SplineCam: Exact Visualization of Deep Neural Network Geometry and Decision Boundary

Supplementary Materials

Codes available at SplineCAM Github
Google Colab demo https://bit.ly/splinecam-demo

We provide the following supplementary materials (SMs) as support of our theoretical and empirical claims. This SM is organized as follows.

Appendix A provides the necessary background results for SplineCam and elaborates on how any deep neural network or implicit neural representation function with piecewise contiuous affine activations are max affine splines. Appendix B provides further implementation details, extending from Sec. 2.2 and including hardware and software requirements.

Appendix C elaborates on the computational complexity of the method. We present experiments assessing the time complexity of SplineCam varying the width of a single layer MLP and varying the volume of the input domain for VGG11. In Appendix D we present new experiments, first Appendix D.1 discusses the change of partition characteristics with training epochs and while varying architecture parameters (e.g., width for MLP and number of filters for a CNN). We also present the change of characteristics across different parts of the input space, e.g., around training samples, test samples and regions off the data manifold. In Appendix D.2 we present quantitative results on the variation of partition statistics while varying the orientation of the 2D input domain of interest. We see that the variation is considerably low between random orientation, showing that a single 2D slice can possibly be good enough to characterize the partition in different parts of the input space. Finally, in Appendix E we expand on the usage of SplineCam and present code blocks as explainers for how the SplineCam framework operates.

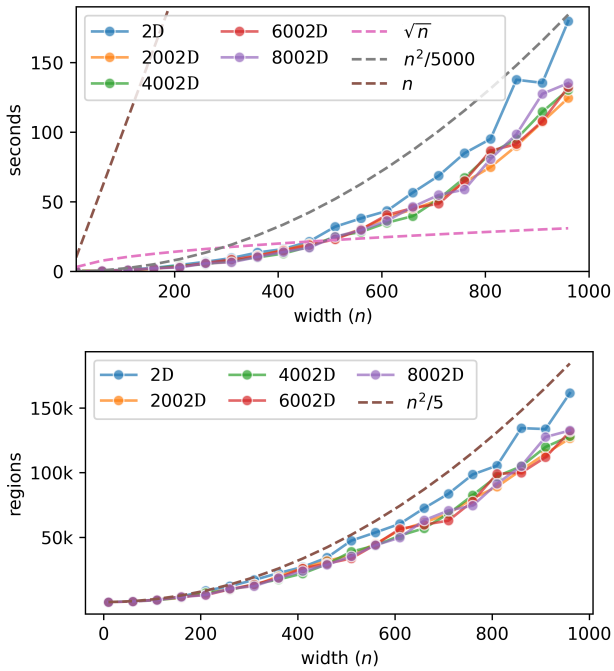

Figure 10. Time complexity of computing the partition regions **(Top)** and growth of the number of regions with width **(Bottom)**, for a single layer randomly initialized MLP with varying width (n) and input dimensionality. For all the input dimensions, we take a randomly oriented, square 2D domain centered on the origin, with an area of 4 units. Note that with increasing input dimensionality, we get reduced number of hyperplane intersections with our 2D domain of interest, therefore we can see a slight reduction in the wall time required and also the number of regions.

A. Background on Continuous Piecewise Affine Deep Networks

A *max-affine spline operator* (MASO) concatenates independent *max-affine spline* (MAS) functions, with each MAS formed from the point-wise maximum of R affine mappings [14, 24]. For our purpose each MASO will express a DN layer and is thus an operator producing a D^ℓ dimensional vector from a $D^{\ell-1}$ dimensional vector and is formally given by

$$\text{MASO}(\boldsymbol{v}; \{\boldsymbol{A}_r, \boldsymbol{b}_r\}_{r=1}^R) = \max_{r=1,\ldots,R} \boldsymbol{A}_r \boldsymbol{v} + \boldsymbol{b}_r, \qquad (7)$$

where $\boldsymbol{A}_r \in \mathbb{R}^{D^\ell \times D^{\ell-1}}$ are the slopes and $\boldsymbol{b}_r \in \mathbb{R}^{D^\ell}$ are the offset/bias parameters and the maximum is taken coordinate-wise. For example, a layer comprising a fully connected operator with weights $\boldsymbol{W}^\ell$ and biases $\boldsymbol{b}^\ell$ followed by a ReLU activation operator corresponds to a (single) MASO with $R = 2, \boldsymbol{A}_1 = \boldsymbol{W}^\ell, \boldsymbol{A}_2 = \boldsymbol{0}, \boldsymbol{b}_1 = \boldsymbol{b}^\ell, \boldsymbol{b}_2 = \boldsymbol{0}$. Note that a MASO is a *continuous piecewise-affine* (CPA) operator [38].

The key background result for this paper is that *the layers of DNs constructed from piecewise affine operators (e.g.,*

convolution, ReLU, and max-pooling) are MASOs [2]:

$$\exists R \in \mathbb{N}^*, \exists \{\boldsymbol{A}_r, \boldsymbol{b}_r\}_{r=1}^{R} \quad (8)$$

$$\text{s.t. } \text{MASO}(\boldsymbol{v}; \{\boldsymbol{A}_r, \boldsymbol{b}_r\}_{r=1}^{R}) = g^{\ell}(\boldsymbol{v}), \forall \boldsymbol{v} \in \mathbb{R}^{D^{\ell-1}}, \quad (9)$$

making any Implicit Neural Representation or even Deep Generative Networks a composition of MASOs.
The CPA spline interpretation enabled from a MASO formulation of DNs provides a powerful global geometric interpretation of the network mapping based on a partition of its input space $\mathbb{R}^S$ into polyhedral regions and a per-region affine transformation producing the network output. The partition regions are built up over the layers via a *subdivision* process and are closely related to Voronoi and power diagrams [4].

