Permutation Invariance of Deep Neural Networks with ReLUs

Diganta Mukhopadhyay[1(✉)], Kumar Madhukar[2], and Mandayam Srivas[1]

[1] Chennai Mathematical Institute, Chennai, India
digantam@cmi.ac.in
[2] Indian Institute of Technology Delhi, Delhi, India
madhukar@cse.iitd.ac.in

Abstract. We look at the problem of verifying *permutation invariance* in Deep Neural Networks (DNNs) – if certain permutations are applied on the inputs, its effect on the outputs will also be a permutation (possibly identity). These properties surface in many interesting practical applications of DNNs, e.g. consider the aircraft collision avoidance system that guides an aircraft to turn right if the sensory inputs suggest an intruder aircraft coming from the left, and *vice-versa*. The naive way of verifying such properties – using two copies of the network and a standard DNN verification technique, e.g. Reluplex – is impracticable as the complexity of this task is exponential in the network size. This paper proposes a sound, abstraction-based technique to establish permutation invariance in DNNs with ReLU as the activation function. The technique computes an over-approximation of the reachable states, and an under-approximation of the safe states, and propagates this information across the layers, both forward and backward. The novelty of our approach lies in a useful *tie-class* analysis, that we introduce for forward propagation, and a scalable 2-polytope under-approximation method that escapes the exponential blow-up in the number of regions during backward propagation. Experiments demonstrate that our method compares favorably with the existing state-of-the-art in DNN verification.

1 Introduction

Artificial neural networks are now ubiquitous. They are increasingly being allowed and used to handle increasingly more complex tasks, that used to be unimaginable for a machine to perform. This includes driving cars, playing games, maneuvering air traffic, recognizing speech, interpreting images and videos, creating art, and numerous other things. While this is exciting, it is crucial to understand that neural networks are responsible for a lot of decision making, some of which can have disastrous consequences if gone wrong. Consider a DNN that is being used to suggest the direction in which an aircraft

The authors are thankful to TCS Research, Pune, India. A substantial part of this work was done when the first two authors were associated with TCS Research, as an intern and as an employee, respectively.

J. V. Deshmukh et al. (Eds.): NFM 2022, LNCS 13260, pp. 318–337, 2022.
https://doi.org/10.1007/978-3-031-06773-0_17

must turn to avoid a possible collision with an intruder aircraft. Informally, such a network is well-behaved if it asks the own ship to turn right (left) when an intruder approaches from the left (right). Consider another network that takes four inputs – the cards dealt to the players in a game of contract bridge – and decides which team can bid *game*. Loosely speaking, if you exchange the hands of partners (*north* and *south*, or *east* and *west*), the decision would not change. However, it will change if, say, you exchange north's hand with east. Such *permutation invariance* properties, for certain permutations at input and output layers, are important to the correctness and robustness of these networks.

Formally, given a DNN $\mathcal{N}$, permutations σ_{in} and σ_{out}, two vectors B_1 and B_2 of dimension as large as the input size of the neural network and a positive real M, the permutation invariance is defined as: *if the inputs of the network lie between B_1 and B_2 component-wise, then permuting the input of the network by σ_{in} leads to the output being permuted by σ_{out} up to a tolerance of M. That is,*

$$B_1 \leq x \leq B_2 \Rightarrow |\sigma_{out}(\mathcal{N}(x)) - \mathcal{N}(\sigma_{in}(x))| \leq M$$

Permutation invariance of DNNs is really a two-safety property, i.e. it can be verified using existing techniques for safety verification of feed-forward neural networks (FFNNs), by composing two copies of the network. A straightforward way to do this would be to encode the network and the property as SMT constraints, and solve it using Z3 [4]. It is invariably more efficient, however, to use specially designed solvers and frameworks such as Reluplex [12] and Marabou [13,14]. Still, these methods do not scale well, and are particularly inapplicable in this case (which requires doubling the network size), as the worst-case complexity of FFNN verification is exponential in the size of the input network.

This paper proposes a technique to verify permutation invariance properties in DNNs with ReLU (Rectified Linear Unit) activation function. Our technique computes, at each layer, an over-approximation of the reachable states (moving forward from the input layer), and also an under-approximation of the safe states (moving backward from the final layer at which the property is specified). If the reachable states fall entirely within the safe region in any of the layers, the property is established. Otherwise, we obtain a witness to exclusion at each layer and do a spuriousness check to see if there is an actual counterexample.

The novelty of our approach lies in the way we propagate information across layers. For the forward propagation of reachable states, as affine regions, we have introduced the notion of *tie classes*. The purpose of tie classes is to group together the *Relu* nodes that will always get inputs of the same sign. This grouping cuts down on the branching required to account for active and inactive states of all the *Relu* nodes during forward propagation. Intuitively, tie classes let us exploit the behavioral symmetry of the network, with respect to the inputs and the permutation. The backward propagation relies on convex polytope propagation. During the propagation one may have to account for multiple cases, based on the possible signs of the inputs to the *Relu* nodes (corresponding to each quadrant of the space in which the polytope resides), leading to an exponential blow-up in the worst case. We address this by proposing a 2-polytope under-approximation

method that is efficient (does not depend on LP/SMT solving), scalable, as well as effective. Note that the forward propagation may also be done using convex polytope propagation (which is how it is usually done, e.g. [19]), but it requires computing the convex hull each time, which is an expensive operation. In contrast, tie-class analysis helps us propagate the affine regions efficiently.

The core contributions of this paper are: *i*) an approach for verifying permutation invariance, based on novel forward- and backward-propagation techniques (Sect. 4), *ii*) a proof of soundness of the proposed approach (in the Appendix), and *iii*) a tool and an experimental evaluation of our approach (Sect. 5).

2 Preliminaries

We represent vectors in n-dimensional space as row matrices, i.e., with one row and n columns. A linear transform T from and n dimensional space to an m dimensional space can then be represented by a matrix M with n rows and m columns, and we have: $T(\boldsymbol{x}) = \boldsymbol{x}M$.

Convex Polytopes. A convex polytope is defined as a conjunction of a set of linear constraints indexed by i of the form $\boldsymbol{x}.\boldsymbol{v_i} \leq c_i$, for fixed (column) vectors $\boldsymbol{v_i}$ and constants c_i. Geometrically, it is a convex region in space enclosed within a set of planar boundaries. Symbolically, we can represent a convex polytope by arranging all the $\boldsymbol{v_i}$s into the columns of a matrix M, and letting the components of a row vector $\boldsymbol{b}$ to be constants b_i: $\boldsymbol{x}M \leq \boldsymbol{b}$.

Pullback. The *pullback* of a convex polytope P (given by $\boldsymbol{x}M_P^{n\times k} \leq \boldsymbol{b_p}$), over an affine transform T (given by $\boldsymbol{x} \rightarrow \boldsymbol{x}M_T^{m\times n} + \boldsymbol{t_T}$), is defined as the set of all points $\boldsymbol{x}$ such that $T(\boldsymbol{x})$ lies inside P, i.e., $T(\boldsymbol{x}) \in P \Leftrightarrow \boldsymbol{x}M_TM_P \leq \boldsymbol{b_P} - \boldsymbol{t_T}M_P$.[1]

Affine Region. An n-dimensional affine subspace is the set of all points generated by linear combinations of a set of *basis* vectors $\boldsymbol{v_i}, 0 \leq i < k$, added to a *center* $\boldsymbol{c}$: $\{\boldsymbol{x} \mid \boldsymbol{x} = (\Sigma_{i=0}^{k-1}\alpha_i\boldsymbol{v_i}) + \boldsymbol{c}$, for some real $\alpha_i\}$.
We define an *affine region* as a constrained affine subspace by bounding the values of α to be between -1 and 1. Formally, an affine region $A[B_A, \boldsymbol{c}]$ generated by a set of basis vectors $\boldsymbol{v_i}, 0 \leq i < k$, represented by a matrix $B_A^{k\times n}$, is defined as the following set of points: $\boldsymbol{x} \in A \Leftrightarrow (\exists \boldsymbol{\alpha}.\ \boldsymbol{x} = \boldsymbol{\alpha}B_A + \boldsymbol{c} \wedge |\boldsymbol{\alpha}| \leq 1)$.

