Deep learning

9.1. Looking at parameters

François Fleuret

https://fleuret.org/dlc/

UNIVERSITÉ
DE GENÈVE

Understanding what is happening in a deep architectures after training is complex and the tools we have at our disposal are limited.

In the case of convolutional feed-forward networks, we can look at

- the network's parameters, filters as images,
- internal activations on a single sample as images,
- derivatives of the response(s) w.r.t. the input,
- maximum-response synthetic samples,
- adversarial samples.

We can also look at distributions of activations on a population of samples at different stages in a model.

Hidden units of a perceptron

Given a one-hidden layer fully connected network $\mathbb{R}^2 \to \mathbb{R}^2$

```
nb_hidden = 20

model = nn.Sequential(
    nn.Linear(2, nb_hidden),
    nn.ReLU(),
    nn.Linear(nb_hidden, 2)
)
```

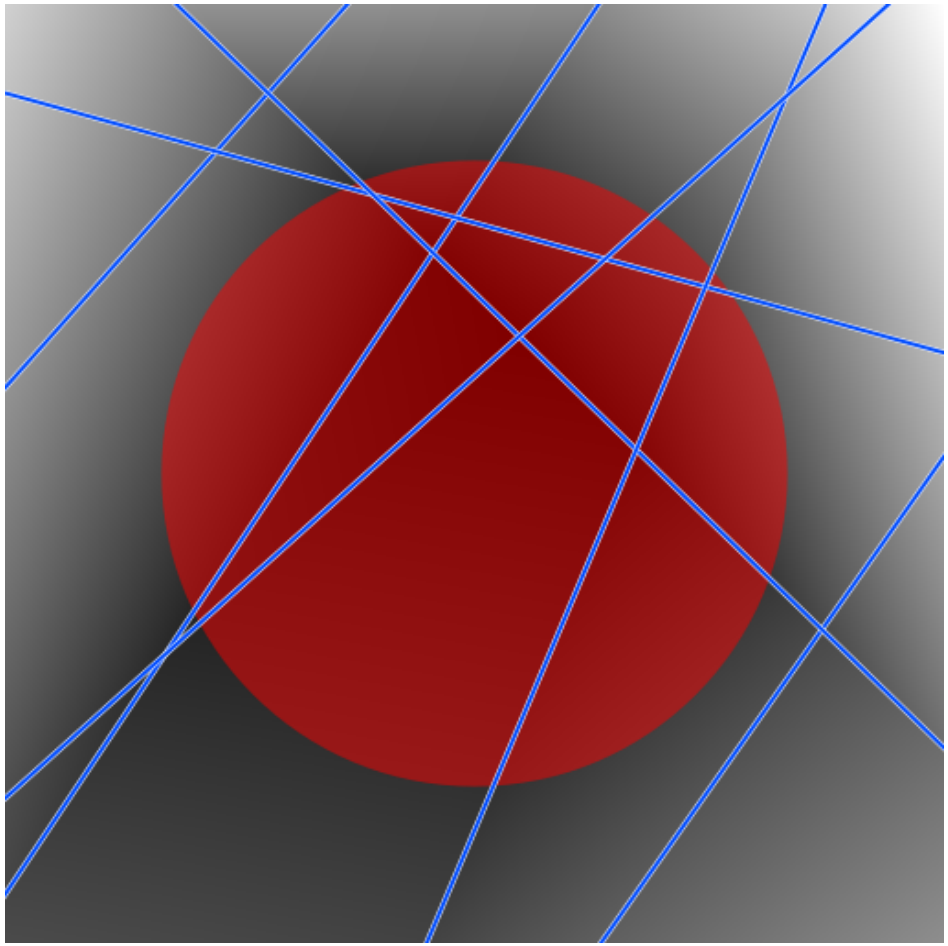
we can visit the parameters (w, b) of each hidden units with

```
for k in range(model[0].weight.size(0)):
    w = model[0].weight[k]
    b = model[0].bias[k]
```

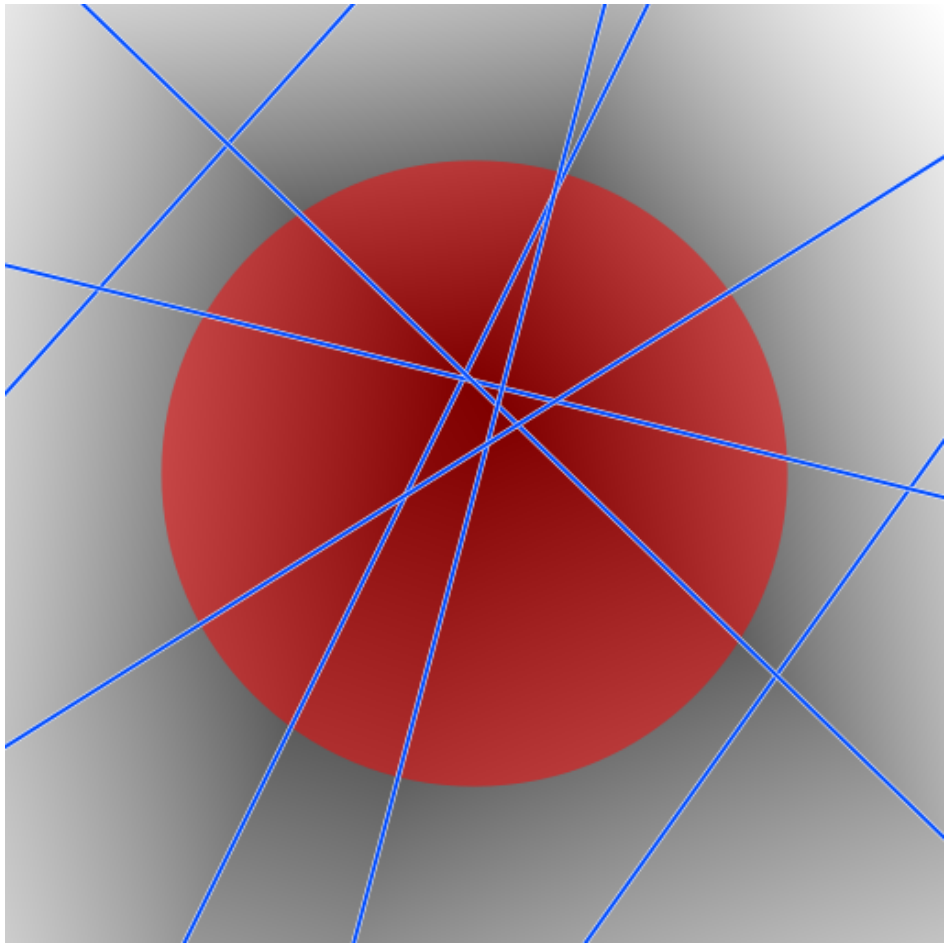
and draw for each the line

$$\{ x : w \cdot x + b = 0 \}.$$

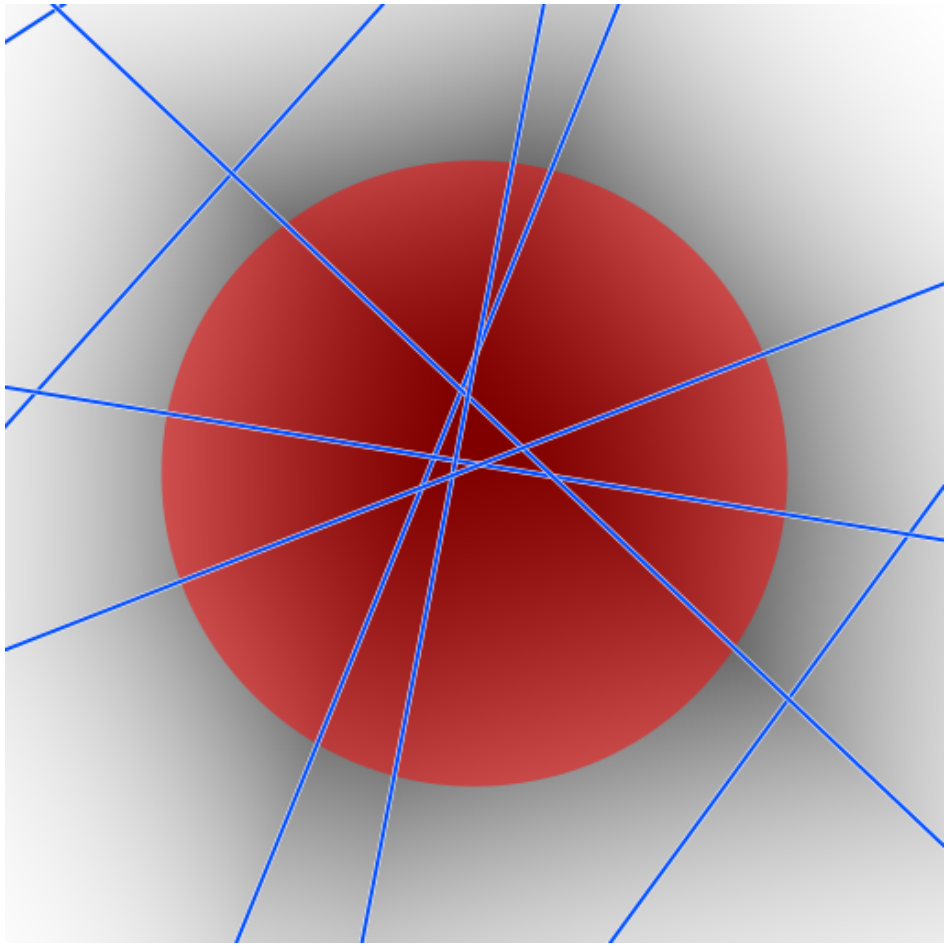
During training, these separations get organized so that their combination partitions properly the signal space.



