Abstraction and Refinement: Towards Scalable and Exact Verification of Neural Networks

JIAXIANG LIU, Shenzhen University, Shenzhen, China
YUNHAN XING, Shenzhen University, Shenzhen, China
XIAOMU SHI, State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
FU SONG, State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
ZHIWU XU, Shenzhen University, Shenzhen, China
ZHONG MING, Shenzhen University & Shenzhen Technology University, Shenzhen, China

As a new programming paradigm, deep neural networks (DNNs) have been increasingly deployed in practice, but the lack of robustness hinders their applications in safety-critical domains. While there are techniques for verifying DNNs with formal guarantees, they are limited in scalability and accuracy. In this article, we present a novel counterexample-guided abstraction refinement (CEGAR) approach for scalable and exact verification of DNNs. Specifically, we propose a novel abstraction to break down the size of DNNs by over-approximation. The result of verifying the abstract DNN is conclusive if no spurious counterexample is reported. To eliminate each spurious counterexample introduced by abstraction, we propose a novel counterexample-guided refinement that refines the abstract DNN to exclude the spurious counterexample while still over-approximating the original one, leading to a sound, complete yet efficient CEGAR approach. Our approach is orthogonal to and can be integrated with many existing verification techniques. For demonstration, we implement our approach using two promising tools, Marabou and Planet, as the underlying verification engines, and evaluate on widely used benchmarks for three datasets `ACAS Xu`, `MNIST`, and `CIFAR-10`. The results show that our approach can boost their performance by solving more problems in the same time limit, reducing on average 13.4%–86.3% verification time of Marabou on almost all the verification tasks, and reducing on average 8.3%–78.0% verification time of Planet on all the verification tasks. Compared to the most relevant CEGAR-based approach, our approach is 11.6–26.6 times faster.

CCS Concepts: • **Computing methodologies** → **Neural networks**; • **Security and privacy** → **Software security engineering**; • **Software and its engineering** → **Software verification;**

Fu Song is also with University of Chinese Academy of Sciences, and Nanjing Institute of Software Technology.

This work is partially supported by the Natural Science Foundation of Guangdong Province (Grants No. 2022A1515011458 and No. 2022A1515010880), Shenzhen Science and Technology Innovation Commission (Grant No. JCYJ20210324094202008), the National Natural Science Foundation of China (Grants No. 62272315, No. 62072309, No. 62372304, and No. 62002228), CAS Project for Young Scientists in Basic Research (Grant No. YSBR-040), ISCAS New Cultivation Project (Grant No. ISCAS-PYFX-202201), and ISCAS Fundamental Research Project (Grants No. ISCAS-JCZD-202302 and No. ISCAS-JCMS-202308).

Authors' addresses: J. Liu, Y. Xing, and Z. Xu, Shenzhen University, Shenzhen, China, 518060; e-mails: jiaxiang0924@gmail.com, xingyunhan@email.szu.edu.cn, xuzhiwu@szu.edu.cn; X. Shi (Corresponding author) and F. Song, State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China, 100190; e-mails: {shixm, songfu}@ios.ac.cn; Z. Ming, Shenzhen University & Shenzhen Technology University, Shenzhen, China, 518060; e-mail: mingz@szu.edu.cn.

ACM 1049-331X/2024/06-ART129
https://doi.org/10.1145/3644387

Additional Key Words and Phrases: Neural networks, abstraction, refinement, CEGAR, formal verification, robustness

ACM Reference Format:
Jiaxiang Liu, Yunhan Xing, Xiaomu Shi, Fu Song, Zhiwu Xu, and Zhong Ming. 2024. Abstraction and Refinement: Towards Scalable and Exact Verification of Neural Networks. *ACM Trans. Softw. Eng. Methodol.* 33, 5, Article 129 (June 2024), 35 pages. https://doi.org/10.1145/3644387

1 INTRODUCTION

Due to surprising breakthroughs in solving various challenging tasks such as image recognition [39] and natural language processing [18], deep learning has arguably become a new programming paradigm that takes over traditional software programs in many areas. For instance, **deep neural networks (DNNs)** are increasingly being deployed in safety-critical applications, e.g., autonomous driving [48] and medical systems [28]. Despite their great success, DNNs are notoriously fragile to slight perturbations and thus are lack of robustness [4, 8, 17, 24, 35, 43], raising serious concerns when they are deployed in safety-critical applications as such errors may lead to catastrophes. Therefore, it is important to guarantee the quality of DNNs before to deploy them in safety-critical applications.

Along with traditional verification and validation research in software engineering, quality assurance of DNNs has attracted a dramatically increasing interest in the software engineering community. A large quantity of testing techniques tailored to DNNs have been proposed, e.g., References [4, 30, 31, 36, 42, 44], cf. Reference [54] for a survey. While testing techniques are often cost-effective in finding violations of properties, e.g., counterexamples (called adversarial examples in the security community) to falsify robustness, they are not able to prove the absence of violations. In another research direction, a variety of formal verification techniques have been proposed to rigorously verify DNNs with formal guarantees [3, 11, 12, 19, 21, 22, 27, 38, 51, 55, 56]. Early work relies on using constraint solvers (e.g., Marabou [22] and Planet [11]), often providing both soundness and completeness guarantees. However, their efficiency and scalability are limited due to the intrinsic computational complexity, e.g., NP-complete even for simple neural networks and properties [21]. To address these issues, abstract interpretation has been used to verify DNNs, at the cost of accuracy [14, 40, 41, 46, 49, 52]. Although few of them incorporate refinement strategies to improve accuracy [41, 49, 52], it remains a great challenge to efficiently and accurately verify large-scale DNNs. One of the most promising techniques used in formal verification to improve the efficiency and scalability is ***counterexample-guided abstraction refinement*** **(CEGAR)** framework [5]. The essential idea of CEGAR is that, when given a target system S to verify, an over-approximation, small-sized system $\bar{S}$ is constructed by *abstraction* and verified by an off-the-shelf tool. The result is always conclusive if no spurious counterexample is reported. Otherwise, to regain precision, the abstract system $\bar{S}$ is refined guided by the spurious counterexample to exclude it. The verification process is repeated on the refined system until the original system is proved or a genuine counterexample is found.

To instantiate the CEGAR framework, one needs to address the following two questions: (i) how to abstract a target system and (ii) how to refine an abstract system. When instantiating CEGAR in DNN verification, there are four technical challenges:

- **C1**: The abstraction should guarantee *soundness*, i.e., if an abstract DNN is proved robust, then the target DNN must be robust.
- **C2**: The abstraction should reduce the network size as much as possible and preserve accuracy as much as possible, since coarse-grained abstract DNNs may result in plenty of spurious counterexamples, thus requiring more refinement steps.

- **C3**: The refinement must preserve soundness as well, similar to challenge C1, and also excludes a given counterexample quickly.
- **C4**: The refinement should regain the accuracy as much as possible, and enlarge the network size as little as possible.

In this article, inspired by the first structure-oriented CEGAR-based approach for DNN verification, named CEGAR-NN [12], we present a scalable and exact CEGAR-based approach for DNN verification by proposing *novel* procedures for abstraction and refinement, which address the above challenges.

We propose a novel abstraction procedure defined as a synergistic integration of two *abstraction primitives*: Merge for merging two hidden neurons and Freeze for removing a hidden neuron, both of which are able to reduce network size. To address challenge C1, these abstraction primitives are well-designed according to the weights and bounds of hidden neurons, thus provide soundness guarantees.

More specifically, we first classify the effect of each hidden neuron into either inc or dec: when the values of the other hidden neurons in the same layer are fixed, the DNN's output always increases (inc) or decreases (dec) with increasing the neuron in question. However, in general, not all the hidden neurons can be classified into inc or dec. Therefore, we propose a novel preprocessing to transform the given DNN into an equivalent one so that all the hidden neurons can be classified into either inc or dec. To guarantee the soundness of the abstraction primitive Merge, it is only applied to two hidden neurons that are either both inc or both dec. The other abstraction primitive Freeze replaces a hidden neuron with its upper bound if it is inc or its lower bound if it is dec. An abstract DNN can be constructed by iteratively applying two abstraction primitives to proper hidden neurons.

While the application order of abstraction primitives does not matter for guaranteeing the soundness of abstraction, the following two issues should be addressed: (i) The Freeze abstraction primitive relies on the upper/lower bounds of neurons that may differ in the original DNN and an intermediate abstract DNN, whilst computing the upper/lower bounds of neurons for all the intermediate abstract DNNs is no doubt a considerable overhead. (ii) The final abstract DNN might become too coarse-grained (i.e., challenge C2) if the abstraction primitives are applied arbitrarily. To solve the first issue, we show that the upper/lower bounds of neurons in the original DNN can be re-used when applying the abstraction primitive Freeze to a neuron of an intermediate abstract DNN. Though the resulting abstract DNN does not necessarily over-approximate the intermediate abstract DNN where the Freeze abstraction primitive is applied, it still over-approximates the original DNN. Hence, the final abstract DNN over-approximates the original DNN when iteratively applying two abstraction primitives. To address challenge C2 (i.e., the second issue), the iterative application of abstraction primitives is guided by a novel strategy that selects an abstraction primitive and proper neurons aimed at minimizing the loss of accuracy during each iteration, thus can abstract out more neurons when sacrificing the same accuracy.

The refinement procedure is defined as a synergistic integration of two novel *refinement primitives*: Split for splitting a merged neuron into two neurons and Recover for recovering a removed neuron. In contrast to abstraction primitives whose application order does not matter for guaranteeing soundness, the application order of refinement primitives matters. For instance, there might be no corresponding incoming and/or outgoing edges for a neuron added by a refinement primitive. Thus, it becomes non-trivial to preserve the soundness of refinement. A simple method to solve this problem is to iteratively apply the corresponding refinement primitives in the reverse order of the applied abstraction primitives, independent of a given counterexample.

While this simple method ensures the soundness of refinement, it may fail to quickly exclude the counterexample. Consequently, a large number of refinement steps should be performed and the size of refined DNNs blows up quickly. Therefore, to address challenge C3, we propose a novel notion, called *dependency graph*, to characterize the dependency between refinement primitives, induced by the applied abstraction primitives. Under the restriction of dependency graphs, the refinement procedure is proved sound, otherwise may not be sound. Last but not least, the refinement procedure is also guided by a novel strategy to address challenge C4, which selects a refinement primitive and proper neurons to regain the most accuracy and quickly exclude the counterexample, thus can keep the network size smaller when restoring the same amount of accuracy.

Our approach is orthogonal to and can be integrated with many existing approaches. For evaluation, we implement our approach as a tool, NARv (**N**etwork **A**bstraction-**R**efinement for **v**erification), using two promising tools, Marabou [22] and Planet [11], as back-end verification engines, both of which are exact. The experimental results show that our approach can boost their performance by reducing up to 86.3% and 78.0% verification time, respectively. Moreover, our approach is 11.6–26.6 times faster than CEGAR-NN, the most relevant structure-oriented CEGAR-based approach for DNN verification [12].

To sum up, the main contributions of this work are as follows:

— We propose a *novel* abstraction procedure that synergistically integrates two abstraction primitives using a novel abstraction strategy, allowing to soundly and effectively reduce the network size when sacrificing the same accuracy.
— We propose a *novel* refinement procedure consisting of two refinement primitives, a notion of dependency graphs and a strategy for their synergistic integration, allowing to soundly refine the network and keep the network size as small as possible when restoring the same amount of accuracy.
— We implement our approach as a tool, NARv, with two promising DNN verification engines Marabou and Planet and conduct an extensive evaluation, demonstrating the efficacy of our approach.

Outline. Section 2 introduces preliminaries. Section 3 presents the overview of our approach. We propose our abstraction procedure and refinement procedure in Sections 4 and 5, respectively. Section 6 reports experimental results. Finally, we discuss related work in Section 7 and conclude the article in Section 8.

To foster further research, our tool, benchmarks, and experimental data are available at https://github.com/formes20/narv.

2 PRELIMINARIES

In this section, we present the background for formal verification of DNNs.

2.1 Deep Neural Networks

A ***fully connected feedforward deep neural network*** (***DNN***) with $\ell+1$ layers is a directed acyclic graph structured in layers, where the 0th layer is *input layer*, the ℓth layer is *output layer*, and the other layers are *hidden layers*. The nodes in each layer are *neurons*, in particular nodes in hidden layers are *hidden neurons*. We denote by $v_{i,j}$ the value of the jth neuron in the ith layer, and by $\boldsymbol{v}_i$ the output vector $(v_{i,1}, \ldots, v_{i,n})^T$ of the ith layer containing n neurons. Sometimes, $v_{i,j}$ also denotes the neuron itself. Each neuron $v_{i,j}$ in the ith layer for $1 \leq i \leq \ell$ is associated with a *bias*, denoted by $b(v_{i,j})$, and is connected by the weighted edges with weight $w(v_{i-1,k}, v_{i,j})$ from the neurons $v_{i-1,k}$ in the $(i-1)$th layer. A DNN computes the output of a given input by propagating it through the network, where the value of each neuron is calculated by applying an *activation function* to the weighted sum of the neuron values from the preceding layer.

Formally, a DNN N with $\ell + 1$ layers is a function $N(\boldsymbol{x})$ defined by

$$N(\boldsymbol{x}) = W_\ell \boldsymbol{v}_{\ell-1} + \boldsymbol{b}_\ell,$$

where $\boldsymbol{v}_0 = \boldsymbol{x}$, $\boldsymbol{v}_i = \sigma(W_i \boldsymbol{v}_{i-1} + \boldsymbol{b}_i)$ for $1 \leq i < \ell$, W_i and $\boldsymbol{b}_i$ are, respectively, the weight matrix and bias vector associated with the ith layer, and σ is an activation function applied in an element-wise manner. In this work, we assume that the activation function σ is *monotonic* (including non-strictly monotonic) and non-negative (i.e., $\sigma(x) \geq 0$ for any input x). This covers many common activation functions. For instance, all the following activation functions are monotonically increasing and non-negative [50]:

— the ReLU activation function $\mathsf{ReLU}(x) = \max(x, 0)$;
— the binary step activation function $\mathsf{BinStep}(x) = (x \geq 0\ ?\ 1 : 0)$;
— the sigmoid activation function $\mathsf{sigmoid}(x) = \frac{e^x}{e^x+1}$;
— the softplus activation function $\mathsf{softplus}(x) = \ln(1 + e^x)$.

For the sake of presentation, we focus on monotonically increasing activation functions, which are more common in the literature than monotonically decreasing activation functions. We remark that monotonically decreasing activation functions could be handled accordingly. On the other hand, non-positive activation functions σ (i.e., $\sigma(x) \leq 0$ for any x) can be treated as activation functions $-\sigma$ where the weights of the ith layers for $2 \leq i \leq \ell$ are negated.

2.2 Formal Verification of DNNs

Given a DNN N, a property P over the inputs $\boldsymbol{x}$ and a property Q over outputs $\boldsymbol{y} = N(\boldsymbol{x})$, a *verification problem* $\varphi = \langle N, P, Q \rangle$ is to determine whether any input $\boldsymbol{x}$ that fulfills P will result in an output $\boldsymbol{y} = N(\boldsymbol{x})$ that satisfies Q, where P forms the input space of interest and Q is the desired property of the DNN N w.r.t. the input space.

In this work, we assume that the output layer of the DNN N only contains a single neuron y, and the output property Q is of the form $y \leq c$ for a given constant c. We should emphasize that this assumption is required for our abstraction, but does not limit the applicability of our approach. Indeed, Elboher et al. [12] have already observed that many output properties Q of interest, including those with arbitrary Boolean structures and involving multiple neurons (e.g., all the properties of the ACAS Xu family of benchmarks [21], L_1 and L_∞ norms for adversarial robustness [4]), can be reduced into the form $y \leq c$ for a single-output DNN N' by adding typically a negligible number of neurons that encode the Boolean structure. We remark that the output property $y \leq c$ in this work represents the desired property of the DNN N while it is the negation of the desired property in Reference [12], i.e., $y > c$. Currently, we do not impose any restrictions on the input property P as our approach is general. In Section 6, we will discuss the restriction of the input property P in our implementation, that comes from the underlying verification engine and tool for computing lower and upper bounds of neurons.

Given a verification problem $\varphi = \langle N, P, Q \rangle$, a DNN $\bar{N}$ is an *over-approximation* of the DNN N if $N(\boldsymbol{x}) \leq \bar{N}(\boldsymbol{x})$ for every input $\boldsymbol{x}$ that fulfills the input property P. Note that $N(\boldsymbol{x}) \leq \bar{N}(\boldsymbol{x})$ implies that $\bar{N}(\boldsymbol{x}) \leq c \Rightarrow N(\boldsymbol{x}) \leq c$. An input $\boldsymbol{x}$ is a *counterexample* of the DNN N if the input $\boldsymbol{x}$ fulfills the input property P but the output property $N(\boldsymbol{x}) \leq c$ does not hold. A counterexample $\boldsymbol{x}$ of the over-approximation $\bar{N}$ is *spurious* on the DNN N if the output property $N(\boldsymbol{x}) \leq c$ holds.

3 OVERVIEW OF OUR APPROACH

Following the standard CEGAR paradigm [5], our CEGAR-based approach is described in Algorithm 1, which invokes two vital components: the abstraction procedure ABSTRACT and the

ALGORITHM 1: CEGAR-based Framework of Our Approach

```
Input: A verification problem $\langle N, P, Q\rangle$
Output: YES if the problem holds; otherwise a counterexample cex
1: Build an abstract DNN $\bar{N} \leftarrow$ ABSTRACT($N, P$)
2: while VERIFY($\langle \bar{N}, P, Q\rangle$) = NO do
3:     Extract a counterexample cex
4:     if cex is a counterexample of $\langle N, P, Q\rangle$ then
5:         return cex
6:     else $\bar{N} \leftarrow$ REFINE($\bar{N}$, cex)
7: return YES
```

refinement procedure Refine. Given a verification problem $\langle N, P, Q\rangle$, Algorithm 1 returns either YES indicating that the problem holds or a counterexample *cex* as the witness of the violation.