Pushforward. The *pushforward* (A_T) of an affine region A (defined by B_A and $\boldsymbol{c_A}$), across an affine transform T, (given by $\boldsymbol{x} \rightarrow \boldsymbol{x}M_T + \boldsymbol{t_T}$), is the set of points: $\boldsymbol{x} \in A_T \Leftrightarrow (\exists \boldsymbol{\alpha}.\ \boldsymbol{x} = \boldsymbol{\alpha}B_AM_T + \boldsymbol{c_A}M_T + \boldsymbol{t_T}\ ,\ |\alpha| \leq 1)$. This is the image of A under T. (In a DNN context, a separate M_T and t_T is associated with each layer that is constructed from the weights and bias used at that layer.)

DNN Notation and Conventions. We number the layers of the neural network as 0, 1, 2, and so on, upto $n-1$. A layer is said to consist of an affine transform followed by a *Relu* layer. The affine transform of layer i is given by $\boldsymbol{x} \rightarrow \boldsymbol{x}W_i + \boldsymbol{b_i}$, where W_i are the weights and $\boldsymbol{b_i}$ are biases. We denote the input

[1] We use bold face to denote vectors, and $A^{p\times q}$ means P is a $p \times q$ matrix.

vectors by $\boldsymbol{x_0}$ feeding into the affine transform of layer 0, and in general for $i > 0$, the input of layer i's affine transform (the output of the $i-1^{th}$ layer's *Relu*) as $\boldsymbol{x_i}$. The output of layer i's affine transform (the input to layer i's *Relu*) is labeled as $\boldsymbol{y_i}$. Finally, the output is $\boldsymbol{x_n}$. Also, we maintain copies of each variable's original and permuted value (using a primed notation). So, we have:

$$\boldsymbol{x_0}, \boldsymbol{x'_0} \rightarrow \boldsymbol{x}W_0 + \boldsymbol{b_0} \rightarrow \boldsymbol{y_0}, \boldsymbol{y'_0} \rightarrow \mathit{Relu} \rightarrow \boldsymbol{x_1}, \boldsymbol{x'_1} \rightarrow \boldsymbol{x}W_1 + \boldsymbol{b_1} \rightarrow \boldsymbol{y_1}, \boldsymbol{y'_1} \rightarrow \mathit{Relu} \rightarrow \cdots \boldsymbol{y_{n-1}}, \boldsymbol{y'_{n-1}} \rightarrow \mathit{Relu} \rightarrow \boldsymbol{x_n}, \boldsymbol{x'_n}$$

Here, W_i and $\boldsymbol{b_i}$ represent the action of the layer on the joint space of $\boldsymbol{x_i}$ and $\boldsymbol{x'_i}$. Then, the invariance property we wish to verify has the following form:

$$B_1 \leq \boldsymbol{x_0}, \boldsymbol{x'_0} \leq B_2 \wedge \boldsymbol{x'_0} = \sigma_{in}(\boldsymbol{x_0}) \Rightarrow |\boldsymbol{x'_n} - \sigma_{out}(\boldsymbol{x_n})| \leq M$$

Note that the precondition here is an affine region and the postcondition is a conjunction of linear inequalities, involving permutations.

3 Informal Overview

Algorithm 1. Overview of our approach

```
1: inputs: 𝒩, n, pre, post
2: globals: reach[n], safe[n]

3: reach[0] ← initPre(pre, 𝒩)
4: safe[n − 1] ← initPost(post, 𝒩)
5: for i ∈ [1 . . . n) do
6:     reach[i] ← forwardPropagate(reach[i − 1], 𝒩)
7: for i ∈ [n − 2 . . . 0) do
8:     safe[i] ← backwardPropagate(safe[i + 1], 𝒩)
9: for i ∈ [1 . . . n) do
10:    if (reach[i] ∧ ¬safe[i]) is unsatisfiable then
11:        return property holds
12:    else      ▷ there must be a satisfying witness
13:        spuriousnessCheck(witness, i)
```

Algorithm 1 presents an overview of our approach. The input to it is the network $\mathcal{N}$ with n layers, and the invariance property given as a (*pre, post*) pair of formulas. The algorithm begins by converting the pre-condition to an affine region by calling *initPre* (line 3) and expressing the postcondition as a convex polytope by calling *initPost* (line 4), without any loss of precision (see Sect. 3.1). Then it propagates the affine region forward, to obtain an over-approximation of the set of reachable values as an affine region at each subsequent layers (line 6). Similarly, an under-approximation of the safe region – as a union of two convex polytopes – is calculated at each layer, propagating the information backward from the output layer (line 8). The property holds if the reachable region at any layer is contained within the safe region (lines 9–13).

If the inclusion check does not succeed, the algorithm attempts to construct an actual counterexample from the witness to the inclusion check failure (see Algorithm 2). In general, pulling back the witness to the first layer is as hard as pulling back the postcondition. So, we try to find several individual input points that lead to something close to the witness at the layer where the inclusion fails,

Algorithm 2. Spuriouness checking algorithm

```
1: procedure spuriousnessCheck (counterexample, layer)
2:     cexes ← [counterexample]                    ▷ list of potential counterexamples
3:     while cexes ≠ ∅ ∧ layer > 0 do
4:         prevCexes ← ∅          ▷ collect (approximate) pullbacks in the prev. reach
5:         for cex ∈ cexes do
6:             prevCexes << pullBackCex(cex, layer, 𝒩) ⋂ reach[layer − 1]
7:         cexes ← prevCexes; layer ← layer − 1
8:     if cexes = ∅ then
9:         return inconclusive          ▷ pullback failed, no potential counterexamples
10:    for cex ∈ cexes do
11:        for j ∈ [0 . . . n) do              ▷ forward simulation of the counterexample
12:            cex ← simulateLayer(cex, j, 𝒩)
13:            if cex ∈ safe[j] then      ▷ spurious c'example, move on to the next one
14:                break
15:        return (property failed, cex)                ▷ actual counterexample found
16:    return inconclusive                  ▷ all potential counterexamples are safe
```

allowing us to check a number of potential counterexamples. In lines 5–6 (Algorithm 2) we repeatedly apply *pullBackCex* and collect these approximate pull back points layer by layer backwards until the input layer. We now simulate these points forward to check if the output of the DNN lies within the safe region in lines 11–17. If for any point it does not, we have successfully constructed a counterexample. Otherwise, if we cannot find any potential counterexamples (line 10), or if all the potential counterexamples are safe (line 17), the witness represents a spurious counterexample and the algorithm returns *inconclusive*. Before getting into the details of *forwardPropagate*, *backwardPropagate*, and *pullBackCex*, we present an example and describe the pre-processing part of our algorithm.

3.1 Running Example

Consider the neural network shown in Fig. 1. Here, we have separated the result of computing the weighted sum from that of the application of the *Relu* into separate nodes, represented by dashed and solid circles respectively. Also, we show the weights as labels on the arrows coming into a combination point (dark circles), and biases as labels of arrows emerging from the point. The arrows for weights that are 0 have been omitted. The values at (output of) each node in the network for the input in the range [0.5 0] are shown in the diagram at that node. This network has the following symmetry property: $0 \leq x_{00}, x_{01}, x'_{00}, x'_{01} \leq 1 \wedge x_{00} = x'_{01} \wedge x_{01} = x'_{00} \Rightarrow |[x_{20}\ x_{21}] - [x'_{21}\ x'_{20}]| \leq 0.1$. This expresses the fact that flipping the inputs leads to the outputs being flipped, σ_{in} and σ_{out} both flip the components.

