VISCNN
Visualizing Interpretable Subgraphs in CNNs

Chris Hamblin

Introduction

Convolutional neural networks (CNNs) are the dominant model for a wide range of contemporary computer vision problems. Their success is largely attributable to computational advances that have allowed researchers to train deep models with millions of parameters, on large, diverse datasets, and 'end-to-end' objective functions. Unfortunately, these attributes make the inner workings of deep CNNs difficult to interpret. Most of the work in CNN interpretability, such as feature visualization[1] and attribution[2], concerns data *representation* by the model's internal features. Such techniques reveal *what* the model features encode, in the conceptual sense, by mapping internal activations back into human-perceivable pixel-space. Much less common are accounts of data *processing*[3], where the CNN is described as an interpretable sequence of computations that generate such representations. So, while a researcher might be quite confident a unit in their model acts as, say, a blue bird detector, they still don't know 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.

Olah et. al. 2020[4] outlines one promising data processing account of CNNs in which feature visualizations are viewed in conjunction with the convolutional kernels that connect them. They show how such kernels transform simple features into complex ones by acting as spatial templates for how the simple features should be arranged. *Fig. 1* (from their paper) shows how a feature encoding cars is generated from features encoding windows, car bodies, and wheels, by convolving such features with kernels that ensure an image patch has wheels at the bottom, car body in the middle, and windows at the top. They call these small graphs of features connected by convolutional kernels *circuits*.

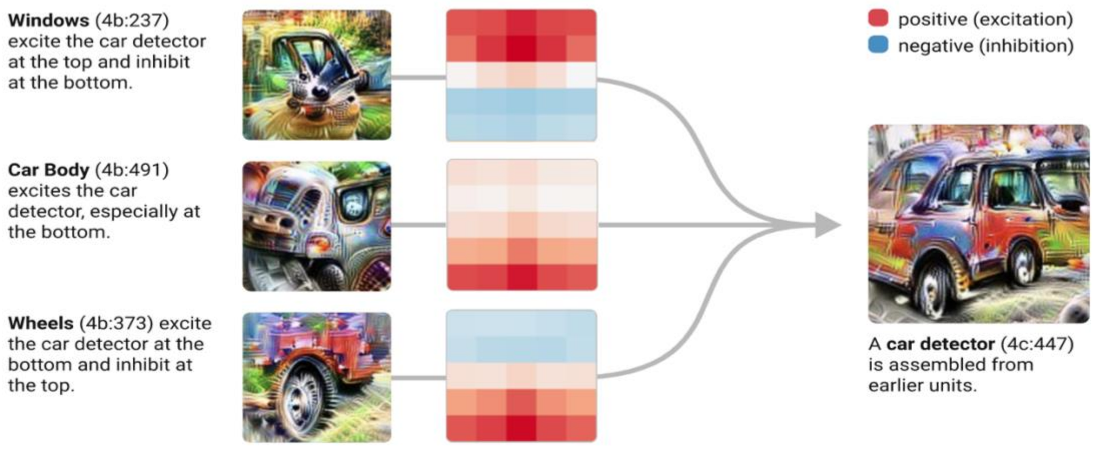

Windows (4b:237) excite the car detector at the top and inhibit at the bottom.
Car Body (4b:491) excites the car detector, especially at the bottom.
Wheels (4b:373) excite the car detector at the bottom and inhibit at the top.
🟥 positive (excitation)
🟦 negative (inhibition)
A **car detector** (4c:447) is assembled from earlier units.

Fig 1. ***'****Circuit*' for a car detector, from *Olah et. al. 2020*[4]

The authors describe many interesting circuits they've discovered in InceptionV1[5], from early, simple circuits encoding curves and spatial frequency, to late-layer circuits encoding whole objects. What is not clear is their process for making such discoveries; how do the authors hone-in on such fine-grained parts of a large network? More importantly, are they really justified in doing so? If we consider a feature to be associated with an individual convolutional filter in a CNN, features in contemporary models conventionally have hundreds of kernels leading directly into them. The three kernels in the preceding car example make for a clean, interpretable circuit, but what justifies ignoring the effects of all the other kernels feeding into the supposed 'car' feature? Considering a similar kernel selection must occur for defining the preceding wheel, car body, and window circuits, defining a full circuit from pixel space inputs to a late-layer feature like the car detector quickly leads to combinatorial explosion. What is needed is a principled, quantitative way of defining sub-circuits in a CNN that generate high-level features and objectives of interest. Furthermore, it would be useful to visualize these circuits within the context of the full neural network circuitry. Towards these ends I've developed **VISCNN**[*], a tool for **V**isualizing **I**nterpretable **S**ubgraphs of **CNN**s. Olah et. al. liken their analysis to microscopy, zooming in to a resolution at which CNN data processing is interpretable. In keeping with the metaphor, VISCNN might best be described as the tool that enables microscopy, a CNN microscope.

