers from the "curse of dimensionality" — with a large number of inputs it becomes increasingly difficult to fix ReLUs by splitting the range of only one input variable. Thus, the input-partitioning strategy does not scale well on such networks (e.g., perception networks), which often have hundreds or thousands of inputs.

ReLU-based Partitioning. A complementary strategy is to partition the search space by fixing ReLUs directly. Consider a VNN formula $\phi := \phi' \wedge (\text{ReLU}(x) = y)$. A call to **partition**$(\phi, 2)$ using the ReLU-based strategy results in two sub-formulas ϕ_1 and ϕ_2, where $\phi_1 := \phi' \wedge (\text{ReLU}(x) = y) \wedge (\mathbf{x} \leq \mathbf{0}) \wedge (\mathbf{y} = \mathbf{0})$ and $\phi_2 := \phi' \wedge (\text{ReLU}(x) = y) \wedge (\mathbf{x} > \mathbf{0}) \wedge (\mathbf{x} = \mathbf{y})$. Note that here, ϕ_1 is capturing the inactive and ϕ_2 the active phase of the ReLU. Next, we consider a heuristic for choosing a ReLU to split on.

C. Polarity-based Branching Heuristics

We want to estimate the difficulty of sub-problems created by a partitioning strategy. One key related metric is the number of bounds that can be tightened as the result of a ReLU-split. As a light-weight proxy for this metric, we propose a metric called *polarity*.

Definition 2. *Given the ReLU constraint ReLU$(x) = y$, and the bounds $a \leq x \leq b$, where $a < 0$, and $b > 0$, the polarity of the ReLU is defined as: $p = \frac{a+b}{b-a}$.*

Polarity ranges from -1 to 1 and measures the symmetry of a ReLU's bounds with respect to zero. For example, if we split on a ReLU constraint with polarity close to 1, the bound on the forward-facing variable in the active case, $[0, b]$, will be much wider than in the inactive case, $[a, 0]$. Intuitively, forward bound tightening would therefore result in tighter bounds in the inactive case. This means the inactive case will probably be much easier than the active case, so the partition is unbalanced and therefore undesirable. On the other hand, a ReLU with a polarity close to 0 is more likely to have balanced sub-problems. We also observe that ReLUs in early hidden layers are more likely to produce bound tightening by forward bound propagation (as there are more ReLUs that depend on them).

We thus propose a heuristic that picks the ReLU whose polarity is closest to 0 among the first $k\%$ unfixed ReLUs, where k is a configurable parameter. Note that, in order to compute polarities, we need all input variables to be bounded, which is a reasonable assumption.

Algorithm 2 Iterative Propagation
___

Input: VNN query ϕ, timeout t
Output: preprocessed query ϕ'.
  $progress \leftarrow \top$; $\phi' \leftarrow \phi$
  while $progress = \top$ **do**
    $progress \leftarrow \bot$
    for r in **getUnfixedReLUs**(ϕ') **do**
      $\psi \leftarrow$ **polarityConstraint**(r)
      $result =$ **solve**$(\phi' \wedge \psi, t)$
      if $result =$ UNSAT **then**
        $\psi' \leftarrow$ **flipPhase**(ψ)
        $\phi' \leftarrow \phi' \wedge \psi'$
        $progress \leftarrow \top$
  return ϕ'
___

D. Fixing ReLU Constraints with Iterative Propagation

As discussed earlier, the performance of D&C depends heavily on the ability to split on ReLUs that result in balanced sub-formulas. However, sometimes a considerable portion of ReLUs in a given neural network cannot be split in this way. To eliminate such ReLUs we propose a preprocessing technique called *iterative propagation*, which aims to discover and fix ReLUs with unbalanced partitions.

Concretely, for each ReLU in the VNN formula, we temporarily fix the ReLU to one of its phases and then attempt to solve the problem with a short timeout. The goal is to detect unbalanced and (hopefully) easy unsatisfiable cases. Pseudocode is presented in Algorithm 2. The algorithm takes as input the formula ϕ and the timeout t, and, if successful, returns the equivalent formula ϕ' which has fewer unfixed ReLUs than ϕ. The outer loop computes the fixed point, while the inner loop iterates through the as-of-yet unfixed ReLUs. For each unfixed ReLU, the **polarityConstraint** function returns a constraint for the phase estimated to be easier using the polarity metric. If the solver returns UNSAT, then we can safely fix the ReLU to its other phase using the **flipPhase** function. We ignore the case where the solver returns SAT, since in practice this only occurs for formulas that are very easy in the first place.

Iterative propagation complements D&C, because the likelihood of finding balanced partitions is increased by fixing ReLUs that lead to unbalanced partitions. Moreover, iterative propagation is highly parallelizable, as each ReLU-fixing attempt can be solved independently. In Section IV, we report results using iterative propagation as a preprocessing step, though it is possible to integrate the two processes more closely, e.g., by performing iterative propagation after every **partition** call.