Preprocessing: The W_i and $\boldsymbol{b_i}$ are calculated as follows: If the weights and bias of layer i are W^i and $\boldsymbol{b^i}$, then $W_i = \left[\begin{smallmatrix} W^i & 0 \\ 0 & W^i \end{smallmatrix}\right]$ and $\boldsymbol{b_i} = [\boldsymbol{b^i}\ \boldsymbol{b^i}]$ as we need to track both the original and permuted values at each layer. For this example:

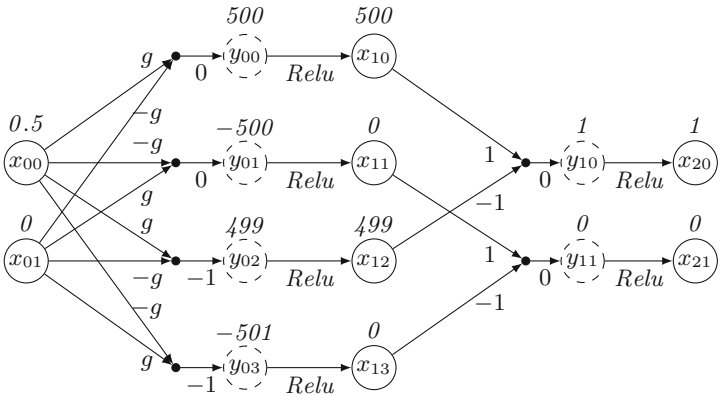

Fig. 1. $\sigma = (0{\to}1, 1{\to}0)$, $g = 1000$

$$W_0 = \begin{bmatrix} 1000 & -1000 & 1000 & -1000 & 0 & 0 & 0 & 0 \\ -1000 & 1000 & -1000 & 1000 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1000 & -1000 & 1000 & -1000 \\ 0 & 0 & 0 & 0 & -1000 & 1000 & -1000 & 1000 \end{bmatrix}$$

$$\boldsymbol{b_0} = [0\ 0\ -1\ -1\ 0\ 0\ -1\ -1]$$

$$W_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

$$\boldsymbol{b_1} = [0\ 0\ 0\ 0]$$

Action of *initPre* **and** *initPost***:** Now, *initPre* calculates *reach*[0] as the following affine region given by basis B_0 and center $\boldsymbol{c_0}$, and *initPost* expresses *safe*[2] as a convex polytope:

$$\begin{aligned} &\textbf{reach}[\mathbf{0}]: && \textbf{safe}[\mathbf{2}]: \\ &\exists \boldsymbol{\alpha}: [\boldsymbol{x_0}\ \boldsymbol{x_0'}] = \boldsymbol{\alpha} B_0 + \boldsymbol{c_0},\ |\boldsymbol{\alpha}| \le 1 && [\boldsymbol{x_2}\ \boldsymbol{x_2'}] \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & -1 & 0 & 1 \\ -1 & 0 & 1 & 0 \end{bmatrix} \le [0.1\ 0.1\ 0.1\ 0.1] \\ &\qquad B_0 = \begin{bmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0.5 & 0 \end{bmatrix} \\ &\qquad \boldsymbol{c_0} = [0.5\ 0.5\ 0.5\ 0.5] \end{aligned} \tag{1}$$

Forward Propagation: *ForwardPropagate* then propagates (1) across the layers to get affine regions that are over-approximations for the reachable region for that layer. While propagation across the linear layer can be done easily via matrix multiplication, propagating across the *Relu* layer is in general hard, since we need to take into account all possible branching behaviors. We do this via a tie class analysis (Sect. 4.1) that exploits the inherent symmetry of the network and precondition. For this network, propagating across the first linear layer gives us an affine region given by the basis and center:

$$B'_0 = \begin{bmatrix} 500 & -500 & 500 & -500 & -500 & 500 & -500 & 500 \\ -500 & 500 & -500 & 500 & 500 & -500 & 500 & -500 \end{bmatrix}$$
$$\boldsymbol{c'_0} = \begin{bmatrix} 0 & 0 & -1 & -1 & 0 & 0 & -1 & -1 \end{bmatrix}$$

Then, propagating across the *Relu* using the tie class analysis (Sect. 4.1) gives us the basis B_1 and center $\boldsymbol{c_1}$ for $reach[1]$. Similarly, the algorithm propagates across the second layer to get B'_1, $\boldsymbol{c'_1}$, B_2 and $\boldsymbol{c_2}$. In this case, the affine region before and after the *Relu* turn out to be the same, and there is no loss in precision going from B'_1 to B_2. The matrices are:

$$B_1 = \begin{bmatrix} 500 & 0 & 0 & 0 & 0 & 500 & 0 & 0 \\ -500 & 0 & 0 & 0 & 0 & -500 & 0 & 0 \\ 0 & -500 & 0 & 0 & -500 & 0 & 0 & 0 \\ 0 & 500 & 0 & 0 & 500 & 0 & 0 & 0 \\ 0 & 0 & 500 & 0 & 0 & 0 & 0 & 500 \\ 0 & 0 & -500 & 0 & 0 & 0 & 0 & -500 \\ 0 & 0 & 0 & -500 & 0 & 0 & -500 & 0 \\ 0 & 0 & 0 & 500 & 0 & 0 & 500 & 0 \end{bmatrix}$$
$$\boldsymbol{c_1} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$B'_1, B_2 = \begin{bmatrix} 500 & 0 & 0 & -500 \\ -500 & 0 & 0 & 500 \\ 0 & -500 & -500 & 0 \\ 0 & 500 & 500 & 0 \\ -500 & 0 & 0 & -500 \\ 500 & 0 & 0 & 500 \\ 0 & 500 & 500 & 0 \\ 0 & -500 & -500 & 0 \end{bmatrix}$$
$$\boldsymbol{c'_1}, \boldsymbol{c_2} = \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix} \quad (2)$$

Inclusion Check: Now, we see that if we substitute $\boldsymbol{x}$ with the form given in $reach[2]$ given by B_2 and $\boldsymbol{c_2}$ above into $safe[2]$ from Eq. 1, the right side of the inequality in $safe[2]$ is a matrix multiplication that evaluates to 0. So, $reach[2]$ is included in $safe[2]$. This is done by an algorithm (Sect. 4.3) that checks this using an LP solver, and since it succeeds in this case, it returns *property holds*.

Note that for this example, it was unnecessary to perform any back propagation of the $safe[i]$ to previous layers, as the inclusion check succeeded at the output layer. In general, back propagation (Sect. 4.2) would be performed to compute under-approximations. Spuriousness check (Sect. 4.3) will be needed if the inclusion check fails.

4 Forward and Backward Propagation

An input precondition of the form $\boldsymbol{B_1} \leq [\boldsymbol{x_0}, \boldsymbol{x'_0}] \leq \boldsymbol{B_2}$ with $\boldsymbol{x'_0} = \sigma(\boldsymbol{x_0})$ can always be converted into an equivalent affine region characterized by the formula $\exists \boldsymbol{\alpha} : [\boldsymbol{x_0}\ \boldsymbol{x'_0}] = \boldsymbol{\alpha} V + \boldsymbol{c},\ |\boldsymbol{\alpha}| \leq 1$ by making corresponding components of V the same according to σ, shifting the origin of V and scaling. This gives us $reach[0]$. Similarly, a postcondition of the form $|\boldsymbol{x'_n} - \sigma_{out}(\boldsymbol{x_n})| \leq M$ can be written as the convex polytope $|[\boldsymbol{x_n}\ \boldsymbol{x'_n}]L| \leq M$ where each column of L calculates one of the differences of corresponding components, giving us $safe_n$. Having $reach[0]$ and $safe_n$, we move on to forward and backward propagation.

