Polyhedral Complex Extraction from ReLU Networks using Edge Subdivision

Arturs Berzins [1 2]

Abstract

A neural network 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 previously studied to characterize theoretical properties of neural networks, but, in practice, extracting it remains a challenge due to its high combinatorial complexity. A natural idea described in previous works is to subdivide the regions via intersections with hyperplanes induced by each neuron. However, we argue that this view leads to computational redundancy. Instead of regions, we propose to subdivide edges, leading to a novel method for polyhedral complex extraction. A key to this are sign-vectors, which encode the combinatorial structure of the complex. Our approach allows to use standard tensor operations on a 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. The code is available on GitHub[1].

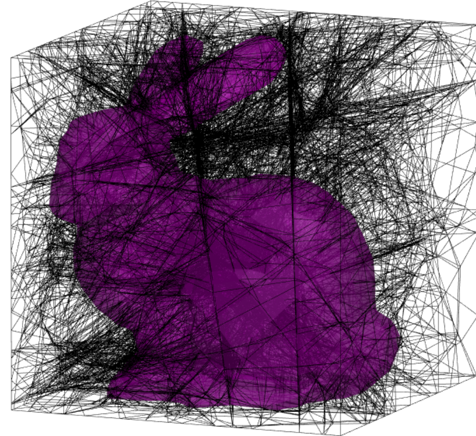

Figure 1. Example of the polyhedral complex extracted from a ReLU NN trained on the signed distance field of a $D = 3$ Stanford bunny.

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 discrete partitioning of the input domain, which provides an additional avenue to study NNs in terms of their expressivity, robustness, training techniques, and unique geometry.

It is known that each affine piece is supported on a convex polyhedral set. The collection of these polyhedral sets forms a polyhedral complex, which is induced by an arrangement of folded hyperplanes representing decision boundaries of neurons (Grigsby & Lindsey, 2022). This complex has been linked to max-affine spline operators, which possess intriguing geometric properties and allow for joint optimization of the domain partition and spline coefficients (Balestriero & Baraniuk, 2021). CPWA NNs have also been used in neural implicit shape representations for boundary mesh extraction (Lei & Jia, 2020) and visualization (Humayun et al., 2022; 2023).

The diverse range of applications motivates a computational method to extract the complex. A trivial approach is to evaluate neuron states at sampled points. While this suffices to visualize an estimated domain partition or compute a lower bound on the number of regions, it does not provide the exact complex. Proposed exact approaches include formulating this as a mixed-integer linear program (Serra et al., 2018) or employing state-flipping (Lei & Jia, 2020). However, the most natural and widely discussed approach is region subdivision, where the complex's regions are successively subdivided from neuron to neuron and layer to layer (Raghu et al., 2017; Hanin & Rolnick, 2019a; Wang, 2022; Humayun et al., 2022; 2023).

Our method is motivated by the observation of redundancy in region subdivision. The continuity of the activation function ensures that a folded hyperplane remains continuous

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

Proceedings of the 40th International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023. Copyright 2023 by the author(s).

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

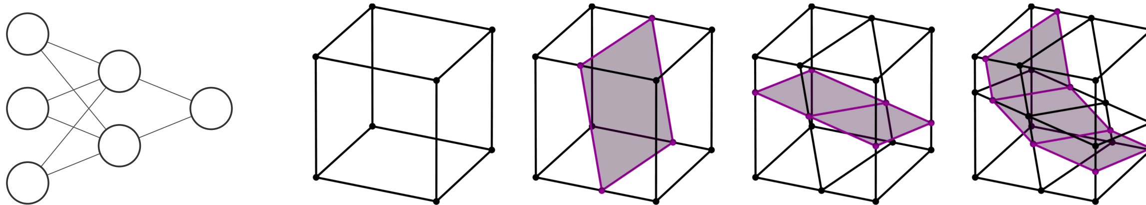

Figure 2. In CPWA NNs, each neuron of each layer sequentially subdivides the polyhedral complex. Each neuron of the first hidden layer contributes an affine hyperplane. Each neuron of the deeper layers contributes a folded hyperplane. Illustrated is the subdivision of a cubic domain in the $D = 3$ input space by the shown NN. While previous methods subdivide the regions (highest dimensional cells), our method subdivides edges.

across another fold (see Figure 4). However, considering each region independently leads to computing the same new vertex or, alternatively, identifying the same redundant hyperplane on all 2^{D-1} regions sharing a common edge, where D is the dimension of the input space. Our method alleviates this redundancy by leveraging continuity and disregarding the regions, instead using solely the unique vertices and edges, i.e. the 1-skeleton. The key idea to *edge subdivision* is to sequentially consider each neuron, i.e. folded hyperplane, evaluate all vertices with the NN and compare the signs of a vertex pair sharing an edge. If the signs differ, linear interpolation determines the location of a new vertex. The edges containing the connectivity information are updated accordingly. For this, we propose to leverage sign-vectors, which indicate the pre-activation sign of every neuron at every point or for every cell of the complex and altogether encode the combinatorial structure of the whole complex.

Our edge subdivision approach is naturally parallel and the use of sign-vectors affords additional structure, which allows to use basic tensor operations in standard ML frameworks and benefit from the GPU. This allows to handle millions of elements in seconds. The method and the implementation are also agnostic to the input dimension D. However, the use in $D > 8$ is impractical even for small networks due to the exponential growth of the complex (see Figure 8).

Our contributions are summarized as follows:

- A novel method to extract the polyhedral complex of a ReLU NNs in general dimensions with a focus on performance.
- A novel set of experiments directly optimizing the geometric properties of the complex enabled by the fast and differentiable access to the polyhedral complex.
- An open source implementation[2] using standard tensor operations in PyTorch and leveraging the GPU.

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

2. Related work

CPWA NNs A NN is itself a CPWA function if it is a composition of affine operators, such as fully-connected layers, convolutional layers, skip connections and CPWA activation functions, such as (leaky) ReLU, absolute value, hard hyperbolic tangent, hard sigmoid, and max-pooling. Many previous authors have investigated CPWA NNs and fully-connected NNs with ReLU activation in particular. Examples include the study of their expressivity in terms of the number of affine regions, (Pascanu et al., 2013; Montúfar et al., 2014; Telgarsky, 2015; 2016; Raghu et al., 2017; Serra et al., 2018; Hanin & Rolnick, 2019a; Sattelberg et al., 2020; Wang, 2022), their connection to adversarial robustness (Jordan et al., 2019; Hein et al., 2019; Daróczy, 2022), max-affine spline operators, vector quantization, and K-means clustering (Balestriero & Baraniuk, 2021), batch-norm (Balestriero & Baraniuk, 2022), affine constraint enforcement (Balestriero & LeCun, 2022), reverse-engineering (Rolnick & Kording, 2020) and the geometry of the regions (Balestriero et al., 2019; Balestriero & Baraniuk, 2021; Grigsby & Lindsey, 2022).