E. Speeding Up Satisfiable Checks with Polarity-Based Direction Heuristics

In this section, we discuss how the polarity metric introduced in Sec. III-C can be used to solve satisfiable instances quickly. When splitting on a ReLU, the Reluplex algorithm faces the same choice as the D&C algorithm. For unsatisfiable cases, the order in which ReLU case splits are done make little difference on average, but for satisfiable instances, it can

be very beneficial if the algorithm is able to hone in on a satisfiable sub-problem. We refer to the strategy for picking which ReLU phase to split on first as the *direction heuristic*.

We propose using the polarity metric to guide the direction heuristic for D&C. If the polarity of a branching ReLU is positive, then we process the active phase first; if the polarity is negative, we do the reverse. Intuitively, formulas with wider bounds are more likely to be satisfiable, and the polarity direction heuristic prefers the phase corresponding to wider bounds for the ReLU's backward-facing variable.

Repairing an assignment when a ReLU is violated can also be guided by polarity (recall the description of the Reluplex procedure from Sec. II), as choosing between forward- or backward-facing variables amounts to choosing which ReLU phase to explore first. Therefore, we use this same direction heuristic to guide the choice of forward- or backward-facing variables when repairing the assignment. For example, suppose constraint $\text{ReLU}(x_b) = x_f$ is part of a VNN formula ϕ. Suppose the range of x_b is $[-2, 1]$, $\mathsf{A}(x_b) = -1$ and $\mathsf{A}(x_f) = 1$, where A is the current variable assignment computed by the Simplex algorithm. To repair this violated ReLU constraint, we can either assign 0 to x_f or assign 1 to x_b. In this case, the ReLU has negative polarity, meaning the negative phase is associated with wider input bounds, so our heuristic chooses to set $\mathsf{A}(x_f) = 0$.

We will see in our experimental results (Sec. IV) that these direction heuristics improve performance on satisfiable instances. Interestingly, they also have a positive performance impact on unsatisfiable instances.

IV. EXPERIMENTAL EVALUATION

In this section, we discuss our implementation of the proposed techniques and evaluate its performance on a diverse set of real-world benchmarks – safety properties of control systems and robustness properties of perception models.

A. Implementation

We implemented the techniques discussed above in Marabou [17], which is an open-source neural network verification tool implementing the Reluplex algorithm. The tool also integrates the symbolic bound tightening techniques introduced in [14]. We refer to Marabou running the D&C algorithm as D&C-Marabou. Two partitioning strategies are supported: the original input-based partitioning strategy and our new ReLU-splitting strategy. All D&C configurations use the following parameters: the initial partition size N_0 is the number of available processors; the initial timeout T_0 is 10% of the network size in seconds; the number of online partitions N is 4; and the timeout factor F is 1.5. The k parameter for the direction heuristic (see Sec. III-C) is set to 5. The per-ReLU timeout for iterative propagation is 2 seconds. When the input dimension is low (≤ 10), symbolic bound tightening is turned on, and the threshold parameter t (see Sec. II) is reduced from 20 to 1. The parameters were chosen using a grid search on a small subset of benchmarks.

B. Benchmarks

The benchmark set consists of network-property pairs, with networks from three different application domains: aircraft collision avoidance (ACAS Xu), aircraft localization (Tiny-TaxiNet), and digit recognition (MNIST). Properties include robustness and domain-specific safety properties.

ACAS Xu. The ACAS Xu family of VNN benchmarks was introduced in [13] and uses prototype neural networks trained to represent an early version of the ACAS Xu decision logic [2]. The ACAS Xu benchmarks are composed of 45 fully-connected feed-forward neural networks, each with 6 hidden layers and 50 ReLU nodes per layer. The networks issue turning advisories to the controller of an unmanned aircraft to avoid near midair collisions. The network has 5 inputs (encoding the relation of the ownship to an intruder) and 5 outputs (denoting advisories: e.g., weak left, strong right). Proving that the network does not produce erroneous advisories is paramount for ensuring safe aviation operation. We consider four realistic properties expected of the 45 networks. These properties, numbered 1–4, are described in [13].

