# VISCNN: A TOOL FOR VISUALIZING INTERPRETABLE SUBGRAPHS IN CNNS

A PREPRINT

**Christopher Hamblin***
*Department of Psychology*
*Harvard University*
chrishamblin@fas.harvard.edu

**George A. Alvarez**
*Department of Psychology*
*Harvard University*
alvarez@wjh.harvard.edu

September 20, 2021

**ABSTRACT**

Deep convolutional neural networks are not only a powerful model class for a range of computer vision problems, but also a highly interpretable model class. Numerous tools have been developed for probing the image filtering process latent in CNNs, with respect to both data representation, *what* latent features in the network encode, and data processing, *how* learned algorithms achieve those representations. Unfortunately, with respect to data processing interpretability, the sheer number of operations performed by deep CNNs makes it difficult to follow the 'full' algorithm that leads from pixel inputs to deep layer features. To combat this barrier to interpretability we've developed VISCNN, a tool for visualizing interpretable subgraphs in CNNs. VISCNN leverages methods from neural network pruning in order to quickly query CNNs for modular subgraphs that preserve the expression of downstream features. These subgraphs can be extracted along a continuum of complexities, allowing the researcher to 'build-up' their understanding of how the network generates rich features. VISCNN natively integrates a range of interpretability tools to aid in the exploration of subgraphs. In this article we lay out the methods underlying VISCNN, provide evidence for the efficacy of those methods, demonstrate the flexible use-cases for the tool, and highlight some interesting discoveries we've made using VISCNN.

***K*eywords** CNN Interpretability · Circuits · Pruning · Feature Visualization · Software tool

## 1 Introduction

There is no doubt that modern deep learning techniques have dramatically changed the way engineers approach computer vision problems, replacing hand-engineered image processing algorithms with large neural networks trained end-to-end. While this has improved model performances on a range of tasks, it leaves the underlying algorithms opaque. As deep-learned models become ubiquitous for computer-vision applications, this poses a notable concern, as it's difficult to know exactly why and when such models will go wrong. Deep-learned models are also not intrinsically valuable to the vision science community, who's primary goal is an explanatory account of how visual processes works. However, if deep-learned vision models could be made interpretable, they would be useful objects for empirical study, as highly accessible alternatives to biological neural systems.

Work in neural network interpretability can be divided into two broad categories, that of data representation, and data processing [1]. Most interpretability efforts in computer vision concern data representation, or *what* the network represents, in the conceptual sense. The focus on representation usually leads to methods for mapping network features back into pixel space, where the researcher can employ their own visual capacities to make inferences about the latent space of the network. Examples of representational interpretability methods include viewing the output feature maps of a cnn filter, the attention map in a visual transformer, viewing dataset examples or feature visualizations [2] that maximally excite/inhibit a latent feature, or attribution methods [3] [4] that highlight pixel regions most responsible for a model's response. Much less common are accounts of data processing, where the model is described as an interpretable

*Chris Hamblin is the corresponding author of this paper. Email him for general questions, and use github issues https://github.com/chrishamblin7/viscnn/issues for questions regarding usage of the subgraph visualizer software.

sequence of computations that generate representations. So, while a researcher might be quite confident a feature in their model acts as, say, a blue bird detector, they often still have no idea how a blue bird detector works. The model has not revealed to the researcher the computational principles that would allow them to hand-engineer such a detector.

The architectural constraint of convolutional neural networks allows for greater data processing interpretability than many other deep models. The kernel convolution operation not only reduces the number of parameters to contend with by replicating weights across space, but also constrains the function space learnable by the network to a familiar subspace. Kernel convolution operations are typical in hand-engineered image processing algorithms, where single kernels can enable filters for edge detection, sharpening, blurring, spatial frequency filtering, etc. Composing filters in a hierarchy enables the construction of more complex detectors, with each kernel acting as a spatial template for the arrangement of features from the previous layer. Olah et. al. 2020 [**?**] show that viewing feature visualizations in conjunction with the kernels connecting them provides strong visual intuition for how simple features are combined into complex ones. The authors describe many interesting 'circuits' they've discovered in InceptionV1 [**?**], from early circuits encoding curves and basic shapes, to late-layer circuits encoding real-world objects.

There could be many interesting circuits embedded in deep CNNs, what remains unclear is a principled process for finding them. Where Olah et. al. typically demonstrate circuits with 3 kernels connecting one feature to the previous layer, many contemporary CNN architectures have hundreds of kernels in each convolutional filter, each connecting to filters in the previous layer that themselves have hundreds of kernels. 'Zooming in' on small parts of a network reveal human comprehensible features, but at the cost of ignoring potentially relevant influences on those features, painting an impoverished picture of what makes those features activate. Additionally, circuits do not exist in isolation, and as such it may be useful to quickly 'zoom out' again, to see how multiple circuits wire together in the larger context of the full network.

VISCNN is a software tool designed to aid in the empirical process of finding circuits embedded in deep CNNs. It does this by first projecting CNNs as graphs, where each edge represents convolution with a single 2D kernel, a human interpretable operation. Next, as these full network graphs are far too large to parse effectively, VISCNN allows the user to *query* the network for subgraphs that preserve user-specified features with respect to a set of user-specified input images at a desired sparsity level. This enables the user to sequentially follow through the image filtering algorithm underlying features in their network. As this querying is based on quickly computable pruning metrics, the VISCNN user can explore different subgraphs in their model interactively in real time through a point-clickable GUI. This paper will proceed in three parts. First, the *Methods* section will explain the techniques we use to project CNN's as a spatially embedded graph, and how we query that graph for important nodes, edges, and connected subgraphs. Second, in the *Using VISCNN* section, we guide you through an example exploratory analysis of Alexnet using the VISCNN GUI, uncovering some interesting latent representations and algorithms for generating them along the way. In the last section, *Experiments*, we validate our subgraph extraction method by demonstrating it is indeed *'feature-preserving'*.