4.1 Forward Propagation Using Tie Classes

Let $reach[j] = \{[\boldsymbol{x_j}\ \boldsymbol{x'_j}] \mid \exists \boldsymbol{\alpha} :\ [\boldsymbol{x_j}\ \boldsymbol{x'_j}] = \boldsymbol{\alpha} B_j + \boldsymbol{c_j},\ |\boldsymbol{\alpha}| \leq 1\}$, be the affine region representing an over-approximation of reachable points at the input to layer j; $forwardPropagate$ constructs $reach[j+1]$ as an affine region that is an over-approximation for the set of all points produced when $reach[j]$ is propagated to the input of layer $j+1$. $reach[j+1]$, is constructed by forward propagating $reach[j]$ first across the affine transform at j to produce an affine region A_j, which is then further forward propagated across the *Relu* layer.

Forward propagation across the linear transform given by $\boldsymbol{x} \rightarrow \boldsymbol{x} W_j + \boldsymbol{b_j}$ is straightforward and precise as it can be computed as a simple linear pushforward across W_j, i.e., $A_j([\boldsymbol{y_j}\ \boldsymbol{y'_j}]) \Leftrightarrow (\exists \boldsymbol{\alpha} :\ [\boldsymbol{y_j}\ \boldsymbol{y'_j}] = \boldsymbol{\alpha} B'_j + \boldsymbol{c'_j},\ |\boldsymbol{\alpha}| \leq 1)$, where $B'_j = B_j W_j$ is the new basis and $\boldsymbol{c'_j} = \boldsymbol{c_j} W_j + \boldsymbol{b_j}$ is the new center.

Propagating A_j across *Relu* is more complex and challenging as it requires, in general, a detailed case analysis of the polarity and strength of the components of the basis vectors and the scaling α; rather than performing it precisely, $reach[j+1]$ is constructed as an affine region that over-approximates the *Relu* image. Several methods can be used to construct an over-approximation that make different tradeoffs between precision and efficiency. One can construct the smallest affine region (or polytope) that includes all the reachable values possible across the *Relu* [19]. Computing the smallest region can be inefficient as it is an optimization problem requiring several expensive LP or convex-hull calls. Our method efficiently constructs an over-approximate affine region that, while sub-optimal, does not need any LP calls and is effective for checking permutation invariance properties.

Our method to construct the over-approximate affine region relies on looking for similarities in the polarity of the components of the vectors belonging to $reach[j]$ that are preserved when a *Relu* is applied to the region. For this, we introduce the notion of *tie classes* associated with an affine region.

Propagating over Relu with Tie Classes. Given an affine region A defined by a basis $\boldsymbol{v_i}$ and center $\boldsymbol{c}$ we define a binary relation, *tied*, over the set of indices[2] denoting the components of any vector $\boldsymbol{x}$ in A as follows.

Definition 1 (Tied). *Given an affine region A characterized by the condition $\exists \alpha_i :\ \boldsymbol{x} = \sum_i \alpha_i \boldsymbol{v_i} + \boldsymbol{c},\ |\alpha_i| \leq 1$, and two indices i_1 and i_2 in the index set, we say i_1 and i_2 are tied iff for every vector $\boldsymbol{x}$ in A the components at i_1 and i_2 have the same sign.*

The binary relation being *tied* is an equivalence relation on the index set of vectors $\boldsymbol{x}$ that generates an equivalence class defined as follows.

Definition 2 (Tie Class). *A tie class for an affine region A is the equivalence class (partitioning) of the index set for the vectors in A induced by the equivalence relation tied for A.*

[2] We assume the indices range from 0 to $n-1$ for vectors of size n.

Consider the affine region generated by the basis $\boldsymbol{v_i}$ and $\boldsymbol{c}$: $\boldsymbol{v_0} = [1\ 0\ 0\ 2]$, $\boldsymbol{v_1} = [0\ 1\ 0.5\ 0]$, $\boldsymbol{c} = [0.5\ 2\ 1\ 1]$. For this region, the indices 0 and 3 are tied because for every vector in the region the component 3 is always 2 times the component 0 , since the component 3 of the $\boldsymbol{v_i}$ and the $\boldsymbol{c}$ are 2 times the component 0. Similarly, indices 1 and 2 are tied as well. For this region, the tie equivalence class is $\{0 : \{0, 3\}, 1 : \{1, 2\}\}$

Tie Class Based Transformation of Basis Vectors. To help construct the basis vectors for the over-approximation of the output of *Relu*, we define a transformation of the set of basis vectors at the input to *Relu*. For each tie class j in the equivalence class induced, and each vector $\boldsymbol{v_i}$ in the input basis set, we construct a vector $\boldsymbol{v_i'^j}$ by setting all the components of $\boldsymbol{v_i}$ that are not in the tie class j to 0. Similarly, we get a $\boldsymbol{c^j}$ from $\boldsymbol{c}$ for each tie class j. For the example above, we have:

$$\boldsymbol{v_0'^0} = [1\ \ 0\ \ 0\ \ 2] \quad \boldsymbol{v_1'^0} = [0\ \ 0\ \ 0\ \ 0] \quad \boldsymbol{c^0} = [0.5\ \ 0\ \ 0\ \ 1]$$
$$\boldsymbol{v_0'^1} = [0\ \ 0\ \ 0\ \ 0] \quad \boldsymbol{v_1'^1} = [0\ \ 1\ \ 0.5\ \ 0] \quad \boldsymbol{c^1} = [0\ \ 2\ \ 1\ \ 0]$$

Lemma 1. *Given* $\boldsymbol{x} = \sum_i \alpha_i \boldsymbol{v_i} + \boldsymbol{c}$, *we can write* $Relu(\boldsymbol{x}) = \sum_{i,j} \alpha_i'^j \boldsymbol{v_i'^j} + \sum_j \beta_j \boldsymbol{c_j}$ *where each* $\alpha_i'^j$ *is either* α_i *or is* 0, *and each* β_j *is either* 0 *or* 1. *Moreover, the components of* $Relu(\boldsymbol{x})$ *with indices in a tie class* j *are* 0 *iff* $\alpha_i'^j$ *and* β_j *are* 0.

This lemma[3] states that there exists an oracle that, given an $\boldsymbol{x}$ in $reach[j]$, can determine whether to set each $\boldsymbol{\alpha_i'^j}$ to $\boldsymbol{\alpha_i}$ or 0 and each β_j to 0 or 1 so that we can express $Relu(\boldsymbol{x})$ in the above form. Regardless of what the oracle chooses we can always replace the condition $\boldsymbol{\alpha_i'^j} = \boldsymbol{\alpha_i} \vee \boldsymbol{\alpha_i'^j} = 0$ with $|\boldsymbol{\alpha_i'^j}| \leq 1$ as an over-approximation. Now, if we can somehow replace $\sum_j \beta_j \boldsymbol{c_j}$ with a single vector, we will have found our output affine region. The following theorem proves that we can replace this sum with $Relu(\boldsymbol{c})$.

Theorem 1. *Given* $\boldsymbol{x} = \sum_i \alpha_i \boldsymbol{v_i} + \boldsymbol{c}, |\boldsymbol{\alpha_i}| \leq 1$, *in an affine region A, there are scalars* $\alpha_i'^j$ *such that:*

1. $Relu(\boldsymbol{x}) = \sum_{i,j} \alpha_i'^j \boldsymbol{v_i'^j} + Relu(\boldsymbol{c})$
2. $|\boldsymbol{\alpha_i'^j}| \leq 1$ *for all* i *and* j.

The above theorem ensures that if we relax the condition on $\boldsymbol{\alpha_i'^j}$ to $|\boldsymbol{\alpha_i'^j}| \leq 1$, the affine region obtained an over-approximation for the Relu image of A. Given $\boldsymbol{v_i}$ and $\boldsymbol{c}$, it is easy to compute $\boldsymbol{v_i'^j}$ and $Relu(\boldsymbol{c})$ if we know what the tie classes are, since this only involves setting certain components to 0. All we need to do now is compute the tie classes for the given $\boldsymbol{v_i}$ and $\boldsymbol{c}$.

[3] A more detailed version of the paper including an appendix with all the proofs and other details is available at https://arxiv.org/abs/2110.09578.