Number of regions The maximum number of regions is known to be polynomial in the width and exponential in the depth and input dimension of the NN (Raghu et al., 2017; Montúfar et al., 2014). In practice, however, both randomly initialized and trained NNs have a number of regions which is much smaller, with the growth being only polynomial in the number of neurons, but still exponential in the number of input dimensions (Hein et al., 2019; Hanin & Rolnick, 2019b;a) making the problem of counting regions NP-hard Wang (2022). As a consequence, for high-dimensional input-spaces even very small NNs have an extremely large number of regions, making their enumeration difficult.

Serra et al. (2018) devise a mixed-integer linear program to count the number of regions in a ReLU-network with an arbitrary input dimension. It is demonstrated that a NN trained on MNIST with 784-dimensional input and a total of just 22 hidden neurons generates $O(10^7)$ regions which takes tens of hours to count on a server-grade machine.

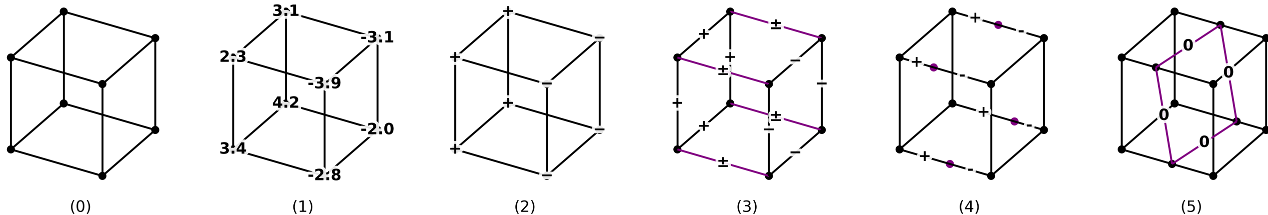

Figure 3. Steps of a single iteration of *edge subdivision*. Starting with the current 1-skeleton (0), evaluate the NN at the vertices (1) and determine the sign of the relevant neuron (2). If the signs of a vertex pair sharing an edge differ, the hyperplane must intersect this edge (3). This intersection 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).

2.1. Complex extraction

Many works provide illustrations of the regions of a 2D input space, which can be acquired by determining the neuron states at points sampled on the image grid. While this serves as an approximation, there are two known methods operating on the exact complex: region subdivision and marching.

Region subdivision Several works describe (Raghu et al., 2017) and implement (Hanin & Rolnick, 2019a; Wang, 2022; Humayun et al., 2022; 2023) region subdivision as a method to extract the exact polyhedral complex from a ReLU-network. Starting with an initial polytope, the idea is to sequentially consider each neuron of each layer. For each neuron calculate the affine map on every existing polytope and determine whether the hyperplane cuts the region in two. Our method builds upon this interpretation, but, instead of the regions, subdivides the edges to solve the redundancy in neighbouring regions.

Marching Lei et al. (2021) propose *Analytical Marching* to extract the 0-isosurface of a CPWA neural implicit shape with a bounded CPWA 2D boundary in 3D space. The algorithm is initialized by identifying a point on the 0-isosurface and the corresponding activation pattern or state of the face. Each edge of a face is the intersection of the face plane and a boundary plane induced by the affine map of some other neuron in the NN. Consequentially, a vertex of a face is the intersection of the face plane and two boundary planes. However, not all potential edges and vertices are valid, so it is checked whether they have the same state, i.e., whether they lie on the same side of all boundary planes as the face itself. Each valid edge is then used to pivot to a neighboring face by flipping the activation corresponding to the edge neuron. Analytical Marching serves as an exact alternative to classic mesh extraction methods, such as marching cubes, offering a trade-off between precision and performance. However, it is unclear how the method generalizes to the full volumetric complex and higher dimensions.

3. Background

It is well known that each of the regions supporting the CPWA NN is an intersection of affine halfspaces forming a convex polyhedral set. Together they partition the input space into a polyhedral complex (Balestriero et al., 2019; Hanin & Rolnick, 2019a; Grigsby & Lindsey, 2022).

We start by introducing the relevant terminology from the classic theory on polyhedral complexes and hyperplane arrangements. We then generalize to folded hyperplane arrangements due to CPWA NNs. Lastly, we discuss the intersection-poset and sign-vectors as a means to exploit the combinatorial structure of the folded hyperplane arrangement.

3.1. Polyhedral complexes

We start by reviewing select facts about polyhedral complexes and refer to a more thorough treatment of the topic in the context of geometry (Grunbaum & Ziegler, 2003; Grunert, 2016) and ReLU networks (Grigsby & Lindsey, 2022).

A *hyperplane* $H := \left\{\mathbf{x} \in \mathbb{R}^D | \mathbf{w}^\top \mathbf{x} - b = 0\right\}$ is the zero-level set of an affine map with the slopes $\mathbf{w} \in \mathbb{R}^D$ and a threshold $b \in \mathbb{R}$. We will assume that all hyperplanes are *non-degenerate*, meaning $\mathbf{w} \neq \mathbf{0}$. The sublevel set H^- and the super-level set H^+ are the negative and positive *half-spaces*, respectively.

A *polyhedral set* $\mathcal{P}$ in $\mathbb{R}^D$ is the closure of an intersection of finitely many half-spaces $H_1^+, ..., H_m^+ \subseteq \mathbb{R}^D$. This is called the *H-representation* of $\mathcal{P}$. The *dimension* of the polyhedral set $\mathcal{P}$ is the dimension of its affine hull.

A hyperplane H in $\mathbb{R}^D$ is a *cutting hyperplane* of $\mathcal{P}$ if there exist $\mathbf{x}_1, \mathbf{x}_2 \in \mathcal{P}$ with $\mathbf{x}_1 \in \mathcal{P} \cap H^+$ and $\mathbf{x}_2 \in \mathcal{P} \cap H^-$.