## 2 Methods

### 2.1 Graph Representation of CNN

Before explaining our method for extracting circuits from a CNN, let's first be explicit about the conventions we're using to represent a CNN as a computational graph. In the case of vanilla ANNs, the conventional graphical representation of a network passes scalar values along edges between nodes. Each edge has an associated scalar weight, which is multiplied by the edge's input scalar to produce the edge's output. A node in the graph has incoming edges and outgoing edges. It adds together the outputs from each incoming edge then applies a non-linear activation function, passing the result as input to each outgoing edge *(Fig 1a)*. Our graphical representation of a CNN will be analogous, except instead of passing scalar values, the graph will pass 2D arrays (images/activation maps). Each edge will have an associated 2D kernel array, which it will convolve with the edge's input array (with some fixed stride, padding, etc. associated with that edge) to produce the edge's output array. A graphical node (convolutional filter) takes the output arrays from each incoming edge and adds them together point-wise, then applies a point-wise activation function, passing the resultant array as input to each outgoing edge *(Fig 1b)*. This graphical representation of a CNN allows us to define circuits as connected subgraphs of the network, as well as visualize those subgraphs within the context of the whole network. Subgraphs through the network can be sequentially followed and interpreted as image filtering pipelines.

### 2.2 Circuit Pruning

Now, suppose we want to extract an interpretable circuit from our network that underlies some feature of interest to us. We want the circuit to be a minimal computational subgraph that outputs a close proxy of our feature. On a given set of input images, we want to minimize the difference between the original feature map values computed in the full network and the values outputted by our circuit. Under such a formulation, its sensible to draw from the neural network pruning literature, where the conventional goal is to tune a small network latent in a large network that performs well on the objective function. Here we are pruning with respect to reconstruction loss on targeted features, rather than

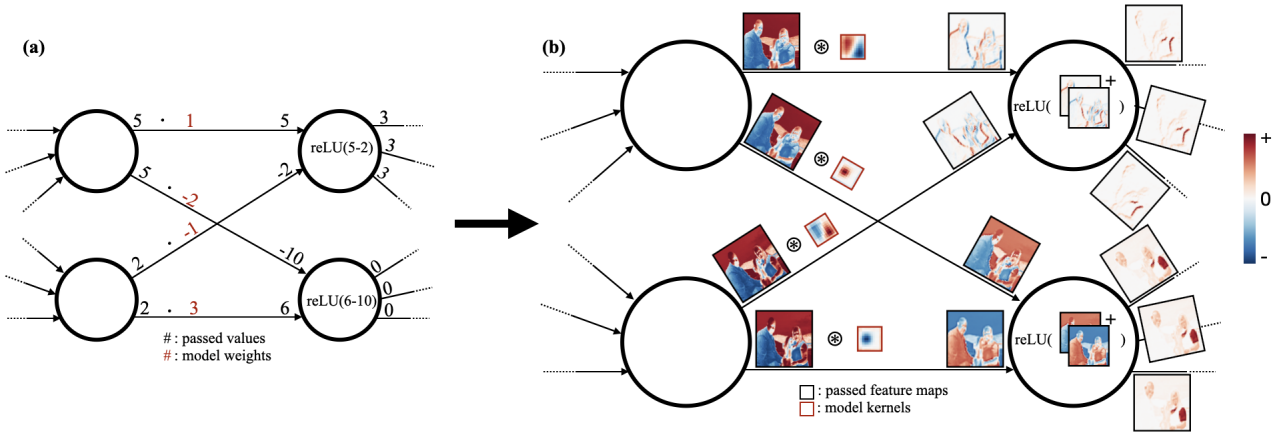


Figure 1: **(a)** The conventional graphical representation of ANNs. **(b)** An analogous graphical representation for CNNs, operating on feature maps instead of scalars

the network's objective loss. Of particular interest to us are methods for *structured* pruning, where architecturally bound groups of weights are pruned, rather than individual weights [**?**, **?**]. Pruning nodes and edges in our CNN graphs corresponds to the structured pruning of convolutional filters and kernels respectively. Pruning techniques can also be organized by how network compression is integrated with the training of the network. At one end of the spectrum are regularization techniques, where a penalty for large weights is built directly into the loss function minimized during training, encouraging weights to zero [**?**, **?**]. Alternatively, pruning and tuning the network can be separated and performed iteratively. Typically with these methods, quickly calculable heuristics are used to assign 'importance' scores to weights (or filters/kernels). A small number of the least 'important' weights are removed, and then the network is fine-tuned, in an iterative cycle [**?**, **?**]. Much recent work has focused on separating pruning and training entirely, such that a large number of weights are removed before training in a 'single-shot' [**?**, **?**].

A thorough comparison of pruning algorithms is beyond the scope of this work, which instead focuses on how pruning can be leveraged for the purposes of network exploration and interpretation. Here we will adopt the structured pruning algorithm of Molchanov, P. et al.(2017), which prunes entire convolutional filters for the purpose of transfer learning large networks to simpler objectives. They assign an importance score to each filter proportional to the change in loss induced by removing the filter from the network, as filters that don't affect the loss upon removal are 'unimportant' to the objective and can be pruned. As removing each filter individually and running the full dataset through the model to check the change in loss is intractable, the authors approximate this change by taylor expansion. Let's consider some particular activation $a_i$ in a feature map. If the filter that produced that activation map were removed, that activation would be 0, Thus for a given loss function $L$, the importance $I$ of the activation is proportional to the change in $L$;

$$I_{a_i} \propto |\triangle L(a_i)| = |L(a_i) - L(a_i = 0)| \tag{1}$$

Where $L(a_i)$ is the current loss with the current activation, and $L(a_i = 0)$ is the loss when that activation is removed (set to 0). The first-order taylor approximation of $L$ at $a_i = 0$ gives;

$$L(a_i = 0) = L(a_i) - \frac{\delta L}{\delta a_i} a_i \tag{2}$$

Plugging (2) into (1);

$$I_{a_i} \propto |\triangle L(a_i)| \approx |L(a_i) - L(a_i) - \frac{\delta L}{\delta a_i} a_i| = |\frac{\delta L}{\delta a_i} a_i| \tag{3}$$

This importance metric of activation*gradient makes a lot of intuitive sense. An activation is important if it is large, setting an activation close to zero already to zero presumably won't affect much. An activation also must have a large gradient to be important; even if an activation is large, if it has a small gradient changing that activation's value might not have a significant downstream effect.

From here, the authors define the importance of an entire feature map $M_j$ on a given dataset $\mathcal{D}$ by averaging the importance scores of all activations in that feature map across all images in the dataset. Supposing $m$ is the length of

vectorized feature map $M_j$, $d$ is number of images in $\mathcal{D}$, and $a_i^k$ is the $i^{th}$ activation in the map given input image *k*;

$$I_{M_j} \propto \frac{1}{dm} \sum_{k=1}^{d} \sum_{i=1}^{m} |\frac{\delta L}{\delta a_i^k} a_i^k| \tag{4}$$

As removing a filter is equivalent to removing the feature map it outputs with respect to $L$, the authors use these output feature map importance scores to iteratively prune low scoring filters. Filters correspond to nodes in our network graph, but as edges also output activation maps (which are added point-wise by the filter/node) we can use the same importance metric to prune edges, applied to the activation maps they output. If we want to prune our network graph with respect to some downstream target feature, instead of the final loss of the network, we can simply replace $L$ in equation (4) with the average activation in that feature's activation map. Then our importance metric will be approximating how the feature's response to an imageset would change, given each node/edge were removed.

