Incremental Verification of Neural Networks

SHUBHAM UGARE, University of Illinois Urbana-Champaign, USA
DEBANGSHU BANERJEE, University of Illinois Urbana-Champaign, USA
SASA MISAILOVIC, University of Illinois Urbana-Champaign, USA
GAGANDEEP SINGH, University of Illinois Urbana-Champaign and VMware Research, USA

Complete verification of deep neural networks (DNNs) can exactly determine whether the DNN satisfies a desired trustworthy property (e.g., robustness, fairness) on an infinite set of inputs or not. Despite the tremendous progress to improve the scalability of complete verifiers over the years on individual DNNs, they are inherently inefficient when a deployed DNN is updated to improve its inference speed or accuracy. The inefficiency is because the expensive verifier needs to be run from scratch on the updated DNN. To improve efficiency, we propose a new, general framework for incremental and complete DNN verification based on the design of novel theory, data structure, and algorithms. Our contributions implemented in a tool named IVAN yield an overall geometric mean speedup of 2.4x for verifying challenging MNIST and CIFAR10 classifiers and a geometric mean speedup of 3.8x for the ACAS-XU classifiers over the state-of-the-art baselines.

CCS Concepts: • **Theory of computation → Program analysis**; **Abstraction**; • **Computing methodologies → Neural networks**.

Additional Key Words and Phrases: Verification, Robustness, Deep Neural Networks

ACM Reference Format:
Shubham Ugare, Debangshu Banerjee, Sasa Misailovic, and Gagandeep Singh. 2023. Incremental Verification of Neural Networks. *Proc. ACM Program. Lang.* 7, PLDI, Article 185 (June 2023), 31 pages. https://doi.org/10.1145/3591299

1 INTRODUCTION

Deep neural networks (DNNs) are being increasingly deployed for safety-critical applications in many domains including autonomous driving [Bojarski et al. 2016], healthcare [Amato et al. 2013], and aviation [Julian et al. 2018]. However, the black-box construction, vulnerability against adversarial changes to in-distribution inputs [Madry et al. 2017; Szegedy et al. 2014], and fragility against out-of-distribution data [Chen et al. 2022; Gokhale et al. 2021] is the main hindrance to the trustworthy deployment of deep neural networks in real-world applications. Recent years have witnessed increasing work on developing verifiers for formally checking whether the behavior of DNNs (see [Albarghouthi 2021; Urban and Miné 2021] for a survey) on an infinite set of inputs is trustworthy or not. For example, existing verifiers can formally prove [Bak et al. 2020; Bunel et al. 2020b,a; Ehlers 2017; Gehr et al. 2018; Wang et al. 2018] that the infinite number of images obtained after varying the intensity of pixels in an original image by a small amount will be classified correctly. Verification yields better insights into the trustworthiness of DNNs than standard test-set accuracy measurements, which only check DNN performance on a finite number of inputs. The insights can be used for selecting the most trustworthy DNN for deployment among a set of DNNs trained for the

Authors' addresses: Shubham Ugare, University of Illinois Urbana-Champaign, USA; Debangshu Banerjee, University of Illinois Urbana-Champaign, USA; Sasa Misailovic, University of Illinois Urbana-Champaign, USA; Gagandeep Singh, University of Illinois Urbana-Champaign and VMware Research, USA.

2475-1421/2023/6-ART185
https://doi.org/10.1145/3591299

same task. Existing verifiers can be broadly classified as either complete or incomplete. Incomplete methods are more scalable but may fail to prove or disprove a trustworthiness property [Gehr et al. 2018; Salman et al. 2019; Singh et al. 2019a, 2018, 2019b; Xu et al. 2020; Zhang et al. 2018]. A complete verifier always verifies the property if the property holds or otherwise returns a counterexample. Complete verification methods are more desirable as they are guaranteed to provide an exact answer for the verification task [Anderson et al. 2020; Bak et al. 2020; Bunel et al. 2020b,a; Ehlers 2017; Ferrari et al. 2022; Fromherz et al. 2021; Gehr et al. 2018; Palma et al. 2021; Wang et al. 2018, 2021; Zhang et al. 2022].

Limitation of Existing Works: The deployed DNNs are modified for reasons such as approximation [Blalock et al. 2020; Gholami et al. 2021], fine-tuning [Tajbakhsh et al. 2016], model repair [Sotoudeh and Thakur 2019], or transfer learning [Weiss et al. 2016]. Various approximations such as quantization, and pruning slightly perturb the DNN weights, and the updated DNN is used for the same task [Gholami et al. 2021; Laurel et al. 2021; TFLite 2017]. Similarly, fine-tuning can also be performed to repair the network on buggy inputs while maintaining the accuracy on the original training inputs [Fu and Li 2022]. Each time a new DNN is created, expensive complete verification needs to be performed to check whether it is trustworthy. A fundamental limitation of all existing approaches for complete verification of DNNs is that the verifier needs to be run from scratch end-to-end every time the network is even slightly modified. As a result, developers still rely on test set accuracy as the main metric for measuring the quality of a trained network. This limitation of existing verifiers restricts their applicability as a tool for evaluating the trustworthiness of DNNs.

This Work: Incremental and Complete Verification of DNNs: In this work, we address the fundamental limitation of existing complete verifiers by presenting IVAN, the first general technique for incremental and complete verification of DNNs. An original network and its updated network have similar behaviors on most of the inputs, therefore the proofs of property on these networks are also related. IVAN accelerates the complete verification of a trustworthy property on the updated network by leveraging the proof of the same property on the original network. IVAN can be built on top of any Branch and Bound (BaB) based method. The BaB verifier recursively partitions the verification problem to gain precision. It is currently the dominant technology for constructing complete verifiers [Anderson et al. 2019; Bak et al. 2020; Bunel et al. 2020b,a; Ehlers 2017; Ferrari et al. 2022; Fromherz et al. 2021; Palma et al. 2021; Wang et al. 2018, 2021; Zhang et al. 2022].

Challenges: The main challenge in building an incremental verifier on top of a non-incremental one is to determine which information to pass on and how to effectively reuse this information. Formal methods research has developed numerous techniques for incremental verification of programs, that reuse the proof from previous revisions for verifying the new revision of the program [Johnson et al. 2013; Lakhnech et al. 2001; O'Hearn 2018; Stein et al. 2021]. However, often the program commits are local changes that affect only a small part of the big program. In contrast, most DNN updates result in weight perturbation across one or many layers of the network. This poses a different and more difficult challenge than incremental program verification. Additionally, DNN complete verifiers employ distinct heuristics for branching. A key challenge is to develop a generic method that incrementally verifies a network perturbed across multiple layers and is applicable to multiple complete verification methods, yet can provide significant performance benefits.

Our Solution: IVAN computes a specification tree – a novel tree data structure representing the trace of BaB – from the execution of the complete verifier on the original network. We design new algorithms to refine the specification tree to create a more compact tree. At a high level, the refinement involves reordering the branching decisions such that the decisions that worked well in the original verification are prioritized. Besides, it removes the branching decisions that worked poorly in the original verification by pruning nodes and edges in the specification tree. IVAN also improves the branching strategy in BaB for the updated network based on the observed

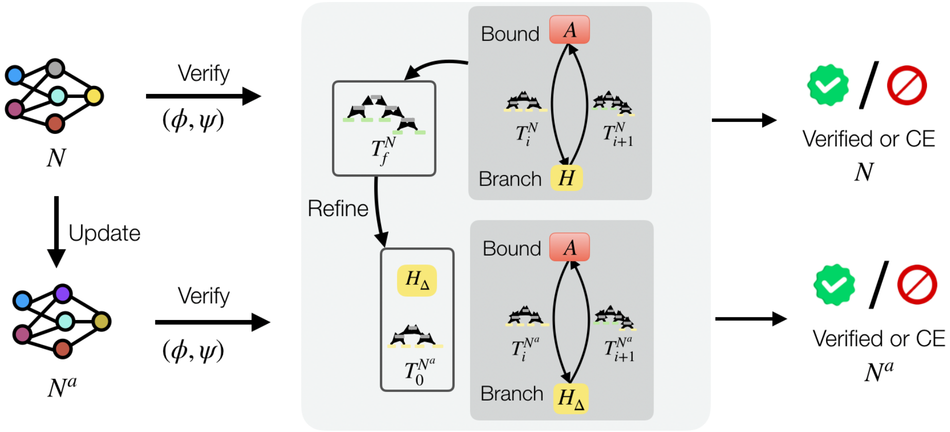


Fig. 1. Workflow of IVAN from left to right. *IVAN* takes the original network N, input specification ϕ and output specification ψ. It is built on top of a BaB-based complete verifier that utilizes an analyzer A for the bounding, and heuristic H for branching. IVAN refines a specification tree T_f^N, result of verifying N, to create a compact tree $T_0^{N^a}$ and updated branching heuristic H_Δ. IVAN performs faster verification of N^a exploiting both $T_0^{N^a}$ and H_Δ.

effectiveness of branching choices when verifying the original DNN. The compact specification tree and the improved branching strategy guide the BaB execution on the updated network to faster verification, compared to non-incremental verification that starts from scratch. IVAN yields up to 43x speedup over the baseline based on state-of-the-art non-incremental verification techniques [Bunel et al. 2020b; Henriksen and Lomuscio 2021; Singh et al. 2018]. It achieves a geometric mean speedup of 2.4x across challenging fully-connected and convolutional networks over the baseline. IVAN is generic and can work with various common BaB branching strategies in the literature (input splitting, ReLU splitting).

Main Contributions: The main contributions of this paper are:

- We present a novel, general framework for incremental and complete DNN verification by designing new algorithms and data structure that allows us to succinctly encode influential branching strategies to perform efficient incremental verification of the updated network.
- We identify a class of network modifications that can be efficiently verified by our framework by providing theoretical bounds on the amount of modifications.
- We implement our approach into a tool named IVAN and show its effectiveness over multiple state-of-the-art complete verification techniques, using distinct branching strategies (ReLU splitting and input splitting), in incrementally verifying both local and global properties of fully-connected and convolutional networks with ReLU activations trained on the popular ACAS-XU, MNIST, and CIFAR10 datasets. Our results show that for MNIST and CIFAR10 classifiers, using the ReLU splitting technique [Henriksen and Lomuscio 2021] IVAN yields a geometric mean speedup of 2.4x over the state-of-the-art baseline [Bunel et al. 2020b; Ehlers 2017]. For ACAS-XU, using the input splitting technique IVAN achieves a geometric mean speedup of 3.8x over RefineZono [Singh et al. 2019c].

IVAN implementation is open-source, publicly available at https://github.com/uiuc-focal-lab/IVAN. An extended version of this paper containing all the proofs and additional experiments is available at https://arxiv.org/abs/2304.01874.

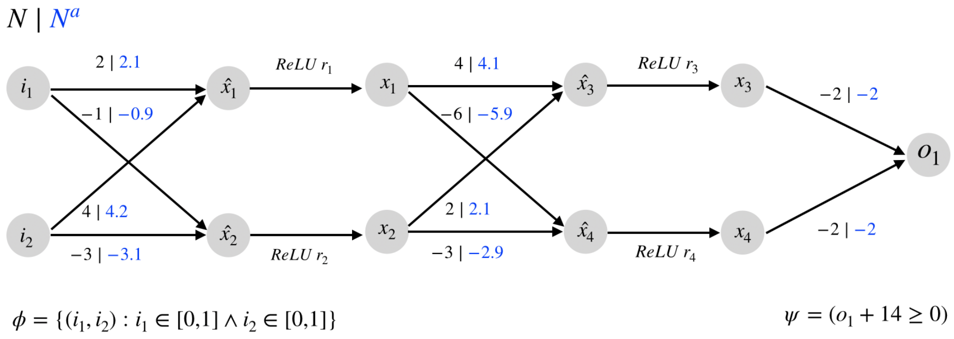


Fig. 2. Example original network N and its perturbation N^a (blue weights). Each layer consists of a linear function followed by the ReLU activation function. ϕ is the input specification and ψ is the output specification.

2 OVERVIEW

Figure 1 illustrates the high-level idea behind the workings of IVAN. It takes as input the original neural network N, the updated network N^a, a local or global input region ϕ, and the output property ψ. The goal of IVAN is to check whether for all inputs in ϕ, the outputs of networks N and N^a satisfy ψ. N and N^a have similar behaviors on the inputs in ϕ, therefore the proofs of the property on these networks are also related. IVAN accelerates the complete verification of the property (ϕ, ψ) on N^a by leveraging the proof of the same property on N.

Neural Network Verifier: Popular verification properties considered in the literature have $\psi := C^T Y \geq 0$, where C is a column vector and $Y = N(X)$, for $X \in \phi$. Most state-of-the-art complete verifiers use BaB to solve this problem. These techniques use an analyzer that computes the linear approximation of the network output Y through a convex approximation of the problem domain. This linear approximation of Y is used to perform the bounding step to show for the lower bound $LB(C^T Y)$ that $LB(C^T Y) \geq 0$. If the bounding step cannot prove the property, the verification problem is partitioned into subproblems using a branching heuristic H. The partitioning splits the problem space allowing a more precise convex approximation of the split subproblems. This leads to gains in the precision of LB computation. Various choices for the analyzer and the branching strategies exist which represent different trade-offs between precision and speed.

IVAN leverages a specification tree representation and novel algorithms to store and transfer the proof of the property from N to N^a for accelerating the verification on N^a. We show the workings of $IVAN$ through the following illustrative example.

2.1 Illustrative Example

We consider the two networks N and N^a with the same architecture as shown in Figure 2. Most practical network updates result in network weight perturbations e.g., quantization, model repair, and fine-tuning. Network N^a is obtained by updating (perturbing the weights) of network N. These networks apply ReLU activation at the end of each affine layer except for the final layer. The weights for the affine layers are shown on the edges. We consider the verification property (ϕ, ψ) such that $\phi = \{(i_1, i_2) : i_1 \in [0, 1] \wedge i_2 \in [0, 1]\}$ and $\psi = (o_1 + 14 \geq 0)$. Let $\mathcal{R} = \{r_1, r_2, r_3, r_4\}$ denote the set of ReLUs in the considered architecture. $\mathcal{R}$ is a function of the architecture of the DNNs and is common for both N and N^a.

Branch and Bound: We consider a complete verifier that uses a sound analyzer A based on the exact encoding of the affine layers and the common triangle linear relaxation [Bunel et al. 2020b,a; Ehlers 2017] for over-approximating the non-linear ReLU function. If due to over-approximation of the ReLU function, the analyzer cannot prove or disprove the property, the verifier partitions the problem by splitting the problem domain. The analyzer is more precise if it separately analyzes the split subproblems and merges the results. There are two main strategies for branching considered in the literature, input splitting [Anderson et al. 2020; Wang et al. 2018], and ReLU splitting [Bunel et al. 2020b,a; Ehlers 2017; Ferrari et al. 2022; Palma et al. 2021]. We show IVAN's effectiveness on both branching strategies in our evaluation (Section 6.1, Section 6.4). However, for this discussion, we focus on ReLU splitting which is scalable for the verification of high-dimensional inputs.

ReLU splitting: An unsolved problem is partitioned into two cases, where the cases assume the input $\hat{x}_i$ to ReLU unit r_i satisfies the predicates $\hat{x}_i \geq 0$ and $\hat{x}_i < 0$ respectively. Splitting a ReLU r_i eliminates the analyzer imprecision in the approximation of r_i. When we split all the ReLUs in $\mathcal{R}$, the analyzer is exact. Nevertheless, splitting all $\mathcal{R}$ is expensive as it requires $2^{|\mathcal{R}|}$ analyzer bounding calls. The state-of-the-art techniques use the heuristic function H to find the best ReLU to split at each step, leading to considerably scalable complete verification.