A hyperplane H in $\mathbb{R}^n$ is a *supporting hyperplane* of $\mathcal{P}$ if $H \cap \mathcal{P} \neq \emptyset$ and H does not cut $\mathcal{P}$.

The intersection $F = H \cap \mathcal{P}$ is a *face* of $\mathcal{P}$ for some supporting hyperplane H of $\mathcal{P}$. $F = \emptyset$ and $F = \mathcal{P}$ are *improper* faces of $\mathcal{P}$, otherwise F is *proper*. A k*-face* of $\mathcal{P}$ is a face of $\mathcal{P}$ of dimension k. A 0-face is a *vertex*, 1-face is an *edge*, and $(D-1)$-face is a *facet*. If $\mathcal{P}$ is bounded, its *V-*

representation is its set of vertices with the convex hull $\mathcal{P}$.
A *polyhedral complex* $\mathcal{C}$ of dimension D is a finite set of polyhedral sets of dimension $k = 0..D$, called the *cells* of $\mathcal{C}$, such that (i) if $C \in \mathcal{C}$ then every face of C is in $\mathcal{C}$; (ii) if $B, C \in \mathcal{C}$ then $B \cap C$ is a single mutual face of both B and C.
The *domain* of $\mathcal{C}$, denoted $|\mathcal{C}|$, is the union of its cells. Conversely, we call $\mathcal{C}$ the *polyhedral decomposition* of the domain $|\mathcal{C}|$.
A *polyhedral subcomplex* of $\mathcal{C}$ is a subset $\mathcal{C}' \subset \mathcal{C}$ such that for every cell C in $\mathcal{C}'$, every face of C is also in $\mathcal{C}'$. The *k-skeleton* of $\mathcal{C}$, denoted $\mathcal{C}_k$, is the subcomplex of all cells of $\mathcal{C}$ of dimension $i = 0..k$.

3.2. Hyperplane arrangements

A *hyperplane arrangement* is a finite set of hyperplanes $\mathcal{H} = \{H_1, \dots, H_m\}$ in $\mathbb{R}^D$. It induces a polyhedral decomposition $\mathcal{C}(\mathcal{H})$ of $\mathbb{R}^D$. A D-dimensional cell in $\mathcal{C}(\mathcal{H})$ or a *region* is the closure of a maximal connected region of $\mathbb{R}^D$ not intersected by any hyperplane in $\mathcal{H}$. For $k = 0..D-1$ the k-dimensional cells in $\mathcal{C}(\mathcal{H})$ are defined inductively as the facets of the $(k+1)$-dimensional cells. A hyperplane arrangement is *generic* if no more than D hyperplanes intersect at any single point.

3.3. Sign-vectors

Given a hyperplane arrangement $\mathcal{H}$, any point $\mathbf{x} \in \mathbb{R}^D$ is assigned a *sign-vector* $\boldsymbol{\sigma}(\mathbf{x}) = (\sigma_i(\mathbf{x}))_{i=1..m}$, with

$$\sigma_i(\mathbf{x}) = \begin{cases} + & \text{if } \mathbf{x} \in H_i^+, \\ 0 & \text{if } \mathbf{x} \in H_i, \\ - & \text{if } \mathbf{x} \in H_i^-. \end{cases} \tag{1}$$

Similarly, every cell C of $\mathcal{C}(\mathcal{H})$ can be associated with a sign-vector $\boldsymbol{\sigma}(C)$ such that

$$C = \bigcap_{i=1}^{m} H_i^{\sigma_i(C)} =: \mathcal{H}^{\boldsymbol{\sigma}(C)} \tag{2}$$

with $H^0 := H$ (Matousek, 2002).

3.4. ReLU networks and folded hyperplane arrangements

For our purposes a fully-connected feed-forward NN f_Θ maps any point $\mathbf{x}$ in the *domain* $\mathcal{D} \subset \mathbb{R}^D$ to a $D^{(L)}$-dimensional output $f_\Theta(\mathbf{x}) \in \mathbb{R}^{D^{(L)}}$. The NN is a composition of L layers with parameters $\Theta = \left\{\Theta^{(l)}\right\}_{l=1..L}$:

$$f_\Theta(\mathbf{x}) = (f^{(L)}_{\Theta^{(L)}} \circ \dots \circ f^{(1)}_{\Theta^{(1)}})(\mathbf{x})\,. \tag{3}$$

Starting at $\mathbf{x}^{(0)} = \mathbf{x}$, the layers are applied successively for $l = 1..L$ as

$$\mathbf{x}^{(l)} = f^{(l)}_{\Theta^{(l)}}(\mathbf{x}^{(l-1)}) = \operatorname{ReLU}(\mathbf{W}^{(l)}\mathbf{x}^{(l-1)} + \mathbf{b}^{(l)}). \tag{4}$$

The layer parameters $\Theta^{(l)} = \left\{\mathbf{W}^{(l)}, \mathbf{b}^{(l)}\right\}$ contain the *weights* $\mathbf{W}^{(l)} \in \mathbb{R}^{D^{(l-1)} \times D^{(l)}}$ and *biases* $\mathbf{b}^{(l)} \in \mathbb{R}^{D^{(l)}}$. For simplicity, we adhere to the main line of work focusing on the use of $\operatorname{ReLU}(x) = \max(0, x)$ as the activation, but the key ideas generalize to any CPWA NN.

In analogy to a hyperplane which is the zero-level set of an affine map, a *folded hyperplane* is the zero-level set of the pre-activation of a neuron. The i-th neuron in the l-th layer induces the folded hyperplane $H_i^{(l)} := \left\{\mathbf{x} \in \mathbb{R}^D | \mathbf{W}_i^{(l)\top}\mathbf{x}^{(l-1)} + b_i^{(l)} = 0\right\}$. Similarly, a finite set of folded hyperplanes $\mathcal{H}$ is a *folded hyperplane arrangement* and induces a polyhedral decomposition of the domain $\mathbb{R}^D$ (Hein et al., 2019; Grigsby & Lindsey, 2022). On each region, the folded hyperplane acts like an affine hyperplane and does not fold. The sign-vector is defined analogously and can be evaluated from the neuron pre-activation: $\sigma_i^{(l)}(\mathbf{x}) = \operatorname{sgn}(\mathbf{W}_i^{(l)\top}\mathbf{x}^{(l-1)} + b_i^{(l)})$.