Defining Subgraphs

Following our previous reasoning, we can define the circuit underlying a CNN feature by first identifying kernels we are justified in *excluding* from that circuit. The circuit is then simply everything left-over, the parts of the network we cannot justifiably exclude. Under such a formulation, its sensible to draw from techniques developed for neural network pruning, where the goal is to identify components of a network that can be removed without significantly affecting overall task performance. Most pruning research concerns the development of quickly computable metrics that approximate parameter 'importance' in this respect, as removing parameters one at a time and checking the network's outputs is computationally intractable. Molchanov, P. et al. (2017)[6] is of particular interest for our purposes, as the paper evaluates metrics for pruning convolutional filters of CNNs. The most effective metric they evaluated approximates the network's loss given an intermediary activation in the network were removed (set to 0) by first-order taylor approximation. Such an approximation can be evaluated over the whole network with a single forward and backward pass, as it relies only on activation values and their gradients. The importance *I* of a particular activation h_i should be proportional to the change in loss *L* induced by setting that activation to 0.

$$I_{h_i} \propto |\Delta L| = |L(h_i) - L(h_i = 0)| \tag{1}$$

The first-order taylor approximation of *L* at $h_i = 0$ gives;

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

[*] https://github.com/chrishamblin7/viscnn

Plugging (2) into (1);

$$I_{h_i} \propto |L(h_i) - L(h_i = 0)| = \left| L(h_i) - (L(h_i) - \frac{\delta L}{\delta h_i} h_i) \right| = \left| \frac{\delta L}{\delta h_i} h_i \right| \quad (3)$$

We can apply this pruning metric to any intermediary activation map M_j in a CNN, such as the output of a filter or a single kernel, by simply averaging the importance scores for all activations in the map. Similarly, we can average a map's pruning importance score over different input images in a set *D* to get the map's importance for processing that particular image set. Supposing *m* is the length of vectorized map M_j, *d* is number of images in *D*, and h_i^k is ith activation in the map given input image *k;*

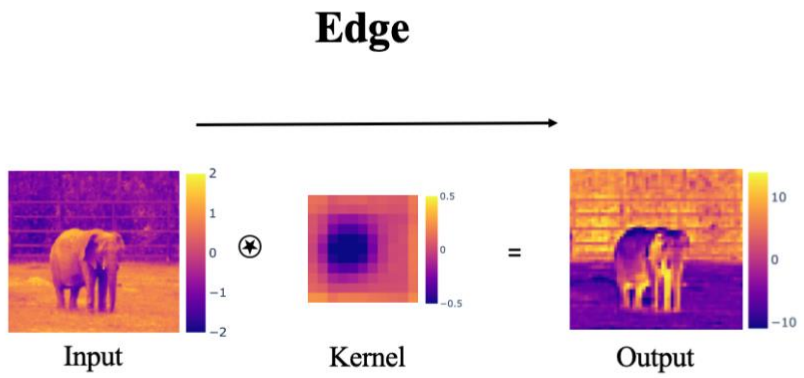
$$I_{M_j} \propto \frac{1}{dm} \sum_{k=1}^{d} \sum_{i=1}^{m} \left| \frac{\delta L}{\delta h_i^k} h_i^k \right| \quad (4)$$

Note that this taylor-approximation importance score is coherent with respect to any differentiable downstream scalar in the network, not just the final loss. We can use the exact same logic to determine how removing kernels and filters would affect some downstream feature's *F* expression over a set of input images. Simply replace *L* in equation (4) with *F*. The features we will be analyzing with VISCNN also correspond to the output activation maps of convolutional filters, but as they must be scalars to calculate a single derivative, we'll define a feature to be the mean of its corresponding activation map. Now for each feature in a CNN we can assign an importance score to each preceding activation map for any set of input images. We can exclude kernels and filters from a feature's circuit if their output activation maps have low importance scores for that feature. By setting different importance thresholds we can define a continuum of circuits from small to large for each feature, with circuit components 'weighted' by their importance scores.

VISCNN Graph Structure

Now that we have a quantitative definition of features' circuits, let's define a schema for visualizing CNNs and these embedded subcircuits. For now, we will only contend with the convolutional operations in the CNN, which we can treat as a unidirectional computational graph passing activation maps. Each edge in the graph will represent a single 2D convolutional kernel from the CNN (*fig 2.a*). Nodes will represent whole convolutional filters, which add together the output activation maps from a collection of edges (kernels) pixelwise, then apply an activation function (*fig 2.b*).

(a)



(b)