TinyTaxiNet. The TinyTaxiNet family contains perception networks used in vision-based *autonomous taxiing*: the task of predicting the position and orientation of an aircraft on the taxiway, so that a controller can accurately adjust the position of the aircraft [25]. The input to the network is a downsampled grey-scale image of the taxiway captured from a camera on the aircraft. The network produces two outputs: the lateral distance to the runway centerline, and the heading angle error with respect to the centerline. Proving that the networks accurately predict the location of the aircraft even when the camera image suffers from small noise is safety-critical. This property can be captured as local adversarial robustness. If the kth output of the network is expected to be b_k for inputs near $\mathbf{a}$, we can check the unsatisfiability of the following VNN formula:

$$(y_k \geq b_k + \epsilon) \wedge \bigwedge_{i=1}^{N} (a_i - \delta \leq x_i \leq a_i + \delta),$$

where $\mathbf{x}$ denotes the actual network input, N the number of network inputs, and y_k the actual k^{th} output. The network is (δ, ϵ)-locally robust on $\mathbf{a}$, only if the formula is unsatisfiable. The training images are compressed to either 2048 or 128 pixels, with value range [0,1]. We evaluate the local adversarial robustness of two networks. TaxiNet1 has 2048 inputs, 1 convolutional layer, 2 feedforward layers, and 128 ReLUs. TaxiNet2 has 128 inputs, 5 convolutional layers, and a total of 176 ReLUs. For each network, we generate 100 local adversarial robustness queries concerning the first output (distance to the centerline). For each model, we sample 100 uniformly random images from the training data, and sample (δ, ϵ) pairs uniformly from the set $\{\langle 0.004, 3\rangle, \langle 0.004, 9\rangle, \langle 0.008, 3\rangle, \langle 0.008, 9\rangle, \langle 0.016, 9\rangle\}$. Setting $\delta = 0.004$ allows a 1 pixel-value perturbation in pixel brightness along each input dimension, and the units of ϵ are meters.

MNIST. In addition to the two neural network families with safety-critical real-world applications, we evaluate our techniques on three fully-connected feed-forward neural networks (MNIST1, MNIST2, MNIST3) trained on the MNIST dataset [26] to classify hand-written digits. Each network has 784 inputs (representing a grey-scale image) with value range [0,1], and 10 outputs (each representing a digit). MNIST1 has 10 hidden layers and 10 neurons per layer; MNIST2 has 10 hidden layers and 20 neurons per layer; MNIST3 has 20 hidden layers and 20 neurons per layer. We consider *targeted robustness* queries, which asks whether, for an input $\mathbf{x}$ and an incorrect output y', there exists a point in the ℓ^∞ δ-ball around x that is classified as y'. We sample 100 such queries for each network, by choosing random training images and random incorrect labels. We choose δ values evenly from $\{0.004, 0.008, 0.0016, 0.0032\}$.

C. Experimental Evaluation

We present the results of the following experiments: 1) Evaluation of each technique's effect on run-time performance of Marabou on the three benchmark sets. We also compare against Neurify, a state-of-the-art solver on the same benchmarks. 2) An analysis of trade-offs when running iterative propagation pre-processing. 3) Exploration of D&C scalability at a large scale, using cloud computing. More details about the experimental setup and results are shown in the Appendix.

1) Evaluation of the techniques on ACAS Xu, TinyTaxiNet, MNIST : We denote the ReLU-based partitioning strategy as **R**, polarity-based direction heuristics as **D**, and iterative propagation as **P**. We denote as **S** a hybrid strategy that uses input-based partitioning on ACAS Xu networks, and ReLU-based partitioning on perception networks. We run four combinations of our techniques: 1) **R**; 2) **S+D**; 3) **S+P**; 4) **S+D+P**, and compare them with two baseline configurations: 1) the sequential mode of Marabou (denoted as **M**); 2) D&C-Marabou with its default input-based partitioning strategy (denoted as **I**).

We compare with Neurify [15], a state-of-the-art solver, on the same benchmarks. Neurify derives over-approximations of the output bounds using techniques such as symbolic interval analysis and linear relaxation. On ACAS Xu benchmarks, it operates by iteratively partitioning the input region to reduce error in the over-approximated bounds (to prove UNSAT) and by randomly sampling points in the input region (to prove SAT). On other networks, Neurify uses off-the-shelf solvers to handle ReLU-nodes whose bounds are potentially overestimated. Neurify also leverages parallelism, as different input regions or linear programs can be checked in parallel.

We run all Marabou configurations and Neurify on a cluster equipped with Intel Xeon E5-2699 v4 CPUs running CentOS 7.7. 8 cores and 64GB RAM are allocated for each job, except for the **M** configuration, which uses 1 processor and 8GB RAM per job. Each job is given a 1-hour wall-clock timeout.

Results. Table I shows a breakdown of the number of solved instances and the run-time for all Marabou configurations and for Neurify. We group the results by SAT and UNSAT instances. For each row, we highlight the entries corresponding to the configuration that solves the most instances (ties broken by run-time). Here are some key observations:

