The inverse problem for neural networks

Marcelo Forets[1][0000−0002−9831−7801] and
Christian Schilling[2][0000−0003−3658−1065]

[1] Universidad de la República, Uruguay
[2] Aalborg University, Denmark

Abstract. We study the problem of computing the preimage of a set under a neural network with piecewise-affine activation functions. We recall an old result that the preimage of a polyhedral set is again a union of polyhedral sets and can be effectively computed. We show several applications of computing the preimage for analysis and interpretability of neural networks.

Keywords: Neural network · Inverse problem · Set propagation · Interpretability.

1 Introduction

We study the inverse problem for neural networks. That is, given a neural network N and a set Y of outputs, we want to compute the preimage, i.e., we want to find all inputs x such that their image under the network is in Y ($N(x) \in Y$).

Computing the preimage has at least two motivations. First, it can be used for specification mining. Besides the obvious benefit of actually obtaining a potential specification, the main application is in interpretability, explaining the function the neural network encodes. This has indeed been investigated before (under the name *rule extraction*) [26,17,4]. As a second motivation, the preimage is also useful if a specification is known. In that case we can analyze whether the specification holds, e.g., whether there exist inputs leading to a set of outputs.

In this paper, we give a complete picture of the preimage computation for piecewise-affine neural networks. For this class, the preimage of a polyhedral set is again a union of (potentially exponentially many) polyhedral sets and can be effectively computed using linear programming. This result has already been obtained in the past [17]. However, that line of work seems to not be well known, a potential reason being that, at the time, piecewise-affine activation functions were not used. That work is indeed only concerned with a piecewise-affine approximation of the then-common sigmoid activation functions. Nowadays, piecewise-affine activation functions are most widely used in practice, and have also been extensively investigated in the formal-methods community. We believe that the community has fruitful applications for the preimage, and the purpose of this paper is to bring these results to the community's attention. We also show a potential use case in interpretability and two extensions by approximation and combination with forward-image computation.

1.1 Related work

The traditional take on the inverse problem for neural networks in the machine-learning community has particularly been studied for image classifiers, where the task is to either highlight which neurons [16] or input pixels [22,29] were most influential in the decision. However, these approaches do not compute the actual inverse but rather implement heuristics to map to one particular input. Many such techniques have been demonstrated to be misleading [24]. For instance, the technique in [29] remembers the inputs to a max-pooling layer, which otherwise cannot be inverted. A related concept is the autoencoder [12], where the decoder maps the output of the encoder back to the input; however, both maps are learned and thus there is no guarantee that the decoder inverts the encoder.

Another direction of inverse computation in neural networks is known as adversarial attacks [9]. The motivation is to find inputs that drive a classifier to a misclassification. For that, an input is mildly perturbed such that it crosses the network's decision boundary.

In abstract interpretation [6], the abstraction function maps an input to an abstract domain, and the concretization function maps back to the (set of) concrete values. Abstract interpretation has also been used for (forward) set propagation in neural networks with abstract domains such as intervals [26,27], polytopes [28], zonotopes [23,21], and polynomial zonotopes [13]. A related approach uses polygons: given a classifier with two-dimensional input domain, the approach partitions the input domain into the preimages of each class [25].

Maire [17] proposed an algorithm to compute the preimage of a union of polyhedra. The algorithm applies to arbitrary activation functions, but only computes an approximate solution in general. To improve scalability, Breutel and Maire use an additional approximation step based on nonlinear optimization [4]. Our goal is the exact computation of the preimage for networks with piecewise-affine activations. Although very close to this work, the above approaches assume a bounded input domain, bounded activation space, or surjective activations, which excludes the (nowadays) widely used rectified linear unit.

Computing the inverse of a function under a set is also known as set inversion, which has been mainly studied in the context of interval constraint propagation, with applications in robotics, control, or parameter estimation [11]. To reduce the approximation error, one can apply iterative forward-backward contractors [5]. We show such an experiment later. Thrun applied this idea to neural networks with bounded input and output domains as well as activation spaces [26].

Recently, a number of works study backward reachability in discrete-time neural-network control systems [19,2,7,20,14]. These approaches do not seem to be aware of the above-mentioned old works, and discuss strategies to compute an approximation of the preimage. This is due to the intractability of computing the exact preimage.

2 The preimage of a piecewise-affine neural network

In this section, we describe an algorithm to compute the preimage of a union of polyhedra under a neural network with piecewise-affine activations.

2.1 Preliminaries

A function $f : \mathbb{R}^m \to \mathbb{R}^n$ is affine if $f(x) = Cx + d$ for some matrix $C \in \mathbb{R}^{n \times m}$ and vector $d \in \mathbb{R}^n$. The function f is piecewise affine if there exists a partitioning of $\mathbb{R}^m$ such that f is affine in each partition.

A (deep) neural network (DNN)[3] N comprises k layers ℓ_i that are sequentially composed such that $N = \ell_k \circ \cdots \circ \ell_1$. Each layer ℓ_i applies an affine map of appropriate dimensions, followed by an activation function, written $\ell_i(x) = \alpha_i(W_i x + b_i)$. Activation functions are one-dimensional maps $\alpha_i : \mathbb{R} \to \mathbb{R}$, and are extended componentwise to vectors. Some common piecewise-affine activation functions are the identity, the rectified linear unit (ReLU), $\rho(x) = \max(x, 0)$, and the leaky ReLU, which is a parametric generalization with $a \geq 0$ defined as

$$\rho_a^\ell(x) = \begin{cases} x & x > 0 \\ ax & x \leq 0. \end{cases}$$

An n-dimensional half-space (or linear constraint) is characterized by a vector $c \in \mathbb{R}^n$ and a scalar $d \in \mathbb{R}$ and represents the set $\{x : c^T x \leq d\}$. A polyhedron is an intersection of half-spaces. A convenient way to write a polyhedron is in the matrix-vector form $Cx \leq d$, where the i-th row of C and d correspond to the i-th half-space. A polytope is a bounded polyhedron.