[5] [6][7][8]

### 2.3 Spatially Projecting the Graph

The last thing we must contend with is how our graph is projected into space. Some of this is determined by our model architecture; for a feed-forward CNN it makes sense that one dimension encode moving forward through the network, layer-by-layer. For the projection of nodes within a layer, we want a 2D topology that reflects each nodes membership in different circuits. There are many reasonable ways to do this, and VISCNN provides the user several options under the *Projection* field in main GUI. The projection we'll primarily utilize in the following demo section will be 'class-wise UMAP'. This projection organizes nodes based on their importance towards preserving model outputs for each distinct class in the training dataset. We can get this class-wise importance by applying our taylor approximation metric with respect to the final model loss, averaged over all images of a given class. For ImageNet [**?**] data, this assigns a 1000-dimensional class-wise importance vector to each node. We can think of each node as positioned in 1000-dimensional class-wise importance space, and we can visualize the distances between nodes within a layer in 2 dimensions with UMAP [9]. We don't want the 2D topology to simply reflect which nodes are generally important and which aren't, but this could have a large effect on the distances between nodes in importance space, as unimportant nodes would be close together near the origin, far from the generally important nodes, which are scattered in positive space. To help ameliorate this we L2 normalize each node's 1000-dimensional importance vector to length 1 before running UMAP. As the resulting projections are not always ideal for a clean user interface, these layer-wise plots can be smoothed to a circle with uniform density (See Supplementary materials). An optimal rotation for each layer projection can be found by selecting those rotations that minimize the weighted length of edges between layers, where edges are weighted by their overall importance score for preserving the loss. A useful alternative projection is the 'feature-viz UMAP' projection. Here each node is assigned a vector based on its activation in response to a feature visualization of each other node in the layer. As such, nodes excited by another's feature visualizations will be close to each other, and those inhibited by another's feature visualization will be far. This is useful when visualizing non-classifier models, as well as early layers that might not have high-level semantic categories attributable to nodes.

## 3 Using VISCNN

### 3.1 Main GUI

VISCNN takes as its starting point a CNN model specified in PyTorch. VISCNN is flexible with respect to the models it can visualize, allowing for custom models provided by the user, but in its current iteration assumes some architectural constraints. Namely, the model must consist of a sequence of convolutional layers without branching or skip connections, as is the case for models like Alexnet, VGG, Lenet, etc (See Supplementary materials). For demonstrative purposes, here we will run through an exploratory analysis of Alexnet trained on Imagenet. While VISCNN assumes a starting point of a static, trained model, and will performing no tuning, it is beneficial to visualize models that have been trained with interpretability in mind. Essentially, the smaller the model, and the sparser its activations, the better. Our version of Alexnet has been trained using hierarchical group sparsity regularization [10] (see supplementary materials), to encourage L2 norm of as many 2D kernels in the network as possible towards 0. This has the effect of eliminating edges from our network graph from the start, forcing the network to have sparse, modular subgraphs that will be easier to analyze.

*Fig 2* shows a screenshot of the main GUI for the VISCNN tool, and the figures that follow are pulled from subsections of that interface. This interface can be roughly divided into three sections; the querying module in the top left, the network graph representation in the top right, and node/edge-wise interpretability tools along the bottom row. With the query module, the user can highlight the graph's nodes and edges based on different criteria, such as those graph components with large weights, those with large output map activations, those with large output map gradients, or those with large activationxgradient, the importance criterion used in Molchanov et. al. Activations are calculated with respect to a set of *input* images and gradients with respect to an *output* target. A good place to start our exploratory

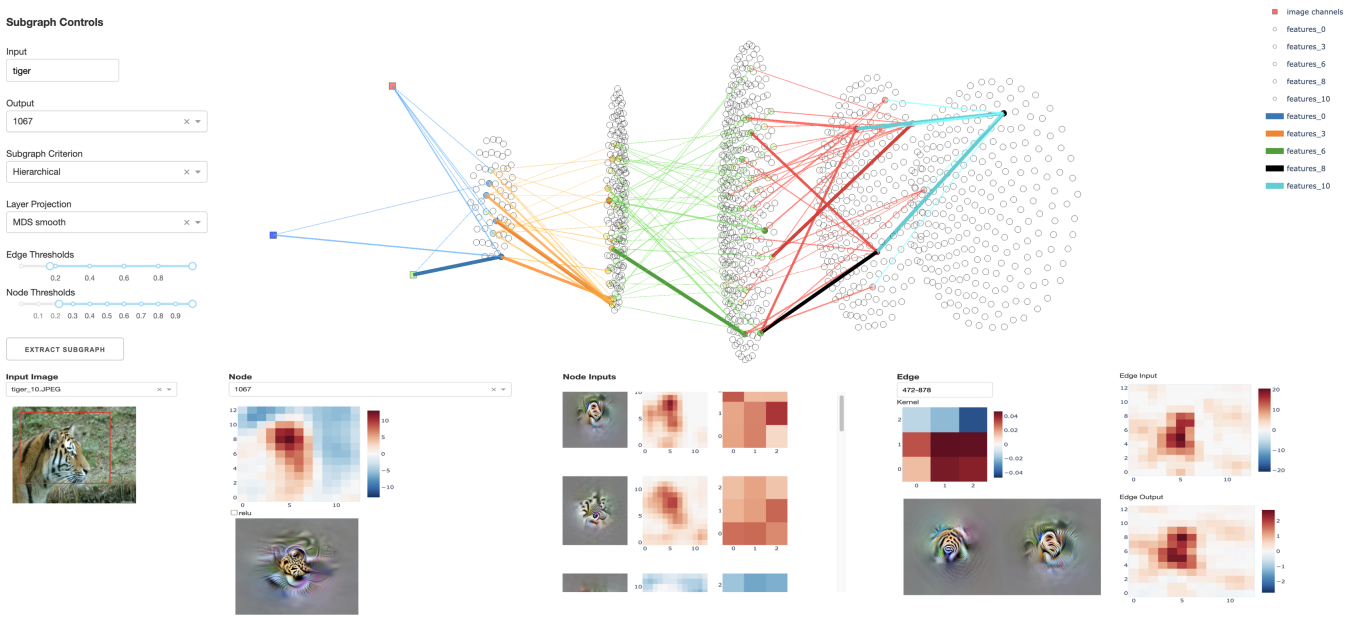


Figure 2: Screenshot of the main display of VISCNN, visualizing the convolutional layers of Alexnet. Users can query for important nodes and edges in the net based on different input output criteria, as well as clickthrough the nodes and edges of the graph to see intermediate activation maps and feature visualizations.