Algorithm 3. Checking tiedness

1: **inputs:** $A, \overrightarrow{v_i}, \overrightarrow{c}, i_1, i_2$
2: **if** $\forall j : \frac{v_j^{i_1}}{v_j^{i_2}} = \frac{c^{i_1}}{c^{i_2}}$ **then return** *tied*
3: **else if** $\overrightarrow{c_{i_1}} \geq 0$ and $\overrightarrow{c_{i_2}} \geq 0$ **then**
4: **if** i_1 or i_2 component of some $\overrightarrow{x} \in A < 0$ **then return** *not tied*
5: **else return** *tied*
6: **else if** $\overrightarrow{c_{i_1}} < 0$ and $\overrightarrow{c_{i_2}} < 0$ **then**
7: **if** i_1 or i_2 component of some $\overrightarrow{x} \in A > 0$ **then return** *not tied*
8: **else return** *tied*
9: **else return** *not tied*

Computing Tie Classes. To compute tie classes, for every pair of indices i_1 and i_2, we check whether i_1 and i_2 are tied, and then group them together. One way to check if two i_1 and i_2 are in the same tie class using two LP queries involving the α_i: one which constrains the value of component i_1 of x to positive and component i_2 to negative, and vice versa. If any of these are feasible, i_1 and i_2 cannot be in the same tie class. Else, they are in the same tie class. This needs to be repeated for each pair of i_1 and i_2, which amounts to $n * (n-1)$ LP calls for n *Relu* nodes, which is inefficient. Instead, we state another property of tie classes that will allow us to compute the tie classes more efficiently:

Theorem 2. *Two indices i_1 and i_2 are in the same tie class if and only if one of the following is true:*

1. *The i_1 and i_2 components of $\boldsymbol{x}$ are always both positive.*
2. *The i_1 and i_2 components of $\boldsymbol{x}$ are always both negative.*
3. *The vector formed by the i_1 and i_2 components of the $\boldsymbol{v_k}$ and $\boldsymbol{c}$ are parallel. In other words, if v_k^l is the l-component of $\boldsymbol{v_k}$, and c^l is the l component of $\boldsymbol{c}$, then $[v_1^{i_1}, v_2^{i_1}, \cdots c^{i_1}] = k[v_1^{i_2}, v_2^{i_2}, \cdots c^{i_2}]$ for some real $k > 0$.*

Algorithm 3 uses Theorem 2 to check if i_1 and i_2 are in the same tie class. The queries in lines 5, 7, 12 and 14 can be reduced to looking for α_j such that $\sum_j \alpha_j v_j^{i_1} + c^{i_1} < 0$. Such queries can be solved via an LP call, but we use Lemma 2 to avoid LP calls and check these queries efficiently.

Lemma 2. *The maximum and minimum values of $\sum_i \alpha_i v_i$, for real α_i, fixed real v_i, constrained by $|\alpha_i| \leq 1$, are $\sum_i |v_i|$ and $-\sum_i |v_i|$ respectively.*

If the network has a lot of inherent symmetry with respect to the input permutation, it is more likely for different neurons in the same layer to be tied together, leading to larger tie classes. This, in turn, reduces the number of basis vectors required to construct our over-approximation of the *Relu* image, and improves the quality of the over-approximation. Thus, we can expect our over-approximation to perform well for checking permutation invariance.

4.2 Backward (Polytope) Propagation

Given a convex polytope $P : \boldsymbol{x}L \leq \boldsymbol{u}$, we aim to symbolically construct a region that reasonably under-approximates $\mathit{WeakestPrecond}(\mathit{Layer}, P)$. Back propagating P across the linear part of a layer is easy as it can be done precisely by simply pulling back P across the affine transform of the layer.

Back propagating it across *Relu* is challenging as $\mathit{WeakestPrecond}(\mathit{Relu}, P)$ may potentially touch all of the exponentially many "non-positive" quadrants. For each non-positive quadrant Q, relu acts on the points in the quadrant by projecting them linearly to the positive quadrant. If this projection is given by $\mathit{Relu}(\boldsymbol{x}) = \boldsymbol{x}\Pi_Q$, we have $\mathit{Relu}(\boldsymbol{x})L \leq \boldsymbol{u} \Leftrightarrow \boldsymbol{x}\Pi_Q L \leq \boldsymbol{u}$. Thus, the inverse image of P over *Relu* restricted to each quadrant is itself a polytope, giving us exponentially many polytopes in $\mathit{WeakestPrecond}(\mathit{Relu}, P)$. Therefore, exact backpropagation is infeasible, and we look for under-approximations to $\mathit{WeakestPrecond}(\mathit{Relu}, P)$.

A sound single polytope solution is to use $P \wedge \boldsymbol{x} \geq 0$ ignoring the entire "non-positive" region at the input, but this is too imprecise. Our compromise solution is to use a union of two polytopes: one that includes the positive region $P \wedge \boldsymbol{x} \geq 0$ and another that includes as much of the non-positive region as possible. To construct a polytope under-approximating the non-positive regions using inexpensive linear algebraic techniques, we use two separate methods, depending on whether P includes the 0 vector or not.

Case 1, P Does not include 0: Of all the non-positive quadrants, we choose the quadrant Q_c that has the center point of $\mathit{reach}[i]$. Then, we take the polytope corresponding to the inverse image of P over *Relu* restricted to Q_c as the under-approximation for the non-positive region. This center point based heuristic for choosing a quadrant is motivated by the fact that if the center point of $\mathit{reach}[i]$ is in Q_c, we know that at-least a part of $\mathit{reach}[i]$ must be in Q_c.

$$\boldsymbol{x}\begin{bmatrix}1 & -1\\ 1 & -1\\ 1 & -1\end{bmatrix} \leq \begin{bmatrix}2 & -1\end{bmatrix} \quad (3) \qquad \boldsymbol{x}\begin{bmatrix}0\,0\,0\\ 0\,1\,0\\ 0\,0\,1\end{bmatrix}\begin{bmatrix}1 & -1\\ 1 & -1\\ 1 & -1\end{bmatrix} = \boldsymbol{x}\begin{bmatrix}0 & 0\\ 1 & -1\\ 1 & -1\end{bmatrix} \leq \begin{bmatrix}2 & -1\end{bmatrix} \quad (4)$$

For example, consider the polytope given in Eq. 3. This touches all of the 7 non-positive quadrants, and so there will be one polytope for each of these in $\mathit{WeakestPrecond}(\mathit{Relu}, P)$. Let us say the center point $\mathit{reach}[i]$ is in the quadrant where the first component of $\boldsymbol{x}$ is negative, and all other components are non-negative. In this component, *Relu* acts by setting the first component to 0, and Π_Q is given by the identity matrix with the uppermost leftmost element set to 0. Then, following the calculations before, Eq. 4 gives us the polytope for the negative side region.

Case 2, P includes the 0 vector: If 0 is inside P, there is a high chance for the center of $\mathit{reach}[i]$ to lie inside the all-negative quadrant, and for the above method to produce $\boldsymbol{x} \leq 0$ as the non-positive polytope. While this polytope may potentially cover a large number of the points in $\mathit{reach}[i]$, the polytope touches P only at the origin. Thus, points in $\mathit{reach}[i]$ that are close to the origin may not be covered. We therefore try to do better by extending $\boldsymbol{x} \leq 0$ a region of the form $\boldsymbol{x} \leq \boldsymbol{\eta}$, where all components of $\boldsymbol{\eta}$ are non-negative.