Given a function (e.g., a DNN) $f : \mathbb{R}^m \to \mathbb{R}^n$, the *image* of a set $\mathcal{X} \subseteq \mathbb{R}^m$ is $f(\mathcal{X}) = \{f(x) : x \in \mathcal{X}\} \subseteq \mathbb{R}^n$. Analogously, the *preimage* of a set $\mathcal{Y} \subseteq \mathbb{R}^n$ is $f^{-1}(\mathcal{Y}) = \{x : f(x) \in \mathcal{Y}\} \subseteq \mathbb{R}^m$. For nonempty sets and injective $g : \mathbb{R}^m \to \mathbb{R}^n$ we note the following facts [10,15]:

$$\mathcal{X} \subseteq f^{-1}(f(\mathcal{X})) \tag{1}$$

$$\mathcal{X} = g^{-1}(g(\mathcal{X})) \tag{2}$$

$$f(\mathcal{X}_1 \cup \mathcal{X}_2) = f(\mathcal{X}_1) \cup f(\mathcal{X}_2) \tag{3}$$

$$f^{-1}(\mathcal{Y}_1 \cup \mathcal{Y}_2) = f^{-1}(\mathcal{Y}_1) \cup f^{-1}(\mathcal{Y}_2) \tag{4}$$

$$f(\mathcal{X}) \cap \mathcal{Y} = f(\mathcal{X} \cap f^{-1}(\mathcal{Y})) \tag{5}$$

The inclusion in Eq. (1) is generally strict. In particular, for the class of DNNs considered here, the image of a (bounded) polytope $\mathcal{X}$ is a union of polytopes, but the preimage of a nonempty polytope (even if consisting of a single element) is a union of (unbounded) polyhedra. Because of the layer-wise architecture of DNNs, the image can be computed by iteratively applying each affine map and activation function. Below we describe how to compute the preimage in the same fashion, most of which has been described in prior work [17].

[3] Typically, *deep* neural networks are neural networks with multiple hidden layers. Here we do not make this distinction and always use the term *DNN*.

2.2 Inverse affine map

Given sets $\mathcal{X} \subseteq \mathbb{R}^m$ and $\mathcal{Y} \subseteq \mathbb{R}^n$, a matrix $W \in \mathbb{R}^{n \times m}$, and a vector $b \in \mathbb{R}^n$, for the affine map $f(x) = Wx + b$, the image of $\mathcal{X}$ is $f(\mathcal{X}) = \{f(x) : x \in \mathcal{X}\} \subseteq \mathbb{R}^n$ and the preimage of $\mathcal{Y}$ is $f^{-1}(\mathcal{Y}) = \{x : Wx + b \in \mathcal{Y}\} \subseteq \mathbb{R}^n$. For a polyhedron $\mathcal{Y}$ written as $Cy \leq d$, we can write the preimage as follows:

$$f^{-1}(\mathcal{Y}) = \{x : C(Wx + b) \leq d\} = \{x : CWx \leq d - Cb\} \tag{6}$$

Note that some of the resulting linear constraints may be redundant or contradictory. In the latter case, the preimage is empty.

Example 1. For the affine map $f(x) = \begin{pmatrix}-0.46 & 0.32\end{pmatrix} x + 2$ and the interval $\mathcal{Y} = [2, 3]$, we get the infinite band $f^{-1}(\mathcal{Y}) = \{x \in \mathbb{R}^2 : 0 \leq \begin{pmatrix}-0.46 & 0.32\end{pmatrix} x \leq 1\}$. $\lhd$

2.3 Inverse piecewise-affine activation function

Given sets $\mathcal{X}, \mathcal{Y} \subseteq \mathbb{R}^n$ and a piecewise-affine activation function α, the image of $\mathcal{X}$ can be computed as follows. Consider the partitioning $\Pi = \bigcup_j \mathcal{P}_j$ of $\mathbb{R}^n$ into the different domains $\mathcal{P}_j$ in which α is an affine function α_j. (Observe that α_j has a diagonal matrix due to componentwise application.) Then we can apply the corresponding affine map to the intersection of $\mathcal{X}$ with each $\mathcal{P}_j$:

$$\alpha(\mathcal{X}) = \alpha\big(\bigcup_j \mathcal{P}_j \cap \mathcal{X}\big) \overset{(3)}{=} \bigcup_j \alpha(\mathcal{P}_j \cap \mathcal{X}) = \bigcup_j \alpha_j(\mathcal{P}_j \cap \mathcal{X})$$

Example 2. Consider the ReLU function ρ for $n = 2$. The partitioning Π consists of four regions with the corresponding affine maps being modified identity matrices, where diagonal entry i is zero if the i-th dimension is non-positive:

$$\mathcal{P}_1 = x_1 > 0 \wedge x_2 > 0 \qquad \rho_1(x) = \begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix} x$$

$$\mathcal{P}_2 = x_1 \leq 0 \wedge x_2 > 0 \qquad \rho_2(x) = \begin{pmatrix}0 & 0\\ 0 & 1\end{pmatrix} x$$

$$\mathcal{P}_3 = x_1 > 0 \wedge x_2 \leq 0 \qquad \rho_3(x) = \begin{pmatrix}1 & 0\\ 0 & 0\end{pmatrix} x$$

$$\mathcal{P}_4 = x_1 \leq 0 \wedge x_2 \leq 0 \qquad \rho_4(x) = \begin{pmatrix}0 & 0\\ 0 & 0\end{pmatrix} x$$