The branching function H scores the ReLUs $\mathcal{R}$ for branching at each unsolved problem to partition the problem. If $\mathcal{R}' \subseteq \mathcal{R}$ denotes the subset of ReLUs that are not split in the current subproblem, then the verifier computes $r = \arg\max_{\mathcal{R}'} H$ to choose the r for the current split. H is a function of the exact subproblem that it branches and hence depends on ϕ, ψ, the network, and the branching assumptions made for the subproblem. However, for the purpose of this running example, we consider a simple constant branching heuristic H that ranks $H(r_1) > H(r_3) > H(r_4) > H(r_2)$ independent of the subproblem and the network. This assumption is only for the illustration of our idea, we show in the evaluation (Section 6) that IVAN can work with state-of-the-art branching heuristics [Bunel et al. 2020b; Henriksen and Lomuscio 2021].

2.2 IVAN Algorithm

Specification Tree: IVAN uses a rooted binary tree data structure to store the trace of splitting decisions during BaB execution. A specification split is a finer specification parameterized by the subset of ReLUs in $\mathcal{R}$. The root node is associated with the specification (ϕ, ψ). All other nodes represent the specification splits obtained by splitting the problem domain recursively. Each internal node in the tree has two children, the result of the branching of the associated specification.

The split decision can be represented as a predicate. For a ReLU r_i with input $\hat{x}_i$, let $r_i^+ := (\hat{x}_i \geq 0)$ and $r_i^- := (\hat{x}_i < 0)$ denote the split decisions. A split of ReLU r_i at node n creates two children nodes n_l and n_r, each encoding the new specification splits. Each edge in the specification tree represents the split decision made at the branching step. An edge connects an internal node with its child node, and we label it with the additional predicate that is assumed by the child subproblem. A split of ReLU r at node n adds nodes n_l and n_r that are connected with edges labeled with predicates r_i^+ and r_i^- respectively. If $\varphi_n = (\phi', \psi)$ is the specification split at n, then $\varphi_{n_l} = (\phi' \wedge r^+, \psi)$ and $\varphi_{n_r} = (\varphi' \wedge r^-, \psi)$. The names of the nodes have no relation to the networks or the property, they are used for referencing a particular specification. However, the edges of the tree are tied to the network architecture through the labels. Although the specification tree is created as a trace of verification of a particular network N, it is only a function of the ReLU units in the architecture of N. This allows us to use the branching decisions in the specification tree for guiding the verification of any updated network N^a that has the same architecture as N. We use $LB_N(n)$ to denote the lower bound $LB(C^T Y)$ obtained by the analyzer A on for the subproblem encoded by n, on the network N.

Figure 3 demonstrates the steps of BaB execution on N. Each node represents the specification refined by BaB. We use function $LB_N(n)$ to denote the $LB(C^T Y) = LB(o_1 + 14)$ value obtained by the analyzer A at node n. The specification is verified for the subproblem of n if the $LB_N(n) \geq 0$. If $LB_N(n) < 0$, the analyzer returns a counterexample (CE). The CE is a point in the convex

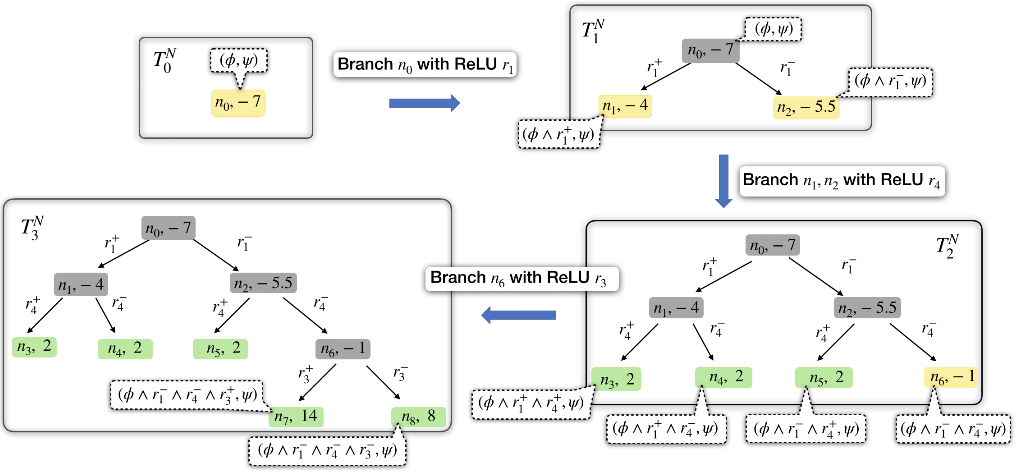
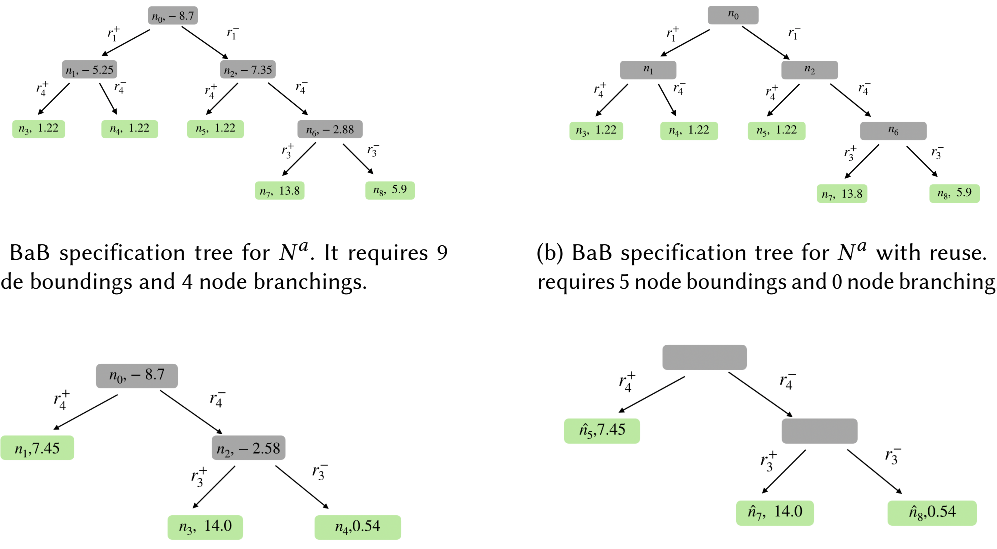

T_0^N
(ϕ, ψ)
$n_0, -7$
Branch n_0 with ReLU r_1
T_1^N
(ϕ, ψ)
$n_0, -7$
r_1^+
r_1^-
$n_1, -4$
$n_2, -5.5$
$(\phi \wedge r_1^+, \psi)$
$(\phi \wedge r_1^-, \psi)$
Branch n_1, n_2 with ReLU r_4
T_2^N
$n_0, -7$
$n_1, -4$
$n_2, -5.5$
r_4^+
r_4^-
$n_3, 2$
$n_4, 2$
$n_5, 2$
$n_6, -1$
$(\phi \wedge r_1^+ \wedge r_4^+, \psi)$
$(\phi \wedge r_1^+ \wedge r_4^-, \psi)$
$(\phi \wedge r_1^- \wedge r_4^+, \psi)$
$(\phi \wedge r_1^- \wedge r_4^-, \psi)$
Branch n_6 with ReLU r_3
T_3^N
$n_0, -7$
$n_1, -4$
$n_2, -5.5$
$n_3, 2$
$n_4, 2$
$n_5, 2$
$n_6, -1$
r_3^+
r_3^-
$n_7, 14$
$n_8, 8$
$(\phi \wedge r_1^- \wedge r_4^- \wedge r_3^+, \psi)$
$(\phi \wedge r_1^- \wedge r_4^- \wedge r_3^-, \psi)$

Fig. 3. Steps in Branch and Bound algorithm for complete verification of N. The nodes are labeled with a name and the $LB_N(n)$. The nodes in the specification tree are annotated with their specifications. The edges are labeled with the branching predicates. Each step in BaB partitions unsolved specifications in T_i^N into specification splits in T_{i+1}^N. The proof is complete when all specification splits corresponding to the leaf nodes are solved.

approximation of the problem domain and it may be possible that it is spurious, and does not belong to the concrete problem domain. If the CE is not spurious, the specification is disproved and the proof halts. But, if the CE is spurious then the problem is unsolved, and it is further partitioned.

In the first step, for the specification (ϕ, ψ) encoded by the root node n_0, the analyzer computes $LB_N(n_0) = -7$, which is insufficient to prove the specification. Further, the CE provided by the analyzer is spurious, and thus the analyzer cannot solve the problem. The root node n_0 specification (ϕ, ψ) is split by ReLU split of r_1 chosen by the heuristic function H. Accordingly, in the specification tree, the node n_0 is split into two nodes n_1 and n_2, with the specification splits $(\phi \wedge r_1^+, \psi)$ and $(\phi \wedge r_1^-, \psi)$ respectively. This procedure of recursively splitting the problem and correspondingly updating the specification tree continues until either all the specifications of the leaf nodes are verified, or a CE is found. In the final specification tree (T_3^N in this case), the leaf nodes are associated with the specifications that the analyzer could solve, and the internal nodes represent the specifications that the analyzer could not solve for network N. For BaB starting from scratch, each node in the specification tree maps to a specification that invoked an analyzer call in BaB execution. Figure 3 presents that the verifier successfully proves the property with a specification tree containing 9 nodes. Thus, the verification invokes the analyzer 9 times and performs 4 nodes branchings for computing LB.

Figure 4a presents the specification tree for N^a at end of the verifying the property (ϕ, ψ). Although the $LB(C^T Y)$ computed by the analyzer for each node specifications is different for N^a compared to N, the final specification tree is identical for both networks. Our techniques in IVAN are motivated by our observation that the final specification tree for network N and its updated version N^a have structural similarities. Moreover, we find that for a DNN update that perturbs the network weight within a fixed bound, these trees are identical. We claim that there are two reasons for this: (i) the specifications that are solved by the analyzer for N are solved by the analyzer for N^a (specifications of the leaf nodes of the specification tree) and (ii) the specifications that are unsolved by the analyzer for N are unsolved for N^a (specifications of the internal nodes of the specification tree). In Section 4.4, we provide theoretical bounds on the network perturbations such that these



(a) BaB specification tree for N^a. It requires 9 node boundings and 4 node branchings.

(b) BaB specification tree for N^a with reuse. It requires 5 node boundings and 0 node branchings.

(c) BaB specification tree for N^a with reorder. It requires 5 node boundings and 2 node branchings.

(d) BaB specification tree for N^a with IVAN. It requires 3 node boundings and 0 node branchings.

Fig. 4. BaB specification tree for various techniques proposed for incremental verification.

claims hold true (Theorem 4). Nevertheless, for networks obtained by perturbation beyond the theoretical bounds, the specification trees are still similar if not identical. In our evaluation, we observe this similarity for large networks with practical updates e.g., quantization (Section 6).

Reuse: We first introduce our concept of specification tree reuse which uses T_f^N, the final tree after verifying N, as the starting tree $T_0^{N^a}$ for the verification of N^a. In contrast, the standard BaB verification starts with a single node tree that represents the unpartitioned initial specification (ϕ, ψ). In the reuse technique, IVAN starts BaB verification of N^a from the leaves of $T_0^{N^a} = T_f^N$. For our running example, analyzer A successfully verifies N^a specifications for all the leaf nodes of the specification tree $T_0^{N^a}$ (Figure 4b). We show that for any specification tree (created on the same network architecture), verifying the subproblem property on all the leaves of the specification tree is equivalent to verifying the main property (ϕ, ψ) (Lemma 1). Verifying the property on N^a from scratch requires 9 analyzer calls and 4 node branchings. However, with the reuse technique, we could prove the property with 5 analyzer calls corresponding to the leaves of $T_0^{N^a}$ and without any node branching. Theorem 4 guarantees that the specification of the leaf nodes should be verified on N^a by the analyzer if the network perturbations are lower than a fixed bound. Although for larger perturbations, we may have to split leaves of $T_0^{N^a}$ further for complete verification, we empirically observe that the reuse technique is still effective to gain speedup on most practical network perturbations.

Reorder: A split is more effective if it leads to fewer further subproblems that the verifier has to solve to prove the property. Finding the optimal split is expensive. Hence, the heuristic H is used to estimate the effectiveness of a split, and to choose the split with the highest estimated effectiveness. Often the estimates are imprecise and lead to ineffective splits. We use $LB_N(n)$ to give an approximation to quantifying the effectiveness of a split. We discuss this exact formulation of the observed effectiveness scores H_{obs} in Section 4.3. Our second concept in IVAN is based on our

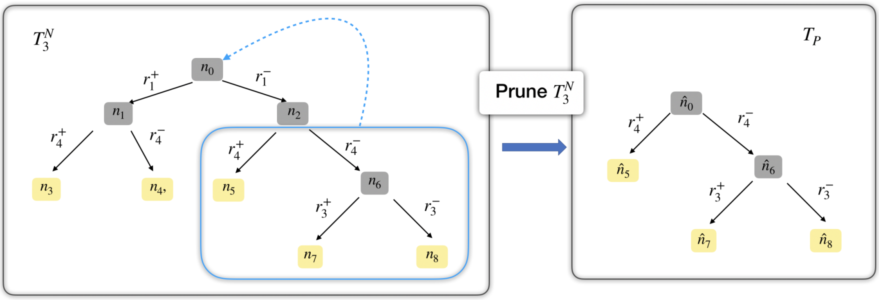


Fig. 5. IVAN removes the ineffective split r_1 at n_0 and construct a new specification tree T_P.

insight that if a particular branching decision is effective for verifying N then it should be effective for verifying N^a. Likewise, if a particular branching decision is ineffective in the verification of N, it should be ineffective in verifying N^a. Based on this insight, we use the observed effectiveness score of splits in verifying N to modify the original branching heuristic H to an improved heuristic H_Δ. H_Δ takes the weighted sum of original branching heuristic H and observed effectiveness scores on N denoted by H_{obs}. We formulate the effectiveness of a split and H_Δ in Section 4.3. For simplicity, in the running example, we rerank the ReLUs based on the observed effectiveness of the splits as $H_\Delta(r_4) > H_\Delta(r_3) > H_\Delta(r_2) > H_\Delta(r_1)$. Figure 4c presents the specification tree for verifying N^a with the updated branching heuristic H_Δ that requires 5 analyzer calls and 2 node branchings. Reorder technique starts from scratch with a different branching order H_Δ and it is incomparable in theory to the reuse technique. In Section 6.2, we observe that reorder works better in most experiments.

Bringing All Together: Our main algorithm combines our novel concepts of specification tree reuse and reorder yielding larger speedups than possible with only reuse or reorder. Specification tree reuse and reorder are not completely orthogonal and thus combining them is not straightforward. Since in reuse we start verifying N^a with the final specification tree T_f^N, the splits are already performed with the original order (r_1, r_4, r_3, r_2 in our example). Our augmented heuristic function H_Δ will have a limited effect if we reuse $T_0^{N^a} = T_f^N$, since the existing tree branches may already be sufficient to prove the property.

Constructing a Pruned Specification Tree: It is difficult to predict the structure of the tree with augmented order. For instance, in our example, N is verified with r_1, r_4, r_3, r_2 order and we have T_f^N branched in that order. However, we cannot predict the final structure of the specification tree if branched with our augmented order r_4, r_3, r_2, r_1 without actually performing those splits from scratch (as it was done in Figure 4c).