TABLE I: Evaluation of the Techniques on ACAS Xu, TinyTaxiNet, MNIST

Bench.	M		I		R		S		S+D		S+P		S+D+P		Neurify	
[# inst.]	#S	Time	#S	Time	#S	Time	#S	Time	#S	Time	#S	Time	#S	Time	#S	Time
ACAS	40	17224	**45**	**4884**	**45**	5009	**45**	**4884**	45	5480	**45**	8419	**45**	7244	39	4167
[180]	101	57398	130	48954	125	45036	130	48954	131	51413	130	50828	131	53717	**133**	1438
TinyTaxi.	34	4591	34	1815	34	433	34	433	34	419	34	533	**35**	1172	**35**	**88**
[200]	141	33909	110	24088	147	23079	147	23079	147	22345	**149**	**20583**	149	21949	146	7158
MNIST	11	2349	19	13032	22	9680	22	9680	26	11727	20	9956	**29**	19351	27	151
[300]	140	64418	78	27134	181	52776	181	52776	183	59195	184	67625	**185**	68307	153	10640
All	85	24164	98	19731	101	15122	101	14997	105	17626	99	18908	**109**	27767	101	4406
[680]	382	155725	318	100176	453	120891	458	124809	461	132953	463	139036	**465**	143973	432	19236

Number of solved instances (#S) and run-time in seconds of different configurations. For each benchmark set, top and bottom rows show data for satisfiable (SAT) and unsatisfiable (UNSAT) instances respectively. The results for configuration **S** are computed virtually from **R** and I.

– On ACAS Xu benchmarks, both input-based partitioning (**I**) and ReLU-based partitioning (**R**) yield performance gain compared with the sequential solver (**M**), with **I** being more effective. On perception networks, **I** solves significantly fewer instances than **M** while **R** continues to be effective.

– Comparing the performance of **S**, **S+D**, and **S+P** suggests that the polarity-based direction heuristics and iterative propagation each improve the overall performance of D&C-Marabou. Interestingly, the polarity-based heuristic improves the performance on not only SAT but also UNSAT instances, suggesting that by affecting how ReLU constraints are repaired, direction heuristics also favorably impact the order of ReLU-splitting. On the other hand, iterative propagation alone only improves performance on UNSAT instances. **S+D+P** solves the most instances among all the Marabou configurations, indicating that the direction heuristics and iterative propagation are complementary to each other.

– **S+D+P** solves significantly more instances than Neurify overall. While Neurify's strategy on Acas Xu benchmarks allows it to dedicate more time on proving UNSAT by rapidly partitioning the input region (thus yielding much shorter run-times than **S+D+P** on that benchmark set), its performance on SAT instances is subject to (un)lucky guesses. When it comes to perception neural networks that are deeper and have higher input dimensions, symbolic bound propagation, on which Neurify heavily relies, becomes more expensive and less effective. By contrast, Marabou does not rely solely on symbolic interval analysis, but in addition uses interval bound-tightening techniques (see [17] for details).

Fig. 3 shows a cactus plot of the 6 Marabou configurations and Neurify on all benchmarks. In this plot, we also include two virtual portfolio configurations: **Virt.-Marabou** takes the best run-time among all Marabou configurations for each benchmark, and **Virt.-All** includes Neurify in the portfolio. Interestingly, **S+D+P** is outperformed by **S+D** in the beginning but surpasses **S+D** after 500 seconds. This suggests that iterative propagation creates overhead for easy instances, but benefits the search in the long run. We also observe that Neurify can solve a subset of the benchmarks very rapidly, but solves very few benchmarks after 1500 seconds. One possible explanation is that Neurify splits the input region and makes solver calls eagerly. While this allows it to resolve some
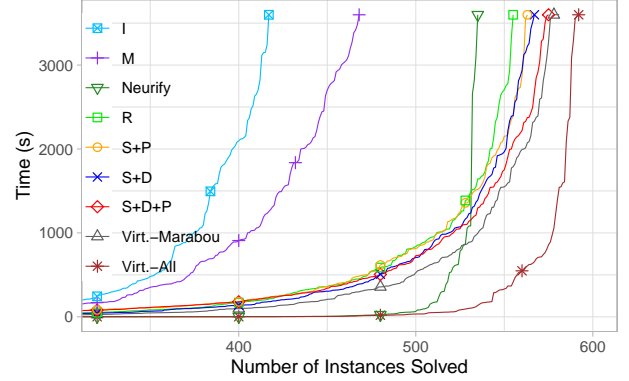

Fig. 3 Cactus plot: all solvers + two virtual best configurations.
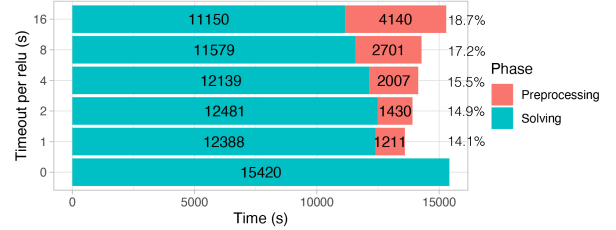

Fig. 4 The effect of varying per-ReLU timeout in preprocessing.

queries quickly, it also results in rapid (exponential) growth of the number of sub-regions and solver calls. By contrast, Marabou splits lazily. While it creates overhead sometimes, it results in more solved instances overall. The **Virt.-All** configuration solves significantly more instances than **Virt.-Marabou**, suggesting that the two procedures are complementary to each other. We note that the bound tightening techniques presented in Neurify can be potentially integrated into Marabou, and the polarity-based heuristics and iterative propagation could also be used to improve Neurify and other VNN tools.