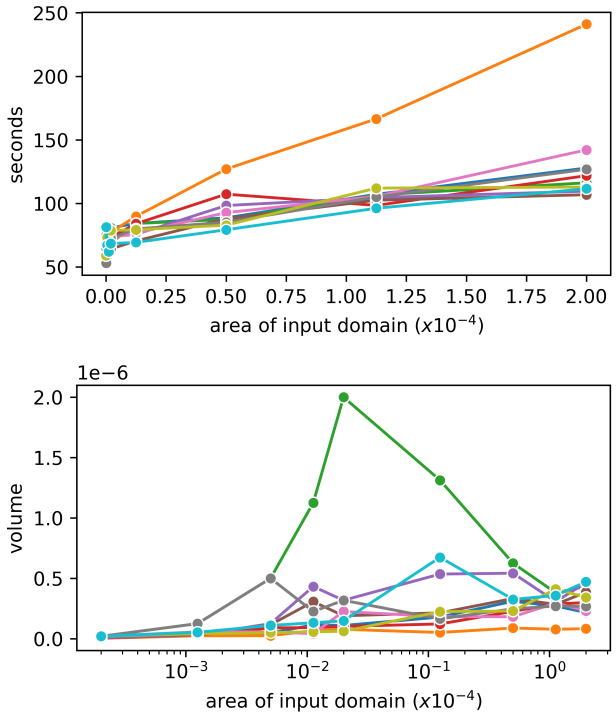

Figure 11. Time complexity of computing the partition regions **Top** and average region volume (ARV) **Bottom** while increasing the area of the input domain, for a VGG11 trained on tinyimagenet-200. Each line corresponds to one of 10 training samples anchored on which the 2D input domains are defined, with random orientations. We see that the required time scales linearly with the size of the input domain. Apart from that, we see that as the neighborhood size is increased for some samples, the ARV increases and then decreases again. This could indicate that while the ARV is small for smaller neighborhoods around training samples while farther away larger regions appear and cause the transient behavior in the curves.

B. Implementation Details

In Sec. 2.2 we provide a summary of how SplineCam works. In this section we provide details about SplineCam implementation and algorithm.

SplineCam is implemented using Pytorch [30] and Graph-tool [31]. All the linear algebra operations are performed using Pytorch and are scalable using GPUs. The region finding operation on the other hand is single threaded. This can be a bottleneck in cases, for example, with DNNs that have more than one layer. In this case, distributing regions across threads, as pairs of hyperplane-sets and a polygon, can result in significant speedups.

Since finding the regions involve solving systems of linear equations, most of the opearations in SplineCam require *double* precision. This can introduce significant memory bottlenecks, especially for large convolutional layers with multiple channels of input and output; for such layers the size of the corresponding Toeplitz matrix representation becomes significantly large with large number of input and output channels. For this reason, we always store such weight matrices as sparse matrices and query rows only when required. We also replace max pool layers with average pool layers for simplicity; in our experiments involving VGG16 and VGG11, we replace the maxpool after the first conv with an average pool and use strided convolutions for deeper layers. Unless specified, we always consider square 2D domains in the input. While characterizing we use the terms volume and area of the regions interchangeably.

In Suppl. E, we provide details on how SplineCam can be used, with modular codes showing how it computes the partition in a layerwise fashion. We also provide a pseudocode for the search algorithm we have proposed to find regions given a graph formed via polygon-hyperplane intersections. All the reported computation times were evaluated on a setup consisting of 8x NVIDIA QUADRO RTX 8000 48GB GPUs and 2x Intel Xeon Gold 5220R.

C. Computational Complexity

In this section we present two sets of experiments that we have performed to assess the computational complexity of SplineCam.

Varying width for a single layer MLP. As we have discussed earlier and have elaborated in Suppl. E, SplineCam performs region wise partition operations, which can be parallelized across sets of regions. To assess the per region performance of SplineCam, we do the following experiment. We take an uninitialized 1 layer ReLU-MLP with n neurons and D input dimensionality. The number of neurons is equal to the number of hyperplanes that would partition space. We vary $n \in \{10...980\}$ and also vary

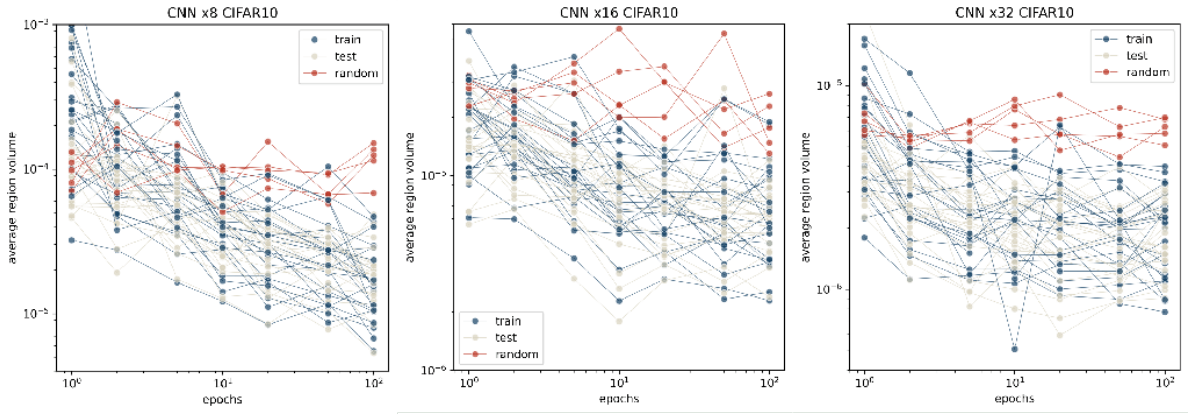

Figure 12. Evolution of average region volume (ARV) for 55 different randomly oriented square domains of the input space. Out of the 55 domains, 25 are centered on CIFAR10 training samples, 25 on test samples and 5 on random locations in the input space. We train a 8 layer CNN as detailed in Suppl. D.1. We notice that as training progresses, the ARV around points on the data manifold gets reduced. For points off the manifold ARV reduces as well, for CNN x8 CIFAR10 it reduces significantly while for CNN x32 it doesn't. In all cases, the off manifold ARV remains larger than the on manifold ARV at 100 training epochs.