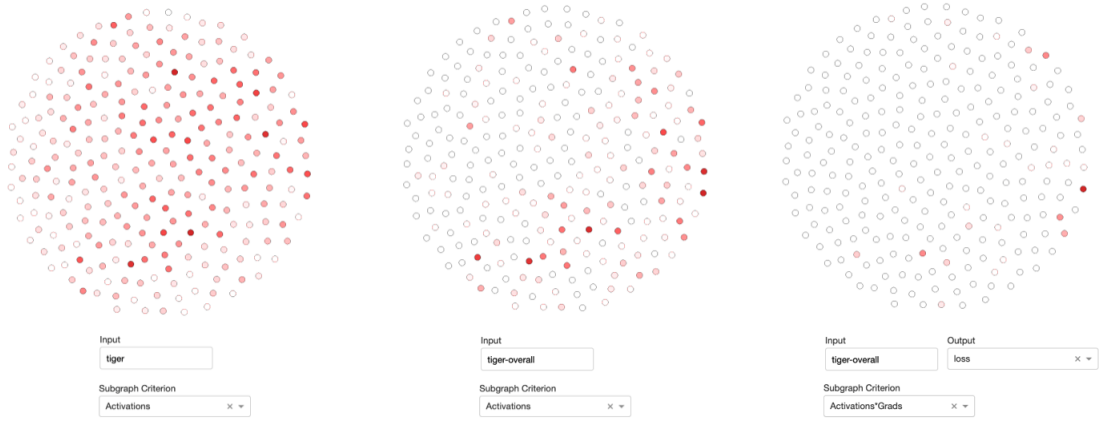


Figure 3: The nodes (filters) the in fourth layer of Alexnet have been isolated in the network graph. Progressively more selective queries leads to a sparse highlighting of nodes in this layer.

analysis is to query for nodes in our network with large activations on *input* images of a particular Imagenet class. This allows us to quickly identify intermediary nodes in our network that might encode something semantically meaningful. The *Input* field in the query module can accept linear combinations of images or imagesets, allowing for queries like "tiger - overall", which highlights nodes with larger activations on images of tigers than activations on the whole of Imagenet (analogous to a neuroimaging 'contrast'). Querying with the *Output* field set to "loss" and the *Criterion* field set to "ActivationsxGradients" highlights graph components that don't just activate in response to tiger images, but are *important* for preserving the network's final loss with respect to tiger images *(Fig 3)*.

Clicking on a node or edge in the network graph selects that graph component for closer inspection in the corresponding *Node* and *Edge* areas along the bottom of the GUI. Here the user can view the feature visualization [**?**] for a selected node/edge, a gradient ascent synthesized image that maximally activates the activation map output for that graph component. Feature visualizations of the most important nodes for various Imagenet classes provide converging evidence of their class-wise importance *(Fig 4)*. The user can easily toggle between feature visualizations with a 'channel-wise' objective and 'neuron-wise' objective, which synthesizes an image optimizing the average activation across the feature map and the activation of the central unit in the feature map respectively.

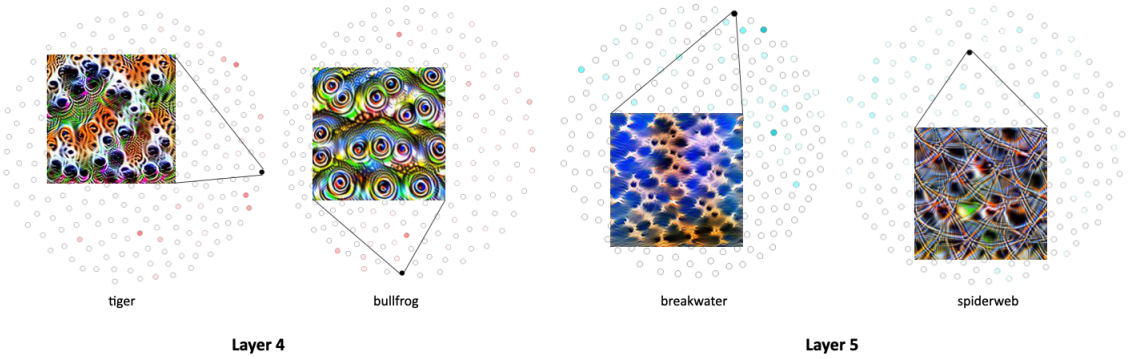


Figure 4: By clicking on nodes (which become black when selected) we can view their associated feature visualizations. Nodes important for preserving the network's loss on a particular class of images often have feature visualizations suggestive of the image class. Feature visualizations are generated using the Lucent library [**?**], a Pytorch implementation of the popular Lucid library [**?**]