Thus we get:

$$\rho(\mathcal{X}) = \rho_1(\mathcal{P}_1 \cap \mathcal{X}) \cup \rho_2(\mathcal{P}_2 \cap \mathcal{X}) \cup \rho_3(\mathcal{P}_3 \cap \mathcal{X}) \cup \rho_4(\mathcal{P}_4 \cap \mathcal{X}) \qquad \lhd$$

The preimage can be computed in an analogous way, using that $\bigcup_j \alpha_j(\mathcal{P}_j) = \alpha(\mathbb{R}^n)$. Note that, while the sets $\mathcal{P}_j$ partition $\mathbb{R}^n$, the sets $\alpha_j(\mathcal{P}_j)$ do not neces-

sarily partition $\mathbb{R}^n$ (only if α is injective).

$$\begin{aligned}\alpha^{-1}(\mathcal{Y}) &= \alpha^{-1}\big(\bigcup_j \alpha_j(\mathcal{P}_j) \cap \mathcal{Y}\big) \\ &\overset{(4)}{=} \bigcup_j \alpha_j^{-1}(\alpha_j(\mathcal{P}_j) \cap \mathcal{Y}) \qquad (7) \\ &\overset{(5)}{=} \bigcup_j \alpha_j^{-1}(\alpha_j(\mathcal{P}_j \cap \alpha_j^{-1}(\mathcal{Y}))) \qquad (8)\end{aligned}$$

One can implement a check for Eq. (7) directly, which is done in [17]. However, there are two issues with Eq. (7).

(1) Some of the resulting linear constraints may be redundant. This is also noted in [17]. For example, in the case of ReLU and assuming that $\mathcal{Y}$ contains the origin, the negative orthant is part of the preimage and can be represented by n linear constraints; instead, the equation suggests to compute the image of the negative orthant (which is the set just containing the origin), which already requires at least $n+1$ linear constraints, then computes the intersection with $\mathcal{Y}$, and finally computes the preimage.

(2) Instead of computing the preimage of the intersection of $\alpha_j(\mathcal{P}_j)$ with $\mathcal{Y}$, it may be more efficient to compute the preimage of $\mathcal{Y}$ first. The reason is that the preimage computation is more efficient for certain set representations, but the intersection may remove the structure from the set.

Eq. (8) is an attempt to mitigate these issues. While this expression looks more complicated, we can simplify it further by considering two cases. Recall that α_j is an affine activation function, i.e., it is applied componentwise, and observe that affine functions $\alpha_j : \mathbb{R} \to \mathbb{R}$ are either injective or constant. In the further case we can directly apply Eq. (2).

$$\alpha_j^{-1}(\alpha_j(\mathcal{P}_j \cap \alpha_j^{-1}(\mathcal{Y}))) = \mathcal{P}_j \cap \alpha_j^{-1}(\mathcal{Y}) \qquad (9)$$

If α_j is constant such that $\alpha_j(x) = c$ for all $x \in \mathcal{P}_j$, then for any $\mathcal{X}$ we have

$$\alpha_j(\mathcal{P}_j \cap \mathcal{X}) = \begin{cases} \{c\} & \mathcal{P}_j \cap \mathcal{X} \neq \emptyset \\ \emptyset & \mathcal{P}_j \cap \mathcal{X} = \emptyset. \end{cases}$$

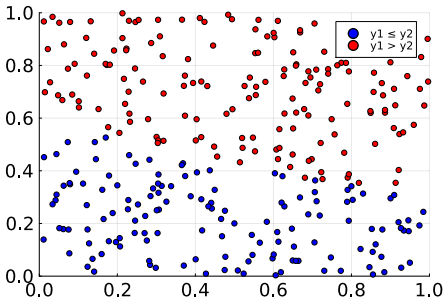
Since $\alpha_j^{-1}(\mathcal{Y}) = \mathcal{P}_j$ if $c \in \mathcal{Y}$ and $\alpha_j^{-1}(\mathcal{Y}) = \emptyset$ if $c \notin \mathcal{Y}$, we get:

$$\alpha_j^{-1}(\alpha_j(\mathcal{P}_j \cap \alpha_j^{-1}(\mathcal{Y}))) = \begin{cases} \mathcal{P}_j & c \in \mathcal{Y} \\ \emptyset & c \notin \mathcal{Y} \end{cases} \qquad (10)$$

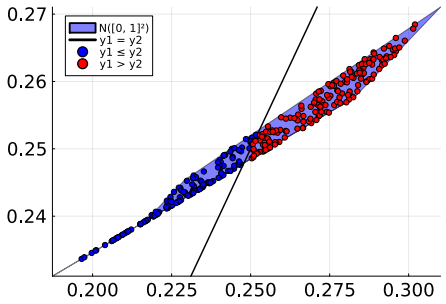
Example 3. We continue with the infinite band from Example 1, which we rename to $\mathcal{Y}$, and compute the inverse under ReLU, i.e., $\rho^{-1}(\mathcal{Y})$. Since we have two dimensions, we have the four partitions from Example 2. There are two cases. Case 1 implements Eq. (9), which applies to $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_3$. Case 2 implements

$$\ell_1(x) = \rho\left(\begin{pmatrix}0.30 & 0.53\\ 0.77 & 0.42\end{pmatrix} x + \begin{pmatrix}0.43\\ -0.42\end{pmatrix}\right) \qquad \ell_3(x) = \begin{pmatrix}0.35 & 0.17\\ -0.04 & 0.08\end{pmatrix} x + \begin{pmatrix}0.03\\ 0.17\end{pmatrix}$$
$$\ell_2(x) = \rho\left(\begin{pmatrix}0.17 & -0.07\\ 0.71 & -0.06\end{pmatrix} x + \begin{pmatrix}-0.01\\ 0.49\end{pmatrix}\right) \qquad N = \ell_3 \circ \ell_2 \circ \ell_1$$

Fig. 1: A simple DNN $N : \mathbb{R}^2 \to \mathbb{R}^2$ for classification.