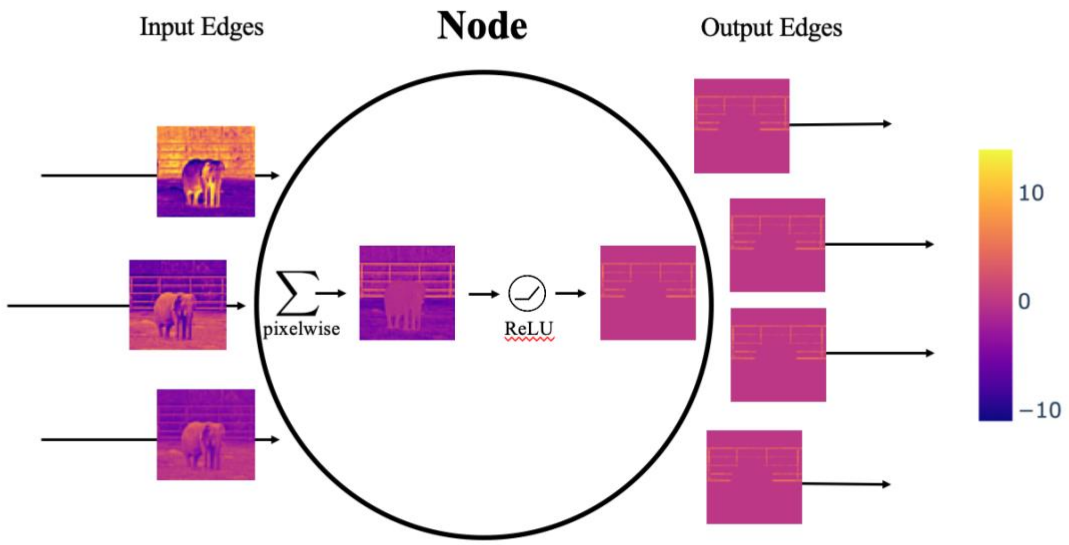


Fig 2. (a) *An example first-layer edge in the graph representation of Alexnet[6]. Here it takes the green channel map of an elephant picture as input, convolves its underlying suppressive 11x11 kernel, and outputs the resultant activation map.* ***(b)*** *An example first-layer node (filter) from graphical Alexnet. It takes the outputs from three edges (one for each color channel), sums the activation maps, and applies a ReLU, ultimately outputting an activation map that is only responsive to the red fence behind the elephant. A subset of edges from layer 2 then take this activation map as input.*

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 the final version of VISCNN will give the user several options under the 'projection' hyperparameter. For the subsequent Alexnet[7] visualization examples in this paper I use the 'Class MDS' projection. 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[8] 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 Multi-Dimensional Scaling. 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 can L2 normalize each node's 1000-dimensional importance vector to length 1 before running MDS.

Putting all these pieces together we get the VISCNN tool (*fig. 3*), which allows researchers to explore the minimal circuits embedded in deep neural networks that generate their rich perceptual features. VISCNN can take an arbitrary feed-forward CNN (with some architectural constraints, will discuss in *future work* section) specified in Pytorch and project the model's convolutional operations into the previously described graph structure. The user can then query the model through its GUI for weighted circuits by specifying a target feature (or loss) the circuit should preserve, an input image or image-set to preserve over, importance metric (there are some other options in addition to the taylor metric), and an importance threshold for circuit inclusion. Finally, the user can pass arbitrary input images through the model and explore its intermediary computations with a point-and-click interface on the nodes and edges, revealing the underlying kernels, feature visualizations, and output activation maps.

