Abstract Neural Networks

Matthew Sotoudeh(✉) and Aditya V. Thakur

University of California, Davis, USA
{masotoudeh,avthakur}@ucdavis.edu

Abstract. Deep Neural Networks (DNNs) are rapidly being applied to safety-critical domains such as drone and airplane control, motivating techniques for verifying the safety of their behavior. Unfortunately, DNN verification is NP-hard, with current algorithms slowing exponentially with the number of nodes in the DNN. This paper introduces the notion of Abstract Neural Networks (ANNs), which can be used to soundly overapproximate DNNs while using fewer nodes. An ANN is like a DNN except weight matrices are replaced by values in a given abstract domain. We present a framework parameterized by the abstract domain and activation functions used in the DNN that can be used to construct a corresponding ANN. We present necessary and sufficient conditions on the DNN activation functions for the constructed ANN to soundly over-approximate the given DNN. Prior work on DNN abstraction was restricted to the interval domain and ReLU activation function. Our framework can be instantiated with other abstract domains such as octagons and polyhedra, as well as other activation functions such as Leaky ReLU, Sigmoid, and Hyperbolic Tangent.

Keywords: Deep Neural Networks · Abstraction · Soundness

1 Introduction

Deep Neural Networks (DNNs), defined formally in Sect. 3, are loop-free computer programs organized into *layers*, each of which computes a linear combination of the layer's inputs, then applies some *non-linear activation function* to the resulting values. The activation function used varies between networks, with popular activation functions including ReLU, Hyperbolic Tangent, and Leaky ReLU [13]. DNNs have rapidly become important in a variety of applications, including image recognition and safety-critical control systems, motivating research into the problem of verifying properties about their behavior [9,18].

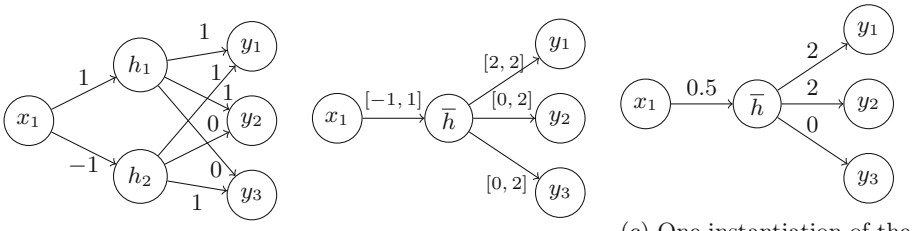
Although they lack loops, the use of non-linear activation functions introduces *exponential branching behavior* into the DNN semantics. It has been shown that DNN verification is NP-hard [18]. In particular, this exponential behavior scales with the number of *nodes* in a network. DNNs in practice have very large numbers of nodes, e.g., the aircraft collision-avoidance DNN ACAS Xu [17] has 300 and a modern image recognition network has tens of thousands [20]. The number of nodes in modern networks has also been growing with time as more effective training methods have been found [3].

D. Pichardie and M. Sighireanu (Eds.): SAS 2020, LNCS 12389, pp. 65–88, 2020.
https://doi.org/10.1007/978-3-030-65474-0_4

One increasingly common way of addressing this problem is to compress the DNN into a smaller proxy network which can be analyzed in its place. However, most such approaches usually do not guarantee that properties of the proxy network hold in the original network (they are unsound). Recently, Prabhakar et al. [29] introduced the notion of *Interval Neural Networks* (INNs), which can produce a smaller proxy network that is *guaranteed* to over-approximate the behavior of the original DNN. While promising, soundness is only guaranteed with a particular activation function (ReLU) and abstract domain (intervals).

In this work, we introduce *Abstract Neural Networks* (ANNs), which are like DNNs except weight matrices are replaced with values in an abstract domain. Given a DNN and an abstract domain, we present an algorithm for constructing a corresponding ANN with fewer nodes. The algorithm works by merging groups of nodes in the DNN to form corresponding abstract nodes in the ANN. We prove necessary and sufficient conditions on the activation functions used for the constructed ANN to over-approximate the input DNN. If these conditions are met, the smaller ANN can be soundly analyzed in place of the DNN. Our formalization and theoretical results generalize those of Prabhakar et al. [29], which are an instantiation of our framework for ReLU activation functions and the interval domain. Our results also show how to instantiate the algorithm such that sound abstraction can be achieved with a variety of different abstract domains (including polytopes and octagons) as well as many popular activation functions (including Hyperbolic Tangent, Leaky ReLU, and Sigmoid).

Outline. In this paper, we aim to lay strong theoretical foundations for research into abstracting neural networks for verification. Section 2 gives an overview of our technique. Section 3 defines preliminaries. Section 4 defines *Abstract Neural Networks* (ANNs). Section 5 presents an algorithm for constructing an ANN from a given DNN. Section 6 motivates our theoretical results with a number of examples. Section 7 proves our soundness theorem. Section 8 discusses related work, while Sect. 9 concludes with a discussion of future work. Code implementing our framework can be found at https://doi.org/10.5281/zenodo.4031610. Detailed proofs of all theorems are in the extended version of this paper [34].



(a) DNN N_1 (b) Corresponding INN (c) One instantiation of the INN

Fig. 1. Example DNN to INN and one of many instantiations of the INN.

2 Motivation

DNNs are often denoted by a graph of the form shown in Fig. 1a. The input node x_1 is assigned the *input value*, then the values of h_1 and h_2 are computed by first a linear combination of the values of the previous layer (in this case x_1) followed by some *non-linear activation function.* The behavior of the network is dependent on the non-linear activation function used. We will assume that the output layer with nodes y_1, y_2, and y_3 uses the identity activation function $I(x) = x$. For the hidden layer with nodes h_1 and h_2 we will consider two scenarios, each using one of the following two activation functions:

$$\sigma(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{otherwise.} \end{cases} \qquad \phi(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0.5x & \text{otherwise.} \end{cases}.$$

Using σ as the activation function for the hidden layer, when $x_1 = 1$ we have $h_1 = \sigma(1x_1) = 1$ and $h_2 = \sigma(-1x_1) = 0$. That in turn gives us $y_1 = I(1h_1 + 1h_2) = 1$, $y_2 = I(1h_1 + 0h_2) = 1$, and $y_3 = I(0h_1 + 1h_2) = 0$.

Using σ as the activation function for the hidden layer, when $x_1 = 1$, we have

$$\begin{array}{lll} h_1 = \sigma(1x_1) = 1 & h_2 = \sigma(-1x_1) = 0 & \\ y_1 = I(1h_1 + 1h_2) = 1 & y_2 = I(1h_1 + 0h_2) = 1 & y_3 = I(0h_1 + 1h_2) = 0. \end{array}$$

Using ϕ as the activation function for the hidden layer, when $x_1 = 1$, we have

$$\begin{array}{lll} h_1 = \phi(1) = 1 & h_2 = \phi(-1) = -0.5 & \\ y_1 = 0.5 & y_2 = 1 & y_3 = -0.5. \end{array}$$

2.1 Merging Nodes

Our goal is to *merge* nodes and their corresponding weights in this DNN to produce a smaller network that over-approximates the behavior of the original one. One way of doing this was proposed by Prabhakar et al. [29], where nodes within a layer can be merged and the *weighted interval hull* of their edge weights is taken. For example, if we merge all of the h_i nodes together into a single $\overline{h}$ node, this process results in an *Interval Neural Network* (INN) shown in Fig. 1b.

Intuitively, given this new INN we can form a *DNN instantiation* by picking any weight within the interval for each edge. We can then find the output of this DNN instantiation on, say, $x_1 = 1$. We take the *output of the INN* on an input x_1 to be the set of *all* such (y_1, y_2, y_3) triples outputted by some such instantiated DNN on x_1.

For example, we can take the instantiation in Fig. 1c. Using the σ activation function, this implies $(y_1 = 1, y_2 = 1, y_3 = 0)$ is in the output set of the INN on input $x_1 = 1$. In fact, the results of Prabhakar et al. [29] show that, if the σ activation function is used, then for *any* input x_1 we will have some assignment to the weights which produces the same output as the original DNN (although

many assignments will produce different outputs—the output set is an *over-approximation* of the behavior of the original network).

However, something different happens if the network were using the ϕ activation function, a case that was not considered by Prabhakar et al. [29]. In that scenario, the original DNN had an output of $(0.5, 1, -0.5)$, so if the INN were to soundly over-approximate it there would need to be some instantiation of the weights where y_1 and y_3 could have opposite signs. But this cannot happen—both will have the same (or zero) sign as $\overline{h}$!

These examples highlight the fact that the soundness of the algorithm from Prabhakar et al. [29] is specific to the ReLU activation function (σ above) and Interval abstract domain. Their results make no statement about whether INNs over-approximate DNNs using different activation functions (such as ϕ above), or if abstractions using different domains (such as the *Octagon* Neural Networks defined in Definition 11) also permit sound DNN over-approximation.

This paper develops a general framework for such DNN abstractions, parameterized by the abstract domain and activation functions used. In this framework, we prove *necessary and sufficient* conditions on the activation functions for a *Layer-Wise Abstraction Algorithm* generalizing that of Prabhakar et al. [29] to produce an ANN soundly over-approximating the given DNN. Finally, we discuss ways to modify the abstraction algorithm in order to soundly over-approximate common DNN architectures that fail the necessary conditions, extending the applicability of model abstraction to almost all currently-used DNNs.

These results lay a solid theoretical foundation for research into Abstract Neural Networks. Because our algorithm and proofs are parameterized by the abstract domain and activation functions used, our proofs allow practitioners to experiment with different abstractions, activation functions, and optimizations without having to re-prove soundness for their particular instantiation (which, as we will see in Sect. 7, is a surprisingly subtle process).

3 Preliminaries

In this section we define Deep Neural Networks and a number of commonly-used activation functions.