We solve this problem with our novel pruning operation that removes ineffective splits from T_f^N and constructs a new compact tree T_P. Figure 5 shows the construction of pruned tree T_P for our running example. We remove the split r_1 at n_0 as it is less effective. Removing r_1 from T_3^N also eliminates the nodes n_1 and n_2. The subtrees rooted at n_1 and n_2 are the result of split r_1. If we undo the split r_1 at node n_0, then n_0 should follow the branching decisions taken by one of its children. For this, we can choose either the subtree of n_0 or n_1, and attach it to n_0. We describe the exact method of choosing which subtree to keep in Section 4.3. For this example, our approach chooses to keep the subtree of node n_2 and eliminates the subtree at node n_1. The pruning procedure leads to the discarding of entire subtrees creating a tree with fewer leaf nodes (leaf nodes n_3, n_4 are deleted in the example along with internal nodes n_1, n_2). Consequently, we obtain a more compact tree with only influential splits in the specification tree.

We start the verification of N^a from the leaf nodes of the pruned tree i.e. $T_0^{N^a} = T_P$. For our running example specification splits of all leaf nodes of T_P are verified by the analyzer and no further splitting is needed. Figure 4d presents the final specification tree in case we initialize the proof with the compact tree obtained from the IVAN algorithm. We show the time complexity of incremental verification in Section 4.2. For the running example, the incremental proof requires only 3 analyzer calls and no branching calls, and it is a significant reduction to the 9 analyzer calls and 5 node branchings performed by the baseline starting from scratch.

3 PRELIMINARIES

In this section, we provide the necessary background on complete neural network verification.

3.1 Neural Network Verification

Neural Networks Neural networks are functions $N : \mathbb{R}^{n_0} \rightarrow \mathbb{R}^{n_l}$. In this work, we focus on layered neural networks obtained by a sequential composition of l layers $N_1 : \mathbb{R}^{n_0} \rightarrow \mathbb{R}^{N_1}, \dots, N_l : \mathbb{R}^{n_{l-1}} \rightarrow \mathbb{R}^{n_l}$. Each layer N_i applies an *affine function* (convolution or linear function) followed by a non-linear activation function to its input. The choices for non-linear activation functions are ReLU, sigmoid, or tanh. $ReLU(x) = max(0, x)$ is most commonly used activation function. In Section 4, we focus on the most common BaB verifiers that partition the problems using ReLU splitting in ReLU networks. The i-th layer of each network $N_i : \mathbb{R}^{n_i} \longrightarrow \mathbb{R}^{n_{i+1}}$ is defined as $N_i(x) = ReLU(A_i X + B_i)$ where $i \in [l]$.

At a high level, neural network verification involves proving that all network outputs corresponding to a chosen set of inputs satisfying the input specification ϕ satisfy a given logical property ψ. We first define the input and output specifications that we consider in this work:

Definition 1 (Input specification). *For a neural network $N : \mathbb{R}^{n_0} \rightarrow \mathbb{R}^{n_l}$, ϕ_t is a connected region and $\phi_t \subseteq \mathbb{R}^{n_0}$. **Input specification** $\phi : \mathbb{R}^{n_0} \rightarrow \{true, false\}$ is a predicate over the input region ϕ_t.*

Definition 2 (output specification). *For a neural network with n_l neurons in the output layer. **output specification** $\psi : \mathbb{R}^{n_l} \rightarrow \{true, false\}$ is a predicate over the output region.*

The output property ψ could be any logical statement taking a truth value true or false. In our paper, we focus on properties that can be expressed as Boolean expressions over linear forms. Most DNN verification works consider such properties.

$$\psi(Y) = (C^T Y \geq 0) \tag{1}$$

We next define the verification problem solved by the verifiers:

Definition 3 (Verification Problem). *The **neural network verification** problem for a neural network N, an input specification ϕ and a logical property ψ is to prove whether $\forall X \in \phi_t .\ \psi(N(X)) =$ true or provide a counterexample otherwise.*

A complete verifier always verifies the property if it holds or returns a counterexample otherwise. Formally, it can be defined as:

Definition 4 (Complete Verifier). *A **complete verifier** V for an input specification ϕ, a neural network N, an output property ψ satisfies the following property:*

$$V(\phi, \psi, N) = \text{Verified} \iff \forall X \in \phi_t . \psi(N(X)) = \mathit{true}$$

3.2 Branch and Bound for Verification

In this Section, we discuss the branch and bound techniques for complete verification of DNNs. The BaB approach in these techniques use a divide-and-conquer algorithm to compute the $LB(C^T Y)$ for proving $(C^T Y \geq 0)$ (Eq. 1). We next discuss the bounding and branching steps in BaB techniques.

Bounding: The bounding step uses an analyzer to find a lower bound $LB(C^T Y)$. In complete verifiers, the analyzers are exact for linear functions (e.g., DeepZ [Singh et al. 2018], DeepPoly [Singh et al. 2019b]). However, they over-approximate the non-linear activation function through a convex over-approximation. We define these sound analyzers as:

Definition 5 (Sound Analyzer). *A* ***sound analyzer*** *A on an input specification ϕ, a DNN N, an output property ψ returns* Verified, Unknown, *or* Counterexample. *It satisfies the following properties:*

$$A(\phi, \psi, N) = \text{Verified} \implies \forall X \in \phi_t.\psi(N(X)) = \mathit{true}$$
$$A(\phi, \psi, N) = \text{Counterexample} \implies \exists X \in \phi_t.\psi(N(X)) = \mathit{false}$$

Branching: If the analyzer cannot prove a property, the BaB verifier partitions the problem into easier subproblems to improve analyzer precision. Algorithm 1 presents the pseudocode for the BaB verification. The algorithm maintains a $Unsolved$ list of problems that are currently not proved or disproved. It initializes the list with the main verification problem. Line 5 performs the bounding step in the BaB algorithm using the analyzer A. For simplicity, we abuse the notation and use $A(prob, N)$ for denoting the analyzer output instead of $A(\phi, \psi, N)$. Here, the $prob$ encapsulates the input and output specifications ϕ, ψ. Line 13 partitions the unsolved problem into subproblems. The algorithm halts when either the A finds a counterexample on one of the subproblems or the list of unsolved problems is empty. There are two common branching strategies for BaB verification, input splitting and ReLU splitting, which we describe next.

Algorithm 1 Branch and Bound

```
 1: function BaB($N, problem$)
 2:     $Unsolved \leftarrow [(problem)]$
 3:     while $Unsolved$ is not empty do
 4:         for $prob \in Unsolved$ do
 5:             $status[prob] = A(prob, N)$                          ▷ Bounding step
 6:         for $prob \in Unsolved$ do
 7:             if $status[prob] = Verified$ then
 8:                 $Unsolved.remove(prob)$                          ▷ Remove verified subproblems
 9:             if $status[prob] = Counterexample$ then
10:                 return $Counterexample$ for $prob$               ▷ Return if a counterexample is found
11:             if $status[prob] = Unknown$ then
12:                 $Unsolved.remove(prob)$
13:                 $[\text{subprob}_1, \text{subprob}_2] \leftarrow \text{split(prob)}$   ▷ Branching step
14:                 $Unsolved.insert(\text{subprob}_1, \text{subprob}_2)$
15:     return $Verified$
```

Input Splitting: In input splitting, the input region ϕ_t for verification is partitioned. The typical choice is to cut a selected input dimension in half while the rest of the dimensions are unchanged. The dimension to cut is decided by the branching strategy used. This technique is known to be δ-complete for any activation function [Anderson et al. 2019], but does not scale for high-dimensional input space. In many computer vision tasks, the input is an image with 1000s of pixels. Thus, a high-dimensional perturbation region on such input cannot be branched efficiently for fast verification.

ReLU Splitting: State-of-the-art techniques that focus on verifying DNNs with high-dimensional input and ReLU activation, use ReLU splitting. We denote a ReLU unit for i-th layer and j-th index as a function $x_{i,j} = \max(\hat{x}_{i,j}, 0)$, where $\hat{x}_{i,j}$ and $x_{i,j}$ are the pre-activation and post-activation values respectively. The analyzer computes lower bounds lb and upper bounds ub for each intermediate variable in the DNN. If $lb(\hat{x}_{i,j}) \geq 0$, then the ReLU unit simply acts as the identify function $x_{i,j} = \hat{x}_{i,j}$. If $ub(\hat{x}_{i,j}) \leq 0$, then the ReLU unit operates as a constant function $x_{i,j} = 0$. In both of these cases, the ReLU unit is a linear function. However, if $lb(\hat{x}_{i,j}) < 0 < ub(\hat{x}_{i,j})$, we cannot linearize the ReLU function exactly. We call such ReLU units ambiguous ReLUs. In ReLU splitting, the unsolved problem is partitioned into two subproblems such that one subproblem assumes $\hat{x}_{i,j} < 0$ and the other assumes $\hat{x}_{i,j} \geq 0$. This partition allows us to linearize the ReLU unit in both subproblems leading to a boost in the overall precision of the analyzer. The heuristic used for selecting which ReLU to split significantly impacts the verifier speed.

BaB for Other Activation Functions: BaB-based verification can work with the most commonly used activation functions (tanh, sigmoid, leaky ReLU).

(1) For piecewise linear activation functions such as leaky ReLU, activation splitting approaches (e.g, ReLU splitting) can be used for complete verification.
(2) For other activation functions (tanh, sigmoid), BaB with activation splitting cannot yield complete verification but can be used to improve the precision of sound and incomplete verification [Dutta et al. 2017; Müller et al. 2021].
(3) Although input splitting is less efficient in the aforementioned cases for high dimensional DNN inputs, it can be applied with any activation function (tanh, sigmoid, ReLU, leaky ReLU).

4 INCREMENTAL VERIFICATION

In this section, we describe our main technical contributions and the IVAN algorithm. We first formally define the specification tree structure used for incremental verification (Section 4.1). Next, we formulate the problem of incremental verification (Section 4.2). In Section 4.3, we illustrate the techniques used in our algorithm. We characterize the effectiveness of our technique by computing a class of networks for which our incremental verification is efficiently applicable in Section 4.4.

4.1 Specification Tree for BaB

IVAN uses the specification tree to store the trace of splitting decisions that the BaB verifier makes on its execution. A specification tree can be used for any BaB branching method (e.g, input splitting), but without loss of generality, our discussion focuses on ReLU splitting. Let $\mathcal{N}$ denote the class of networks with the same architecture, and let $\mathcal{R}$ denote the set of ReLUs in this architecture. The specification tree captures the ReLU splitting decisions and the split specifications in the execution of BaB for a property (ϕ, ψ), where we define $(\phi, \psi) := \phi \longrightarrow \psi$.

For a ReLU r_i with input $\hat{x}_i$, let $r_i^+ := (\hat{x}_i \geq 0)$ and $r_i^- := (\hat{x}_i < 0)$. We define a split decision as:

Definition 6 (Split Decision). *For a ReLU $r \in \mathcal{R}$, a split decision is $r^? \in \{r^+, r^-\}$ where $r^?$ is assigned the predicate r^+ or r^-.*

A specification split of (ϕ, ψ) is a specification stronger than (ϕ, ψ) parameterized by the subset of ReLUs in $\mathcal{R}$ and the corresponding split decisions. Formally,

Definition 7 (Specification Split). *For a set of ReLUs $\mathcal{R}' = \{r_1, r_2 \ldots r_k\} \subseteq \mathcal{R}$, and ReLU split decision $r_i^? \in \{r_i^-, r_i^+\}$ for each r_i, the corresponding specification split of (ϕ, ψ) is $(\phi \wedge r_1^? \wedge r_2^? \wedge \ldots r_k^?, \psi)$.*

Since $\emptyset \subseteq \mathcal{R}$, (ϕ, ψ) is a split specification of itself. Let $\mathcal{S}$ denote the set of specification splits that can be obtained from (ϕ, ψ). Each node n in the tree encodes a specification split in $\mathcal{S}$. Each edge in the specification tree is labeled with a ReLU split decision $r^?$. Let $Nodes(T)$ denote the nodes of the tree T and $Leaves(T)$ denote the leaves of the tree T.

Mapping Nodes to Specification Splits: The specification associated with the root node is (ϕ, ψ). The function $Children(n)$ maps a node n to either the pair of its children or $\emptyset$ if n has no children. If n_l and n_r are the children of node n and $\varphi_n = (\phi', \psi)$ is the specification split at n, then $\varphi_{n_l} = (\phi' \wedge r^+, \psi)$

and $\varphi_{n_r} = (\varphi' \wedge r^-, \psi)$. For the specifications $\varphi_n, \varphi_{n_l}, \varphi_{n_r}$ the following statement holds:

$$(\varphi_{n_l} \wedge \varphi_{n_r}) \Longleftrightarrow \varphi_n \tag{2}$$

This relationship implies that verifying the parent node specification is equivalent to verifying the two children node's specifications. Formally, we can now define the specification tree as:

Definition 8 (specification tree). *Given a set of ReLU $\mathcal{R}$, a rooted full binary tree T is a* ***specification tree****, if for a node $n \in Nodes(T)$, and nodes $n_l, n_r \in Children(n)$, edge (n, n_l) is labeled with predicate r^+ and edge (n, n_r) is labeled with predicate r^-, for $r \in \mathcal{R}$.*

Algorithm 2 *Split* operation

```
function Split$(T, n, r)$
    Input: Specification tree $T$, a leaf node $n \in Leaves(T)$, a ReLU $r \in \mathcal{R}$ for splitting the node
    Output: returns newly added nodes
    $n_l \leftarrow$ Add_Child$(n, r^+)$
    $n_r \leftarrow$ Add_Child$(n, r^-)$
    return $n_l, n_r$
```

BaB uses a branching function H for choosing the ReLU to split. We define this branching function in terms of the node n of the specification tree as:

Definition 9 (Branching Heuristic). *Given a set of ReLU $\mathcal{R}$, a network N, and a node n in the specification tree, if $\mathcal{P} \subseteq \mathcal{R}$ denote the set of ReLUs split in the path from the root node of the specification tree to n then the branching heuristic $H(N, n, r)$ computes a score $h \in \mathbb{R}$ estimating the effectiveness of ReLU $r \in \mathcal{R}/\mathcal{P}$ for splitting the specification (φ_n) of the node n.*

We next state the split operation on a specification tree. Algorithm 2 presents the steps in the split operation.

- *Split Operation:* Every ReLU split adds two nodes to the specification tree at a given leaf node n. The BaB algorithm chooses the ReLU $\arg\max_{r \in \mathcal{R}/\mathcal{P}} H(N, n, r)$ to split at node n using the heuristic function.

4.2 Incremental Verification: Problem Formulation

Give a set of networks $\mathcal{N}$ with the same architecture with a set of ReLUs $\mathcal{R}$, $\mathcal{T}_{\mathcal{N}}$ be the set of all specification trees defined over $\mathcal{R}$. There exists a partial order $(<)$ on $\mathcal{T}_{\mathcal{N}}$ through standard subgraph relation. BaB execution on a network $N \in \mathcal{N}$ traces a sequence of trees $T_0, T_1 \dots T_f \in \mathcal{T}_{\mathcal{N}}$ such that $T_i < T_{i+1}$. It halts with the final tree T_f when it either verifies the property or finds a counterexample. The construction of T_{i+1} from T_i depends on the branching function H (Definition 9).