To solve a verification problem $\langle N, P, Q\rangle$, we first build an over-approximation $\bar{N}$ of the DNN N by invoking the procedure Abstract (line 1). The procedure Abstract first preprocesses the DNN N by transforming it into an equivalent DNN N' such that increasing the value of each single hidden neuron in the DNN N' either increases or decreases the network's output, thus establishing monotonicity between the value of each hidden neuron and the network's output. Then, we build the over-approximation $\bar{N}$ from the DNN N' by a synergistic integration of two novel abstraction primitives: Merge and Freeze, where Merge merges two neurons with the same monotonicity into a single one while Freeze deletes one neuron from the network. Both Merge and Freeze build an over-approximation of a given DNN, thus provide soundness guarantees (i.e., challenge C1). To address challenge C2, abstraction primitives are iteratively applied according to a novel strategy until a given accuracy threshold is reached. The strategy is designed to minimize the loss of accuracy during each iteration, thus reduces the network size as much as possible when sacrificing the same accuracy. To achieve this, we measure the loss of accuracy induced by applying abstraction primitives.

Next, we check if the verification problem $\langle \bar{N}, P, Q\rangle$ holds or not by invoking a sound and complete verification engine Verify (line 2). If the verification problem $\langle \bar{N}, P, Q\rangle$ holds, then we can conclude that the original problem $\langle N, P, Q\rangle$ holds as well, because the abstract DNN $\bar{N}$ over-approximates the original DNN N. Otherwise, a counterexample *cex* is extracted from the verification problem $\langle \bar{N}, P, Q\rangle$. If *cex* is a genuine counterexample of the original verification problem $\langle N, P, Q\rangle$, then *cex* is reported as the witness of the violation to the property Q (line 5). If the counterexample *cex* is a spurious counterexample of $\langle N, P, Q\rangle$, then the abstract DNN $\bar{N}$ is refined to exclude *cex* by invoking the procedure Refine (line 6).

The procedure Refine is also a synergistic integration of two novel refinement primitives: Split and Recover, where Split splits an abstract neuron into the original two neurons while Recover gets back a deleted neuron. However, applying Split or Recover without any restriction does not necessarily yield an over-approximation of the original DNN N (i.e., challenge C3). To address this issue, Split and Recover are applied according to a dependency graph, a novel notion proposed to characterize their dependency. To address challenge C4, the refinement primitives are iteratively applied according to a novel strategy until the spurious counterexample *cex* is excluded, where the strategy is designed to regain the most accuracy during each iteration, thus keeps the network size as small as possible when restoring the same amount of accuracy. To achieve this, we propose a profit function parameterized by the counterexample to measure the accuracy that can be restored via refinement primitives on different neurons.

In the following two sections, we elaborate the details of the abstraction procedure Abstract and the refinement procedure Refine.

4 NETWORK ABSTRACTION

In this section, we present our abstraction procedure Abstract. We first present the preprocessing of Abstract and then the abstraction primitives.

4.1 Preprocessing

Let us fix a DNN N with $\ell+1$ layers. To over-approximate the DNN N, all hidden neurons should be classified into either inc or dec, indicating their monotonic effect on the network's output. A neuron is inc if increasing its value, while keeping all the inputs unchanged, increases the network's output. Symmetrically, a neuron is dec if decreasing its value increases the network's output. To formalize inc and dec, for each i with $0 \leq i \leq \ell + 1$, we define the function $N^{(i)}$ as follows:

$$N^{(i)}(\boldsymbol{x}) = \begin{cases} \sigma(W_i \boldsymbol{x} + \boldsymbol{b}_i), & \text{if } 0 < i < \ell; \\ W_i \boldsymbol{x} + \boldsymbol{b}_i, & \text{if } i = \ell; \\ \boldsymbol{x}, & \text{if } i = 0 \text{ or } i = \ell + 1. \end{cases}$$

Intuitively, the function $N^{(i)}$ denotes the input-output relation of the ith layer of the DNN N. We remark that the DNN N with $\ell + 1$ layers does not have the $(\ell + 1)$th layer, and the identity function $N^{(\ell+1)}$ is introduced for a succinct formalization. We denote by $N^{(\geq i)}$ the composition of the functions $N^{(i)}, \ldots, N^{(\ell+1)}$, i.e., $N^{(\ell+1)} \circ \cdots \circ N^{(i)}$, and by $N^{(\leq i)}$ the composition of the functions $N^{(0)}, \ldots, N^{(i)}$, i.e., $N^{(i)} \circ \cdots \circ N^{(0)}$. Obviously, $N^{(\geq 0)} = N^{(\leq \ell+1)} = N$.

A neuron $v_{i,j}$ in the ith layer is inc (respectively, dec) if the function $N^{(\geq i+1)}$ is monotonically increasing (respectively, decreasing) w.r.t. the neuron $v_{i,j}$, namely, $N^{(\geq i+1)}(\boldsymbol{v}) \leq N^{(\geq i+1)}(\boldsymbol{v}')$ for any pair of inputs $\boldsymbol{v}$ and $\boldsymbol{v}'$ that agree on all the entries except that the jth entry of $\boldsymbol{v}$ is less (respectively, larger) than the jth entry of $\boldsymbol{v}'$. We denote by $s(v_{i,j}) = 1$ if $v_{i,j}$ is an inc neuron and $s(v_{i,j}) = -1$ if $v_{i,j}$ is a dec neuron.

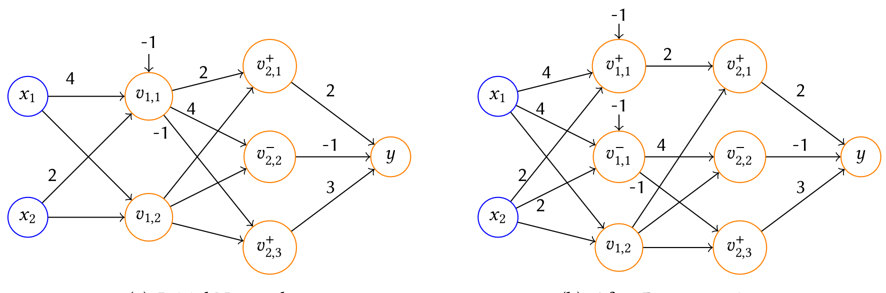
However, in general, not all hidden neurons of a given DNN can be simply classified into either inc or dec. To address this issue, we transform the given DNN N into an equivalent DNN N', namely, $N(\boldsymbol{x}) = N'(\boldsymbol{x})$ for any input $\boldsymbol{x}$, so that all hidden neurons of N' can be classified into either inc or dec.

Recall that we assume the given DNN N has a unique output neuron y. The output y in the ℓth layer is inc, since the function $N^{(\geq \ell+1)}(y) = N^{(\ell+1)}(y) = y$ is monotonically increasing w.r.t. y. The preprocessing procedure starts with setting the single output y as inc, and proceeds backwards layer by layer. Suppose all the neurons after the ith layer have been classified as either inc or dec. A hidden neuron $v_{i,j}$ in the ith layer is split into two new neurons $v_{i,j}^{+}$ and $v_{i,j}^{-}$ such that:

(1) the neurons $v_{i,j}^{+}$ and $v_{i,j}^{-}$ have the same weighted incoming edges and biases of the hidden neuron $v_{i,j}$, namely,
   - for every neuron $v_{i-1,k}$ in the $(i-1)$th layer,
   $$w(v_{i-1,k}, v_{i,j}^{+}) = w(v_{i-1,k}, v_{i,j}^{-}) = w(v_{i-1,k}, v_{i,j});$$
   - and $b(v_{i,j}^{+}) = b(v_{i,j}^{-}) = b(v_{i,j})$.

(2) the neuron $v_{i,j}^{+}$ preserves (i) positive-weighted outgoing edges of $v_{i,j}$ that connect to inc neurons in the $(i+1)$th layer and (ii) negative-weighted outgoing edges of $v_{i,j}$ that connect to dec neurons in the $(i+1)$th layer. That is, for every neuron $v_{i+1,k}$ in the $(i+1)$th layer of the DNN N',
$$w(v_{i,j}^{+}, v_{i+1,k}) = \begin{cases} w(v_{i,j}, v_{i+1,k}), & \text{if } w(v_{i,j}, v_{i+1,k}) \cdot s(v_{i+1,k}) > 0; \\ 0, & \text{otherwise.} \end{cases}$$

(3) the neuron $v_{i,j}^{-}$ preserves (i) positive-weighted outgoing edges of $v_{i,j}$ that connect to dec neurons in the $(i+1)$th layer and (ii) negative-weighted outgoing edges of $v_{i,j}$ that connect



(a) Initial Network (b) After Preprocessing $v_{1,1}$

Fig. 1. Example for preprocessing.

to inc neurons in the $(i+1)$th layer. That is, for every neuron $v_{i+1,k}$ in the $(i+1)$th layer of the DNN N',

$$w(v^-_{i,j}, v_{i+1,k}) = \begin{cases} w(v_{i,j}, v_{i+1,k}), & \text{if } w(v_{i,j}, v_{i+1,k}) \cdot s(v_{i+1,k}) < 0; \\ 0, & \text{otherwise.} \end{cases}$$

Note that the neuron $v^+_{i,j}$ (respectively, $v^-_{i,j}$) can be omitted if the weights $w(v^+_{i,j}, v_{i+1,k})$ (respectively, $w(v^-_{i,j}, v_{i+1,k})$) of all its outgoing edges are 0, i.e., $w(v_{i,j}, v_{i+1,k}) \cdot s(v_{i+1,k}) \leq 0$ (respectively, $w(v_{i,j}, v_{i+1,k}) \cdot s(v_{i+1,k}) \geq 0$) for any k. Indeed, the neuron $v^+_{i,j}$ (respectively, $v^-_{i,j}$) plays the same role as the neuron $v_{i,j}$ that can be classified into inc (respectively, dec) without splitting into two neurons.

Intuitively, when the neuron $v^+_{i,j}$ increases, all the inc (respectively, dec) neurons in the $(i+1)$th layer increase (respectively, decrease), since they connect to $v^+_{i,j}$ by positive-weighted (respectively, negative-weighted) edges. Both the increase of inc neurons and decrease of dec neurons in the $(i+1)$th layer will increase the network's output by definition. The case of $v^-_{i,j}$ follows similarly. Hence the new neuron $v^+_{i,j}$ is inc and the neuron $v^-_{i,j}$ is dec, namely, $s(v^+_{i,j}) = 1$ and $s(v^-_{i,j}) = -1$.

Example 4.1. Consider the DNN shown in Figure 1(a). Suppose the last two layers have been preprocessed. Consider neuron $v_{1,1}$ whose bias is -1 and the related weights are associated with edges. The DNN after preprocessing the neuron $v_{1,1}$ is depicted in Figure 1(b), where: $v_{1,1}$ is split into two new neurons $v^+_{1,1}$ and $v^-_{1,1}$ that have the same incoming edges and biases as $v_{1,1}$, the neuron $v^+_{1,1}$ keeps the outgoing edge of weight 2 pointing to the inc neuron $v^+_{2,1}$, the neuron $v^-_{1,1}$ keeps the outgoing edge of weight 4 pointing to the dec neuron $v^-_{2,2}$ and the outgoing edge of weight -1 pointing to the inc neuron $v^+_{2,3}$. The neurons $v^+_{1,1}$ and $v^-_{1,1}$ are now, respectively, inc and dec.

After the preprocessing of all the hidden neurons, we obtain a new DNN N' that is equivalent to the original DNN N, in which each hidden neuron is classified into either inc or dec. Thus, we have:

LEMMA 4.2. *Any (single-output) DNN N can be transformed into an equivalent DNN N' where each hidden neuron is classified into either* inc *or* dec*, by increasing the number of hidden neurons by a factor of at most 2.*

PROOF. Let $N_0, N_1, \ldots, N_n$ be the sequence of constructed DNNs during the preprocessing, where $N_0 = N$, $N' = N_n$, and for every k with $0 \leq k < n$, the DNN N_{k+1} is obtained from the DNN N_k by splitting one neuron. Let us apply induction on the index k. The base case $k = 0$ immediately follows from the facts that $N_0 = N$, $N^{(\geq \ell+1)}(\boldsymbol{x}) = \boldsymbol{x}$ and the output neuron y is set as inc. We show the inductive step $k \geq 1$.

Suppose the DNN N_k is obtained from the DNN N_{k-1} by splitting the hidden neuron $v_{i,j}$ in the ith layer into two new neurons $v^+_{i,j}$ and $v^-_{i,j}$. Note that $i < \ell$. By applying induction hypothesis, the

DNNs N and N_{k-1} are equivalent, and all the neurons of the form $v^+_{i',j'}$ (respectively, $v^-_{i',j'}$) are inc (respectively, dec) in the DNN N_{k-1}. It remains to show that the DNNs N_k and N_{k-1} are equivalent and all the neurons of the form $v^+_{i',j'}$ (respectively, $v^-_{i',j'}$) are inc (respectively, dec) in the DNN N_k for any i' and j'.

According to the construction of N_k, the functions $N_k^{(\leq i-1)}$ and $N_{k-1}^{(\leq i-1)}$ (respectively, $N_k^{(\geq i+2)}$ and $N_{k-1}^{(\geq i+2)}$) are equivalent. Thus, to prove that the DNNs N_k and N_{k-1} are equivalent, it suffices to show that $N_k^{(i+1)} \circ N_k^{(i)} = N_{k-1}^{(i+1)} \circ N_{k-1}^{(i)}$. Suppose in the DNN N_{k-1}, the $(i-1)$th layer has n_1 neurons, the ith layer has n_2 neurons and the $(i+1)$th layer has n_3 neurons.

Consider an input $\boldsymbol{x} = (x_1, \ldots, x_{n_1})^T$, let $\boldsymbol{y} = (y_1, \ldots, y_{n_2})^T$, where for every t with $1 \leq t \leq n_2$,

$$y_t = \sigma\left(\sum_{j'=1}^{n_1}(w(v_{i-1,j'}, v_{i,t}) \cdot x_{j'}) + b(v_{i,t})\right).$$

Then, we have $N_{k-1}^{(i)}(\boldsymbol{x}) = \boldsymbol{y}$. Since the new neurons $v^+_{i,j}$ and $v^-_{i,j}$ have the same weighted incoming edges and biases, we get that

$$N_k^{(i)}(\boldsymbol{x}) = (y_1, \ldots, y_{j-1}, y_j, y_j, y_{j+1}, \ldots, y_{n_2})^T.$$

Let $\boldsymbol{z} = (z_1, \ldots, z_{n_3})^T$, where for every t with $1 \leq t \leq n_3$,

$$z_t = \begin{cases} \sum_{j'=1}^{n_2}(w(v_{i,j'}, v_{i+1,t}) \cdot y_{j'}) + b(v_{i+1,t}), & \text{if } i+1 = \ell; \\ \sigma\big(\sum_{j'=1}^{n_2}(w(v_{i,j'}, v_{i+1,t}) \cdot y_{j'}) + b(v_{i+1,t})\big), & \text{if } i+1 < \ell. \end{cases}$$

Then, we have $N_{k-1}^{(i+1)} \circ N_{k-1}^{(i)}(\boldsymbol{x}) = \boldsymbol{z}$.

Suppose $N_k^{(i+1)}((y_1, \ldots, y_{j-1}, y_j, y_j, y_{j+1}, \ldots, y_{n_2})^T) = (z'_1, \ldots, z'_{n_3})^T$. Then, for every t with $1 \leq t \leq n_3$, we have

$$z'_t = \begin{cases} \left(\begin{array}{l} \sum_{j'=1}^{j-1}(w(v_{i,j'}, v_{i+1,t}) \cdot y_{j'}) + \sum_{j'=j+1}^{n_2}(w(v_{i,j'}, v_{i+1,t}) \cdot y_{j'}) + b(v_{i+1,t}) \\ + w(v^+_{i,j}, v_{i+1,t}) \cdot y_j + w(v^-_{i,j}, v_{i+1,t}) \cdot y_j \end{array}\right), & \text{if } i+1 = \ell; \\ \sigma\left(\begin{array}{l} \sum_{j'=1}^{j-1}(w(v_{i,j'}, v_{i+1,t}) \cdot y_{j'}) + \sum_{j'=j+1}^{n_2}(w(v_{i,j'}, v_{i+1,t}) \cdot y_{j'}) + b(v_{i+1,t}) \\ + w(v^+_{i,j}, v_{i+1,t}) \cdot y_j + w(v^-_{i,j}, v_{i+1,t}) \cdot y_j \end{array}\right), & \text{if } i+1 < \ell. \end{cases}$$

According to the definitions of $w(v^+_{i,j}, v_{i+1,t})$ and $w(v^-_{i,j}, v_{i+1,t})$, we have

$$w\left(v^+_{i,j}, v_{i+1,t}\right) \cdot y_j + w\left(v^-_{i,j}, v_{i+1,t}\right) \cdot y_j = w\left(v_{i,j}, v_{i+1,t}\right) \cdot y_j.$$

Therefore, $\boldsymbol{z} = (z_1, \ldots, z_{n_3})^T = (z'_1, \ldots, z'_{n_3})^T$, implying that $N_k^{(i+1)} \circ N_k^{(i)} = N_{k-1}^{(i+1)} \circ N_{k-1}^{(i)}$.