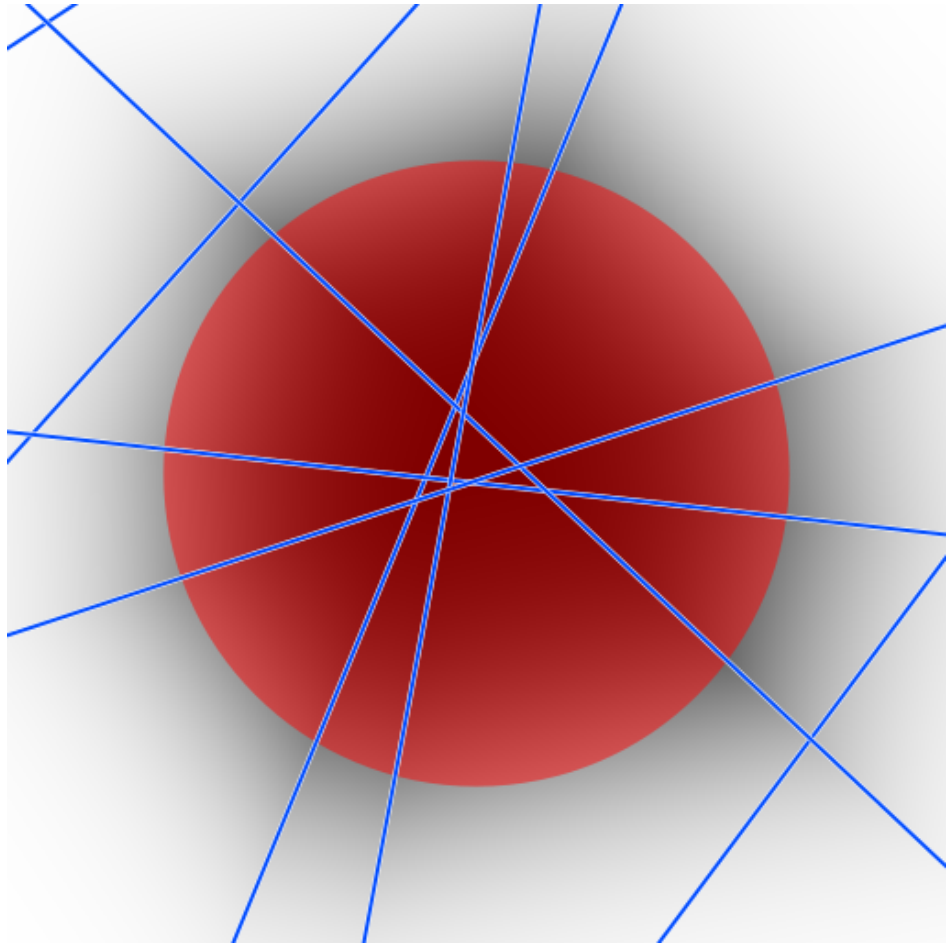
Iteration 1



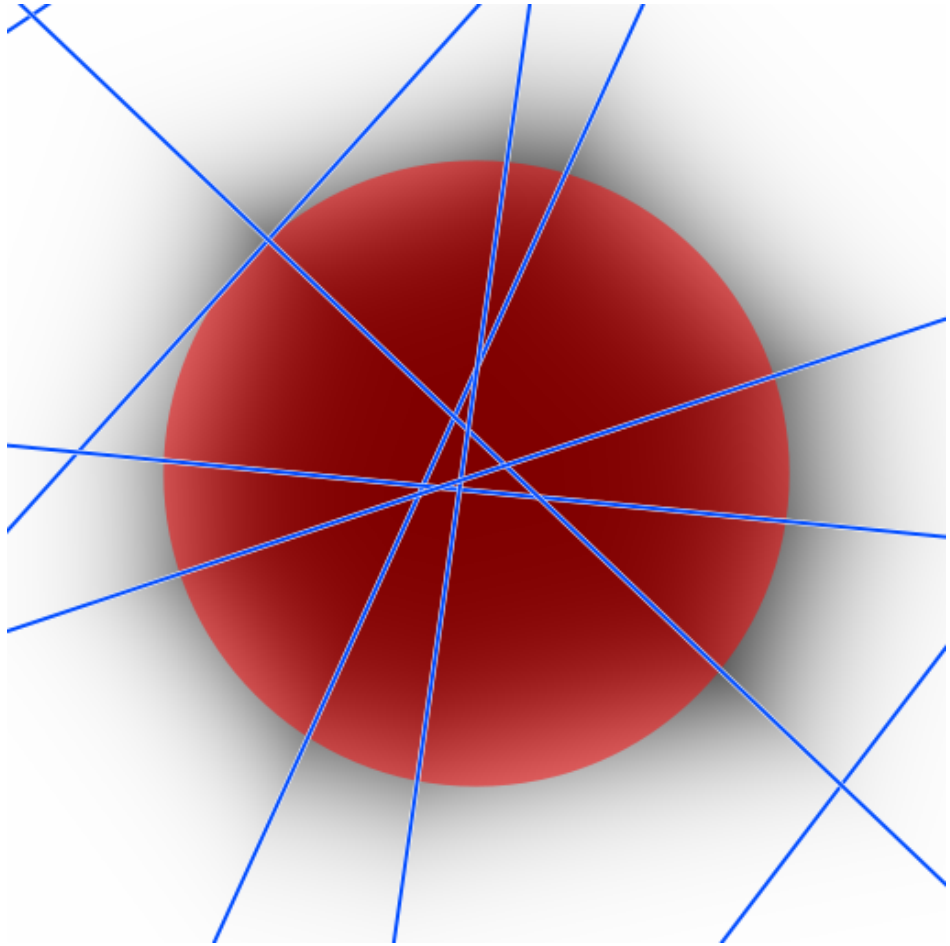
Iteration 4



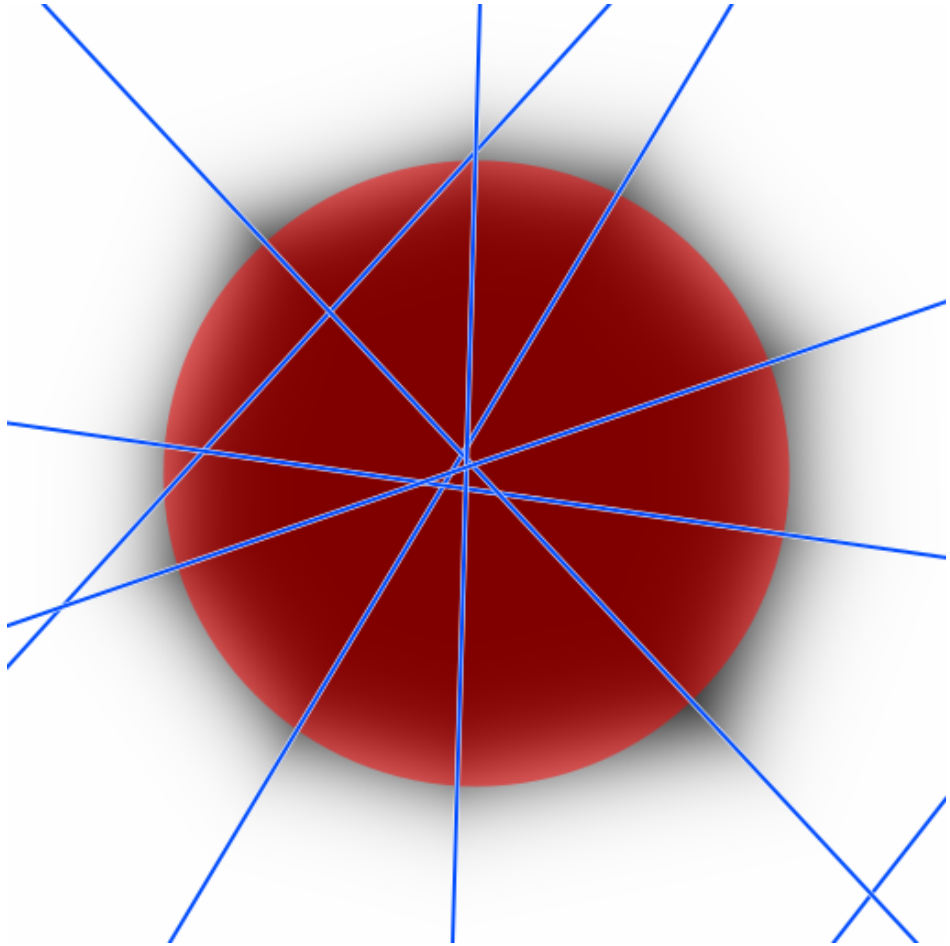
Iteration 7



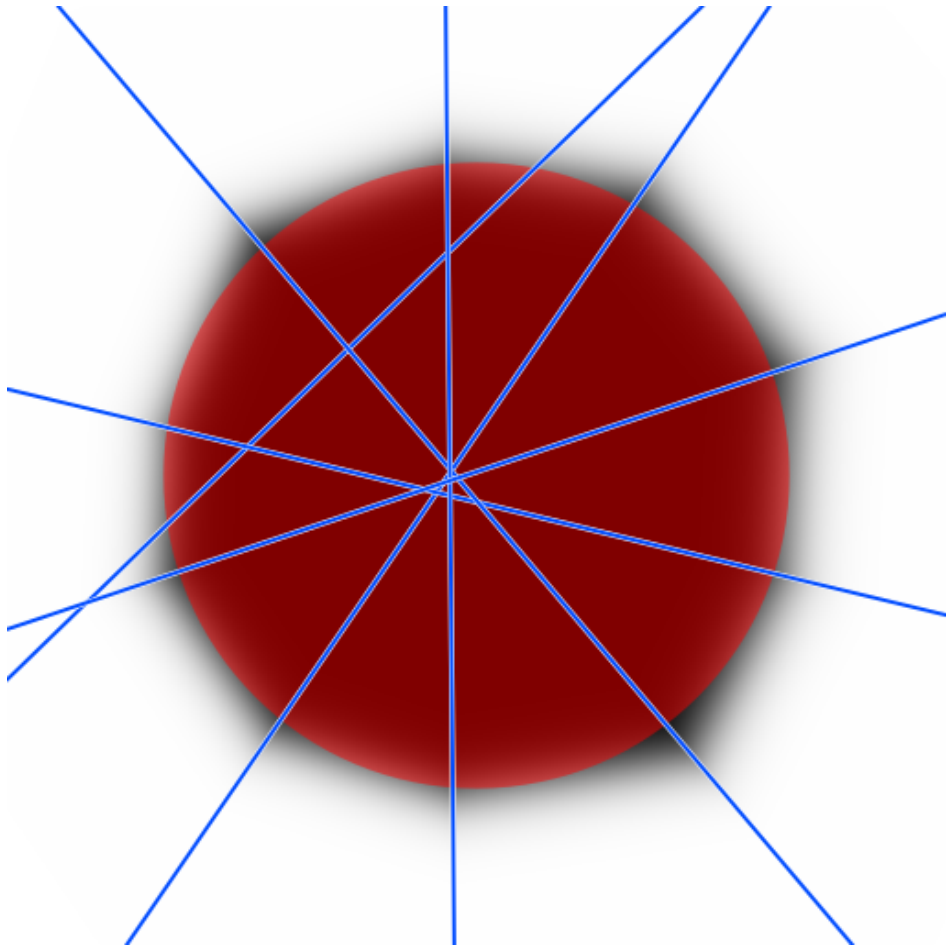
Iteration 10



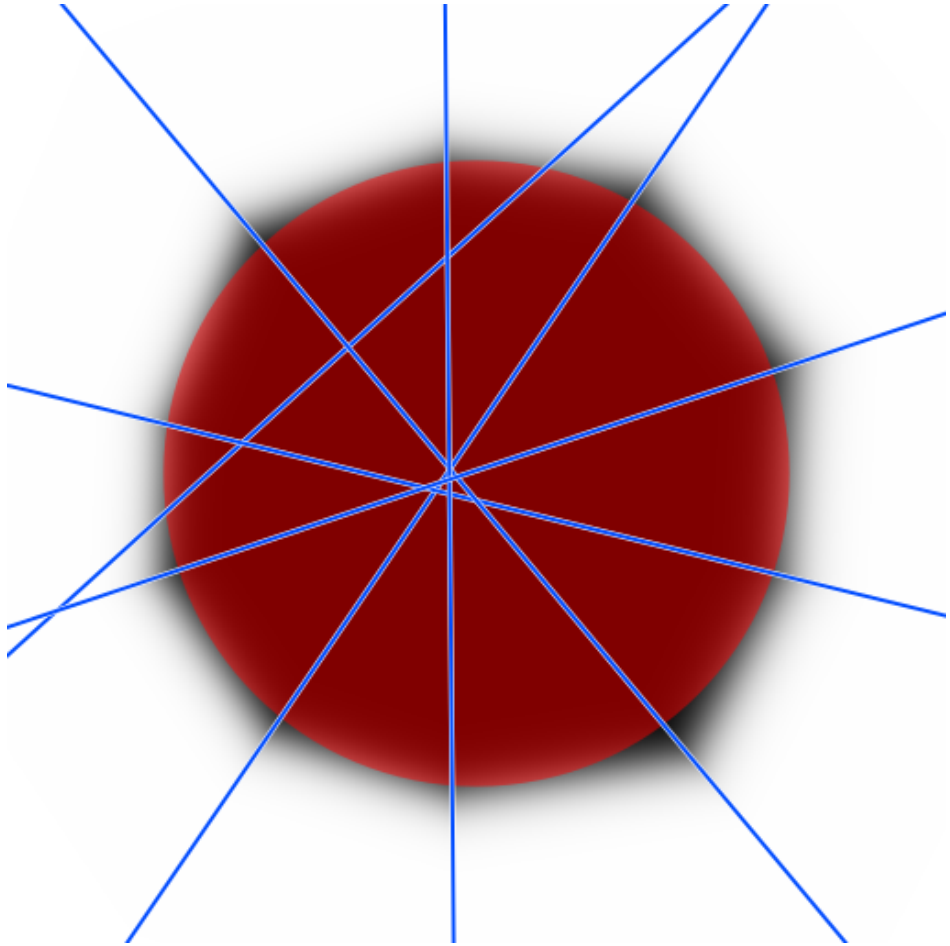
Iteration 16



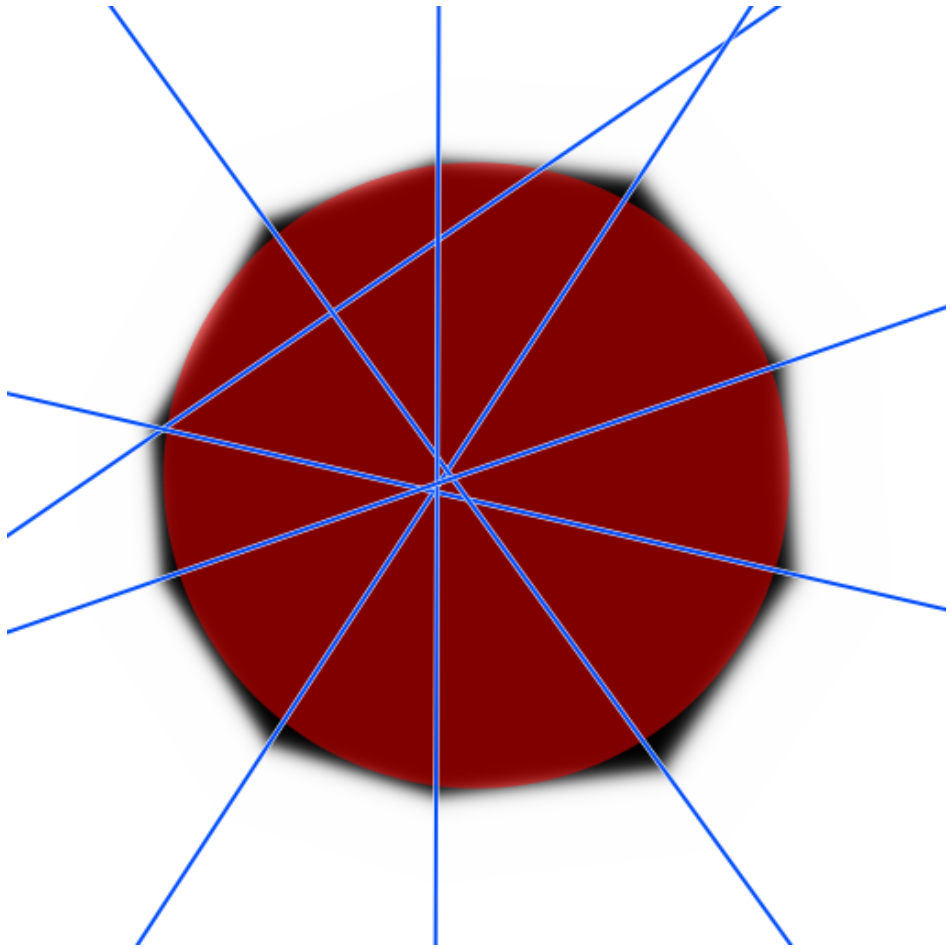
Iteration 34



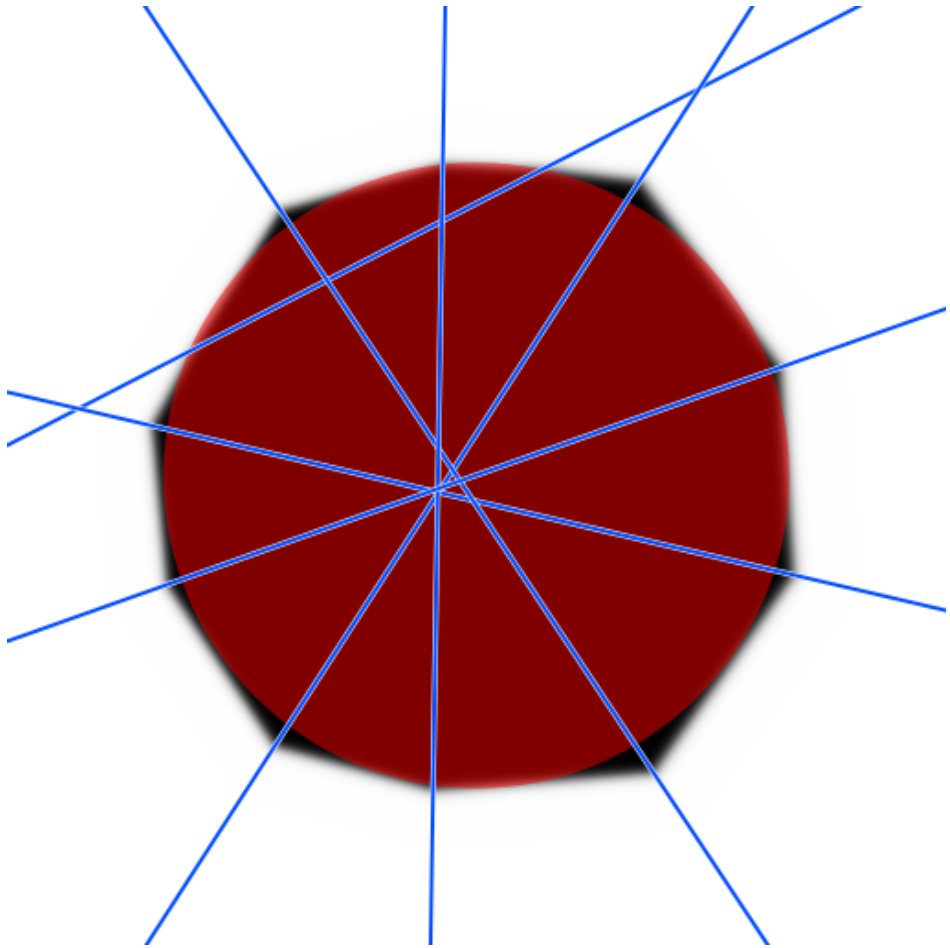
Iteration 77



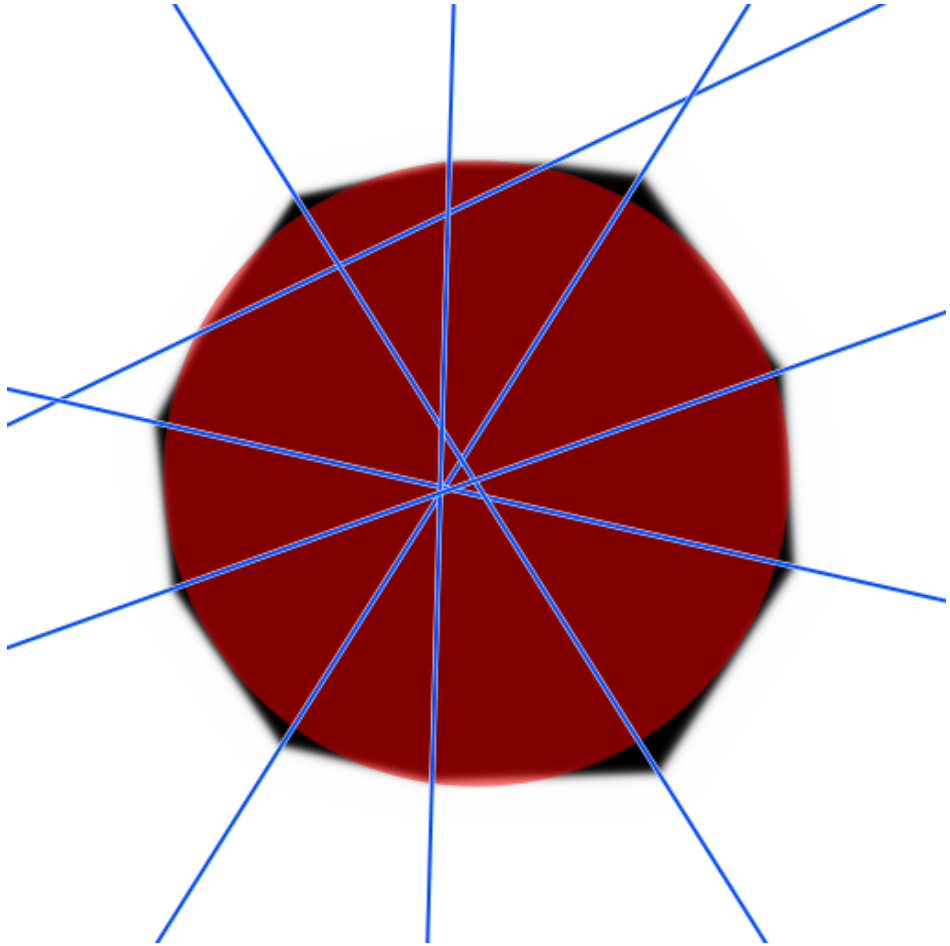
Iteration 100



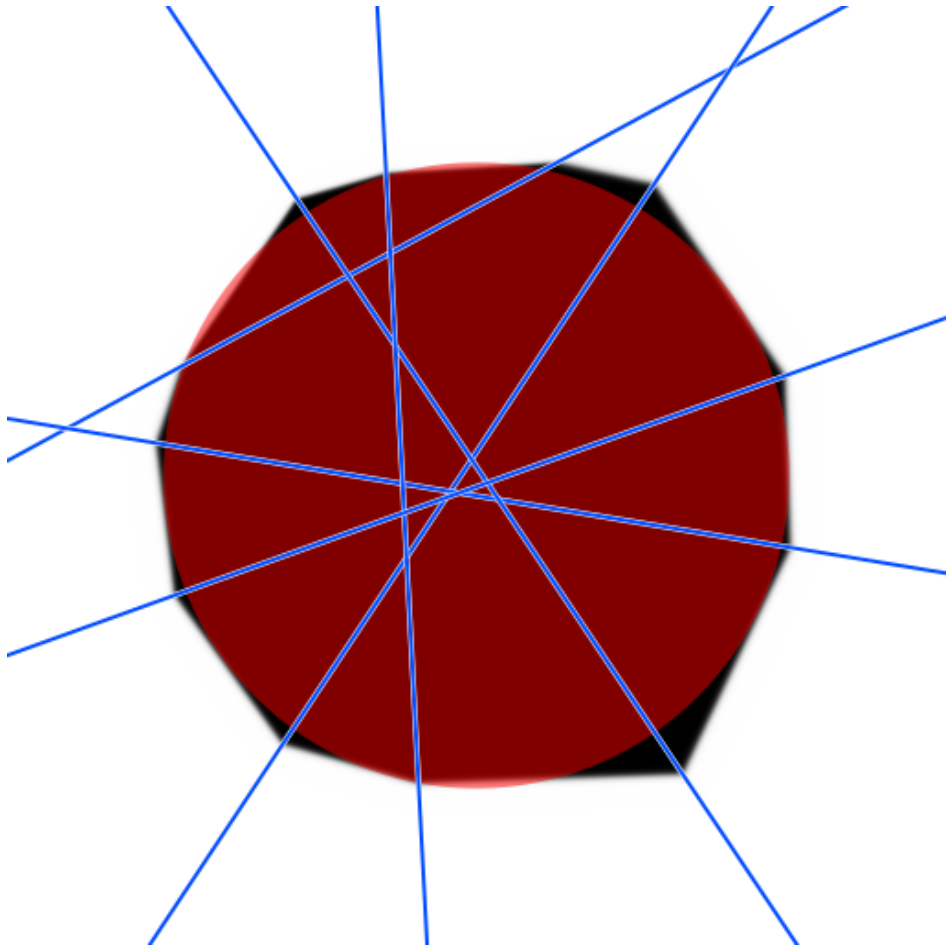
Iteration 703



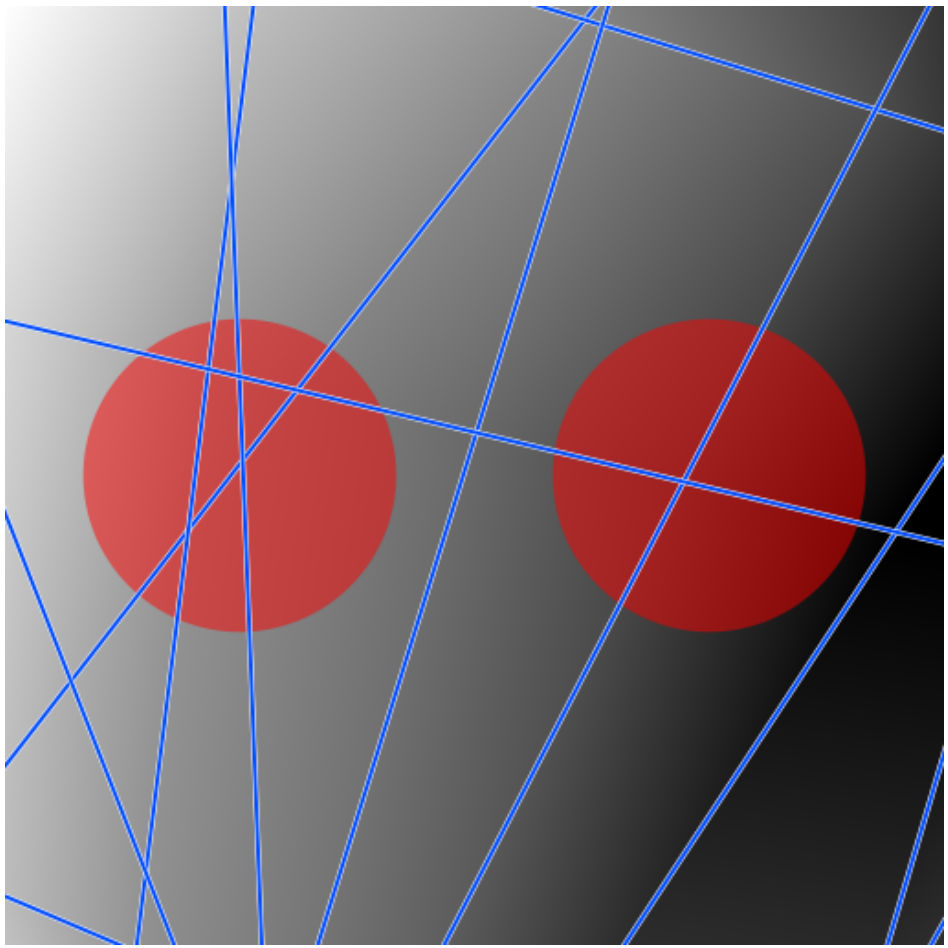
Iteration 1407



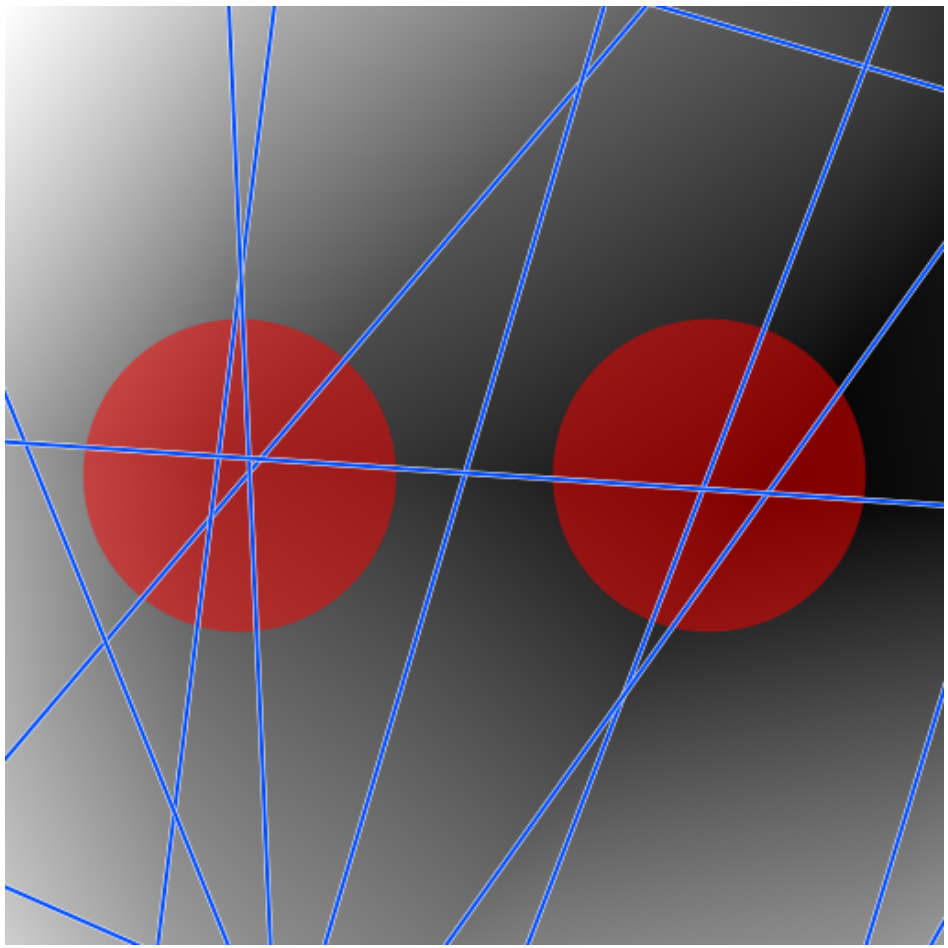
Iteration 2789



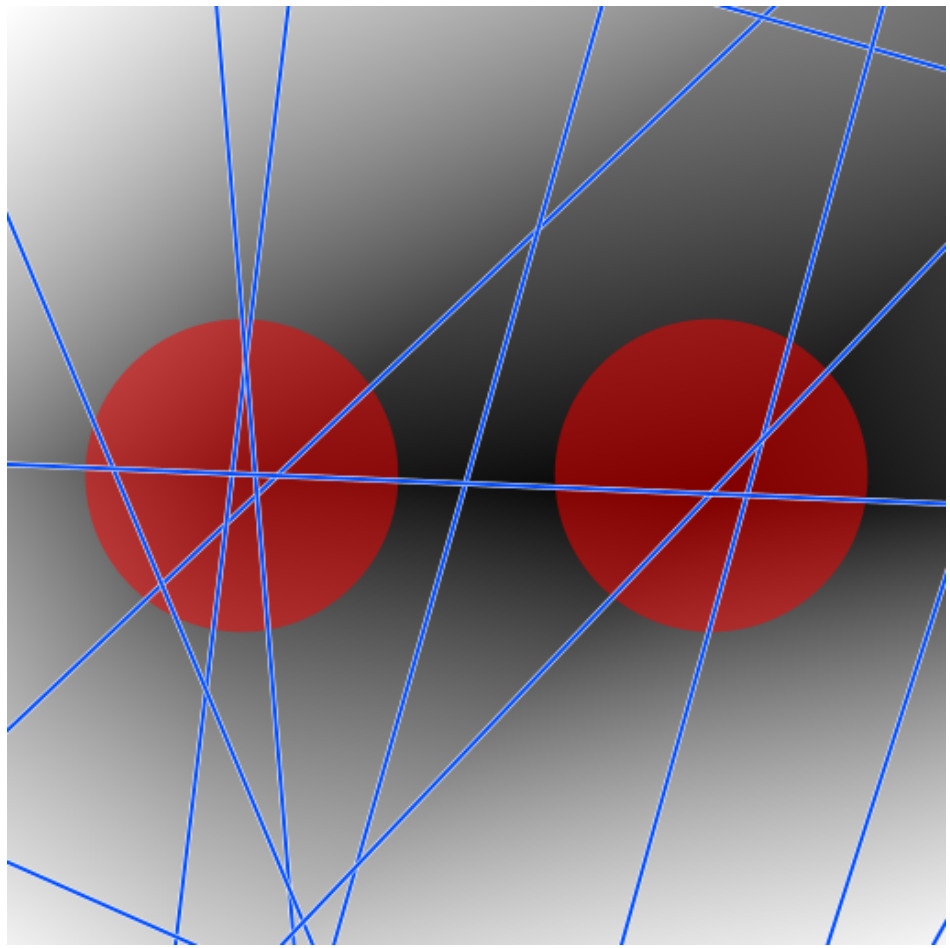
Iteration 9999



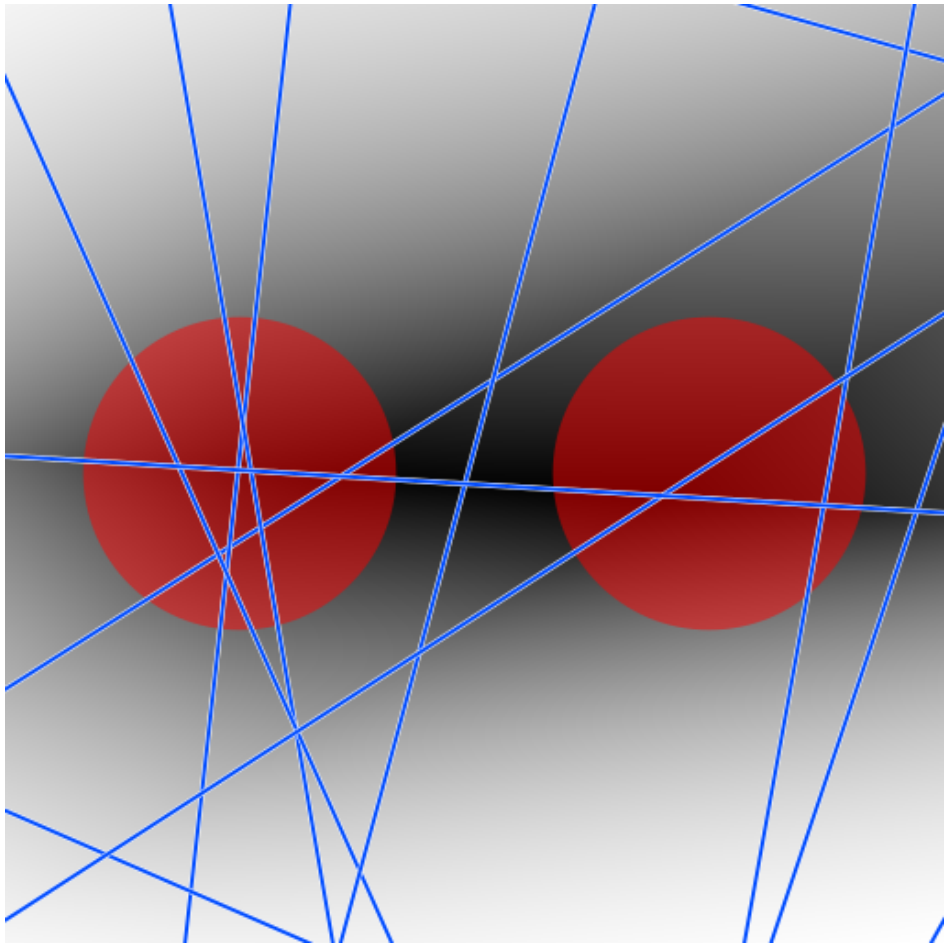
Iteration 1



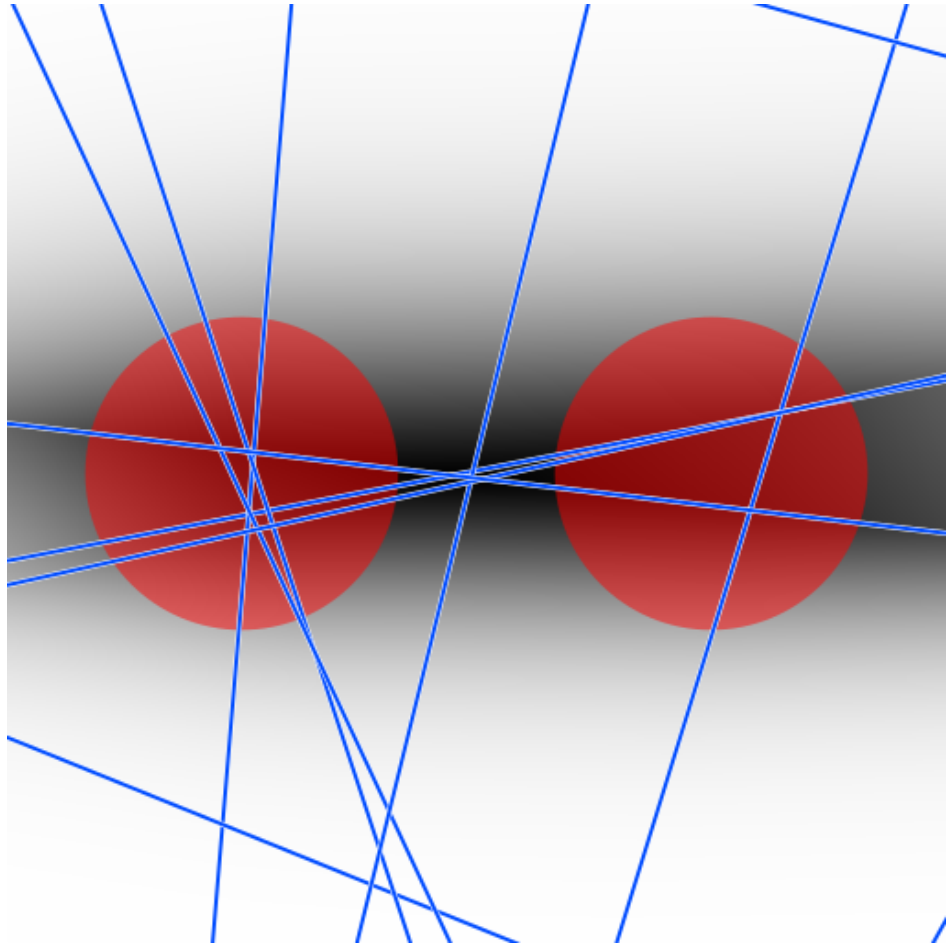
Iteration 4



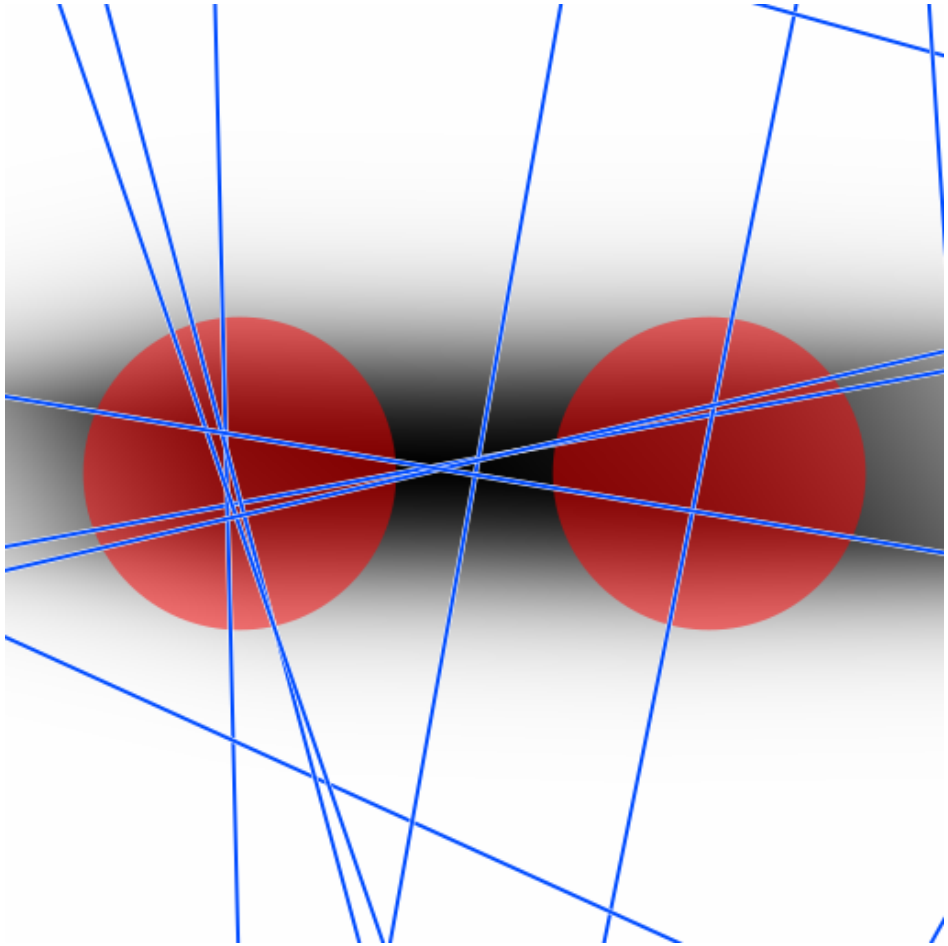
Iteration 7



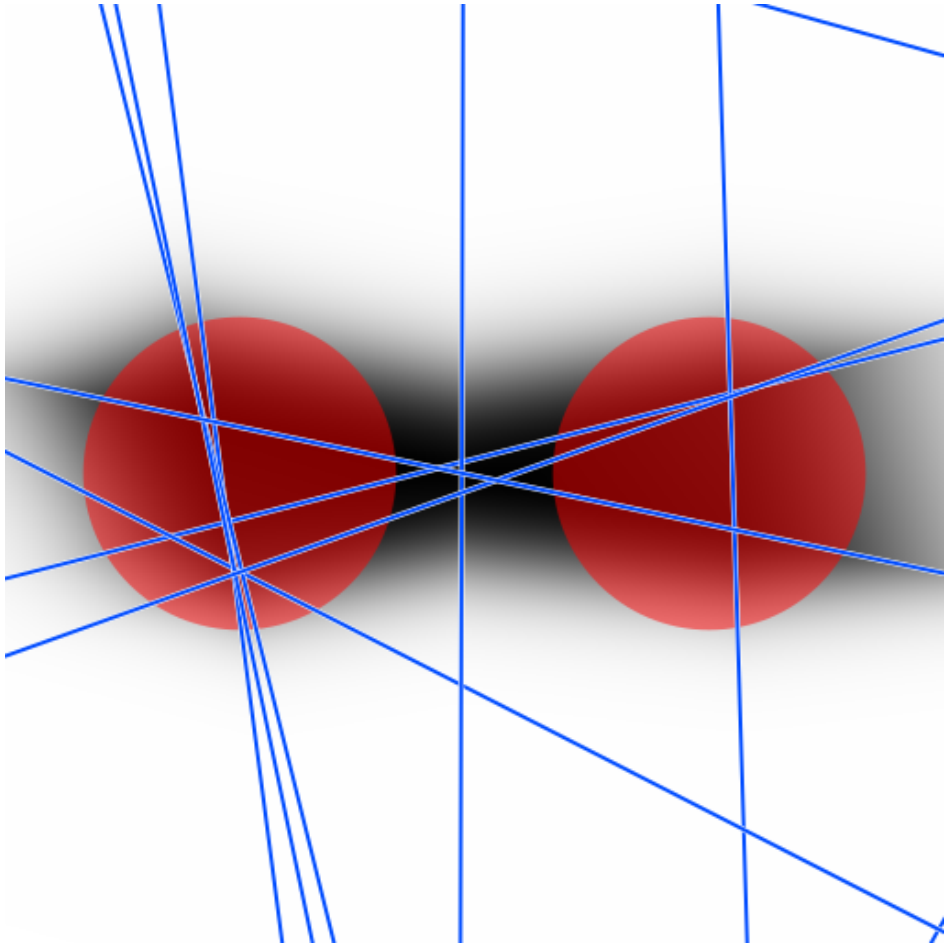
Iteration 10



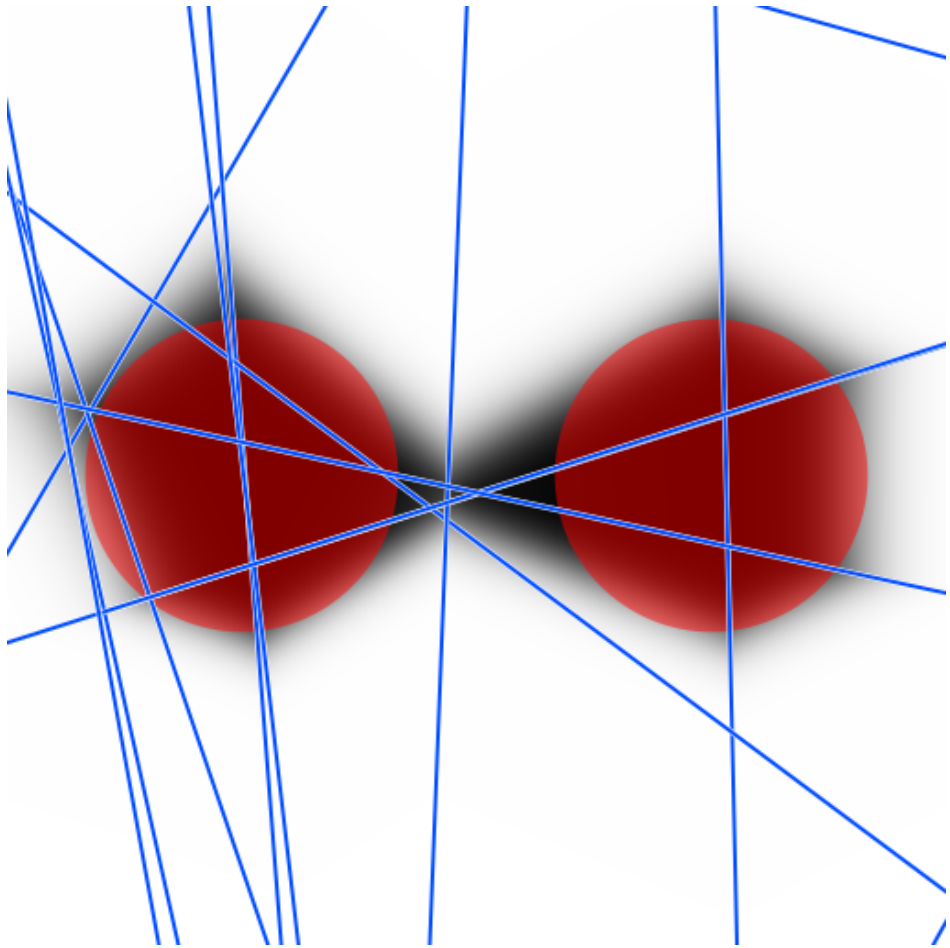