Incremental Verification: The incremental verification problem is to efficiently reuse the information from the execution of verification of network N for the faster verification of its updated version N^a. Standard BaB for verification of N^a starts with a single node tree while the incremental verifier starts with a tree $T_0^{N^a} \in \mathcal{T}_{\mathcal{N}}$ that is not restricted to be a tree with a single node. We modify the final specification tree T_f^N from the verification of N to construct $T_0^{N^a}$. The branching heuristic H_Δ for incremental verification is derived from the branching heuristic H based on the efficacy of various branching decisions made during the proof for N. Formally, the complete incremental verifier we propose is defined as:

Definition 10 (Complete and Incremental Verifier). *A* ***Complete and Incremental Verifier*** *V_Δ takes a neural network N^a, an input specification ϕ, an output property ψ, analyzer A, the branching heuristic H_Δ and the initial tree $T_0^{N^a}$. $V_\Delta(N^a, \phi, \psi, T_0^{N^a}, H_\Delta)$ returns* Verified *if N^a satisfies the property (ϕ, ψ), otherwise, it returns a* Counterexample.

Algorithm 3 presents the incremental verifier algorithm for verifying the perturbed network. It takes H_Δ and $T_0^{N^a}$ as input. It maintains a list of active nodes which are the nodes corresponding to the specifications that are yet to be checked by the analyzer. It initializes the list of active nodes with leaves of tree $T_0^{N^a}$ (line 2). The main loop runs until the active list is empty (line 3) or it

Algorithm 3 Verifying Perturbed Network

Input: N^a, property (ϕ,ψ), Initial specification tree $T_0^{N^a}$, branching heuristic H_Δ
Output: *Verified*if the specification (ϕ,ψ) is verified, otherwise a *Counterexample*

```
1:  $T^{N^a} \leftarrow$ Initialize $T^{N^a}$ as $T_0^{N^a}$
2:  $Active = Leaves(T_0^{N^a})$                          ▷ Initialize active list as $Leaves(T_0^{N^a})$
3:  while $Active$ is not empty do
4:      for $n \in Active$ do
5:          $status[n] \leftarrow A(n)$                    ▷ Bounding step
6:      for $n \in Active$ do
7:          if $status[n] = Verified$ then
8:              $Active.remove(n)$                         ▷ Remove verified nodes
9:          if $status[n] = Counterexample$ then
10:             $Active.empty()$
11:             return $Counterexample$for $n$             ▷ Return if a counterexample is found
12:         if $status[n] = Unknown$ then
13:             $Active.remove(n)$
14:             $r_{chosen} \leftarrow \arg\max_{r \in \mathcal{R}} H_\Delta(N, n, r)$   ▷ Use $H_\Delta$ to choose the split ReLU
15:             $n_l, n_r \leftarrow Split(T^{N^a}, n, r_{chosen})$   ▷ Branching step
16:             $Active.insert(n_l, n_r)$
17: return $Verified$
```

discovers a counterexample (line 9). At each iteration, it runs the analyzer on each node in the active list (line 5). The nodes that are *Verified* are removed from the list (line 8), whereas the nodes that result in *Unknown* are split. The new children are added to the active list (line 12).

Optimal Incremental Verification: We define the partial function $Time_\Delta : \mathcal{T}_N \times \mathcal{T}_N \rightharpoonup \mathbb{R}$, $Time_\Delta(T_0^{N^a}, T_f^{N^a})$ for a fixed complete incremental verifier V_Δ as the time taken by V_Δ that starts from $T_0^{N^a}$ and halts with the final tree $T_f^{N^a}$. $Time_h(H, H_\Delta)$ and $Time_t(T_f^N, T_0^{N^a})$ are the time for constructing H_Δ from H, and $T_0^{N^a}$ from T_f^N respectively. We pose the optimal incremental verification problem as an optimization problem of finding the best $H_\Delta, T_0^{N^a}$ such that the time of incremental verification is minimized. Formally, we state the problem as:

$$\underset{H_\Delta, T_0^{N^a}}{\arg\min} \left[Time_\Delta(T_0^{N^a}, T_f^{N^a}) + Time_h(H, H_\Delta) + Time_t(T_f^N, T_0^{N^a}) \right] \tag{3}$$

The search space for $T_0^{N^a}$ is exponential in terms of $\mathcal{R}$, and the search space for H_Δ is infinite. Further, $Time_\Delta$ is a complicated function of $H_\Delta, T_0^{N^a}$ that does not have a closed-form formulation. As a result, it is not possible to find an optimal solution.

Simplifying Assumptions: To make the problem tractable we make a simplifying assumption that for all networks with the same architecture, each branching and bounding step on each invocation takes a constant time t_H and t_A respectively. We can now compute $Time_\Delta(T_0^{N^a}, T_f^{N^a})$ as:

Theorem 1. *($Time_\Delta$ for incremental verification). If the incremenatl verifier V_Δ halts with the final tree $T_f^{N^a}$, then $Time_\Delta(T_0^{N^a}, T_f^{N^a}) = (t_A + t_H) \cdot \left(|Nodes(T_f^{N^a})| + \frac{1 - |Nodes(T_0^{N^a})|}{2} \right) - t_H \cdot |Leaves(T_f^{N^a})|$.*

The proof of the theorem is in Appendix 9.2.

In this work, we focus on a class of algorithms for which the preprocessing times $Time_h(H, H_\Delta)$ and $Time_t(T_f^N, T_0^{N^a})$ are $<<$ $Time_\Delta(T_0^{N^a}, T_f^{N^a})$. Furthermore, we also focus on branching heuristics used in practice where $t_H << t_A$. Equation 3 simplifies to finding H_Δ and $T_0^{N^a}$ such that the following expression $Time_\Delta(T_0^{N^a}, T_f^{N^a}) = t_A \cdot \left(|Nodes(T_f^{N^a})| + \frac{1-|Nodes(T_0^{N^a})|}{2}\right)$ is minimized. Rewriting and ignoring the constant term we get

$$Time_\Delta(T_0^{N^a}, T_f^{N^a}) = t_A \cdot \left(\frac{|Nodes(T_f^{N^a})| - |Nodes(T_0^{N^a})|}{2} + \frac{|Nodes(T_f^{N^a})|}{2}\right) \tag{4}$$

4.3 IVAN Algorithm for Incremental Verification

We describe the novel components of our algorithm and present the full workflow in Algorithm 5. Our first technique called reuse focuses on minimizing $|Nodes(T_f^{N^a})| - |Nodes(T_0^{N^a})|$ in Equation 4. Our second reorder technique focuses on minimizing $|Nodes(T_f^{N^a})|$. The H_Δ, $T_0^{N^a}$ obtained by reuse and reorder are distinct. IVAN algorithm combines these distinct solutions, to reduce $Time_\Delta(T_0^{N^a}, T_f^{N^a})$.

Reuse: This technique is based on the observation that the BaB specification trees should be similar for small perturbations in the network. Accordingly, in the method, we use the final specification tree for N as the initial tree for the verification of N^a i.e. $T_0^{N^a} = T_f^N$, and keep the $H_\Delta = H$ unchanged. We formally characterize a set of networks obtained by small perturbation for which $T_0^{N^a} = T_f^N$ is sufficient for verifying N^a without any further splitting in Section 4.4.

Reorder: Reorder technique focuses on improving the branching heuristic H such that it reduces $|Nodes(T_f^{N^a})|$, and $T_0^{N^a}$ is single node tree with n_0 encoding the specification (ϕ, ψ). If we start $T_0^{N^a} = T_f^N$, $|Nodes(T_f^{N^a})|$ is at least $|Nodes(T_f^N)|$, and thus, we start $T_0^{N^a}$ from scratch allowing the technique to minimize $|Nodes(T_f^{N^a})|$. We create a branching function H_Δ from H with the following two changes. (i) The splits that worked effectively for the verification of the N should be prioritized. (ii) The splits that were not effective should be deprioritized. To formalize the effectiveness of splits, we define the $LB_N(n)$ as the lower bound computed by the analyzer A on the network N for proving the property φ_n encoded by the node n. Further, using the function LB_N we define an improvement function I_N represents the effectiveness of a ReLU split at a specific node as:

$$I_N(n, r) = \min(LB_N(n_r) - LB_N(n), LB_N(n_l) - LB_N(n)) \tag{5}$$

where $n_l, n_r \in Children(n)$ in the specification tree T_f^N. We use I_N to define the observed effectiveness $H_{obs}(r)$ from a split r on the entire specification tree for N. It is defined as the mean of the improvement over each node where split r was made. Let $Q \subset Nodes(T_f^N)$ denote a set of nodes where split r was made. Then,

$$H_{obs}(r) = \frac{\sum_{n \in Q} I_N(n, r)}{|Q|}. \tag{6}$$

Using the $H_{obs}(r)$ score we update the existing branching function as:

$$H_\Delta(n, r) = \alpha \cdot H(n, r) + (1 - \alpha) \cdot (H_{obs}(r) - \theta). \tag{7}$$

Here, we introduce two hyperparameters α and θ. The hyperparameter $\alpha \in [0, 1]$ controls the importance given to the actual heuristic score and the observed improvement from the verification on N. If $\alpha = 1$, then H_Δ depends only on the original branching heuristic score. If $\alpha = 0$, then it fully relies on observed split scores. The hyperparameter θ ensures that our score positively changes score for r that have $H_{obs}(r) > \theta$ and negatively change scores for $H_{obs}(r) < \theta$.

Constructing a Pruned Specification Tree: The two reordering goals of prioritizing and deprioritizing effective and ineffective splits are difficult to combine with reuse. However, instead of starting from scratch, we can construct a specification tree T_P from T_f^N excluding the ineffective splits. For $n \in Nodes(T_f^N)$, where ReLU r splits n, we denote the set of bad splits as the set $\mathcal{B}(T_f^N)$ of the pairs (n, r) such that the improvement score $I_N(n, r) \leq \theta$. For $(n, r) \in \mathcal{B}(T_f^N)$ while constructing the pruned tree our algorithm chooses a child n_k of n. If a ReLU r_k is split at n_k in T_f^N, it performs a split r_k in the corresponding node in T_P, and skips over the bad split r. The subtree corresponding to the other child $n_{k'}$ is eliminated and not added to our pruned tree. We choose n_k such that:

$$n_k = \underset{n_u \in Children(n)}{\arg\min} \; LB_N(n_u) - LB_N(n) \tag{8}$$

We choose such n_k over $n_{k'}$ since $LB_N(n)$ is closer to $LB_N(n_k)$ than $LB_N(n_{k'})$. Further, combining Equation 5 and 8, we can show $(LB_N(n_k) - LB_N(n)) < \theta$, i.e. their difference is bounded. We anticipate that on the omission of the split r, the subtree corresponding to n_k is a better match to the necessary branching decisions following n than $n_{k'}$.

Algorithm 4 presents the top-down construction of T_P. The algorithm starts from the root of T_f^N and recursively traverses through the children constructing T_P. It maintains a queue Q of nodes yet to be explored and a map M that maps nodes from the tree T_f^N to the corresponding new nodes in T_P. At a node n, if (n, r) is not a bad split, it performs the split r at the corresponding mapping $\hat{n}$. Otherwise, if r_k is the split at n_k, it skips over r and performs the split of r_k at $\hat{n}$. The newly created children from a split of $\hat{n}$ are associated with children of n_k using M. The children of n_k are added in the Q and they are recursively processed in the next iteration for further constructing T_P. T_P is still a specification tree satisfying the Definition 4.1 by construction. The specifications φ_n of a node n in T_P can be constructed using a path from the root to n.

Algorithm 4 Creating a Pruned Tree

```
Input: specification tree $T_f^{N^a}$, hyperparameter $\theta$
Output: Pruned tree $T_P$
$n_{root} \leftarrow$ root of $T_f^{N^a}$, $\hat{n}_{root} \leftarrow$ copy of $n_{root}$
$T_P \leftarrow$ Initialize a new tree with $\hat{n}_{root}$
$Q \leftarrow$ Initialize list with $n_{root}$
$M \leftarrow$ Initialize an empty map
$M[n_{root}] \leftarrow \hat{n}_{root}$
while $Q$ is not empty do
    $n \leftarrow Q.pop()$; $r \leftarrow$ split at node $n$; $\hat{n} \leftarrow M[n]$
    if $I_N(n, r) < \theta$ then
        $n_k \leftarrow \arg\min_{n_k \in Children(n)} LB_N(n_k) - LB_N(n)$
        $r_k \leftarrow$ split at node $n_k$
        $n_l, n_r \leftarrow n_k.children$; $\hat{n}_l, \hat{n}_r \leftarrow Split(T_P, \hat{n}, r_k)$
        $M[n_l] \leftarrow \hat{n}_l; M[n_r] \leftarrow \hat{n}_r$
        $Q.push(n_l); Q.push(n_r)$
    else
        $n_l, n_r \leftarrow n.children$; $\hat{n}_l, \hat{n}_r \leftarrow Split(T_P, \hat{n}, r)$
        $M[n_l] \leftarrow \hat{n}_l; M[n_r] \leftarrow \hat{n}_r$
        $Q.push(n_l); Q.push(n_r)$
return $T_P$
```

Algorithm 5 Incremental Verification Algorithm

```
Input: Original network $N$,
Perturbed network $N^a$,
property $(\phi, \psi)$,
analyzer $A$,
branching heuristic $H$,
hyperparameters
$\alpha$ and $\theta$,
incremental verifier $V_\Delta$
Output: Verification result for $N$ and $N^a$
$resultN, T_f^N \leftarrow V(N, \phi, \psi, H)$
$T_0^{N^a} \leftarrow$ PrunedTree$(T_f^N, \theta)$
$H_\Delta \leftarrow$ UpdateH$(H, T_f^N, \theta, \alpha)$
$resultN^a \leftarrow V_\Delta(N^a, \phi, \psi, T_0^{N^a}, H_\Delta)$    ▷ Incremental verification step calls Algorithm 3
return $resultN, resultN^a$
```

Main algorithm: Algorithm 5 presents IVAN's main algorithm for incremental verification that combines all the aforementioned techniques. It takes as inputs the original network N, a perturbed network N^a, input specification ϕ, and an output property ψ. It prunes the final tree T_f^N obtained in the verification of N and constructs $T_0^{N^a}$ (line 2). It computes the updated branching heuristic H_Δ using Equation 7 (line 3). It uses $T_0^{N^a}$ and H_Δ for performing fast incremental verification of networks N^a (line 4).

We next state the following lemma that states - verifying the property (ϕ, ψ) is equivalent to verifying the specifications for all the leaves.

Lemma 1. *The specifications encoded by the leaf nodes of a specification tree T maintain the following invariance.*

$$\left(\bigwedge_{n \in leaves(T)} \varphi_n \right) \iff (\phi \rightarrow \psi)$$

We next use the lemma to prove the soundness and completeness of our algorithm. All the proofs are in Appendix 9.2.

Theorem 2. *(Soundness of Verification Algorithm). If Algorithm 5 verifies the property (ϕ, ψ) for the network N^a, then the property must hold.*

Theorem 3. *(Completeness of Verification Algorithm). If for the network N^a, the property (ϕ, ψ) holds then Algorithm 5 always terminates and produces* Verified *as output.*

Scope of IVAN: IVAN utilizes the specification tree to store the trace of the BaB proof. The IVAN algorithm enhances this tree by reusing and refining it to enable faster BaB proof of updated networks. Our paper focuses on using IVAN to verify ReLU networks with BaB that implements ReLU splitting. However, we expect that IVAN's principles can be extended to networks with other activation functions (tanh, sigmoid, leaky ReLU) for which BaB has been applied for verification.

4.4 Network Perturbation Bounds

In this section, we formally characterize a class of perturbations on a network N where our proposed "Reuse" technique attains maximum possible speed-up. Specifically, we focus on modifications affecting only the last layer which represent many practical network perturbations (e.g, transfer learning, fine-tuning). The last layer modification assumption is only for our theoretical results in this section. Our experiments make no such assumption and consider perturbations applied across the original network.