Now, we show that all the neurons of the form $v^+_{i',j'}$ (respectively, $v^-_{i',j'}$) are inc (respectively, dec) in the DNN N_k for any i' and j'. Since $N_k^{(\geq i+2)}$ and $N_{k-1}^{(\geq i+2)}$ are equivalent and all the neurons of the form $v^+_{i',j'}$ (respectively, $v^-_{i',j'}$) are inc (respectively, dec), i.e., $s(v^+_{i',j'}) = 1$ (respectively, $s(v^-_{i',j'}) = -1$) in the DNN N_{k-1} for any i' and j', we get that all the neurons of the form $v^+_{i',j'}$ (respectively, $v^-_{i',j'}$) are inc (respectively, dec) in the DNN N_k for any i' and j' such that $i' \neq i$ and $j' \neq j$. Since for every t with $1 \leq t \leq n_3$, $w(v^+_{i,j}, v_{i+1,t}) > 0$ iff $s(v_{i+1,t}) = 1$, $w(v^+_{i,j}, v_{i+1,t}) < 0$ iff $s(v_{i+1,t}) = -1$, (respectively, $w(v^-_{i,j}, v_{i+1,t}) > 0$ iff $s(v_{i+1,t}) = -1$, $w(v^+_{i,j}, v_{i+1,t}) < 0$ iff $s(v_{i+1,t}) = 1$), we get that the function $N_k^{(\geq i+1)} = N_k^{(\geq i+2)} \circ N_k^{(i+1)}$ is monotonically increasing (respectively, decreasing) w.r.t. the neuron $v^+_{i,j}$ (respectively, $v^-_{i,j}$). Therefore, all the neurons of the form $v^+_{i',j'}$ (respectively, $v^-_{i',j'}$) are inc (respectively, dec) in the DNN N_k for any i' and j'. □

Remark 1. Recall that we assumed the activation function σ in the DNN N is monotonically increasing. To preprocess DNNs with monotonically decreasing activation functions, it suffices to replace (i) the condition $w(v_{i,j}, v_{i+1,k}) \cdot s(v_{i+1,k}) > 0$ in Item (2) by $w(v_{i,j}, v_{i+1,k}) \cdot s(v_{i+1,k}) < 0$, and (ii) the condition $w(v_{i,j}, v_{i+1,k}) \cdot s(v_{i+1,k}) < 0$ in Item (3) by $w(v_{i,j}, v_{i+1,k}) \cdot s(v_{i+1,k}) > 0$.

By Lemma 4.2, we hereafter assume that each given DNN has been preprocessed and all its hidden neurons have been classified into either inc or dec.

4.2 Abstraction Primitives

As aforementioned, we propose two novel abstraction primitives: Merge and Freeze, to construct over-approximations of DNNs.

4.2.1 The Merge Primitive. The Merge primitive is to merge a pair of hidden neurons with same label inc/dec in the same layer into a single one. We seek to increase the values of inc neurons and decrease the values of dec neurons, ensuring that the network's output always increases. Suppose we are constructing an over-approximation $\bar{N}$ of N. Let $\bar{w}$ and $\bar{b}$ denote, respectively, the weights and biases in the constructed network $\bar{N}$. The Merge primitive merges two hidden inc neurons $v_{i,j}$ and $v_{i,k}$ into a new inc neuron $v_{i,t}$ via the following steps:

(1) all the weighted edges of the neurons $v_{i,j}$ and $v_{i,k}$ are removed;
(2) both neurons $v_{i,j}$ and $v_{i,k}$ are replaced by a new neuron $v_{i,t}$;
(3) from each neuron $v_{i-1,p}$ in the preceding layer, a weighted incoming edge to $v_{i,t}$ is added as
$$\bar{w}(v_{i-1,p}, v_{i,t}) = \max\{w(v_{i-1,p}, v_{i,j}),\ w(v_{i-1,p}, v_{i,k})\};$$
(4) to each neuron $v_{i+1,q}$ in the succeeding layer, a weighted outgoing edge from $v_{i,t}$ is added as
$$\bar{w}(v_{i,t}, v_{i+1,q}) = w(v_{i,j}, v_{i+1,q}) + w(v_{i,k}, v_{i+1,q});$$
(5) the bias of $v_{i,t}$ is $\bar{b}(v_{i,t}) = \max\{b(v_{i,j}),\ b(v_{i,k})\}$.

Intuitively, the max operation in steps (3) and (5) guarantees that $v_{i,t}$ is no less than the original neurons $v_{i,j}$ and $v_{i,k}$. By the definition of the weighted outgoing edges, it amounts to increasing or unchanging $v_{i,j}$ and $v_{i,k}$ in the DNN N. Since both neurons $v_{i,j}$ and $v_{i,k}$ are inc, it is ensured that the output is either unchanged or increased by Merge. The Merge primitive for two dec neurons in the same layer is defined similarly except that the max operation is replaced by the min operation in steps (3) and (5).

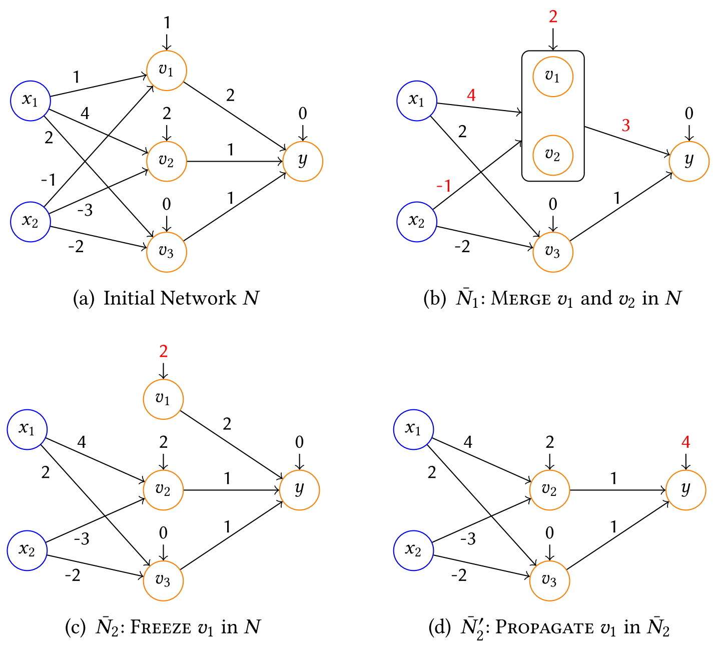
Hereafter, a neuron produced by the Merge primitive is called an *abstract neuron*, otherwise an *atomic neuron*.

Example 4.3. Consider the inc neurons v_1 and v_2 of the DNN N shown in Figure 2(a). After merging the neurons v_1 and v_2 by the Merge primitive, we obtain the DNN $\bar{N}_1$ shown in Figure 2(b), where for the abstract neuron of $\{v_1, v_2\}$, the weight of its incoming edge from the neuron x_1 is $4 = \max\{1, 4\}$, its bias is $2 = \max\{1, 2\}$, and its outgoing edge to the neuron y is weighted $3 = 2 + 1$. Given an input $\boldsymbol{x_0} = (1, 1)^T$, we have $\bar{N}_1(\boldsymbol{x_0}) = 15 > 5 = N(\boldsymbol{x_0})$.

The following lemma justifies the soundness of Merge.

Lemma 4.4. *Let $\bar{N}$ be the abstract DNN constructed from the DNN N by one single application of Merge. It holds that $\bar{N}(\boldsymbol{x}) \geq N(\boldsymbol{x})$ for each input $\boldsymbol{x}$.*

Proof. We assume that the DNN $\bar{N}$ is obtained from the DNN N by merging two hidden neurons $v_{i,j}$ and $v_{i,k}$ into a new neuron $v_{i,t}$. Note that $i < \ell$. W.l.o.g., we assume $j < k$. To show that $\bar{N}(\boldsymbol{x}) \geq N(\boldsymbol{x})$ for each input $\boldsymbol{x}$, it suffices to prove that for each possible input $\boldsymbol{x}$ of the ith layer, for any neuron $v_{i+1,h}$ in the $(i + 1)$th layer:



(a) Initial Network N

(b) $\bar{N}_1$: Merge v_1 and v_2 in N

(c) $\bar{N}_2$: Freeze v_1 in N

(d) $\bar{N}_2'$: Propagate v_1 in $\bar{N}_2$

Fig. 2. Example for abstraction primitives.

- the hth entry of $\bar{N}^{(i+1)} \circ \bar{N}^{(i)}(\boldsymbol{x})$ is no less than the hth entry of $N^{(i+1)} \circ N^{(i)}(\boldsymbol{x})$ if the neuron $v_{i+1,h}$ is inc;
- the hth entry of $\bar{N}^{(i+1)} \circ \bar{N}^{(i)}(\boldsymbol{x})$ is no more than the hth entry of $N^{(i+1)} \circ N^{(i)}(\boldsymbol{x})$ if the neuron $v_{i+1,h}$ is dec.

Suppose in the DNN N, the $(i-1)$th layer has n_1 neurons, the ith layer has n_2 neurons and the $(i+1)$th layer has n_3 neurons. Consider an input $\boldsymbol{x} = (x_1, \ldots, x_{n_1})^T$ of the ith layer. Note that $x_{j'} \geq 0$ for any j' with $1 \leq j' \leq n_1$, as the activation function σ is non-negative. Let $\boldsymbol{y} = (y_1, \ldots, y_{n_2})^T$, where for every h with $1 \leq h \leq n_2$, $y_h = \sigma\big(\sum_{j'=1}^{n_1}(w(v_{i-1,j'}, v_{i,h}) \cdot x_{j'}) + b(v_{i,h})\big)$. Then, we have $N^{(i)}(\boldsymbol{x}) = \boldsymbol{y}$.

By the definition of $v_{i,t}$, we have $\bar{N}^{(i)}(\boldsymbol{x}) = (y_1, \ldots, y_{j-1}, y_t, y_{j+1}, \ldots, y_{k-1}, y_{k+1}, \ldots, y_{n_2})^T$, where

$$y_t = \begin{cases} \sigma\left(\sum_{j'=1}^{n_1}(\max\{w(v_{i-1,j'}, v_{i,j}), w(v_{i-1,j'}, v_{i,k})\} \cdot x_{j'}) + \max\{b(v_{i,j}),\ b(v_{i,k})\}\right), & \text{if } v_{i,j}, v_{i,k} \text{ are inc;} \\ \sigma\left(\sum_{j'=1}^{n_1}(\min\{w(v_{i-1,j'}, v_{i,j}), w(v_{i-1,j'}, v_{i,k})\} \cdot x_{j'}) + \min\{b(v_{i,j}),\ b(v_{i,k})\}\right), & \text{if } v_{i,j}, v_{i,k} \text{ are dec.} \end{cases}$$

Let $\boldsymbol{z} = (z_1, \ldots, z_{n_3})^T$, where for every h with $1 \leq h \leq n_3$,

$$z_h = \begin{cases} \sum_{j'=1}^{n_2}(w(v_{i,j'}, v_{i+1,h}) \cdot y_{j'}) + b(v_{i+1,h}), & \text{if } i+1 = \ell; \\ \sigma\left(\sum_{j'=1}^{n_2}(w(v_{i,j'}, v_{i+1,h}) \cdot y_{j'}) + b(v_{i+1,h})\right), & \text{if } i+1 < \ell. \end{cases}$$

Then, we have $N^{(i+1)} \circ N^{(i)}(\boldsymbol{x}) = \boldsymbol{z}$.

Suppose $\bar{N}^{(i+1)}((y_1, \ldots, y_{j-1}, y_t, y_{j+1}, \ldots, y_{k-1}, y_{k+1}, \ldots, y_{n_2})^T) = (z_1', \ldots, z_{n_3}')^T$. For every h with $1 \leq h \leq n_3$, let

$$z_h'' = \left(\begin{array}{c} \sum_{j'=1}^{j-1}(w(v_{i,j'}, v_{i+1,h}) \cdot y_{j'}) + \sum_{j'=j+1}^{k-1}(w(v_{i,j'}, v_{i+1,h}) \cdot y_{j'}) + \sum_{j'=k+1}^{n_2}(w(v_{i,j'}, v_{i+1,h}) \cdot y_{j'}) \\ + (w(v_{i,j}, v_{i+1,h}) + w(v_{i,k}, v_{i+1,h})) \cdot y_t + b(v_{i+1,h}) \end{array} \right).$$

Then,

$$z'_h = \begin{cases} z''_h, & \text{if } i+1 = \ell; \\ \sigma\left(z''_h\right), & \text{if } i+1 < \ell. \end{cases}$$

For every h with $1 \leq h \leq n_3$, we have

$$z''_h - \sum_{j'=1}^{n_2}(w(v_{i,j'}, v_{i+1,h}) \cdot y_{j'}) + b(v_{i+1,h}) = w(v_{i,j}, v_{i+1,h}) \cdot (y_t - y_j) + w(v_{i,k}, v_{i+1,h}) \cdot (y_t - y_k).$$

We distinguish whether $v_{i,j}$ and $v_{i,k}$ are both inc or both dec.

- If $v_{i,j}$ and $v_{i,k}$ are inc, then for every h with $1 \leq h \leq n_3$, $w(v_{i,j}, v_{i+1,h}) \cdot s(v_{i+1,h}) > 0$, $w(v_{i,k}, v_{i+1,h}) \cdot s(v_{i+1,h}) > 0$, $y_t \geq y_j$ and $y_t \geq y_k$.
  - If $s(v_{i+1,h}) = 1$, i.e., $v_{i+1,h}$ is inc, then $w(v_{i,j}, v_{i+1,h}) > 0$ and $w(v_{i,k}, v_{i+1,h}) > 0$. Together with $y_t \geq y_j$ and $y_t \geq y_k$, we get that $z''_h \geq \sum_{j'=1}^{n_2}(w(v_{i,j'}, v_{i+1,h}) \cdot y_{j'}) + b(v_{i+1,h})$. Since the activation function σ is monotonically increasing, we have $z'_h \geq z_h$.
  - If $s(v_{i+1,h}) = -1$, i.e., $v_{i+1,h}$ is dec, then $w(v_{i,j}, v_{i+1,h}) < 0$ and $w(v_{i,k}, v_{i+1,h}) < 0$. Together with $y_t \geq y_j$ and $y_t \geq y_k$, we get that $z''_h \leq \sum_{j'=1}^{n_2}(w(v_{i,j'}, v_{i+1,h}) \cdot y_{j'}) + b(v_{i+1,h})$. Since the activation function σ is monotonically increasing, we have $z'_h \leq z_h$.
- If $v_{i,j}$ and $v_{i,k}$ are dec, then for every h with $1 \leq h \leq n_3$, $w(v_{i,j}, v_{i+1,h}) \cdot s(v_{i+1,h}) < 0$, $w(v_{i,k}, v_{i+1,h}) \cdot s(v_{i+1,h}) < 0$, $y_t \leq y_j$ and $y_t \leq y_k$.
  - If $s(v_{i+1,h}) = 1$, i.e., $v_{i+1,h}$ is inc, then $w(v_{i,j}, v_{i+1,h}) < 0$ and $w(v_{i,k}, v_{i+1,h}) < 0$. Together with $y_t \leq y_j$ and $y_t \leq y_k$, we get that $z''_h \geq \sum_{j'=1}^{n_2}(w(v_{i,j'}, v_{i+1,h}) \cdot y_{j'}) + b(v_{i+1,h})$. Since the activation function σ is monotonically increasing, we have $z'_h \geq z_h$.
  - If $s(v_{i+1,h}) = -1$, i.e., $v_{i+1,h}$ is dec, then $w(v_{i,j}, v_{i+1,h}) > 0$ and $w(v_{i,k}, v_{i+1,h}) > 0$. Together with $y_t \leq y_j$ and $y_t \leq y_k$, we get that $z''_h \leq \sum_{j'=1}^{n_2}(w(v_{i,j'}, v_{i+1,h}) \cdot y_{j'}) + b(v_{i+1,h})$. Since the activation function σ is monotonically increasing, we have $z'_h \leq z_h$.

We conclude the proof. □

4.2.2 *The Freeze Primitive.*

One application of Merge reduces the network size (i.e., the number of neurons) by 1, but may reduce the network accuracy as well. The reduced accuracy sometimes can be considerable, for instance, when the two weights in the max operation have a big difference as in Example 4.3. To alleviate this issue, we introduce another abstraction primitive, named Freeze, to freeze hidden neurons using constants.

Consider a hidden neuron $v_{i,j}$ and a constant a. If a is an upper bound of the neuron $v_{i,j}$, then freezing the neuron $v_{i,j}$ by a amounts to increasing or keeping $v_{i,j}$. Thus, the network's output is guaranteed to non-decrease when $v_{i,j}$ is inc. Similarly, if a is a lower bound of the dec neuron $v_{i,j}$, then the network's output is also guaranteed to non-decrease by applying Freeze.

Formally, let $\text{ub}^N(v_{i,j})$ and $\text{lb}^N(v_{i,j})$, respectively, denote an upper bound and a lower bound of the neuron $v_{i,j}$ in the DNN N, that is, $\text{lb}^N(v_{i,j}) \leq v_{i,j} \leq \text{ub}^N(v_{i,j})$ for any input that fulfills the input property P. The Freeze abstraction primitive builds an over-approximation $\bar{N}$ of the DNN N as follows: for a hidden neuron $v_{i,j}$,

(1) all weighted incoming edges to the neuron $v_{i,j}$ are removed, i.e.,

$$\bar{w}(v_{i-1,p}, v_{i,j}) = 0, \text{ for each neuron } v_{i-1,p} \text{ in the } (i-1)\text{th layer;}$$

(2) the value of the neuron $v_{i,j}$ is replaced by a constant as

$$\bar{b}(v_{i,j}) = \begin{cases} \text{ub}^N(v_{i,j}), & \text{if } v_{i,j} \text{ is inc;} \\ \text{lb}^N(v_{i,j}), & \text{if } v_{i,j} \text{ is dec.} \end{cases}$$

Intuitively, the abstraction primitive Freeze produces neurons whose all incoming edges are weighted by 0 and their values are indeed their biases $\bar{b}(v_{i,j})$.