2) Costs of Iterative Propagation: As mentioned in Sec. 2, intuitively, the longer the time budget during iterative propagation, the more ReLUs should get fixed. To investigate this trade-off between the number of fixed ReLUs and the overhead, we choose a smaller set of benchmarks (40 ACAS Xu benchmarks, 40 TinyTaxiNet benchmarks, and 40 MNIST benchmarks), and vary the timeout parameter t of iterative propagation. Each job is run with 32 cores, and a wall-clock timeout of 1 hour, on the same cluster as in Experiment IV-C1.
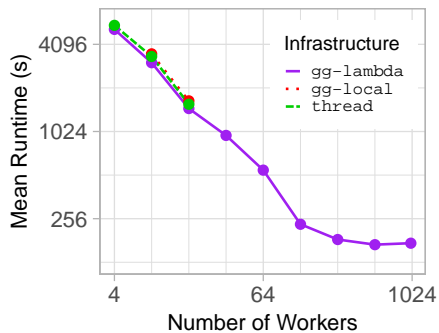
Fig. 5: Ultra-Scalability of D&C.

Results. Fig. 4 shows the preprocessing time + solving time of different configurations on commonly solved instances. The percentage next to each bar represents the average percentage of fixed ReLUs. Though the run-time and unfixed ReLUs continue to decrease as we invest more in iterative propagation, performing iterative propagation no longer provides performance gain when the per-ReLU-timeout exceeds 8 seconds.

3) Ultra-Scalability of D&C: D&C-Marabou runs on a single machine, which intrinsically limits its scalability to the number of hardware threads. To investigate how the D&C algorithm scales with much higher degrees of parallelism, we implemented it on top of gg [27], a platform for expressing parallelizable computations and executing them either *locally*, using different processes, or *remotely*, using cloud services. We call this implementation of D&C, *gg-Marabou.*

We measure the performance of D&C and gg-Marabou at varying levels of parallelism to establish that they perform similarly and to evaluate the scalability of the D&C algorithm. Our experiments use three underlying infrastructures: D&C-Marabou (denoted `thread`), gg-Marabou executed locally (`gg-local`), and gg-Marabou executed remotely on AWS Lambda [28] (`gg-lambda`). We vary the parallelism level, p, from 4 to 16 for the local infrastructures and from 4 to 1000 for `gg-lambda`. For `gg-lambda`, we run 3 tests per benchmark, taking the median time to mitigate variation from the network. From the UNSAT ACAS Xu benchmarks which D&C-Marabou can solve in under two hours using 4 cores, we chose 5 of the hardest instances. We set $T_0 = 5$ s, $F = 1.5$, $N = 2^{\lfloor (5 + \log_2 p)/3 \rfloor}$ and use the input-based partitioning strategy.

Results. Fig. 5 shows how mean runtime (across benchmarks) varies with parallelism level and infrastructure. Our first conclusion from Fig. 5 is that gg does **not** introduce significant overhead; at equal parallelism levels, all infrastructures perform similarly. Our second conclusion is that gg-Marabou scales well up to over a hundred workers. This is shown by the constant slope of the runtime/parallelism level line up to over a hundred workers. We note that the slope only flattens when total runtime is small: a few minutes.

V. RELATED WORK

Over the past few years, a number of tools for verifying neural network have emerged and broadly fall into two categories — precise and abstraction-based methods. Precise approaches are complete and usually encode the problem as an SAT/SMT/MILP constraint [13, 17, 29, 30]. Abstraction-based methods are not necessarily complete and abstract the search space using intervals [14, 15] or more complex abstract domains [31]–[33]. However, most of these approaches are sequential, and for details, we refer the reader to the survey by Liu et al. [34]. To the best of our knowledge, only Marabou [17] and Neurify [15] (and its predecessor Relu-Val [14]) leverage parallel computing to speed up verification.

As mentioned in Sec. IV, Neurify combines symbolic interval analysis with linear relaxation to compute tighter output bounds and uses off-the-shelf solvers to derive more precise bounds for ReLUs. These interval analysis techniques lend themselves well to parallelization, as independent linear programs can be created and checked in parallel. By contrast, D&C-Marabou creates partitions of the original query and solves them in parallel. Neurify supports a selection of hard-coded benchmarks and properties and often requires modifications to support new properties, while Marabou provides verification support for a wide range of properties.