Iteration 16



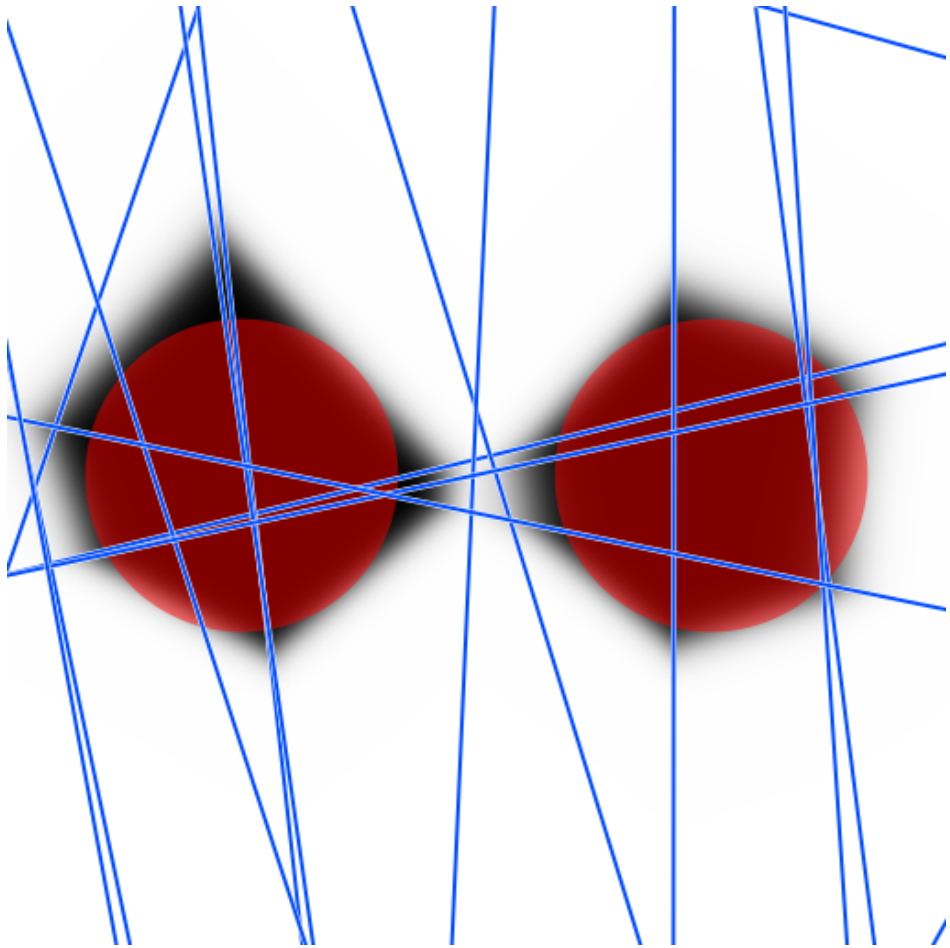
Iteration 34



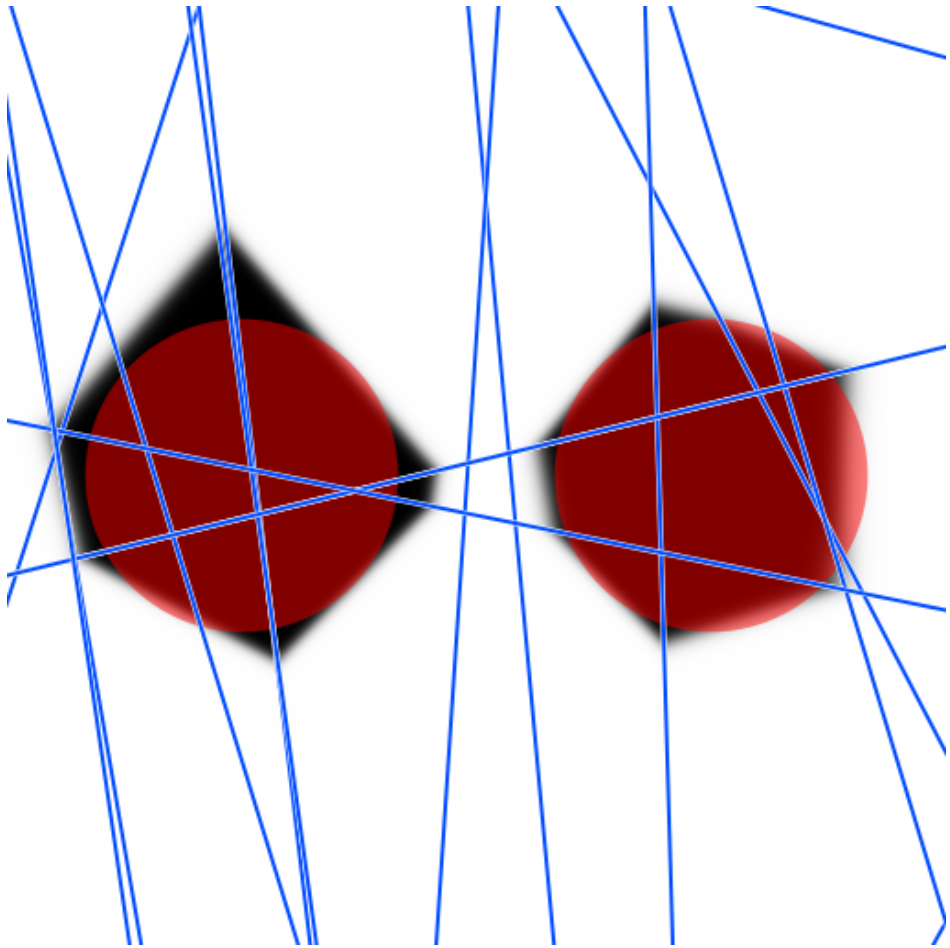
Iteration 100



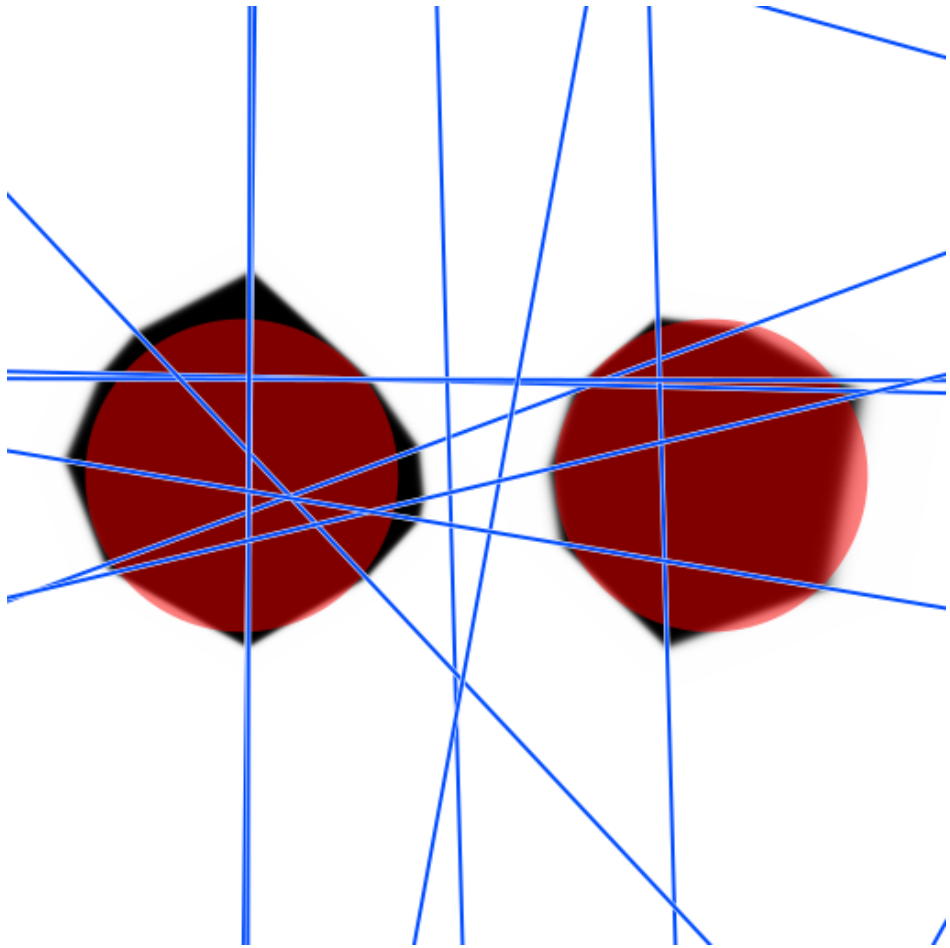
Iteration 272



Iteration 556



Iteration 2222



Iteration 9999

Convnet filters

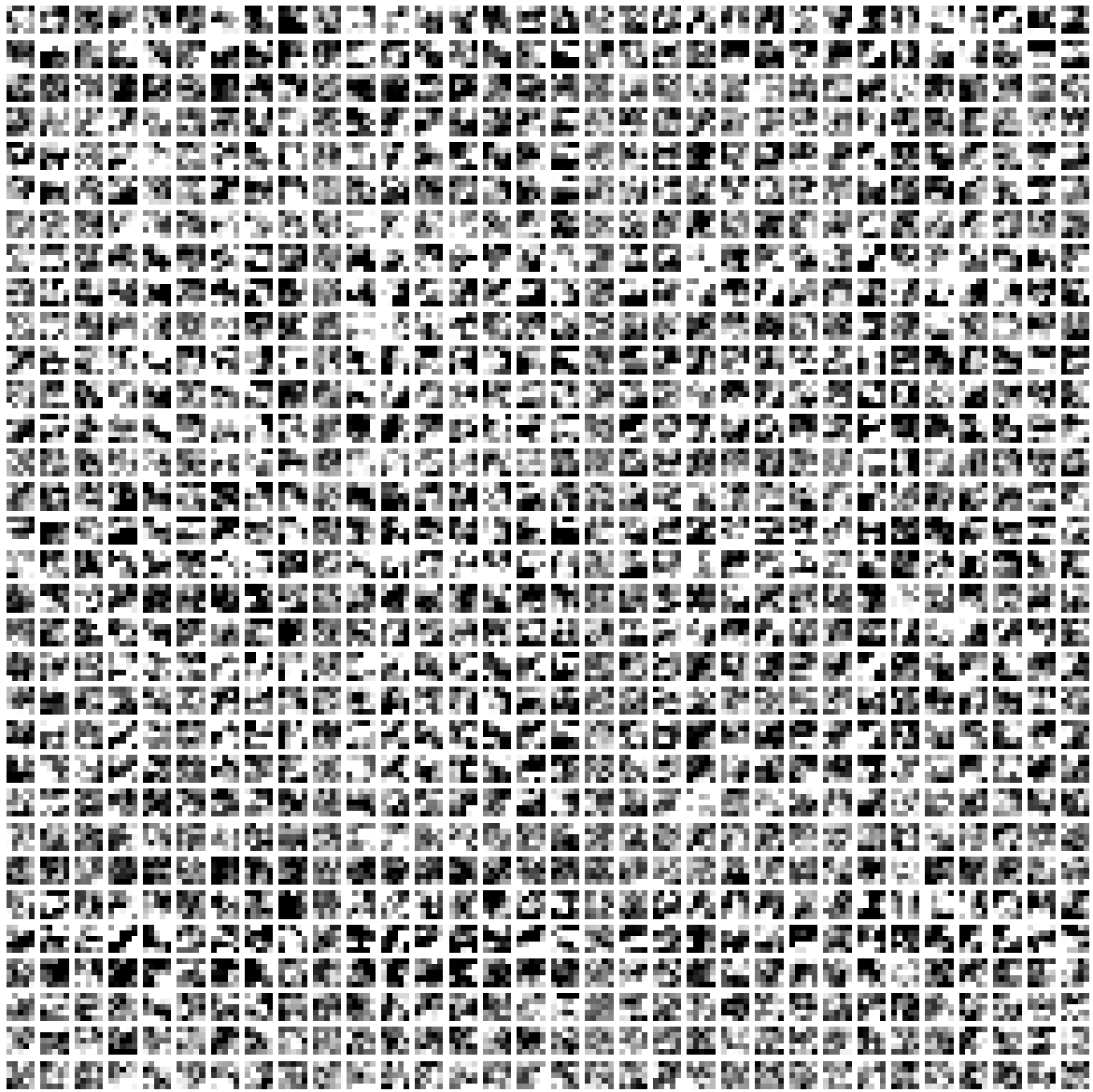
A similar analysis is complicated to conduct with real-life networks given the high dimension of the signal.

The simplest approach for convnets consists of looking at the filters as images.

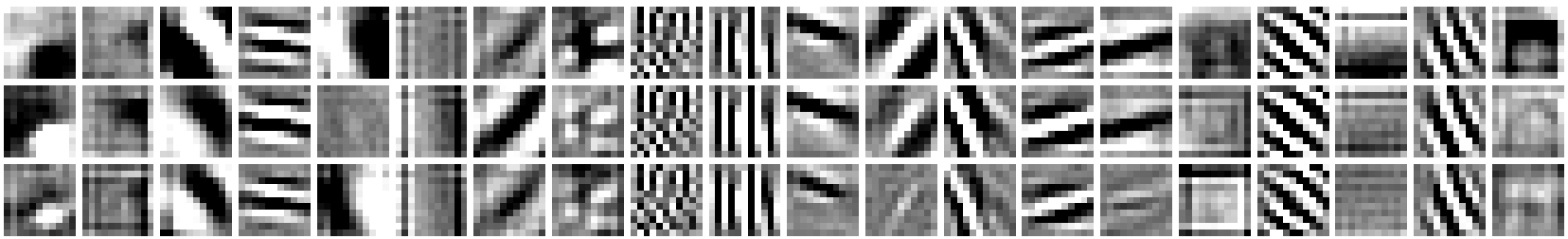
While it is quite reasonable in the first layer, since the filters are indeed consistent with the image input, it is far less so in the subsequent layers.

LeNet's first convolutional layer ($1 \rightarrow 32$), all filters

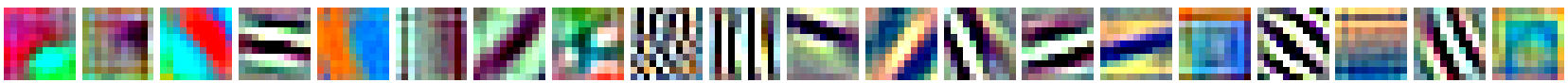
LeNet's second convolutional layer ($32 \rightarrow 64$), first 32 filters out of 64



AlexNet's first convolutional layer ($3 \rightarrow 64$), first 20 filters out of 64