Hereafter, neurons introduced by applying the primitive Freeze are called *constant neurons.* Note that the required upper/lower bounds of neurons can be computed efficiently by techniques such as References [41, 49].

Example 4.5. Consider the DNN shown in Figure 2(a). Assume that the range of the inc neuron v_1 is $[0, 2]$. The abstract DNN $\bar{N}_2$ constructed from the DNN N by applying the abstraction primitive Freeze on the neuron v_1 is shown in Figure 2(c). All the incoming edges to the neuron v_1 are removed. Its bias is replaced with its upper bound 2. Given the input $\boldsymbol{x_0} = (1,1)^T$, $\bar{N}_2(\boldsymbol{x_0}) = 7 < 15 = \bar{N}_1(\boldsymbol{x_0})$ shows that the abstraction primitive Freeze can be more accurate than the abstraction primitive Merge in some situations.

The following lemma provides the soundness of Freeze.

Lemma 4.6. *Let $\bar{N}$ be the abstract DNN constructed from the DNN N of a verification problem $\langle N, P, Q\rangle$ by a single application of Freeze. It holds that $\bar{N}(\boldsymbol{x}) \geq N(\boldsymbol{x})$ for each input $\boldsymbol{x}$ that fulfills the input property P.*

Proof. We assume that the abstract DNN $\bar{N}$ is obtained from the DNN N by freezing the hidden neuron $v_{i,j}$. Note that $i < \ell$. Suppose the $(i-1)$th layer in the DNN N has n_1 neurons. For each possible input $\boldsymbol{x}$ of the ith layer, the values of all the hidden neurons $v_{i,j'}$ such that $j \neq j'$ are the same in $\bar{N}^{(i)}(\boldsymbol{x})$ and $N^{(i)}(\boldsymbol{x})$. Thus, to show that $\bar{N}(\boldsymbol{x}) \geq N(\boldsymbol{x})$ for each input $\boldsymbol{x}$ that fulfills the input property P, it suffices to prove that for each possible input $\boldsymbol{x}$ of the ith layer,

- the jth entry of $\bar{N}^{(i)}(\boldsymbol{x})$ is no less than the jth entry of $N^{(i)}(\boldsymbol{x})$ if the neuron $v_{i,j}$ is inc,
- the jth entry of $\bar{N}^{(i)}(\boldsymbol{x})$ is no more than the jth entry of $N^{(i)}(\boldsymbol{x})$ if the neuron $v_{i,j}$ is dec.

The jth entry of $N^{(i)}(\boldsymbol{x})$ is $\sigma\big(\sum_{j'=1}^{n_1}(w(v_{i-1,j'}, v_{i,j}) \cdot x_{j'}) + b(v_{i,j})\big)$, and the jth entry of $\bar{N}^{(i)}(\boldsymbol{x})$ is $\mathrm{ub}^N(v_{i,j})$ (respectively, $\mathrm{lb}^N(v_{i,j})$) if $v_{i,j}$ is inc (respectively, dec). The result immediately follows from the fact that $\mathrm{lb}^N(v_{i,j}) \leq \sigma\big(\sum_{j'=1}^{n_1}(w(v_{i-1,j'}, v_{i,j}) \cdot x_{j'}) + b(v_{i,j})\big) \leq \mathrm{ub}^N(v_{i,j})$. □

One may notice that the application of the abstraction primitive Freeze does not reduce the network size. To further eliminate constant neurons created by Freeze, we propose a new procedure Propagate. Consider a constant neuron $v_{i,j}$ in the DNN N, Propagate works as follows:

(1) to propagate the value of the neuron $v_{i,j}$ to the succeeding layer, for each neuron $v_{i+1,q}$ in the $(i+1)$th layer, we set:
$$b'(v_{i+1,q}) = b(v_{i+1,q}) + w(v_{i,j}, v_{i+1,q}) \cdot b(v_{i,j});$$
(2) the constant neuron $v_{i,j}$ and its outgoing edges are removed.

Example 4.7. Consider the constant neuron v_1 in the DNN $\bar{N}_2$ shown in Figure 2(c). By applying Propagate to the neuron v_1, v_1 is removed and its value 2 is propagated to the neuron y, resulting in the DNN shown in Figure 2(d).

By definition, the obtained DNN N' after Propagate is equivalent to the DNN N. Thus, it is trivial to get that:

Lemma 4.8. *Let N' be the DNN constructed from the DNN N by a single application of Propagate on a constant neuron. It holds that $N'(\boldsymbol{x}) = N(\boldsymbol{x})$ for each input $\boldsymbol{x}$.*

Freeze and Propagate can cooperate to delete hidden neurons, hence reducing network size. We design Propagate as an individual procedure instead of merging with Freeze due to the following reasons: (i) to improve abstraction efficiency, Propagate is invoked only once before invoking the verification engine Verify; (ii) to keep Freeze *local*, i.e., without affecting the succeeding layer, as locality makes abstraction steps less dependent on each other, thus allows us to extend abstraction primitives further in Section 4.3.

Following Lemmas 4.4, 4.6, and 4.8, we conclude that our two abstraction primitives Merge and Freeze (followed by Propagate) do construct over-approximations.

Corollary 4.9. *Let $\bar{N}$ be the DNN obtained from the DNN N of a verification problem $\langle N, P, Q\rangle$ by first iteratively applying abstraction primitives* Merge *and/or* Freeze*, finally invoking* Propagate *to eliminate all constant neurons. It holds that $\bar{N}(\boldsymbol{x}) \geq N(\boldsymbol{x})$ for each input $\boldsymbol{x}$ that fulfills the input property P.*

Given a DNN N to verify, after iteratively applying our abstraction primitives, we get an abstract DNN $\bar{N}$. Corollary 4.9 ensures that $\bar{N}$ is an over-approximation of N, namely, if the verification problem $\langle \bar{N}, P, Q\rangle$ holds, then the verification problem $\langle N, P, Q\rangle$ holds as well. Therefore, we solve challenge C1.

4.3 Generalizing the Freeze Abstraction Primitive

Starting from a DNN $N = N_0$, iterating the application of abstraction primitives derives a sequence of DNNs $N_1, N_2, \ldots, N_k$. Corollary 4.9 guarantees that the abstract DNN N_j is an over-approximation of the DNN N_i for any $0 \leq i < j \leq k$. However, when applying the Freeze primitive to some hidden neuron of a DNN N_i, its upper/lower bounds w.r.t. the current DNN N_i is required by definition, but the upper/lower bounds in N_i may be different from those in other DNNs N_j for $j \neq i$ due to abstraction. To apply the primitive Freeze, a naive approach is to calculate the upper/lower bounds each time before applying Freeze, which is no doubt a considerable overhead. To mitigate this issue, we generalize the abstraction primitive Freeze based on the following observation.

Observe that an abstraction primitive applied to some hidden neurons in the ith layer does not change the values of the other neurons in the same layer and all the neurons in the jth layer if $j < i$, thus their upper/lower bounds are preserved. A more efficient way is to calculate the bounds only once in the original DNN N, and all abstraction primitives are applied backwards layer by layer. That is, any abstraction primitive to neurons in the ith layer must be applied *before* any application of abstraction primitives to the neurons in the jth layer for $j < i$. This constrained order enables all the applications of the abstraction primitive Freeze to make use of the upper/lower bounds calculated in the original DNN N, thus avoids the considerable overhead.

Based on the above observation, we introduce a generalized abstraction primitive, called Freeze_+^M, to freeze a hidden neuron w.r.t. a given DNN M. We emphasize that Freeze_+^M is parameterized by the given DNN M. To construct an abstract DNN $\bar{N}$ from the DNN N by applying Freeze_+^M on a hidden neuron $v_{i,j}$, the primitive Freeze_+^M works almost the same as Freeze except that Freeze_+^M leverages upper/lower bounds w.r.t. the given DNN M instead of the DNN N. More precisely, Freeze_+^M works as follows:

(1) all weighted incoming edges to the neuron $v_{i,j}$ are removed, i.e.,

$$\bar{w}(v_{i-1,p}, v_{i,j}) = 0, \text{ for each neuron } v_{i-1,p} \text{ in the } (i-1)\text{th layer;}$$

(2) the value of the neuron $v_{i,j}$ is replaced by a constant as

$$\bar{b}(v_{i,j}) = \begin{cases} \text{ub}^M(v_{i,j}), & \text{if } v_{i,j} \text{ is inc;} \\ \text{lb}^M(v_{i,j}), & \text{if } v_{i,j} \text{ is dec.} \end{cases}$$

Compared to the primitive Freeze, the bounds $\text{ub}^M(v_{i,j})$ and $\text{lb}^M(v_{i,j})$ of the neuron $v_{i,j}$ in the DNN M are used instead of the bounds $\text{ub}^N(v_{i,j})$ and $\text{lb}^N(v_{i,j})$ of the neuron $v_{i,j}$ in the DNN N.

However, in general, the primitive Freeze$_{+}^{M}$ does not provide any soundness guarantees like Lemma 4.6 for the abstraction primitive Freeze, namely, the obtained DNN $\bar{N}$ does not necessarily over-approximate the DNN N after applying the primitive Freeze$_{+}^{M}$. Nevertheless, Lemma 4.6 can be generalized to Freeze$_{+}^{M}$ if the upper bound $\mathrm{ub}^{M}(v_{i,j})$ (respectively, lower bound $\mathrm{lb}^{M}(v_{i,j})$) of the neuron $v_{i,j}$ in the DNN M is also an upper (respectively, lower) bound of the neuron $v_{i,j}$ in the DNN N. Thus, we have:

Lemma 4.10. *Given a verification problem $\langle N, P, Q\rangle$ and a DNN M, let $\bar{N}$ be the abstract DNN constructed from the DNN N by applying the primitive Freeze$_{+}^{M}$ to a hidden neuron v. $\bar{N}(\boldsymbol{x}) \geq N(\boldsymbol{x})$ holds for each input $\boldsymbol{x}$ that fulfills the input property P, if the upper bound $\mathrm{ub}^{M}(v)$ (respectively, lower bound $\mathrm{lb}^{M}(v)$) of the neuron v in the DNN M is also an upper bound (respectively, lower bound) of the neuron v in the DNN N.*

Proof. Suppose the upper bound $\mathrm{ub}^{M}(v)$ (respectively, lower bound $\mathrm{lb}^{M}(v)$) used in Freeze$_{+}^{M}$ is also an upper (respectively, lower) bound of v in N. Then the application of Freeze$_{+}^{M}$ on the neuron v in the DNN N conforms to the application of Freeze on the neuron v in the DNN N. The result immediately follows by applying Lemma 4.6. □

Given a DNN $N = N_0$, taking $M = N$, a sequence of DNNs $N_1, N_2, \ldots, N_k$ can be constructed by iteratively applying the primitives Merge and Freeze$_{+}^{N}$. If all the primitives are applied backwards layer by layer, then each time a neuron v in the DNN N_i is frozen by Freeze$_{+}^{N}$ (i.e., Freeze$_{+}^{M}$), the upper bound $\mathrm{ub}^{M}(v)$ (respectively, lower bound $\mathrm{lb}^{M}(v)$) of the neuron v in the DNN M is also an upper bound (respectively, lower bound) of the neuron v in the DNN N_i if it exists. Consequently, by Lemma 4.10, the abstract DNN N_j must be an over-approximation of the DNN N_i for any $0 \leq i \leq j \leq k$. Hence, the soundness of iteratively applying the abstraction primitives Merge and Freeze$_{+}^{N}$ on the given DNN N is guaranteed if the primitives are applied backwards layer by layer. We remark that the Freeze$_{+}^{N}$ primitive cannot be applied to abstract neurons that are created by Merge, as their upper/lower bounds are not available in the original DNN N.

Nonetheless, the constrained application order of abstraction primitives (i.e., applying abstraction primitives backwards layer by layer) may be too restricted to achieve a tight abstraction. To avoid this problem, we show that the application order of the abstraction primitives Merge and Freeze$_{+}^{N}$ on the given DNN N can be relaxed, while guaranteeing that the obtained abstract DNNs are over-approximations of the original DNN N. Specifically, we show that for a sequence of DNNs $N_1, N_2, \ldots, N_k$ constructed by iteratively applying Merge and Freeze$_{+}^{N}$ on the given DNN $N = N_0$ in any fixed order, although the DNN N_i for $i > 0$ may not be an over-approximation of the DNN N_j with $0 < j < i$, the DNN N_i must be an over-approximation of the original DNN N, which is sufficient to solve challenge C1.

To formalize the above property, we refer to an instance of applying the Merge primitive to two hidden neurons as a Merge step. Similarly, an instance of applying the Freeze$_{+}^{N}$ primitive to a hidden neuron is called a Freeze$_{+}^{N}$ step.

Lemma 4.11. *Given a verification problem $\langle N, P, Q\rangle$, let $N_1, N_2, \ldots, N_k$ be the sequence of abstract DNNs constructed by a series of Merge and Freeze$_{+}^{N}$ steps on the DNN $N = N_0$ in any fixed order. For any $i \geq 0$, $N_i(\boldsymbol{x}) \geq N(\boldsymbol{x})$ holds for each input $\boldsymbol{x}$ that fulfills the input property P.*

To prove Lemma 4.11, we need the following property for abstraction primitives Merge and Freeze$_{+}^{N}$.

Proposition 4.12. *Given a verification problem $\langle N, P, Q\rangle$, a DNN N_i, and a length-2 sequence $\tau = \alpha_1\alpha_2$ consisting of Merge and/or Freeze$_{+}^{N}$ steps. Assume that applying τ on the DNN N_i yields* *a DNN N_i'. If α_2 is a* Freeze$_+^N$ *step, then a DNN N_i'' can be constructed by applying the sequence $\tau' = \alpha_2\alpha_1$ on N_i. Moreover, N_i'' is identical to N_i'.*

Proof. Thanks to the locality of the primitive Freeze$_+^N$, it is straightforward to prove that N_i'' is identical to N_i' by case analysis on:

- α_1 is a Merge step or a Freeze$_+^N$ step;
- α_1 and α_2 are applied in the same layer, in the adjacent layers, or otherwise.

Note that the case, where α_1 is a Merge step producing an abstract neuron v and α_2 is a Freeze$_+^N$ step applied to v, should be excluded, since Freeze$_+^N$ is forbidden to be applied to abstract neurons. Then the result follows. □

Given a sequence of DNNs $N_0, N_1, \ldots, N_k$ produced by iteratively applying a sequence $\tau = \beta_1\beta_2 \ldots \beta_k$ of Merge and/or Freeze$_+^N$ steps on $N = N_0$. Proposition 4.12 can be leveraged to permute the steps in τ, yielding a new sequence τ', while guaranteeing that applying τ' on the DNN N yields the same final DNN N_k. Formally, Proposition 4.12 provides the *(string) rewrite rule* [9]

$$\alpha_1\alpha_2 \longrightarrow \alpha_2\alpha_1, \quad \text{if } \alpha_2 \text{ is a Freeze}_+^N \text{ step,}$$

to rewrite a sequence of Merge and/or Freeze$_+^N$ steps.

Now we are ready for proving Lemma 4.11.

Proof of Lemma 4.11. Assume that $N_1, N_2 \ldots, N_k$ is the sequence of abstract DNNs constructed by iteratively applying the sequence $\tau = \beta_1\beta_2 \cdots \beta_k$ of Merge and/or Freeze$_+^N$ steps on the DNN $N = N_0$. Let us apply induction on the length k.

The base case $k = 0$ is trivial. When $k > 0$, we first locate the Merge step in τ with the largest index j, i.e., the latest Merge step. There are several cases:

- Such a Merge step does not exist, that is, τ contains only Freeze$_+^N$ steps. Then τ can be freely permuted by Proposition 4.12, obtaining a new sequence τ' of Freeze$_+^N$ steps where the steps happen backwards layer by layer. Applying τ' on the DNN N still produces the DNN N_k by Proposition 4.12 and N_k is an over-approximation of the DNN N by Lemma 4.10.
- $j = k$. The DNN N_{k-1} is an over-approximation of the DNN N by induction hypothesis, and the DNN N_k is an over-approximation of the DNN N_{k-1} by Lemma 4.4. The result immediately follows.
- $j < k$. Since $\beta_{j+1}, \beta_{j+2}, \ldots, \beta_k$ are all Freeze$_+^N$ steps by the choice of j, by Proposition 4.12, the β_j step can be moved to the end of the sequence τ, resulting a new sequence $\tau' = \beta_1 \cdots \beta_{j-1}\beta_{j+1} \cdots \beta_k\beta_j$. Assume that applying τ' on the DNN N produces a sequence of DNNs $N_1', N_2', \ldots, N_k'$. The DNN N_k' is identical to the DNN N_k by Proposition 4.12. The DNN N_{k-1}' is an over-approximation of the DNN N by induction hypothesis, and the DNN N_k' is an over-approximation of the DNN N_{k-1}' by Lemma 4.4.

We conclude our proof. □

From Lemma 4.11, it follows directly that:

Theorem 4.13. *Let $\bar{N}$ be the abstract DNN constructed from the DNN N of a verification problem $\langle N, P, Q\rangle$ by a series of* Merge *and* Freeze$_+^N$ *steps. It holds that $\bar{N}(\boldsymbol{x}) \geq N(\boldsymbol{x})$ for each input $\boldsymbol{x}$ that fulfills the input property P.*

For a given verification problem $\langle N, P, Q\rangle$, Theorem 4.13 guarantees that we only need to calculate the upper/lower bounds of hidden neurons *once* on the original DNN N and the abstraction procedure Abstract can iteratively apply the abstraction primitives Merge and Freeze_+^N to hidden neurons *in any order* to construct over-approximations of the original DNN N without loss of soundness.