Theorem 3 in Grigsby & Lindsey (2022) states that almost every ReLU network is generic, which we assume throughout this work.

3.5. Polyhedral combinatorics

A partially ordered set or *poset* is the pair $(\mathcal{S}, \leq)$ of the set $\mathcal{S}$ together with a binary relation $\leq$ on $\mathcal{S}$ (called an *ordering*) satisfying three axioms: reflexivity ($x \leq x$ for all x), transitivity ($x \leq y$ and $y \leq z$ implies $x \leq z$), and weak anti-symmetry (if $x \leq y$ and $y \leq x$, then $x = y$). For any two elements $x, y \in \mathcal{S}$ the *meet* $x \wedge y$ is the greatest lower bound of x and y. Similarly, the *join* $x \vee y$ is the least upper bound of x and y. Neither need exist, but if they do then they are unique (Kishimoto & Levi, 2019).

We will introduce some terminology from graph theory to denote relationships in the poset. $y \in \mathcal{S}$ is an *ascendant* of $x \in \mathcal{S}$ if $x < y$. Conversely, x is a *descendant* of y. The closest common ascendant of both x, y is the join $x \vee y$. The closest common descendant of both x, y is the meet $x \wedge y$. $y \in \mathcal{S}$ is a *parent* of $x \in \mathcal{S}$ if $x < y$ and no $z \in \mathcal{S}$ satisfies $x < z < y$. Conversely, x is a *child* of y.
The *intersection-poset* of the polyhedral complex $\mathcal{C}$ is the poset $(\mathcal{C}, \subseteq)$ of its cells ordered by inclusion.

4. Method

Our method is motivated by the observation illustrated in Figure 4. Due to the continuity of the activation function, all folded hyperplanes are continuous across each other. However, the existing subdivision methods consider each

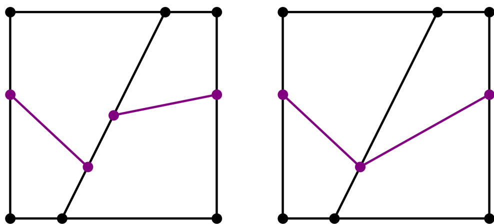

Figure 4. Motivation in $D = 2$: due the continuity of the activation, the two new edges share the same vertex on the common edge of the two regions. Processing each region individually is redundant.

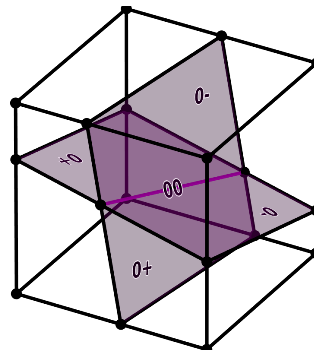

Figure 5. The parenting $(k+1)$-faces of a k-face can be obtained by perturbing each zero in its sign-vector at a time. Here, $k = 1$ and the first $m = 6$ entries of the sign vector are hidden for visual clarity since they are all $+$. This edge and all its ascendants are interior cells.

region independently. As a consequence, upon the intersection with a new folded hyperplane, each new vertex is computed independently 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.
Our method alleviates this redundancy by taking into account the continuity and disregarding the regions, instead using only the unique vertices and edges, i.e. the 1-skeleton. Edge subdivision preserves the iterative structure of considering each neuron in each layer sequentially, for each neuron subdividing the edges in five steps:

(1) Evaluate the NN at the vertices;

(2) Get the sign-vectors of the vertices;

(3) Find splitting edges by comparing the signs of vertex pairs;

(4) For each splitting edge, compute the new vertex using interpolation and split the edge;

(5) Build the intersecting edges (connecting new vertices across splitting faces).

This process is illustrated in Figure 3 and the steps are detailed in the following. We start by discussing how to recover the combinatorial structure of the complex from the sign-vectors.

4.1. Perturbation using sign-vectors

All the combinatorial relationships described in Section 3.5 can be easily evaluated using sign-vectors. However, instead of just determining the relationship of two given cells, edge subdivision and optional post-processing steps rely on building parent cells.

For now, assume the unbounded domain $\mathbb{R}^D$. The number of zeros in the sign-vector of a k-cell is $(D-k)$. To construct all the parents of this cell, take one of the zeros at a time and set it to $+$ or $-$. Consequentially, a k-cell has $2(D-k)$ parent cells for $k = 0..D-1$. We call this process *perturbation* and illustrate it in Figure 5.
Repeating this for all k-cells in the complex constructs all $(k+1)$-cells, including the ordering relations. Hence, the $(k+1)$-skeleton can be built from the k-skeleton. Starting from the 1-skeleton, perturbation can be applied sequentially to reconstruct the whole intersection-poset, including the regions.

Instead of the unbounded $\mathbb{R}^D$, we will operate on a bounded polyhedral domain $\mathcal{D}$, which is given by the intersection of m affine halfspaces. The motivation for this is a simpler implementation since bounded edges have exactly two vertices allowing to use simpler data-structures. In this setting, special care must be taken with *boundary cells*. These are the cells for which any of the first m entries of the sign-vector is 0 (conversely, the first m entries of an interior cell are $+$). Since we are not interested in cells outside the domain, we perturb the zeros on the boundary only toward the interior $+$. So under the consideration of the bounded domain, a k-cell has $z + 2(D-k-z)$ parent cells, where z is the number of zeros in the first m signs.

4.2. Edge subdivision

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 $C^0 = C \cap H$ is a new $(k-1)$-cell. Additionally, H splits C into two new k-cells $C^+ = C \cap H^+$ and $C^- = C \cap H^-$. 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.

4.2.1. STEPS 0-4

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, which we label V^+, V^- (step 3). The new vertex $V^0 = E \cap H$ on the splitting edge E can be computed by linearly interpolating the positions of V^+, V^- weighted by their pre-activation values. This new vertex takes the sign 0 w.r.t. H. The old splitting edge E is removed from $\mathcal{E}$ and the two new *split* edges $E^+ = E \cap H^+$ with vertices V^+, V^0 and $E^- = E \cap H^-$ with vertices V^-, V^0 are added to $\mathcal{E}$. The new signs of E^+, V^+ and E^-, V^- w.r.t. H are trivially $+$ and $-$, respectively (step 4).