3.1 Deep Neural Networks

In Sect. 2, we represented neural networks by *graphs.* While this is useful for intuition, in Sect. 4 we will talk about, e.g., *octagons of layer weight matrices*, for which the graph representation makes significantly less intuitive sense. Hence, for the rest of the paper we will use an entirely equivalent *matrix representation* for DNNs, which will simplify the definitions, intuition, and proofs considerably. With this notation, we think of nodes as *dimensions* and layers of nodes as *intermediate spaces.* We then define a *layer* to be a transformation from one intermediate space to another.

Definition 1. *A* DNN layer from n to m dimensions *is a tuple* (W, σ) *where* W *is an* $m \times n$ *matrix and* $\sigma : \mathbb{R} \to \mathbb{R}$ *is an arbitrarily-chosen* activation function.

We will often abuse notation such that, for a vector $\boldsymbol{v}$, $\sigma(\boldsymbol{v})$ is the vector formed by applying σ to each component of $\boldsymbol{v}$.

Definition 2. *A* Deep Neural Network (DNN) with layer sizes $s_0, s_1, \ldots, s_n$ *is a collection of* n *DNN layers* $(W^{(1)}, \sigma^{(1)}), \ldots, (W^{(n)}, \sigma^{(n)})$*, where the* $(W^{(i)}, \sigma^{(i)})$ *layer is from* s_{i-1} *to* s_i *dimensions.*

Every DNN has a corresponding *function,* defined below.

Definition 3. *Given a DNN from* s_0 *to* s_n *dimensions with layers* $(W^{(i)}, \sigma^{(i)})$*, the* function corresponding to the DNN *is the function* $f : \mathbb{R}^{s_0} \to \mathbb{R}^{s_n}$ *given by* $f(\boldsymbol{v}) = \boldsymbol{v}^{(n)}$*, where* $\boldsymbol{v}^{(i)}$ *is defined inductively by* $\boldsymbol{v}^{(0)} = \boldsymbol{v}$ *and* $\boldsymbol{v}^{(i)} = \sigma^{(i)}(W^{(i)}(\boldsymbol{v}^{(i-1)}))$.

Where convenient, we will often refer to the corresponding function as the DNN or vice-versa.

Example 1. The DNN N_1 from Fig. 1a, when using the σ hidden-layer activation function, is represented by the layers $\left(\begin{bmatrix} 1 \\ -1 \end{bmatrix}, \sigma\right)$ and $\left(\begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}, I\right)$. The *function corresponding to the DNN* is given by $N_1(x_1) = \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \sigma\left(\begin{bmatrix} 1 \\ -1 \end{bmatrix} [x_1]\right)$.

3.2 Common Activation Functions

There are a number of commonly-used activation functions, listed below.

Definition 4. *The* Leaky Rectified Linear Unit *(LReLU) [22],* Rectified Linear Unit *(ReLU),* Hyperbolic Tangent *(*tanh*), and* Threshold *(*thresh*) activation functions are defined:*

$$\mathrm{LReLU}(x; \mathtt{c}) \coloneqq \begin{cases} x & x \geq 0 \\ \mathtt{c}x & x < 0 \end{cases}, \qquad \mathrm{ReLU}(x) \coloneqq \mathrm{LReLU}(x; 0),$$

$$\tanh \coloneqq \frac{e^{2x} - 1}{e^{2x} + 1}, \qquad \mathrm{thresh}(x; \mathtt{t}, \mathtt{v}) \coloneqq \begin{cases} x & \text{if } x \geq \mathtt{t} \\ \mathtt{v} & \text{otherwise} \end{cases}.$$

Here LReLU *and* thresh *actually represent* families *of activation functions parameterized by the constants* $\mathtt{c}, \mathtt{t}, \mathtt{v}$*. The constants used varies between networks.* $\mathtt{c} = 0$ *is a common choice for the* LReLU *parameter, hence the explicit definition of* ReLU*.*

All of these activation functions are present in standard deep-learning toolkits, such as Pytorch [26]. Libraries such as Pytorch also enable users to implement new activation functions. This variety of activation functions used in practice will motivate our study of necessary and sufficient conditions on the activation function to permit sound over-approximation.

4 Abstract Neural Networks

In this section, we formalize the syntax and semantics of Abstract Neural Networks (ANNs). We also present two types of ANNs: Interval Neural Networks (INNs) and Octagon Neural Networks (ONNs).

An ANN is like a DNN except the weights in each layer are represented by an abstract value in some abstract domain. This is formalized below.

Definition 5. *An $n \times m$* weight set abstract domain *is a lattice $\mathcal{A}$ with Galois connection $(\alpha_{\mathcal{A}}, \gamma_{\mathcal{A}})$ with the powerset lattice $\mathcal{P}(\mathbb{R}^{n \times m})$ of $n \times m$ matrices.*

Definition 6. *An* ANN layer from n to m dimensions *is a triple $(\mathcal{A}, A, \sigma)$ where A is a member of the weight set abstraction $\mathcal{A}$ and $\sigma : \mathbb{R} \to \mathbb{R}$ is an arbitrarily-chosen* activation function.

Thus, we see that each ANN layer $(\mathcal{A}, A, \sigma)$ is associated with a set of weights $\gamma_{\mathcal{A}}(A)$. Finally, we can define the notion of an ANN:

Definition 7. *An* Abstract Neural Network (ANN) with layer sizes $s_0, s_1, \ldots, s_n$ *is a collection of n ANN layers $(\mathcal{A}^{(i)}, A^{(i)}, \sigma^{(i)})$, where the ith layer is from s_{i-1} to s_i dimensions.*

We consider the output of the ANN to be the set of outputs of all *instantiations* of the ANN into a DNN, as illustrated in Fig. 2.

Definition 8. *We say a DNN with layers $(W^{(i)}, \sigma^{(i)})$ is* an instantiation of *an ANN T with layers $(\mathcal{A}^{(i)}, A^{(i)}, \sigma^{(i)})$ if each $W^{(i)} \in \gamma_{\mathcal{A}^{(i)}}(A^{(i)})$. The set of all DNNs that are instantiations of an ANN T is given by $\gamma(T)$.*

The semantics of an ANN naturally lift those of the DNN instantiations.

Definition 9. *For an ANN T from s_0 to s_n dimensions, the* function corresponding to *T is the set-valued function $T : \mathbb{R}^{s_0} \to \mathcal{P}(\mathbb{R}^{s_n})$ defined by $T(\boldsymbol{v}) \coloneqq \{g(\boldsymbol{v}) \mid g \in \gamma(T)\}$.*

Space constraints prevent us from defining a full Galois connection here, however one can be established between the lattice of ANNs of a certain *architecture* and the powerset of DNNs of the same architecture.

The definition of an ANN above is agnostic to the actual abstract domain(s) used. For expository purposes, we now define two particular types of ANNs: *Interval Neural Networks* (INNs) and *Octagon Neural Networks* (ONNs).

Definition 10. *An* Interval Neural Network (INN) *is an ANN with layers $(\mathcal{A}^{(i)}, A^{(i)}, \sigma^{(i)})$, where each $\mathcal{A}^{(i)}$ is an* interval hull domain *[5]. The interval hull domain represents sets of matrices by their* component-wise interval hull.

Notably, the definition of INN in Prabhakar et al. [29] is equivalent to the above, except that they further assume every activation function $\sigma^{(i)}$ is the ReLU function.

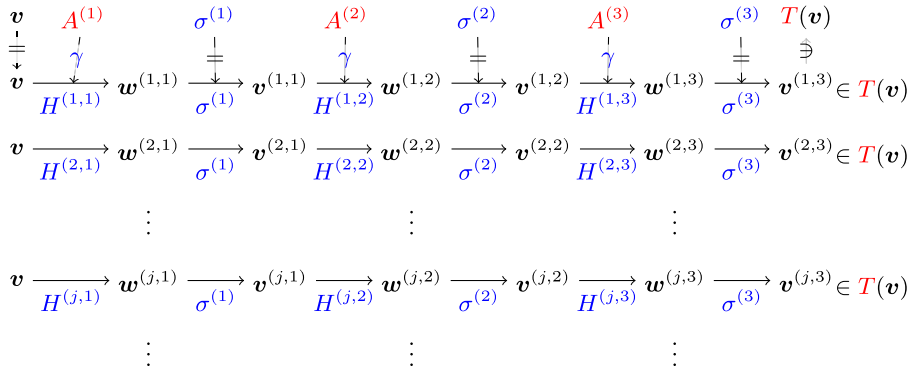


Fig. 2. Visualization of ANN semantics for a 3-layer ANN T (first row). Different DNN *instantiations* (other rows) of T are formed by replacing each abstract weight matrix $A^{(i)}$ by some concrete weight matrix $H^{(j,i)} \in \gamma(A^{(i)})$. $\boldsymbol{v}^{(j,3)}$ is the output of each instantiation on the input $\boldsymbol{v}$. The set of all such outputs producable by some valid instantiation is taken to be the output $T(\boldsymbol{v})$ of the ANN on vector $\boldsymbol{v}$.

Example 2. We first demonstrate the interval hull domain: $\gamma_{\text{Int}}\left(\begin{bmatrix}[-1,1] & [0,2]\\ [-3,-2] & [1,2]\end{bmatrix}\right) = \left\{\begin{bmatrix}a & b\\ c & d\end{bmatrix} \mid a \in [-1,1], b \in [0,2], c \in [-3,-2], d \in [1,2]\right\}$. We can thus define a two-layer INN $f(\boldsymbol{v}) \coloneqq \begin{bmatrix}[0,1] & [0,1]\end{bmatrix} \text{ReLU}\left(\begin{bmatrix}[-1,1] & [0,2]\\ [-3,-2] & [1,2]\end{bmatrix}\boldsymbol{v}\right)$. We can instantiate this network in a variety of ways, for example $g(\boldsymbol{v}) \coloneqq \begin{bmatrix}0.5 & 1\end{bmatrix} \text{ReLU}\left(\begin{bmatrix}0 & 2\\ -2.5 & 1.5\end{bmatrix}\boldsymbol{v}\right) \in \gamma(f)$. Taking arbitrarily $(1,1)^T$ as an example input, we have $g((1,1)^T) = \begin{bmatrix}1\end{bmatrix} \in f((1,1)^T)$. In fact, $f((1,1)^T)$ is the set of *all* values that can be achieved by such instantiations, which in this case is the set given by $f((1,1)^T) = \begin{bmatrix}[0,3]\end{bmatrix}$.