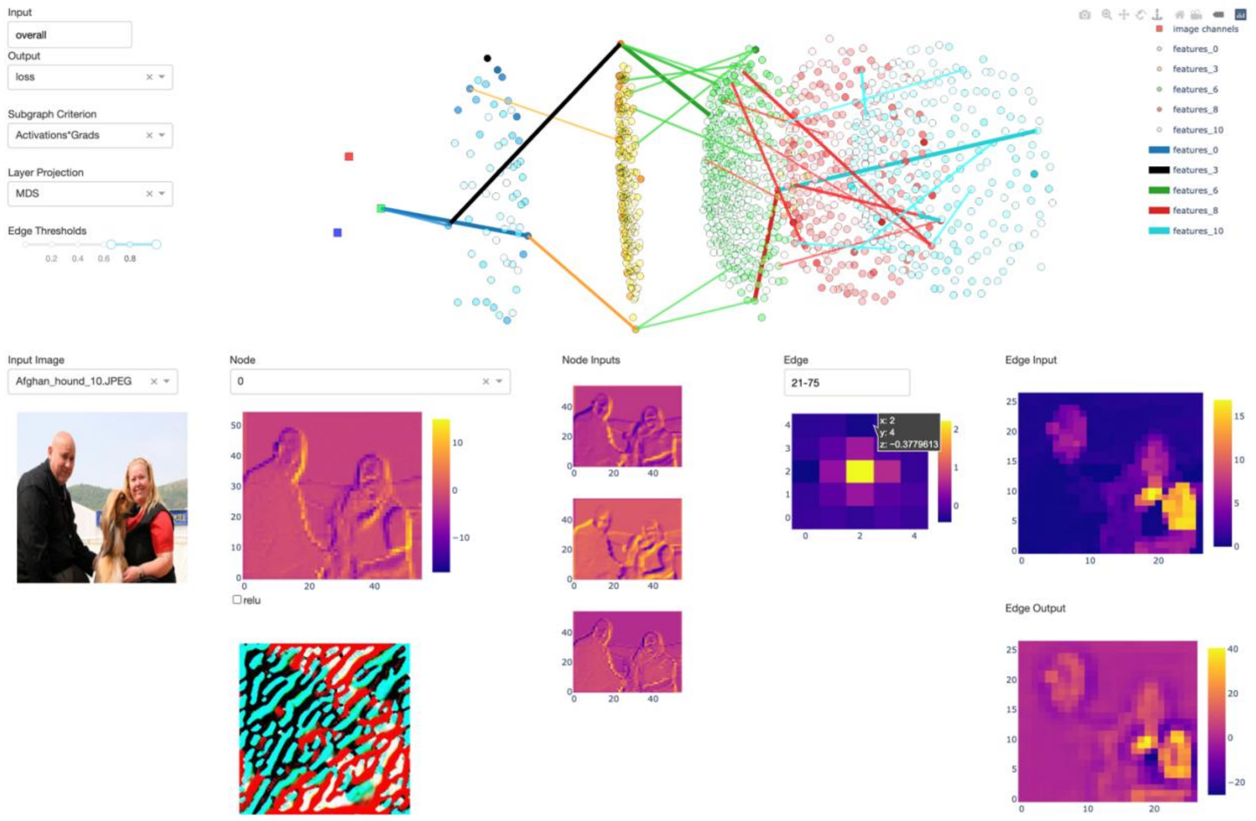
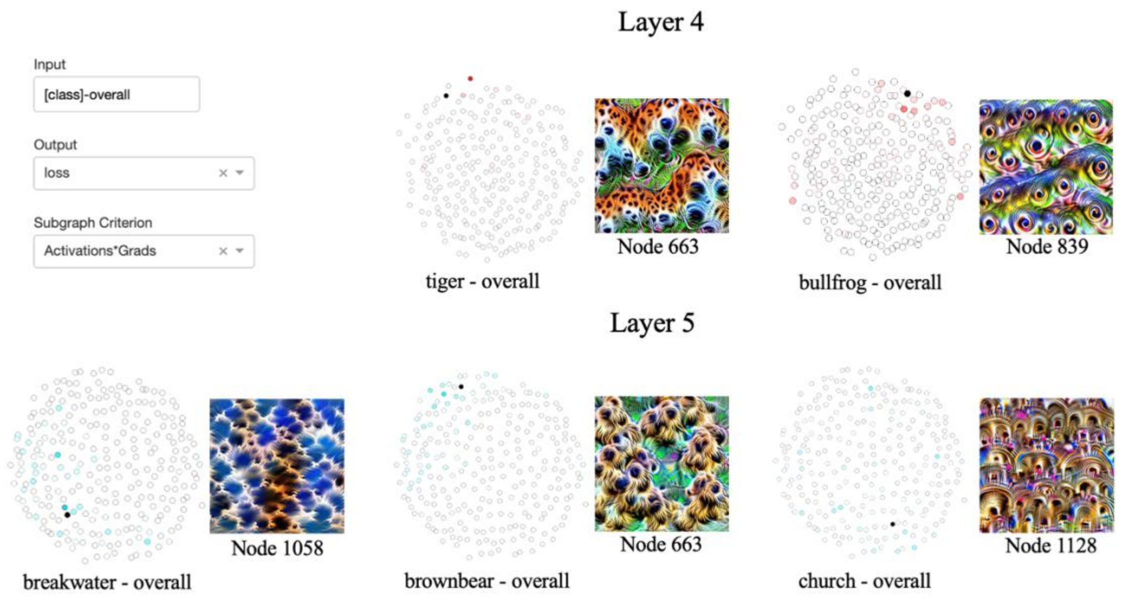

Input: overall
Output: loss
Subgraph Criterion: Activations*Grads
Layer Projection: MDS
Edge Thresholds: 0.2 0.4 0.6 0.8
Input Image: Afghan_hound_10.JPEG
Node: 0
Edge: 21-75
x: 2
y: 4
z: -0.3779613

Fig. 3 Screenshot from VISCNN. The user is visualizing the most general (and not very informative) circuit, preserving final loss over all images in the dataset. The black node and edge have been selected (shown in bottom half). Try it for yourself at https://github.com/chrishamblin7/viscnn

Example Experiments

VISCNN can be used in many different ways as it's primarily an exploratory tool, but here I'll demonstrate a general pipeline by which VISCNN can facilitate the circuit analysis found in Olah et. al. 2020 as applied to Alexnet. I like to start by selecting late layer nodes based on their importance for classify different classes in the image set. VISCNN can take in linear combinations of input image-sets for determining importance scores, so I like to select nodes based on their class importance – their overall importance (*fig 4)*. This selects nodes that are important for a particular class over and above their general importance, and usually selects for a small set of nodes per layer.