4.2.2. STEP 5

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 $E^0 = F \cap H$ an *intersecting* edge.

In a naive approach, it would seem that we need to track the 2-faces in order to intersect them with H. However, by induction, this would require to track the whole complex. This is impractical due to the amount and layout of the memory – for $k > 1$, a k-cell can have an arbitrary number of facets, as opposed to exactly 2 vertices for each edge in a bounded domain. Instead, we propose to intersect the 2-faces *implicitly*.

This is enabled by another observation – every bounded splitting 2-face has exactly two splitting edges. Furthermore, the intersecting edge connects the two new vertices on those two splitting edges. Since we have already determined the splitting edges, we use them to implicitly identify splitting 2-faces and append the intersecting edges to $\mathcal{E}$.

Given two splitting edges we can perform a simple adjacency check using their sign vectors. However, checking all possible pairs has a quadratic memory complexity $\mathcal{O}(|\hat{\mathcal{E}}|^2)$ in the number of splitting edges $|\hat{\mathcal{E}}|$. This is infeasible even for moderately sized NN (see Section 5.1.2).

Instead, we propose a much more efficient method. For each splitting edge build its parenting 2-faces using perturbation as described in Section 4.1. We have a list of splitting 2-faces each pointing to a single splitting edge. In this list, each 2-face comes up exactly twice. We pair the two edges associated with the same 2-face. With at most $2(D-1)$ 2-faces per edge, the memory requirement is only $\mathcal{O}(2(D-1)|\hat{\mathcal{E}}|)$. Lastly, it remains to add the intersecting edge to $\mathcal{E}$. Its sign-vector is inherited from the splitting face with a 0 appended w.r.t. H.

This concludes a single iteration of edge subdivision, which is repeated for every neuron in every layer.

4.2.3. COMPLEXITY ANALYSIS

In Appendix A, we elaborate that all the steps (1-5) of edge subdivision can be implemented in linear time and memory complexity in the number of vertices $|\mathcal{V}|$, edges $|\mathcal{E}|$, or splitting edges $|\hat{\mathcal{E}}|$. We argue further that $O(|\mathcal{E}|)$ and $O(|\hat{\mathcal{E}}|)$ can be replaced with $O(|\mathcal{V}|)$, concluding that the total algorithm is $O(|\mathcal{V}|)$, and hence optimal.

The number of regions in a randomly initialized or trained NN is known to be $o(N^D/D!)$ where N is the total number of neurons (Hanin & Rolnick, 2019b). Conservatively assuming a proportional number of vertices $|\mathcal{V}|$ (see Figure 8), we can obtain the complexity of the algorithm with respect to the NN architecture.

5. Experiments

We start by describing, validating, and timing our implementation of edge subdivision. As described in Sections 1 and 2, the access to the exact polyhedral complex is intriguing in many theoretical and practical applications. Instead, we consider how the speed and differentiability of our method enable a novel experiment in which an optimization objective is formulated on the geometric properties of the extracted complex. Lastly, we discuss an approach to pruning NN parameters and test a method to modify edge subdivision if the goal is to extract just an iso-level-set, i.e. decision boundary.

5.1. 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, using standard operations step 5 can only be implemented in sub-optimal log-linear time using sorting to pair up identical rows of a tensor. This step can be implemented in linear time using hash-tables, but since efficient hashing on the GPU with custom length keys is non-trivial (Jünger et al., 2020; Awad et al., 2023), we hope to address this in future work.

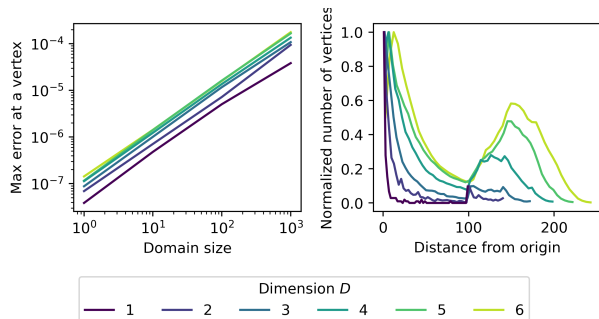

Figure 6. Left: Maximum error over all vertices is at least seven orders of magnitude smaller than the size of the hypercube domain serving as a validation. Right: Distribution of the vertex distances from the origin, normalized by the maximum value. The two distinct modes correspond to the interior and boundary vertices.

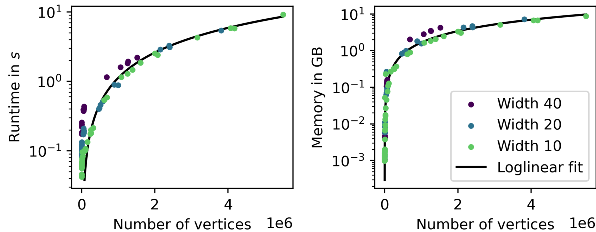

Figure 7. Our implementation shows log-linear scaling w.r.t. the number of vertices. This is due to a sub-optimal implementation of step 5 using sorting. Leveraging hash tables would improve the whole implementation to linear complexity.

5.1.1. VALIDATION

We first test a basic necessary (although insufficient) condition for the validity of our implementation. For each found vertex, the neurons corresponding to 0 entries in its sign-vector should be 0 at the vertex location. This is indeed satisfied up to a numerical error. Figure 6 (left) shows that the maximum numerical error is roughly seven orders of magnitude smaller than the extent of the domain and even less in lower dimensions. A fixed NN with 4 layer depth and 10 neuron width is used throughout. This error should not be confused with the geometric error in the position of the vertex, which is closely related, but not quantified here.

5.1.2. PERFORMANCE

We investigate the time and memory behaviour of our implementation by counting vertices and edges on the bounded unit hypercube domain. We consider NNs of four layers and widths of $10, 20, 40$ for input dimensions $D = 1..10$. The sizes of the experiments are mainly limited by the memory. The tests are performed on an NVIDIA RTX 3090. Since no data is on the CPU, the maximum allocated memory is measured for just the GPU.

The results are illustrated in Figure 8. Even moderately sized NNs induce complexes with millions of cells, especially as the input dimension increases. It is known that the number of regions scales exponentially with the number of input dimensions. For the number of vertices and edges, we observe a subexponential growth.