Our work is inspired by the *Cube-and-Conquer* algorithm [18], which targets very hard SAT problems. Cube-and-Conquer is a divide-and-conquer technique that partitions a Boolean satisfiability problem into sub-problems by conjoining cubes —a cube is a conjunction of propositional literals— to the original problem and then employing a conflict-driven SAT solver [35] to solve each sub-problem in parallel. The propositional literals used in cubes are chosen using look-ahead [36] techniques. Our approach uses similar ideas, but in the VNN domain.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a set of techniques that leverage parallel computing to improve the scalability of neural network verification. We described an algorithm based on partitioning the verification problem in an iterative manner and explored two strategies that work by partitioning the input space or by splitting on ReLUs, respectively. We introduced a branching heuristic and a direction heuristic, both based on the notion of polarity. We also introduced a highly parallelizable pre-processing algorithm for simplifying neural network verification problems. Our experimental evaluation shows the benefit of these techniques on existing and new benchmarks. A preliminary experiment with ultra-scaling using the gg platform on Amazon Lambda also shows promising results.

Future work includes: i) Investigating more dynamic strategies for choosing hyper-parameters of the D&C framework. ii) Investigating different ways to interleave iterative propagation with D&C. iii) Investigating the scalability of ReLU-based partitioning to high levels of parallelism. iv) Improving the performance of the underlying solver, Marabou, by integrating conflict analysis (as in CDCL SAT solvers and SMT solvers) and more advanced bound propagation techniques such as those used by Neurify. v) Extending the techniques to handle

other piecewise-linear activation functions such as hard tanh and leaky ReLU, to which the notion of polarity applies.

REFERENCES

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[2] K. D. Julian, M. J. Kochenderfer, and M. P. Owen, "Deep neural network compression for aircraft collision avoidance systems," *Journal of Guidance, Control, and Dynamics*, vol. 42, no. 3, pp. 598–608, 2019. [Online]. Available: https://doi.org/10.2514/1.G003724

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *NIPS*, 2012, pp. 1106–1114.

[4] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal processing magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[5] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, p. 484, 2016.

[6] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. J. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," in *ICLR (Poster)*, 2014.

[7] M. Cissé, Y. Adi, N. Neverova, and J. Keshet, "Houdini: Fooling deep structured prediction models," *CoRR*, vol. abs/1707.05373, 2017.

[8] A. Kurakin, I. J. Goodfellow, and S. Bengio, "Adversarial examples in the physical world," in *ICLR (Workshop)*. OpenReview.net, 2017.

[9] L. Pulina and A. Tacchella, "An abstraction-refinement approach to verification of artificial neural networks," in *CAV*, ser. Lecture Notes in Computer Science, vol. 6174. Springer, 2010, pp. 243–257.

[10] ——, "Challenging SMT solvers to verify neural networks," *AI Commun.*, vol. 25, no. 2, pp. 117–135, 2012.

[11] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *TACAS*, ser. Lecture Notes in Computer Science, vol. 4963. Springer, 2008, pp. 337–340.

[12] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *CAV*, ser. Lecture Notes in Computer Science, vol. 6806. Springer, 2011, pp. 171–177.

[13] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer, "Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks," in *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*, 2017, pp. 97–117.

[14] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana, "Formal security analysis of neural networks using symbolic intervals," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 1599–1614.

[15] ——, "Efficient formal safety analysis of neural networks," in *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*, 2018, pp. 6369–6379. [Online]. Available: http://papers.nips.cc/paper/7873-efficient-formal-safety-analysis-of-neural-networks

[16] R. Ehlers, "Formal verification of piece-wise linear feed-forward neural networks," in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2017, pp. 269–286.

[17] 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*, 2019, pp. 443–452.

[18] M. Heule, O. Kullmann, S. Wieringa, and A. Biere, "Cube and conquer: Guiding CDCL SAT solvers by lookaheads," in *Haifa Verification Conference*, ser. Lecture Notes in Computer Science, vol. 7261. Springer, 2011, pp. 50–65.

[19] M. J. H. Heule, O. Kullmann, and V. W. Marek, "Solving and verifying the boolean pythagorean triples problem via cube-and-conquer," in *SAT*, ser. Lecture Notes in Computer Science, vol. 9710. Springer, 2016, pp. 228–245.

[20] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *ICML*. Omnipress, 2010, pp. 807–814.

[21] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *Proc. icml*, vol. 30, no. 1, 2013, p. 3.

[22] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Towards proving the adversarial robustness of deep neural networks," *arXiv preprint arXiv:1709.02802*, 2017.

[23] D. Gopinath, H. Converse, C. Pasareanu, and A. Taly, "Property inference for deep neural networks," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov 2019, pp. 797–809.

[24] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(*t*)," *J. ACM*, vol. 53, no. 6, pp. 937–977, 2006.

[25] K. D. Julian, R. Lee, and M. J. Kochenderfer, "Validation of image-based neural network controllers through adaptive stress testing," *arXiv preprint arXiv:2003.02381*, 2020.

[26] "The MNIST database of handwritten digits Home Page," http://yann.lecun.com/exdb/mnist/.