We leave the derivation of perturbation bounds corresponding to the full IVAN to future work as it requires theoretically modeling the effect of arbitrary network perturbations on DNN output as well as complex interactions between "Reuse" and "Reorder" techniques. Given a specification tree T and network architecture $\mathcal{N}$, we identify a set of neural networks $\mathbf{C}_T(\mathcal{N})$ such that any network $N^a \in \mathbf{C}_T(\mathcal{N})$ can be verified by reusing T.

We assume the weights are changed by the weight perturbation matrix $\mathcal{E}$. If $N_l = ReLU(A_l \cdot X + B_l)$ then last layer of N^a is $N_l^a = ReLU((A_l + \mathcal{E}) \cdot X + B_l)$.

Definition 11 (Last Layer Perturbed Network). *Given a network N with architecture $\mathcal{N}$, the set of last layer perturbed networks is $\mathcal{M}(N, \delta) \subseteq \mathcal{N}$, such that if $N^a \in \mathcal{M}(N, \delta)$ then $(\forall i \in [l-1]) \cdot N_i = N_i^a$, $N_l = ReLU(A_l \cdot X + B_l)$, $N_l^a = ReLU((A_l + \mathcal{E}) \cdot X + B_l)$ and $\|\mathcal{E}\|_F \leq \delta$.* [1]

We next compute the upper bound of δ, for which if the property can be proved/disproved using specification tree T in N then the same property can be proved/disproved in N^a using the same T. Therefore, once we have the proof tree T that verifies the property in N we can reuse T for

[1] $\| \cdot \|_F$ denotes the Frobenius norm of a matrix

Table 1. Models and the perturbation ϵ used for the evaluation for incremental verification.

Model	Architecture	Dataset	#Neurons	Training Method	ϵ
ACAS-XU Networks	6 × 50 linear layers	ACAS-XU	300	Standard [Julian et al. 2019]	-
FCN-MNIST	2 × 256 linear layers	MNIST	512	Standard	0.02
CONV-MNIST	2 Conv, 2 linear layers	MNIST	9508	Certified Robust [Balunovic and Vechev 2020]	0.1
CONV-CIFAR	2 Conv, 2 linear layers	CIFAR10	4852	Empirical Robust [Dong et al. 2018]	$\frac{2}{255}$
CONV-CIFAR-WIDE	2 Conv, 2 linear layers	CIFAR10	6244	Certified Robust [Wong and Kolter 2018a]	$\frac{4}{255}$
CONV-CIFAR-DEEP	4 Conv, 2 linear layers	CIFAR10	6756	Certified Robust [Wong and Kolter 2018a]	$\frac{4}{255}$

verifying any perturbed network $N^a \in \mathcal{M}(N, \delta)$. Assuming the property (ϕ, ψ) and the analyzer A are the same for any perturbed network $N^a \in \mathcal{M}(N, \delta)$ the upper bound of δ only depends on N and T.

We next introduce some useful notations that help us explicitly compute the upper bound of δ. Given T let $\mathcal{F}(N_i, T)$ be the over-approximated region computed by the analyzer A that contains all feasible outputs N_i of the i-th layer of the original network. Note $\mathcal{F}(N_i, T)$ depends on the ϕ and analyzer A but we omit them to simplify the notation. Let $V_{\mathcal{T}}(N, T)$ denote whether the property (ϕ, ψ) can be verifed on network N with T. Proof of Theorem 4 is presented in Section 9.3

$$LB(\mathcal{F}(N_l, T)) = \min_{Y \in \mathcal{F}(N_l, T)} C^T Y \tag{9}$$

$$V_{\mathcal{T}}(N, T) = (LB(\mathcal{F}(N_l, T)) \geq 0) \tag{10}$$

$$\eta(N, T) = \max_{Y \in \mathcal{F}(N_{l-1}, T)} \|Y\|_2 \tag{11}$$

Theorem 4. *If $\delta \leq \frac{|LB(\mathcal{F}(N_l,T))|}{\|C\|_2 \cdot \eta(N,T)}$ then for any perturbed network $N^a \in \mathcal{M}(N, \delta)$ $V_{\mathcal{T}}(N, T) \iff V_{\mathcal{T}}(N^a, T)$.*

The proof of the theorem is in Appendix 9.3.

5 METHODOLOGY

Networks and Properties. We evaluate IVAN on models with various architectures that are trained with different training methods. Similar to most of the previous literature, we verify L_∞-based local robustness properties for MNIST and CIFAR10 networks and choose standard ϵ values used for evaluating complete verifiers. For the verification of global properties in Section 6.4 we use the standard set of ACAS-XU properties that are part of the VNN-COMP benchmarks [Bak et al. 2021]. Table 1 presents the evaluated models and the choice of ϵ for the local robustness properties.

Network Perturbation. Similar to previous works [Paulsen et al. 2020a; Ugare et al. 2022], we use quantization to generate perturbed networks. Specifically, we use int8 and int16 post-training quantizations. The quantization scheme has the form [TFLite 2017]: $r = s(q - zp)$. Here, q is the quantized value and r is the real value; s which is the scale and zp which is the zero point are the parameters of quantization. Our experiments use symmetric quantization with $zp = 0$.

Baseline. We use the following baseline BaB verifiers:

- For proving the local robustness properties, we use LP-based triangle relaxation for bounding [Bunel et al. 2020b; Ehlers 2017], and we use the estimation based on coefficients of the zonotopes for choosing the ReLU splitting [Henriksen and Lomuscio 2021].
- For the verification of ACAS-XU global properties, we use RefineZono [Singh et al. 2019c]. RefineZono uses DeepZ [Singh et al. 2018] analyzer with input splitting. This baseline is used only for experiments in Section 6.4.

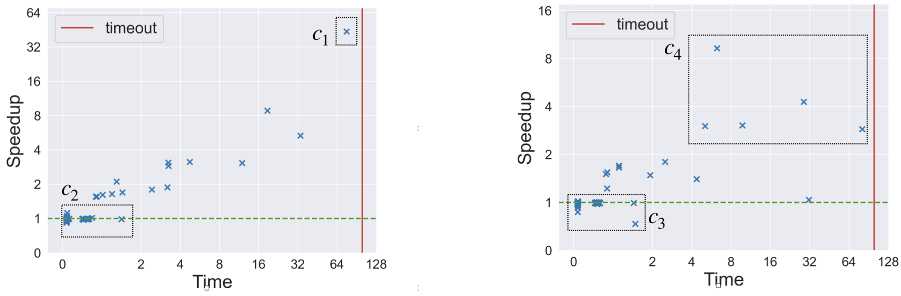
Experimental Setup. We use 64 cores of an AMD Ryzen Threadripper CPU with the main memory of 128 GB running the Linux operating system. The code for our tool is written in Python. We use the GUROBI [Gurobi Optimization, LLC 2018] solver for our LP-based analyzer.

Hyperparameters. We use Optuna tuner [Akiba et al. 2019] for tuning the hyperparameters. We present more details and sensitivity analysis of the hyperparameters in Section 6.3.

6 EXPERIMENTAL EVALUATION

We evaluate the effectiveness of IVAN in verifying the local robustness properties of the quantized networks. We then analyze how various tool components contribute to the overall result. We further show the sensitivity of speedup obtained by IVAN to the hyperparameters. We also stress-test IVAN on large random perturbation to the network. Finally, we evaluate the effectiveness of IVAN on global property verification with input splitting.

6.1 Effectiveness of IVAN



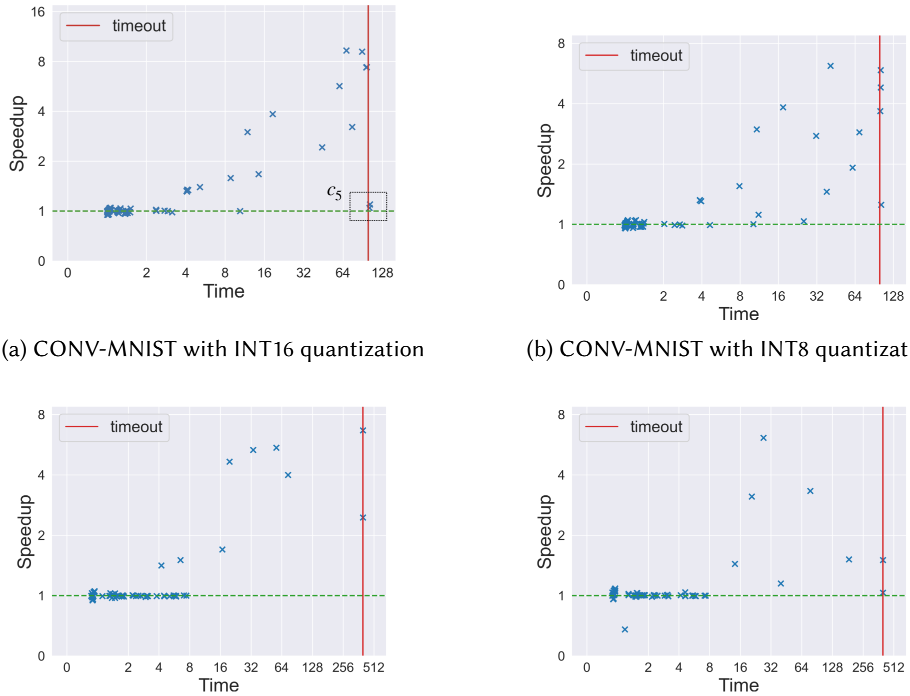
(a) FCN-MNIST with INT16 quantization (b) FCN-MNIST with INT8 quantization

Fig. 6. IVAN speedup for the verification of local robustness properties on FCN-MNIST .

Figure 6 presents the speedup obtained by IVAN on FCN-MNIST . The x-axis displays the time taken by the baseline verifier for the verification in Seconds. The y-axis denotes the speedup obtained by IVAN over the baseline on a specific verification instance. Each cross in the plot shows results for a specific verification property. The vertical line denotes the timeout for the experiment and the dashed line is to separate instances that have a speedup greater than 1x.

We observe that IVAN gets higher speedup on hard instances that take more time for verification on the baseline. IVAN has a small overhead for storing the specification tree compared to the baseline. For hard specifications that result in large specification trees, this overhead is insignificant compared to the improvement in the verification time. Our techniques that reuse and refine the tree focus on speeding up such hard specifications. However, for specifications that are easy to prove with small specification trees, we see a slight slowdown in verification time. Since these easy specifications are verified quickly by both IVAN and the baseline, they are irrelevant in overall verification time over all the specifications. For instance, the box labeled by c_2 in Figure 6a contains all of the 83 cases with low ($< 1.2x$) speedup on int16 quantized network. Despite low speedup, all of them take 16.27s to verify with IVAN. Whereas the case labeled by c_1 alone takes 75.54s on the baseline and 1.73s on IVAN, leading to a 43x speedup – caused by reducing BaB tree size from 345 nodes to 28 nodes on pruning, out of which only 14 leaf nodes are active and lead to analyzer calls.

We observe a similar pattern in the case of the int8 quantized network in Figure 6b. It shows that the cases confined in box c_3, despite having lower speedup, take relatively less time. The cases included in box c_4 in Figure 6b have a much higher impact on the overall verification time. Box c_3 includes the majority of the low-speedup 83 cases that take a total of 18.44s time for verification with IVAN. Whereas for 5 cases in box c_4 with higher speedups, take 401 analyzer calls with baseline and 118 analyzer calls with IVAN. Accordingly, solving them takes 130.6s with the baseline and 40.26s with IVAN, leading to a 3.3x speedup.



(a) CONV-MNIST with INT16 quantization

(b) CONV-MNIST with INT8 quantization

(c) CONV-CIFAR-WIDE with INT16 quant.

(d) CONV-CIFAR-WIDE with INT8 quant.

Fig. 7. IVAN speedup for the verification of local robustness properties.

Figure 7 presents speedup for several other networks. IVAN is notably more effective on hard-to-verify specifications that take more than 10s to verify using the baseline. It achieves 3.1x geomean speedup on such cases. In many cases, we see more solved cases by IVAN over the baseline. For instance, the box c_5 in Figure 7a contains 2 cases that baseline does not solve within the timeout of 100s, but IVAN solves them in 90.6s and 95.8s each. We show speedup vs. time plots for other networks (CONV-CIFAR , CONV-CIFAR-DEEP) and more statistics of our evaluation in Appendix 9.1.

6.2 Overall Speedup

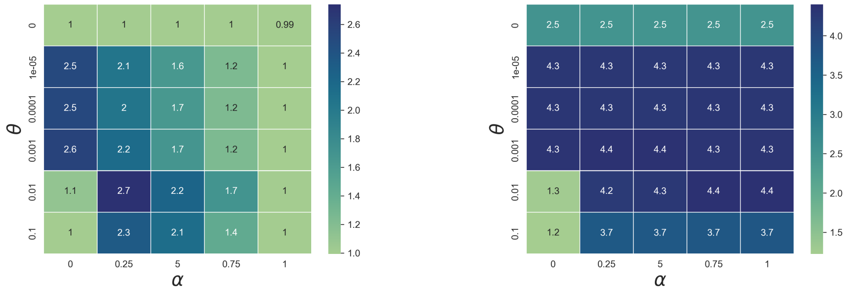
We observe no cases when the baseline verifies the property and IVAN exceeds the timeout. We cannot compute the speedup for the cases where the baseline exceeds the timeout. Therefore, we compute the overall speedup over the set S that denotes all the cases that are solved by the baseline within the time limit. $\tau_B(c)$ and $\tau_{IVAN}(c)$ denote the time taken by baseline and IVAN on the case c respectively, then we compute the overall speedup as $Sp = \frac{\sum_{c \in S} \tau_B(c)}{\sum_{c \in S} \tau_{IVAN}(c)}$.

Table 2 presents the comparison of the contribution of each technique used in IVAN for each model. Column *+Solved* in each case displays the number of extra verification problems solved by the technique in comparison to the baseline. Columns in IVAN[Reuse] present results on only using the reuse technique. Columns in IVAN[Reorder] show results on using the reorder technique. Columns in IVAN present the results on using all techniques from Section 4. Column *Sp* for each technique demonstrates the overall speedup obtained compared to the baseline. We observe that in most case combination of all techniques performs better than reuse and reordering. We see that reorder performs better than reuse except for one case (FCN-MNIST on int8).

Table 2. Ablation study for overall speedup across all properties for different techniques in IVAN.

Model	Approximation	IVAN[Reuse]		IVAN[Reorder]		IVAN	
		Sp	*+Solved*	*Sp*	*+Solved*	*Sp*	*+Solved*
FCN-MNIST	int16	2.51x	0	2.71x	0	**4.43x**	0
	int8	1.07x	0	1.64x	0	**2.02x**	0
CONV-MNIST	int16	1.62x	0	2.15x	0	**3.09x**	2
	int8	1.27x	2	1.34x	3	**1.71x**	4
CONV-CIFAR	int16	1.02x	0	1.57x	2	**2.52x**	2
	int8	1.08x	0	1.53x	0	**1.78x**	0
CONV-CIFAR-WIDE	int16	1.43x	1	1.51x	0	**1.87x**	2
	int8	0.75x	0	**1.62x**	1	1.53x	**2**
CONV-CIFAR-DEEP	int16	1.64x	0	2.29x	0	**3.21x**	0
	int8	1.15x	0	1.13x	1	**1.25x**	1

6.3 Hyperparameter Sensitivity Analysis



(a) IVAN[Reorder]

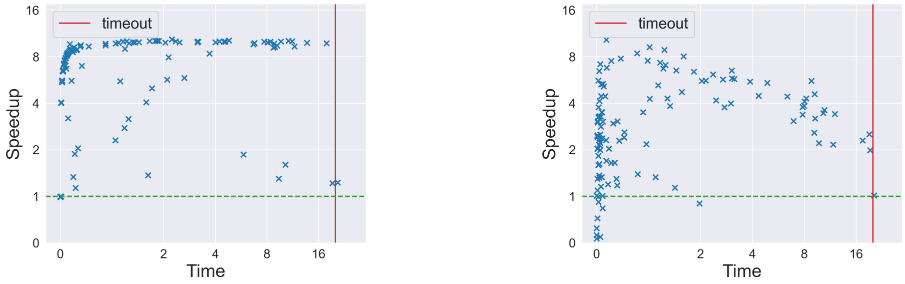
(b) IVAN

Fig. 8. Speedup for the combination of hyperparameter values on FCN-MNIST with int16 quantization.