or as RGB images



AlexNet's second convolutional layer ($64 \rightarrow 192$). First 15 channels (out of 64) of the first 20 filters (out of 192).

Deep learning

9.2. Looking at activations

François Fleuret

https://fleuret.org/dlc/

UNIVERSITÉ
DE GENÈVE

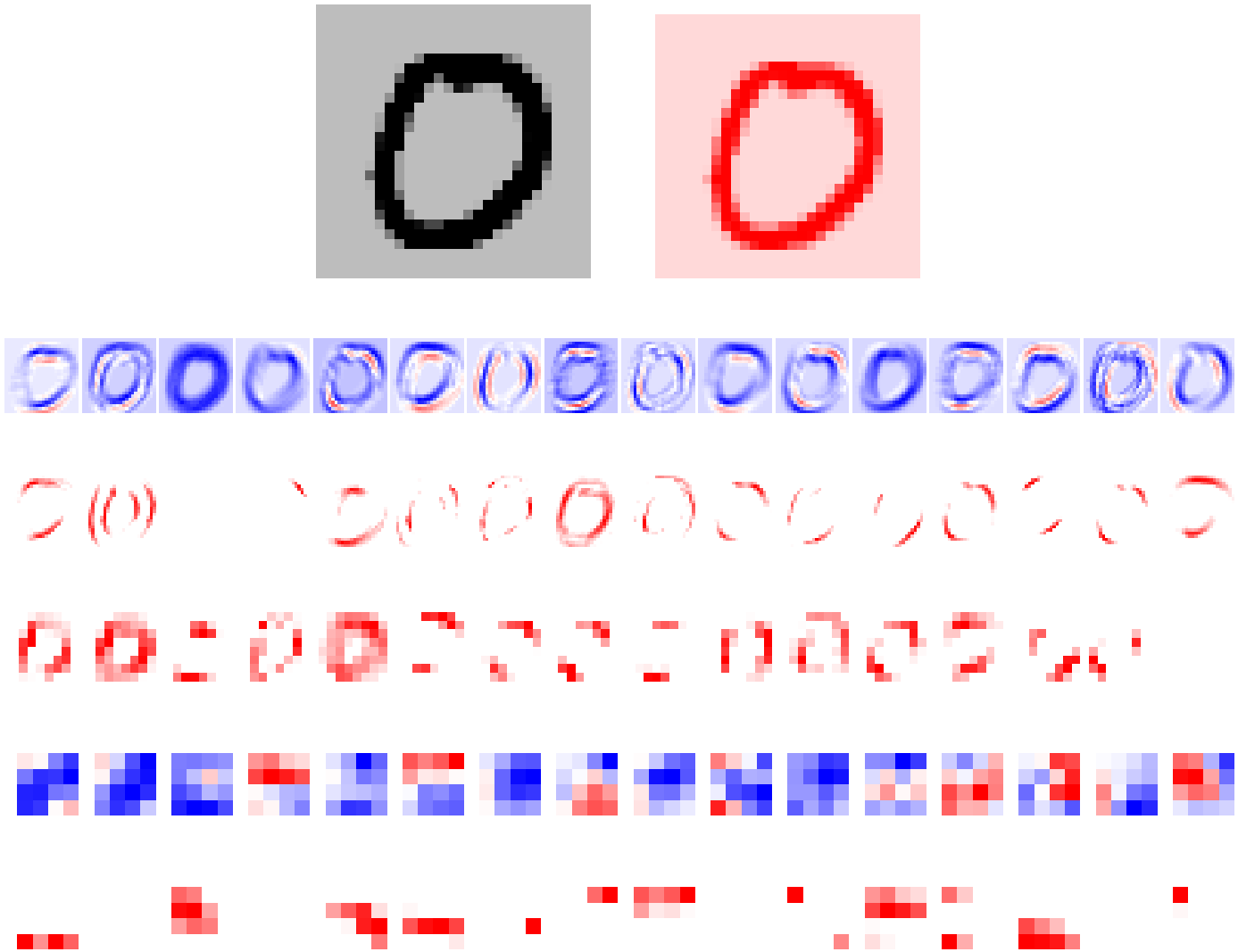
An alternative approach is to look at the activations themselves.

Since the convolutional layers maintain the 2d structure of the signal, the activations can be visualized as images, where the local coding at any location of an activation map is associated to the original content at that same location.

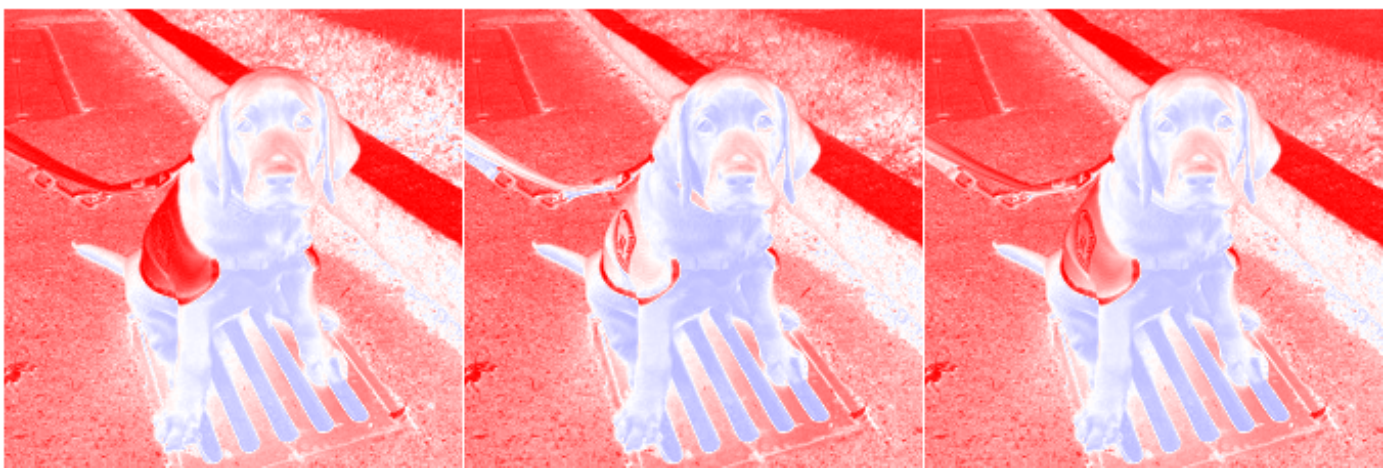
Given the large number of channels, we have to pick a few at random.

Since the representation is distributed across multiple channels, individual channel have usually no clear semantic.

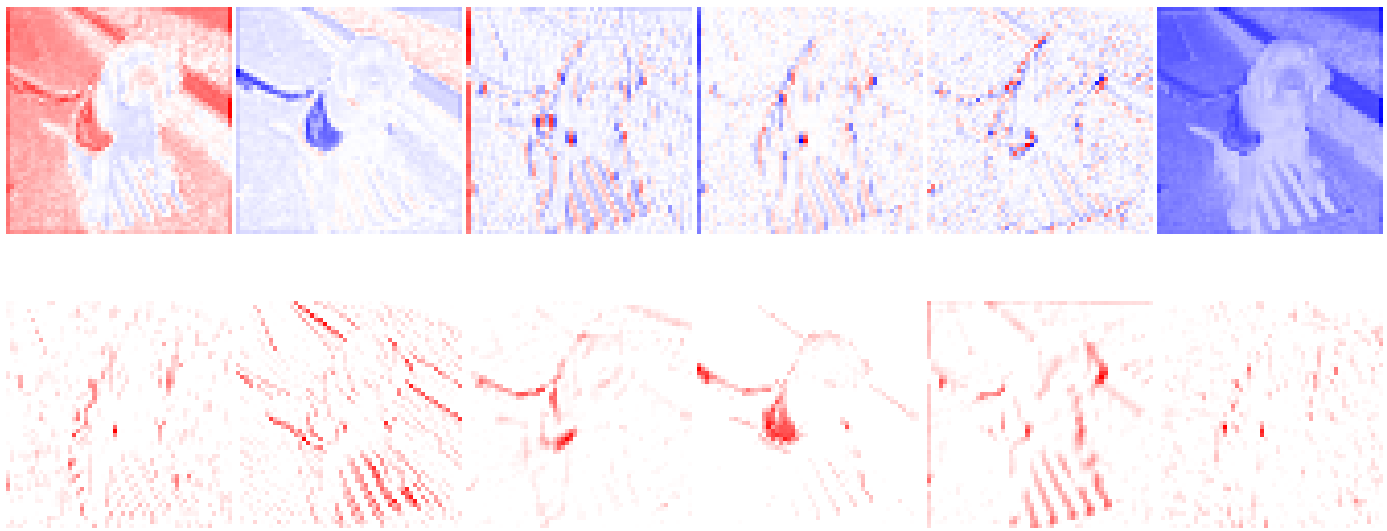
A MNIST character with LeNet (LeCun et al., 1998).