$D \in \{2, 2002, 4002, 6002, 8002\}$ and plot in Fig. 10 the wall time in seconds, required to compute the partition. As the input domain, we consider a randomly oriented 2D domain centered on the origin with $4sq$ unit area. We see that the computation time complexity is upper bounded by $\mathcal{O}(n^2/5000)$ within the range of n in question. With increasing D we see a reduced number of hyperplane intersections, therefore we see a reduction in required time.

Effect of the area of the input domain. For this experiment, we take a VGG11 model and increase the size (area) of the input domain. We present the wall time vs area plot in Fig. 11. We can see that increasing the area (or volume) of the input region, monotonically increases the required time for computing the partition, in approximately a linear fashion. Even though increasing the area of the input domain should increase the number of first layer intersections, we see that the effect of that remains linear within the range. Note that, we can also scale the area of the input domain by breaking the input domain into multiple 2D polygonal regions of equal area and using separate threads/gpus to perform computation. This way we can also parallelize the partition computation and scale across memory instead of time.

D. Extra Experiments

D.1. Evolution of partition statistics while training

MLP trained on MNIST. For this experiment we train an MLP with depth 5 and width $\in \{8, 16, 32, 16, 128\}$. We train the MLPs for 50 training epochs on the MNIST dataset, and evaluate the partition statistics via SplineCam, for 25 training and 25 validation samples of randomly selected from MNIST. We present average region volume (ARV) distributions per training epoch in Fig. 14. The first thing to notice is that for smaller width networks, ARV is bimodal across all epochs, while for width 64 and onwards, the mode with higher volume regions vanishes. ARV also shifts towards lower volumes as the width of the networks are increased. While training samples tend to have lower ARV, the lower ends of the distributions differ between training and test samples; for widths 32, 64, and 128 we see distinct low ARV tails which are not visible for the test samples. This shows that for some of the training samples, the partition regions of the network are smaller, indicating that the model has more representation capacity for such sample neighborhoods [28]. This could be a possible indication for memorization of some training samples. Another thing to notice is that during the first epoch, the avg partition volumes are significantly lower. As training progresses, first ARV shifts to the right (larger) and then slowly shifts to the left. As we increase width, the starting ARV of the network becomes small in general compared to the ARV for the last training epoch. In Fig. 15 we also present the distributions of the number of regions (NR) in the neighborhood of the same samples used for Fig. 14.

CNN trained on CIFAR10. For this experiment, we train an 8 layer CNN with 6 convolutional layers and 2 fully connected layers. The number of filters for the convolutional layers are set as $\{\lfloor \ell/2 \rfloor \times mul \ : \ell = 1...6\}$, where $\lfloor . \rfloor$ is the floor operation, and $mul \in \{8, 16, 32\}$ is a width multiplier. We see that, similar to Fig. 14, the ARV gets reduced with increased width. For training, we can see longer tails towards lower ARV, indicating denser regions near some training samples. One thing that is noticeable here is that contrary to MLPs, the ARV of neighborhoods near training samples monotonically decrease in most cases

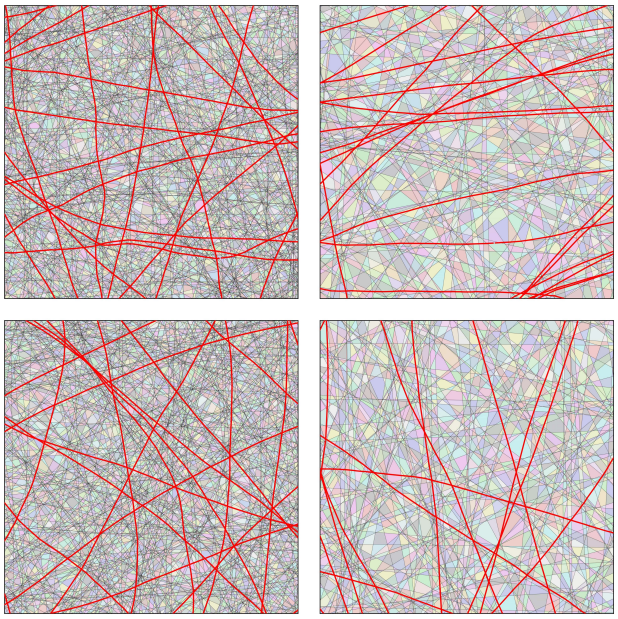

Figure 13. Partition visualization for a VGG11 model trained on TinyImagenet. We take 10 samples from the training set for which the model posteriors have the lowest entropy, and present here neighborhoods for two samples with high partition density **(Left)** and two samples with low partition density **(Right)**. We also highlight in red, the sets of points for which any two classes from the dataset has equal probability. We see that the denser partition regions have more such lines compared to sparser regions, suggesting correlations with generalization [35]

for the CNNs. This could be due to the complexity of the task, CIFAR10 classification being a harder task compared to MNIST, the region density is required to be significantly higher compared to early training.