We notice that $\boldsymbol{x} \leq \boldsymbol{\eta} \Rightarrow 0 \leq \mathit{Relu}(\boldsymbol{x}) \leq \boldsymbol{\eta}$. Thus, $\boldsymbol{\eta}$ should satisfy the soundness condition $\forall \boldsymbol{y}\ 0 \leq \boldsymbol{y} \leq \boldsymbol{\eta} \Rightarrow \boldsymbol{y}L \leq \boldsymbol{u}$. To cover as many points as possible in the region, we try to maximize the "volume" $\prod_i \eta_i$, where η_i are the components of $\boldsymbol{\eta}$. If P has a single linear inequality, we can do this by

constraining $\boldsymbol{\eta}$ to the boundary, and solving for the gradient of $\prod_i \eta_i$ to be 0. This reduces to solving a set of linear inequalities. We repeat this procedure for each inequality j in P to get an $\boldsymbol{\eta_j}$ and take the component-wise minimum to get the final $\boldsymbol{\eta}$. For example, the columns 1 and 2 of polytope 5 below give us the $\boldsymbol{\eta_1}$ and $\boldsymbol{\eta_2}$ in Eq. 6.

$$\boldsymbol{x} \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix} \leq \begin{bmatrix} 2 & 2 \end{bmatrix} \quad (5) \qquad \begin{array}{c} \boldsymbol{\eta_1} = \begin{bmatrix} 1 & 0.5 \end{bmatrix}, \boldsymbol{\eta_2} = \begin{bmatrix} 0.5 & 1 \end{bmatrix} \\ \boldsymbol{\eta} = min(\boldsymbol{\eta_1}, \boldsymbol{\eta_2}) = \begin{bmatrix} 0.5 & 0.5 \end{bmatrix} \end{array} \quad (6)$$

Thus, we backpropagate a polytope across *Relu* to get a union of two polytopes. If we repeat this process at each layer, the number of polytopes will double at each layer, leading to an exponential blowup. To avoid this, we keep this 2-polytope under-approximation only to perform inclusion check (line 10 in Algorithm 1). The polytope corresponding to the negative region is dropped before it is subsequently back propagated further into earlier layers.

4.3 Inclusion Checking and Counterexample Propagation

Our goal is to check whether *reach*$[i]$, given by basis B and center $\boldsymbol{c}$, is included in *safe*$[i]$, given as union of $P_1 : \boldsymbol{x}L_1 \leq \boldsymbol{u_1}$ and $P_2 : \boldsymbol{x}L_2 \leq \boldsymbol{u_2}$.

The first challenge in inclusion checking comes from the fact that *safe* is a disjunction of two polytopes. In the case when P comes from **Case 2** above, we notice that the two polytopes P_1 and P_2 lie entirely in the opposite sides of the plane separating the selected quadrant from the positive quadrant. This allows us to reduce the inclusion check to seeing if all points in *reach*$[i]$ on each side of the plane lie entirely inside the corresponding polytope. For **Case 1** we do not have any such separating plane, but here inclusion holds iff for all i, all points in *reach*$[i]$ above the $x_i \geq \eta_i$ plane lies in the positive side polytope. Thus, in both cases, we have reduced inclusion checking to a query of the form $(\exists \boldsymbol{\alpha} : \ \boldsymbol{x} = \boldsymbol{\alpha}B + \boldsymbol{c} \wedge \ |\boldsymbol{\alpha}| \leq 1 \wedge \ \boldsymbol{x}.\boldsymbol{v} \geq k) \Rightarrow \boldsymbol{x}L \leq \boldsymbol{u}$.

To solve the above query, we pull $\boldsymbol{x}L \leq \boldsymbol{u}$ and $\boldsymbol{x}.\boldsymbol{v} \leq k$ back to the space of $\boldsymbol{\alpha}$ using the linear transform given by B and $\boldsymbol{c}$ to get P_α and K_α respectively. This gives us a bounded polytope inclusion query, which can be solved by optimizing the objective given by each inequality of P_α with K_α as constraint. This we solve via an LP call, thus reducing inclusion checking to multiple simple LP calls.

If the inclusion fails, we obtain a point $\boldsymbol{w}$ that witnesses the violation of the inclusion at layer i. Since back-propagating this witness to the input layer to generate a counterexample is in general as hard as backpropagating the *safe* regions, in *pullBackCex* we generate multiple approximate back-propagations of $\boldsymbol{w}$ across a layer, which map to points close to $\boldsymbol{w}$ when taken across the layer. We do this by first generating several candidate back-propagated points $\boldsymbol{x}$ in the *reach* randomly. Then, we project each $\boldsymbol{x}$ towards the pullback of $\boldsymbol{w}$ with respect to the action of the layer restricted to $\boldsymbol{x}$'s quadrant. Finally, we discard all $\boldsymbol{x}$ that under the action of the layer lead to points that have euclidean distance more than D from $\boldsymbol{w}$, where D is a parameter that we tune. Doing this backwards layer by layer gives us many points in the input layer which approximately map to $\boldsymbol{w}$, and we check if these violate the property.

4.4 Example (continued from Sect. 3.1)

Details of *initPre*: For the input points $[x_{00}\ x_{01}\ x'_{00}\ x'_{01}\ ]$, $x_{00} = x'_{01}$, and $x_{01} = x'_{00}$. This can be expressed as saying that $[x_{00}\ x_{01}\ x'_{00}\ x'_{01}\ ]$ is a linear combination of the rows of: $\begin{bmatrix}1&0&0&1\\0&1&1&0\end{bmatrix}$.

Now, as the points also have components in the range $[0, 1]$, we can shift the origin to 0.5 and scale by 0.5 to get the affine region $\boldsymbol{\alpha}B_0 + \boldsymbol{c_0}$, $|\boldsymbol{\alpha}| \leq 1$ with

$$B_0 = \begin{bmatrix}0.5&0&0&0.5\\0&0.5&0.5&0\end{bmatrix} \quad \boldsymbol{c_0} = \begin{bmatrix}0.5\ 0.5\ 0.5\ 0.5\end{bmatrix}$$

Forward Propagation Across Layer 1: Now, we follow the algorithm as it pushes 1 forward across the layers of the network to get the postconditions at various points. Firstly, 1 is pushed forward across linear layer 0 by taking the pushforward with respect to W_0 and $\boldsymbol{b_0}$ to get:

$$B'_0 = \begin{bmatrix}500&-500&500&-500&-500&500&-500&500\\-500&500&-500&500&500&-500&500&-500\end{bmatrix}$$
$$\boldsymbol{c'_0} = \begin{bmatrix}0\ 0\ -1\ -1\ 0\ 0\ -1\ -1\end{bmatrix}$$

Now, the algorithm performs the tie class analysis to push B'_0 and $\boldsymbol{c'_0}$ across the *Relu* to get B_1 and $\boldsymbol{c_1}$. Here, using 3 of Theorem 2 the algorithm determines that the tie classes of the columns are $\{0, 5\}$, $\{1, 4\}$, $\{2, 7\}$,$\{3, 6\}$. We note that if $\boldsymbol{x_0}$ and $\boldsymbol{x'_0}$ are related by the permutation that swaps the components, the pairs of variables in each of the above tie class will actually have the same value. Thus, the tie class is capturing a weaker over-approximation of this strict symmetry property. Now, for each tie class, all the columns of all the basis vectors in B'_0 not in the tie class is set to 0, and collecting the resulting vectors gives us B_1; $\boldsymbol{c_1}$ is simply given by $\mathit{Relu}(\boldsymbol{c'_0})$. As before, B_1 is pushed across linear layer 1 to get B'_1. Both these matrices are given in Eq. 2 in Sect. 3.1. Again, the algorithm performs a tie class analysis, getting $\{0, 3\}$ and $\{1, 2\}$. This again is a weakening of the fact that these pairs of variables are actually equal. Note that the basis B_2 gotten on the other side of the *Relu* in this case is actually the same as B'_1.

5 Experiments

We have implemented this in Python, using the *numpy* and *scipy* libraries for linear algebra and LP solving, respectively. Our experiments were run on an Intel i7 9750H processor with 6 cores and 12 threads with 32 GB RAM. The artifacts are available for evaluation at https://github.com/digumx/permcheck/tree/nfm22.