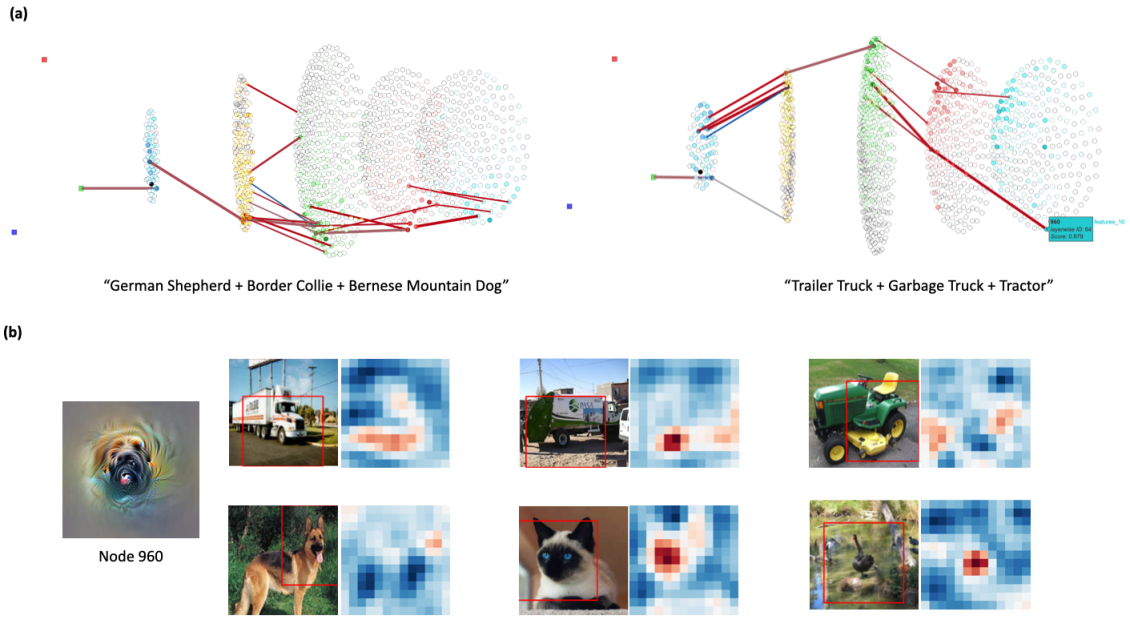


Figure 5: a) Important nodes and edges for dog classes on the left and truck classes on the right over all convolutional layers of Alexnet. b) Feature visualization of 'misplaced' node 960, and its feature map responses to truck and animal images.

We can also use this bottom panel to select an *input image*. This image is separate from those used to query the graph structure, rather they are passed through the model to generate intermediate feature maps, which are displayed in the adjacent 'node' and 'edge' panels. As its often difficult to identify exactly which activations in a feature map correspond to pixel regions in the input image, we've built in a functionality where clicking an activation in a feature map outlines its receptive field in the input image.

Querying more than one layer at a time with respect to multiple images often reveals a sensible organization in our graph projection. *Fig 5.a* shows the result of two queries, one for dog classes "Border Collie + German Shepard + Bernese Mountain Dog" and the other for truck classes "Tractor + Trailer Truck + Garbage Truck". The returned important nodes and edges clearly separate to opposite sides of the graph, reflecting the large perceptual difference between these two sets of images. But notice the important truck node 960 in the final layer seems to be 'misplaced', residing down in what is presumably animal territory? Simply clicking on the node to view its feature visualization and inspecting some of its feature maps quickly resolves the issue. It's a detector for black spheres, and as such detects both tires and animal features *Fig 5.b*.

So far, we've mostly explored data *representation* in network nodes (convolutional filters) as revealed through VISCNN. We have not explored the role of edges connecting nodes, which perform individual 2D kernel convolutions

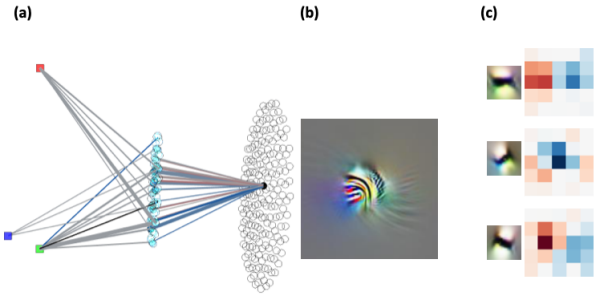


Figure 6: a) Highlighted graph of important nodes a edges leading to curve detector. b) Feature visualization of curve detector. c) Top edges leading into curve detector.

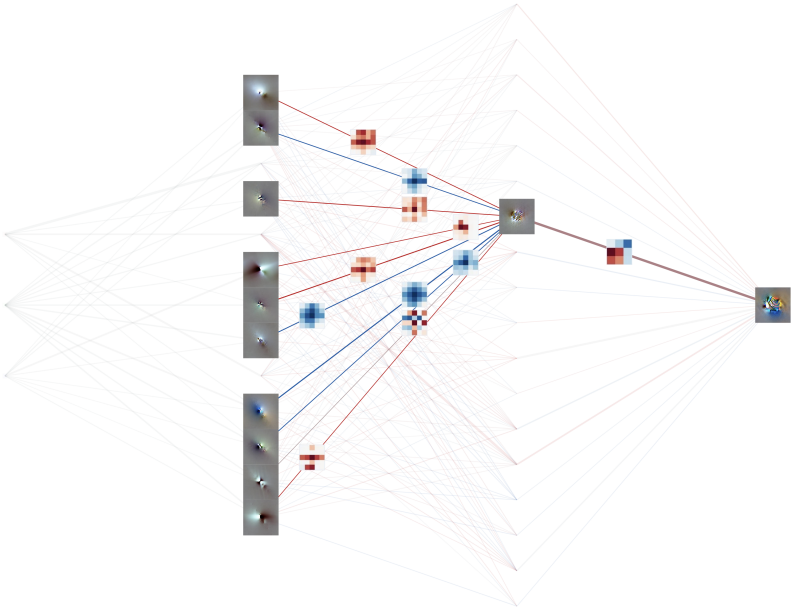

Figure 7: Screenshot of the circuit GUI for VISCNN. User has highlighted an intermediate node, to view its incoming and outgoing connections.

on feature maps, and are critical to understanding data *processing* in the network, one of our primary motivations. As data processing is far more difficult to interpret than data representation, let's take a few steps back from the late-layer semantic nodes we've been analyzing and look to the early layers, starting with how VISCNN reveals the algorithm underlying a layer 2 curve feature. When selecting a node for closer inspection in the node panel, the user is shown by default an additional list of *node inputs*. Each node input in the dropdown list corresponds to an edge leading into the selected node, with edges sorted by edge *importance* as determined by the query module. Each entry displays the edge's output feature map, it's kernel matrix, and the feature visualization for its input node. Viewing the feature visualizations and kernels of these *top* edges often provides a rough sketch of the algorithm implemented to generate the target feature with respect to representations in the previous layer. *Fig 6* shows these top ranking edges leading into a layer two curve feature. The curve feature is connected to simple edge features from the first layer. The kernels connecting these edge features is positive is locations where the edge should be tangent with the curve, and inhibitory where the edge is perpendicular to it. As such, most of the lines representing these ends in *Fig 6.a* are grey, as they are necessarily both exhitatory and inhibitory (edges that are mostly exhitatory or inhibitory are colored red and blue respectively).