In the rest of this article, since we always use the original DNN N of a verification problem $\langle N, P, Q\rangle$ for the generalized abstraction primitive Freeze_+^N and the calculation of upper/lower bounds, for the sake of simplicity, the superscript N will be dropped from Freeze_+^N when it is clear from the context. When referring to *abstraction steps/primitives*, we now mean Merge and Freeze_+.

4.4 Abstraction Strategy

Having only abstraction primitives is not enough to accomplish a good abstraction procedure due to challenge C2, i.e., reducing the network size as much as possible while preserving accuracy as much as possible. To address this challenge, two questions have to be answered:

(1) *what should be done for a single abstraction step?*
(2) *how many abstraction steps should be performed?*

The objective for the first question is to introduce less inaccuracy at each abstraction step, thus admitting more abstraction steps and reducing the size more, when sacrificing the same accuracy. For this purpose, we propose to first locate less important neurons and then abstract them, where a neuron is less important if it contributes less to the network's output. We measure the importance of neurons by their *estimated values*, where the estimated value $EV(v_{i,j})$ of a hidden neuron $v_{i,j}$ estimates its value using the mean of its upper and lower bounds, i.e.,

$$EV(v_{i,j}) = \frac{1}{2}\left(\text{ub}^N(v_{i,j}) + \text{lb}^N(v_{i,j})\right).$$

After a hidden neuron $v_{i,j}$ with the minimal estimated value $EV(v_{i,j})$ is found, we need to decide which abstraction primitive—Merge or Freeze_+—should be applied to minimize the loss of accuracy. Therefore, we measure the loss of accuracy induced by abstraction primitives, based on which abstraction primitive is chosen.

The loss $L^{\mathsf{f}}(v_{i,j})$ of accuracy by applying the abstraction primitive Freeze_+ on a hidden neuron $v_{i,j}$ is measured by the difference between the estimated value $EV(v_{i,j})$ and the constant $\bar{b}(v_{i,j})$ used for freezing the neuron $v_{i,j}$:

$$L^{\mathsf{f}}(v_{i,j}) = |\bar{b}(v_{i,j}) - EV(v_{i,j})|.$$

As applying the abstraction primitive Merge on two neurons $v_{i,j}$ and $v_{i,k}$ produces an abstract neuron $v_{i,t}$, the loss $L^{\mathsf{m}}(v_{i,j}, v_{i,k})$ of accuracy by applying Merge on $v_{i,j}$ and $v_{i,k}$ should depend on the changes in weights as well as the values of both neurons. Thus, a linear combination of the estimated values is used to estimate $L^{\mathsf{m}}(v_{i,j}, v_{i,k})$:

$$L^{\mathsf{m}}(v_{i,j}, v_{i,k}) = R(v_{i,j}, v_{i,t}) \cdot EV(v_{i,j}) + R(v_{i,k}, v_{i,t}) \cdot EV(v_{i,k}),$$

where each coefficient $R(\cdot)$ is a ratio characterizing changes in the weights of incoming edges to the new neuron $v_{i,t}$:

$$R(v_{i,j}, v_{i,t}) = \frac{\sum_p |\bar{w}(v_{i-1,p}, v_{i,t}) - w(v_{i-1,p}, v_{i,j})|}{\sum_p |w(v_{i-1,p}, v_{i,j})|}.$$

Based on the above loss measurements, we propose a value-guided abstraction strategy, described in Algorithm 2, to synergistically apply abstraction primitives. It determines which abstraction step should be chosen at the current iteration. It starts by selecting a hidden neuron $v_{i,j}$ with the minimal estimated value $EV(v_{i,j})$ as the target neuron (line 1). Then it chooses among all

ALGORITHM 2: Value-guided Abstraction Strategy

```
Input: A DNN $\bar{N}$, a mapping $EV$ from neurons to their estimated values
Output: A new abstract DNN $\bar{N}'$, updated $EV$
 1: Select a hidden neuron $v_{i,j}$ with minimal $EV(v_{i,j})$
 2: $minLoss \leftarrow \infty$, $bestStep \leftarrow \bot$
 3: for each hidden neuron $v_{i,k}$ of same label as $v_{i,j}$ do
 4:     if $L^{\mathsf{m}}(v_{i,j}, v_{i,k}) < minLoss$ then
 5:         $minLoss \leftarrow L^{\mathsf{m}}(v_{i,j}, v_{i,k})$, $bestStep \leftarrow \textsc{Merge}(v_{i,j}, v_{i,k})$
 6: if $L^{\mathsf{f}}(v_{i,j}) < minLoss$ then
 7:     $minLoss \leftarrow L^{\mathsf{f}}(v_{i,j})$, $bestStep \leftarrow \textsc{Freeze}_{+}(v_{i,j})$
 8: Apply $bestStep$ on $\bar{N}$ to construct $\bar{N}'$
 9: Update $EV$ according to $bestStep$
10: return $\bar{N}'$, $EV$
```

available abstraction steps the one that loses the least accuracy (lines 3–7). At the end, this optimal abstraction step is applied to the DNN $\bar{N}$ resulting in an abstract DNN $\bar{N}'$ (line 8). Furthermore, the mapping EV is updated accordingly for next use. We note that the upper/lower bounds of the original DNN are used to compute estimated values and apply the $\textsc{Freeze}_{+}$ abstraction primitive.

For the second question on the number of abstraction steps, we adopt the terminating condition proposed in Reference [12]. A set X of inputs satisfying the input property P is sampled before abstraction. The $\textsc{Abstract}$ procedure iteratively applies abstraction steps to build a sequence of over-approximations $N_1, N_2, \ldots, N_k$ from the original DNN N according to the value-guided abstraction strategy (i.e., Algorithm 2) until $N_k(\boldsymbol{x})$ for some input $\boldsymbol{x} \in X$ violates the output property Q.

Remark that the value-guided abstraction strategy together with the terminating condition is a trade-off to address challenge C2.

5 NETWORK REFINEMENT

In this section, we present our refinement procedure $\textsc{Refine}$.

5.1 Refinement Primitives

Given a verification problem $\langle N, P, Q\rangle$ and a counterexample *cex* of an abstract DNN $\bar{N}$, which is spurious on the original DNN N, to address challenge C3, the goal of refining the over-approximation $\bar{N}$ of the DNN N is to construct a network $\bar{N}'$ satisfying the following two conditions:

- $\bar{N}(\boldsymbol{x}) \geq \bar{N}'(\boldsymbol{x}) \geq N(\boldsymbol{x})$ for each input $\boldsymbol{x}$ fulfilling the input property P;
- $\bar{N}'(cex)$ satisfies the output property Q.

That means refinement primitives should construct a DNN $\bar{N}'$, which over-approximates N and is itself over-approximated by $\bar{N}$, to exclude the spurious counterexample *cex* of N.

Generally speaking, we define refinement primitives as the inverses of abstraction primitives. Particularly, *refinement steps*, the instances of refinement primitives, are the inverses of abstraction steps. Therefore, we introduce two refinement primitives: $\textsc{Split}$ and $\textsc{Recover}$, corresponding to the abstraction primitives $\textsc{Merge}$ and $\textsc{Freeze}_{+}$, respectively. The $\textsc{Split}$ refinement primitive splits an abstract neuron back into two neurons that were merged by applying the abstraction primitive $\textsc{Merge}$; and the $\textsc{Recover}$ refinement primitive recovers a constant neuron into the status before applying the abstraction primitive $\textsc{Freeze}_{+}$.

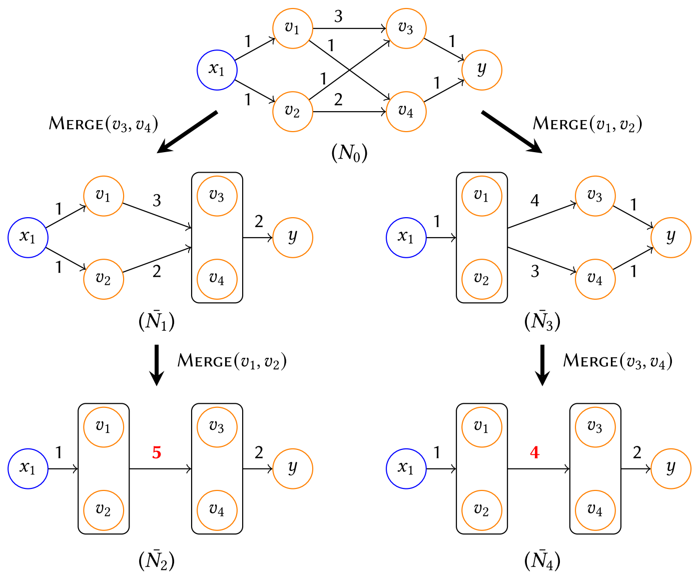


Fig. 3. Abstract DNNs by applying same Merge steps in different orders.

Now the question is, *can we freely choose any abstract or constant neuron in the given abstract DNN $\bar{N}$ to perform a refinement step without damaging soundness (i.e., challenge C3), like what we do when performing an abstraction step?* Unfortunately, the answer is negative.

Example 5.1. Consider the DNN N_0 shown in Figure 3, where all neurons are inc. Assume all biases are 0. By iteratively applying the abstraction steps Merge(v_3, v_4) and Merge(v_1, v_2) (respectively, Merge(v_1, v_2) and Merge(v_3, v_4)) on the DNN N_0, we obtain the DNN $\bar{N}_2$ (respectively, $\bar{N}_4$) as depicted in Figure 3. It is easy to see that the abstract DNNs $\bar{N}_2$ and $\bar{N}_4$ are different. Consequently, if the refinement primitive Split is applied to the abstract neuron of $\{v_1, v_2\}$ in $\bar{N}_4$, then there is no way to get weighted edges from the neurons v_1 and v_2 to the abstract neuron of $\{v_3, v_4\}$. Similarly, if Split is applied to the abstract neuron of $\{v_3, v_4\}$ in $\bar{N}_2$, then there is no way to get weighted edges from the abstract neuron of $\{v_1, v_2\}$ to the neurons v_3 and v_4.

Example 5.1 shows that different orders of abstraction steps may result in different over-approximations. Conversely, to reverse the effects of the abstraction steps, the refinement steps must follow some specific order as well.

A straightforward idea to solve this problem is to iteratively apply the corresponding refinement steps in the reverse order of the abstraction steps, independent of the given counterexample *cex*. However, although refining an abstract DNN in this way provides soundness, the counterexample *cex* may not be quickly excluded. This motivates us to introduce a dependency relation between abstraction steps, which restricts the order of refinement steps, but is more flexible than the reverse order such that proper refinement steps can be applied to quickly exclude the counterexample *cex*.

Definition 5.2. Given a sequence of abstraction steps $\tau = \beta_1\beta_2 \cdots \beta_m$, the *dependency relation* induced by τ is defined as follows:

(i) each abstraction step β_i applied in the kth layer depends on the Freeze$_+$ step β_j that happens in the k'th layer for any $k' > k$;

(ii) the Merge step β_i that merges v and v' in the kth layer, depends on (a) all abstraction steps producing v and v' (if any), and (b) all Merge steps β_j at both the layers $k-1$ and $k+1$ if $j < i$.

Intuitively, Item (i) ensures that the inverses of all abstraction steps β_i at the kth layer happen before the inverses of all the Freeze_+ steps at the k'th layer for $k < k'$, so that Lemma 4.10 is applicable when the corresponding abstraction step β_i is performed on the refined DNN $\bar{N}'$, i.e., ensuring $\bar{N}(\boldsymbol{x}) \geq \bar{N}'(\boldsymbol{x})$. Item (ii) guarantees well-definedness of the Merge step on v and v', because it requires information provided by the abstraction steps producing v and v' as well as all Merge steps that happened at adjacent layers.

A *dependency graph* $\mathcal{G}$ for a sequence τ of abstraction steps is a directed acyclic graph derived by the dependency relation, where the vertices are abstraction steps and an edge from β_i to β_j exists if β_i depends on β_j. We prove that:

Theorem 5.3. *Given a verification problem $\langle N, P, Q\rangle$, let $\bar{N}$ be the abstract DNN constructed from N by applying a sequence τ of abstraction steps, $\mathcal{G}$ the dependency graph for τ, and $\bar{N}'$ the DNN refined from $\bar{N}$ by a refinement step γ. It holds that $\bar{N}(\boldsymbol{x}) \geq \bar{N}'(\boldsymbol{x}) \geq N(\boldsymbol{x})$ for each input $\boldsymbol{x}$ fulfilling the input property P, if γ is the inverse of an abstraction step β in $\mathcal{G}$ that has no incoming edges.*

To prove Theorem 5.3, we first present the following two propositions, which can be proved in a similar way to the proof of Proposition 4.12, by case analysis on whether α_1 and α_2 are applied in the same layer or not.

Proposition 5.4. *Given a verification problem $\langle N, P, Q\rangle$, a DNN N_i, and a length-2 sequence $\tau = \alpha_1\alpha_2$ of Merge steps. Assume that applying τ on the DNN N_i yields a DNN N_i'. If*

(i) α_1 and α_2 are not applied in two adjacent layers, and
(ii) α_1 and α_2 do not depend on each other when they are applied in the same layer,

then a DNN N_i'' can be constructed by applying the sequence $\tau' = \alpha_2\alpha_1$ on the DNN N_i. Moreover, N_i'' is identical to N_i'.

Proposition 5.5. *Given a verification problem $\langle N, P, Q\rangle$, a DNN N_i, and a length-2 sequence $\tau = \alpha_1\alpha_2$ where α_1 is a Freeze_+^N step and α_2 is a Merge step. Assume that applying τ on the DNN N_i yields a DNN N_i'. If α_2 is not applied to the constant neuron created by α_1, then a DNN N_i'' can be constructed by applying the sequence $\tau' = \alpha_2\alpha_1$ on the DNN N_i. Moreover, N_i'' is identical to N_i'.*

Utilizing Propositions 4.12, 5.4, and 5.5, we are now ready to prove Theorem 5.3:

Proof of Theorem 5.3. Assume that $\tau = \beta_1\beta_2\cdots\beta_k$ and the chosen abstraction step β that has no incoming edges in $\mathcal{G}$ is the jth step in τ, namely, $\beta = \beta_j$. For any j' with $j < j' \leq k$, it is straightforward to show that the length-2 sequence $\beta_j\beta_{j'}$ satisfies the premises of Propositions 4.12, 5.4, or 5.5, by the definition of dependency relation (Definition 5.2) and case analysis on:

- β_j is a Merge step or a Freeze_+^N step;
- $\beta_{j'}$ is a Merge step or a Freeze_+^N step;
- Assuming β_j and $\beta_{j'}$ are applied in the ith and i'th layers, respectively, whether $i < i'-1$, $i = i'-1$, $i = i'$, $i = i'+1$ or $i > i'+1$.

Then Propositions 4.12, 5.4, and 5.5 can be repeatedly applied in τ to the β_j step and its succeeding step, resulting in a new sequence $\tau' = \beta_1\cdots\beta_{j-1}\beta_{j+1}\cdots\beta_k\beta_j$. Assume that applying τ' on N produces a sequence of DNNs $N_1', N_2'\ldots, N_k'$. The DNN N_k' is identical to $\bar{N}$ by Propositions 4.12, 5.4, and 5.5. We get that N_{k-1}' is identical to $\bar{N}'$ by assumption. Therefore, the DNN N_{k-1}', thus $\bar{N}'$, is an over-approximation of the DNN N by Theorem 4.13. The DNN N_k', thus $\bar{N}$, is

ALGORITHM 3: Counterexample-guided Refinement Strategy

```
Input: Original DNN $N$, abstract DNN $\bar{N}$, dependency graph $\mathcal{G}$, counterexample $cex$
Output: Refined DNN $\bar{N}'$, updated $\mathcal{G}$
1: Extract the set $C$ of candidate abstraction steps from $\mathcal{G}$
2: $maxGain \leftarrow 0$, $bestStep \leftarrow \perp$
3: for each candidate step $s$ in $C$ do
4:     if $P_{cex}(s) > maxGain$ then
5:         $maxGain \leftarrow P_{cex}(s)$,  $bestStep \leftarrow$ refinement step for $s$
6: Build $\bar{N}'$ from $\bar{N}$ by applying the refinement step $bestStep$
7: Update dependency graph $\mathcal{G}$
8: return $\bar{N}'$, $\mathcal{G}$
```

an over-approximation of the DNN $\bar{N}'$ by Lemma 4.4 if β_j is a Merge step and by Lemma 4.10 if β_j is a Freeze_{+}^{N} step. □

In summary, to refine an over-approximation $\bar{N}$, we first construct the dependency graph $\mathcal{G}$ from the sequence of abstraction steps deriving $\bar{N}$. We pick any abstraction step in $\mathcal{G}$ with no incoming edges, i.e., not depended by others, and perform a corresponding refinement step. By Theorem 5.3, the refined DNN $\bar{N}'$ in such a way ensures soundness, thus, partially solving challenge C3.

Example 5.6. Recall Example 5.1 in the left part of Figure 3. The abstract DNN $\bar{N}_2$ is obtained from N_0 by applying the sequence $\tau = [\text{Merge}(v_3, v_4), \text{Merge}(v_1, v_2)]$ of Merge steps. The corresponding dependency graph $\mathcal{G}$ contains two vertices $\text{Merge}(v_3, v_4)$ and $\text{Merge}(v_1, v_2)$, as well as an edge from $\text{Merge}(v_1, v_2)$ to $\text{Merge}(v_3, v_4)$. To soundly refine $\bar{N}_2$, only $\text{Merge}(v_1, v_2)$ can be picked to reverse.