Definition 11. *An* Octagon Neural Network (ONN) *is an ANN with layers* $(\mathcal{A}^{(i)}, A^{(i)}, \sigma^{(i)})$*, where each* $\mathcal{A}^{(i)}$ *is an* octagon hull domain *[23]. The octagon hull domain represents sets of matrices by octagons in the space of their components.*

Example 3. Octagons representing a set of $n \times m$ matrices can be thought of exactly like an octagon in the vector space $\mathbb{R}^{n \cdot m}$. Unfortunately, this is particularly difficult to visualize in higher dimensions, hence in this example we will stick to the case where $nm = 2$.

Let O_1, O_2 be octagons such that

$$\gamma_{\texttt{Oct}}(O_1) = \left\{\begin{bmatrix}a\\ b\end{bmatrix} \mid a - b \leq 1, -a + b \leq 1, a + b \leq 2, -a - b \leq 2\right\},$$
$$\gamma_{\texttt{Oct}}(O_2) = \left\{\begin{bmatrix}a & b\end{bmatrix} \mid a - b \leq 2, -a + b \leq 3, a + b \leq 4, -a - b \leq 5\right\}.$$

We can thus define a two-layer ONN $f(\boldsymbol{v}) \coloneqq O_2 \text{ReLU}\,(O_1 \boldsymbol{v})$. One instantiation of this ONN f is the DNN $g(\boldsymbol{v}) \coloneqq \begin{bmatrix}3 & 1\end{bmatrix} \text{ReLU}\left(\begin{bmatrix}0.5\\1.5\end{bmatrix}\boldsymbol{v}\right) \in \gamma(f)$. We can confirm that $g(1) = \begin{bmatrix}3\end{bmatrix} \in f(1)$.

We can similarly define Polyhedra Neural Networks (PNNs) using the polyhedra domain [6].

5 Layer-Wise Abstraction Algorithm

Given a large DNN, how might we construct a smaller ANN which soundly *over-approximates* that DNN? We define over-approximation formally below.

Definition 12. *An ANN T* over-approximates *a DNN N if, for every $\boldsymbol{v} \in \mathbb{R}^n$, $N(\boldsymbol{v}) \in T(\boldsymbol{v})$.*

Remark 1. By Definition 9, then, T over-approximates N if, for every $\boldsymbol{v}$ we can find some instantiation $T_{\boldsymbol{v}} \in \gamma(T)$ such that $T_{\boldsymbol{v}}(\boldsymbol{v}) = N(\boldsymbol{v})$.

Algorithm 3 constructs a small ANN that, under certain assumptions discussed in Sect. 2, soundly over-approximates the large DNN given. The basic idea is to *merge* groups of dimensions together, forming an ANN where each dimension in the ANN represents a collection of dimensions in the original DNN. We formalize the notion of "groups of dimensions" as a *layer-wise partitioning.*

Definition 13. *Given a DNN with layer sizes $s_0, s_1, \ldots, s_n$, a* layer-wise partitioning $\mathbb{P}$ of the network *is a set of partitionings $\mathbb{P}^{(0)}, \mathbb{P}^{(1)}, \ldots, \mathbb{P}^{(n)}$ where each $\mathbb{P}^{(i)}$ partitions $\{1, 2, \ldots, s_i\}$. For ease of notation, we will write partitionings with set notation but assume they have some intrinsic ordering for indexing.*

Remark 2. To maintain the same number of input and output dimensions in our ANN and DNN, we assume $\mathbb{P}^{(0)} = \{\{1\}, \{2\}, \ldots, \{s_0\}\}$ and $\mathbb{P}^{(n)} = \{\{1\}, \{2\}, \ldots, \{s_n\}\}$.

Example 4. Consider the DNN corresponding to the function $f(x_1) = \begin{bmatrix}1 & 1\\1 & 0\\0 & 1\end{bmatrix} \text{ReLU}\left(\begin{bmatrix}1\\-1\end{bmatrix}[x_1]\right)$. The layer sizes are $s_0 = 1, s_1 = 2, s_2 = 3$. Hence, one valid layer-wise partitioning is to merge the two inner dimensions: $\mathbb{P}^{(0)} = \{\{1\}\}$ $\mathbb{P}^{(1)} = \{\{1, 2\}\}$ $\mathbb{P}^{(2)} = \{\{1\}, \{2\}, \{3\}\}$. Here we have, e.g., $\mathbb{P}_1^{(0)} = \{1\}$, $\mathbb{P}_1^{(1)} = \{1, 2\}$, and $\mathbb{P}_3^{(2)} = \{3\}$.

Algorithm 1: $\widehat{\alpha}(M, \mathcal{P}^{in}, \mathcal{P}^{out}, \mathcal{A})$

Input: Matrix M. Partitionings $\mathcal{P}^{in}$, $\mathcal{P}^{out}$ with $|\mathcal{P}^{in}| = k$. Abstract domain $\mathcal{A}$.
Output: Abstract element representing all merges of M.

$S \leftarrow \{\}$
$w \leftarrow (|\mathcal{P}_1^{in}|, |\mathcal{P}_2^{in}|, \ldots, |\mathcal{P}_k^{in}|)$
for $C \in \text{PCMs}(\mathcal{P}^{in})$ **do**
  for $D \in \text{PCMs}(\mathcal{P}^{out})$ **do**
    $S \leftarrow S \cup \{\text{ScaleCols}(D^T M C, w))\}$
return $\alpha_{\mathcal{A}}(S)$

Algorithm 2: $\widehat{\alpha}_{bin}(M, \mathcal{P}^{in}, \mathcal{P}^{out}, \mathcal{A})$

Input: Matrix M. Partitionings $\mathcal{P}^{in}$, $\mathcal{P}^{out}$ with $|\mathcal{P}^{in}| = k$. Abstract domain $\mathcal{A}$.
Output: Abstract element representing all binary merges of M

$S \leftarrow \{\}$
$w \leftarrow (|\mathcal{P}_1^{in}|, |\mathcal{P}_2^{in}|, \ldots, |\mathcal{P}_k^{in}|)$
for $C \in \text{BinPCMs}(\mathcal{P}^{in})$ **do**
  for $D \in \text{BinPCMs}(\mathcal{P}^{out})$ **do**
    $S \leftarrow S \cup \{\text{ScaleCols}(D^T M C, w))\}$
return $\alpha_{\mathcal{A}}(S)$

Algorithm 3: AbstractLayerWise$\langle \mathfrak{A}, \Sigma \rangle (N, \mathbb{P}, \mathcal{A})$

Input: DNN N consisting of n layers $(W^{(i)}, \sigma^{(i)})$ with each $\sigma^{(i)} \in \Sigma$. Layer-wise partitioning $\mathbb{P}$ of N. List of n abstract weight domains $\mathcal{A}^{(i)} \in \mathfrak{A}$.
Output: An ANN with layers $(\mathcal{A}^{(i)}, A^{(i)}, \sigma^{(i)})$ where $A^{(i)} \in \mathcal{A}^{(i)} \in \mathfrak{A}$.

$A \leftarrow [\ ]$
for $i \in \{1, 2, \ldots, n\}$ **do**
  $A^{(i)} \leftarrow \widehat{\alpha}(W^{(i)}, \mathbb{P}^{(i-1)}, \mathbb{P}^{(i)}, \mathcal{A}^{(i)})$
  A.append$\big((\mathcal{A}^{(i)}, A^{(i)}, \sigma^{(i)})\big)$
return A

Our layer-wise abstraction algorithm is shown in Algorithm 3. For each layer in the DNN, we will call Algorithm 1 to abstract the set of *mergings* of the layer's weight matrix. This abstract element becomes the abstract weight $A^{(i)}$ for the corresponding layer in the constructed ANN.

The functions PCMs and ScaleCols are defined more precisely below.

Definition 14. *Let P be some partition, i.e., non-empty subset, of $\{1, 2, \ldots, n\}$. Then a vector $\boldsymbol{c} \in \mathbb{R}^n$ is a* partition combination vector *(PCV) if (i) each component c_i is non-negative, (ii) the components of c_i sum to one, and (iii) $c_i = 0$ whenever $i \notin P$.*

Definition 15. *Given a partitioning $\mathcal{P}$ of $\{1, 2, \ldots, n\}$ with $|\mathcal{P}| = k$, a* partitioning combination matrix *(PCM) is a matrix $C = \begin{bmatrix} | & | & & | \\ \boldsymbol{c_1} & \boldsymbol{c_2} & \cdots & \boldsymbol{c_k} \\ | & | & & | \end{bmatrix}$, where each $\boldsymbol{c_i}$ is a PCV of partition $\mathcal{P}_i$. We refer to the set of all such PCMs for a partitioning $\mathcal{P}$ by* PCMs$(\mathcal{P})$.

Definition 16. *A PCM is* binary *if each entry is either 0 or 1. We refer to the set of all binary PCMs for a partitioning $\mathcal{P}$ as* $\text{BinPCMs}(\mathcal{P})$.

Definition 17. *For an $n \times m$ matrix M, PCM C of partitioning $\mathcal{P}^{in}$ of $\{1, 2, \ldots, m\}$, and PCM D for partitioning $\mathcal{P}^{out}$ of $\{1, 2, \ldots, n\}$, we call $D^T MC$ a* merging of M.

The jth column in MC is a convex combination of the columns of M that belong to partition $\mathcal{P}^{in}_j$, weighted by the jth column of C. Similarly, the ith row in $D^T M$ is a convex combination of the rows in M that belong to partition $\mathcal{P}^{out}_i$. In total, the i, jth entry of merged matrix $D^T MC$ is a convex combination of the entries of M with indices in $\mathcal{P}^{out}_i \times \mathcal{P}^{in}_j$. This observation will lead to Theorem 1 in Sect. 5.1.