We have compared our algorithm with the Marabou [13,14] implementation of the Reluplex [12] on a few DNNs of various sizes with the following target behavior: for n inputs, there should be n outputs so that if input i is the largest among all the inputs, output i should be 1. These networks have three layers

excluding the input layer, with sizes $2n(n-1)$, $n(n-1)$ and n respectively. Formally, we check that $0 \leq \boldsymbol{x} \leq 1 \Rightarrow |\sigma(\mathcal{N}(\boldsymbol{x})) - \mathcal{N}(\sigma(\boldsymbol{x}))| \leq \epsilon$, where σ represents the permutation sending $1 \rightarrow 2, 2 \rightarrow 3 \cdots n \rightarrow 1$ cyclically, and ϵ varies across the experiments. Note that if the network follows the target behavior, then this property should hold.

We first demonstrate our algorithm on a set of hand-crafted networks solving the above problem for which we have manually fixed the weights. The network has been manually engineered so that the first and second layers perform pairwise comparisions of the input, and the third layer combines the results of these comparisions logically to produce the output.

In general, as the input to the above DNN varies within the precondition region, the input to the *Relu* nodes can regularly switch between positive and negative. This can potentially lead to an exponential blowup in the number of case-splits. However, since permuting the inputs of this DNN leads to a more complicated permutation of the intermediate layers, intuitively we should be able to easily verify the property using an effective abstraction. The columns labelled *Safe* of Table 1 compares the time taken by our algorithm and by Marabou on these networks and demonstrates that the over-approximation and under-approximation used in our algorithm form an effective abstraction for this example, and is likely to be so for similar, symmetric networks.

We also test our algorithm on an unsafe problem using the same hand-crafted network from the previous example. To do so, we change the permutation on the output side to be the identity permutation, leading to a property that clearly should never hold. The results are given in the columns labelled *Unsafe* of Table 1 and show that our counterexample search is able to find counterexamples in a way that is competitive with Marabou, especially for networks with 8 or more inputs.

Table 1. Comparison of Marabou and our algorithm on safe and unsafe synthetic networks

Inputs	Size	Safe			Unsafe		
		Our algorithm	Marabou		Our algorithm	Marabou	
			Time	Splits		Time	Splits
3	21	0.074	4.833	2046	0.048	0.187	68
4	40	0.112	>100.8	>11234	0.074	0.202	38
5	65	0.163	>101.9	>5186	0.132	0.267	47
6	96	0.269	>100.1	>2243	0.233	0.603	60
7	133	0.493	>106.8	>1533	0.422	1.085	64
8	176	0.911	>126.5	>475	0.809	71.89	299
9	225	1.477	>183.9	>467	1.508	5.011	91
10	280	2.276	>158.7	>394	2.157	29.09	202

Finally, we compare the performance of our algorithm with Marabou on two sets of trained DNNs. The first is a set of DNNs for the same problem that have been trained using SGD to have the target behavior described above, using a large number of randomly generated input and corresponding correct output for training. We compare the algorithms on trained networks of various sizes, and with various values of ϵ.

The results (Table 2) show that for these examples, most networks are unsafe, and as the size of the network increases, both our algorithm and Marabou is able to find counterexamples. However, the time taken by Marabou increases significantly for the larger networks, eventually timing out for the largest examples, while our algorithm scales much better. The table also shows that for smaller networks Marabou performs better than our algorithm. We believe this is due to the inefficiencies in our prototype implementation compared to Marabou. For some small networks, our algorithm is unable to find a proof or counterexample, however we believe this issue can be handled with a counterexample guided refinement procedure in the future (Fig. 2).

Table 2. Comparison of Marabou and our algorithm on trained networks. the time is given in *seconds*.

Network				Our algorithm		Marabou		
n	Size	ϵ	Accuracy	Time	Result	Time	Splits	Result
3	21	0.1	94.0%	0.023	CEX	0.023	10	CEX
3	21	0.5	100%	0.249	INCONS	0.034	16	CEX
3	21	0.9	100%	0.204	INCONS	1.330	274	SAFE
5	65	0.1	97.1%	0.197	CEX	0.684	35	CEX
5	65	0.5	99.5%	0.188	CEX	0.682	35	CEX
6	96	0.1	98.0%	0.012	CEX	3.070	85	CEX
6	96	0.3	98.6%	0.018	CEX	3.138	85	CEX
7	133	0.1	87.5%	0.011	CEX	5.651	84	CEX
7	133	0.3	96.1%	0.012	CEX	5.810	86	CEX
8	176	0.1	65.7%	0.012	CEX	44.42	258	CEX
8	176	0.3	68.5%	1.584	CEX	42.80	258	CEX
9	255	0.1	58.4%	1.193	CEX	>120.3	>228	TO
9	255	0.3	70.2%	1.310	CEX	>127.9	>179	TO
10	280	0.1	20.8%	4.040	CEX	>130.4	>58	TO
10	280	0.3	31.0%	3.966	CEX	>125.0	>58	TO

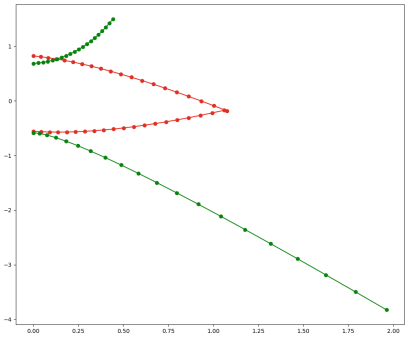

Fig. 2. Example craft paths (Color figure online)

The second set of examples involve DNNs that we have trained to solve a smaller-scale simpler version of the collision avoidance problem. Here, we consider a craft moving through 2D space with a given initial position and velocity under a given constant acceleration. The DNN must take as input the initial position, velocity and constant acceleration for a pair of crafts and determine weather they will collide. In the attached figure, the green plots show the trajectory of a non-colliding pair of crafts, and the red plots show the trajectory of a colliding pair in the dataset. There is an inherent symmetry to this problem: if we swap the two crafts, the output of the DNN should remain the same. We generated a dataset of 100000 pairs of craft position, velocity and acceleration, labeled as colliding and non-colliding. On this dataset we trained (using SGD) DNNs with various sizes, and used them to compare Marabou and our method on the problem of verifying invariance under swapping for different values of ϵ. The data is given in the Table 3.

Table 3. Comparison of Marabou and our algorithm on collision avoidance networks.

Network			Our algorithm		Marabou		
Size	ϵ	Accuracy	Time	Result	Time	Splits	Result
33	0.1	76.4%	0.059	CEX	0.123	27	CEX
33	0.5	97.2%	0.337	CEX	0.312	40	CEX
33	0.7	99.5%	1.440	INCONS	0.325	47	CEX
33	0.9	100%	1.679	INCONS	>121.0	>12466	TO
52	0.1	81.6%	0.093	CEX	0.808	26	CEX
52	0.7	98.9%	10.13	INCONS	5.692	140	CEX
52	0.9	100%	10.94	INCONS	>121.0	>4084	TO
90	0.1	90.0%	0.433	CEX	10.13	100	CEX
90	0.5	98.6%	1.906	CEX	10.07	101	CEX
90	0.9	100%	36.25	INCONS	>121.0	>745	TO
138	0.1	92.3%	0.564	CEX	27.52	108	CEX
138	0.9	99.9%	31.34	INCONS	>121.0	> 274	TO
318	0.1	91.7%	2.328	CEX	>121.0	>118	TO
318	0.3	94.9%	2.373	CEX	>121.0	>118	TO
318	0.5	96.8%	2.445	CEX	>121.0	>118	TO
488	0.1	92.5%	10.15	CEX	>121.0	>6	TO
488	0.3	94.5%	9.932	CEX	>121.0	>6	TO

The results (Table 3) demonstrate the performance of our algorithm on a realistic example. Again we find that most of these networks are unsafe. However for some networks our algorithm returns inconclusive and Marabou times out, and these networks may indeed be safe. We again find that while Marabou is faster on smaller networks, for the larger networks Marabou begins to time out more and more frequently, while our algorithm scales much better. We also find that for certain networks, our algorithm returns inconclusive, however the larger of these networks are hard for Marabou as well.