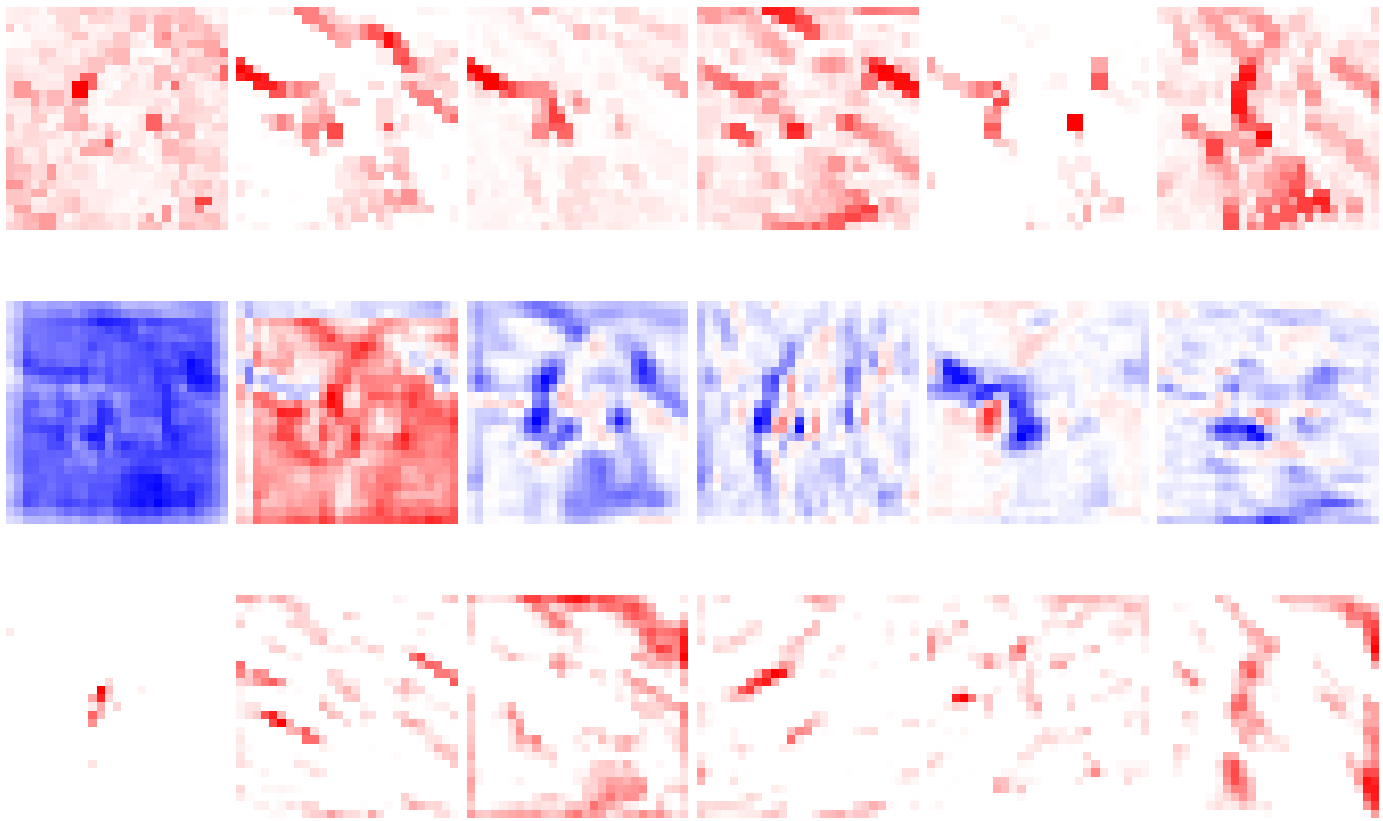
An RGB image with AlexNet (Krizhevsky et al., 2012).



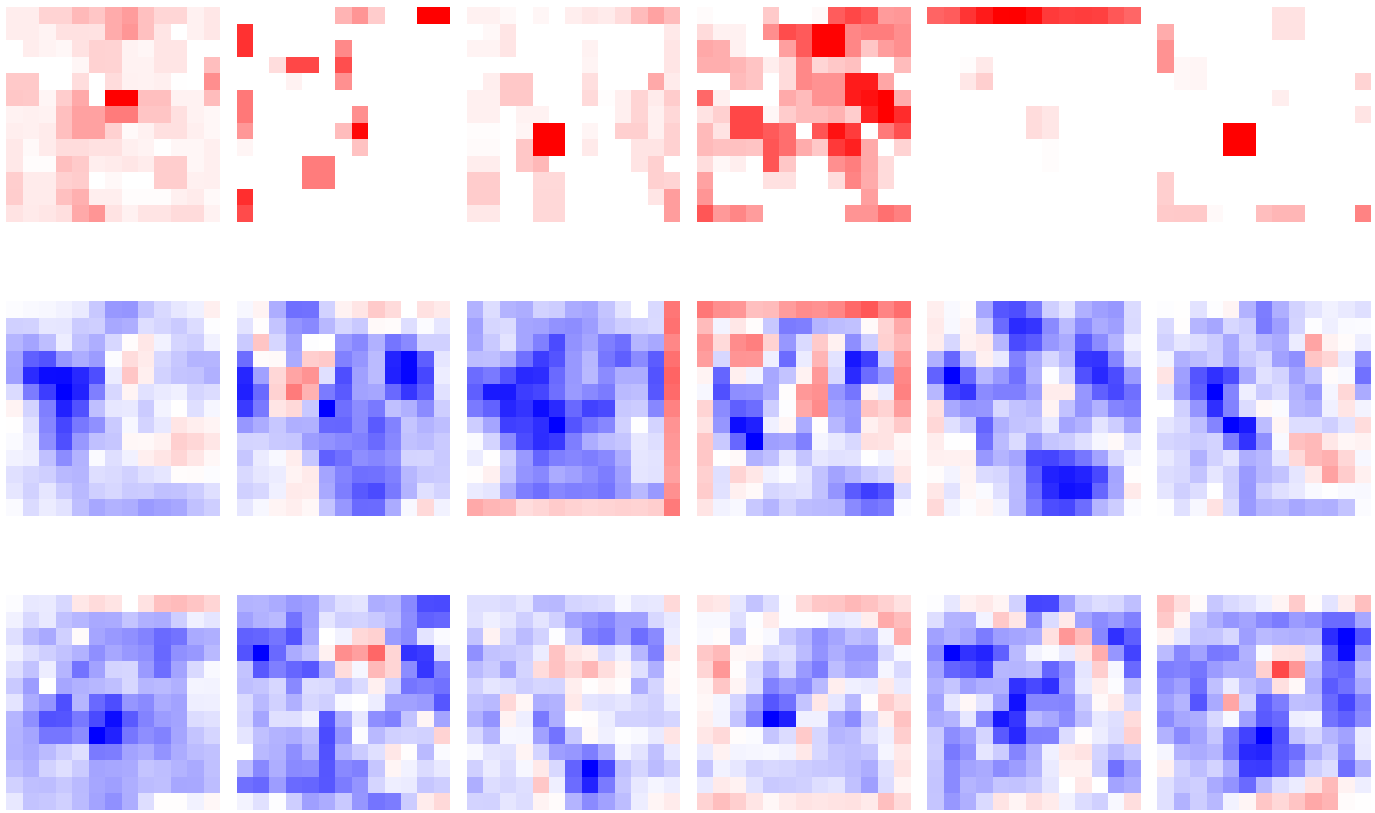
An RGB image with AlexNet (Krizhevsky et al., 2012).



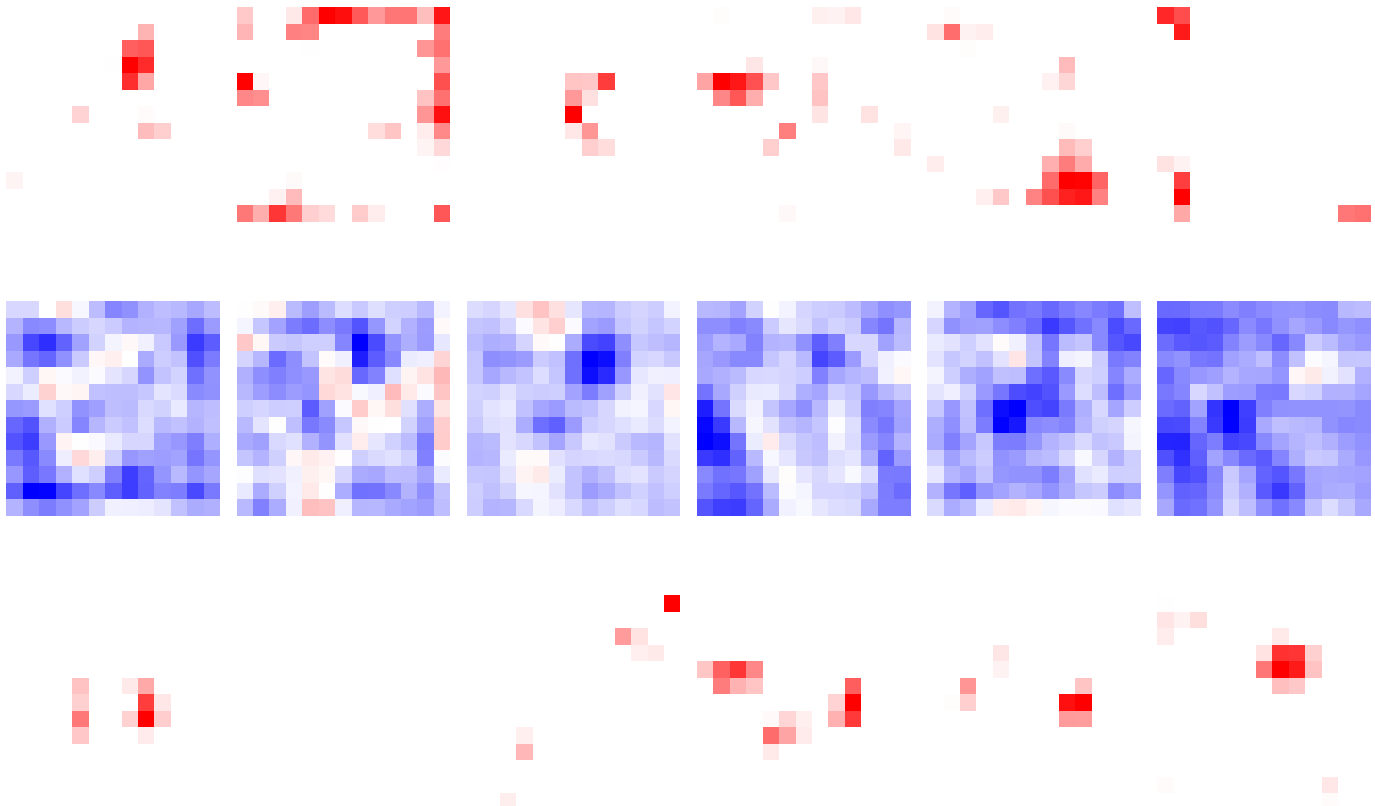
An RGB image with AlexNet (Krizhevsky et al., 2012).



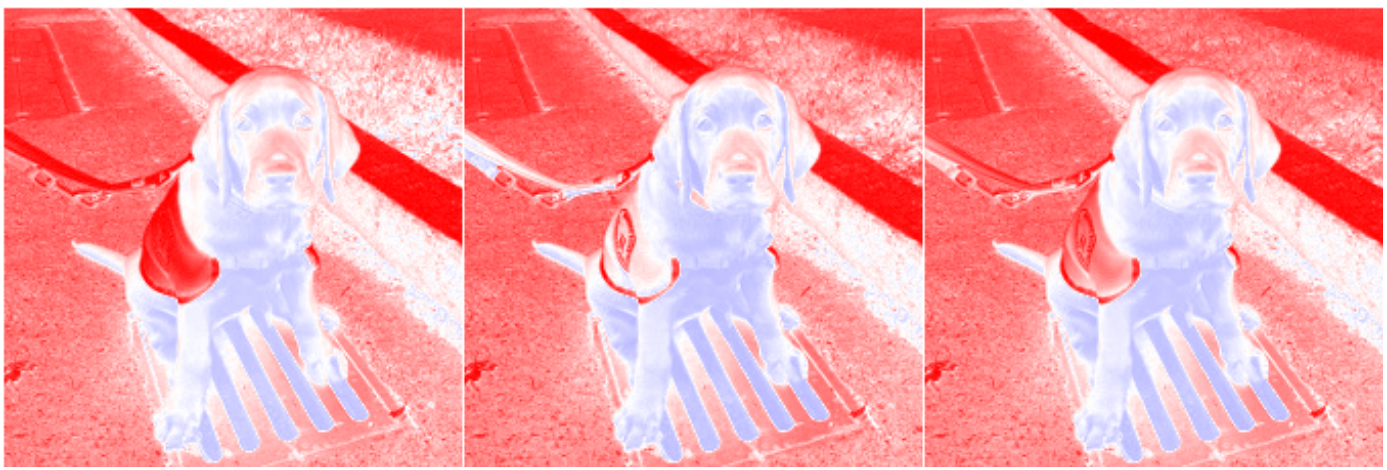
An RGB image with AlexNet (Krizhevsky et al., 2012).



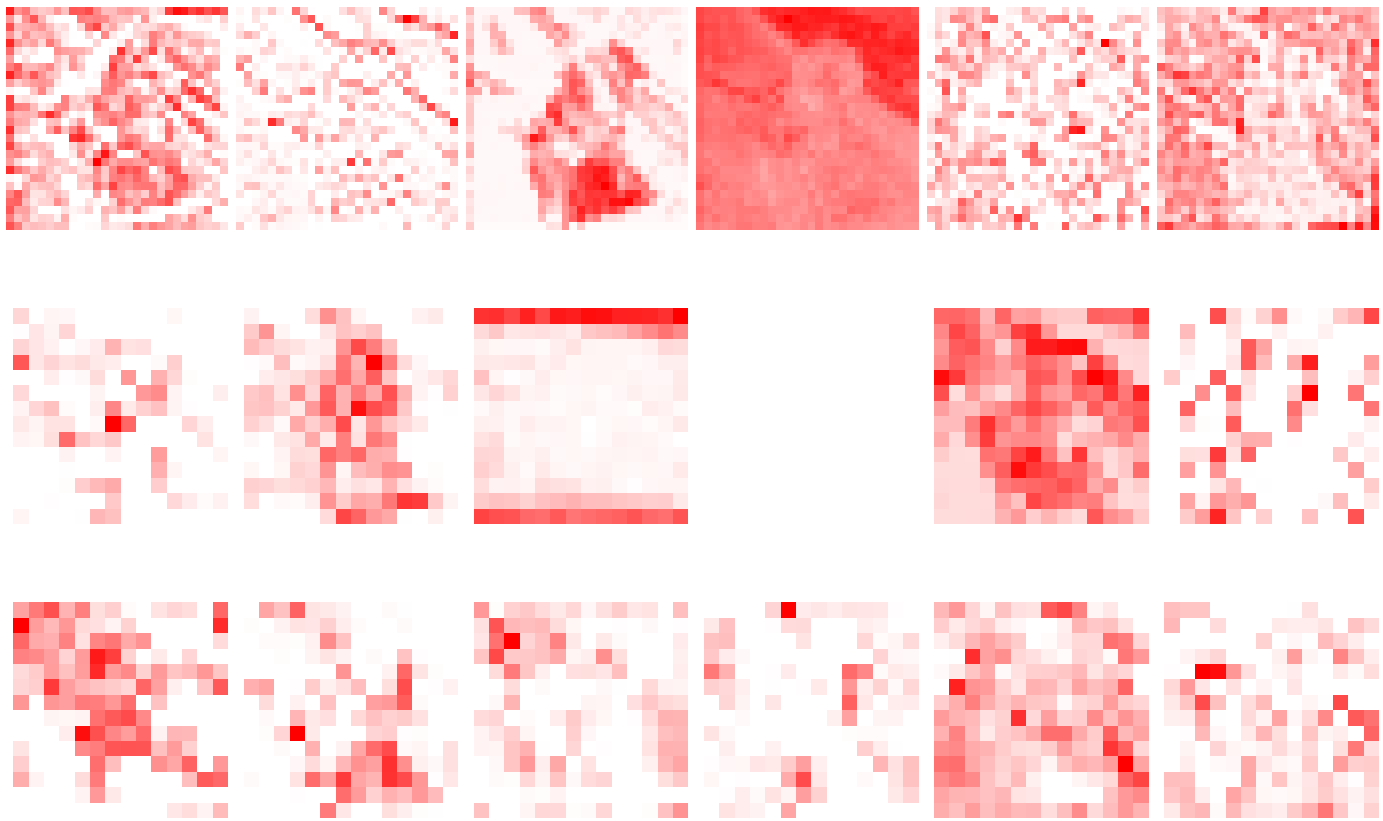
An RGB image with AlexNet (Krizhevsky et al., 2012).



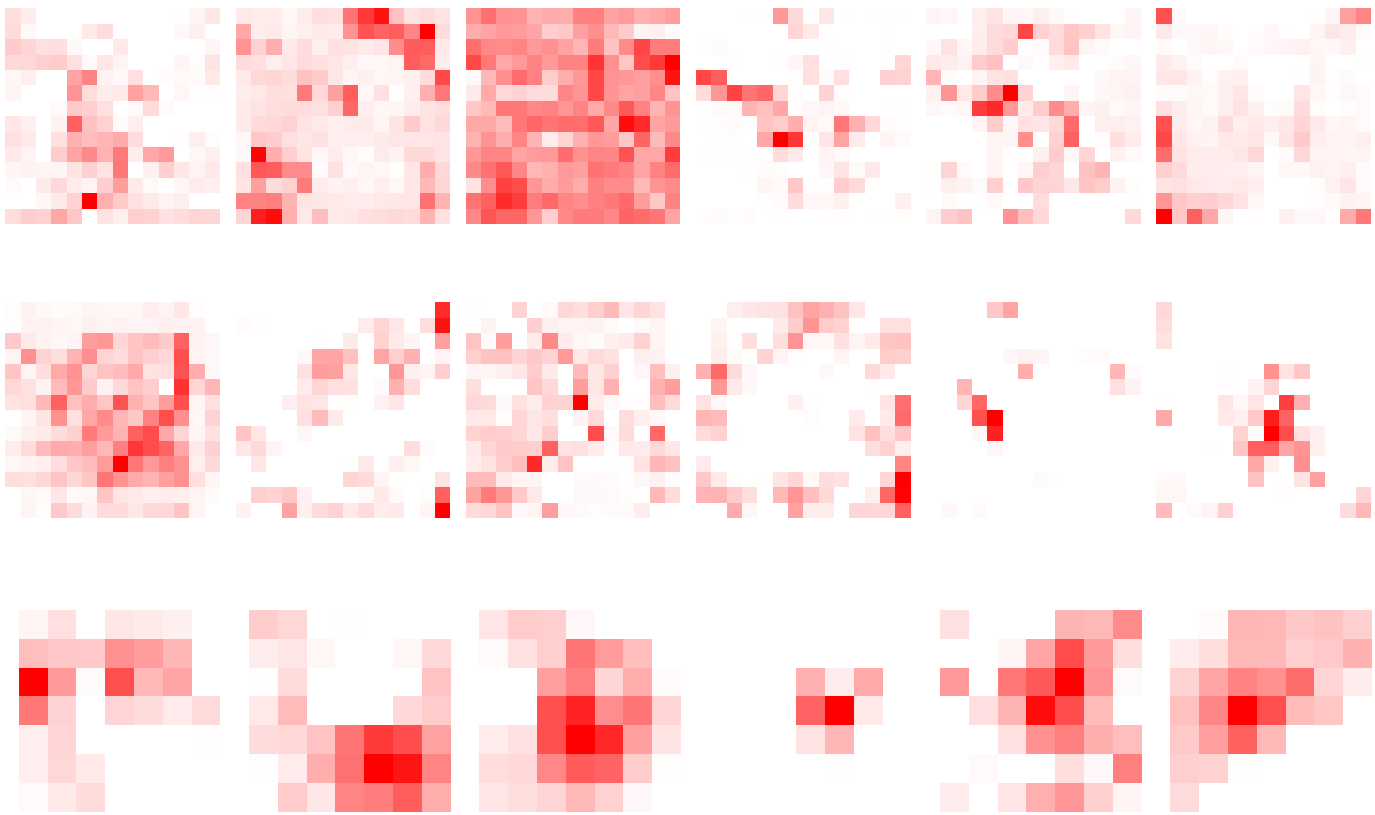
ILSVRC12 with ResNet152 (He et al., 2015).