Definition 18. *Given a matrix M, the* column-scaled matrix *formed by weights $w_1, w_2, \ldots, w_k$ is the matrix with entries given component-wise by* $\text{ScaleCols}(M, (w_1, \ldots, w_k))_{i,j} \coloneqq M_{i,j} w_j$.

Intuitively, column-scaling is needed because what were originally n dimensions contributing to an input have been collapsed into a single representative dimension. This is demonstrated nicely for the specific case of Interval Neural Network and ReLU activations by Figs. 3 and 4 in Prabhakar et al. [29].

Example 5. Given the matrix $M = \begin{bmatrix} m_{1,1} & m_{1,2} & m_{1,3} \\ m_{2,1} & m_{2,2} & m_{2,3} \\ m_{3,1} & m_{3,2} & m_{3,3} \\ m_{4,1} & m_{4,2} & m_{4,3} \end{bmatrix}$, partitioning $\mathbb{P}^{(0)} = \{\{1, 3\}, \{2\}\}$ of the input dimensions and $\mathbb{P}^{(1)} = \{\{2, 4\}, \{1, 3\}\}$ of the output dimensions, we can define a PCM for $\mathbb{P}^{(0)}$ as $C \coloneqq \begin{bmatrix} 0.25 & 0 \\ 0 & 1 \\ 0.75 & 0 \end{bmatrix}$ and a PCM for $\mathbb{P}^{(1)}$ as: $D \coloneqq \begin{bmatrix} 0 & 0.99 \\ 0.4 & 0 \\ 0 & 0.01 \\ 0.6 & 0 \end{bmatrix}$. We can then compute the *column–merged matrix* $MC = \begin{bmatrix} 0.25m_{1,1} + 0.75m_{1,3} & m_{1,2} \\ 0.25m_{2,1} + 0.75m_{2,3} & m_{2,2} \\ 0.25m_{3,1} + 0.75m_{3,3} & m_{3,2} \\ 0.25m_{4,1} + 0.75m_{4,3} & m_{4,2} \end{bmatrix}$, and furthermore the *column-row–merged matrix* $D^T MC = \begin{bmatrix} 0.4(0.25m_{2,1} + 0.75m_{2,3}) + 0.6(0.25m_{4,1} + 0.75m_{4,3}) & 0.4m_{2,2} + 0.6m_{4,2} \\ 0.99(0.25m_{1,1} + 0.75m_{1,3}) + 0.01(0.25m_{3,1} + 0.75m_{3,3}) & 0.99m_{1,2} + 0.01m_{3,2} \end{bmatrix}$.

Finally, we can column-scale this matrix like so:

$$\text{ScaleCols}(D^T MC, (2, 2)) = \begin{bmatrix} 0.8(0.25m_{2,1} + 0.75m_{2,3}) + 1.2(0.25m_{4,1} + 0.75m_{4,3}) & 0.8m_{2,2} + 1.2m_{4,2} \\ 1.98(0.25m_{1,1} + 0.75m_{1,3}) + 0.02(0.25m_{3,1} + 0.75m_{3,3}) & 1.98m_{1,2} + 0.02m_{3,2} \end{bmatrix}.$$

5.1 Computability

In general, there are an infinite number of mergings. Hence, to actually compute $\widehat{\alpha}$ (Algorithm 1) we need some non-trivial way to compute the abstraction of the infinite set of mergings. If the abstract domain $\mathcal{A}^{(i)}$ is *convex*, it can be shown that one only needs to iterate over the binary PCMs, of which there are finitely many, producing a computationally feasible algorithm.

Definition 19. *A weight set abstract domain* $\mathcal{A}$ *is* convex *if, for any set* S *of concrete values,* $\gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(S))$ *is convex.*

Many commonly-used abstractions—including intervals [5], octagons [23], and polyhedra [6]—are convex.

Theorem 1. *If* $\mathcal{A}$ *is convex, then* $\widehat{\alpha}(M, \mathcal{P}^{in}, \mathcal{P}^{out}, \mathcal{A}) = \widehat{\alpha}_{bin}(M, \mathcal{P}^{in}, \mathcal{P}^{out}, \mathcal{A})$.

Remark 3. Consider PCMs C and D corresponding to merged matrix $D^T W^{(i)} C$. We may think of C and D as vectors in the vector space of matrices. Then their outer product $D \otimes C$ forms a convex coefficient matrix of the binary mergings R of $W^{(i)}$, such that $(D \otimes C)R = D^T W^{(i)} C$. From this intuition, it follows that the converse to Theorem 1 *does not* hold, as every matrix E cannot be decomposed into vectors $D \otimes C$ as described (i.e., not every matrix has rank 1). Hence, the convexity condition may be slightly weakened. However, we are not presently aware of any abstract domains that satisfy such a condition but not convexity.

Example 6. Let $W^{(i)} = \begin{bmatrix} 1 & -2 & 3 \\ 4 & -5 & 6 \\ 7 & -8 & 9 \end{bmatrix}$ and consider $\mathbb{P}^{(i-1)} = \{\{1,2\},\{3\}\}$ and $\mathbb{P}^{(i)} = \{\{1,3\},\{2\}\}$. Then we have the binary PCMs $\text{BinPCMs}(\mathbb{P}^{(i-1)}) = \left\{ \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \right\}$ and $\text{BinPCMs}(\mathbb{P}^{(i)}) = \left\{ \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} \right\}$. These correspond to the column-scaled binary mergings $\left\{ \begin{bmatrix} 2 & 3 \\ 8 & 6 \end{bmatrix}, \begin{bmatrix} -4 & 3 \\ -10 & 6 \end{bmatrix}, \begin{bmatrix} 14 & 9 \\ 8 & 6 \end{bmatrix}, \begin{bmatrix} -16 & 9 \\ -10 & 6 \end{bmatrix} \right\}$.

We can take any PCMs such as $C = \begin{bmatrix} 0.75 & 0 \\ 0.25 & 0 \\ 0 & 1 \end{bmatrix}$ for $\mathbb{P}^{(i-1)}$ as well as $D = \begin{bmatrix} 0.5 & 0 \\ 0 & 1 \\ 0.5 & 0 \end{bmatrix}$ for $\mathbb{P}^{(i)}$, resulting in the scaled merging $\text{ScaleCols}(D^T W^{(i)} C, (2,1)) = \begin{bmatrix} 3.5 & 6 \\ 3.5 & 6 \end{bmatrix}$. According to Theorem 1, we can write this as a convex combination of the four column-scaled binary merged matrices. In particular, we find the combination

$$\begin{aligned} \begin{bmatrix} 3.5 & 6 \\ 3.5 & 6 \end{bmatrix} =& (1.5/2)(1)(0.5)(1) \begin{bmatrix} 2 & 3 \\ 8 & 6 \end{bmatrix} + (0.5/2)(1)(0.5)(1) \begin{bmatrix} -4 & 3 \\ -10 & 6 \end{bmatrix} \\ &+ (1.5/2)(1)(0.5)(1) \begin{bmatrix} 14 & 9 \\ 8 & 6 \end{bmatrix} + (0.5/2)(1)(0.5)(1) \begin{bmatrix} -16 & 9 \\ -10 & 6 \end{bmatrix}. \end{aligned}$$

We can confirm that this is a convex combination, as

$$(1.5/2)(1)(0.5)(1) + (0.5/2)(1)(0.5)(1) + (1.5/2)(1)(0.5)(1) + (0.5/2)(1)(0.5)(1) = 1.$$

Because we can find such a convex combination for any such non-binary merging in terms of the binary ones, and because the abstract domain is assumed to be convex, including only the binary mergings will ensure that *all* mergings are represented by the abstract element $A^{(i)}$.

5.2 Walkthrough Example

Example 7. Consider again the DNN from Example 4 corresponding to $f(x_1) = \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \sigma\left(\begin{bmatrix} 1 \\ -1 \end{bmatrix} \begin{bmatrix} x_1 \end{bmatrix} \right)$, the partitioning $\mathbb{P}^{(0)} = \{\{1\}\}, \mathbb{P}^{(1)} = \{\{1,2\}\}, \mathbb{P}^{(2)} =$

$\{\{1\},\{2\},\{3\}\}$, which collapses the two hidden dimensions, and assume the abstract domains $\mathcal{A}^{(i)}$ are all convex.

For the input layer, we have $w = (1)$, because the only partition in $\mathbb{P}^{(0)}$ has size 1. Similarly, the only binary PCM for $\mathbb{P}^{(0)}$ is $C = \begin{bmatrix}1\end{bmatrix}$. However, there are two binary PCMs for $\mathbb{P}^{(1)}$, namely $D = \begin{bmatrix}1\\0\end{bmatrix}$ or $D = \begin{bmatrix}0\\1\end{bmatrix}$. These correspond to the binary merged matrices $\begin{bmatrix}1\end{bmatrix}$ and $\begin{bmatrix}-1\end{bmatrix}$. Hence, we get $A^{(1)} = \alpha_{\mathcal{A}^{(1)}}(\{\begin{bmatrix}1\end{bmatrix}, \begin{bmatrix}-1\end{bmatrix}\})$, completing the first layer.

For the output layer, we have $w = (2)$, because the only partition in $\mathbb{P}^{(1)}$ contains *two* nodes. Hence, the column scaling will need to play a role: because we have merged two dimensions in the domain, we should interpret any value from that dimension as being from *both* of the dimensions that were merged. We have two binary mergings, namely $\begin{bmatrix}1\\1\\0\end{bmatrix}$ and $\begin{bmatrix}1\\0\\1\end{bmatrix}$, which after rescaling gives us $A^{(2)} = \alpha_{\mathcal{A}^{(2)}}\left(\left\{\begin{bmatrix}2\\2\\0\end{bmatrix}, \begin{bmatrix}2\\0\\2\end{bmatrix}\right\}\right)$.