(a) 300 uniform samples $x \in [0,1]^2$. Colors show the classification of $y = N(x)$.



(b) Forward image $N([0,1]^2)$ with samples from Fig. 2(a) under N.

Fig. 2: Sampled inputs and outputs for the DNN N from Fig. 1.

Eq. (10), which applies to $\mathcal{P}_4$. Since the ReLU constant is $c = (0,0)^T$, we get $\mathcal{X}_4 := \mathcal{P}_4$ because $c \in \mathcal{Y}$. Overall the result is $\mathcal{X}_1 \cup \mathcal{X}_2 \cup \mathcal{X}_3 \cup \mathcal{X}_4$, where

$$\mathcal{X}_1 := \mathcal{P}_1 \cap \rho_1^{-1}(\mathcal{Y}) = \{x \in \mathbb{R}^2 : 0 \le \begin{pmatrix}-0.46 & 0.32\end{pmatrix} x \le 1 \wedge x_1 > 0\}$$
$$\mathcal{X}_2 := \mathcal{P}_2 \cap \rho_2^{-1}(\mathcal{Y}) = \{x \in \mathbb{R}^2 : x_1 \le 0 \wedge 0 \le x_2 \le 3.125\}$$
$$\mathcal{X}_3 := \mathcal{P}_3 \cap \rho_3^{-1}(\mathcal{Y}) = \emptyset$$
$$\mathcal{X}_4 := \mathcal{P}_4 = \{x \in \mathbb{R}^2 : x_1 \le 0 \wedge x_2 \le 0\}. \qquad \triangleleft$$

2.4 Inverse piecewise-affine DNN

The procedure for a whole DNN is the straightforward alternation of Eq. (7) and Eq. (6) for each layer. In our implementation, instead of Eq. (7) we use Eq. (8) with a case distinction for whether to simplify to Eq. (9) or Eq. (10). The procedure is summarized in Algorithm 1.

Regarding complexity, given a DNN with k layers each of dimension n, it is well-known that the network can map a polyhedron to $\mathcal{O}(b^{kn})$ polyhedra, where b is the number of affine pieces in the activation function (e.g., $b = 2$ for ReLU) [18]. The same holds for the preimage.

Example 4. We consider the DNN N from Fig. 1. Each layer is two-dimensional for convenience of plotting. We view the network as a classifier based on which

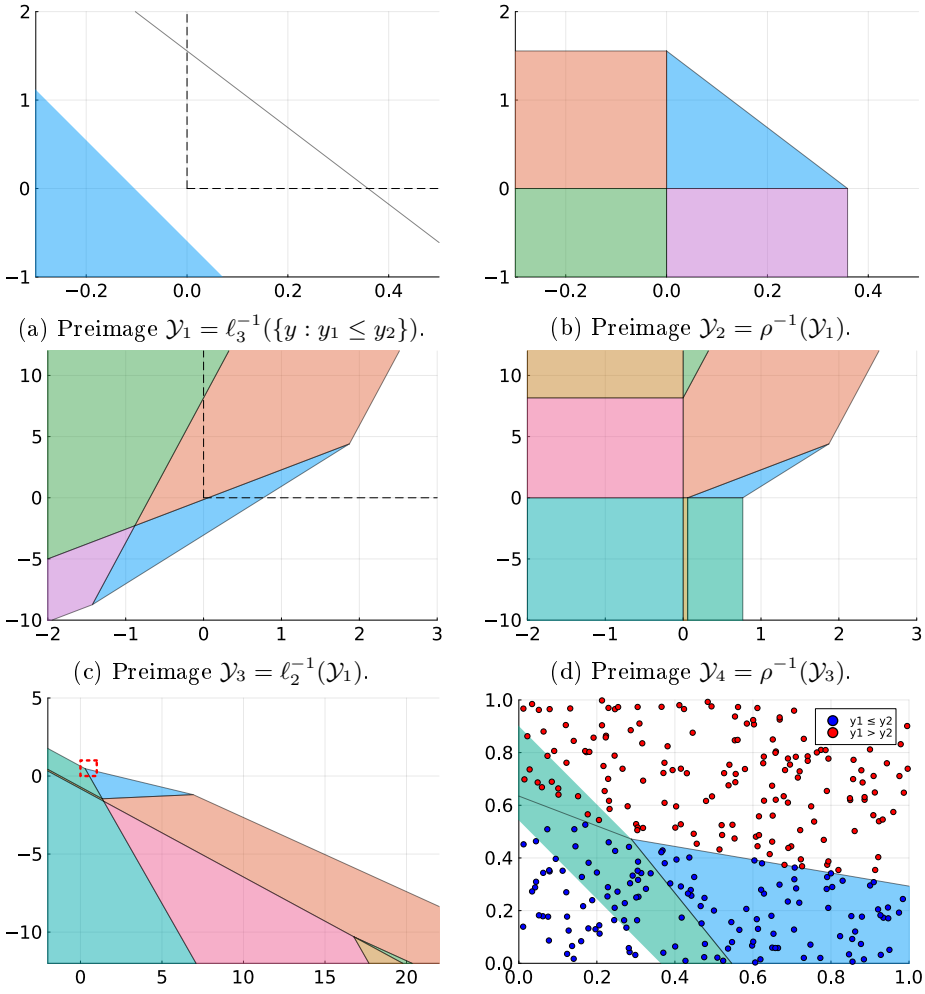
Algorithm 1 Preimage computation for piecewise-affine neural networks

```
# inverse neural network
function preimage(Z, N)
    let ℓ_1, ..., ℓ_k be the layers of N
    for ℓ in ℓ_k, ..., ℓ_1
        let W, b, α be the components of ℓ
        Y = preimage(Z, α)
        X = preimage(Y, W, b)
        Z = X
    end
    return X
end

# inverse piecewise-affine activation function
function preimage(Z, α)
    Y = ∅
    let n be the dimension of Z
    # loop over partitions and corresponding affine functions
    for (Pj, αj) in PWA_partitioning(α, n)
        let A_αj * x = b_αj be the affine representation of αj
        if iszero(A_αj) # constant case, Eq. (10)
            if b_αj ∈ Z
                Y = Y ∪ Pj
            end
        else  # injective case, Eq. (9)
            Q = preimage(Z, A_αj, b_αj)
            Y = Y ∪ (Pj ∩ Q)
        end
    end
    return Y
end

# inverse affine map, Eq. (6)
function preimage(Y, W, b)
    let C ≤ d be the constraints of Y
    M = C * W
    v = d - C * b
    return Polyhedron(M ≤ v)
end
```