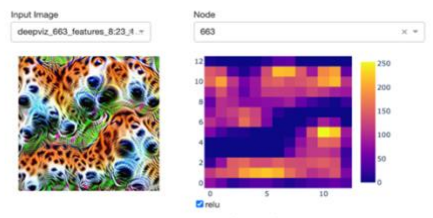

Input: [class]-overall

Output: loss

Subgraph Criterion: Activations*Grads

Fig. 4 *Nodes that are important for classify particular image classes are activated by synthesized images that are very reminiscent of those classes themselves. Here we can make out the fur of tigers and brownbears, the eyes of frogs, the arches of churches, and the waves of breakwater. In these example figures, important GUI hyperparameter settings are shown on the left.*

Node 663, a *'tiger'* node, has a particularly rich feature visualization, suggesting a tiger's orange fur and white underbelly against a green forest. We can validate it's a good feature visualization by feeding the synthesized image into the model as input and checking that node 663's output activation map is highly responsive *(fig 5)*. The synthesized image definitely excites node 663, driving activation values in the map over 200. On non-tiger natural images, unit activations in this map rarely exceed 4. However, we can also see that the green regions don't activate this map at all, so we can safely amend our original hypothesis that this node responds to tigers against a green background. We can simply ignore the green regions as areas where the visualizer failed to converge.

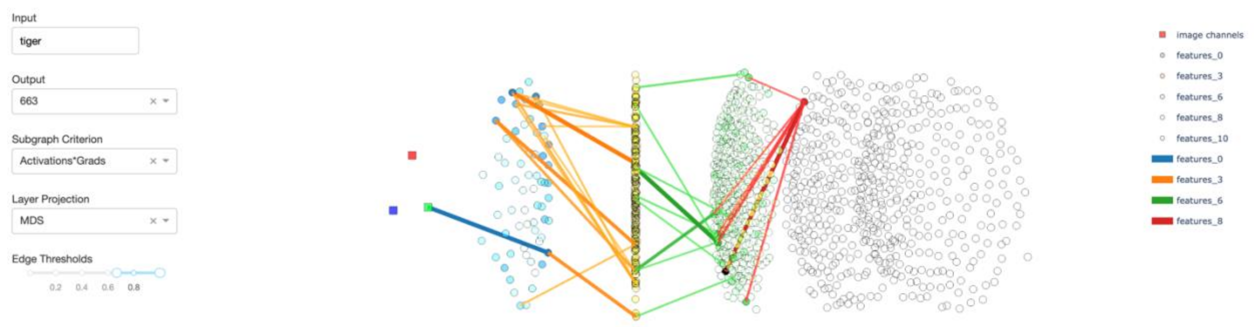

Input Image: deepviz_663_features_8:23_4

Node: 663

relu ☑

Fig 5. *Activation of node 663*

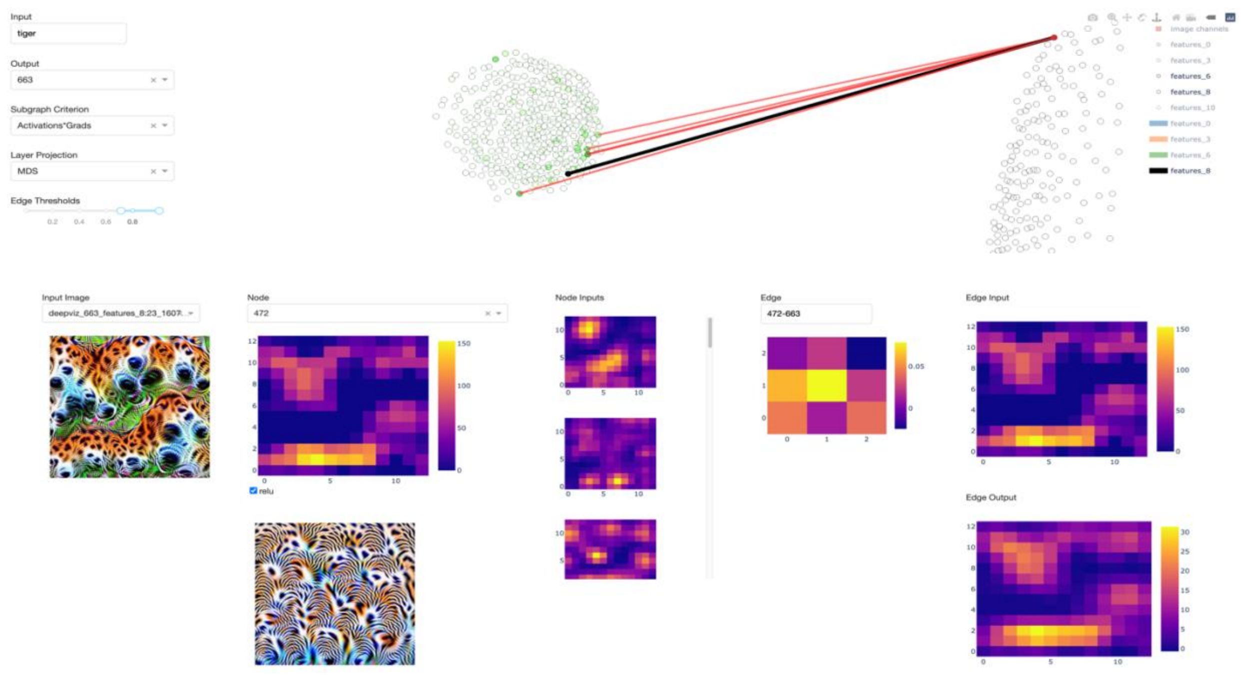
We can now visualize the circuit generating node/feature 663 by changing the ‘output’ hyperparameter from ‘loss’ to ‘663’ *(fig 6):*