In total then, the returned ANN can be written $(A^{(1)}, \sigma)$, $(A^{(2)}, x \mapsto x)$ or in a more functional notation as $g(x) = A^{(2)}\sigma(A^{(1)}x)$, where in either case $A^{(1)} = \alpha_{\mathcal{A}^{(1)}}(\{\begin{bmatrix}1\end{bmatrix}, \begin{bmatrix}-1\end{bmatrix}\})$, and $A^{(2)} = \alpha_{\mathcal{A}^{(2)}}\left(\left\{\begin{bmatrix}2\\2\\0\end{bmatrix}, \begin{bmatrix}2\\0\\2\end{bmatrix}\right\}\right)$.

Note in particular that, while the operation of the algorithm was agnostic to the exact abstract domains $\mathfrak{A}$ and activation functions Σ used, the semantics of the resulting ANN depend *entirely* on these. Hence, correctness of the algorithm will depend on the abstract domain and activation functions satisfying certain conditions. We will discuss this further in Sect. 6.

6 Layer-Wise Abstraction: Instantiations and Examples

This section examines a number of examples. For some DNNs, Algorithm 3 will produce a soundly over-approximating ANN. For others, the ANN will provably *not* over-approximate the given DNN. We will generalize these examples to necessary and sufficient conditions on the activation functions Σ used in order for AbstractLayerWise$\langle\mathfrak{A}, \Sigma\rangle(N, \mathbb{P}, \mathcal{A})$ to soundly over-approximate N.

6.1 Interval Hull Domain with ReLU Activation Functions

Consider again the DNN from Example 7 given by $f(x_1) = \begin{bmatrix}1 & 1\\1 & 0\\0 & 1\end{bmatrix} \text{ReLU}\left(\begin{bmatrix}1\\-1\end{bmatrix}\begin{bmatrix}x_1\end{bmatrix}\right)$ and partitioning which merges the two intermediate dimensions. Using the interval hull domain in Example 7 gives the corresponding INN: $g(x_1) = \begin{bmatrix}[2,2]\\[0,2]\\[0,2]\end{bmatrix} \text{ReLU}\left(\begin{bmatrix}[-1,1]\end{bmatrix}\begin{bmatrix}x_1\end{bmatrix}\right)$.

In fact, because the ReLU activation function and interval domain was used, it follows from the results of Prabhakar et al. [29] that g in fact over-approximates

f. To see this, consider two cases. If $x_1 > 0$, then the second component in the hidden dimension of f will always become 0 under the activation function. Hence, $f(x_1) = \begin{bmatrix}1\\1\\0\end{bmatrix}\text{ReLU}\left(\begin{bmatrix}1\end{bmatrix}\begin{bmatrix}x_1\end{bmatrix}\right) = \begin{bmatrix}2\\2\\0\end{bmatrix}\text{ReLU}\left(\begin{bmatrix}0.5\end{bmatrix}\begin{bmatrix}x_1\end{bmatrix}\right)$, which is a valid instantiation of the weights in g. Otherwise, if $x_1 \leq 0$, we find $f(x_1) = \begin{bmatrix}2\\0\\2\end{bmatrix}\text{ReLU}\left(\begin{bmatrix}-0.5\end{bmatrix}\begin{bmatrix}x_1\end{bmatrix}\right)$, which is again a valid instantiation. Hence in all cases, the true output $f(x_1)$ can be made by some valid instantiation of the weights in g. Therefore, $f(x_1) \in g(x_1)$ for all x_1 and so g over-approximates f.

Sufficiency Condition. The soundness of this particular instantiation can be generalized to a sufficiency theorem, Theorem 2, for soundness of the layer-wise abstraction algorithm. Its statement relies on the activation function satisfying the *weakened intermediate value property,* which is defined below:

Definition 20. *A function* $f : \mathbb{R} \to \mathbb{R}$ *satisfies the* Weakened Intermediate Value Property (WIVP) *if, for every* $a_1 \leq a_2 \leq \cdots \leq a_n \in \mathbb{R}$*, there exists some* $b \in [a_1, a_n]$ *such that* $f(b) = \frac{\sum_i f(a_i)}{n}$.

Every continuous function satisfies the IVP and hence the WIVP. Almost all commonly-used activation functions, except for thresh, are continuous and, therefore, satisfy the WIVP. However, the WIVP is not equivalent to the IVP, as the below proof shows by constructing a function f such that $f((a,b)) = \mathbb{Q}$ for any non-empty open interval (a,b).

We now state the soundness theorem below, which is proved in Sect. 7.

Theorem 2. *Let* $\mathfrak{A}$ *be a set of weight set abstract domains and* Σ *a set of activation functions. Suppose (i) each* $\sigma \in \Sigma$ *has entirely non-negative outputs, and (ii) each* $\sigma \in \Sigma$ *satisfies the Weakened Intermediate Value Property (Definition 20). Then* $T = \text{AbstractLayerWise}\langle\mathfrak{A}, \Sigma\rangle(N, \mathbb{P}, \mathcal{A})$ *(Algorithm 3) soundly over-approximates the DNN* N.

6.2 Interval Hull Domain with *Leaky* ReLUs

Something different happens if we slightly modify f in Example 7 to use an activation function producing *negative values* in the intermediate dimensions. This is quite common of activation functions like Leaky ReLU and tanh, and was not mentioned by Prabhakar et al. [29]. For example, we will take the Leaky ReLU function (Definition 4) with $\mathtt{c} = 0.5$ and consider the DNN $f(x_1) = \begin{bmatrix}1 & 1\\1 & 0\\0 & 1\end{bmatrix}\text{LReLU}\left(\begin{bmatrix}1\\-1\end{bmatrix}\begin{bmatrix}x_1\end{bmatrix}; 0.5\right)$. Using the same partitioning gives us the INN $g(x_1) = \begin{bmatrix}[2,2]\\[0,2]\\[0,2]\end{bmatrix}\text{LReLU}\left(\begin{bmatrix}[-1,1]\end{bmatrix}\begin{bmatrix}x_1\end{bmatrix}; 0.5\right)$.

Surprisingly, this small change to the activation function in fact makes the constructed ANN no longer over-approximate the original DNN. For example, note that $f(1) = \begin{bmatrix}0.5 & 1 & -0.5\end{bmatrix}^T$ and consider $g(1)$. In g, the output of the LReLU

is one-dimensional, hence, it will have either positive, negative, or zero sign. But no matter how the weights in the final matrix are instantiated, every component of $g(1)$ will have *the same (or zero) sign*, and so $f(1) \notin g(1)$, because $f(1)$ has mixed signs.

Necessary Condition: Non-negative Values. We can generalize this counterexample to the following necessary condition on soundness:

Theorem 3. *Suppose some $\sigma \in \Sigma$ is an activation function with neither entirely non-negative nor entirely non-positive outputs, and every $\mathcal{A} \in \mathfrak{A}$ is at least as precise as the interval hull abstraction. Then there exists a neural network N that uses σ and a partitioning $\mathbb{P}$ such that $T = \text{AbstractLayerWise}\langle\mathfrak{A}, \Sigma\rangle(N, \mathbb{P}, \mathcal{A})$ does not over-approximate N.*

Handling Negative Values. Thankfully, there is a workaround to support sometimes-negative activation functions. The constructive theorem below implies that a given DNN can be modified into a *shifted* version of itself such that the input-output behavior on any arbitrary bounded region is retained, but the intermediate activations are all non-negative.

Theorem 4. *Let N be a DNN and suppose that, on some input region R, the output of the activation functions are lower-bounded by a constant C. Then, there exists another DNN N', with at most one extra dimension per layer, which satisfies (i) $N'(x) = N(x)$ for any $x \in R$, (ii) N' has all non-negative activation functions, and (iii) the new activation functions σ' are of the form $\sigma'(x) = \max(\sigma(x) + |C|, 0)$.*

Notably, the proof of this theorem is *constructive* with a straightforward construction. The one requirement is that a lower-bound C be provided for the output of the nodes in the network. This lower-bound need not be tight, and can be computed quickly using the same procedure discussed for upper bounds immediately following Eq. 1 in Prabhakar et al. [29]. For tanh in particular, its output is always lower-bounded by -1 so we can immediately take $C = -1$ for a network using only tanh activations.

6.3 Interval Hull Abstraction with Non-continuous Functions

Another way that the constructed ANN may not over-approximate the DNN is if the activation function does not satisfy the Weakened Intermediate Value Property (WIVP) (Definition 20). For example, consider the threshold activation function (Definition 4) with parameters $\mathtt{t} = 1$, $\mathtt{v} = 0$ and the same overall network, i.e. $f(x_1) = \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \text{thresh}\left(\begin{bmatrix} 1 \\ -1 \end{bmatrix}[x_1]; 1, 0\right)$ and the same partitioning. We get the INN $g(x_1) = \begin{bmatrix} [2,2] \\ [0,2] \\ [0,2] \end{bmatrix} \text{thresh}\left([[-1,1]][x_1]; 1, 0\right)$. We have $f(1) = \begin{bmatrix} 1 & 1 & 0 \end{bmatrix}^T$, however, in $g(1)$, no matter how we instantiate the $[-1, 1]$ weight, the output

of the thresh unit will either be 0 or 1. But then the output of the first output component must be either 0 or 2, neither of which is 1, and so g does *not* over-approximate f.

Necessary Condition: WIVP. We can generalize this example to the following necessary condition:

Theorem 5. *Suppose some $\sigma \in \Sigma$ is an activation function which does not satisfy the WIVP, and every $\mathcal{A} \in \mathfrak{A}$ is at least as precise as the interval hull abstraction. Then there exists a neural network N using only the identity and σ activation functions and partitioning $\mathbb{P}$ such that $T = \text{AbstractLayerWise}\langle\mathfrak{A}, \Sigma\rangle(N, \mathbb{P}, \mathcal{A})$ does not over-approximate N.*

While this is of some theoretical curiosity, in practice almost all commonly-used activation functions do satisfy the WIVP. Nevertheless, if one does wish to use such a function, one way to soundly over-approximate it with an ANN is to replace the *scalar* activation function with a *set-valued* one. The ANN semantics can be extended to allow picking any output value from the activation function in addition to any weight from the weight set.