of the two output values is larger. To provide some intuition, Fig. 2(a) shows samples in the input domain $[0,1]^2$ with their associated class. Fig. 2(b) shows the (forward) image of the domain, $N([0,1]^2)$, which is a union of polytopes and can be computed by standard (forward) set propagation. The figure also shows the classification boundary (black diagonal).



(a) Preimage $\mathcal{Y}_1 = \ell_3^{-1}(\{y : y_1 \leq y_2\})$.

(b) Preimage $\mathcal{Y}_2 = \rho^{-1}(\mathcal{Y}_1)$.

(c) Preimage $\mathcal{Y}_3 = \ell_2^{-1}(\mathcal{Y}_1)$.

(d) Preimage $\mathcal{Y}_4 = \rho^{-1}(\mathcal{Y}_3)$.

(e) Preimage $\mathcal{Y}_5 = \ell_1^{-1}(\mathcal{Y}_3) = N^{-1}(\mathcal{Y}_0)$ with the original domain $[0, 1]^2$ in red.

(f) Preimage $N^{-1}(\mathcal{Y}_0)$ added to Fig. 2(a).

Fig. 3: Complete example for the DNN N from Fig. 1. The black dashed lines indicate the codomain of the ReLU activation function.

Now we apply the algorithm to compute the preimage. Fig. 3 shows different snapshots of the algorithm. We compute the preimage of the set $\mathcal{C}_2 = \{y : y_1 \leq y_2\}$ (a half-space) of all inputs that classify as class 2 (blue dots). Since the last layer (ℓ_3) has an identity activation, the preimage $\mathcal{Y}_1 = \ell_3^{-1}(\mathcal{C}_2)$ is just the inverse affine map. Since this map is invertible here, the preimage is just another half-space (Fig. 3(a)). Next we compute the preimage under the ReLU activation (Fig. 3(b)). The nonnegative part of $\mathcal{Y}_1$ remains, together with the negative extensions. After computing the preimage under the next affine map (Fig. 3(c))

$$\ell_1(x) = \rho\left(\begin{pmatrix} 0.63 \\ 0.17 \\ -0.79 \end{pmatrix} x + \begin{pmatrix} 0.06 \\ -2.25 \\ -2.27 \end{pmatrix}\right) \qquad \ell_3(x) = \begin{pmatrix} 1.92 & 1.01 & 0.83 \end{pmatrix} x + 0.33$$

$$\ell_2(x) = \rho\left(\begin{pmatrix} 0.78 & 2.17 & 0.39 \\ -0.72 & -0.75 & 1.02 \\ -0.14 & -0.64 & -0.32 \end{pmatrix} x + \begin{pmatrix} -2.42 \\ -1.23 \\ 0 \end{pmatrix}\right) \qquad N = \ell_3 \circ \ell_2 \circ \ell_1$$

Fig. 4: A DNN $N : \mathbb{R}^1 \to \mathbb{R}^1$ for approximating the parabola $f(x) = x^2/20$.

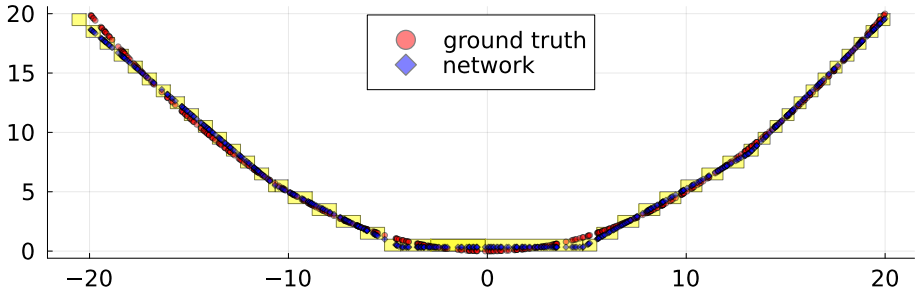


Fig. 5: Preimage computation of the DNN in Fig. 4.

we see that the purple set will get removed with the next ReLU operation. While the previous sets could have been simplified to one polyhedron, the next ReLU preimage (Fig. 3(d)) is non-convex. Finally, we compute the preimage under the last affine map (Fig. 3(e)). The box in red marks the original domain $[0, 1]^2$. Fig. 3(f) shows the preimage clipped to this domain. ◁

3 Applications and extensions

We implemented the above-described algorithm in the reachability framework JuliaReach [3], particularly in the set library LazySets [8]. In this section, we report on several experiments and discuss potential extensions. The code to repeat these experiments is available online [1].