Input: tiger
Output: 663
Subgraph Criterion: Activations*Grads
Layer Projection: MDS
Edge Thresholds

Fig 6. *Sample circuit that preserves node 633 responses to the set of tiger images*

This is a bit too complex to follow, so let’s focus in on the last two layers of the circuit for our first pass at a kernel analysis *(fig 7).*



Input: tiger
Output: 663
Subgraph Criterion: Activations*Grads
Layer Projection: MDS
Edge Thresholds

Input Image: deepviz_663_features_8:23_1607...
Node: 472
Edge: 472-663

Fig 7. *Visualizing the most important node in the previous layer for our target tiger feature**,** we discover a node sensitive to black stripes on an orange white background, a component of the textures in our original feature visualization. Its also very responsive to the original visualization. The kernel connecting this feature to our target node would prefer this feature be in the center or bottom portion of the image patch, perhaps because white tiger fur is generally below the orange.*

Supposing we develop agrasp of this circuit between layer 3 and node 663, we can then take a step back and observe the circuit from layer 2 to an important node in layer 3, in a new effort to understand what *it* encodes *(fig 8)*.

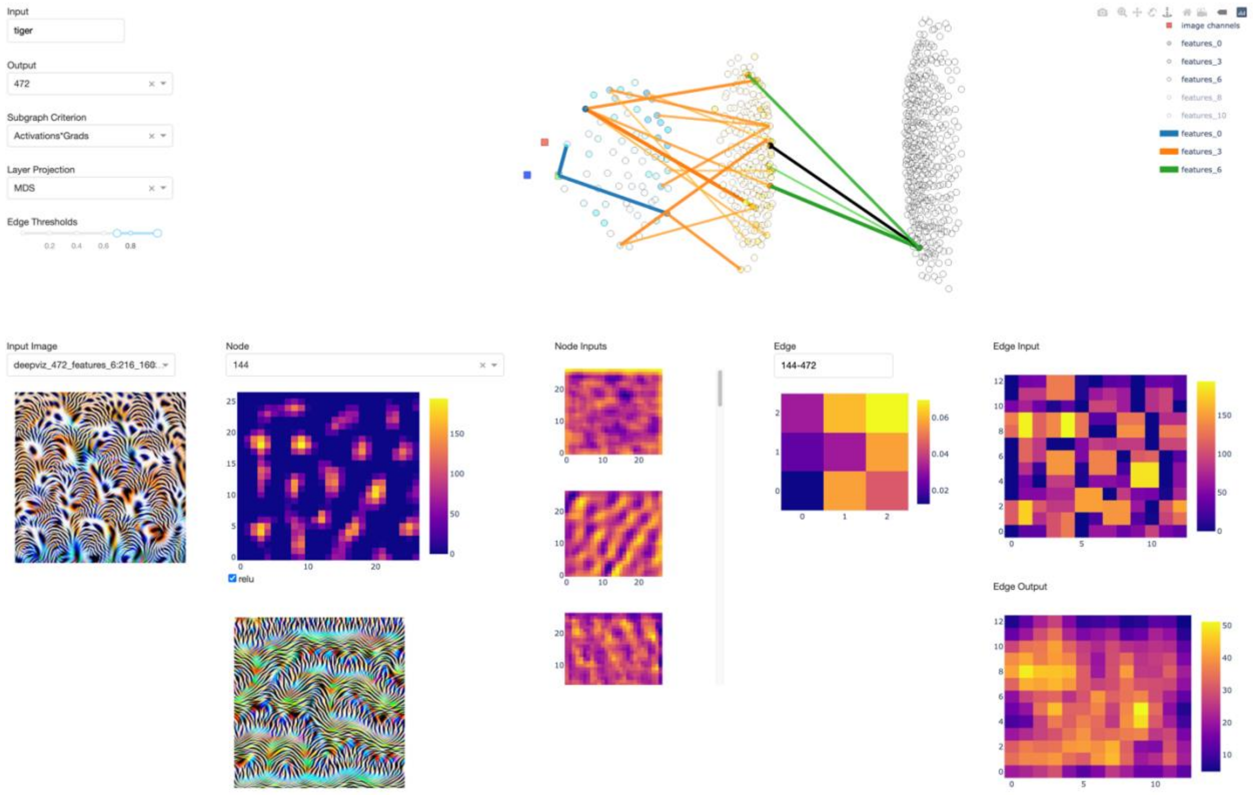

Input
tiger
Output
472
Subgraph Criterion
Activations*Grads
Layer Projection
MDS
Edge Thresholds
Input Image
deepviz_472_features_6:216_160:
Node
144
relu
Node Inputs
Edge
144-472
Edge Input
Edge Output

Fig 8. *The circuit underlying our previously identified black on white/orange stripe node, and visualizations of an important node and edge leading to it.*

Such analyses get very complicated very quickly, and getting a handle on VISCNNs strengths, weaknesses, and quirks is going to take a lot of exploration. For better or worse, it’s clear that the space for such exploration is quite large.