Figure 8 plots the heatmap for IVAN speedup on various hyperparameter values. The x-axis shows the hyperparameter α value and the y-axis shows the θ value. Each point in the greed is annotated with the observed *Sp* on setting the corresponding hyperparameter values. Figure 8a presents the plot for IVAN with on reorder technique. $(\alpha, \theta) = (0.25, 0.01)$ is the highest speedup point. Choosing $\theta = 0$ implies that are not deprioritizing the splitting decisions that did not work well. In that case, we observe no speedup with reordering, showing the necessity of θ in our H_Δ formulation. Figure 8b presents the same plot for our main algorithm that also reuses the pruned tree. We observe that the speedup is less sensitive to hyperparameter value changes in this plot. This is expected since reordering starts from single node $T_0^{N^a}$ and purely relied on H_Δ formulation for the speedup. While our main technique also reuses the tree, even when $\theta = 0$ it can get ~2.5x speedup.

6.4 Global Properties with Input Splitting

We show that IVAN is effective in speeding up the state-of-the-art verifier RefineZono [Singh et al. 2019c] when verifying global properties. This baseline employs input splitting based on a strong branching strategy. Figure 9 presents the speedup achieved by IVAN over this baseline. Overall, IVAN achieves a 9.5x speedup in the int16 quantization case and a 3.1x speedup in the int8 quantization case. Previous work has observed that ACAS-XU properties take a large number of splits with most analyzers. For the int16 case, the average value of $|T_f^{N^a}|$ with our baseline is 285.4.



(a) ACAS-XU networks with INT16 quantization

(b) ACAS-XU networks with INT8 quantization

Fig. 9. IVAN speedup for the verification of global ACAS-XU properties.

The baseline takes a total of 305s time for verifying cases that have large tree $|T_f^{N^a}| > 5$. IVAN verifies those properties in 32s.

6.5 Random Weight Perturbations

In this experiment, we stress-test IVAN for incremental verification by applying uniform random perturbation on the DNN weights. Here, we perturbed each weight in the network by 2%, 5%, and 10%. Even the smallest of these perturbations (2%) to each of the weights already induces larger overall changes in the network than those caused by practical methods such as quantization, pruning, and fine-tuning that often non-uniformly affect specific layers of the network. For each network and perturbation, we run IVAN and the baseline to verify 100 properties and compute the average speedup of IVAN over the baseline.

Table 3. IVAN speedup on uniform random weight perturbations

Model	Weight perturbation		
	2%	5%	10%
FCN-MNIST	1.65x	1.57x	0.87x
CONV-MNIST	1.97x	0.57x	0.57x
CONV-CIFAR	1.29x	1.09x	0.69x
CONV-CIFAR-WIDE	1.42x	1.08x	0.96x
CONV-CIFAR-DEEP	1.32x	1.06x	1.05x

Table 3 presents the average speedups obtained by IVAN. Each row shows the IVAN speedup under various weight perturbations for a particular network. We see that in most cases IVAN speedup reduces as the perturbations to the weights increase. It is because the specification tree for the perturbed network is no longer similar to the one for the original network. If IVAN is used in such cases, it uses suboptimal splits leading to higher verification time.

7 RELATED WORK

Neural Network Verification: Recent works introduced several techniques for verifying properties of neural networks [Anderson et al. 2019, 2020; Bunel et al. 2020b; Ehlers 2017; Kabaha and Drachsler-Cohen 2022; Katz et al. 2017b; Laurel et al. 2022; Tjeng et al. 2017; Wang et al. 2018, 2021; Yang et al. 2022]. For BaB-based complete verification, previous works used distinct strategies for ReLU splitting. Ehlers [2017] and Katz et al. [2017a] used random ReLU selection for splitting. Wang et al. [2018] computes scores based on gradient information to rank ambiguous ReLU nodes. Similarly, Bunel et al. [2020b] compute scores based on a formula based on the estimation equations in [Wong and Kolter 2018b]. Henriksen and Lomuscio [2021] use coefficients of zonotopes for these scores.
Incremental Neural Network Verification: Fischer et al. [2022] presented the concept of sharing certificates between specifications. They reuse the proof for L_∞ specification computed with abstract interpretation-based analyzers based on the notion of proof templates, for faster verification of patch and geometric perturbations. Ugare et al. [2022] showed that the reusing of proof is possible

between networks. It uses a similar concept of network adaptable proof templates. It is limited to certain properties (patch, geometric, L_0) and works with abstract interpretation-based incomplete verifiers. Wei and Liu [2021] considers incremental incomplete verification of relatively small DNNs with last-layer perturbation. All of these works cannot handle incremental and complete verification of diverse specifications, which is the focus of our work

Differential Neural Network Verification: ReluDiff [Paulsen et al. 2020a] presented the concept of differential neural network verification. The follow-up work of [Paulsen et al. 2020b] made it more scalable. ReluDiff can be used for bounding the difference in the output of an original network and a perturbed network corresponding to an input region. ReluDiff uses input splitting to perform complete differential verification. Our method is complementary to ReluDiff and can be used to speed up the complete differential verification with multiple perturbed networks, performing it incrementally. Cheng and Yan [2020] reuse previous interval analysis results for the verification of the fully-connected networks where the specifications are only defined over the last linear layer of an updated network. In contrast, IVAN performs end-to-end verification and operates on a more general class of networks, specifications, and perturbations.

Warm Starting Mixed Integer Linear Programming (MILP) Solvers: State-of-the-art MILP solvers such as GUROBI [Gurobi Optimization, LLC 2018] and CPLEX [Cplex 2009] support warm starting that can accelerate the optimization performance. MILP can warm start based on initial solutions that are close to the optimal solution. This allows MILP solvers to avoid exploring paths that do not improve on the provided initial solution and can help the solver to converge faster. The exact implementation details of these closed-sourced commercial solvers are unavailable. Regardless, our experiments with MILP warm starting of GUROBI for incremental DNN verification showed insignificant speedup.

Incremental Program Verification: Incremental verification has improved the scalability of traditional program verification to an industrial scale [Johnson et al. 2013; Lakhnech et al. 2001; O'Hearn 2018; Stein et al. 2021]. Incremental program analysis tasks reuse partial results [Yang et al. 2009], constraints [Visser et al. 2012] and precision information [Beyer et al. 2013] from previous runs for faster analysis of individual commits. Frequently, the changes made by the program are limited to a small portion of the overall program (and its analysis requires significant attention to the impact on control flow). whereas DNN updates typically alter the weights of multiple layers throughout the network (but with no impact on control flow). Therefore, incremental DNN verification presents a distinct challenge compared to the incremental verification of programs.

Incremental SMT Solvers: Modern SMT solvers such as Z3 [De Moura and Bjørner 2008] and CVC5 [Barbosa et al. 2022] during constraint solving learn lemmas, which are later reused to solve similar problems. The incrementality of these solvers is restricted to the addition or deletion of constraints. They do not consider reuse in cases when the constraints are perturbed as in our case.

8 CONCLUSION

Current complete approaches for DNN verification re-run the verification every time the network is modified. In this paper, we presented IVAN, the first general, incremental, and complete DNN verifier. IVAN captures the trace of the BaB-based complete verification through the specification tree. We evaluated our IVAN on combinations of networks, properties, and updates. IVAN achieves up to 43x speedup and geometric mean speedup of 2.4x in verifying DNN properties.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments. This research was supported in part by NSF Grants No. CCF-1846354, CCF-1956374, CCF-2008883, CCF-2217144, CCF-2238079, CNS-2148583, USDA NIFA Grant No. NIFA-2024827 and Qualcomm innovation fellowship.

REFERENCES

Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. 2019. Optuna: A Next-generation Hyperparameter Optimization Framework. In *Proceedings of the 25rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.*

Aws Albarghouthi. 2021. *Introduction to Neural Network Verification.* verifieddeeplearning.com. arXiv:2109.10317 [cs.LG] http://verifieddeeplearning.com.

Filippo Amato, Alberto López, Eladia María Peña-Méndez, Petr Vaňhara, Aleš Hampl, and Josef Havel. 2013. Artificial neural networks in medical diagnosis. *Journal of Applied Biomedicine* 11, 2 (2013).

Greg Anderson, Shankara Pailoor, Isil Dillig, and Swarat Chaudhuri. 2019. Optimization and Abstraction: A Synergistic Approach for Analyzing Neural Network Robustness. In *Proc. Programming Language Design and Implementation (PLDI).*

Ross Anderson, Joey Huchette, Will Ma, Christian Tjandraatmadja, and Juan Pablo Vielma. 2020. Strong mixed-integer programming formulations for trained neural networks. *Mathematical Programming* (2020).

Stanley Bak, Changliu Liu, and Taylor T. Johnson. 2021. The Second International Verification of Neural Networks Competition (VNN-COMP 2021): Summary and Results. *CoRR* abs/2109.00498 (2021). arXiv:2109.00498 https://arxiv.org/abs/2109.00498

Stanley Bak, Hoang-Dung Tran, Kerianne Hobbs, and Taylor T. Johnson. 2020. Improved Geometric Path Enumeration for Verifying ReLU Neural Networks. In *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12224)*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer, 66–96. https://doi.org/10.1007/978-3-030-53288-8_4

Mislav Balunovic and Martin Vechev. 2020. Adversarial Training and Provable Defenses: Bridging the Gap. In *International Conference on Learning Representations.* https://openreview.net/forum?id=SJxSDxrKDr

Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13243)*, Dana Fisman and Grigore Rosu (Eds.). Springer, 415–442. https://doi.org/10.1007/978-3-030-99524-9_24

Dirk Beyer, Stefan Löwe, Evgeny Novikov, Andreas Stahlbauer, and Philipp Wendler. 2013. Precision Reuse for Efficient Regression Verification. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013).* Association for Computing Machinery, New York, NY, USA, 389–399. https://doi.org/10.1145/2491411.2491429

Davis W. Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John V. Guttag. 2020. What is the State of Neural Network Pruning?. In *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020.*

Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. 2016. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316* (2016).

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

Rudy R Bunel, Oliver Hinder, Srinadh Bhojanapalli, and Krishnamurthy Dvijotham. 2020a. An efficient nonconvex reformulation of stagewise convex optimization problems. *Advances in Neural Information Processing Systems* 33 (2020).

Jiefeng Chen, Yixuan Li, Xi Wu, Yingyu Liang, and Somesh Jha. 2022. Robust Out-of-distribution Detection for Neural Networks. In *AAAI-22 Workshop on Adversarial Machine Learning and Beyond.*

Chih-Hong Cheng and Rongjie Yan. 2020. Continuous Safety Verification of Neural Networks. arXiv:2010.05689 [cs.LG]

IBM ILOG Cplex. 2009. V12. 1: User's Manual for CPLEX. *International Business Machines Corporation* 46, 53 (2009), 157.

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08).* Springer-Verlag, Berlin, Heidelberg, 337–340.

Yinpeng Dong, Fangzhou Liao, Tianyu Pang, Hang Su, Jun Zhu, Xiaolin Hu, and Jianguo Li. 2018. Boosting Adversarial Attacks With Momentum. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR).*

Souradeep Dutta, Susmit Jha, Sriram Sankaranarayanan, and Ashish Tiwari. 2017. Output Range Analysis for Deep Neural Networks. *CoRR* abs/1709.09130 (2017). arXiv:1709.09130 http://arxiv.org/abs/1709.09130

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

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.* https://openreview.net/forum?id=l_amHf1oaK

Marc Fischer, Christian Sprecher, Dimitar I. Dimitrov, Gagandeep Singh, and Martin T. Vechev. 2022. Shared Certificates for Neural Network Verification. In *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13371)*, Sharon Shoham and Yakir Vizel (Eds.). Springer, 127–148. https://doi.org/10.1007/978-3-031-13185-1_7

Aymeric Fromherz, Klas Leino, Matt Fredrikson, Bryan Parno, and Corina Pasareanu. 2021. Fast Geometric Projections for Local Robustness Certification. In *International Conference on Learning Representations*. https://openreview.net/forum?id=zWy1uxjDdZJ

Feisi Fu and Wenchao Li. 2022. Sound and Complete Neural Network Repair with Minimality and Locality Guarantees. In *International Conference on Learning Representations*. https://openreview.net/forum?id=xS8AMYiEav3

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)*.

Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. 2021. A Survey of Quantization Methods for Efficient Neural Network Inference. *CoRR* abs/2103.13630 (2021). arXiv:2103.13630

Tejas Gokhale, Rushil Anirudh, Bhavya Kailkhura, Jayaraman J. Thiagarajan, Chitta Baral, and Yezhou Yang. 2021. Attribute-Guided Adversarial Training for Robustness to Natural Perturbations. In *AAAI*. AAAI Press, 7574–7582.

Gurobi Optimization, LLC. 2018. Gurobi Optimizer Reference Manual.

Patrick Henriksen and Alessio Lomuscio. 2021. DEEPSPLIT: An Efficient Splitting Method for Neural Network Verification via Indirect Effect Analysis. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, Zhi-Hua Zhou (Ed.). International Joint Conferences on Artificial Intelligence Organization, 2549–2555. https://doi.org/10.24963/ijcai.2021/351 Main Track.

Kenneth Johnson, Radu Calinescu, and Shinji Kikuchi. 2013. An Incremental Verification Framework for Component-Based Software Systems. In *Proceedings of the 16th International ACM Sigsoft Symposium on Component-Based Software Engineering* (Vancouver, British Columbia, Canada) *(CBSE '13)*. Association for Computing Machinery, New York, NY, USA, 33–42. https://doi.org/10.1145/2465449.2465456

Kyle D. Julian, Mykel J. Kochenderfer, and Michael P. Owen. 2018. Deep Neural Network Compression for Aircraft Collision Avoidance Systems. *CoRR* abs/1810.04240 (2018).

Kyle D. Julian, Mykel J. Kochenderfer, and Michael P. Owen. 2019. Deep Neural Network Compression for Aircraft Collision Avoidance Systems. *Journal of Guidance, Control, and Dynamics* 42, 3 (mar 2019), 598–608. https://doi.org/10.2514/1.g003724

Anan Kabaha and Dana Drachsler-Cohen. 2022. Boosting Robustness Verification of Semantic Feature Neighborhoods. https://doi.org/10.48550/ARXIV.2209.05446

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

Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. 2017b. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10426)*. https://doi.org/10.1007/978-3-319-63387-9_5

Yassine Lakhnech, Saddek Bensalem, Sergey Berezin, and Sam Owre. 2001. Incremental Verification by Abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems: 7th International Conference, TACAS 2001*, T. Margaria and W. Yi (Eds.), Vol. 2031. Springer-Verlag, Genova, Italy, 98–112.

