Polyhedral Complex Extraction from ReLU Networks using Edge Subdivision

Arturs Berzins [1 2]

Abstract

A NN consisting of piecewise affine building blocks, such as fully-connected layers and ReLU activations, is itself a piecewise affine function supported on a polyhedral complex. This complex has been studied to characterize theoretical properties of NNs and linked to geometry representations, but, in practice, extracting it remains a challenge. Previous works subdivide the regions via intersections with hyperplanes induced by each neuron. Instead, we propose to subdivide the edges, leading to a novel method for polyhedral complex extraction. This alleviates computational redundancy and affords efficient data-structures. A key to this are sign-vectors, which encode the combinatorial structure of the complex. Our implementation (available on GitHub) uses standard tensor operations and can run exclusively on the GPU, taking seconds for millions of cells on a consumer grade machine. Motivated by the growing interest in neural shape representation, we use the speed and differentiablility of our method to optimize geometric properties of the complex. Our code is available on GitHub[1].

1. Introduction

A NN is a CPWA function if it is a composition of CPWA operators, most notably fully-connected layers and rectified linear unit (ReLU) activations. The CPWA nature induces a polyhedral complex decomposing the input domain, which has provided an additional avenue to study NNs in terms of their expressivity, robustness, and training techniques. The geometry of the partition has also been linked to max-affine spline operators, power-diagrams (Balestriero et al., 2020), and implicit shape representations (Lei et al., 2021; Humayun et al., 2023).

The diverse range of applications motivates a computational method for extracting this complex. Previous approaches include formulating this as a mixed-integer linear program (Serra et al., 2018) for region counting or employing state-flipping to extract a 2D level-set in 3D space (Lei et al., 2021). However, the most widely discussed approach is region subdivision (Hanin & Rolnick, 2019; Humayun et al., 2023). The key idea is to sequentially consider each neuron of each layer, calculating the affine map on every existing region and determining whether the affine hyperplane cuts the region in two (see Figure 1).

However, we observe a redundancy in region subdivision (illustrated in Figure 3a and detailed in Section 3) which is due to the continuity of the activation function. We alleviate this redundancy by instead subdividing the *edges*. A key to our approach are sign-vectors, which for every cell of the complex indicate the neuron pre-activation signs and fully encode the combinatorial structure of the complex.

In practice, since we track only the vertices and bounded edges (as opposed to higher-dimensional faces with arbitrary number of facets), we benefit from simple data-structures and operations, and can run fully on the GPU. This allows to handle millions of cells in seconds, outperforming a previous region-based approach roughly 20 times while also being agnostic to the input dimension D. In lower dimensions, the fast and differentiable access to the polyhedral complex allows to optimize geometric properties of the extracted complex, which we demonstrate in a novel experiment. However, the use in $D > 8$ is impractical even for small networks due to the known exponential growth of the complex (see Figure 4a). We argue the algorithm is linear time and memory and provide our log-linear `PyTorch` implementation, hoping to facilitate the theoretical study of CPWA NNs and their application in geometry. For more details, we refer to our full work (Berzins, 2023).

2. Background

We will assume familiarity with polyhedral complexes and hyperplane arrangements, referring to a more thor-

[1]Department of Mathematics and Cybernetics, SINTEF, Oslo, Norway [2]Department of Mathematics, University of Oslo, Oslo, Norway. Correspondence to: Arturs Berzins <arturs.berzins@sintef.no>.

Presented at the 2^{nd} Annual Workshop on Topology, Algebra, and Geometry in Machine Learning (TAG-ML) at the 40^{th} International Conference on Machine Learning, Honolulu, Hawaii, USA. 2023. *Copyright 2023 by the author(s).*

[1]github.com/arturs-berzins/relu_edge_subdivision

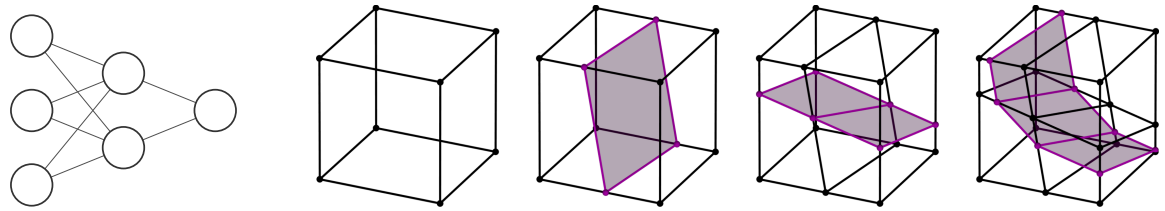


Figure 1. In CPWA NNs, each neuron of each layer sequentially subdivides the polyhedral complex. Each neuron of the first hidden layer contributes an affine hyperplane. Each neuron of the deeper layers contributes a folded hyperplane. Illustrated is the subdivision of a cubic domain in the $D = 3$ input space by the shown NN. While previous methods subdivide the regions (highest dimensional cells), our method subdivides edges.