Future Work

This paper describes the motivation, design principles, and mathematical techniques of the VISCNN tool. Future work primarily revolves around putting VISCNN to work, generating insights into the data processing principles latent in CNN circuits. Having started down this analysis path, I’ve determined these three additional features would be highly useful for the tool.

1. *Feature Visualizations for Edges*

As features are visualized with respect to output activation maps, it should be possible to generate feature visualizations associated with the edges in our graph. Such visualizations would allow the researcher to see how a single kernel application transforms a visualization, as well as how visualizations are summed together within nodes. I've implemented code to fetch these visualizations in the same manner as the visualizations for nodes, but unfortunately the visualizer does not converge, generating noisy images that do not activate the edge's output activation map *(fig 9)*. I believe this to be an implementation bug rather than suggestive of any special properties of edge features. Pytorch does not natively support accessing the activation maps output by edges, as such computational intermediaries are buried within the C+ implementation of Pytorch's nn.Conv2d module. Dissecting the Conv2d module to view these activation maps was perhaps the hairiest software challenge in writing VISCNN, so I am not entirely surprised a bug has emerged when generating feature visualizations with respect to these maps.

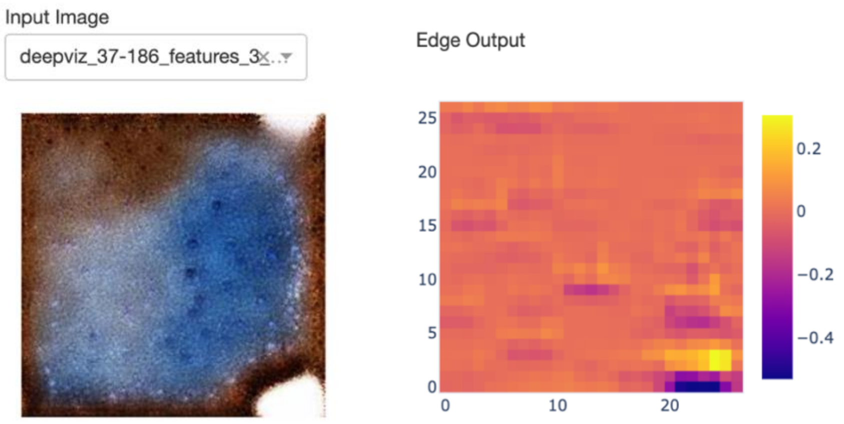


Fig 9. *Dud feature visualization for edge between nodes 37 and 187.*

2. *Receptive Field of Activation Map Units*

Most CNN architectures include pooling and convolution operations that diminish the resolution of activation maps as one traverses deeper into the network. Our analysis often requires comparing images in pixel space to these downstream maps, and it can sometimes be difficult to determine which discrete region of an input image corresponds to a particular pixel in an activation map. It would make for a speedier and more rigorous analysis if hovering over a pixel in an activation map automatically drew a receptive field bounding box over the corresponding part of the input image.

3. *Model Agnostic*

While VISCNN is flexible with respect to the convolutional models it visualizes, there are some architectural features it can't currently handle, such as skip connections and branching. This wouldn't be a big problem if weren't for the fact that the InceptionV1[5] architecture involves branching connections, as InceptionV1 is the most popular architecture for feature visualization and circuit research. Feature visualization of filters within InceptionV1 leads to much richer images than any other model *(fig 10)*. This is likely due to the depth of its convolutional layers, and relatively small fully-connected classifier. Additionally, InceptionV1 uses 1x1 convolutional kernels to linearly combine filters, creating layers with far fewer filters than other architectures. These sparse filters bear a larger representational load, perhaps explaining their rich visualizations. Additionally, having layers with a minimal number of filters (nodes) is an architectural way of ameliorating the kernel selection problem when defining circuits, one of the central problems that motivated the VISCNN tool in the first place.