Counting the regions is possible as described in Section 4.1, but requires an additional significant amount of memory and time, since using perturbation requires to store and group $2(D-1)|\mathcal{E}|$ cells which is up to $\mathcal{O}(10^8)$ for some of the considered cases.

In Figure 7 we validate our complexity analysis. We reuse the previous results to plot the runtime and memory over the number of vertices, which as expected is log-linear due to the sub-optimal use of sorting.

Lastly, in Figure 9 we compare to SplineCam (Humayun et al., 2023) which is a region subdivision method specifically for $D = 2$. Over the considered tests, our method is on average 20 times faster, since SplineCam uses graph structures on the CPU.

5.2. Vertex distribution

We study the effect of the domain being bounded. We perform edge subdivision on a $\mathcal{D} = [-100, 100]^D$ hypercube domain. For each considered dimension, this is repeated on five randomly initialized NNs of 4 layer depth and 10 neuron width.

For every vertex, we compute its distance from the origin $r = ||\mathbf{x}||_2$. Figure 6 (right) shows a bi-modal distribution of r. For $r < 100$, we observe an exponentially decaying density of vertices. These are the interior vertices due to the folded hyperplane arrangement. Additionally, there are the domain boundary vertices, which intersect at least one of the hyperplanes defining the domain. These are located at $r \geq 100$, which corresponds to the second mode of the distribution. For a trained NN, we would generally expect a different distribution in the first mode, for example, the vertices to concentrate more tightly around the training data.

This illustrates a limitation of performing edge subdivision on a bounded domain. If we do not care about the cells on the artificially inserted boundary, then having a large proportion of boundary cells is undesirable for the performance. One simple solution is to replace the hypercube domain with a simplex domain, whose number of vertices (edges) grows linearly (quadratically) with D as opposed to exponentially. This also motivates a future work for extending edge subdivision to unbounded domains.

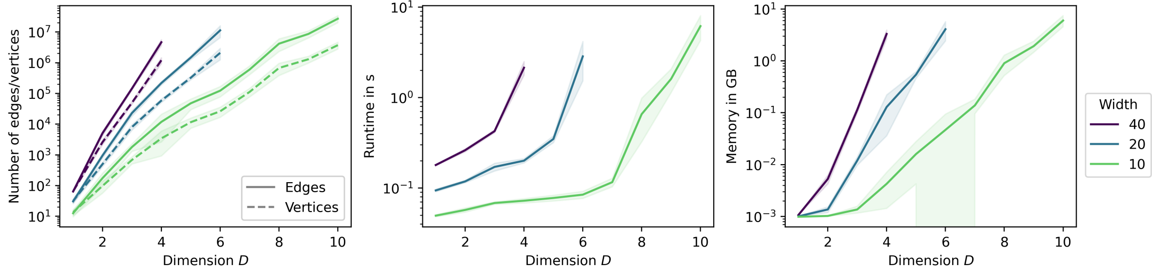

Figure 8. Edge and vertex counts, runtime, and memory usage of randomly initialized NNs of four layers and different widths and input dimensions. Mean ± standard deviation over 5 runs.

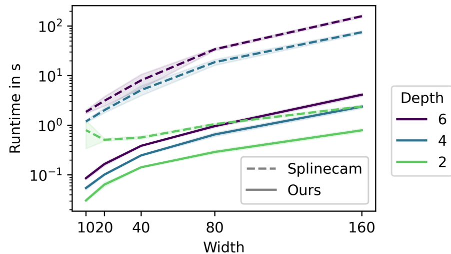

Figure 9. We achieve a 20 times speed-up on average over SplineCam (Humayun et al., 2023) which is also limited to $D = 2$.

5.3. Geometric loss

We utilize the differentiability and speed of edge subdivision to consider a novel experiment in which an optimization objective is formulated on geometric properties of the extracted complex. In Figure 10, we start with a ReLU NN with two hidden layers of 50 neurons each and $D = 2$. The NN is first 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. The normalization is such that $c = 1$ for the most compact shape – the circle. Using c as the loss, the bunny shape converges to a circle in 100 iterations with a standard Adam optimizer.

In general, any geometric loss that depends on the vertex positions can be formulated and optimized, e.g. edge lengths, angles, areas, volumes, curvatures, and other quantities from discrete differential geometry. This holds for both the boundary and the volumetric shape, as well as the polyhedral complex (i.e. the mesh) itself.

5.4. Pruning

Finally, we consider two approaches to pruning, focusing on a geometric context due to the intuitive interpretation. Consider an implicit neural representation of a bounded geometry. Important in this view is the boundary of the shape, similar to the decision boundary in a classification task. We can view the ReLU NN as a compact storage format and many geometric properties of a shape can be computed from just its boundary.

5.4.1. Parameter pruning

In the first view, we refer to *pruning* as a NN compression technique in which some parameters are removed after training with a negligible drop in the NN performance (Lee et al., 2018). In the context of preserving the shape or decision boundary, all the folded hyperplanes which do not intersect the boundary can be removed. This corresponds to pruning the respective neurons. For ReLU, the boundary can be completely contained in either on the negative or positive half-space of a non-intersecting neuron. The negative neurons can be removed completely as they do not contribute any value anywhere on the shape. The folded hyperplanes of such neurons are highlighted in red in Figure 11. Since the training data is localized on the unit square, the folded hyperplanes not intersecting this domain correspond to *dying-ReLUs* – neurons which for all training samples are in the rectifying 0 region of ReLU. However, there are also folded hyperplanes intersecting the domain but not the shape itself. Removing all of these allows to compress the $2, 50, 50, 1$ NN down to $2, 25, 19, 1$, reducing the number of parameters from 2751 to 589.

This can be pruned further by also considering the converse case – neurons for which the whole boundary is in the linear activation of ReLU. Since each such neuron contributes the same affine function everywhere on the shape, any linearly dependent (in general any $> D$ neurons of the same layer) can be compressed down to D while adjusting the outgoing weights accordingly.