For example, consider again the $\text{thresh}(x; 1, 0)$ activation function. It can be completed to a set-valued activation function which satisfies the WIVP such as $\text{thresh}'(x; 1, 0) \coloneqq \begin{cases} \{x\} & \text{if } x > 1 \\ \{a \mid a \in [0, 1]\} & \text{if } x = 1 \\ \{0\} & \text{otherwise} \end{cases}$. The idea is that we "fill the gap" in the graph. Whereas in the original threshold function we had an issue because there was no $x \in [0, 1]$ which satisfied $\text{thresh}(x; 1, 0) = \frac{f(0)+f(1)}{2} = \frac{1}{2}$, on the set-valued function we can take $x = 1 \in [0, 1]$ to find $\frac{1}{2} \in \text{thresh}'(1; 1, 0)$.

6.4 Powerset Abstraction, ReLU, and $\widehat{\alpha}_{bin}$

Recall that $\widehat{\alpha}$ (Algorithm 1) requires abstracting the, usually-infinite, set of *all* merged matrices $D^T W^{(i)} C$. However, in Sect. 5.1 we showed that for convex abstract domains it suffices to only consider the finitely-many *binary* mergings. The reader may wonder if there are abstract domains for which it is *not* sufficient to consider only the binary PCMs. This section presents such an example.

Suppose we use the same ReLU DNN f as in Sect. 6.1, for which we noted before the corresponding INN over-approximates it. However, suppose instead of intervals we used the *powerset* abstract domain, i.e., $\alpha(S) = S$ and $A \sqcup B = A \cup B$. If we (incorrectly) used $\widehat{\alpha}_{bin}$ instead of $\widehat{\alpha}$, we would get the powerset ANN $g(x_1) = \left\{ \begin{bmatrix} 2 \\ 2 \\ 0 \end{bmatrix}, \begin{bmatrix} 2 \\ 0 \\ 2 \end{bmatrix} \right\} \text{ReLU}\left(\{[1], [-1]\}\,[x_1]\right)$. Recall that $f(1) = \begin{bmatrix} 1 & 1 & 0 \end{bmatrix}^T$. However, with $g(1)$, the first output will always be either 0 or 2, so g does *not* over-approximate f. The basic issue is that to get the correct output, we need to instantiate the inner weight to 0.5, which is in the convex hull of the original weights, but is not either one of the original weights itself.

Note that, in this particular example, it is possible to find an ANN that over-approximates the DNN using only finite sets for the abstract weights. However,

this is only because ReLU is piecewise-linear, and the size of the sets needed will grow exponentially with the number of dimensions. For other activation functions, e.g., tanh infinite sets are required in general.

In general, non-convex abstract domains will need to use some other method of computing an over-approximation of $\widehat{\alpha}$. One general-purpose option is to use techniques such as those developed for symbolic abstraction [36] to iteratively compute an over-approximation of the true $A^{(i)}$ and use that instead.

7 Proof of Sufficient Conditions

We now prove Theorem 2, which provides sufficient conditions on the activation functions for which Algorithm 3 produces an ANN that soundly over-approximates the given DNN.

The structure of the proof is illustrated in Fig. 3. To show that ANN T over-approximates DNN N, we must show that $N(\boldsymbol{v}) \in T(\boldsymbol{v})$ for every $\boldsymbol{v}$. This occurs, by definition, only if there exists some *instantiation* $T_{\boldsymbol{v}} \in \gamma(T)$ of T for which $N(\boldsymbol{v}) = T_{\boldsymbol{v}}(\boldsymbol{v})$. Recall that an instantiation of an ANN is a DNN formed by replacing each abstract weight $A^{(i)}$ with a concrete weight matrix $H^{(i)} \in \gamma(A^{(i)})$. In particular, our proof will proceed layer-by-layer. On an input $\boldsymbol{v} = \boldsymbol{v}^{(0)}$, the ith layer of DNN N maps $\boldsymbol{v}^{(i-1)}$ to $\boldsymbol{v}^{(i)}$ until the output $\boldsymbol{v}^{(n)}$ is computed. We will prove that, for each abstract layer $(A^{(i)}, \sigma^{(i)}, \mathcal{A}^{(i)})$, there is a matrix $H^{(i)} = \overline{\gamma}(A^{(i)}, \boldsymbol{v}^{(i-1)}) \in \gamma(A^{(i)})$ for which the instantiated layer $(H^{(i)}, \sigma^{(i)})$, roughly speaking, also maps $\boldsymbol{v}^{(i-1)}$ to $\boldsymbol{v}^{(i)}$. However, by design the abstract layer will have fewer dimensions, hence the higher-dimensional $\boldsymbol{v}^{(i-1)}$ and $\boldsymbol{v}^{(i)}$ may not belong to its domain and range (respectively). We resolve this by associating with each vector $\boldsymbol{v}^{(i)}$ in the intermediate spaces of N a *mean representative* vector ${\boldsymbol{v}^{(i)}}_{/\mathbb{P}^{(i)}}$ in the intermediate spaces of $T_{\boldsymbol{v}}$. Then we can rigorously prove that the instantiated layer $(H^{(i)}, \sigma^{(i)})$ maps ${\boldsymbol{v}^{(i-1)}}_{/\mathbb{P}^{(i-1)}}$ to ${\boldsymbol{v}^{(i)}}_{/\mathbb{P}^{(i)}}$. Applying this fact inductively gives us $T_{\boldsymbol{v}}({\boldsymbol{v}}_{/\mathbb{P}^{(0)}}) = {(N(\boldsymbol{v}))}_{/\mathbb{P}^{(n)}}$. Because $\mathbb{P}^{(0)}$ and $\mathbb{P}^{(n)}$ are the singleton partitionings, this gives us exactly the desired relationship $T_{\boldsymbol{v}}(\boldsymbol{v}) = N(\boldsymbol{v})$.

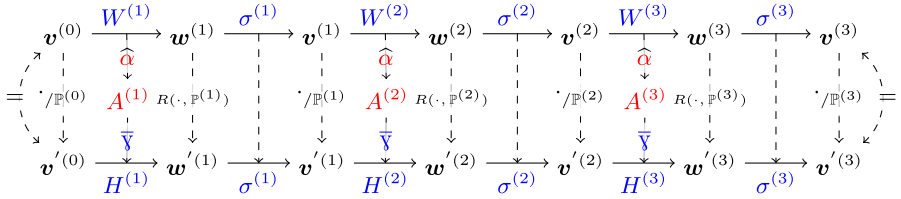


Fig. 3. Visualization of the relationships between concrete, abstract, and instantiated elements in the soundness proof. The original DNN's action on an input vector $\boldsymbol{v}^{(0)}$ is shown on the top row. This DNN is abstracted to an ANN, represented by the $A^{(i)}$s on the middle row. We will show that we can instantiate the ANN such that the instantiation has the same output as the original DNN on $\boldsymbol{v}^{(0)}$.

Algorithm 4: $\overline{\gamma}(M, \mathcal{P}^{in}, \mathcal{P}^{out}, \boldsymbol{v}, \boldsymbol{w}')$

Input: An $n \times m$ matrix M. Partitionings $\mathcal{P}^{in}$, $\mathcal{P}^{out}$. A vector $\boldsymbol{v}$ with non-negative entries. A vector $\boldsymbol{w}' \in R(M\boldsymbol{v}, \mathcal{P}^{out})$.

Output: A matrix $H \in \gamma(\widehat{\alpha}(M, \mathcal{P}^{in}, \mathcal{P}^{out}))$ such that $H(\boldsymbol{v}_{/\mathcal{P}^{in}}) = \boldsymbol{w}'$.

$C, D \leftarrow 0_{|\mathcal{P}^{in}| \times n}, 0_{|\mathcal{P}^{out}| \times m}$
for $i = 1, 2, \ldots, |\mathcal{P}^{in}|$ **do**
  for $j \in \mathcal{P}_i^{in}$ **do**
    $C_{j,i} \leftarrow v_j / (\sum_{k \in \mathcal{P}_i^{in}} v_k)$
$\boldsymbol{w} \leftarrow M\boldsymbol{v}$
for $i = 1, 2, \ldots, |\mathcal{P}^{out}|$ **do**
  $a, b \leftarrow \text{argmax}_{p \in \mathcal{P}_i^{out}} w_p, \text{argmin}_{p \in \mathcal{P}_i^{out}} w_p$
  $D_{a,i} \leftarrow (w'_i - w_b)/(w_a - w_b)$
  $D_{b,i} \leftarrow 1 - D_{a,i}$
$\boldsymbol{s} \leftarrow (|\mathcal{P}_1^{in}|, \ldots, |\mathcal{P}_{|\mathcal{P}^{in}|}^{in}|)$
return ScaleCols $\left(D^T M C, \boldsymbol{s}\right)$

7.1 Vector Representatives

Our proof relies heavily on the concept of representatives.

Definition 21. *Given a vector* $\boldsymbol{v} = (v_1, v_2, \ldots, v_n)$ *and a partitioning* $\mathcal{P}$ *of* $\{1, 2, \ldots, n\}$ *with* $|\mathcal{P}| = k$*, we define the* convex representative set *of* $\boldsymbol{v}$ *under* $\mathcal{P}$ *to be* $R(\boldsymbol{v}, \mathcal{P}) = \left\{(z_1, z_2, \ldots, z_k) \mid \forall j. \min_{h \in \mathcal{P}_j} v_h \leq z_j \leq \max_{h \in \mathcal{P}_j} v_h\right\}$.

$R(\boldsymbol{v}, \mathcal{P})$ is referred to as $AV(\boldsymbol{v})$ in Prabhakar et al. [29], and is always a box in $\mathbb{R}^k$.