We also visualize in Fig. 12 the evolution of ARV with training epochs for CNNs with different width multipliers. We visualize for 25 training, 25 test and 5 random samples, a randomly oriented 2D neighborhood. We notice that as training progresses, the ARV around points on the data manifold gets reduced. For points off the manifold ARV reduces as well, for CNN x8 CIFAR10 it reduces significantly while it doesn't as much for CNN x32. In all cases, the off manifold ARV remains larger than the on manifold ARV at 100 training epochs.

D.2. Variation of region statistics between random orientations

For this experiment we take 5 random training samples from TinyImagenet and calculate partition statistics for 20 randomly oriented 2D domains with area 0.005, centered on each sample. To assess the variability between different 2D domains we first look at the region volume (RV) statistic for the partition generated by a VGG11 model. Region volumes can vary both for a given 2D input domain and between different input domains. The maximum in-domain RV standard deviation over 20 different orientations is $\{7.3955e-07, 2.2665e-07, 3.0617e-06, 1.0149e-06, 2.2171e-06\}$ for each sample. The between orientation ARV standard deviation is $\{5.5993e-08, 7.4666e-09, 1.3462e{-}07, 3.9948e{-}08, 1.2935e{-}07\}$ for each sample, which is an order smaller than the in-domain RV standard deviation. This is an indication that SplineCam statistics for a single 2D slice could possibly be accurate enough to not require multiple 2D slices, even for high dimensional inputs.

D.3. Extra Figures

In the following section we present some figures complementing the experiments done above.

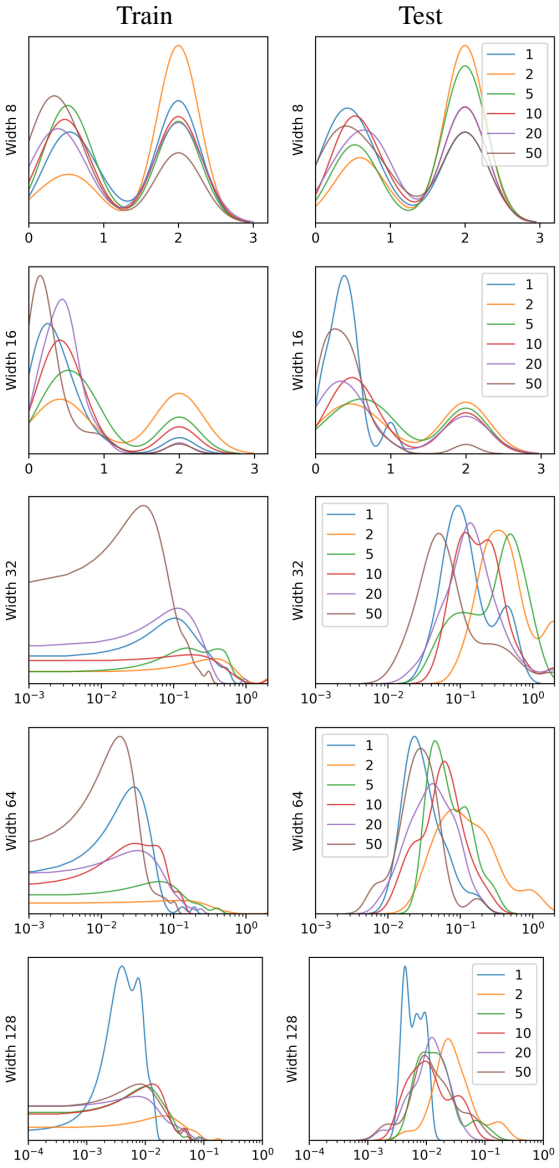

Figure 14. Distribution of *average region volume* (ARV) across training epochs, in the neighborhood of 25 train **(Left)** and 25 test **(Right)** samples from MNIST. We train an MLP with depth 5 and vary its width between $\{8, 16, 32, 16, 128\}$. ARV is considerably large and bimodal for smaller widths; as network width is increased ARV becomes smaller and unimodal, with long smaller volume tails for training samples. This discrepancy between training and test samples, possibly indicates memorization.