[27] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. Winstein, "From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers," in *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, 2019, pp. 475–488. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/fouladi

[28] "AWS lambda," https://docs.aws.amazon.com/lambda/index.html.

[29] R. Ehlers, "Formal verification of piece-wise linear feed-forward neural networks," *CoRR*, vol. abs/1705.01320, 2017. [Online]. Available: http://arxiv.org/abs/1705.01320

[30] N. Narodytska, S. Kasiviswanathan, L. Ryzhyk, M. Sagiv, and T. Walsh, "Verifying properties of binarized deep neural networks," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[31] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev, "Ai2: Safety and robustness certification of neural networks with abstract interpretation," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 3–18.

[32] G. Singh, T. Gehr, M. Mirman, M. Püschel, and M. Vechev, "Fast and effective robustness certification," in *Advances in Neural Information Processing Systems*, 2018, pp. 10 802–10 813.

[33] G. Singh, T. Gehr, M. Püschel, and M. Vechev, "An abstract domain for certifying neural networks," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–30, 2019.

[34] C. Liu, T. Arnon, C. Lazarus, C. Barrett, and M. J. Kochenderfer, "Algorithms for verifying deep neural networks," 2019.

[35] J. P. M. Silva, I. Lynce, and S. Malik, "Conflict-driven clause learning SAT solvers," in *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009, vol. 185, pp. 131–153.

[36] M. Heule and H. van Maaren, "Look-ahead based SAT solvers," in *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009, vol. 185, pp. 155–184.

APPENDIX

A. More Details on gg-Marabou

The `gg` platform is a tool for expressing parallelizable computations and executing them. To use it, the programmer expresses their computation as *task graph*: a dependency graph of tasks, where each task is an executable program (e.g., a binary or shell script) that reads some input files and produces some output files. These output files can encode the result of the task, or an extension to the task graph that must be executed in order to produce that result. In our implementation of the D&C algorithm on top of `gg`, each task runs the base solver with a timeout. If the solver completes, the task returns the result, otherwise it returns a task graph extension encoding the division of the problem into sub-queries.

The local part of the `gg` experiment is run on a machine with 24 Xeon E5-2687W v4 CPUs, 132GB RAM, running Ubuntu 20.04.

B. More Details on Evaluation of Techniques

We present here a more detailed report of the runtime performance of different configurations and Neurify, as shown in Table II. We break down the ACAS Xu benchmark family by properties, and the other two benchmark sets by networks.

Fig. 6 shows the log-scaled pairwise comparisons between different configurations.

Bench. [# inst.]	M S	M T	I S	I T	R S	R T	S+D S	S+D T	S+P S	S+P T	S+D+P S	S+D+P T	Neurify S	Neurify T
ACAS1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
45	17	32455	42	37125	37	33141	43	40936	42	36783	43	40107	**45**	**558**
ACAS2	34	17210	**39**	**4863**	**39**	4985	**39**	5456	**39**	8228	**39**	7074	33	4167
45	0	0	4	5461	4	5121	4	4042	4	4070	4	4156	**4**	**88**
ACAS3	**3**	9	**3**	10	**3**	12	**3**	13	**3**	91	**3**	83	**3**	**0**
45	**42**	18254	**42**	4900	**42**	4569	**42**	4826	**42**	6571	**42**	6295	**42**	**742**
ACAS4	**3**	5	**3**	11	**3**	12	**3**	11	**3**	100	**3**	87	**3**	**0**
45	**42**	6689	**42**	1468	**42**	2205	**42**	1609	**42**	3404	**42**	3159	**42**	**49**
ACAS	40	17224	**45**	**4884**	**45**	5009	**45**	5480	**45**	8419	**45**	7244	39	4167
180	101	57398	130	48954	125	45036	131	51413	130	50828	131	53717	**133**	**1438**
TinyTaxiNet1	**11**	1168	**11**	1579	**11**	370	**11**	356	**11**	337	**11**	357	**11**	**85**
100	84	27773	63	17883	89	14052	89	12521	89	14651	**89**	**14683**	81	7148
TinyTaxiNet2	23	3423	23	236	23	63	23	63	23	196	24	815	**24**	**3**
100	57	6136	47	6205	58	9027	58	9824	60	5932	60	7266	**65**	**10**
TinyTaxiNet	34	4591	34	1815	34	433	34	419	34	533	**35**	1172	**35**	**88**
200	141	33909	110	24088	147	23079	147	22345	149	20583	**149**	**21949**	146	7158
MNIST1	9	2178	11	3658	12	2190	12	1412	12	3682	**13**	**5715**	8	108
100	73	11880	47	12387	**80**	**12999**	80	15213	80	14090	80	13571	54	2285
MNIST2	2	171	5	3494	6	3246	7	3787	4	1782	9	9140	**13**	**6**
100	37	22069	17	5698	46	14576	46	14833	**48**	**19026**	47	15141	45	3247
MNIST3	0	0	3	5880	4	4244	**7**	6528	4	4492	**7**	**4496**	6	36
100	30	30469	14	9049	55	25201	57	29149	56	34509	**58**	**39595**	54	5108
MNIST	11	2349	19	13032	22	9680	26	11727	20	9956	**29**	**19351**	27	151
300	140	64418	78	27134	181	52776	183	59195	184	67625	**185**	**68307**	153	10640
All	85	24164	98	19731	101	15122	105	17626	99	18908	**109**	**27767**	101	4406
680	382	155725	318	100176	453	120891	461	132953	463	139036	**465**	**143973**	432	19236