One representative will be particularly useful, so we give it a specific notation:

Definition 22. *Given a vector* $(v_1, v_2, \ldots, v_n)$ *and a partitioning* $\mathcal{P}$ *of* $\{1, 2, \ldots, n\}$ *with* $|\mathcal{P}| = k$*, we define the* mean representative *of* v *under* $\mathcal{P}$ *to be* $v_{/\mathcal{P}} = \left(\frac{\sum_{j \in \mathcal{P}_1} v_j}{|\mathcal{P}_1|}, \ldots, \frac{\sum_{j \in \mathcal{P}_k} v_j}{|\mathcal{P}_k|}\right)$

Example 8. Consider the vector $\boldsymbol{v} := (5, 6, 11, 2, 1)$ and the partitioning $\mathcal{P} = \{\{1, 3\}, \{2, 4, 5\}\}$. Then we have $\boldsymbol{v}_{/\mathcal{P}} = ((5 + 11)/2, (6 + 2 + 1)/3) = (8, 3)$ and $R(\boldsymbol{v}, \mathcal{P}) = \{(z_1, z_2) \mid z_1 \in [5, 11], z_2 \in [1, 6]\}$.

7.2 Proof of Soundness Theorem

The operation $\overline{\gamma}$ presented in Algorithm 4 shows how to instantiate an abstract weight matrix such that it has input/output behavior corresponding to that of the original DNN layer. We now prove the correctness of Algorithm 4.

Lemma 1. *Given any $\boldsymbol{w}' \in R(M\boldsymbol{v}, \mathcal{P}^{in})$, a vector $\boldsymbol{v}$ with non-negative entries, and $H = \overline{\gamma}(M, \mathcal{P}^{in}, \mathcal{P}^{out}, \boldsymbol{v}, \boldsymbol{w}')$, then $H \in \gamma(\widehat{\alpha}(M, \mathcal{P}^{in}, \mathcal{P}^{out}))$ and $H(\boldsymbol{v}_{/\mathcal{P}^{in}}) = \boldsymbol{w}'$.*

Proof. To prove correctness of Algorithm 4, it suffices to show that (i) C and D are PCMs, and (ii) the returned matrix H satisfies the equality $H(\boldsymbol{v}_{/\mathcal{P}^{in}}) = \boldsymbol{w}'$.

C is a PCM by construction: The ith column only has non-zero entries for rows that are in the ith partition. The sum of all entries in a column is $\sum_{j \in \mathcal{P}_i^{in}} v_j / (\sum_{k \in \mathcal{P}_i^{in}} v_k) = 1$. All entries are non-negative by assumption on $\boldsymbol{v}$.

D is also a PCM: The ith column only has two entries. It suffices to show that $D_{a,i}$ is in $[0, 1]$, which follows because $\boldsymbol{w}' \in R(M\boldsymbol{v}, \mathcal{P}^{out})$ implies w'_i is in between the minimum b and maximum a.

By associativity, line 11 is equivalent to returning $H = D^T M E$ where $E = \text{ScaleCols}(C, \boldsymbol{s})$. Thus, to show that $H(\boldsymbol{v}_{/\mathcal{P}^{in}}) = \boldsymbol{w}'$, it suffices to show (i) that $E(\boldsymbol{v}_{/\mathcal{P}^{in}}) = \boldsymbol{v}$, and (ii) that $D^T M \boldsymbol{v} = \boldsymbol{w}'$.

Note that here $E_{j,i} = C_{j,i} |\mathcal{P}_i^{in}|$. Then to show (i), consider any index $j \in \mathcal{P}_i^{in}$. Then we find that the jth output component of $E(\boldsymbol{v}_{/\mathcal{P}^{in}})$ is $(v_j / (\sum_{k \in \mathcal{P}_i^{in} v_k})) |\mathcal{P}_i^{in}| ((\sum_{k \in \mathcal{P}_i^{in} v_k}) / |\mathcal{P}_i^{in}|) = v_j$. Hence, the entire output vector is $\boldsymbol{v}$.

To show (ii), note that each column of D is exactly the convex combination that produces the output w'_i from the maximum/minimum indices of $M\boldsymbol{v}$.

In total then, the returned matrix is in $\gamma(\widehat{\alpha}(M, \mathcal{P}^{in}, \mathcal{P}^{out}))$ and satisfies $H(\boldsymbol{v}_{/\mathcal{P}^{in}}) = \boldsymbol{w}'$. □

The next lemma implies that we can always find such a $\boldsymbol{w}' \in R(M\boldsymbol{v}, \mathcal{P}^{in})$ satisfying the relations in Fig. 3.

Lemma 2. *Let σ be an activation function satisfying the WIVP, $\boldsymbol{w}$ any vector, and $\mathcal{P}$ a partitioning the dimensions of $\boldsymbol{w}$. Then there exists a vector $\boldsymbol{w}' \in R(\boldsymbol{w}, \mathcal{P})$ such that $\sigma(\boldsymbol{w}') = (\sigma(\boldsymbol{w}))_{/\mathcal{P}}$.*

Proof. Because $\sigma^{(i)}$ is defined to be a component-wise activation function, we can assume WLOG that $\mathbb{P}^{(i)}$ has only a single partition, i.e., $\mathbb{P}^{(i)} = \{\{1, 2, \ldots, s^{(i)}\}\}$.

In that case, label the components of $\boldsymbol{w}^{(i)}$ such that $w_1^{(i)} \leq w_2^{(i)} \leq \ldots \leq w_n^{(i)}$. Then the statement of the lemma is equivalent to the assertion that there exists some $b \in [w_1^{(i)}, w_n^{(i)}]$ such that $\sigma^{(i)}(b) = (\sum_j w_j^{(i)})/n$. But this is exactly the definition of the WIVP. Hence, by assumption that $\sigma^{(i)}$ satisfies the WIVP, we complete the proof. □

We are finally prepared to prove the soundness theorem. It is restated here for clarity.

Theorem 2. *Let $\mathfrak{A}$ be a set of weight set abstract domains and Σ a set of activation functions. Suppose (i) each $\sigma \in \Sigma$ has entirely non-negative outputs, and (ii) each $\sigma \in \Sigma$ satisfies the Weakened Intermediate Value Property (Definition 20). Then $T = \text{AbstractLayerWise}\langle \mathfrak{A}, \Sigma \rangle(N, \mathbb{P}, \mathcal{A})$ (Algorithm 3) soundly over-approximates the DNN N.*

Proof. A diagram of the proof is provided in Fig. 3.

Consider the ith layer. By Lemma 2, there exists some vector $\boldsymbol{w}^{'(i)} \in R(\boldsymbol{w}^{(i)}, \mathbb{P}^{(i)})$ such that $\sigma^{(i)}(\boldsymbol{w}^{'(i)}) = \boldsymbol{v}_{/\mathbb{P}^{(i)}}$. Furthermore, by Lemma 1 there exists some $H^{(i)} \in \gamma(A^{(i)})$ such that $H^{(i)}(\boldsymbol{v}^{(i-1)}{}_{/\mathbb{P}^{(i-1)}}) = \boldsymbol{w}^{'(i)}$. Therefore, in total we can instantiate the ith abstract layer to $(H^{(i)}, \sigma^{(i)})$, which maps $\boldsymbol{v}^{(i-1)}{}_{/\mathbb{P}^{(i-1)}}$ to $\boldsymbol{v}^{(i)}{}_{/\mathbb{P}^{(i)}}$.

By applying this construction to each layer, we find an instantiation of the ANN that maps $\boldsymbol{v}^{(0)}{}_{/\mathbb{P}^{(0)}}$ to $\boldsymbol{v}^{(n)}{}_{/\mathbb{P}^{(n)}}$. Assuming $\mathbb{P}^{(0)}$ and $\mathbb{P}^{(n)}$ are the singleton partitionings, then, we have that the instantiation maps $\boldsymbol{v}^{(0)} = \boldsymbol{v}$ to $\boldsymbol{v}^{(n)} = N(\boldsymbol{v})$, as hoped for. Hence, $N(\boldsymbol{v}) \in T(\boldsymbol{v})$ for any such vector $\boldsymbol{v}$, and so the ANN overapproximates the original DNN. □

8 Related Work

The recent results by Prabhakar et al. [29] are the closest to this paper. Prabhakar et al. introduce the notion of Interval Neural Networks and a sound quotienting (abstraction) procedure when the ReLU activation function is used. Prabhakar et al. also proposed a technique for verification of DNNs using ReLU activation functions by analyzing the corresponding INN using a MILP encoding. Prabhakar et al. leaves open the question of determining the appropriate partitioning of the nodes, and their results assume the use of the ReLU activation function and interval domain. We have generalized their results to address the subtleties of other abstract domains and activation functions as highlighted in Sect. 6.

There exists prior work [2,8,27] on models using interval-weighted neural networks. The goal of such approaches is generally to represent uncertainty, instead of improve analysis time of a corresponding DNN. Furthermore, their semantics are defined using interval arithmetic instead of the more-precise semantics we give in Sect. 4. Nevertheless, we believe that future work may consider applications of our more general ANN formulation and novel abstraction algorithm to the problem of representing uncertainty.

There have been many recent approaches exploring formal verification of DNNs using abstractions. ReluVal [38] computes interval bounds on the outputs of a DNN for a given input range. Neurify [37] extends ReluVal by using symbolic interval analysis. Approaches such as DeepPoly [33] and AI^2 [9] perform abstract interpretation of DNNs using more expressive numerical domains such as polyhedra and zonotopes. In contrast, Abstract Neural Networks introduced in this paper use abstract values to represent the weight matrices of a DNN, and are a different way of applying abstraction to DNN analysis.

This paper builds upon extensive literature on numerical abstract domains [5,6,23,24], including libraries such as APRON [16] and PPL [1]. Of particular relevance are techniques for verification of floating-point computation [4,28,28].

Techniques for compression of DNNs reduce their size using heuristics [7,14,15]. They can degrade accuracy of the network, and do not provide theoretical

guarantees. Gokulanathan et al. [12] use the Marabou Verification Engine [19] to simplify neural networks so that the simplified network is equivalent to the given network. Shriver et al. [32] refactor the given DNN to aid verification, though the refactored DNN is not guaranteed to be an overapproximation.