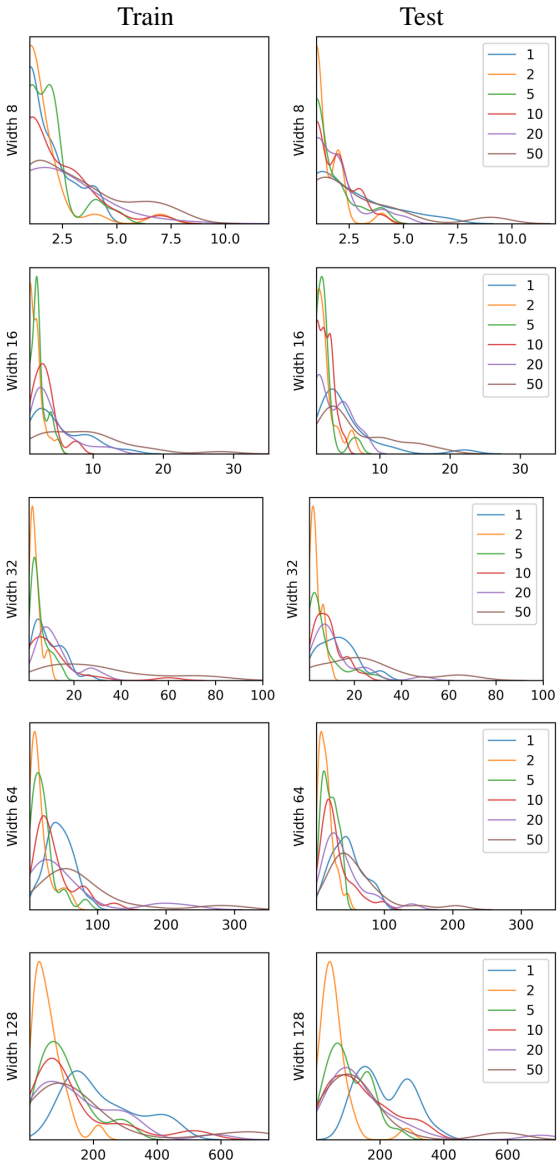

Figure 15. Distribution of *number of regions* (NR) across training epochs, in the neighborhood of 25 train **(Left)** and 25 test **(Right)** samples from MNIST. We train an MLP with depth 5 and vary its width between $\{8, 16, 32, 16, 128\}$. NR is small for smaller widths and increases significantly as the network width is increased. For larger networks, the distributions have large NR tails. We can also see a shift towards lower NR right after epoch 1 and a slow shift of the distribution towards larger NR as training progresses.

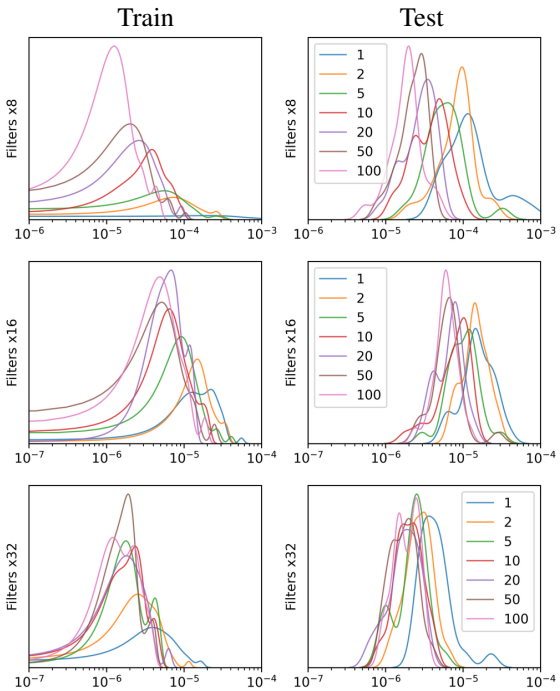

Figure 16. Distribution of *average region volume* (ARV) across training epochs, in the neighborhood of 25 train **(Left)** and 25 test **(Right)** samples from CIFAR10. We train a CNN with 6 convolutional layers and 2 fully connected layers. The number of filters for the layers are set as $\{\lfloor \ell/2 \rfloor \times mul \ : \ell = 1...6\}$, where $\lfloor . \rfloor$ is the floor operation, and $mul \in \{8, 16, 32\}$ is a width multiplier. We see that, similar to Fig. 14, the ARV gets reduced with increased width. For training, we can see longer tails towards lower ARV, indicating that for some training samples the regions become very small. For both train and test, with increasing number of epochs, the ARV distribution mean shifts towards lower ARV.

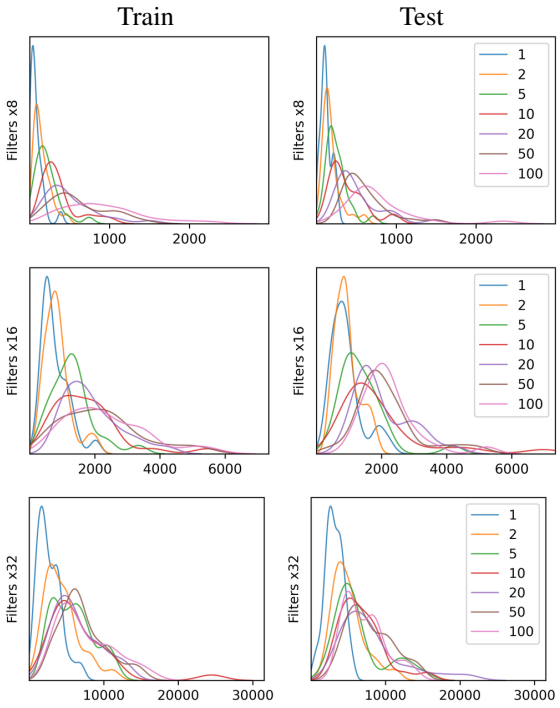

Figure 17. Distribution of *number of regions* (NR) across training epochs, in the neighborhood of 25 train **(Left)** and 25 test **(Right)** samples from CIFAR10. We train a CNN with 6 convolutional layers and 2 fully connected layers. The number of filters for the layers are set as $\{\lfloor \ell/2 \rfloor \times mul \ : \ell = 1...6\}$, where $\lfloor . \rfloor$ is the floor operation, and $mul \in \{8, 16, 32\}$ is a width multiplier. We see that, similar to Fig. 15, the NR significantly increases with increased width. For training, we can see longer tails towards higher NR, indicating that for some training samples there is high region density in the neighborhood. For both train and test, with increasing number of epochs, the NR distribution mean shifts towards higher NR.