3.1 Interpretability

We can use the preimage computation to learn about the function-approximation capabilities of a DNN. As a case study, we look at the parabola $f(x) = x^2/20$. We trained a DNN with two hidden layers, each with 3 neurons, based on 100 samples from the domain $[-20, 20]$. The resulting DNN is given in Fig. 4.

In Fig. 5 we plot the output of 500 samples of the function f (red) as well as the DNN approximation (blue). In addition, we partition the codomain $[0, 20]$

into 20 uniform intervals and compute the preimages (yellow boxes). By construction, the blue samples lie inside the preimages.

In this case we could have obtained similar results via forward-image computation because, by assuming that the DNN approximates the training dataset well, we could have just partitioned the domain $[-20, 20]$ instead. However, this does generally not work if we want to find the preimage of a subset in the codomain that the DNN does not map training data to, since we are clueless where in the domain to search. For example, we can ask for the preimage in the interval $[100, 105]$. This lets us study the generalization capabilities of the DNN. The result is a union of two intervals, $[-80.88, -77.35]$ and $[68.46, 71.48]$. The ground truth consists of the two intervals $\pm[44.72, 45.83]$. We thus see that the DNN does not generalize well, and neither does it preserve the symmetry well. However, the DNN seems to preserve both a negative and a positive preimage.

We can also easily prove that the DNN does not output negative numbers. For that we compute $N^{-1}(\{y : y \leq 0\}) = \emptyset$.

3.2 Approximation schemes

The bottleneck of the calculations is the inverse activation function, due to the partitioning. This motivates to seek approximate solutions. We are generally interested in solutions that either contain the true solution (overapproximation) or are contained in the true solution (underapproximation).

A simple approach to compute an underapproximation of the preimage considers the partitioning of the sets (as, e.g., in Fig. 3(b)) as a search space and selects only one of the sets to continue with. In general, this search may end up in an empty set (dead end), in which case one has to backtrack and pick the next set using some search heuristics (e.g., breadth-first or depth-first search). In experiments on DNNs with many neurons per layer, where explicit partitioning is infeasible, we noticed that most of the sets are indeed empty.

Instead of underapproximations, we can also consider overapproximations. Using abstract interpretation, we can choose an abstract domain to simplify the calculations. Here we consider the standard interval approximation.

As noted before, in general, the preimage of a DNN is unbounded, which makes an interval approximation useless. To make sure that we receive a bounded preimage for each layer, we consider injective activation functions and layers of the same size. We use the DNN in Fig. 1 but with leaky ReLUs ($\rho^{\ell}_{0.01}$ in layer 1 and $\rho^{\ell}_{0.02}$ in layer 2). First we compute an overapproximation $\mathcal{Y}$ of the image of the domain (i.e., $\mathcal{Y} \supseteq N([0, 1]^2)$) using interval approximation. Then we compute the preimage $N(\mathcal{Y})$, using the exact algorithm, as well as obtain an overapproximation by applying an interval approximation for inverting the leaky-ReLU activations. The approximate calculations are $\approx$100x faster than with the exact algorithm, but they also yield a coarser result, as shown in Fig. 6.

Interval approximation is also attractive for activations that are not piecewise affine. We discuss this in the next experiment.

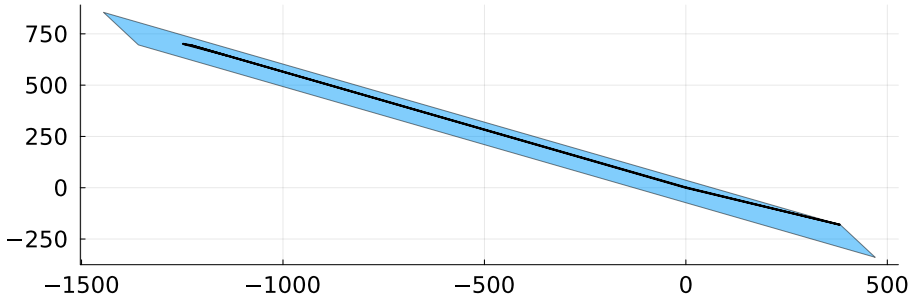


Fig. 6: Preimage computation of the DNN in Fig. 1 but with leaky ReLUs ($\rho^{\ell}_{0.01}$ and $\rho^{\ell}_{0.02}$). The thick black shape is the exact preimage. The blue shape is the preimage where we applied a box approximation inverting the activations.

$$\ell_1(x) = \sigma\left(\begin{pmatrix}-4.60248 & 4.74295\\ -3.19378 & 2.90011\end{pmatrix} x + \begin{pmatrix}2.74108\\ -1.49695\end{pmatrix}\right) \qquad N = \ell_2 \circ \ell_1$$
$$\ell_2(x) = \sigma\left(\begin{pmatrix}-4.57199 & 4.64925\end{pmatrix} x + 2.10176\right)$$

Fig. 7: A DNN to approximate the XOR function (taken from [26]).

3.3 Forward-backward computation

The approach in [26] propagates intervals forward and backward in a DNN, just as the interval approximation explained above. The benefit of this scheme is that it applies to activations that are not piecewise affine, such as the sigmoid function $\sigma(x) = 1/(1+e^{-x})$. Backward propagation of intervals is easy for strictly monotonic activations like the sigmoid: one just applies the inverse function to the lower and upper bound.