9 Conclusion and Future Directions

We introduced the notion of an *Abstract Neural Network (ANN)*. The weight matrices in an ANN are represented using numerical abstract domains, such as intervals, octagons, and polyhedra. We presented a framework, parameterized by abstract domain and DNN activation function, that performs layer-wise abstraction to compute an ANN given a DNN. We identified necessary and sufficient conditions on the abstract domain and the activation function that ensure that the computed ANN is a sound over-approximation of the given DNN. Furthermore, we showed how the input DNN can be modified in order to soundly abstract DNNs using rare activation functions that do not satisfy the sufficiency conditions are used. Our framework is applicable to DNNs that use activation functions such as ReLU, Leaky ReLU, and Hyperbolic Tangent. Our framework can use convex abstract domains such as intervals, octagons, and polyhedra. Code implementing our framework can be found at https://doi.org/10.5281/zenodo.4031610. Detailed proofs of all theorems are in the extended version of this paper [34].

The results in this paper provide a strong theoretical foundation for further research on abstraction of DNNs. One interesting direction worth exploring is the notion of completeness of abstract domains [11] in the context of Abstract Neural Networks. Our framework is restricted to convex abstract domains; the use of non-convex abstract domains, such as modulo intervals [25] or donut domains [10], would require a different abstraction algorithm. Algorithms for computing symbolic abstraction might show promise [21,30,31,35,36].

This paper focused on feed-forward neural networks. Because convolutional neural networks (CNNs) are special cases of feed-forward neural networks, future work can directly extend the theory in this paper to CNN models as well. Such future work would need to consider problems posed by non-componentwise activation functions such as MaxPool, which do not fit nicely into the framework presented here. Furthermore, extensions for recursive neural networks (RNNs) and other more general neural-network architectures seems feasible.

On the practical side of things, it would be worth investigating the impact of abstracting DNNs on the verification times. Prabhakar et al. [29] demonstrated that their abstraction technique improved verification of DNNs. The results in this paper are a significant generalization of the results of Prabhakar et al., which were restricted to interval abstractions and ReLU activation functions. We believe that our approach would similarly help scale up verification of DNNs.

Acknowledgments. We thank the anonymous reviewers and Cindy Rubio González for their feedback on this work.

References

1. Bagnara, R., Hill, P.M., Zaffanella, E.: The parma polyhedra library: toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Sci. Comput. Program. **72**(1–2), 3–21 (2008). https://doi.org/10.1016/j.scico.2007.08.001
2. Beheshti, M., Berrached, A., de Korvin, A., Hu, C., Sirisaengtaksin, O.: On interval weighted three-layer neural networks. In: Proceedings 31st Annual Simulation Symposium (SS 1998), 5–9 April 1998, Boston, MA, USA. pp. 188–194. IEEE Computer Society (1998). https://doi.org/10.1109/SIMSYM.1998.668487
3. Brown, T.B., et al.: Language models are few-shot learners. CoRR abs/2005.14165 (2020). https://arxiv.org/abs/2005.14165
4. Chen, L., Miné, A., Cousot, P.: A sound floating-point polyhedra abstract domain. In: Ramalingam, G. (ed.) APLAS 2008. LNCS, vol. 5356, pp. 3–18. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89330-1_2
5. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Graham, R.M., Harrison, M.A., Sethi, R. (eds.) Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977, pp. 238–252. ACM (1977). https://doi.org/10.1145/512950.512973
6. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Aho, A.V., Zilles, S.N., Szymanski, T.G. (eds.) Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978, pp. 84–96. ACM Press (1978). https://doi.org/10.1145/512760.512770
7. Deng, L., Li, G., Han, S., Shi, L., Xie, Y.: Model compression and hardware acceleration for neural networks: a comprehensive survey. Proc. IEEE **108**(4), 485–532 (2020). https://doi.org/10.1109/JPROC.2020.2976475
8. Garczarczyk, Z.A.: Interval neural networks. In: IEEE International Symposium on Circuits and Systems, ISCAS 2000, Emerging Technologies for the 21st Century, Geneva, Switzerland, 28–31 May 2000, Proceedings. pp. 567–570. IEEE (2000). https://doi.org/10.1109/ISCAS.2000.856123
9. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.T.: AI2: safety and robustness certification of neural networks with abstract interpretation. In: 2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21–23 May 2018, San Francisco, California, USA, pp. 3–18. IEEE Computer Society (2018). https://doi.org/10.1109/SP.2018.00058
10. Ghorbal, K., Ivančić, F., Balakrishnan, G., Maeda, N., Gupta, A.: Donut domains: efficient non-convex domains for abstract interpretation. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 235–250. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27940-9_16
11. Giacobazzi, R., Ranzato, F., Scozzari, F.: Making abstract interpretations complete. J. ACM **47**(2), 361–416 (2000). https://doi.org/10.1145/333979.333989
12. Gokulanathan, S., Feldsher, A., Malca, A., Barrett, C.W., Katz, G.: Simplifying neural networks with the marabou verification engine. CoRR abs/1910.12396 (2019). http://arxiv.org/abs/1910.12396
13. Goodfellow, I.J., Bengio, Y., Courville, A.C.: Deep Learning. Adaptive Computation and Machine Learning. MIT Press (2016). http://www.deeplearningbook.org/

14. Han, S., Mao, H., Dally, W.J.: Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In: Bengio, Y., LeCun, Y. (eds.) 4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2–4, 2016, Conference Track Proceedings (2016). http://arxiv.org/abs/1510.00149
15. Iandola, F.N., Moskewicz, M.W., Ashraf, K., Han, S., Dally, W.J., Keutzer, K.: Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. CoRR abs/1602.07360 (2016). http://arxiv.org/abs/1602.07360
16. Jeannet, B., Miné, A.: Apron: a library of numerical abstract domains for static analysis. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 661–667. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_52
17. Julian, K.D., Kochenderfer, M.J., Owen, M.P.: Deep neural network compression for aircraft collision avoidance systems. CoRR abs/1810.04240 (2018). http://arxiv.org/abs/1810.04240
18. Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: an efficient SMT solver for verifying deep neural networks. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 97–117. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_5
19. Katz, G., et al.: The marabou framework for verification and analysis of deep neural networks. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 443–452. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_26
20. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Bartlett, P.L., Pereira, F.C.N., Burges, C.J.C., Bottou, L., Weinberger, K.Q. (eds.) Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3–6, 2012, Lake Tahoe, Nevada, United States, pp. 1106–1114 (2012). http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks
21. Li, Y., Albarghouthi, A., Kincaid, Z., Gurfinkel, A., Chechik, M.: Symbolic optimization with SMT solvers. In: Jagannathan, S., Sewell, P. (eds.) The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, San Diego, CA, USA, January 20–21, 2014. pp. 607–618. ACM (2014). https://doi.org/10.1145/2535838.2535857
22. Maas, A., Hannun, A., Ng, A.: Rectifier nonlinearities improve neural network acoustic models. In: Proceedings of the International Conference on Machine Learning (2013)
23. Miné, A.: The octagon abstract domain. High. Order Symb. Comput. **19**(1), 31–100 (2006). https://doi.org/10.1007/s10990-006-8609-1
24. Miné, A.: Tutorial on static inference of numeric invariants by abstract interpretation. Found. Trends Program. Lang. **4**(3–4), 120–372 (2017). https://doi.org/10.1561/2500000034
25. Nakanishi, T., Joe, K., Polychronopoulos, C.D., Fukuda, A.: The modulo interval: a simple and practical representation for program analysis. In: Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques, Newport Beach, California, USA, October 12–16, 1999, pp. 91–96. IEEE Computer Society (1999). https://doi.org/10.1109/PACT.1999.807422

26. Paszke, A., et al.: Pytorch: An imperative style, high-performance deep learning library. In: Wallach, H.M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E.B., Garnett, R. (eds.) Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8–14 December 2019, Vancouver, BC, Canada, pp. 8024–8035 (2019). http://papers.nips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library
27. Patiño-Escarcina, R.E., Callejas Bedregal, B.R., Lyra, A.: Interval computing in neural networks: one layer interval neural networks. In: Das, G., Gulati, V.P. (eds.) CIT 2004. LNCS, vol. 3356, pp. 68–75. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30561-3_8
28. Ponsini, O., Michel, C., Rueher, M.: Verifying floating-point programs with constraint programming and abstract interpretation techniques. Autom. Softw. Eng. **23**(2), 191–217 (2016). https://doi.org/10.1007/s10515-014-0154-2
29. Prabhakar, P., Afzal, Z.R.: Abstraction based output range analysis for neural networks. In: Wallach, H.M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E.B., Garnett, R. (eds.) Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8–14 December 2019, Vancouver, BC, Canada, pp. 15762–15772 (2019). http://papers.nips.cc/paper/9708-abstraction-based-output-range-analysis-for-neural-networks
30. Reps, T., Sagiv, M., Yorsh, G.: Symbolic implementation of the best transformer. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 252–266. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24622-0_21
31. Reps, T., Thakur, A.: Automating abstract interpretation. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 3–40. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_1
32. Shriver, D., Xu, D., Elbaum, S.G., Dwyer, M.B.: Refactoring neural networks for verification. CoRR abs/1908.08026 (2019). http://arxiv.org/abs/1908.08026
33. Singh, G., Gehr, T., Püschel, M., Vechev, M.T.: An abstract domain for certifying neural networks. Proc. ACM Program. Lang. **3**(POPL), 411–4130 (2019). https://doi.org/10.1145/3290354
34. Sotoudeh, M., Thakur, A.V.: Abstract neural networks. CoRR abs/2009.05660 (2020). http://arxiv.org/abs/2009.05660
35. Thakur, A., Elder, M., Reps, T.: Bilateral algorithms for symbolic abstraction. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 111–128. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33125-1_10
36. Thakur, A., Reps, T.: A method for symbolic computation of abstract operations. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 174–192. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_17

37. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Efficient formal safety analysis of neural networks. In: Bengio, S., Wallach, H.M., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (eds.) Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3–8 December 2018, Montréal, Canada, pp. 6369–6379 (2018). http://papers.nips.cc/paper/7873-efficient-formal-safety-analysis-of-neural-networks
38. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Formal security analysis of neural networks using symbolic intervals. In: Enck, W., Felt, A.P. (eds.) 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15–17, 2018, pp. 1599–1614. USENIX Association (2018). https://www.usenix.org/conference/usenixsecurity18/presentation/wang-shiqi