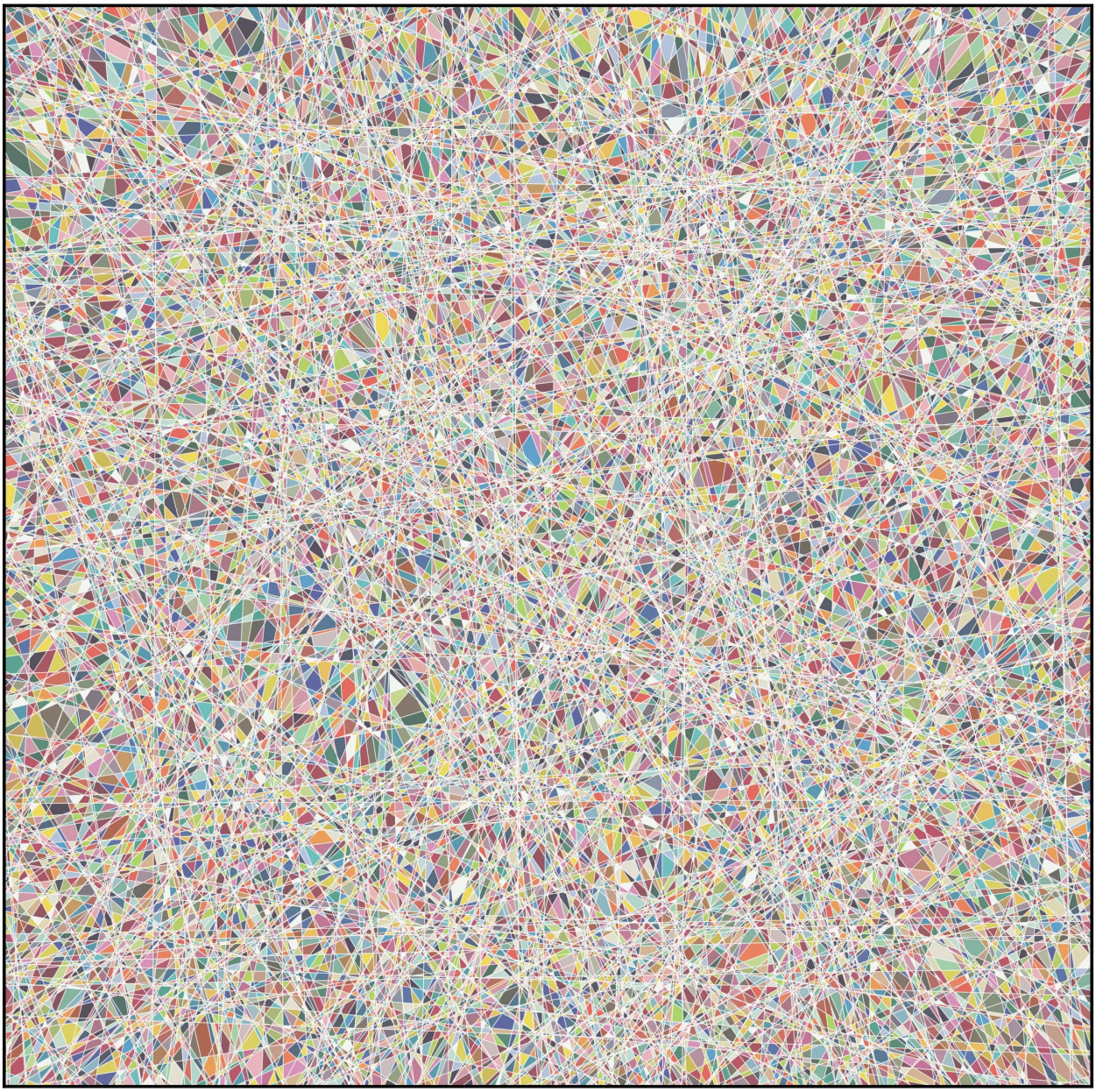

Figure 18. Partition visualization of a randomly initialized single layer MLP with 1000 hyperplanes and input dimensionality of 8002 for a randomly oriented 2D square domain centered on the origin. The partition contains 132569 regions and takes $134s$ to compute.