We repeat an experiment from [26]. The DNN (given in Fig. 7) uses sigmoid activations and was trained to implement a real approximation of the XOR function, i.e., it should output a value close to 1 if and only if one of the inputs is close to 1. The input domain is $[0,1]^2$ and we consider five scenarios where we add additional input or output constraints.

The results are shown in Fig. 8. Each row consists of one experiment and each column corresponds to one neuron. The horizontal axis shows the number of iterations, with iteration 0 being the initial configuration. The vertical axis shows the range of each neuron valuation. Input neuron 1 is always further constrained to a smaller interval. In experiments 1, 2, and 5, the second input neuron is also further constrained, and in the last three experiments, the output neuron is further constrained.

We can see that both the image and the preimage of the DNN can get refined by forward resp. backward propagating bounds. In the first experiment, we obtain a proof of the following statement: If $x_1 \in [0,2]$ and $x_2 \in [0.8,1]$, then

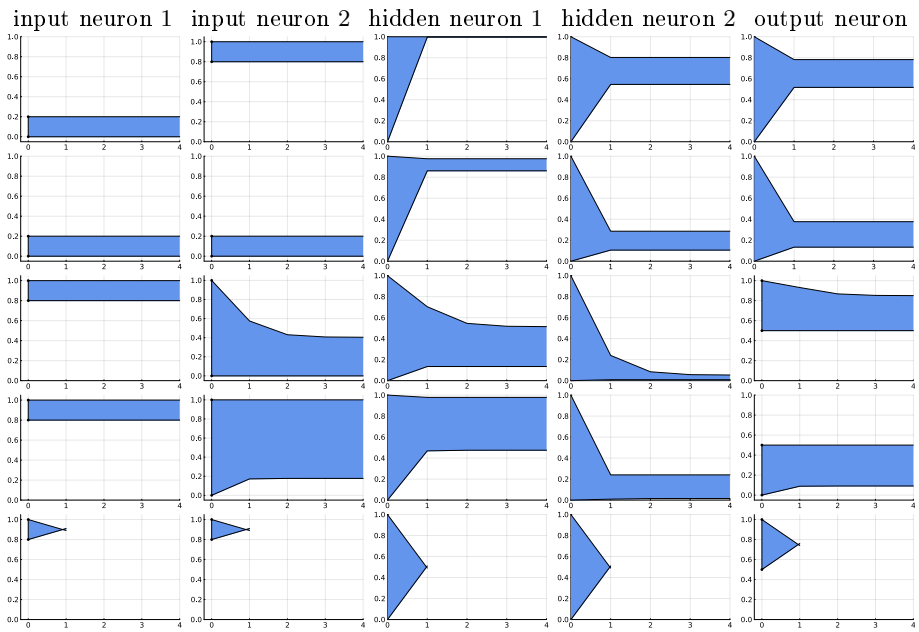


Fig. 8: Experiment from [26] for the DNN in Fig. 7. Each row consists of one experiment and each column corresponds to one neuron. In the last experiment, the sets become empty after the first iteration, which we represented graphically by converging and crossing bounds.

$N(x) \in [0.51, 0.79]$. Note that this can also be achieved with just forward-image computation (the result of which corresponds to the input- and output-neuron bounds after one iteration). In the third experiment, we obtain a proof of the following statement: If $x_1 \in [0, 2]$ and $N(x) \in [0.5, 1]$, then $x_2 \in [0, 0.41]$ and $N(x) \in [0.5, 86]$. Unlike with just forward propagation, we can derive statements about the inputs, and we also obtained tighter output bounds than what forward-image computation would ($N(x) \in [0.5, 0.94]$). The last experiment finds that the input and output constraints are incompatible, and the sets become empty. Thus we obtain a proof of the following statement: Either $x_1, x_2 \notin [0, 2]$ or $N(x) \notin [0.5, 1]$. We refer to [26] for further explanation.

4 Conclusion

In this paper, we have presented a complete picture of the computation of the preimage of a DNN with piecewise-affine activation functions. While a similar approach has been presented before [17], we believe that it has not been appreciated in the formal-methods community, and we also filled a small technical gap to allow the application to the common class of ReLU networks. We have

discussed applications in interpretability, scalability improvements via over- and underapproximations, and a combined forward and backward computation.

We see many opportunities for future work. First, the extension to other activation functions has been proposed via piecewise-affine approximation [17]. This approximation can also be implemented in a conservative way. Second, we conjecture that practical abstraction methods can be found, similar to what has been achieved in forward-image computations. Finally, we envision applications related to robustness and adversarial attacks. Once we have computed the preimage, we can use it to search for instances of interest in it (e.g., an input that maximizes another objective). Consider a classifier and the preimage of a small region around a decision boundary. The preimage contains all instances that will make the decision change with a small perturbation. We can obtain samples from the preimage set to determine whether the uncertain classification is indeed reasonable. Also, since we can compute the preimage of all decision boundaries, we can effectively obtain a partition of the input space.

Acknowledgments

We thank the anonymous reviewers for helpful comments to improve this paper. This research was partly supported by DIREC - Digital Research Centre Denmark and the Villum Investigator Grant S4OS.