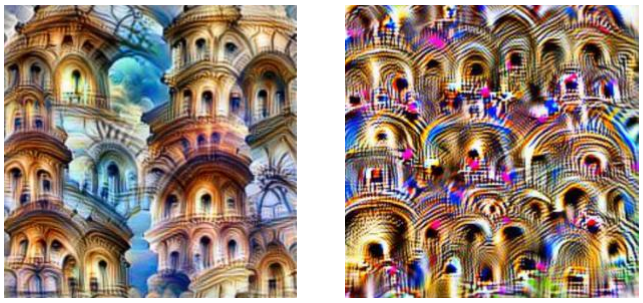
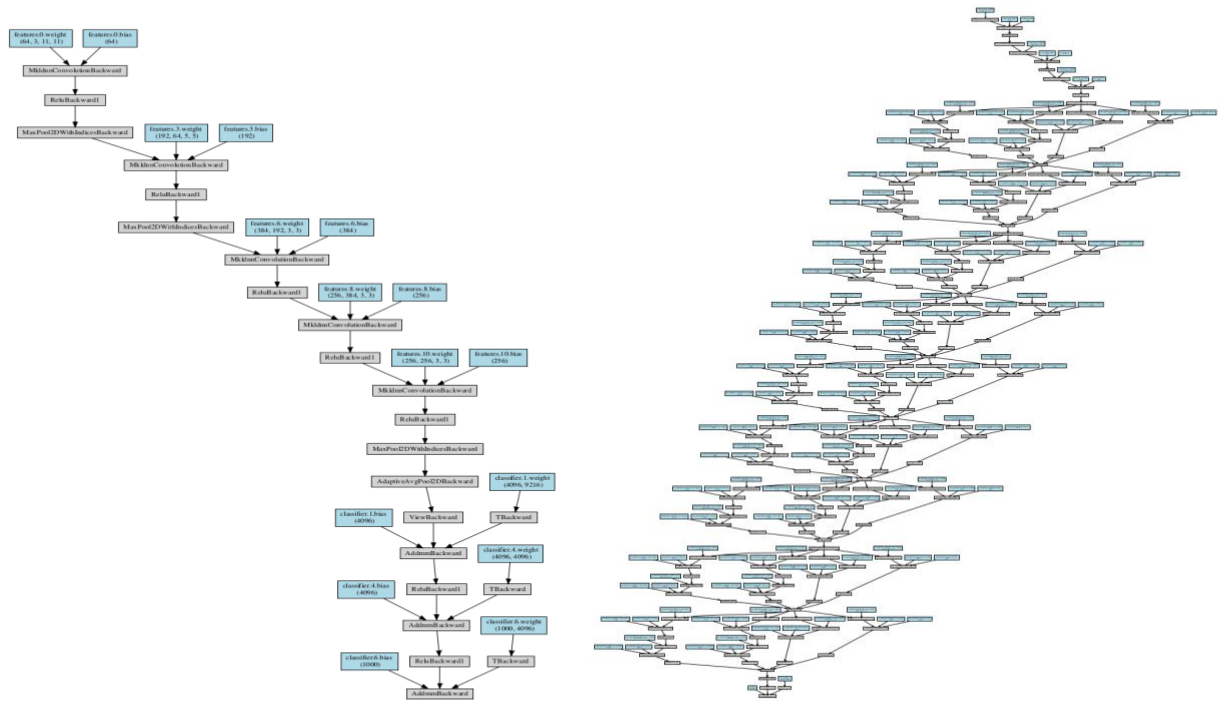


Fig 10. *'Arch' feature in InceptionV1 (left) and Alexnet (right)*

One possible avenue towards making VISCNN model agnostic is to find a way to translate between the PytorchViz[*] package and VISCNN. PytorchViz uses the autograd trace to draw a graph of the execution trace of arbitrary Pytorch models *(fig 11)*. Using a similar technique to define VISCNN graphs would allow the end user to seamlessly visualize any CNN model specified in Pytorch.


features.0.weight (64, 3, 11, 11)
features.0.bias (64)
MkldnnConvolutionBackward
ReluBackward1
MaxPool2DWithIndicesBackward
features.3.weight (192, 64, 5, 5)
features.3.bias (192)
MkldnnConvolutionBackward
ReluBackward1
MaxPool2DWithIndicesBackward
features.6.weight (384, 192, 3, 3)
features.6.bias (384)
MkldnnConvolutionBackward
ReluBackward1
features.8.weight (256, 384, 3, 3)
features.8.bias (256)
MkldnnConvolutionBackward
ReluBackward1
features.10.weight (256, 256, 3, 3)
features.10.bias (256)
MkldnnConvolutionBackward
ReluBackward1
MaxPool2DWithIndicesBackward
AdaptiveAvgPool2DBackward
classifier.1.weight (4096, 9216)
classifier.1.bias (4096)
ViewBackward
TBackward
AddmmBackward
classifier.4.weight (4096, 4096)
classifier.4.bias (4096)
ReluBackward1
TBackward
AddmmBackward
classifier.6.weight (1000, 4096)
classifier.6.bias (1000)
ReluBackward1
TBackward
AddmmBackward
Fig 11. *PytorchViz execution graphs for Alexnet (left) and InceptionV1 (right)*

[*] https://github.com/szagoruyko/pytorchviz

References

[1] Olah, C., Mordvintsev, A. & Schubert, L. (2017) Feature visualization. Distill
[2] Sattarzahed, S. et al. (2020) Explaining convolutional neural networks through attribution-based input sampling and block-wise feature aggregation. eprint arXiv:2010.00672
[3] **]** Gilpin, L. et al. (2018) Explaining explanations: an overview of interpretability of machine learning. Institute of Electrical and Electronics Engineers (IEEE)
[4] Olah, C. et al. (2020) Zoom in: An introduction to circuits. Distill
[5] Szegedy, C. et al. (2014) Going deeper with convolutions. (ILSVRC)
[6] Molchanov, P. et al. (2017) Pruning convolutional neural networks for resource efficient inference. International Conference on Learning Representations (ICLR)
[7]Krizhevsky, A., Sutskever, I. & Hinton, G. (2012) ImageNet classification with deep convolutional neural networks. NeurIPs
[8] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., & Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. IEEE conference on computer vision and pattern recognition