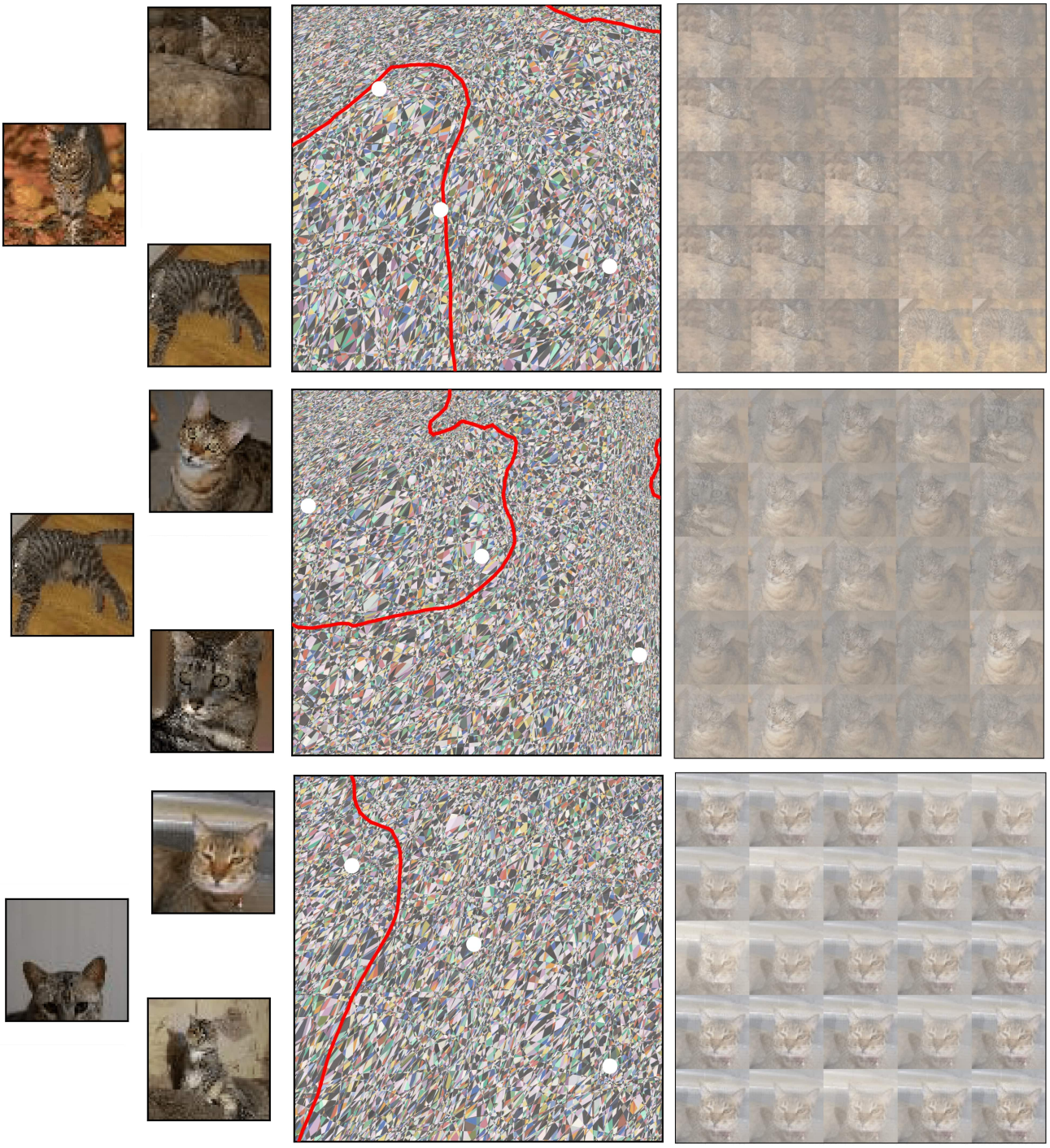

Figure 19. Partition visualization of a convolutional neural network, trained for binary classification of tinyimagenet Tabby Cat and Egyptian Cat classes. **Left** Samples that are used as anchor points to determine the 2D slice with an area of 450 units. **Middle** The partitioning of input space induced by the model as well as the decision boundary (in red). **Right** Randomly sampled points from the decision boundary. Samples from the decision boundary visually represent a linear combination of the three anchor samples, while the weights are determined by the non-linear decision boundary. For example, for the top and bottom rows, samples from the boundary look biased towards two of the three anchor points. Computing the partition regions take 7.46 mins, 13.96 mins and 5.02 mins respectively, with each of the partitions containing 82817, 119895, and 60455 regions. The number of regions is positively correlated with the curvature of the decision boundary in the neighborhood.