ILSVRC12 with ResNet152 (He et al., 2015).



ILSVRC12 with ResNet152 (He et al., 2015).



Yosinski et al. (2015) developed analysis tools to visit a network and look at the internal activations for a given input signal.

This allowed them in particular to find units with a clear semantic in an AlexNet-like network trained on ImageNet.

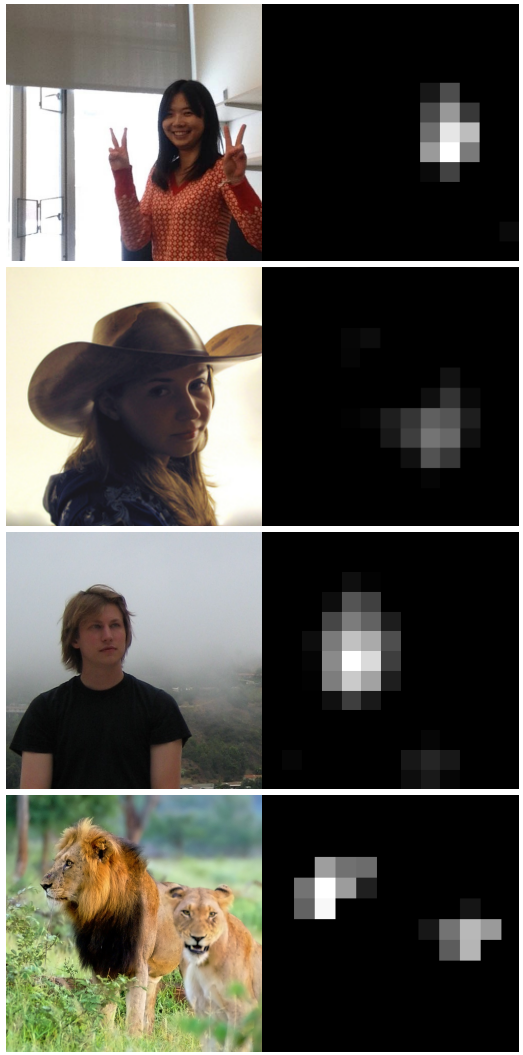
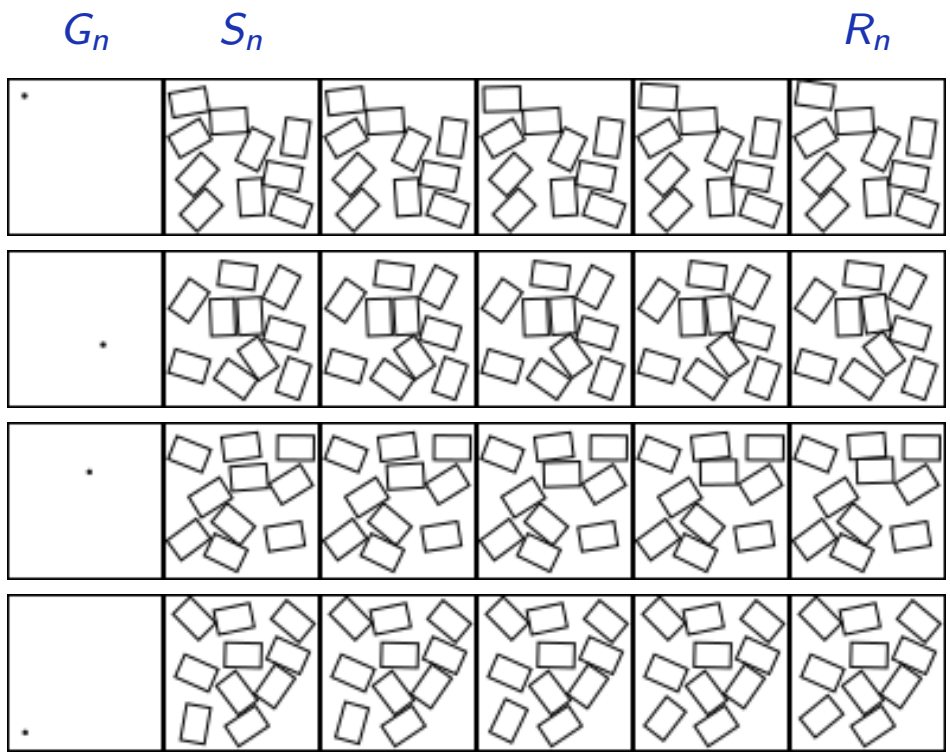


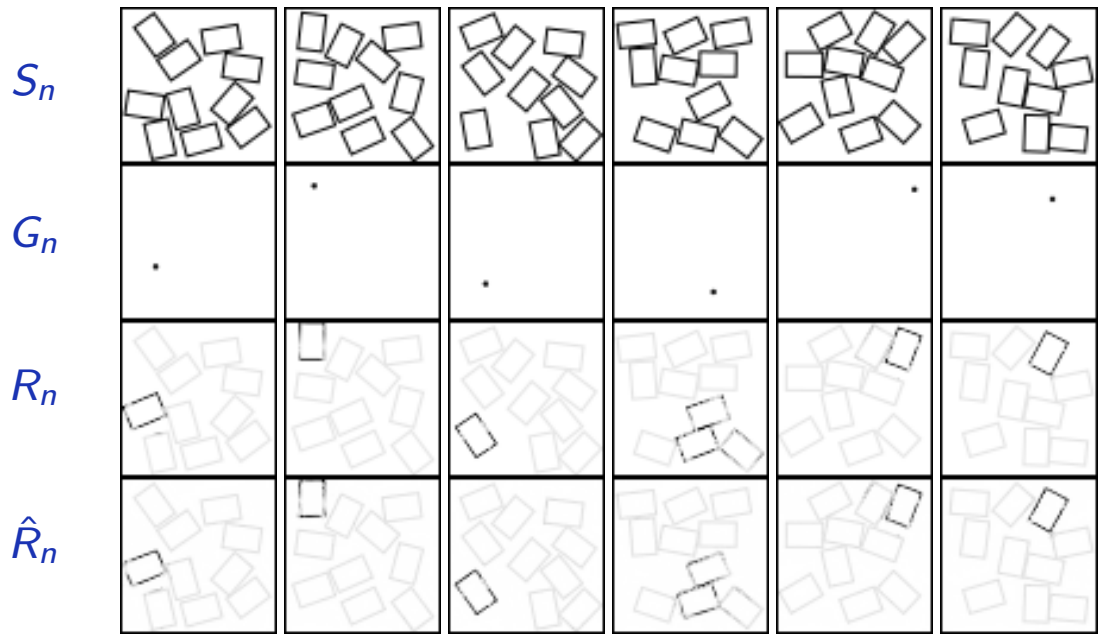
Figure 2. A view of the 13×13 activations of the 151st channel on the conv5 layer of a deep neural network trained on ImageNet, a dataset that does not contain a face class, but does contain many images with faces. The channel responds to human and animal faces and is robust to changes in scale, pose, lighting, and context, which can be discerned by a user by actively changing the scene in front of a webcam or by loading static images (e.g. of the lions) and seeing the corresponding response of the unit. Photo of lions via Flickr user arnolouise, licensed under CC BY-NC-SA 2.0.

(Yosinski et al., 2015)

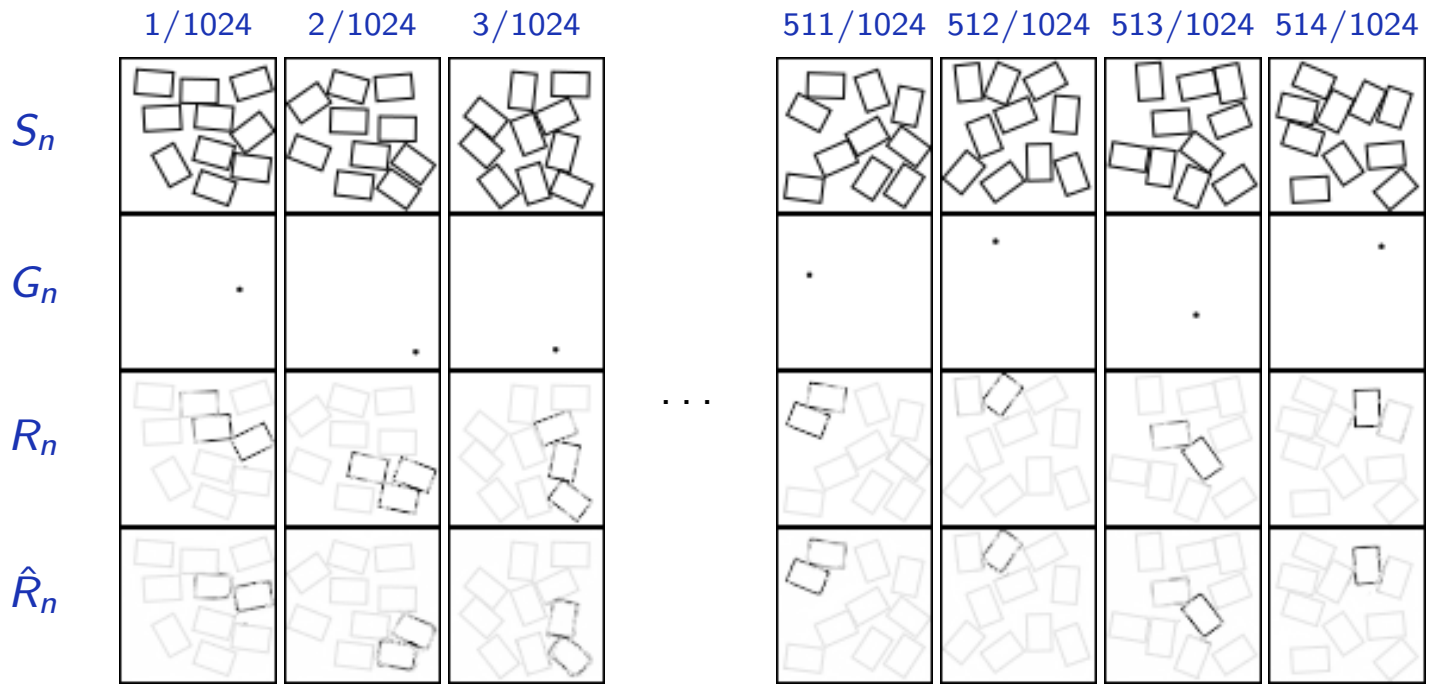
Prediction of 2d dynamics with a 18 layer residual network.



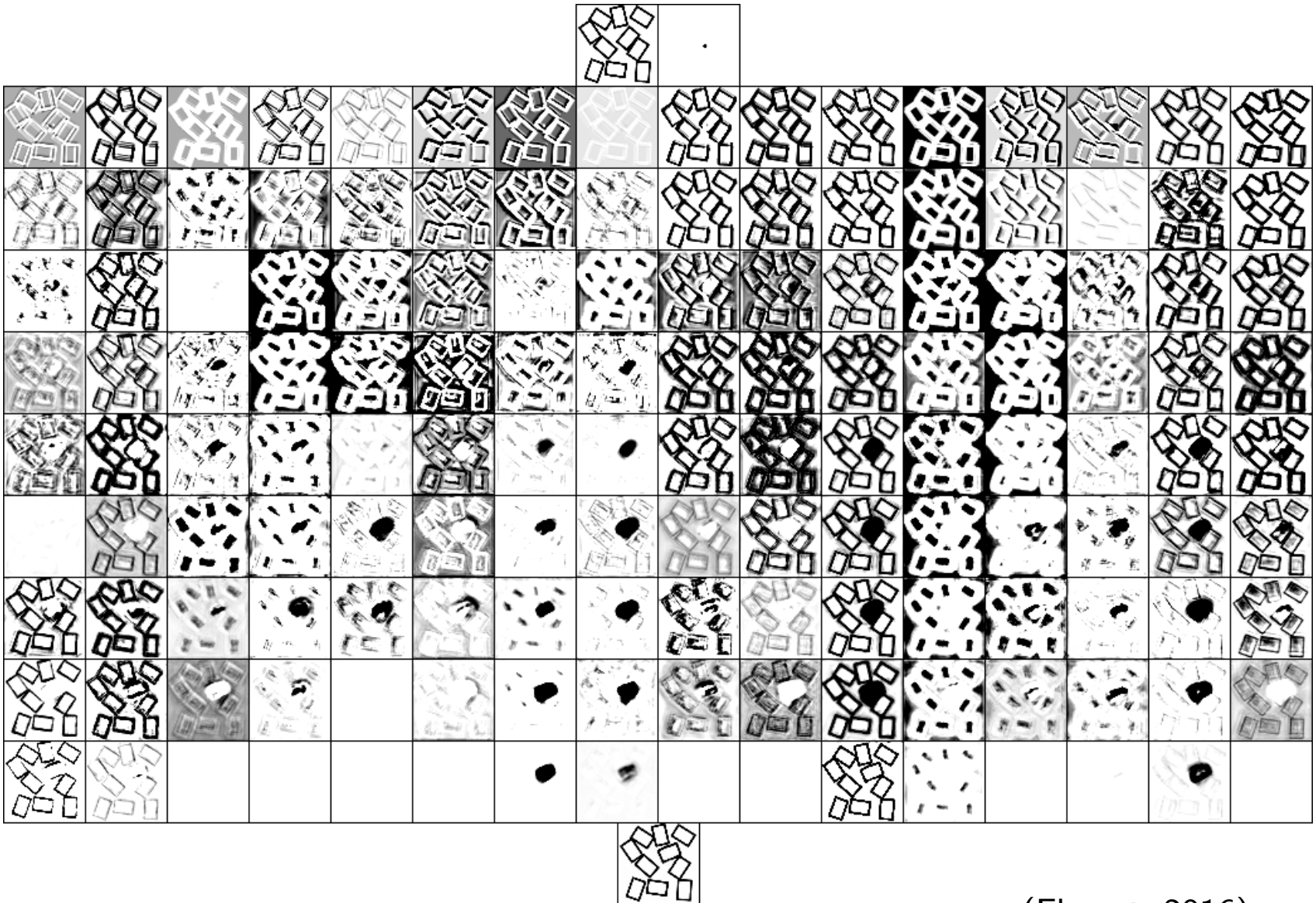
(Fleuret, 2016)



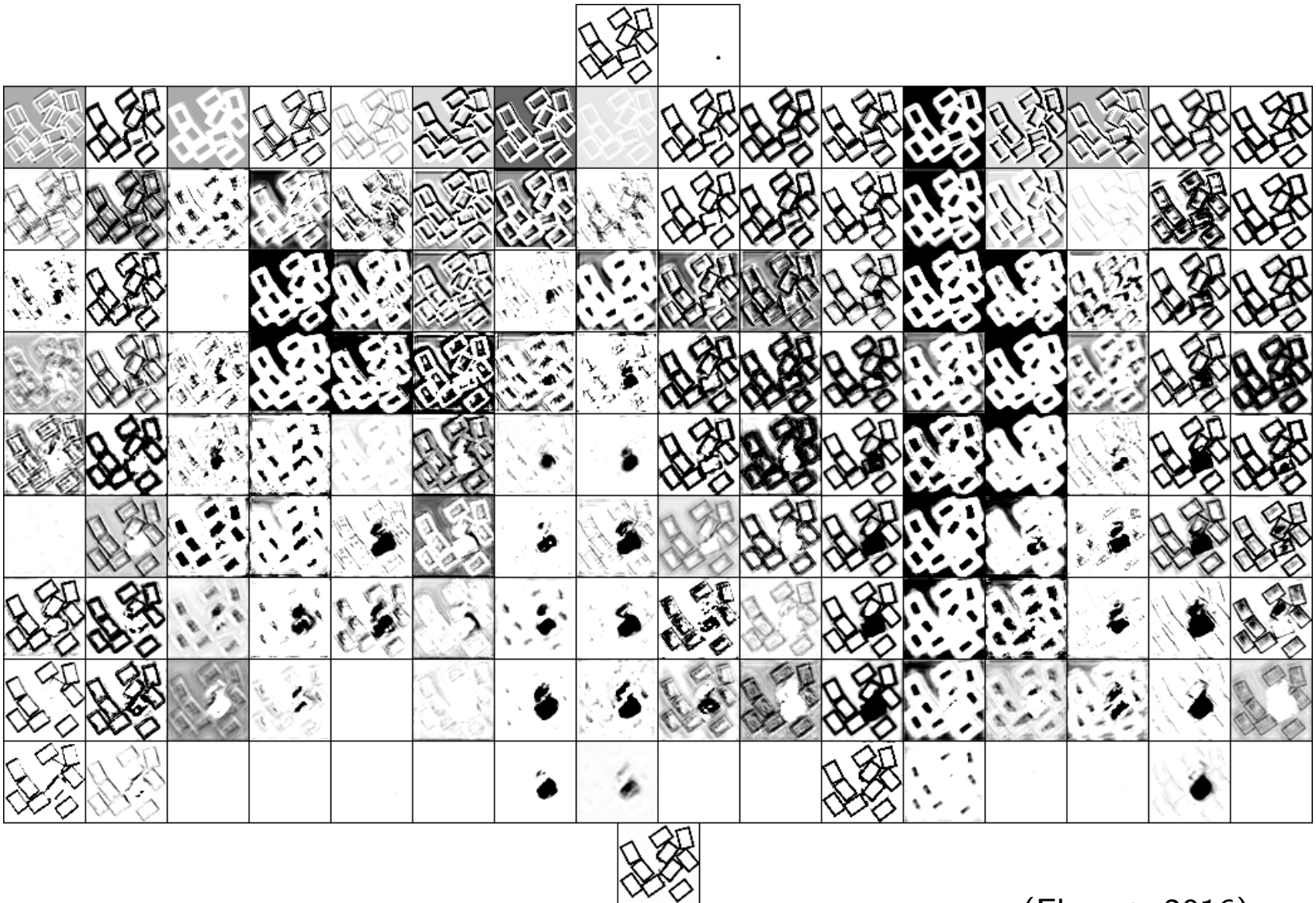
(Fleuret, 2016)



(Fleuret, 2016)



(Fleuret, 2016)



(Fleuret, 2016)

Layers as embeddings

In the classification case, the network can be seen as a series of processings aiming as disentangling classes to make them easily separable for the final decision.

In this perspective, it makes sense to look at how the samples are distributed spatially after each layer.

The main issue to do so is the dimensionality of the signal. If we look at the total number of dimensions in each layer:

- A MNIST sample in a LeNet goes from 784 to up to 18k dimensions,
- A ILSVRC12 sample in ResNet152 goes from 150k to up to 800k dimensions.

This requires a mean to project a [very] high dimension point cloud into a 2d or 3d "human-brain accessible" representation

We have already seen PCA and k-means as two standard methods for dimension reduction, but they poorly convey the structure of a smooth low-dimension and non-flat manifold.

It exists a plethora of methods that aim at reflecting in low-dimension the structure of data points in high dimension.

Given data-points in high dimension

$$\mathscr{D} = \left\{ x_n \in \mathbb{R}^D, \ n = 1, \ldots, N \right\}$$

the objective of data-visualization is to find a set of corresponding low-dimension points

$$\mathscr{E} = \left\{ y_n \in \mathbb{R}^C, \ n = 1, \ldots, N \right\}$$

such that the positions of the ys "reflect" that of the xs.

The **t-Distributed Stochastic Neighbor Embedding** (t-SNE) proposed by van der Maaten and Hinton (2008) optimizes with SGD the y_is so that the distributions of distances to close neighbors of each point are preserved.

It actually matches for $\mathbb{D}_{\mathrm{KL}}$ two distance-dependent distributions: Gaussian in the original space, and Student t-distribution in the low-dimension one.