TABLE II: Number of solved instances (S) and run time in seconds (T) of different configurations. For each family, top and bottom rows show data for satisfiable (SAT) and unsatisfiable (UNSAT) instances respectively.
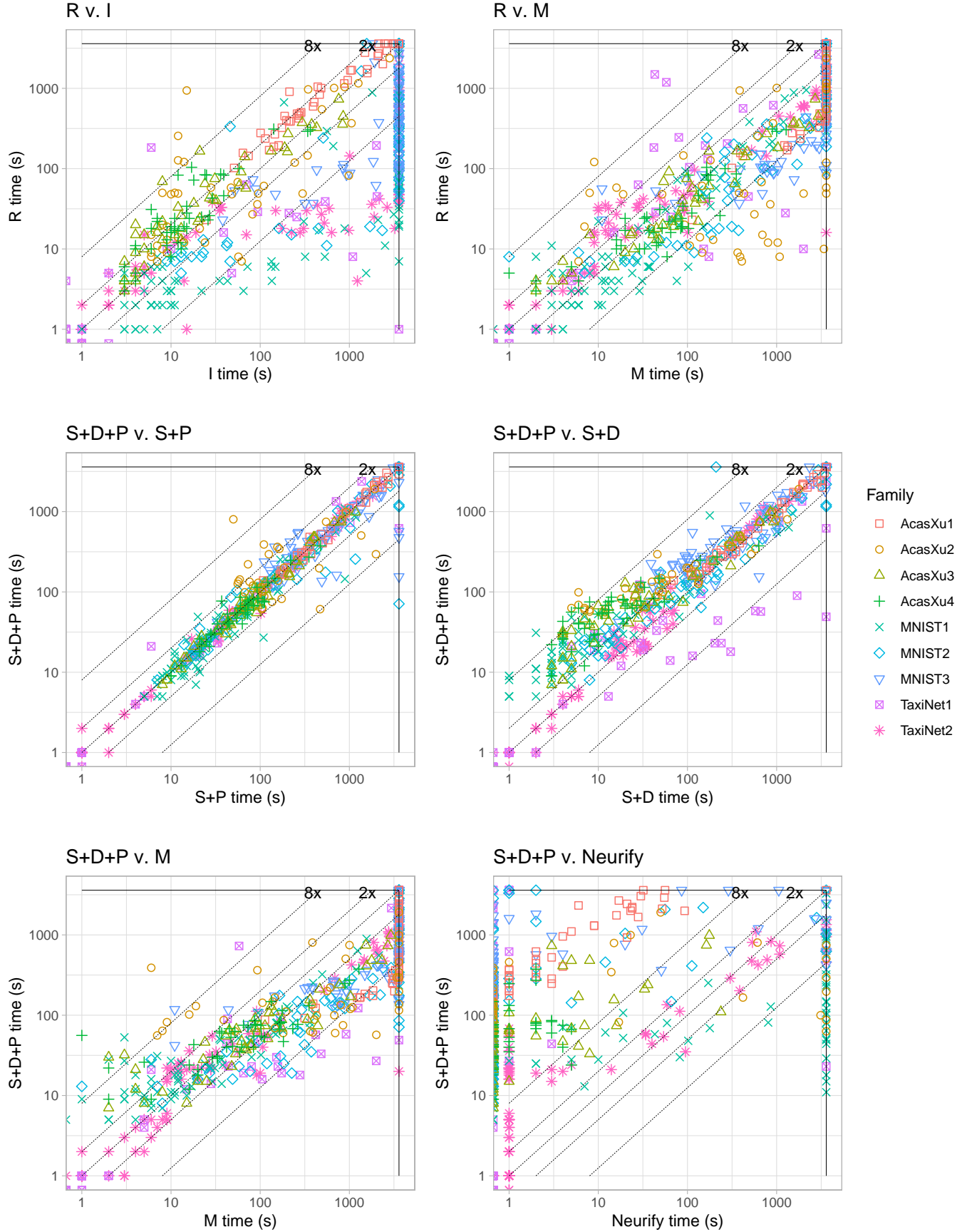
Fig. 6: Pairwise comparison between different configurations on all benchmarks.