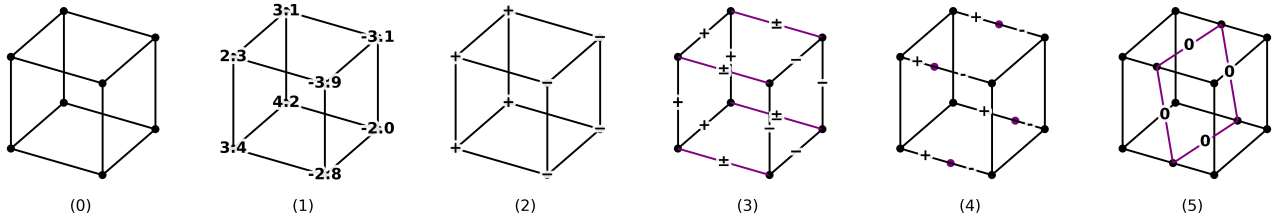


Figure 2. Steps of a single iteration of *edge subdivision*. Starting with the current 1-skeleton (0), evaluate the NN at the vertices (1) and determine the sign of the relevant neuron (2). If the signs of a vertex pair sharing an edge differ, the hyperplane must intersect this edge (3). This intersection is a new vertex whose location interpolates the coordinates and values of the vertex pair and splits the edge in two (4). To build new edges, connect the new vertices sharing a face (5).

ough treatment of this topic in the context of ReLU networks by Grigsby & Lindsey (2022) and the references therein. Analogous to an affine hyperplane, a *folded hyperplane* is the kernel of the pre-activation of a neuron: $H_i^{(l)} := \left\{ \mathbf{x} \in \mathbb{R}^D \mid \mathbf{W}_i^{(l)} \mathbf{x}^{(l-1)}(\mathbf{x}) + b_i^{(l)} = 0 \right\}$, where $\mathbf{x}^{(l)}$ is the output of the l-th layer of a ReLU NN. It is well known that the *folded hyperplane arrangement* $\mathcal{H}$ partitions the input space into a polyhedral complex $\mathcal{C}(\mathcal{H})$. Any point $\mathbf{x} \in \mathbb{R}^D$ and any cell $C \in \mathcal{C}(\mathcal{H})$ can be assigned a *sign-vector* , whose entries $+$, , 0 indicate the sign of the pre-activation or, geometrically, if $\mathbf{x}$ is right, left, or exactly on the folded hyperplane. The sign-vectors encode the combinatorial structure of the complex. We call $y \in \mathcal{C}$ an *ascendant* of $x \in \mathcal{C}$ if $x \quad y$ and a *parent* of $x \in \mathcal{C}$ if $x \quad y$ and no $z \in \mathcal{C}$ satisfies $x \quad z \quad y$.

3. Method

Due to the continuity of the activation function, all folded hyperplanes are continuous across each other (Figure 3a). If each region is considered independently, upon the intersection with a new folded hyperplane, each new vertex is discovered 2^{D-1} times in V-representation, since an edge has 2^{D-1} ascendant regions in an unbounded arrangement. Similarly, in H-representation, the hyperplane redundancy check performed via linear programming arrives at the same conclusion on all 2^{D-1} ascendant regions of the shared edge.

We resolve this redundancy by taking into account the continuity and using only the unique vertices and edges, i.e. the 1-skeleton. For each neuron in each layer we subdivide the edges in five steps illustrated in Figure 2 and detailed in the following.

To understand how the 1-skeleton is subdivided, i.e. how new 0- and 1-cells are created, consider a new hyperplane H cutting a k-cell C. Their intersection $C^0 = C \setminus H$ is a new $(k-1)$-cell. Additionally, H splits C into two new k-cells $C^+ = C \setminus H^+$ and $C^- = C \setminus H^-$. So there are exactly two mechanisms for creating $(k-1)$-cells: (i) splitting a $(k-1)$-cell with H which preserves the dimension and (ii) intersecting a k-cell with H which lowers the dimension. Focusing on $k = 0, 1, 2$ as sources for new $0, 1$-cells leads us to edge subdivision.

Let $\mathcal{V}$ and $\mathcal{E}$ be the set of all vertices and edges at the current iteration (**step 0**). Let H be the next folded hyperplane to intersect with and recall that it behaves like an affine hyperplane on each region, folding at its facets. Per generality assumption, no vertex in $\mathcal{V}$ will intersect H. Each vertex in $\mathcal{V}$ is $+$ or a sign w.r.t. H. These signs can be determined from the pre-activation values of the neuron corresponding to H, obtained by simply evaluating the NN at the vertices (**steps 1, 2**).

We call $E \in \mathcal{E}$ a *splitting* edge if H cuts E. Splitting edges can be identified by their two vertices having opposite signs (**step 3**), which we label V^+, V^-. The new vertex $V^0 = E \setminus H$ on the splitting edge E can be computed by linearly interpolating the positions of V^+, V^- weighted by

their pre-activation values (**step 4**). This new vertex takes the sign 0 w.r.t. H. The old splitting edge E is removed from E and the two new *split* edges $E^+ = E \setminus H^+$ with vertices V^+, V^0 and $E \quad = E \setminus H \quad$ with vertices $V \quad , V^0$ are added to E. The new signs of E^+, V^+ and $E \quad , V \quad$ w.r.t. H are trivially $+$ and $\quad$, respectively.