Jacob Laurel, Rem Yang, Atharva Sehgal, Shubham Ugare, and Sasa Misailovic. 2021. Statheros: Compiler for Efficient Low-Precision Probabilistic Programming. In *Design Automation Conference (DAC)*. 787–792.

Jacob Laurel, Rem Yang, Shubham Ugare, Robert Nagel, Gagandeep Singh, and Sasa Misailovic. 2022. A General Construction for Abstract Interpretation of Higher-Order Automatic Differentiation. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 161 (oct 2022), 29 pages. https://doi.org/10.1145/3563324

Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2017. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083* (2017).

Christoph Müller, Francois Serre, Gagandeep Singh, Markus Püschel, and Martin Vechev. 2021. Scaling Polyhedral Neural Network Verification on GPUs. *Proc. Machine Learning and Systems (MLSys)* (2021).

Peter W. O'Hearn. 2018. Continuous Reasoning: Scaling the impact of formal methods. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, Anuj Dawar and Erich Grädel (Eds.). ACM, 13–25. https://doi.org/10.1145/3209108.3209109

Alessandro De Palma, Harkirat S. Behl, Rudy R. Bunel, Philip H. S. Torr, and M. Pawan Kumar. 2021. Scaling the Convex Barrier with Active Sets. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*.

Brandon Paulsen, Jingbo Wang, and Chao Wang. 2020a. ReluDiff: differential verification of deep neural networks. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020.* https://doi.org/10.1145/3377811.3380337

Brandon Paulsen, Jingbo Wang, Jiawei Wang, and Chao Wang. 2020b. NEURODIFF: Scalable Differential Verification of Neural Networks using Fine-Grained Approximation. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020.* https://doi.org/10.1145/3324884.3416560

Hadi Salman, Greg Yang, Huan Zhang, Cho-Jui Hsieh, and Pengchuan Zhang. 2019. A Convex Relaxation Barrier to Tight Robustness Verification of Neural Networks. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada.*

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

Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin Vechev. 2018. Fast and effective robustness certification. *Advances in Neural Information Processing Systems* 31 (2018).

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).

Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. 2019c. Boosting Robustness Certification of Neural Networks. In *International Conference on Learning Representations.*

Matthew Sotoudeh and Aditya V. Thakur. 2019. Computing Linear Restrictions of Neural Networks. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada.*

Benno Stein, Bor-Yuh Evan Chang, and Manu Sridharan. 2021. Demanded abstract interpretation. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 282–295. https://doi.org/10.1145/3453483.3454044

Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. 2014. Intriguing properties of neural networks. In *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings.*

Nima Tajbakhsh, Jae Y Shin, Suryakanth R Gurudu, R Todd Hurst, Christopher B Kendall, Michael B Gotway, and Jianming Liang. 2016. Convolutional neural networks for medical image analysis: Full training or fine tuning? *IEEE transactions on medical imaging* 35, 5 (2016), 1299–1312.

TFLite. 2017. TF Lite post-training quantization. https://www.tensorflow.org/lite/performance/post_training_quantization.

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

Shubham Ugare, Gagandeep Singh, and Sasa Misailovic. 2022. Proof transfer for fast certification of multiple approximate neural networks. *Proc. ACM Program. Lang.* 6, OOPSLA (2022), 1–29. https://doi.org/10.1145/3527319

Caterina Urban and Antoine Miné. 2021. A Review of Formal Methods applied to Machine Learning. https://doi.org/10.48550/ARXIV.2104.02466

Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. 2012. Green: Reducing, Reusing and Recycling Constraints in Program Analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (Cary, North Carolina) *(FSE '12).* Association for Computing Machinery, New York, NY, USA, Article 58, 11 pages. https://doi.org/10.1145/2393596.2393665

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

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 Complete and Incomplete Neural Network Verification. *arXiv preprint arXiv:2103.06624* (2021).

Tianhao Wei and Changliu Liu. 2021. Online Verification of Deep Neural Networks under Domain or Weight Shift. *CoRR* abs/2106.12732 (2021). arXiv:2106.12732 https://arxiv.org/abs/2106.12732

Karl Weiss, Taghi M Khoshgoftaar, and DingDing Wang. 2016. A survey of transfer learning. *Journal of Big data* 3, 1 (2016), 1–40.

Eric Wong and Zico Kolter. 2018a. Provable defenses against adversarial examples via the convex outer adversarial polytope. In *International Conference on Machine Learning.*

Eric Wong and Zico Kolter. 2018b. Provable Defenses against Adversarial Examples via the Convex Outer Adversarial Polytope. In *Proceedings of the 35th International Conference on Machine Learning.*

Kaidi Xu, Zhouxing Shi, Huan Zhang, Yihan Wang, Kai-Wei Chang, Minlie Huang, Bhavya Kailkhura, Xue Lin, and Cho-Jui Hsieh. 2020. Automatic Perturbation Analysis for Scalable Certified Robustness and Beyond. (2020).

Guowei Yang, Matthew B. Dwyer, and Gregg Rothermel. 2009. Regression model checking. In *2009 IEEE International Conference on Software Maintenance.* 115–124. https://doi.org/10.1109/ICSM.2009.5306334

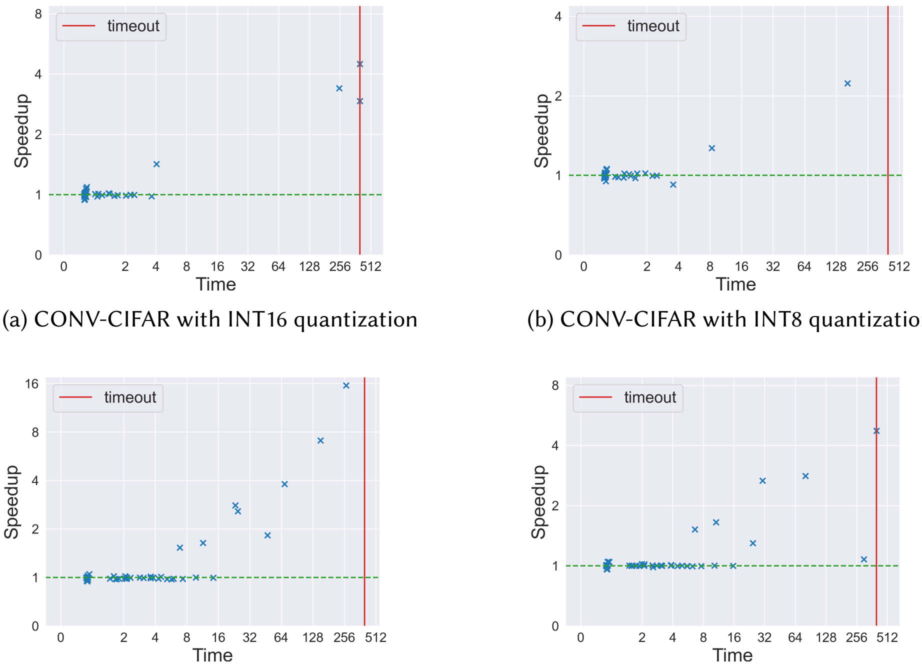
Rem Yang, Jacob Laurel, Sasa Misailovic, and Gagandeep Singh. 2022. Provable Defense Against Geometric Transformations. arXiv:2207.11177 [cs.LG]

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. In *Advances in Neural Information Processing Systems*, Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (Eds.). https://openreview.net/forum?id=5haAJAcofjc

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

9 APPENDIX

9.1 Evaluation Results



(a) CONV-CIFAR with INT16 quantization

(b) CONV-CIFAR with INT8 quantization

(c) CONV-CIFAR-DEEP with INT16 quantization

(d) CONV-CIFAR-DEEP with INT8 quantization

Fig. 10. IVAN speedup for the verification of locus robustness properties.

We give more detailed statistics on our experiments in Table 4. We separate the results into two cases easy instances ($|T_f^N| \leq 5$) and hard instances ($|T_f^N| > 5$). IVAN focuses on hard instances, thus we observe more speedup in those cases. Column v/c/u shows the number of instances that are *Verified*, *Counterexample*, and *Unknown* respectively. Columns $Cost_{base}$ and $Cost_{IVAN}$ present the average number of analyzer calls made by baseline and IVAN respectively. Columns $Solved_{base}$ and $Solved_{IVAN}$ show the number of instances solved by the baseline and IVAN respectively. Columns $Time_{base}$ and $Time_{IVAN}$ give the time taken for verification by the baseline and IVAN respectively.

							$\|T_f^N\| \leq 5$				$\|T_f^N\| > 5$			
Model	Perturbation	Cases	v/c/u	v/c/u	$Cost_{base}$	$Cost_{IVAN}$	$Solved_{base}$	$Solved_{IVAN}$	$Time_{base}$	$Time_{IVAN}$	$Solved_{base}$	$Solved_{IVAN}$	$Time_{base}$	$Time_{IVAN}$
FCN-MNIST	int16	100	85/13/2	85/13/2	7.53	1.94	86	86	19.06	18.06	12	12	160.39	22.81
	int8	100	85/13/2	85/13/2	7.24	4.04	86	86	20.73	20.14	12	12	174.87	79.84
CONV-MNIST	int16	96	74/8/14	76/8/12	8.64	3.74	72	72	123.22	113.51	10	12	779.4	298.91
	int8	96	71/8/17	75/8/13	8.95	3.71	70	70	110.59	104.65	9	13	713.35	284.01
CONV-CIFAR	int16	56	33/8/15	35/8/13	30.16	7.12	40	40	44.07	42.85	1	3	1054.29	300.36
	int16	56	33/8/15	33/8/15	5.63	3.07	40	40	47.77	46.09	1	1	164.28	73.61
CONV-CIFAR-WIDE	int8	73	29/32/12	31/32/10	11.16	3.44	56	56	131.14	127.89	5	7	1002.33	272.6
	int16	73	30/32/11	32/32/9	17.56	11.19	56	56	123.08	123.98	6	8	1168.83	829.82
CONV-CIFAR-DEEP	int16	59	27/23/9	27/23/9	6.32	1.76	45	45	152.37	131.44	5	5	557.32	90.24
	int8	59	25/23/11	26/23/10	9.04	4.47	45	45	151.32	137.61	3	4	812.87	396.85

Table 4. Summary of statistics on verifying all models with baseline and IVAN

We observe that IVAN significantly improves the verifier's performance for verification of specifications that need specification trees of size greater than 5. These proofs contribute to a substantial portion of overall time, and thus IVAN offers a large overall speedup. We observe that

IVAN offers insignificant improvement in the verification time for cases with a small specification tree T_f^N. These cases are verified by the baseline in less time. There is not too much our techniques such as reordering and pruning and can do for the already compact trees.

9.2 Proofs for Theorems

Theorem 1. *(Time$_\Delta$ for incremental verification). If the incremenatl verifier V_Δ halts with the final tree $T_f^{N^a}$, then $Time_\Delta(T_0^{N^a}, T_f^{N^a}) = (t_A + t_H) \cdot \left(|Nodes(T_f^{N^a})| + \frac{1-|Nodes(T_0^{N^a})|}{2}\right) - t_H \cdot |Leaves(T_f^{N^a})|$.*

Proof. The incremental verifier starts by bounding $Leaves(T_0^{N^a})$ to check the property, and if needed, it recursively branches the nodes further.

Consequently, the verifier performs the bounding step for $Leaves(T_0^{N^a})$ and all the new nodes in $T_f^{N^a}$ added to $T_0^{N^a}$.

It performs the branching step for all newly added internal nodes in the specification tree. The number of new internal nodes can be computed as $(|Nodes(T_f^{N^a})| - |Leaves(T_f^{N^a})| - |Nodes(T_0^{N^a})| + |Leaves(T_0^{N^a})|)$

Accordingly, we can compute:

$$\begin{aligned}
Time_\Delta(T_0^{N^a}, T_f^{N^a}) &= t_A \cdot (|Nodes(T_f^{N^a})| - |Nodes(T_0^{N^a})| + |Leaves(T_0^{N^a})|) \\
&\quad + t_H \cdot (|Nodes(T_f^{N^a})| - |Leaves(T_f^{N^a})| - |Nodes(T_0^{N^a})| + |Leaves(T_0^{N^a})|) \\
&= (t_A + t_H) \cdot (|Nodes(T_f^{N^a})| - |Nodes(T_0^{N^a})| + |Leaves(T_0^{N^a})|) - t_H \cdot |Leaves(T_f^{N^a})|
\end{aligned}$$

□

Lemma 1. *The specifications encoded by the leaf nodes of a specification tree T maintain the following invariance.*

$$\left(\bigwedge_{n \in leaves(T)} \varphi_n\right) \iff (\phi \rightarrow \psi)$$

Proof. We prove this claim using structural induction on the specification tree.

It is trivially true for the specification tree of a single node. Since $\varphi_{n_{root}} = \phi \rightarrow \psi$, we can conclude $\varphi_{n_{root}} \iff \phi \rightarrow \psi$.

For the inductive step, we assume that for a tree T_i with i splits, the hypothesis is true. If we split node $n_s \in Leaves(T_i)$, we get a tree T_{i+1} with leaf nodes n_s^+ and n_s^-.

$$\left(\bigwedge_{n \in leaves(T_i)} \varphi_n\right) \iff (\phi \rightarrow \psi) \qquad \text{(Induction hypothesis)}$$

$$\left(\bigwedge_{n \in leaves(T_i)/n_s} \varphi_n\right) \wedge \varphi_{n_s} \iff (\phi \rightarrow \psi)$$

$$\left(\bigwedge_{n \in Leaves(T_i)/n_s} \varphi_n\right) \wedge \varphi_{n_s^+} \wedge \varphi_{n_s^-} \iff (\phi \rightarrow \psi) \qquad \text{(From equation 2)}$$

$$\left(\bigwedge_{n \in Leaves(T_{i+1})} \varphi_n\right) \iff (\phi \rightarrow \psi) \qquad \text{(Combining previous equations)}$$

Hence, the invariance is true for the specification tree T_{i+1} as well. This completes our induction, and hence, our hypothesis is proved. □

Lemma 2. *(Termination). Algorithm 5 always terminates.*

Proof. At each specification tree node, we split a ReLU $r \in \mathcal{R}$ that was not split before. Since $|\mathcal{R}|$ is finite, the specification tree cannot have depth $> |\mathcal{R}|$. Thus, Algorithm 3 always terminates. □

Theorem 2. *(Soundness of Verification Algorithm). If Algorithm 5 verifies the property (ϕ, ψ) for the network N^a, then the property must hold.*

Proof. Let $\mathcal{T}_{\mathcal{N}}$ be the set of specification trees over the architecture $\mathcal{N}$, such that $N, N^a \in \mathcal{N}$ By construction $T_f^N \in \mathcal{T}_{\mathcal{N}}$. Algorithm 5 prunes T_f^N to get the tree $T_0^{N^a}$. We see that $T_0^{N^a} \in \mathcal{T}_{\mathcal{N}}$ since our deletion operation preserves the specification tree property of the tree.
Further branching from $T_0^{N^a}$ leads to the final tree $T_f^{N^a}$ during the incremental verification. The branching step performs the multiple *Split* operations, and therefore $T_f^{N^a} \in \mathcal{T}_{\mathcal{N}}$.
Algorithm 3 removes a node from the active list only when it is verified. (Line 2) and A is a sound analyzer for the bounding step for verifying each node (Definition 5).
Thus, if Algorithm 5 returns *Verified* then for each leaf node $n \in Leaves(T)$, φ_n holds.
Since $T_f^{N^a} \in \mathcal{T}_{\mathcal{N}}$, we can use the Lemma 1 and conclude that the property (ϕ, ψ) must hold. □