Though small in number, our benchmarks are challenging due to their size and complexity of verification. We attribute the efficiency of our approach to a number of design elements that are crucial in our approach – a layer by layer analysis, abstractions (that help reduce case-splits), under-approximations (that lead to good counterexamples), algebraic manipulations instead of LP/SMT calls, etc. A downside of our algorithm is that it may sometimes return *inconclusive*. A counterexample-guided refinement procedure can help tackle this issue.

6 Related Work

The field of DNN verification has gained significant attention in recent years. DNNs are increasingly being used in safety- and business-critical systems, making it crucial to formally argue that the presence of ML components do not compromise on the essential and desirable system-properties. Efforts in formal verification of neural networks have relied on abstraction-refinement [7,15,16], constraint-solving [1,5,6,20], abstract interpretation [9,17,18], layer-by-layer search [10,21], two-player games [22], dependency analysis [3] and several other approaches [11,23].

The most closely related work to ours is using a DNN verification engine such as Reluplex [12] and Marabou [13,14] to verify permutation invariance properties by reasoning over two copies of the network. Reasoning over multiple copies also comes up in the context of verifying Deep Reinforcement Learning Systems [8]. However, verification of DNNs is worst-case exponential in the size of the network and therefore our proposal to handle permutation invariance directly (instead of multiplying the network-size) holds a lot of promise.

Polytope propagation has been quite useful in the context of DNN verification (e.g. [19,24]). In the case of forward propagation, however, it requires computing the convex hull each time, which is an expensive. In contrast, our tie-class analysis helps us propagate the affine regions efficiently. In the backward direction, even though we rely on convex polytope propagation, we mitigate the worst-case exponential blow-up by using a 2-polytope under-approximation method that does not depend on LP or SMT solving, and is both scalable and effective.

In general, the complexity of a verification exercise can be mitigated by abstraction techniques, e.g. [2,7] for DNNs. The essential idea is to let go of an exact computation, which is achieved by merging of neurons in [7]. In [15], the authors propose construction of a simpler neural network with fewer neurons,

using interval weights, to over-approximate the output range of the original neural network. Our work is similar in spirit, in that it avoids exact computation unless really necessary for establishing the property. In practice, these techniques can even be used complementary to one another.

7 Conclusion

We presented a technique to verify permutation invariance in DNNs, based on novel forward- and backward-propagation methods. Our approach is sound (not just for permutation invariance properties, but for general safety properties too), efficient, and scalable. It is natural to wonder whether the approximately computed reach and safe regions may be refined to eliminate spurious counterexamples, and continue propagation till the property is proved or refuted. Our approach is definitely amenable to a counterexample-guided refinement. In particular, the spurious counterexamples can guide us to split *Relu* nodes (to refine over-approximations), and add additional safe regions (to refine under-approximations). This would require us to maintain sets of affine regions and convex polytopes at each layer, which is challenging but an interesting direction to pursue.

References

1. Akintunde, M., Lomuscio, A., Maganti, L., Pirovano, E.: Reachability analysis for neural agent-environment systems. In: Thielscher, M., Toni, F., Wolter, F., (eds.), Principles of Knowledge Representation and Reasoning: Proceedings of the Sixteenth International Conference, KR 2018, Tempe, Arizona, 30 October - 2 November 2018, pp. 184–193. AAAI Press (2018)
2. Ashok, P., Hashemi, V., Křetínský, J., Mohr, S.: DeepAbstract: neural network abstraction for accelerating verification. In: Hung, D.V., Sokolsky, O. (eds.) ATVA 2020. LNCS, vol. 12302, pp. 92–107. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59152-6_5
3. Botoeva, E., Kouvaros, P., Kronqvist, J., Lomuscio, A., Misener, R.: Efficient verification of relu-based neural networks via dependency analysis. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 34, no. 04, 3291–3299 (2020)
4. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
5. Dutta, S., Jha, S., Sankaranarayanan, S., Tiwari, A.: Output range analysis for deep feedforward neural networks. In: Dutle, A., Muñoz, C., Narkawicz, A. (eds.) NFM 2018. LNCS, vol. 10811, pp. 121–138. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-77935-5_9
6. Ehlers, R.: Formal verification of piece-wise linear feed-forward neural networks. In: D'Souza, D., Narayan Kumar, K. (eds.) ATVA 2017. LNCS, vol. 10482, pp. 269–286. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68167-2_19
7. Elboher, Y.Y., Gottschlich, J., Katz, G.: An abstraction-based framework for neural network verification. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification. pp, pp. 43–65. Springer International Publishing, Cham (2020)

8. Eliyahu, T., Kazak, Y., Katz, G., Schapira, M.: Verifying learning-augmented systems. In: Kuipers, F.A., Caesar, M.C. (eds.), ACM SIGCOMM 2021 Conference, Virtual Event, USA, 23–27 August 2021, pp. 305–318. ACM (2021)
9. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.: Ai2: safety and robustness certification of neural networks with abstract interpretation. In: 2018 IEEE Symposium on Security and Privacy (SP), Los Alamitos, CA, USA, IEEE Computer Society, May 2018
10. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 3–29. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_1
11. Jacoby, Y., Barrett, C., Katz, G.: Verifying recurrent neural networks using invariant inference. In: Hung, D.V., Sokolsky, O. (eds.) ATVA 2020. LNCS, vol. 12302, pp. 57–74. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59152-6_3
12. 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
13. 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
14. Kazak, Y., Barrett, C.W., Katz, G., Schapira, M.: Verifying deep-rl-driven systems. In: Proceedings of the 2019 Workshop on Network Meets AI & ML, NetAI@SIGCOMM 2019, Beijing, China, 23 August 2019, pp. 83–89. ACM (2019)
15. 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)
16. Pulina, L., Tacchella, A.: An abstraction-refinement approach to verification of artificial neural networks. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 243–257. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_24
17. Singh, G., Gehr, T., Mirman, M., Püschel, M., Vechev, M.T.: Fast and effective robustness certification. 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, December 3–8, 2018, Montréal, Canada, pp. 10825–10836 (2018)
18. Singh, G., Gehr, T., Püschel, M., Vechev, M.T.: An abstract domain for certifying neural networks. Proc. ACM Program. Lang. **3**(POPL), 41:1–41:30 (2019)
19. Sotoudeh, M., Thakur, A.V.: SyReNN: a tool for analyzing deep neural networks. In: TACAS 2021. LNCS, vol. 12652, pp. 281–302. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72013-1_15
20. Tjeng, V., Xiao, K.Y., Tedrake, R.: Evaluating robustness of neural networks with mixed integer programming. In: ICLR (2019)
21. Wicker, M., Huang, X., Kwiatkowska, M.: Feature-guided black-box safety testing of deep neural networks. In: Beyer, D., Huisman, M. (eds.), Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, 14–20 April 2018, Proceedings, Part I, volume 10805 of LNCS, pp. 408–426. Springer (2018)

22. Wu, M., Wicker, M., Ruan, W., Huang, X., Kwiatkowska, M.: A game-based approximate verification of deep neural networks with provable guarantees. Theor. Comput. Sci. **807**, 298–329 (2020)
23. Xiang, W., Tran, H., Johnson, T.T.: Output reachable set estimation and verification for multilayer neural networks. IEEE Trans. Neural Netw. Learn. Syst. **29**(11), 5777–5783 (2018)
24. Zhang, H., Shinn, M., Gupta, A., Gurfinkel, A., Le, N., Narodytska, N.: Verification of recurrent neural networks for cognitive tasks via reachability analysis. In: Giacomo, G.D. (eds.) et al., ECAI 2020–24th European Conference on Artificial Intelligence, 29 August-8 September 2020, Santiago de Compostela, Spain, August 29 - September 8, 2020 - Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020), volume 325 of Frontiers in Artificial Intelligence and Applications, pp. 1690–1697. IOS Press (2020)