## 4 Experiments

### 4.1 Validating Subgraphs

Its important we confirm subgraphs generated with our method are actually *feature preserving*. By this we mean the activations output by subgraphs for features are close to those activations returned by the original features in the

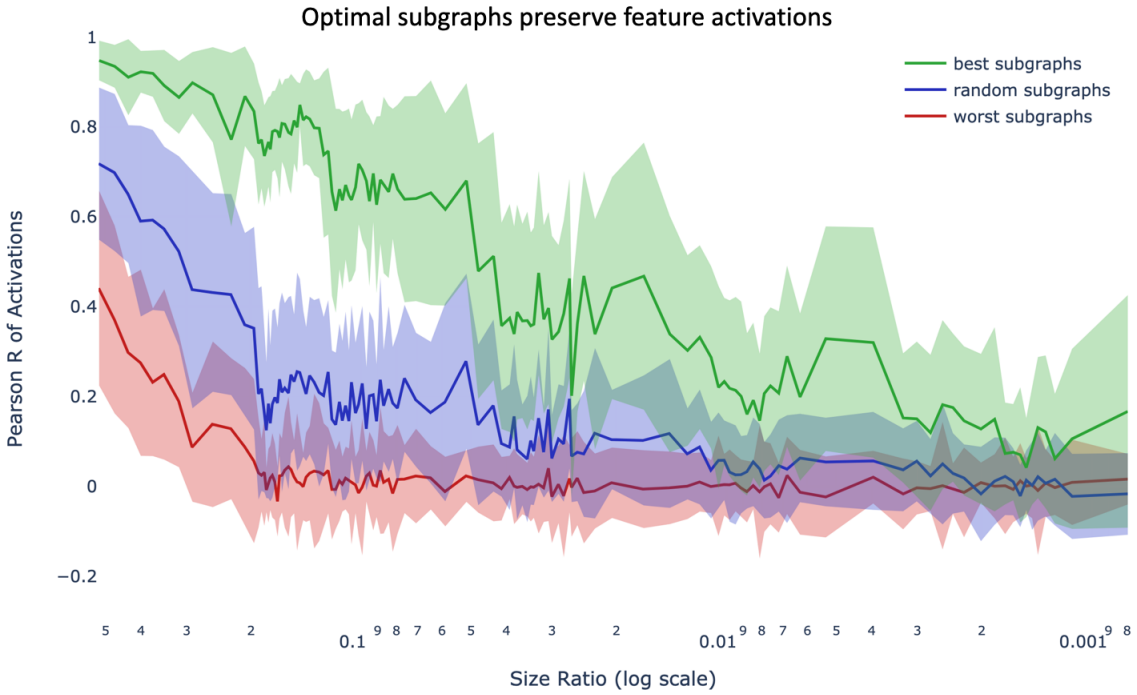


Figure 8: Subgraphs extracted using our method preserve feature responses well, even when 95% of edges are removed from the graph.

full network. Really, we want the activations between subgraph features and original features simply to be correlated. Suppose a subgraph feature returned all the original feature values but divided by 2. Such a subgraph would still give a perfect account of what that feature does and does not respond to, it's merely off by a normalizing factor. To test these correlations in the general case, we extracted 4 subgraphs per node in our network, meeting 4 different cumulative importance thresholds. These subgraphs are extracted with respect to a random spanning set of 2000 input images, with 2 examples per image class. We then extract 2 architecturally identical networks for each of these subgraphs, a *random* subgraph, in which random nodes and edges are selected for addition to the graph, and a *worst* subgraph, in which nodes and edges are added to the graph in order from worst to best importance scores (rather than best to worst as is done for the original *best* subgraphs). A different, random spanning set of 2000 images is then passed through the original model and each subgraph, and every original node activation is correlated with the associated activation produced by each subgraph. In The correlation values for each subgraph type are plotted against the size ratio of subgraphs and the original feature graph (as per the number of edges).

A more qualitative validation of subgraphs can be accomplished through revisualizing their target feature. A good subgraph's feature visualization should reflect the original visualization of the feature its meant to capture. *Fig 9.a* conveys such visualizations and the resemblance is clear, especially when compared to visualizations for 'random' and 'worst' subgraphs as described above. *Fig 9.b* shows how the feature degrades as a subgraph becomes progressively sparser. Conversely, one could think of this as 'building up' a feature through making a subgraph more complex.

### 4.2 Sub-Feature Subgraphs

All the subgraphs we've extracted so far attempt to capture features as a whole, maintaining their responses across the whole of pixel space. This may lead to sparse, parsable subgraphs for early features like curve detectors, but more complex features later in the network can still have subgraphs with $10^4$ edges (a 10-fold decrease from the full graph leading to the feature, but still daunting). These features are *complex* largely due to their invariances; they show an equivalent response to stimuli that at another level of representation are very different. In this experiment, we leverage this invariance structure to break up complex late-layer features into multiple, dissociable subgraphs, each responsible for detecting one sub-feature. The invariant feature can be thought of as an 'or' operator over the various sub-features generated by these subgraphs.

Rather than scour the network for features that seem to activate in response to perceptually dissimilar image patches, we took a passive, data-driven approach to extracting sub-feature subgraphs. For each feature (node) in Alexnet, we identified the top 300 image patches from ImageNet that generated the highest activations for that feature. This

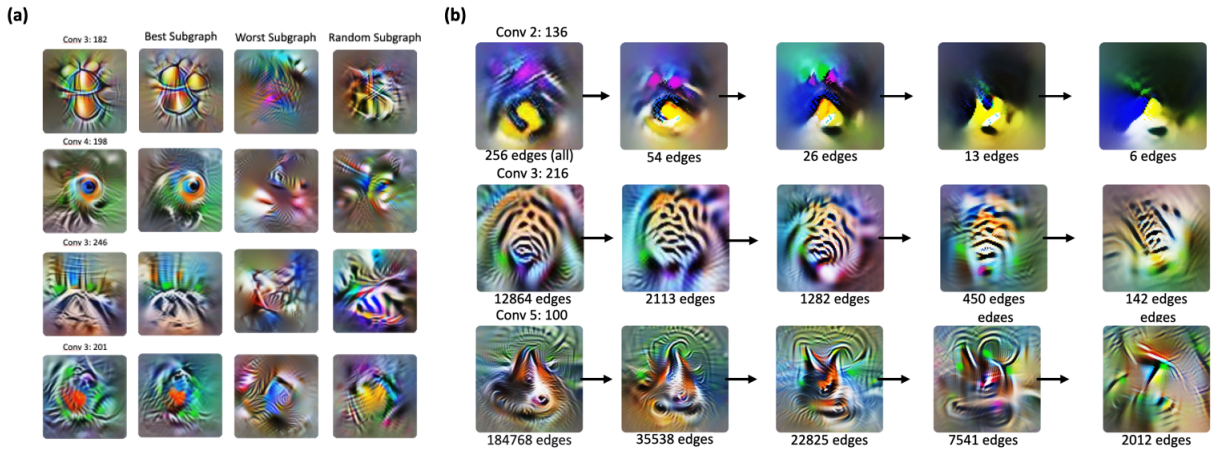


Figure 9: a) Revisualizations of feature subgraphs. b) Revisualizations with different sparsities.

was done under the constraint that each image patch be located in a different image in the dataset, to avoid selecting spatially adjacent image patches that are highly correlated or overlapping. We then considered how these 300 image patches activate all *other* nodes in the target node's layer. If two image patches have close activation vectors in the layer, it's not unusual that our target node is highly responsive to both, the image patches just have similar features in general. If however, two image patches have far activation vectors in the layer, the fact that our target feature is invariantly responsive to both image patches suggests separable data processing streams acting on each image patch, which are combined by the target feature.