5.2 Counterexample-guided Refinement

Similar to the abstraction procedure, to address challenge C4, the following two questions should be considered in the refinement procedure: *which available refinement step should be chosen*, and *how many steps should be performed*?

According to Theorem 5.3, any abstraction step in the dependency graph without incoming edges can be chosen for refinement. Thus, there may exist several applicable refinement steps during an iteration of the refinement procedure. Contrary to the abstraction strategy, we expect a refinement step to restore more accuracy, which corresponds to the candidate abstraction step that reduced more accuracy. Therefore, we estimate the gain of accuracy for reversing a candidate abstraction step s via the following profit function $P_{cex}(\cdot)$ w.r.t. the spurious counterexample cex,

$$P_{cex}(s) = \begin{cases} |\sum_{v \in \mathcal{V}_{\bar{v}}} v(cex) - \bar{v}(cex)|, & \text{if } s \text{ is Merge yielding } \bar{v}; \\ |v(cex) - \bar{b}(\bar{v})|, & \text{if } s \text{ is Freeze}_{+} \text{ on } v \text{ yielding } \bar{v}; \end{cases}$$

where $\mathcal{V}_{\bar{v}}$ denotes the atomic neurons in the original DNN N from which $\bar{v}$ is obtained, and $v(cex)$ (respectively, $\bar{v}(cex)$) denotes the exact value of a neuron v (respectively, $\bar{v}$) under the input cex to the original DNN N (respectively, the abstract DNN $\bar{N}$).

Based on the profit function $P_{cex}(\cdot)$, we propose a counterexample-guided refinement strategy (cf. Algorithm 3) that chooses a candidate refinement step, aimed at regaining the most accuracy. However, the strategy in Algorithm 3 does not guarantee that the counterexample cex is ruled out by a single refinement step, i.e., cex may still be a counterexample of the refined DNN $\bar{N}'$. To solve the second question for the refinement procedure, the routine in Algorithm 3 is repeated until cex is ruled out from the refined DNN.

The counterexample-guided refinement strategy, coupled with the terminating condition and refinement primitives, solves challenges C3 and C4.

6 EVALUATION

We implement our approach in a tool, NARv (**N**etwork **A**bstraction-**R**efinement for **v**erification). NARv utilizes the promising symbolic interval analysis tool ReluVal [49] for computing the upper/lower bounds of neurons, while the back-end verification engine can be configured with any sound tool that can produce a counterexample when the verification problem does not hold.

Following many previous works, e.g., CEGAR-NN [12], AI^2 [14], DeepZ [40], DeepPoly [41], ReluVal [49], and DeepSRGR [52], in this implementation, we consider input properties in conjunctions of linear constraints, which are able to express L_1 and L_∞ norms [4] and all the properties of the 2nd International Verification of Neural Networks Competition (VNN-COMP'21) [1].

To evaluate NARv, two promising tools, Marabou [22] and Planet [11], are integrated as the back-end verification engines. Both Marabou and Planet are sound and complete, thus NARv is sound and complete. Marabou is chosen because it is the one with the least startup overhead among 11 state-of-the-art DNN verification tools participating in VNN-COMP'22 (cf. Table 6 in the summary and results of the competition [33]), which is important for a back-end engine that would be invoked frequently. Planet is chosen because it is comparable to Marabou (even better on some benchmarks) [22]. Furthermore, both Marabou and Planet employ a uniform solving strategy for all verification problems, without the need for manually tuning the optimized options for different benchmarks [33]. We do not consider α-β-CROWN [53] or MN-BaB [13], because (i) their startup overhead is costly (the average overhead for each single run of Marabou is merely 0.2 s, whereas the overheads of α-β-CROWN and MN-BaB are 6.7 s and 8.2 s, respectively), (ii) they do not have a uniform solving strategy thus requiring manual fine-tuning for different networks to be optimal, and (iii) they both utilize GPUs, which currently are not supported in our tool.

The experiments are designed to answer the following two research questions:

RQ1: Effectiveness. Can our CEGAR-based verification approach NARv boost the sound and complete verification tools Marabou and Planet?

RQ2: Performance. Does NARv outperform CEGAR-NN [12], the most relevant structure-oriented CEGAR-based approach?

We evaluate NARv on three widely used benchmarks and datasets: 45 real-world DNNs from `ACAS Xu` [20], one DNN trained on dataset `MNIST` [26], and two DNNs trained on `CIFAR-10` [23]. The DNNs trained on `MNIST` and `CIFAR-10` are relatively large. Not all of complete methods are effective in solving useful verification problems in acceptable time for them [47].

`ACAS Xu` is a collision avoidance system built for unmanned aircrafts. The system consists of 45 real-world DNNs, each of which has 310 neurons, including 5 inputs, 5 outputs, and 6 hidden layers with 50 neurons per layer (denoted by 6×50). The inputs take normalized data from airborne sensors, representing the relative position and speed of intruders. The outputs provide five kinds of turning advisories to prevent the aircraft from collision.

`MNIST` is a standard dataset for handwritten digit recognition. The DNN used in our evaluation is provided by VNN-COMP'21. It contains 1,306 neurons, including 2×256 hidden neurons. Due to the 28×28 format of the images, the input layer takes 784 pixels in greyscale and the output layer has 10 neurons producing the classification scores for the 10 possible digits.

`CIFAR-10` is a colored image dataset that consists of 60,000 32×32 RGB images in 10 classes (e.g., cat or dog). The two DNNs used in our evaluation are collected from the benchmarks of the ERAN toolset [25]. They have 3,072 input neurons representing the pixel values of the 3 color channels and 10 outputs as the classification results. The sizes of their hidden layers are 4×100 and 6×100, respectively. Their total sizes are hence 3,482 and 3,682.

The properties to be verified are the robustness of DNNs against adversarial examples. We verify whether the classification result for each input on a target DNN remains the same after adding small perturbations onto that input, where the perturbations are limited within a given threshold δ using the L_∞ norm [4]. All the experiments are run on a Linux server with two Intel Xeon Sliver-4214 CPUs and 64 GB RAM. The timeout is set to 1 h for each ACAS Xu verification problem, whereas 10 h for each verification problem on MNIST and CIFAR-10 DNNs.

The experimental results show that our approach is able to significantly boost the performance of both Marabou and Planet and significantly outperforms CEGAR-NN.

6.1 RQ1: Effectiveness Evaluation

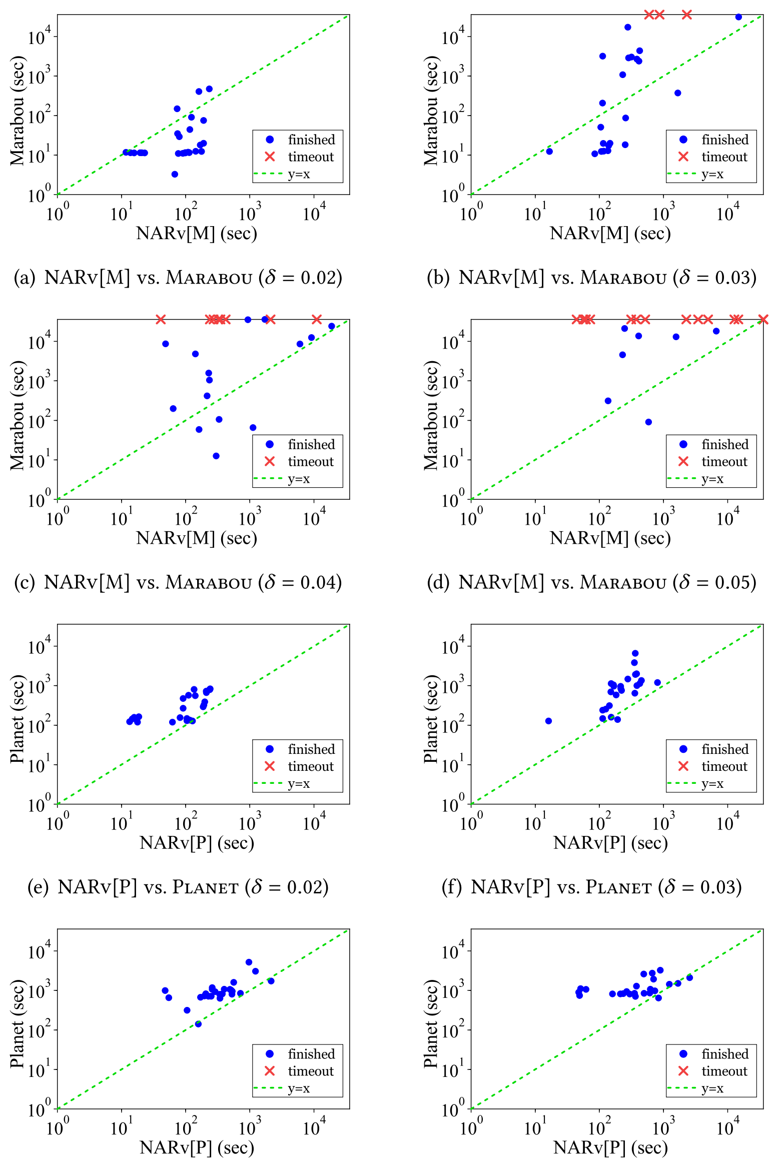
Setup. To answer **RQ1**, we evaluate the effectiveness of NARv on the relatively large networks trained on MNIST and CIFAR-10. Two NARv configurations NARv[M] and NARv[P] are set up, respectively, with Marabou and Planet as the back-end verification engines. NARv[M] and NARv[P] are then compared with Marabou and Planet, respectively, on the MNIST and the CIFAR-10 networks. For each network, we verify robustness against 4 perturbation thresholds δ ranging: from 0.02 to 0.05 for the MNIST network, and from 0.001 to 0.004 for the CIFAR-10 networks. These thresholds are selected, since the robustness threshold is close to 0.06 for the MNIST network, and 0.005 for the CIFAR-10 networks, thus yielding interesting yet hard robustness verification problems. There are 25 and 12 verification problems for each perturbation threshold δ in each MNIST and CIFAR-10 network, respectively.

6.1.1 Results on MNIST. Figure 4 shows the experimental results on the MNIST network for the effectiveness evaluation. Figures 4(a)–4(d) (respectively, Figures 4(e)–4(h)) report the comparisons between NARv[M] and Marabou (respectively, NARv[P] and Planet). The x-axes and y-axes refer to the verification time (in seconds, logscale) spent by NARv[M] and Marabou (respectively, NARv[P] and Planet). The blue dots represent the solved problems and the red crosses on the top and right borders are the cases where Marabou (respectively, Planet) and NARv[M] (respectively, NARv[P]) run out of time. Therefore, the dots and crosses distributed above the green line indicate the cases where NARv[M] (respectively, NARv[P]) is faster.

Overall, the results show that most of the dots and crosses are situated above the green line, indicating that our tools, NARv[M] and NARv[P], are faster in most cases, especially for the more difficult problems with large perturbation thresholds, e.g., $\delta = 0.04$ and $\delta = 0.05$. More specifically, from Figures 4(a)–4(d), Marabou appears to have a good performance when the perturbation threshold δ is small, as the solving space of the verification problem is small. When $\delta = 0.02$, it is faster than NARv[M] in most cases. However, it begins to time out (10 h) when $\delta \geq 0.03$. NARv[M] nevertheless is able to solve most of the verification problems for all perturbation thresholds, running faster in most cases when $\delta \geq 0.03$. From Figures 4(e)–4(h), Planet successfully verifies all the verification problems, just as our tool NARv[P] does. But NARv[P] moreover succeeds in spending less time than Planet for almost all the problems.

The more detailed statistics are displayed in Table 1, where the average verification time in seconds (“Avg. time”), the median of verification time in seconds (“Med. time”), and the numbers of successfully verified problems (“#Verified”) by each tool are depicted.

We can observe that NARv[M] and NARv[P] solve more problems and in general are much faster than Marabou and Planet. This reveals that our approach is able to significantly boost the performance of both Marabou and Planet in most cases on the MNIST network. The best improvements appear when $\delta = 0.03$, reducing 86.3% and 78.0% verification time, respectively, for



(a) NARv[M] vs. Marabou ($\delta = 0.02$)
(b) NARv[M] vs. Marabou ($\delta = 0.03$)
(c) NARv[M] vs. Marabou ($\delta = 0.04$)
(d) NARv[M] vs. Marabou ($\delta = 0.05$)
(e) NARv[P] vs. Planet ($\delta = 0.02$)
(f) NARv[P] vs. Planet ($\delta = 0.03$)
(g) NARv[P] vs. Planet ($\delta = 0.04$)
(h) NARv[P] vs. Planet ($\delta = 0.05$)

Fig. 4. Verification Time (logscale) on MNIST Network, with Marabou, Planet, and their Boosted Versions

Marabou and Planet. More specifically, the numbers of verified problems by Marabou become less and less with the increase of δ. When $\delta = 0.05$, it only successfully solves 7 (28%) problems out of all the 25 verification problems. In contrast, NARv[M] is able to solve most of the problems for all perturbation thresholds except for $\delta = 0.05$. When $\delta = 0.05$, it solves 20 problems, namely, 52% (= (20 − 7)/25) more than Marabou. As a result, NARv[M] achieves a 98.5% reduction in median verification time. On average, NARv[M] spends 71.7% (≈ (28,465.46 − 8,056.73)/28,465.46) less verification time than Marabou when $\delta = 0.05$. Compared over Planet, which is able to solve all the MNIST verification problems as depicted in Table 1, NARv[P] also solves all the problems,

Table 1. Effectiveness Evaluation with Marabou and Planet

		Avg. time (s)	Med. time (s)	#Verified	Avg. time (s)	Med. time (s)	#Verified
Network	δ	**NARv[M]**			**Marabou**		
MNIST	0.02	99.54 (△ 65.0%)	98.52 (△ 730.8%)	25 (-)	60.34	11.86	25
	0.03	971.70 (▼ 86.3%)	251.99 (▼ 32.8%)	25 (▲ 12%)	7,084.99	375.10	22
	0.04	2,458.05 (▼ 85.6%)	328.05 (▼ 96.2%)	25 (▲ 40%)	17,063.42	8,647.61	15
	0.05	8,056.73 (▼ 71.7%)	552.47 (▼ 98.5%)	20 (▲ 52%)	28,465.46	36,000.00	7
CIFAR-10 4×100	0.001	470.96 (▼ 54.0%)	185.88 (▼ 46.4%)	12 (-)	1,023.08	346.64	12
	0.002	1,634.30 (▼ 61.5%)	559.43 (▼ 87.4%)	12 (▲ 8%)	4,241.73	4,431.79	11
	0.003	13,823.15 (▼ 38.8%)	14,543.92 (▼ 28.2%)	12 (▲ 25%)	22,600.87	20,266.84	9
	0.004	27,421.56 (▼ 13.4%)	36,000.00 (-)	5 (▲ 17%)	31,646.65	36,000.00	3
CIFAR-10 6×100	0.001	1,581.66 (▼ 82.1%)	280.68 (▼ 96.1%)	12 (-)	8,824.25	7,141.30	12
	0.002	12,545.33 (▼ 56.0%)	253.61 (▼ 99.3%)	8 (▲ 33%)	28,515.69	36,000.00	4
	0.003	15,241.70 (▼ 57.7%)	483.98 (▼ 98.7%)	7 (▲ 58%)	36,000.00	36,000.00	0
	0.004	15,200.91 (▼ 57.8%)	407.53 (▼ 98.9%)	7 (▲ 58%)	36,000.00	36,000.00	0
Network	δ	**NARv[P]**			**Planet**		
MNIST	0.02	118.91 (▼ 67.5%)	110.85 (▼ 58.6%)	25 (-)	366.39	267.54	25
	0.03	259.54 (▼ 78.0%)	214.83 (▼ 77.7%)	25 (-)	1,182.32	964.02	25
	0.04	454.14 (▼ 60.3%)	338.33 (▼ 60.0%)	25 (-)	1,143.16	845.88	25
	0.05	585.59 (▼ 54.1%)	492.23 (▼ 49.8%)	25 (-)	1,275.57	979.83	25
CIFAR-10 4×100	0.001	5,192.57 (▼ 40.0%)	4,094.63 (▼ 49.9%)	12 (-)	8,650.30	8,170.67	12
	0.002	25,522.47 (▼ 26.4%)	34,119.01 (▼ 5.2%)	6 (▲ 33%)	34,656.71	36,000.00	2
	0.003	27,138.98 (▼ 24.6%)	36,000.00 (-)	3 (▲ 25%)	36,000.00	36,000.00	0
	0.004	33,027.16 (▼ 8.3%)	36,000.00 (-)	1 (▲ 8%)	36,000.00	36,000.00	0
CIFAR-10 6×100	0.001	12,682.02 (▼ 63.4%)	285.15 (▼ 99.2%)	8 (▲ 58%)	34,641.92	36,000.00	1
	0.002	19,097.04 (▼ 47.0%)	23,858.54 (▼ 33.7%)	7 (▲ 58%)	36,000.00	36,000.00	0
	0.003	15,249.89 (▼ 57.6%)	487.20 (▼ 98.6%)	7 (▲ 58%)	36,000.00	36,000.00	0
	0.004	21,187.41 (▼ 41.1%)	36,000.00 (-)	5 (▲ 42%)	36,000.00	36,000.00	0

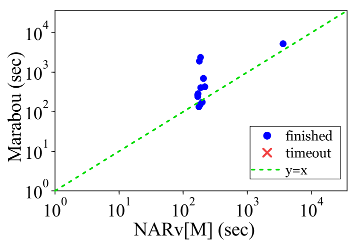
meanwhile spends less time on average for all the perturbation thresholds, reducing at least 54.1% verification time.