5.4.2. Pruning during edge subdivision

Extracting the whole complex and selecting just the boundary is wasteful, even if the NN is compressed as described

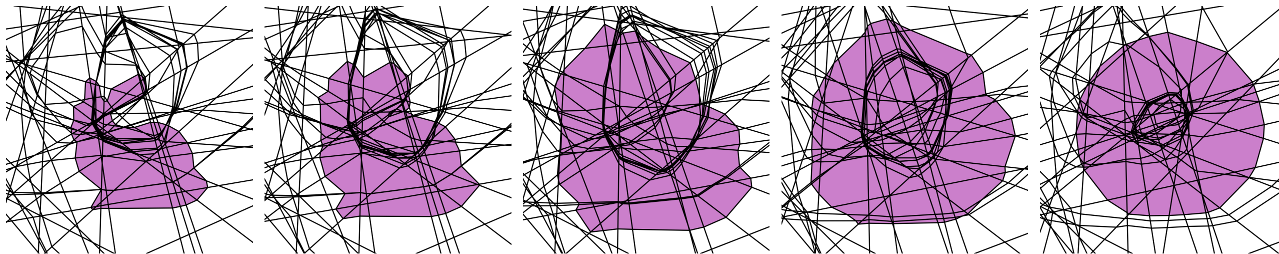

Figure 10. The extracted skeleton at initial, intermediate, and final iteration of optimizing the shape compactness. The bunny shape converges to a circle in 100 iterations of a standard Adam optimizer with the loss formulated on the extracted complex.

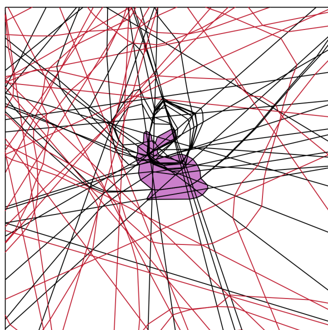

Figure 11. Parameter pruning. Highlighted in red are folded hyperplanes for which the purple shape is contained fully in the hyperplane's negative halfspace. Since after ReLU such a neuron is 0 everywhere, it can simply be removed. A similar pruning approach can be taken to non-intersecting positive neurons.

above. We propose a complementary pruning strategy specifically for during edge subdivision. It intends to prune all edges and vertices, for whom we can say with confidence that they will not contribute toward the boundary.

Recall, that a new vertex is created only where an edge splits and such splitting edges are detected by the signs of their vertex pairs disagreeing. We can compute the sign-vector of all current vertices even at any intermediate iteration of subdivision. An edge can be pruned if both its vertices have the same signs w.r.t. all future neurons. For the considered 2D bunny geometry this reduces the number of edges from $757/2424/2576$ to $301/76/228$ after finishing each layer ($399/1240/1316$ to $305/118/194$ for vertices). While the described condition is sufficient for pruning, it may perhaps be improved further, providing an alternative to Analytical Marching (Lei & Jia, 2020).

6. Conclusions

In this work, we observed a redundancy in region subdivision and proposed a novel edge subdivision method for

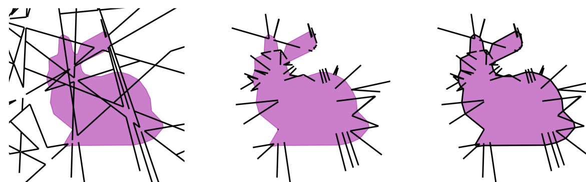

Figure 12. Pruning during edge subdivision reduces the number of vertices and edges in each iteration by looking at future sign-vectors. Displayed are the preserved intermediate edges after each layer.

extracting the exact polyhedral complex from ReLU NNs. Our approach allowed to use simple data structures and tensor operations to leverage the GPU improving the performance over previous methods over 20 times. The speed and differentiability allowed us to propose novel applications in which a loss can be formulated on the extracted complex. While we hope this opens interesting avenues in geometry, in higher dimensions the method is limited by the exponential growth of the complex. Further limitations and future research directions include extending the method to unbounded domains, non-generic arrangements, and other CPWA architectures, as well as improving the pruning strategies for more efficient extraction of level-sets. However, the main outlook for improved performance is replacing sorting with hash-tables, improving the whole implementation to linear time and memory in the number of vertices.

Acknowledgements

This was supported by the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement number 860843.

References

Awad, M. A., Ashkiani, S., Porumbescu, S. D., Farach-Colton, M., and Owens, J. D. Analyzing and implementing GPU hash tables. In *SIAM Symposium on Algo-*

rithmic Principles of Computer Systems, APOCS23, pp. 33–50, January 2023. doi: 10.1137/1.9781611977578.ch3. URL https://escholarship.org/uc/item/6cb1q6rz.

Balestriero, R. and Baraniuk, R. Batch normalization explained. *ArXiv*, abs/2209.14778, 2022.

Balestriero, R. and Baraniuk, R. G. Mad max: Affine spline insights into deep learning. *Proceedings of the IEEE*, 109(5):704–727, May 2021. doi: 10.1109/jproc.2020.3042100.

Balestriero, R. and LeCun, Y. Police: Provably optimal linear constraint enforcement for deep neural networks, 2022. URL https://arxiv.org/abs/2211.01340.

Balestriero, R., Cosentino, R., Aazhang, B., and Baraniuk, R. G. The geometry of deep networks: Power diagram subdivision. *Advances in Neural Information Processing Systems*, 32(NeurIPS), 2019. ISSN 10495258.

Daróczy, B. Gaussian perturbations in relu networks and the arrangement of activation regions. *Mathematics*, 10 (7), 2022. ISSN 2227-7390. doi: 10.3390/math10071123. URL https://www.mdpi.com/2227-7390/10/7/1123.

Grigsby, J. E. and Lindsey, K. On transversality of bent hyperplane arrangements and the topological expressiveness of relu neural networks. *SIAM Journal on Applied Algebra and Geometry*, 6(2):216–242, 2022. doi: 10.1137/20M1368902. URL https://doi.org/10.1137/20M1368902.

Grunbaum, B. and Ziegler, G. M. *Convex Polytopes*. Graduate Texts in Mathematics. Springer, New York, NY, 2 edition, may 2003.

Grunert, R. *Piecewise Linear Morse Theory*. PhD thesis, Freie Universit¨at Berlin, 2016. URL https://refubium.fu-berlin.de/handle/fub188/12531.

Hanin, B. and Rolnick, D. Complexity of linear regions in deep networks. *ArXiv*, abs/1901.09021, 2019a.

Hanin, B. and Rolnick, D. Deep relu networks have surprisingly few activation patterns. In Wallach, H. M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E. B., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pp. 359–368, 2019b. URL https://proceedings.neurips.cc/paper/2019/hash/9766527f2b5d3e95d4a733fcfb77bd7e-Abstract.html.