In order to find separable subgraphs underlying the same feature using our method, we need to identify sets of exhitatory images for the feature that have internally close activation vectors, but distant activation vectors across sets. If we extract subgraphs for the feature using these image sets to calculate importance scores, we are in effect extracting subgraphs that generate the feature's high activations *just* to those images, a presumably different computation than that underlying the model's high response to the other image sets. We can search for these separable image sets using HDBScan clustering [11] of the activation vectors associated with image patches. The HDBSCan algorithm works well in high-dimensions and requires only a minimum cluster size as its hyperparameter. It will return anywhere from 0 to $\frac{numsamples}{minclustersize}$ clusters, allowing us to ignore features that don't show good clustering, and break other features into potentially many subgraphs.

Using a minimum cluster size of 20, we checked for clusters in the exhitatory image patches for every node in Alexnet, then generated subgraphs for each cluster found. *Fig 10* shows 3 examples of *polysemantic* features uncovered through this process. The first shows high activations to both images of terrier faces and tall metal tubes with horizontal striations, the second to textured red blobs and fine black and white mesh grids, and the third to monkey faces and text with a boxy surround. The feature visualizations for subgraphs associated with each of these clusters are clearly distinct from each other, and reminiscent of the image patches within each cluster. These cluster-wise, sub-feature subgraphs are often quite separable, as measured by the ratio of the intersection of their edges and the union of their edges. This separability is directly attributable to the distinct image sets used to calculate the subgraphs. Looking at the 15 most separable sub-feature subgraphs, if we scramble each associated image set into two mixed sets, the corresponding subgraphs for these new image sets are highly entangled, jumping from a .24 intersection ratio (.039 std) to a .70 intersection ratio (.061 std).

## 5 Conclusion Future Work

Going forward, we'd like to extend VISCNN in three main directions. First, we'd like to augment the software to allow for more architectural variation. This will enable us to study the effects of branching and skip connections, and explore models like InceptionV1 [**?**], which is known to have more semantic features than Alexnet or VGGnet, leading to cleaner feature visualizations *(Fig 11)*. This also opens the possibility of studying models with very different objectives than classification, such as styleGAN2 [**?**]. Second, we'd like to study the effects of different pruning algorithms. Of particular interest to us are algorithms for 'single-shot' pruning such as Grasp [5], which use the Hessian to approximate the interaction effects that result from removing a large number of parameters all at once. Lastly, we look forward to simply putting the tools we've built to use, unpacking the latent image filter algorithms in CNNs.