Finally, we should emphasize that verified robustness with large perturbation threshold δ is more interesting in practice.

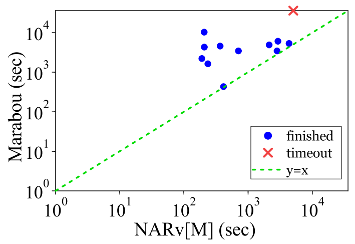
6.1.2 Results on `CIFAR-10`. The experimental results on the two `CIFAR-10` networks are depicted in Figures 5 and 6, respectively, where Figures 5(a)–5(d) and 6(a)–6(d) (respectively, Figures 5(e)–5(h) and 6(e)–6(h)) compare the verification time required by NARv[M] and Marabou (respectively, NARv[P] and Planet) for each verification problem.

It is easy to observe that most of the dots and crosses are situated strictly above the green line in both Figures 5 and 6, particularly in Figure 6 that shows the results on the larger 6×100 `CIFAR-10` network. This indicates that our approach can significantly boost Marabou and Planet in most cases. We point out that Planet can hardly solve any verification problems in the 6×100 `CIFAR-10` network (cf. Figures 6(e)–6(h)). Nevertheless, boosted by our approach, NARv[P] is able to solve many verification problems within the time limit (10 h).

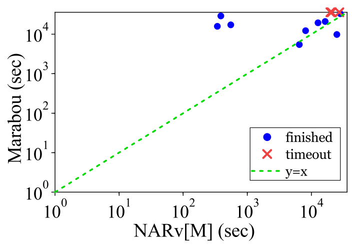
Table 1 gives the detailed results. It is worth noting that Planet fails to solve any problems before timing out when $\delta \geq 0.002$ on the 6×100 `CIFAR-10` network. It successfully solves only 1 (2.1%) problem of all 48 problems for 4 different perturbation thresholds. Boosted by our approach,



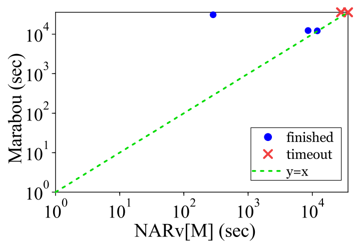
(a) NARv[M] vs. Marabou (δ = 0.001)



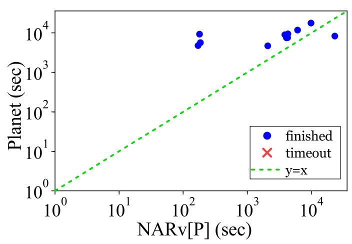
(b) NARv[M] vs. Marabou (δ = 0.002)



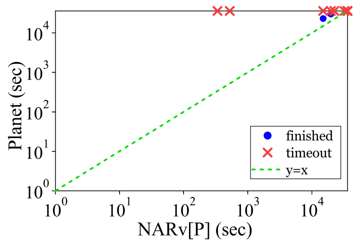
(c) NARv[M] vs. Marabou (δ = 0.003)



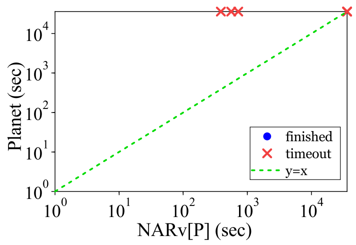
(d) NARv[M] vs. Marabou (δ = 0.004)



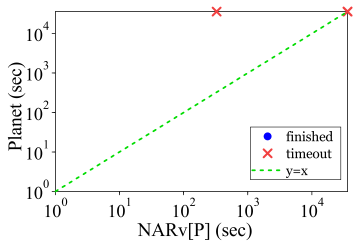
(e) NARv[P] vs. Planet (δ = 0.001)



(f) NARv[P] vs. Planet (δ = 0.002)



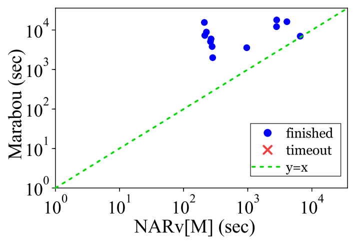
(g) NARv[P] vs. Planet (δ = 0.003)



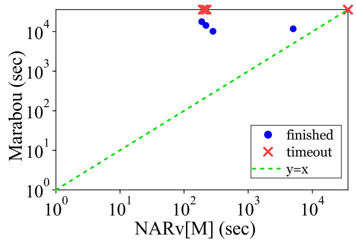
(h) NARv[P] vs. Planet (δ = 0.004)

Fig. 5. Verification Time (logscale) on 4 × 100 `CIFAR-10` Network, with Marabou, Planet, and Boosted Versions

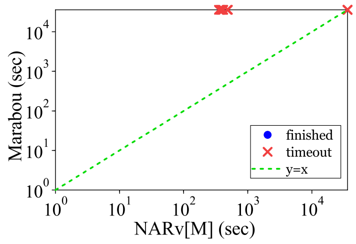
NARv[P] is able to solve 7 (58%) problems when δ is 0.002 and 0.003, and 5 (42%) problems when $\delta = 0.004$, succeeding in 54.2% ($\approx (8+7+7+5-1)/48$) more in total for that network. On the other hand, NARv[M] is able to solve 37.5% more problems in total than Marabou when verifying the 6 × 100 `CIFAR-10` network, achieving at least a 96.1% reduction in median verification time. As for the 4 × 100 `CIFAR-10` network, NARv[M] (respectively, NARv[P]) verifies 12.5% (respectively, 16.7%) more problems in total than Marabou (respectively, Planet).



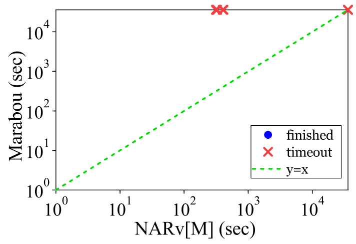
(a) NARv[M] vs. Marabou (δ = 0.001)



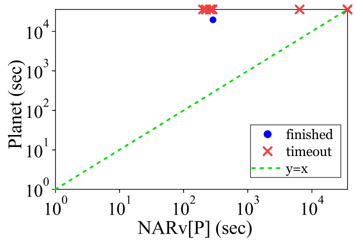
(b) NARv[M] vs. Marabou (δ = 0.002)



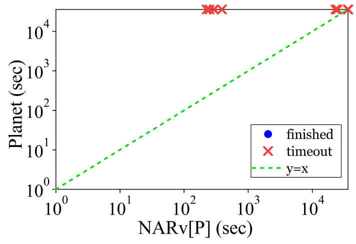
(c) NARv[M] vs. Marabou (δ = 0.003)



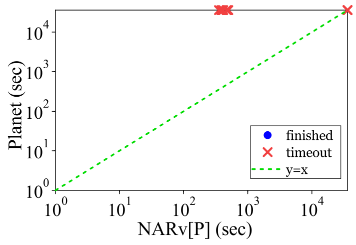
(d) NARv[M] vs. Marabou (δ = 0.004)



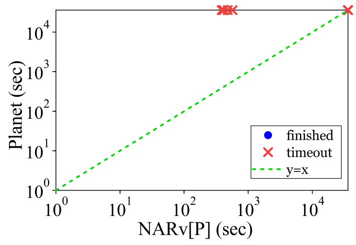
(e) NARv[P] vs. Planet (δ = 0.001)



(f) NARv[P] vs. Planet (δ = 0.002)



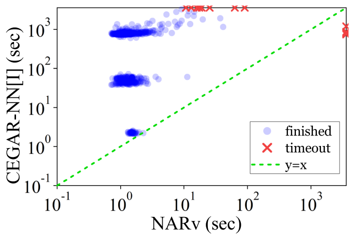
(g) NARv[P] vs. Planet (δ = 0.003)



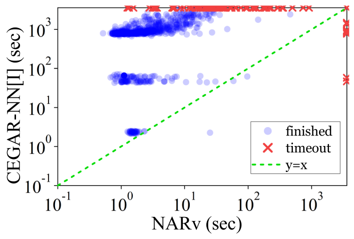
(h) NARv[P] vs. Planet (δ = 0.004)

Fig. 6. Verification Time (logscale) on 6×100 `CIFAR-10` Network, with Marabou, Planet, and Boosted Versions

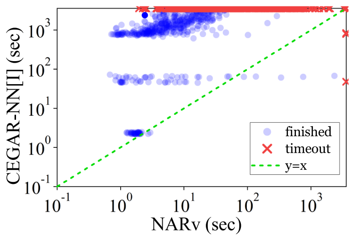
Answer to RQ1: NARv can significantly boost the performance of two promising tools, Marabou and Planet, solving more verification tasks, reducing on average 13.4%–86.3% verification time of Marabou on almost all the verification tasks, and reducing on average 8.3%–78.0% verification time of Planet on all the verification tasks.



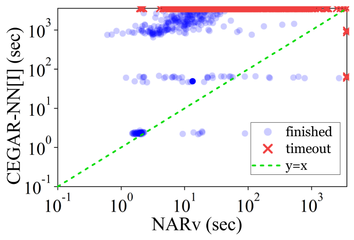
(a) $\delta = 0.01$



(b) $\delta = 0.02$



(c) $\delta = 0.03$



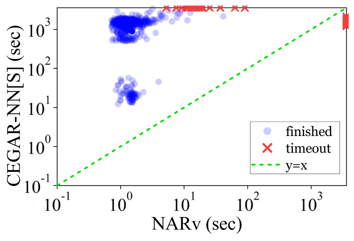
(d) $\delta = 0.04$

Fig. 7. Comparison to CEGAR-NN[I] on ACAS Xu (in seconds, logscale)

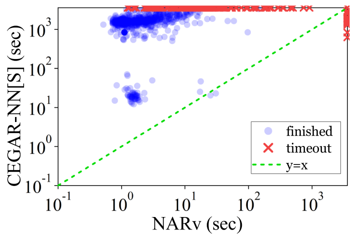
6.2 RQ2: Performance Evaluation

Setup. To answer **RQ2**, we compare NARv over CEGAR-NN, both of which use the same back-end verification engine Marabou. CEGAR-NN provides two abstraction strategies, named *indicator-guided abstraction* and *abstraction to saturation*, where the former iteratively merges two neurons until some pre-sampled input violates the given property while the latter aggressively and iteratively merges two neurons producing the smallest over-approximation. We refer to CEGAR-NN with those abstraction strategies as CEGAR-NN[I] and CEGAR-NN[S], respectively. NARv is then compared with both of them using the same benchmark ACAS Xu as adopted in Reference [12]. The perturbation threshold δ is set from 0.01 to 0.04, because we find that the robustness thresholds of most ACAS Xu networks are close to 0.03.

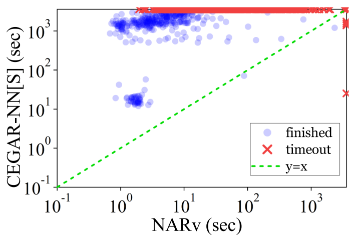
Results. Figure 7 depicts the comparison of the verification time between NARv and CEGAR-NN[I] for each problem. We can observe that most of the dots and crosses are located above the green line for all perturbation thresholds, indicating that NARv is significantly more efficient than CEGAR-NN[I] on most verification problems. We further notice that the points interestingly cluster into three groups w.r.t. the verification time of CEGAR-NN[I] for all perturbation thresholds. The verification time spent by CEGAR-NN[I] in the first group is only around 2–3 s, standing for the cases where the robustness is falsified quickly by confirming a genuine counterexample when sampling the inputs in X (cf. Section 4.4), even before abstraction. The second group, on the other hand, which requires around 35–70 s by CEGAR-NN[I], represents the cases where the verification succeeds when verifying the abstract DNN $\bar{N}$ produced by the abstraction procedure Abstract, without the need for invoking the refinement procedure Refine (cf. Algorithm 1). When refinement is required, the verification time increases, since the sizes of refined networks increase, constituting the third group. The comparison between NARv and CEGAR-NN[S] is illustrated in Figure 8. Similar to Figure 7, it demonstrates that NARv is significantly more efficient than CEGAR-NN[S] on most verification problems. Likewise, the points cluster into groups. The first group represents the cases where the verification on the aggressively



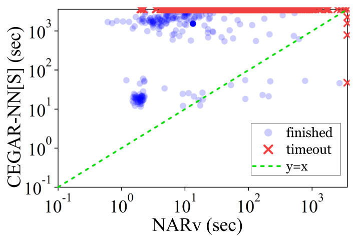
(a) δ = 0.01



(b) δ = 0.02



(c) δ = 0.03



(d) δ = 0.04

Fig. 8. Comparison to CEGAR-NN[S] on ACAS Xu (in seconds, logscale)

Table 2. Abstraction and Verification Results on ACAS Xu

	NARv	CEGAR-NN[I]	CEGAR-NN[S]
δ	Avg. size		
0.01	**271** (▼ 10%)	860 (△ 187%)	828 (△ 176%)
0.02	**210** (▼ 30%)	882 (△ 194%)	824 (△ 175%)
0.03	**212** (▼ 29%)	887 (△ 196%)	784 (△ 161%)
0.04	**229** (▼ 24%)	884 (△ 195%)	684 (△ 128%)
δ	#Verified		
0.01	888 (98.7%)	**892** (99.1%)	884 (98.2%)
0.02	**881** (97.9%)	815 (90.6%)	731 (81.2%)
0.03	**893** (99.2%)	497 (55.2%)	368 (40.9%)
0.04	**886** (98.4%)	327 (36.3%)	233 (25.9%)
δ	Avg. time (s)		
0.01	**49.82**	629.16 (12.6×)	1,373.35 (27.6×)
0.02	**88.10**	1,184.92 (13.4×)	1,980.36 (22.5×)
0.03	**105.04**	2,331.45 (22.2×)	2,835.72 (27.0×)
0.04	**169.46**	2,742.36 (16.2×)	3,034.63 (17.9×)

abstracted small networks succeeds, most of which do not need refinement. The other group with more verification time by CEGAR-NN[S] contains the cases where refinement is necessary.

Table 2 reports the details of all the results, where columns "Avg. time" and "#Verified" are the same as above, and columns "Avg. size" give the average sizes of the abstract/refined networks when the tools successfully verify the problems.

Recall that there are 300 hidden neurons in each given ACAS Xu network. The results show that NARv is effective in reducing network sizes for all perturbation thresholds, and the best reduction is 30% (= $(300 - 210)/300$) on average when $\delta = 0.02$. However, neither of the two abstraction strategies of CEGAR-NN is effective in reducing network sizes. CEGAR-NN[I] and CEGAR-NN[S] even increase the network sizes by 128%–196% on average. There are three main reasons. First, the network preprocessing of CEGAR-NN quadruples the network sizes in the worst case, while ours at most doubles the network sizes. Second, our abstraction synergistically integrates two abstraction primitives to reduce the network size as much as possible when sacrificing the same accuracy, while CEGAR-NN does not. Third, our refinement synergistically integrates two refinement primitives to keep the network size as small as possible when restoring the same amount of accuracy, while CEGAR-NN does not.

Thanks to the effectiveness of our approach in reducing network sizes, NARv is able to successfully solve the most problems in a reasonable time among the three tools when $\delta \geq 0.02$, and is comparable to the other two when $\delta = 0.01$, as the cost of DNN verification is exponential in the network size. Unsurprisingly, NARv is also much faster than both CEGAR-NN[I] and CEGAR-NN[S]. When $\delta = 0.01$, NARv is on average 11.6 ($\approx (629.16 - 49.82)/49.82$) times faster than CEGAR-NN[I], and 26.6 ($\approx (1{,}373.35 - 49.82)/49.82$) times faster than CEGAR-NN[S]. In particular, when $\delta = 0.04$, NARv successfully verifies 72.6% ($\approx (886 - 233)/900$) more problems than CEGAR-NN[S], yet spending only 5.6% ($\approx 169.46/3{,}034.63$) of time on average.

Answer to RQ2: NARv significantly outperforms CEGAR-NN, and is at least 11.6 times faster than CEGAR-NN on average.

We remark that the improvement of NARv over Marabou on the MNIST and CIFAR-10 networks is less than the improvement of NARv over CEGAR-NN on ACAS Xu networks, because CEGAR-NN is not effective in reducing the network sizes and thus fails to improve Marabou on ACAS Xu networks when δ ranges from 0.01 to 0.04. According to the results reported in Reference [12], on average, CEGAR-NN is able to achieve 66% reduction in network sizes on the solved ACAS Xu benchmarks when $\delta = 0.1$. In this work, we set δ from 0.01 to 0.04, because we find that the robustness thresholds of most ACAS Xu networks are close to 0.03, which is consistent with the results of Reference [21] on one ACAS Xu network and five inputs.

6.3 Threats to Validity

Our approach is designed for fully connected feedforward deep neural networks. Convolutional neural networks could be verified by equivalently transforming into fully connected ones [32], but utilizing particular properties of convolutional constructs to build abstraction is certainly an interesting future work. The feasibility has been shown in the concurrent work [34]. However, it remains an open problem for recurrent neural networks.

The verification engines Marabou and Planet adopted in the experiments are primarily based on SMT solving and thus complete but computationally expensive. Our structure-oriented CEGAR approach is proposed to boost them. We have not evaluated with the abstract interpretation approaches, which can be seen as computation-oriented abstraction, thus are incomplete, e.g., References [14, 41]. Although refinement techniques also have been proposed, e.g., input refinement [49], such computation-oriented abstraction-refinement frameworks are orthogonal to our structure-oriented one. Investigating the synergy between them is left for future work, for instance, using them at a back-end verification engine in our approach.