This completes intersecting and splitting edges with the folded hyperplane. However, new edges are also formed where H intersects 2-faces. We call F a *splitting* 2-face if H cuts F. We call their intersection $E^0 = F \setminus H$ an *intersecting* edge. Every bounded splitting 2-face has exactly two splitting edges. A naive approach is doing an adjacency check between all pairs of splitting edges. However, the quadratic complexity is prohibitive even for moderately sized NNs.

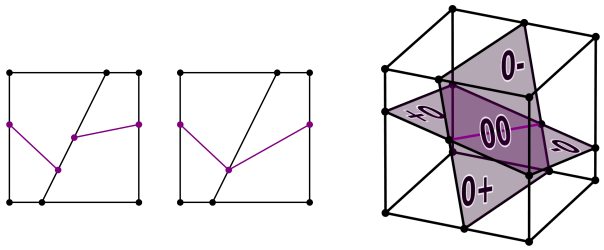
Instead, we implicitly build the splitting faces. In general, we can determine the parents of a cell by *perturbing* each zero in the sign-vector $\quad$ to $+$ or $\quad$ (example in Figure 3b). Since $\quad$ of an interior k-cell has $(D \quad k)$ zeros, this process constructs all $2(D \quad k)$ parent sign-vectors. For a boundary cell, we perturb the zeros only toward the positive interior.

During **step 5**, we end up with a list of splitting 2-faces each pointing to a single splitting edge. In this list, each 2-face comes up exactly twice. The two edges associated with the same 2-face must be paired, which can in principle be done in linear time and memory, e.g., using hash-tables. Lastly, it remains to add the intersecting edge to E. Its sign-vector is inherited from the splitting face with a 0 appended w.r.t. H.

Complexity analysis All the steps of edge subdivision can be implemented in linear time and memory complexity in the number of vertices jVj, edges jEj, or splitting edges $\mathsf{j\hat{E}j}$. We argue further that $O(\mathsf{jEj})$ and $O(\mathsf{j\hat{E}j})$ can be replaced with $O(\mathsf{jVj})$, concluding that the total algorithm is $O(\mathsf{jVj})$. The number of vertices can be further related to the NN architecture.

4. Experiments

Implementation We implement the algorithm in `PyTorch`. Since only vertex positions and bounded edges with exactly two vertices are stored, edge subdivision can run efficiently and exclusively on the GPU. The steps 0-4 can be implemented using standard tensor operations. However, restricting ourselves to standard operations available in `PyTorch`, step 5 can only be implemented in sub-optimal log-linear time via sorting. We hope to address this in the future, noting that performant hashing on the GPU with custom length keys is an open research problem. The time and memory behaviour of our implementation, as well as the size of the complex are measured in Figure 4a. We also compare to a recent method for $D = 2$ by Humayun et al. (2023), measuring a speed-up of 20 times and more for larger networks (Figure 4b). In additional experiments, we also perform validation, confirm the predicted log-linear scaling of the implementation, and study the spatial distribution of the vertices (Berzins, 2023).



(a) (b)

Figure 3. (a) Motivation in $\mathsf{D} = 2$: due the continuity of the activation, the two new edges share the same vertex on the common edge of the two regions. Processing each region independently is redundant. (b) The parenting $(\mathsf{k}+1)$-faces of a k-face can be obtained by perturbing each zero in its sign-vector at a time. Here, $\mathsf{k} = 1$ and the first 6 entries of the sign-vector w.r.t. the boundary planes are hidden for visual clarity since they are all $+$.

Optimizing the complex As described in Sections 1, the access to the exact polyhedral complex is intriguing in many applications. Since most of these have been demonstrated before, we focus a novel experiment in which an optimization objective is formulated on the geometric properties of the extracted complex, shown in Figure 5. We start with a $D = 2$ ReLU NN with two hidden layers of 50 neurons each, trained as a neural implicit representation of a bunny. Then, in each iteration we extract the polyhedral complex and compute the *shape compactness* $c = 4\pi A/P^2$ as the ratio of the area A and the perimeter P. Using c as the loss, the bunny shape converges to a circle in 100 iterations using a standard Adam optimizer. In general, any loss that depends on the vertex positions can be formulated, e.g. edge lengths, angles, areas, volumes, curvatures, and other quantities from discrete differential geometry.
In our full work (Berzins, 2023), we also study pruning of NN parameters and a modification of edge subdivision to speed up the extraction a specific iso-level-set, i.e. decision boundary, akin to Lei et al. (2021), illustrated in Figure 6.

5. Conclusions

We observed a redundancy in subdividing regions and alleviated it by subdividing edges instead, leading to a novel method for extracting the exact polyhedral complex from ReLU NNs. We exploited the structure of the complex encoded in the sign-vectors, as well as simple data structures and operations on the GPU, outpacing a previous method 20-fold. The speed and differentiability allowed us to consider a novel application in which a loss is formulated on the ex-