The scikit-learn toolbox

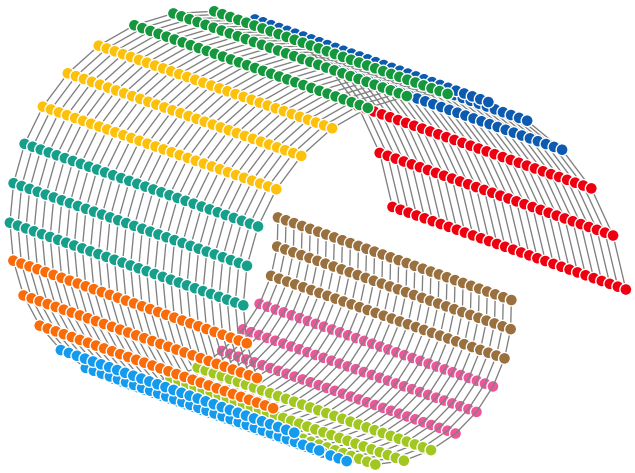
http://scikit-learn.org/

is built around SciPy, and provides many machine learning algorithms, in particular embeddings, among which an implementation of t-SNE.

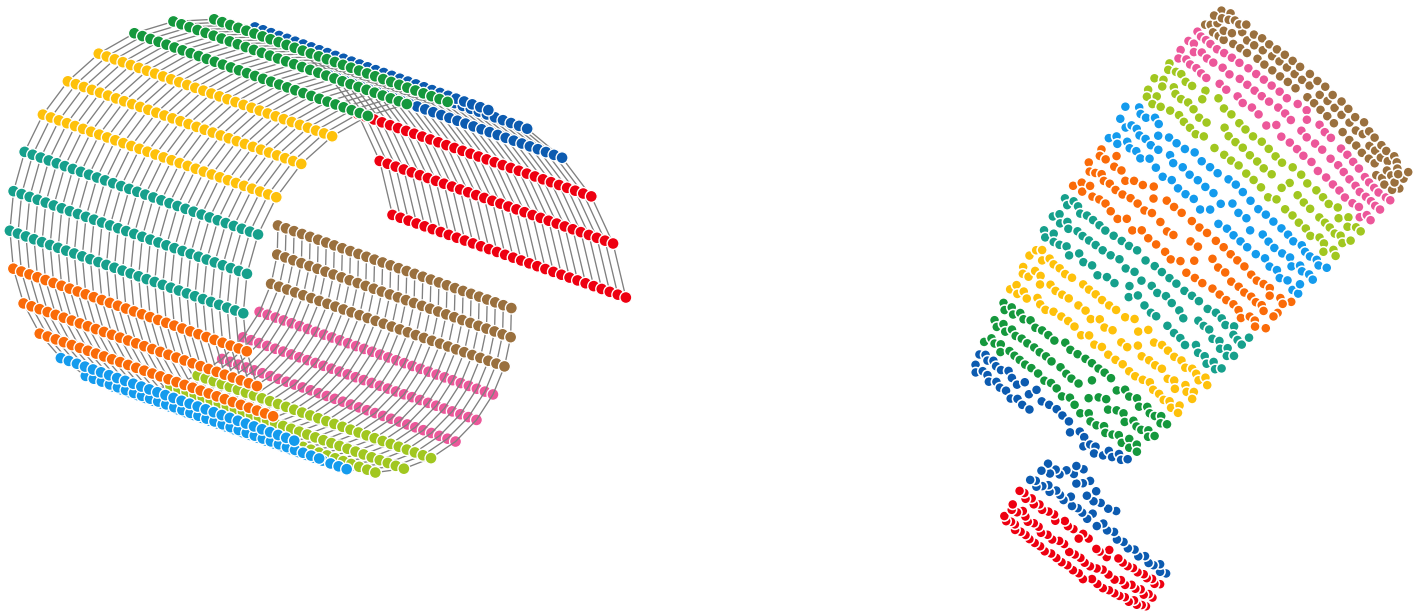
The only catch to use it in PyTorch is the conversions to and from numpy arrays.

```
from sklearn.manifold import TSNE

# x is the array of the original high-dimension points
x_np = x.numpy()
y_np = TSNE(n_components = 2, perplexity = 50).fit_transform(x_np)
# y is the array of corresponding low-dimension points
y = torch.from_numpy(y_np)
```

`n_components` specifies the embedding dimension and `perplexity` states [crudely] how many points are considered neighbors of each point.

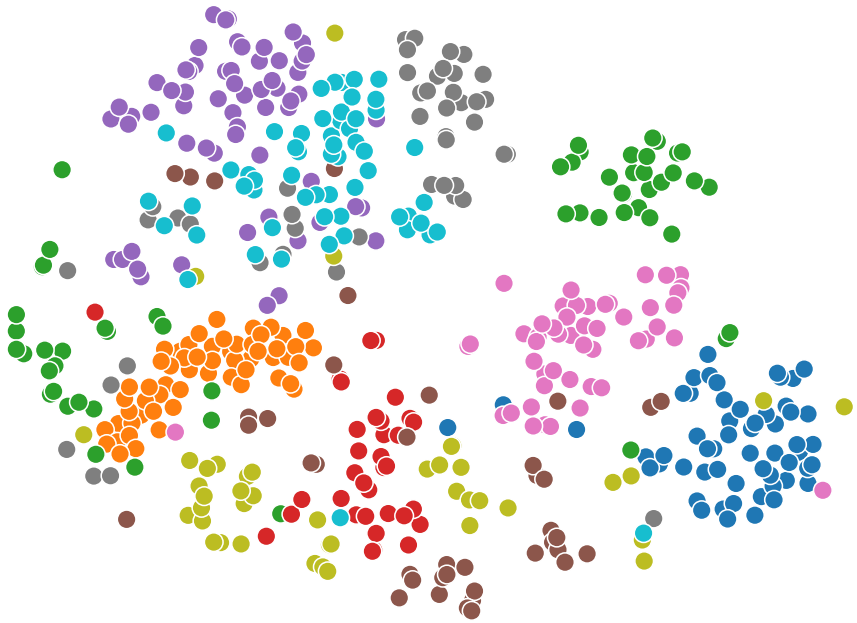


t-SNE unrolling of the swiss roll (with one noise dimension)

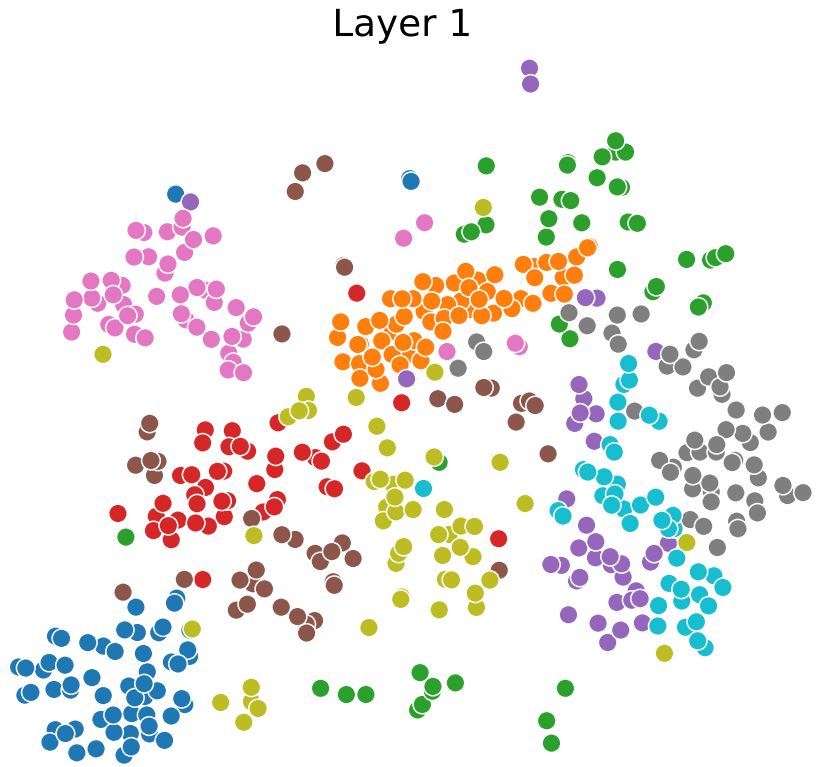


t-SNE unrolling of the swiss roll (with one noise dimension)

Input

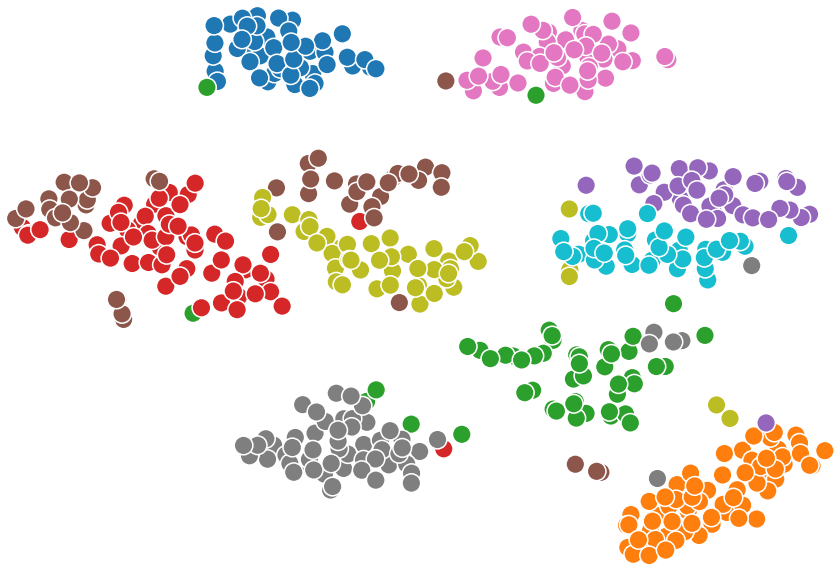


t-SNE for LeNet on MNIST



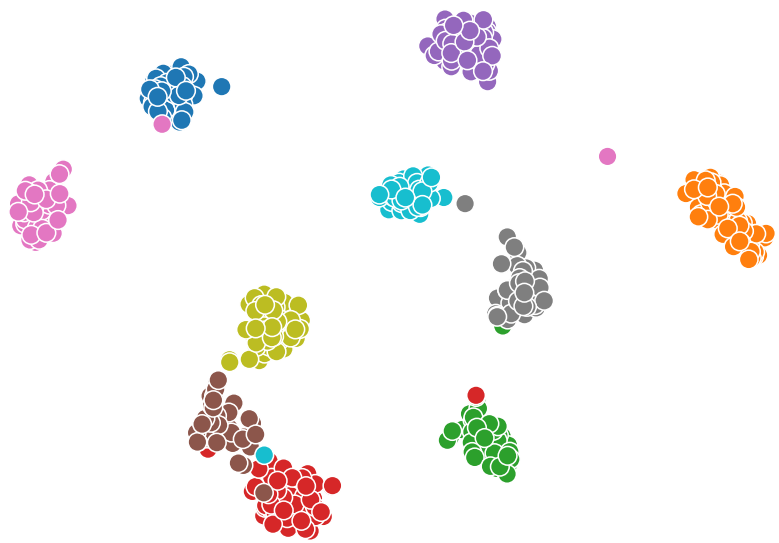
t-SNE for LeNet on MNIST

Layer 4



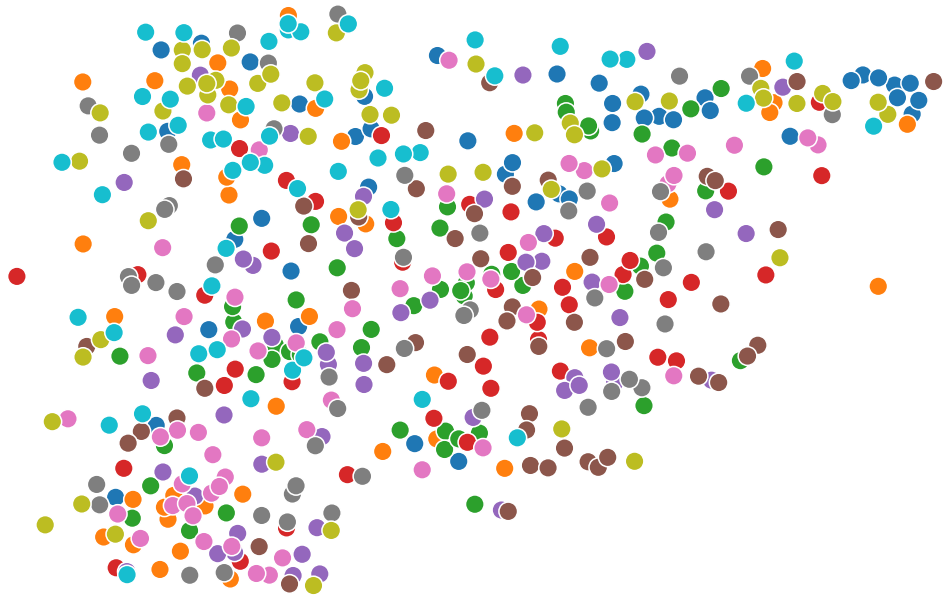
t-SNE for LeNet on MNIST

Layer 7

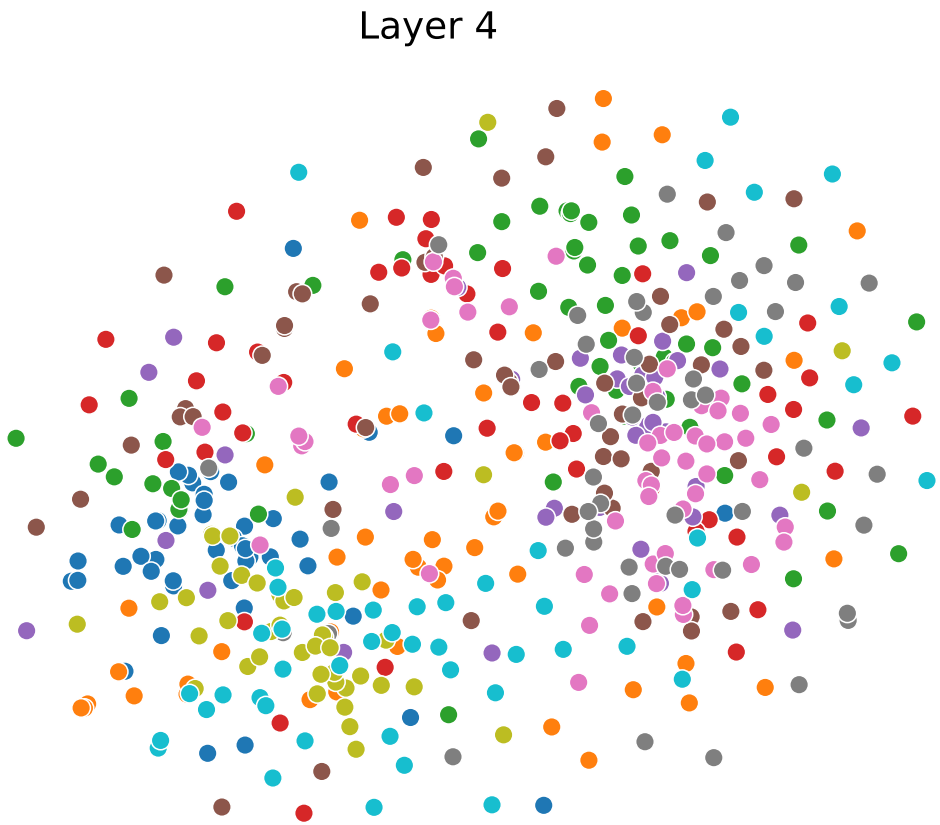


t-SNE for LeNet on MNIST

Input

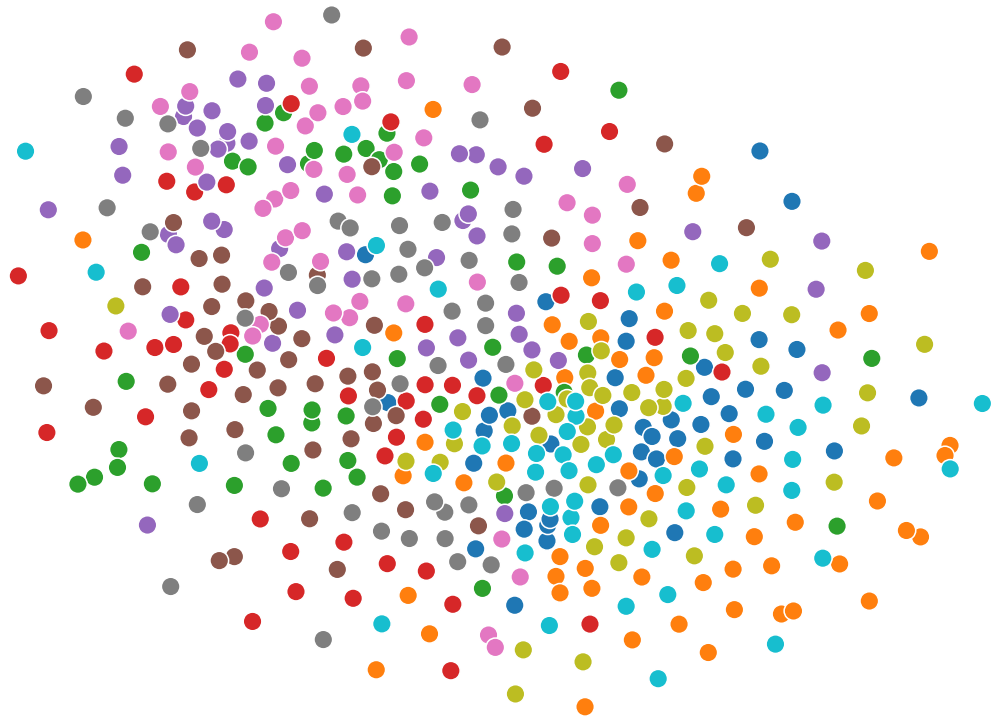


t-SNE for a home-baked ResNet (no pooling, 66 layers) CIFAR10



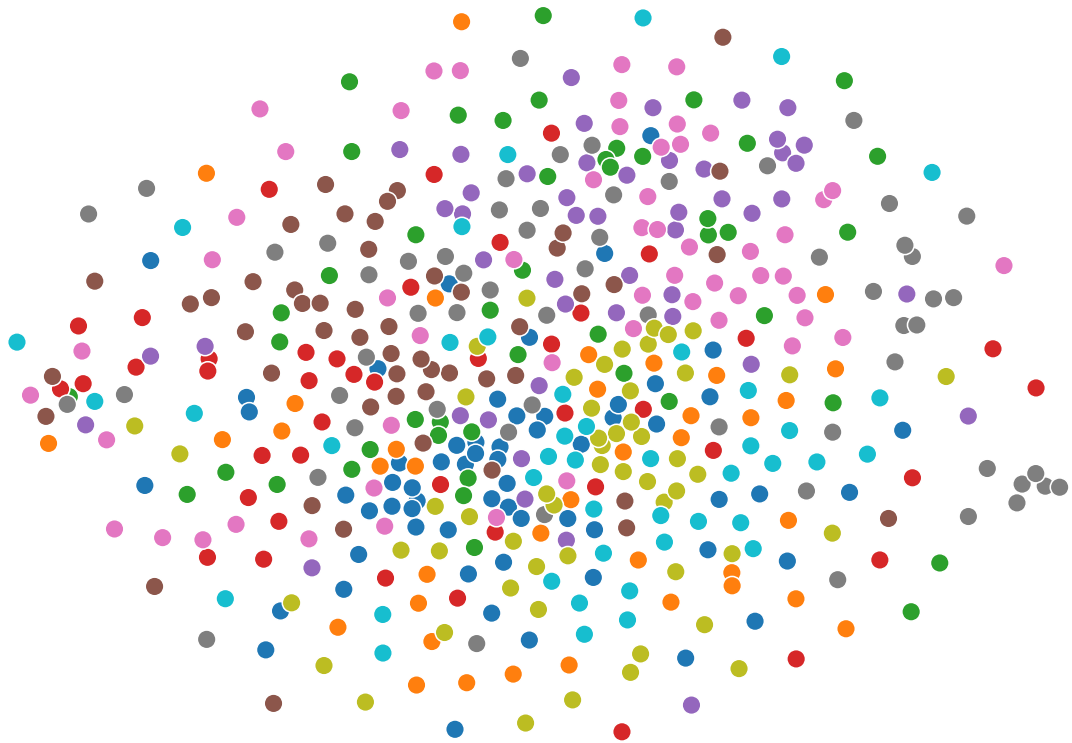
t-SNE for a home-baked ResNet (no pooling, 66 layers) CIFAR10