Our approach may fail to boost DNN verification when the verification problems are very easy to solve. More effective heuristics should be added to improve the efficiency for such verification problems. For instance, abstract interpretation based approaches could be leveraged before applying our approach. Nevertheless, our approach can significantly boost the verification of hard problems, which are arguably more interesting and important in practice.

7 RELATED WORK

A large and growing body of work studies heuristic search or other dynamic analysis techniques to test robustness of neural networks, e.g., References [4, 30, 31, 36, 42, 44], cf. Reference [54] for a survey. They are often effective in finding adversarial examples or violations of properties, as opposed to proving the absence of violations. Thus, they are orthogonal to formal verification considered in this work.

The simple and earlier neural network verification makes use of constraint solving, where a verification problem is reduced to the solving of constraints, e.g., SAT/SMT solving [11, 19, 21, 22, 38], LP/MILP solving [10, 27, 29, 45, 51]. Most of these works focus on DNNs with the ReLU activation function. Although these techniques are often sound and complete, they are limited in scalability. Our work targets this scalability problem and boosts existing DNN verification techniques by reducing network sizes.

To improve the scalability, some other DNN verification techniques utilize the idea of abstraction through abstract interpretation [6]. These techniques include ReluVal [49], AI^2 [14], DeepZ [40], DeepPoly [41], NNV [46], DeepSRGR [52], and so on. The key idea is to use well-designed numerical abstract domains, such as boxes [6], zonotopes [15], and polyhedra [7], to over-approximate the computations on sets of inputs in the target DNNs. Due to the over-approximation, these techniques are incomplete. To improve accuracy, some of them incorporate refinement strategies, such as input region splitting [49] and output bound tightening [41, 52]. Different from them, the CEGAR-based approach adopted in this work is based on the structure of DNNs, rather than on the computations, thus is orthogonal to them.

There are other structure-oriented techniques. The abstraction technique proposed in Reference [37] represents the network weights via intervals and merges randomly chosen neurons by merging intervals. A k-means clustering algorithm is leveraged to cluster neurons [2]. In contrast to ours, they rely on specific underlying verification engines and do not provide refinement mechanisms when some spurious counterexamples are reported, thus are incomplete. Gokulanathan et al. [16] proposed to identify and remove inactive neurons via DNN verification problem solving without harming accuracy. But it has to invoke a large number of DNN verification queries.

The closest work to ours is CEGAR-NN [12], which proposed the first structure-oriented CEGAR approach for DNN verification. Inspired by CEGAR-NN, although the high-level CEGAR framework is the same, our work makes three significant technical contributions: (i) our preprocessing at most *doubles* the network size, whilst CEGAR-NN at most *quadruples* the size, leaving a heavy load to the following abstraction procedure as shown in our experimental results; (ii) we provide two complementary abstraction (respectively, refinement) primitives that can induce less inaccuracy (respectively, regain more accuracy) whereas CEGAR-NN provides only one abstraction (respectively, refinement) primitive; (iii) we propose a novel abstraction (respectively, refinement) strategy to syncretize abstraction (respectively, refinement) steps, achieving significantly better performance than CEGAR-NN.

Finally, we remark that any sound tool with the ability to produce counterexamples when properties are violated could be used as the back-end verification engine in our approach.

8 CONCLUSION

We have presented a novel CEGAR-based approach for scalable and exact verification of neural networks. Specifically, we have proposed two structure-oriented abstraction primitives and an abstraction strategy to syncretize abstraction steps, resulting in a novel abstraction procedure. We also have proposed two corresponding refinement primitives and a refinement strategy to syncretize refinement steps, resulting in a novel refinement procedure. We have implemented our approach in a tool and conducted an extensive evaluation on three widely used benchmarks. The results demonstrate that our approach can significantly boost the efficiency of two promising verification tools, Marabou and Planet, without loss of accuracy, in particular, for difficult verification problems. Our approach also significantly outperforms the most relevant structure-oriented CEGAR-based approach, CEGAR-NN, namely, 11.6–26.6 times faster.

ACKNOWLEDGMENTS

The authors thank the anonymous referees for their valuable comments and suggestions.

REFERENCES

[1] Stanley Bak, Changliu Liu, and Taylor T. Johnson. 2021. The 2nd International Verification of Neural Networks Competition (VNN-COMP'21): Summary and Results. CoRR abs/2109.00498 (2021). https://arxiv.org/abs/2109.00498

[2] Pranav Ashok, Vahid Hashemi, Jan Kretínský, and Stefanie Mohr. 2020. DeepAbstract: Neural network abstraction for accelerating verification. In *Proceedings of the18th International Symposium on Automated Technology for Verification and Analysis (ATVA'20) (Lecture Notes in Computer Science, Vol. 12302)*, Dang Van Hung and Oleg Sokolsky (Eds.). Springer, 92–107. https://doi.org/10.1007/978-3-030-59152-6_5

[3] Rudy Bunel, Ilker Turkaslan, Philip H. S. Torr, Pushmeet Kohli, and M. Pawan Kumar. 2017. Piecewise Linear Neural Network verification: A comparative study. Retrieved from http://arxiv.org/abs/1711.00455

[4] Nicholas Carlini and David A. Wagner. 2017. Towards evaluating the robustness of neural networks. In *Proceedings of the IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. 39–57. https://doi.org/10.1109/SP.2017.49

[5] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-guided abstraction refinement. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00) (Lecture Notes in Computer Science, Vol. 1855)*, E. Allen Emerson and A. Prasad Sistla (Eds.). Springer, 154–169. https://doi.org/10.1007/10722167_15

[6] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, Robert M. Graham, Michael A. Harrison, and Ravi Sethi (Eds.). ACM, 238–252. https://doi.org/10.1145/512950.512973

[7] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages*, Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski (Eds.). ACM Press, 84–96. https://doi.org/10.1145/512760.512770

[8] Nilesh N. Dalvi, Pedro M. Domingos, Mausam, Sumit K. Sanghai, and Deepak Verma. 2004. Adversarial classification. In *Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 99–108. https://doi.org/10.1145/1014052.1014066

[9] Nachum Dershowitz and Jean-Pierre Jouannaud. 1990. Rewrite systems. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, Jan van Leeuwen (Ed.). Elsevier and MIT Press, 243–320. https://doi.org/10.1016/b978-0-444-88074-1.50011-1

[10] Souradeep Dutta, Susmit Jha, Sriram Sankaranarayanan, and Ashish Tiwari. 2018. Output range analysis for deep feedforward neural networks. In *Proceedings of the 10th International Symposium on NASA Formal Methods (NFM'18) (Lecture Notes in Computer Science, Vol. 10811)*, Aaron Dutle, César A. Muñoz, and Anthony Narkawicz (Eds.). Springer, 121–138. https://doi.org/10.1007/978-3-319-77935-5_9

[11] Rüdiger Ehlers. 2017. Formal verification of piece-wise linear feed-forward neural networks. In *Proceedings of the 15th International Symposium on Automated Technology for Verification and Analysis ATVA('17) (Lecture Notes in Computer Science, Vol. 10482)*, Deepak D'Souza and K. Narayan Kumar (Eds.). Springer, 269–286. https://doi.org/10.1007/978-3-319-68167-2_19

[12] Yizhak Yisrael Elboher, Justin Gottschlich, and Guy Katz. 2020. An abstraction-based framework for neural network verification. In *Proceedings of the 32nd International Conference Computer Aided Verification (CAV'20) (Lecture Notes in Computer Science, Vol. 12224)*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer, 43–65. https://doi.org/10.1007/978-3-030-53288-8_3

[13] Claudio Ferrari, Mark Niklas Mueller, Nikola Jovanović, and Martin Vechev. 2022. Complete verification via multi-neuron relaxation guided branch-and-bound. In *Proceedings of the International Conference on Learning Representations*.

[14] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin T. Vechev. 2018. AI2: Safety and robustness certification of neural networks with abstract interpretation. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'18)*. IEEE Computer Society, 3–18. https://doi.org/10.1109/SP.2018.00058

[15] Khalil Ghorbal, Eric Goubault, and Sylvie Putot. 2009. The zonotope abstract domain Taylor1+. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV'09) (Lecture Notes in Computer Science, Vol. 5643)*, Ahmed Bouajjani and Oded Maler (Eds.). Springer, 627–633. https://doi.org/10.1007/978-3-642-02658-4_47

[16] Sumathi Gokulanathan, Alexander Feldsher, Adi Malca, Clark W. Barrett, and Guy Katz. 2020. Simplifying neural networks using formal verification. In *Proceedings of the 12th International Symposium on NASA Formal Methods (NFM'20) (Lecture Notes in Computer Science, Vol. 12229)*, Ritchie Lee, Susmit Jha, and Anastasia Mavridou (Eds.). Springer, 85–93. https://doi.org/10.1007/978-3-030-55754-6_5

[17] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and harnessing adversarial examples. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR'15)*.

[18] Geoffrey Hinton, Li Deng, Dong Yu, George E. Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N. Sainath, and Brian Kingsbury. 2012. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Process. Mag.* 29, 6 (2012), 82–97. https://doi.org/10.1109/MSP.2012.2205597

[19] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. 2017. Safety verification of deep neural networks. In *Proceedings of the 29th International Conference on Computer Aided Verification (CAV'17) (Lecture Notes in Computer Science, Vol. 10426)*, Rupak Majumdar and Viktor Kuncak (Eds.). Springer, 3–29. https://doi.org/10.1007/978-3-319-63387-9_1

[20] Kyle D. Julian, Mykel J. Kochenderfer, and Michael P. Owen. 2019. Deep neural network compression for aircraft collision avoidance systems. *J. Guid. Control Dynam.* 42, 3 (2019), 598–608.

[21] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. 2017. Reluplex: An efficient SMT solver for verifying deep neural networks. In *Proceedings of the 29th International Conference on Computer Aided Verification (CAV'17) (Lecture Notes in Computer Science, Vol. 10426)*, Rupak Majumdar and Viktor Kuncak (Eds.). Springer, 97–117. https://doi.org/10.1007/978-3-319-63387-9_5

[22] Guy Katz, Derek A. Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljic, David L. Dill, Mykel J. Kochenderfer, and Clark W. Barrett. 2019. The Marabou framework for verification and analysis of deep neural networks. In *Proceedings of the 31st International Conference on Computer Aided Verification (CAV'19)*, Isil Dillig and Serdar Tasiran (Eds.). Springer, 443–452. https://doi.org/10.1007/978-3-030-25540-4_26

[23] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. Retrieved from https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf

[24] Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. 2017. Adversarial examples in the physical world. In *Proceedings of the 5th International Conference on Learning Representations (ICLR'17)*.

[25] SRI Lab. 2022. *ETH Robustness Analyzer for Neural Networks (ERAN)*. Retrieved from https://github.com/eth-sri/eran

[26] Yann LeCun. 1998. The MNIST database of handwritten digits. Retrieved from http://yann.lecun.com/exdb/mnist/

[27] Wang Lin, Zhengfeng Yang, Xin Chen, Qingye Zhao, Xiangkun Li, Zhiming Liu, and Jifeng He. 2019. Robustness verification of classification deep neural networks via linear programming. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'19)*. Computer Vision Foundation / IEEE, 11418–11427. https://doi.org/10.1109/CVPR.2019.01168

[28] Geert Litjens, Thijs Kooi, Babak Ehteshami Bejnordi, Arnaud Arindra Adiyoso Setio, Francesco Ciompi, Mohsen Ghafoorian, Jeroen A. W. M. van der Laak, Bram van Ginneken, and Clara I. Sánchez. 2017. A survey on deep learning in medical image analysis. *Med. Image Anal.* 42 (2017), 60–88. https://doi.org/10.1016/j.media.2017.07.005

[29] Alessio Lomuscio and Lalit Maganti. 2017. An approach to reachability analysis for feed-forward ReLU neural networks. Retrieved from http://arxiv.org/abs/1706.07351

[30] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. 2018. DeepGauge: Multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 120–131.

[31] Shiqing Ma, Yingqi Liu, Wen-Chuan Lee, Xiangyu Zhang, and Ananth Grama. 2018. MODE: Automated neural network model debugging via state differential analysis and input selection. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 175–186.

[32] Wei Ma and Jun Lu. 2017. An equivalence of fully connected layer and convolutional layer. Retrieved from http://arxiv.org/abs/1712.01252

[33] Mark Niklas Müller, Christopher Brix, Stanley Bak, Changliu Liu, and Taylor T. Johnson. 2022. The third international verification of neural networks competition (VNN-COMP 2022): Summary and results. Retrieved from https://arXiv:2212.10376

[34] Matan Ostrovsky, Clark W. Barrett, and Guy Katz. 2022. An abstraction-refinement approach to verifying convolutional neural networks. In *Proceedings of the 20th International Symposium on Automated Technology for Verification and Analysis*, Ahmed Bouajjani, Lukás Holík, and Zhilin Wu (Eds.), Vol. 13505. Springer, 391–396.

[35] Nicolas Papernot, Patrick D. McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. 2016. The limitations of deep learning in adversarial settings. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P'16)*. 372–387.

[36] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2019. DeepXplore: Automated whitebox testing of deep learning systems. *Commun. ACM* 62, 11 (2019), 137–145.

[37] Pavithra Prabhakar and Zahra Rahimi Afzal. 2019. Abstraction based output range analysis for neural networks. In *Proceedings of the 32nd Annual Conference on Neural Information Processing Systems (NeurIPS'19)*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 15762–15772.

[38] Luca Pulina and Armando Tacchella. 2010. An abstraction-refinement approach to verification of artificial neural networks. In *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV'10) (Lecture Notes in Computer Science, Vol. 6174)*, Tayssir Touili, Byron Cook, and Paul B. Jackson (Eds.). Springer, 243–257. https://doi.org/10.1007/978-3-642-14295-6_24

[39] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael S. Bernstein, Alexander C. Berg, and Fei-Fei Li. 2015. ImageNet Large Scale Visual Recognition Challenge. *Int. J. Comput. Vis.* 115, 3 (2015), 211–252. https://doi.org/10.1007/s11263-015-0816-y

[40] Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin T. Vechev. 2018. Fast and effective robustness certification. In *Proceedings of the 31st Annual Conference on Neural Information Processing Systems (NeurIPS'18)*, Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (Eds.). 10825–10836.

[41] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin T. Vechev. 2019. An abstract domain for certifying neural networks. *Proc. ACM Program. Lang.* 3, POPL (2019), 41:1–41:30. https://doi.org/10.1145/3290354

[42] Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. 2018. Concolic testing for deep neural networks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 109–119.

[43] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. 2014. Intriguing properties of neural networks. In *Proceedings of the 2nd International Conference on Learning Representations*.

[44] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering*. 303–314.

[45] Vincent Tjeng and Russ Tedrake. 2017. Verifying neural networks with mixed integer programming. Retrieved from http://arxiv.org/abs/1711.07356

[46] Hoang-Dung Tran, Diego Manzanas Lopez, Patrick Musau, Xiaodong Yang, Luan Viet Nguyen, Weiming Xiang, and Taylor T. Johnson. 2019. Star-based reachability analysis of deep neural networks. In *Proceedings of the 3rd World Congress on Formal Methods—The Next 30 Years (FM'19) (Lecture Notes in Computer Science, Vol. 11800)*, Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira (Eds.). Springer, 670–686. https://doi.org/10.1007/978-3-030-30942-8_39

[47] Caterina Urban and Antoine Miné. 2021. A review of formal methods applied to machine learning. Retrieved from https://arXiv:2104.02466

[48] Chris Urmson and William Whittaker. 2008. Self-driving cars and the urban challenge. *IEEE Intell. Syst.* 23, 2 (2008), 66–68. https://doi.org/10.1109/MIS.2008.34

[49] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. 2018. Formal security analysis of neural networks using symbolic intervals. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security'18)*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 1599–1614. Retrieved from https://www.usenix.org/conference/usenixsecurity18/presentation/wang-shiqi

[50] Wikipedia. 2022. Activation Function. Retrieved from https://en.wikipedia.org/wiki/Activation_function

[51] Eric Wong and J. Zico Kolter. 2018. Provable defenses against adversarial examples via the convex outer adversarial polytope. In *Proceedings of the 35th International Conference on Machine Learning (ICML'18) (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer G. Dy and Andreas Krause (Eds.). PMLR, 5283–5292. Retrieved from http://proceedings.mlr.press/v80/wong18a.html

[52] Pengfei Yang, Renjue Li, Jianlin Li, Cheng-Chao Huang, Jingyi Wang, Jun Sun, Bai Xue, and Lijun Zhang. 2021. Improving neural network verification through spurious region guided refinement. In *Proceedings of the 27th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'21), Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS'21) (Lecture Notes in Computer Science, Vol. 12651)*, Jan Friso Groote and Kim Guldstrand Larsen (Eds.). Springer, 389–408. https://doi.org/10.1007/978-3-030-72016-2_21

[53] 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 *Proceedings of the Annual Conference on Advances in Neural Information Processing Systems*.

[54] Jie M. Zhang, Mark Harman, Lei Ma, and Yang Liu. 2022. Machine learning testing: Survey, landscapes and horizons. *IEEE Trans. Software Eng.* 48, 2 (2022), 1–36.

[55] Yedi Zhang, Zhe Zhao, Guangke Chen, Fu Song, and Taolue Chen. 2021. BDD4BNN: A BDD-based quantitative analysis framework for binarized neural networks. In *Proceedings of the 33rd International Conference on Computer Aided Verification.* 175–200.

[56] Yedi Zhang, Zhe Zhao, Guangke Chen, Fu Song, Min Zhang, Taolue Chen, and Jun Sun. 2022. QVIP: An ILP-based formal verification approach for quantized neural networks. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering.* 82:1–82:13.

Received 6 March 2023; revised 8 October 2023; accepted 11 January 2024