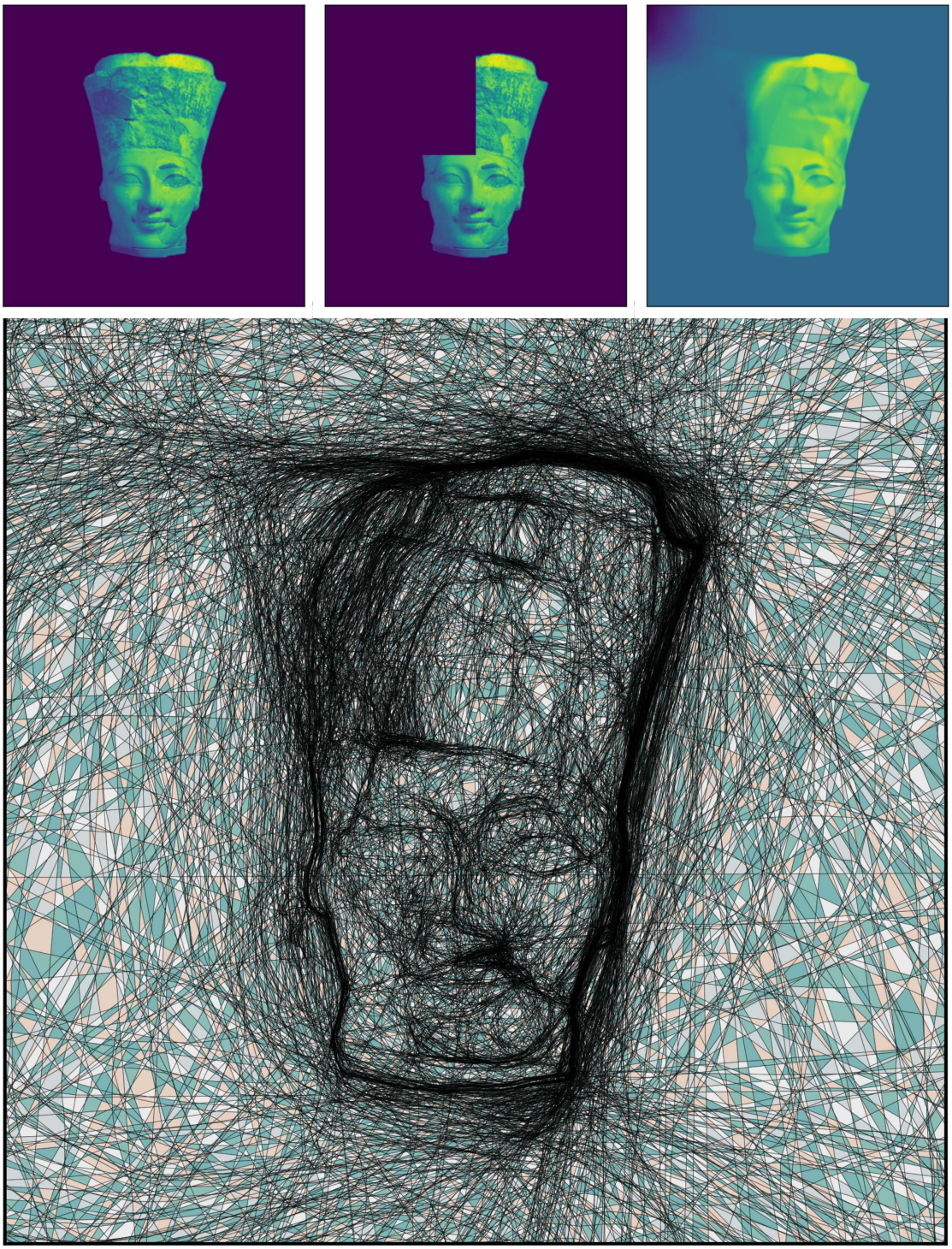

Figure 20. Training and visualizing the partition of a 2D INR trained on an inpainting task. Top row shows the original training image, the training image with a section cropped out and the predictions of a trained MLP with width 256 and depth 6. The MLP, via implicit regularization, learns a smooth surface in place of the discontinuity. Note how a lot of neurons are placed in the cropped region, and also around the foreground boundaries, to allow more curvature while fitting.

E. Usage of SplineCam

We provide SplineCam as a python toolbox that can wrap any given Pytorch [30] *sequential* network, containing a set of supported modules.

To begin, first we have to define a 2D input space region of interest (ROI). The region of interest is can be a polytopal region at the input of the network defined via vertices. Since all the region finding operations are performed in 2D, we also require an orthogonal projection matrix that projects vectors from the input space on to the 2D ROI. Following this we can use the SplineCam library to wrap a given model.

```
import torch
import splinecam

## given torch model and domain (ROI) as a list
    of vertices

T = splinecam.utils.get_proj_mat(domain)

model.cuda()
model.eval()
model.type(torch.float64)

print('Wrapping model with SplineCam...')
NN = splinecam.wrappers.model_wrapper(
    model,
    input_shape=model.input_shape,
    T = T,
    dtype = torch.float64,
    device = 'cuda'
)

## check .forward() and matmul operation
    equivalence
print('Verifying wrapped model...')
flag =  NN.verify()
print('Model.forward and matmul equivalence check
    ', flag)
assert flag

#
```

Listing 1. Wrapping a model with splinecam

SplineCam supports custom layers as well. Each SplineCam layer requires a submodules to return the weights, the intersection pattern and activation pattern of the layer. We refer the reader to our codes for details. The wrapped SplineCam model contains a verification method, to ensure that the affine operations and the forward operations (which can be different from the affine operation based on implementation, e.g., convolution) of the model result in the same value for random inputs.

SplineCam can take any set of 2D domains as ROI, with corresponding projection matrices. This allows SplineCam to be used to visualize the partition for piecewise linear subspaces in the input space. The following example shows how SplineCam computes the partition in a layerwise fashion.

```

## for a given list of polygons in 2D and
    corresponding projection matrices

Abw = T
out_cyc = poly

for current_layer in range(1,len(NN.layers)):

    ## given a set of 2D regions and a target
    layer, find all new regions in 2D
    out_cyc,out_idx = splinecam.graph.
    to_next_layer_partition(
        cycles = out_cyc,
        Abw = Abw,
        NN = NN,
        current_layer = current_layer,
    )

    ## acquire region centroids
    means = splinecam.utils.get_region_means(
    out_cyc)

    ## pass each region centroid to next layer
    means = NN.layers[:current_layer].forward(
    means)

    ## get activation mask for each region
    q = NN.layers[current_layer].
    get_activation_pattern(means)

    ## query network weights
    Wb =  NN.layers[current_layer].get_weights()

    ## calculate affine parameters per region
    Abw = splinecam.utils.get_Abw(
            q = q,
            Wb = Wb.to_dense(),
            incoming_Abw = Abw)

#
```

Listing 2. Modular code for computing spline partition

One of the key algorithms in the `to_next_layer_partition(.)` function is the search algorithm that allows us to find cycles from a given graph. The following codeblock presents a pseudocode of our heuristic breadth first search method.

```

from graph_tool import topology

def find_cycles(V=input_graph,start_edge=
    input_edge):
    '''
    Given a undirected graph and a starting edge
    find cycles from that edge
    '''

    ## Convert V to a bidirectional graph. This
    allows us to control
```

```
    ## the number of traversals for each edge
    V = convert_to_bidirectional(V)
    edge_q.append(start_edge)

    ## if edge is a boundary edge
    V.remove_edge(start_edge)

    out_cycles = []

    for e in edge_q:

        remove_q = []
        vertices = []
        vertex_id = []

        ## if no way out of v0 or no way in for
    v1, continue
        if not (V.get_in_degrees(e.vertex1)>1
            ) and (
            V.get_out_degrees(e.vertex0)>1):
            continue

        ## if the edge doesn't exist, continue
        if V.edge(e) is None and V.edge(e.vertex1
    ,e.vertex0) is None:
            continue

        ## add opposite path to removal queue
        remove_q.append(V.edge(e.vertex1,
                    e.vertex0)))

        ## bfs
        vs,es = topology.shortest_path(V,
                 source=e.vertex0,
                 target=e.vertex1,
                )

        out_cycles.append([V.vertex_index[each]
    for each in vs])

        for new_e in es:

            ## if boundary edge remove edges in
    both directions
            if V.ep['layer'][new_e] == -1:
                remove_q.append(new_e)
                remove_q.append(V.edge(new_e.
    vertex1,new_e.vertex0))

            else:
                ## remove only one direction and
    append to queue
                remove_q.append(new_e)
                edge_q.append(new_e)

        for each in remove_q:
            V.remove_edge(each)

    return out_cycles
```

Listing 3. Pseudocode function for finding cycles given a undirected graph