References

1. Repeatability package. https://github.com/JuliaReach/AISoLA2023_RE
2. Bak, S., Tran, H.: Neural network compression of ACAS Xu early prototype is unsafe: Closed-loop verification through quantized state backreachability. In: NFM. LNCS, vol. 13260, pp. 280–298. Springer (2022). https://doi.org/10.1007/978-3-031-06773-0_15
3. Bogomolov, S., Forets, M., Frehse, G., Potomkin, K., Schilling, C.: JuliaReach: a toolbox for set-based reachability. In: HSCC. pp. 39–44. ACM (2019). https://doi.org/10.1145/3302504.3311804
4. Breutel, S., Maire, F., Hayward, R.: Extracting interface assertions from neural networks in polyhedral format. In: ESANN. pp. 463–468 (2003), https://www.esann.org/sites/default/files/proceedings/legacy/es2003-72.pdf
5. Chabert, G., Jaulin, L.: Contractor programming. Artif. Intell. **173**(11), 1079–1100 (2009). https://doi.org/10.1016/j.artint.2009.03.002
6. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL. pp. 238–252. ACM (1977). https://doi.org/10.1145/512950.512973
7. Everett, M., Bunel, R., Omidshafiei, S.: DRIP: Domain refinement iteration with polytopes for backward reachability analysis of neural feedback loops. IEEE Control. Syst. Lett. **7**, 1622–1627 (2023). https://doi.org/10.1109/LCSYS.2023.3260731
8. Forets, M., Schilling, C.: LazySets.jl: Scalable symbolic-numeric set computations. Proceedings of the JuliaCon Conferences **1**(1), 11 (2021). https://doi.org/10.21105/jcon.00097

9. Goodfellow, I.J., Shlens, J., Szegedy, C.: Explaining and harnessing adversarial examples. In: ICLR (2015), http://arxiv.org/abs/1412.6572
10. Halmos, P.R.: Naive set theory. van Nostrand (1960)
11. Jaulin, L.: A boundary approach for set inversion. Eng. Appl. Artif. Intell. **100**, 104184 (2021). https://doi.org/10.1016/j.engappai.2021.104184
12. Kingma, D.P., Welling, M.: Auto-encoding variational bayes. In: ICLR (2014), http://arxiv.org/abs/1312.6114
13. Kochdumper, N., Schilling, C., Althoff, M., Bak, S.: Open- and closed-loop neural network verification using polynomial zonotopes. In: NFM. LNCS, vol. 13903, pp. 16–36. Springer (2023). https://doi.org/10.1007/978-3-031-33170-1_2
14. Kotha, S., Brix, C., Kolter, Z., Dvijotham, K., Zhang, H.: Provably bounding neural network preimages. CoRR **abs/2302.01404** (2023). https://doi.org/10.48550/arXiv.2302.01404
15. Lee, J.: Introduction to topological manifolds. Springer Science & Business Media (2010)
16. Mahendran, A., Vedaldi, A.: Understanding deep image representations by inverting them. In: CVPR. pp. 5188–5196. IEEE Computer Society (2015). https://doi.org/10.1109/CVPR.2015.7299155
17. Maire, F.: Rule-extraction by backpropagation of polyhedra. Neural Networks **12**(4-5), 717–725 (1999). https://doi.org/10.1016/S0893-6080(99)00013-1
18. Montúfar, G., Pascanu, R., Cho, K., Bengio, Y.: On the number of linear regions of deep neural networks. In: NeurIPS. pp. 2924–2932 (2014), https://proceedings.neurips.cc/paper/2014/hash/109d2dd3608f669ca17920c511c2a41e-Abstract.html
19. Rober, N., Everett, M., How, J.P.: Backward reachability analysis for neural feedback loops. In: CDC. pp. 2897–2904. IEEE (2022). https://doi.org/10.1109/CDC51059.2022.9992847
20. Rober, N., Everett, M., Zhang, S., How, J.P.: A hybrid partitioning strategy for backward reachability of neural feedback loops. In: ACC. pp. 3523–3528. IEEE (2023). https://doi.org/10.23919/ACC55779.2023.10156051
21. Schilling, C., Forets, M., Guadalupe, S.: Verification of neural-network control systems by integrating Taylor models and zonotopes. In: AAAI. pp. 8169–8177. AAAI Press (2022). https://doi.org/10.1609/aaai.v36i7.20790
22. Simonyan, K., Vedaldi, A., Zisserman, A.: Deep inside convolutional networks: Visualising image classification models and saliency maps. In: ICLR (2014), https://arxiv.org/abs/1312.6034
23. Singh, G., Gehr, T., Mirman, M., Püschel, M., Vechev, M.T.: Fast and effective robustness certification. In: NeurIPS. pp. 10825–10836 (2018), https://proceedings.neurips.cc/paper/2018/hash/f2f446980d8e971ef3da97af089481c3-Abstract.html
24. Sixt, L., Granz, M., Landgraf, T.: When explanations lie: Why many modified BP attributions fail. In: ICML. PMLR, vol. 119, pp. 9046–9057 (2020), http://proceedings.mlr.press/v119/sixt20a.html
25. Sotoudeh, M., Tao, Z., Thakur, A.V.: SyReNN: A tool for analyzing deep neural networks. Int. J. Softw. Tools Technol. Transf. **25**(2), 145–165 (2023). https://doi.org/10.1007/s10009-023-00695-1
26. Thrun, S.: Extracting symbolic knowledge from artificial neural networks. Tech. rep., University of Bonn (1994), https://www.cs.cmu.edu/~thrun/papers/thrun.nn_rule_extraction.ps.gz

27. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Formal security analysis of neural networks using symbolic intervals. In: USENIX Security Symposium. pp. 1599–1614 (2018), https://www.usenix.org/conference/usenixsecurity18/presentation/wang-shiqi
28. Yang, X., Johnson, T.T., Tran, H., Yamaguchi, T., Hoxha, B., Prokhorov, D.V.: Reachability analysis of deep ReLU neural networks using facet-vertex incidence. In: HSCC. ACM (2021). https://doi.org/10.1145/3447928.3456650
29. Zeiler, M.D., Fergus, R.: Visualizing and understanding convolutional networks. In: ECCV. LNCS, vol. 8689, pp. 818–833. Springer (2014). https://doi.org/10.1007/978-3-319-10590-1_53