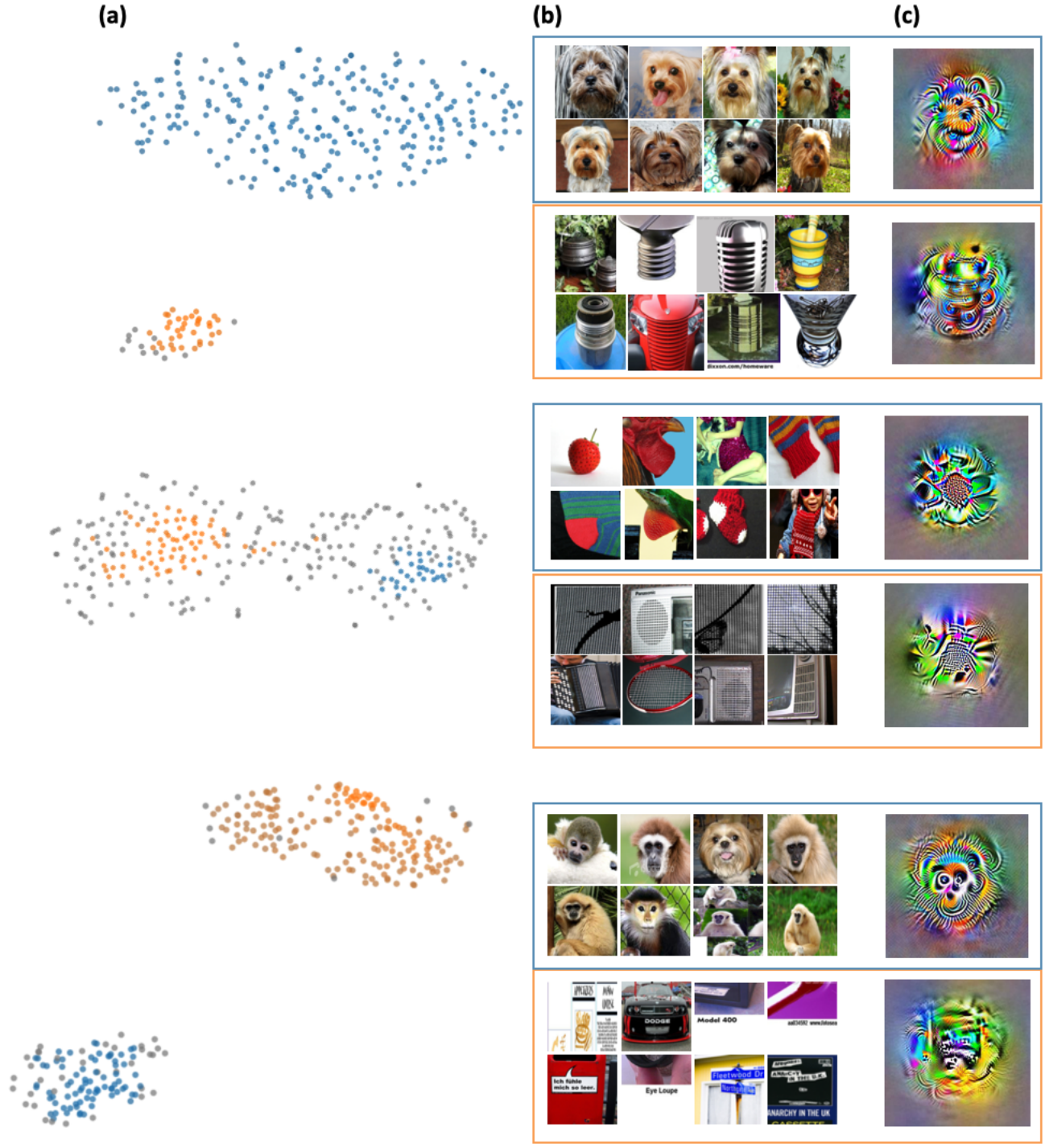


Figure 10: Three polysemantic features with separable subgraphs. **(a)** A UMAP projection of the layerwise activation vectors on the 300 most exciting image patches for the target feature, clustered (in the original dimensions) using HDBSCAN. **(b)** The top 8 image patches with highest activations for the target feature in each cluster. **(c)** feature visualizations of the subgraph extracted for each cluster.

Figure 11: 'Arch' feature in InceptionV1 (left) and Alexnet (right)

## References


[1] Leilani H. Gilpin, David Bau, Ben Z. Yuan, Ayesha Bajwa, Michael Specter, and Lalana Kagal. Explaining explanations: An overview of interpretability of machine learning. 2019.

[2] Dumitru Erhan, Y. Bengio, Aaron Courville, and Pascal Vincent. Visualizing higher-layer features of a deep network. *Technical Report, Univeristé de Montréal*, 01 2009.

[3] Sam Sattarzadeh, Mahesh Sudhakar, Anthony Lem, Shervin Mehryar, K. N. Plataniotis, Jongseong Jang, Hyunwoo Kim, Yeonjeong Jeong, Sangmin Lee, and Kyunghoon Bae. Explaining convolutional neural networks through attribution-based input sampling and block-wise feature aggregation, 2020.

[4] Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. *CoRR*, abs/1311.2901, 2013.

[5] Chaoqi Wang, Guodong Zhang, and Roger Baker Grosse. Picking winning tickets before training by preserving gradient flow. *CoRR*, abs/2002.07376, 2020.

[6] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient transfer learning. *CoRR*, abs/1611.06440, 2016.

[7] Robert Geirhos, Patricia Rubisch, Claudio Michaelis, Matthias Bethge, Felix A. Wichmann, and Wieland Brendel. Imagenet-trained cnns are biased towards texture; increasing shape bias improves accuracy and robustness. *CoRR*, abs/1811.12231, 2018.

[8] Wieland Brendel and Matthias Bethge. Approximating cnns with bag-of-local-features models works surprisingly well on imagenet. *CoRR*, abs/1904.00760, 2019.

[9] Leland McInnes, John Healy, and James Melville. Umap: Uniform manifold approximation and projection for dimension reduction, 2018. cite arxiv:1802.03426Comment: Reference implementation available at http://github.com/lmcinnes/umap.

[10] Kakeru Mitsuno, Jun'ichi Miyao, and Takio Kurita. Hierarchical group sparse regularization for deep convolutional neural networks. *CoRR*, abs/2004.04394, 2020.

[11] Leland McInnes, John Healy, and Steve Astels. hdbscan: Hierarchical density based clustering. *Journal of Open Source Software*, 2(11):205, 2017.

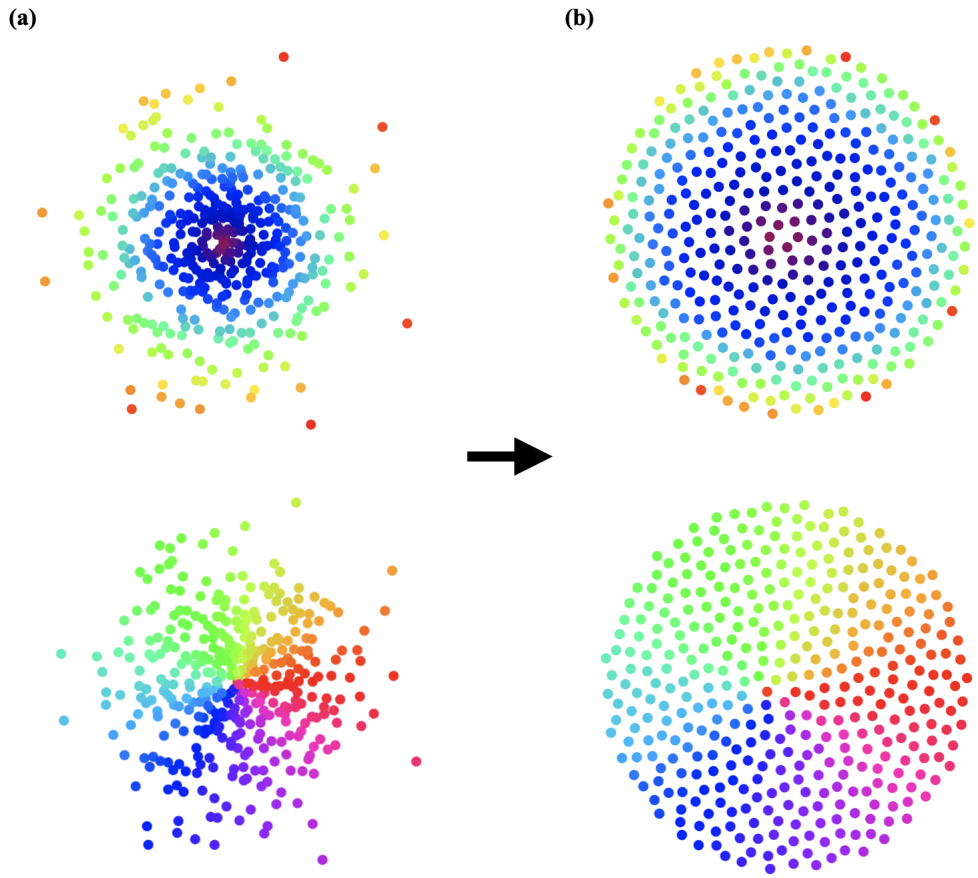


Figure 12: **(a)** MDS projected points, color coded by eccentricity (top), and polar angle (bottom). **(b)**Points with original color coding after smoothed by force field.

## 6 Supplementary Materials

### 6.1 Taylor Expansion Importance Score

### 6.2 Feature Visualization

### 6.3 Force Smoothing Layer Projections

### 6.4 Heirarchical Sparsity Regularization