Theorem 3. *(Completeness of Verification Algorithm). If for the network N^a, the property (ϕ, ψ) holds then Algorithm 5 always terminates and produces* Verified *as output.*

Proof. The proof of termination is in Lemma 2. We prove the claim, (ϕ, ψ) holds then the Algorithm 5 returns *Verified* through contradiction. Suppose Algorithm 5 does not return *Verified.* Since the algorithm always terminates, it must terminate with a *Counterexample.*
From Lemma 1 we know that $(\phi \rightarrow \psi) \implies \varphi_n$. This can be transformed to $\neg\varphi_n \implies \neg(\phi \rightarrow \psi)$
Thus, if our algorithm returns a *Counterexample* for a specification tree node n that implies $\neg\varphi_n$ holds for some node n.
Hence, this statement implies $\neg(\phi \rightarrow \psi)$
This contradicts the assumption of this theorem. Hence, Algorithm 5 must return *Verified.* □

9.3 Proofs for Network Perturbation Bounds

Theorem 4. *If $\delta \leq \frac{|LB(\mathcal{F}(N_l,T))|}{\|C\|_2 \cdot \eta(N,T)}$ then for any perturbed network $N^a \in \mathcal{M}(N, \delta)$ $V_{\mathcal{T}}(N,T) \iff V_{\mathcal{T}}(N^a,T)$.*

Proof. We first show that if $\delta \leq \frac{|LB(\mathcal{F}(N_l,T))|}{\|C\|_2 \cdot \eta(N,T)}$ then $\neg V_{\mathcal{T}}(N,T) \implies \neg V_{\mathcal{T}}(N^a,T)$. The specification tree T could not verify the property on N then $\exists Y \in \mathcal{F}(N_l,T)$ such that $C^T Y = LB(\mathcal{F}(N_l,T)) < 0$. We show that the same specification tree (T) can not prove the property on any $N^a \in \mathcal{M}(N,\delta)$ by showing that $LB(N_l^a,T) < 0$. In the following part of proof we show that $\exists Y' \in \mathcal{F}(N_l^a,T)$ such that $C^T Y' < 0$ which makes $LB(\mathcal{F}(N_l^a,T)) < 0$.

$$\begin{aligned}
C^T Y' &= C^T Y + C^T (Y' - Y) \\
&\leq C^T Y + \|C\|_2 \|(Y' - Y)\|_2 \\
&\leq C^T Y + \|C\|_2 \cdot \delta \cdot \eta(N_l,T) \quad \text{(Using Lemma 3)} \\
&< LB(\mathcal{F}(N_l,T)) + |LB(\mathcal{F}(N_l,T))| \leq 0 \quad \text{given} \quad LB(\mathcal{F}(N_l,T)) < 0
\end{aligned}$$

We now show that $\delta \leq \frac{|LB(\mathcal{F}(N_l,T))|}{\|C\|_2 \cdot \eta(N,T)}$ then $V_{\mathcal{T}}(N,T) \implies V_{\mathcal{T}}(N^a,T)$. We prove this by contradiction. Suppose $LB(N_l^a,T) < 0$ then $\exists Y' \in \mathcal{F}(N_l^a,T)$ such that $C^T Y' < 0$. Swapping N with N^a in lemma 3 we can show $Y \in \mathcal{F}(N_l,T)$ such that $\|Y - Y'\|_2 \leq \delta \cdot \eta(N^a,T)$. Given perturbation is done only at the final layer $\forall i \in [l-1]$ $N_i = N_i^a$ which implies $\eta(N,T) = \eta(N^a,T)$. As $Y \in \mathcal{F}(N_l,T)$ then $C^T y \geq LB(\mathcal{F}(N_l,T))$.

$$\begin{aligned}
C^T Y' &= C^T Y + C^T(Y' - Y) \\
&\geq C^T Y - \|C\|_2 \|(Y' - Y)\|_2 \\
&\geq C^T Y - \|C\|_2 \cdot \delta \cdot \eta(N,T) \quad \text{(Using Lemma 3)} \\
&\geq LB(\mathcal{F}(N_l,T)) - |LB(\mathcal{F}(N_l,T))| \geq 0 \quad \text{given} \quad LB(\mathcal{F}(N_l,T)) \geq 0
\end{aligned}$$

The above derivation shows that $C^T Y' \geq 0$ which contradicts the assumptions that $C^T Y' < 0$ and $LB(\mathcal{F}(N^a,T)) < 0$. □

Theorem 5. *The incremental verification time on any perturbed network* $N^a \in \mathcal{M}(N,\delta)$ *with* $\delta \leq \frac{|LB(\mathcal{F}(N_l,T))|}{\|C\|_2 \cdot \eta(N,T)}$ *is* $t_A \cdot |Leaves(T)|$ *provided* $T_0^{N^a} = T$ *and* $V_{\mathcal{T}}(N,T) = True$.

Proof. This result directly follows from Theorem 4. As $V_{\mathcal{T}}(N,T) = True$, the proposed algorithm will terminate within $Leaves(T)$ number of bounding steps. □

$$LB(N,\mathbf{T}) = \min_{T \in \mathbf{T}} \frac{|LB(\mathcal{F}(N_l,T))|}{\eta(N,T)} \tag{12}$$

For any network $N^a \in \mathcal{M}(N,\delta)$ with $\delta \leq \frac{|LB(N_l,\mathbf{T})|}{\|C\|_2}$ the baseline verifier can only verify the property with T_f^N. (using results from Theorem 4). Therefore, the baseline verifier makes at least $|Nodes(T_f^N)|$ number of analyzer calls before terminating. While for any perturbed network N^a the incremental verifier always terminates within $|Leaves(T_f^N)|$ analyzer calls. (Theorem 5) Assuming all bounding steps take the same time then the speed up achieved by the incremental verifier over the baseline is $\frac{|Nodes(T_f^N)|}{|Leaves(T_f^N)|}$. Note we assume the same branching heuristic is used by both the baseline and incremental verifier such that $\mathbf{T}_{IVAN} = \mathbf{T}_{baseline} = \mathbf{T}$.

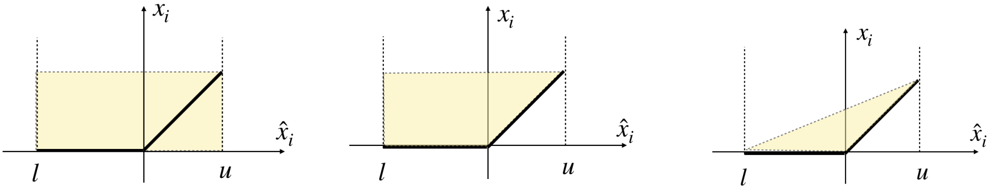
Theorem 6. *For any perturbed network* $N^a \in \mathcal{M}(N,\delta)$ *with* $\delta \leq \frac{|LB(N,\mathbf{T})|}{\|C\|_2}$ *the incremental verifier with* $T_0^{N^a} = T_f^N$ *always achives speed up of* $\frac{|Nodes(T_f^N)|}{|Leaves(T_f^N)|}$ *over the baseline verifier provided* $V_{\mathcal{T}}(N,T_f^N) = True$ *and the branching heuristic H is unchanged.*

Proof. As $\frac{|LB(N,\mathbf{T})|}{\|C\|_2} \leq \frac{|LB(\mathcal{F}(N,T_f^N))|}{\|C\|_2 \cdot \eta(N,T_f^N,l-1)}$ the proposed algorithm always terminate within $|Leaves(T_f^N)|$ number of bounding steps for any $N^a \in \mathcal{M}(N,\delta)$. (Using Theorem 5). Apart from T_f^N all specification trees $T \in \mathbf{T}$ were unsuccessful in verifying the property for N. As shown in the following derivation all specification trees apart from T_f^N will fail to prove the property for any network $N^a \in \mathcal{M}(N,\delta)$.

$$\begin{aligned}
\forall T \in \mathbf{T}(T < T_f^N) &\implies \neg V_{\mathcal{T}}(N,T) \\
&\implies \neg V_{\mathcal{T}}(N^a,T) \quad \text{[Using Theorem 4]}
\end{aligned}$$

□

In this part, we briefly explain how the analyzers handle non-linear activation functions like *ReLU* unit while verifying neural networks. This is helpful in understanding the following proofs. Let $x = ReLU(\hat{x})$ represents a relu unit with input $\hat{x}$ and output x. As described in Section 3.2 we cannot linearize ambiguous *ReLU* units where $lb(\hat{x}) < 0 < ub(\hat{x})$. Therefore, the analyzer over-approximates the output of ambiguous *ReLU* unit using convex relaxation. sec:proofs2 convex relaxations for ambiguous *ReLU*s shown in Fig. 11, the box relaxation is easiest to compute but is imprecise. On the other hand, triangle relaxation is the most precise but complicated, while quadrilateral relaxation achieves a middle ground between them. For all theoretical derivation presented below, we assume that the analyzer uses quadrilateral relaxation because it is more precise than box relaxation and simpler than triangle relaxation. Let $A^{\#}_{ReLU}(\hat{x})$ denotes over-approximated convex region of $ReLU(\hat{x})$. For any ambiguous *ReLU* unit, $x = ReLU(\hat{x})$ the $A^{\#}_{ReLU}(\hat{x})$ under quadrilateral relaxation is defined by the following constraints.



(a) Box relaxation (b) Qudrilateral relaxation (c) Triangle relaxation

Fig. 11. Different convex relaxations of ambiguous *ReLU* units. The dark line represents the actual output of the *ReLU* unit and the shaded regions represent the over-approximated convex relaxations of the output.

$$x \geq 0 \qquad x \geq \hat{x} \qquad ub(\hat{x}) \geq x \tag{13}$$

Lemma 3. *Let N and N^a be two l-layer networks with the same architecture and weight perturbation made only at the last layer l. If $\|\mathcal{E}\|_F \leq \delta$ then $\forall Y \in \mathcal{F}(N_l, T), \exists Y' \in \mathcal{F}(N^a_l, T)$ such that $\|Y - Y'\|_2 \leq \delta \cdot \eta(N, T)$.*

Proof. Let $Y[j]$ denotes the j-th coordinate of Y and $A[j]$ and $\mathcal{E}[j]$ represent the j-th row of A and $\mathcal{E}$ respectively. $Y \in \mathcal{F}(N_l, T)$ then $\exists X \in \mathcal{F}(N_{(l-1)}, T)$ such that $\hat{Y} = A_l X + B_l$ and $(\forall j)$, $Y[j] \in A^{\#}_{ReLU}(\hat{Y}[j])$. As first $l - 1$ layers of both N and N^a are same then $X \in \mathcal{F}(N^a_{l-1}, T)$. Let $\hat{Y'} = (A_l + \mathcal{E})X + B_l$. We first show that $(\forall j)$, $|\hat{Y}[j] - \hat{Y'}[j]| \leq \|\mathcal{E}[j]\|_2 \cdot \eta(N, T)$.

$$\begin{aligned} |\hat{Y}[j] - \hat{Y'}[j]| &= |\mathcal{E}[j]X| \\ &\leq \|\mathcal{E}[j]\|_2 \cdot \|X\|_2 \\ &\leq \|\mathcal{E}[j]\|_2 \cdot \eta(N, T) \end{aligned}$$

In the following part of the proof we show how to constuct $Y'[j]$ such that $|Y'[j] - Y[j]| \leq \|\mathcal{E}[j]\|_2 \cdot \eta(N, T)$ while ensuring that $Y'[j] \in A^{\#}_{ReLU}(\hat{Y'}[j])$. In this case $A^{\#}_{ReLU}(Y'[j])$ is the convex region defined by constraints presented in Eq. 13.

- **Case 1:** $\hat{Y'}[j] \geq 0$
  - **Case 1.a** $ub(\hat{Y'}[j]) \geq Y[j] \geq \hat{Y'}[j])$
    In this case $Y'[j] = Y[j]$ satisfies the constraints defined in Eq. 13. Therefore, $Y'[j] = Y[j] \in A^{\#}_{ReLU}(Y'\hat{[}j])$.

- **Case 1.b** $\hat{Y'}[j] > Y[j]$
  $Y \in \mathcal{F}(N_l, T)$ then $Y[j] \in A^{\#}_{ReLU}(\hat{Y}[j])$ and $Y[j] \geq \hat{Y}[j]$. We show below for $Y'[j] = \hat{Y'}[j] \in A^{\#}_{ReLU}(\hat{Y}[j])$ and $|Y[j] - Y'[j]| < \|\mathcal{E}[j]\|_2 \cdot \eta(N, T)$

$$\begin{aligned}|Y[j] - Y'[j]| &= |Y[j] - \hat{Y'}[j]| \quad \text{(for Case 1.b we define } Y'[j] = \hat{Y}[j]) \\ &\leq |\hat{Y}[j] - \hat{Y'}[j]| \quad (\textit{given} \;\; \hat{Y'}[j] > Y[j] \geq \hat{Y}[j]) \\ &\leq \|\mathcal{E}[j]\|_2 \cdot \eta(N, T)\end{aligned}$$

- **Case 1.c** $Y[j] > ub(\hat{Y'}[j])$
  For this case we define $Y'[j] = ub(\hat{Y'}[j])$. We show below that $|Y[j] - Y'[j]| \leq \|\mathcal{E}[j]\|_2 \cdot \eta(N, T)$.

$$\begin{aligned}|Y[j] - Y'[j]| &= |Y[j] - ub(\hat{Y'}[j])| \quad \text{(for Case 1.c we define } Y'[j] = ub(\hat{Y'}[j])) \\ &\leq |ub(\hat{Y}[j]) - ub(\hat{Y'}[j])| \quad (\textit{given} \;\; ub(\hat{Y}[j]) \geq Y[j] > ub(\hat{Y'}[j])) \\ &\leq \|\mathcal{E}[j]\|_2 \cdot \eta(N, T)\end{aligned}$$

- **Case 2:** $\hat{Y'}[j] < 0$
  - **Case 2.a** $ub(\hat{Y'}[j]) \geq Y[j] \geq 0$
    Similar to case 1.a we define $Y'[j] = Y[j]$.
  - **Case 2b.** $Y[j] > ub(\hat{Y'}[j])$
    For this case we define $Y'[j] = max(0, ub(\hat{Y'}[j]))$. The proof $|Y[j] - Y'[j]| \leq \|\mathcal{E}[j]\|_2 \cdot \eta(N, T)$ is same as case 1.c.

In all the previous cases we assumed j-th *ReLU* unit of the final layer is not splitted. Otherwise for $Y[j]$ either 0 or $Y[j] = \hat{Y}[j]$. Similarly $Y'[j]$ either 0 or $Y'[j] = \hat{Y'}[j]$. In all of these cases $|Y[j] - Y'[j]| \leq \|\mathcal{E}[j]\|_2 \cdot \eta(N, T)$ as we already proved $|\hat{Y}[j] - \hat{Y'}[j]| \|\mathcal{E}[j]\|_2 \cdot \eta(N, T)$ All these cases we have shown that $(\forall j)\ |Y[j] - Y'[j]| \leq \|\mathcal{E}[j]\|_2 \cdot \eta(N, T)$ with $Y' \in \mathcal{F}(N^a, T)$

$$\begin{aligned}\|Y - Y'\|_2^2 &= \sum_{j=1}^{n_{l+1}} |Y[j] - Y'[j]|^2 \\ \|Y - Y'\|_2^2 &\leq \eta(N, T)^2 \cdot \sum_{j=1}^{n_{l+1}} \mathcal{E}[j]^2 \\ \|Y - Y'\|_2 &\leq \delta \cdot \eta(N, T)\end{aligned}$$

□