Hein, M., Andriushchenko, M., and Bitterwolf, J. Why ReLU networks yield high-confidence predictions far away from the training data and how to mitigate the problem. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, June 2019. doi: 10.1109/cvpr.2019.00013. URL https://doi.org/10.1109/cvpr.2019.00013.

Humayun, A. I., Balestriero, R., and Baraniuk, R. Exact visualization of deep neural network geometry and decision boundary. In *NeurIPS 2022 Workshop on Symmetry and Geometry in Neural Representations*, 2022. URL https://openreview.net/forum?id=VSLbmsoZxai.

Humayun, A. I., Balestriero, R., Balakrishnan, G., and Baraniuk, R. Splinecam: Exact visualization and characterization of deep network geometry and decision boundaries. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2023.

Jordan, M., Lewis, J., and Dimakis, A. G. Provable certificates for adversarial examples: Fitting a ball in the union of polytopes. *Advances in Neural Information Processing Systems*, 32(NeurIPS), 2019. ISSN 10495258.

Jünger, D., Kobus, R., Müller, A., Hundt, C., Xu, K., Liu, W., and Schmidt, B. Warpcore: A library for fast hash tables on gpus. In *2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pp. 11–20, 2020. doi: 10.1109/HiPC50609.2020.00015.

Kishimoto, D. and Levi, R. Polyhedral products over finite posets. *Kyoto Journal of Mathematics*, 2019.

Lee, N., Ajanthan, T., and Torr, P. H. Snip: Single-shot network pruning based on connection sensitivity. *arXiv preprint arXiv:1810.02340*, 2018.

Lei, J. and Jia, K. Analytic Marching: An analytic meshing solution from deep implicit surface Networks. *37th International Conference on Machine Learning, ICML 2020*, PartF16814:5745–5754, 2020.

Lei, J., Jia, K., and Ma, Y. Learning and meshing from deep implicit surface networks using an efficient implementation of analytic marching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021. doi: 10.1109/tpami.2021.3135007. URL https://doi.org/10.1109/tpami.2021.3135007.

Matousek, J. *Lectures on Discrete Geometry*. Graduate texts in mathematics. Springer, New York, NY, May 2002.

Montúfar, G., Pascanu, R., Cho, K., and Bengio, Y. On the number of linear regions of deep neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, pp. 2924–2932, Cambridge, MA, USA, 2014. MIT Press.

Pascanu, R., Montufar, G., and Bengio, Y. On the number of response regions of deep feed forward networks with piece-wise linear activations, 2013.

Raghu, M., Poole, B., Kleinberg, J., Ganguli, S., and Sohl-Dickstein, J. On the expressive power of deep neural networks. In Precup, D. and Teh, Y. W. (eds.), *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 2847–2854. PMLR, 06–11 Aug 2017.

Rolnick, D. and Kording, K. Reverse-engineering deep ReLU networks. In III, H. D. and Singh, A. (eds.), *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 8178–8187. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/rolnick20a.html.

Sattelberg, B., Cavalieri, R., Kirby, M., Peterson, C., and Beveridge, R. Locally Linear Attributes of ReLU Neural Networks, 2020. URL http://arxiv.org/abs/2012.01940.

Serra, T., Tjandraatmadja, C., and Ramalingam, S. Bounding and counting linear regions of deep neural networks. In *ICML*, 2018.

Telgarsky, M. Representation benefits of deep feedforward networks. *ArXiv*, abs/1509.08101, 2015.

Telgarsky, M. Benefits of depth in neural networks. In Feldman, V., Rakhlin, A., and Shamir, O. (eds.), *29th Annual Conference on Learning Theory*, volume 49 of *Proceedings of Machine Learning Research*, pp. 1517–1539, Columbia University, New York, New York, USA, 23–26 Jun 2016. PMLR.

Wang, Y. Estimation and comparison of linear regions for relu networks. In Raedt, L. D. (ed.), *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, pp. 3544–3550. International Joint Conferences on Artificial Intelligence Organization, 7 2022. doi: 10.24963/ijcai.2022/492. URL https://doi.org/10.24963/ijcai.2022/492. Main Track.

A. Complexity analysis

The algorithm operates on the vertices and (splitting) edges, so it is natural to first consider the complexity w.r.t. these inputs. Let $|\mathcal{V}|$, $|\mathcal{E}|$ and $|\hat{\mathcal{E}}|$ denote the number of vertices, edges and splitting edges at iteration i. The complexities of the steps are:

(1) $O(|\mathcal{V}|)$ to evaluate the NN at $|\mathcal{V}|$ vertices (requires matrix multiplications and non-linearities). However, complexity of each evaluation also depends on the size of the NN, but let us assume this is much less than $|\mathcal{V}|$.

(2) $O(|\mathcal{V}|)$ to get signs of $|\mathcal{V}|$ values

(3) $O(|\mathcal{E}|)$ to compare two signs per edge to identify splitting edges

(4) $|\hat{\mathcal{E}}|$ to linearly interpolate $|\hat{\mathcal{E}}|$ vertex pairs of the $|\hat{\mathcal{E}}|$ splitting edges

(5) $O((D-1)|\hat{\mathcal{E}}|)$ to build the intersecting edges. As described, there are at most $2(D-1)|\hat{\mathcal{E}}|$ 2-faces after perturbation. Each 2-face is associated with exactly 2 splitting edges, which we need to pair up. Using hash tables, this can be performed in linear time.

It is non-trivial to relate $|\mathcal{E}|$, $|\mathcal{V}|$ and $|\hat{\mathcal{E}}|$, but we will assume that there is a linear relationship. We see this empirically for $|\mathcal{E}|$ and $|\mathcal{V}|$ in Figure 8. The number of splitting edges can be upper bounded by $|\hat{\mathcal{E}}| < |\mathcal{E}|D/i$ using Theorem 5 in (Hanin & Rolnick, 2019b) where D is the input dimension and $i >> D$ is the iteration (i.e., number of neurons considered already). This upper bound agrees with some empirical tests. Hence, we replace $O(|\mathcal{E}|)$ and $|\hat{\mathcal{E}}|$ with $O(|\mathcal{V}|)$. Since all the steps in the algorithm are linear, the algorithm in total is linear $O(|\mathcal{V}|)$.