Layer 14



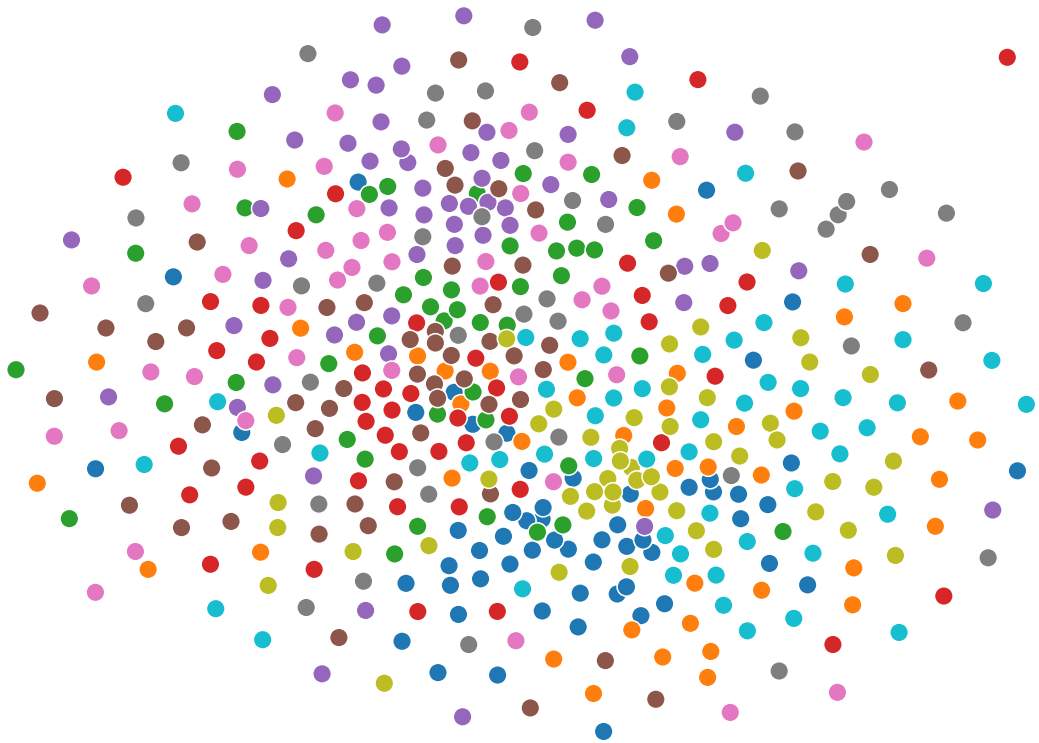
t-SNE for a home-baked ResNet (no pooling, 66 layers) CIFAR10

Layer 24

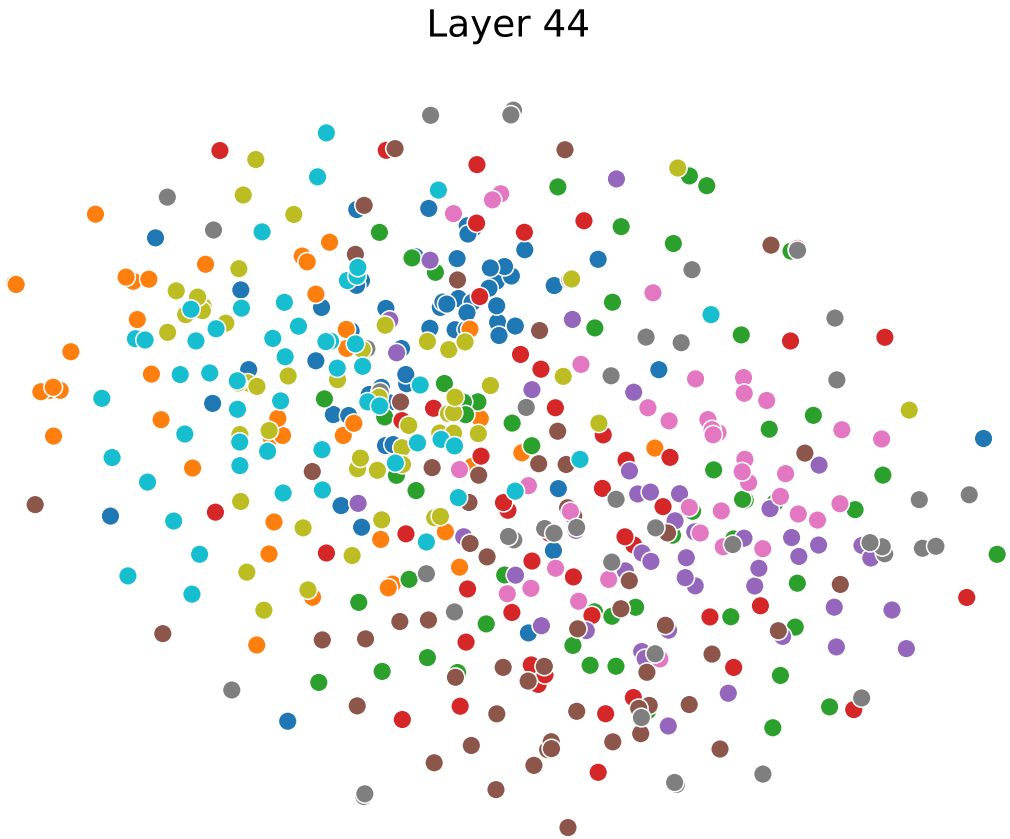


t-SNE for a home-baked ResNet (no pooling, 66 layers) CIFAR10

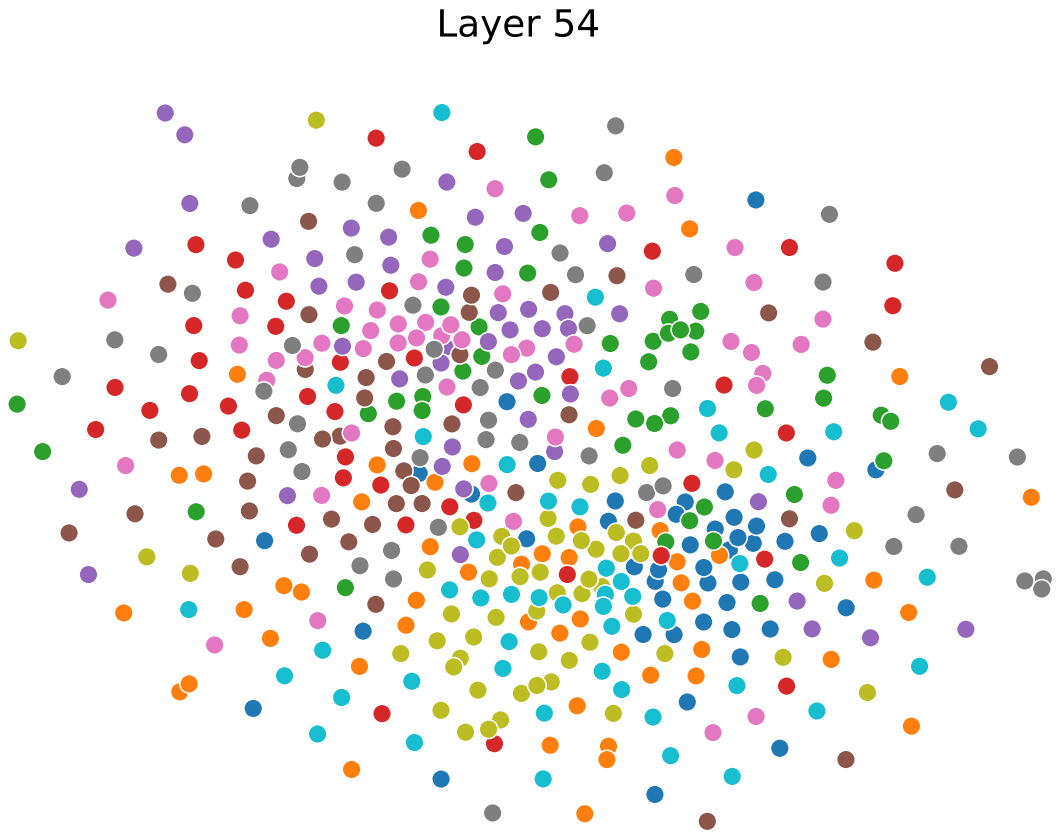
Layer 34



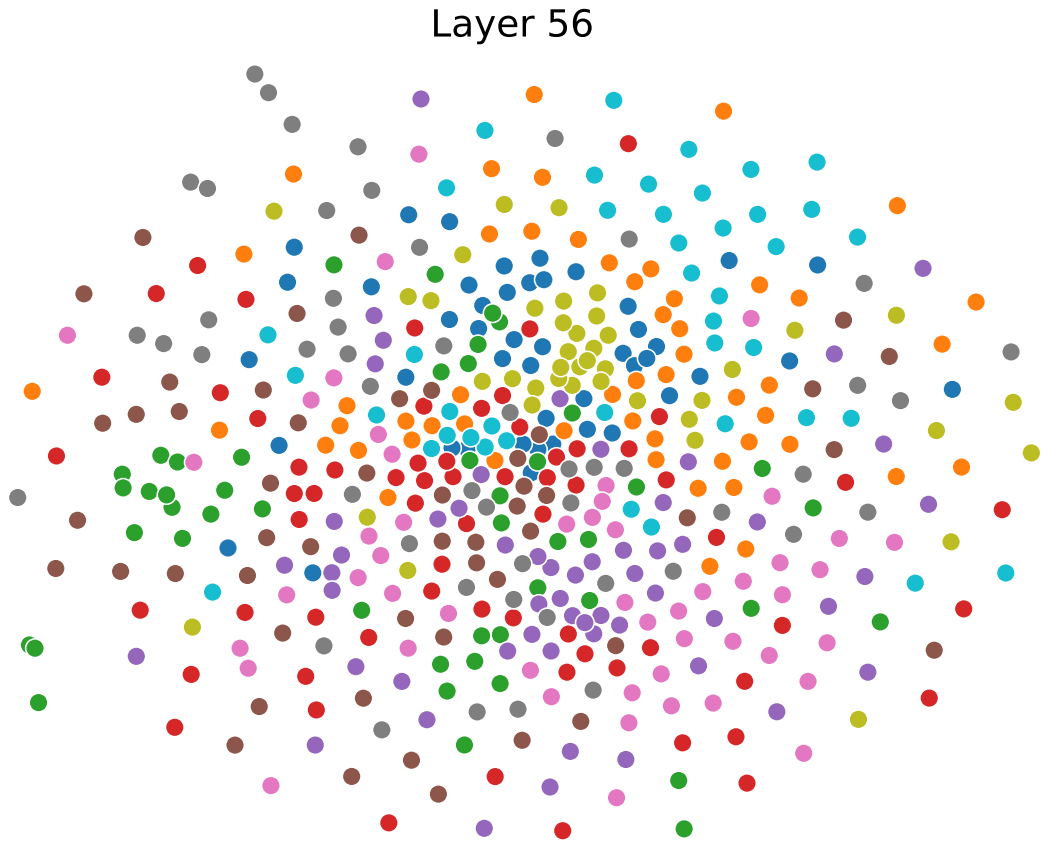
t-SNE for a home-baked ResNet (no pooling, 66 layers) CIFAR10



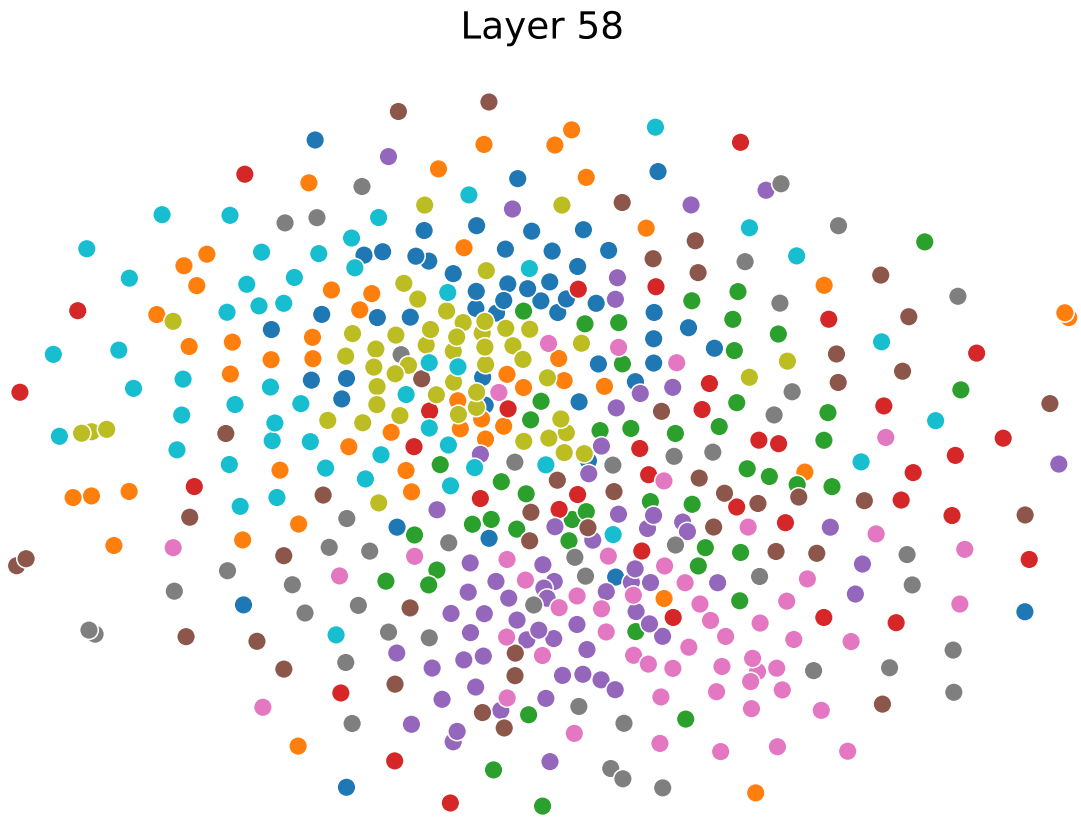
t-SNE for a home-baked ResNet (no pooling, 66 layers) CIFAR10



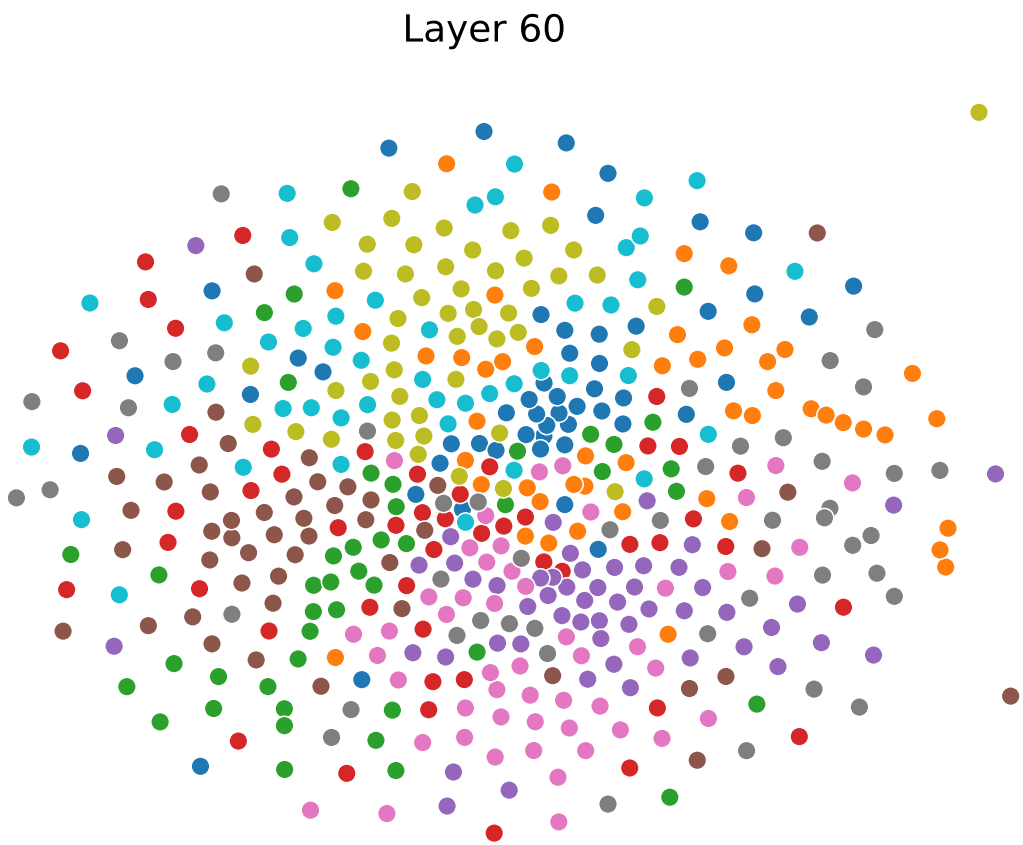
t-SNE for a home-baked ResNet (no pooling, 66 layers) CIFAR10



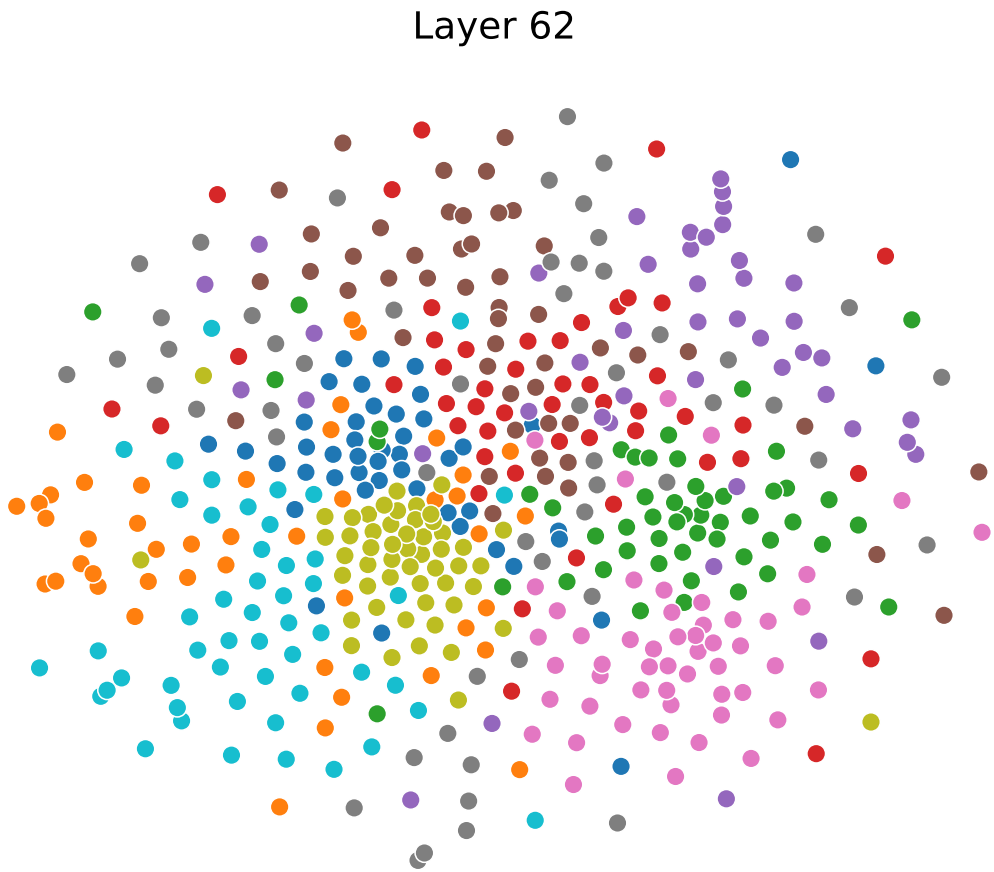
t-SNE for a home-baked ResNet (no pooling, 66 layers) CIFAR10



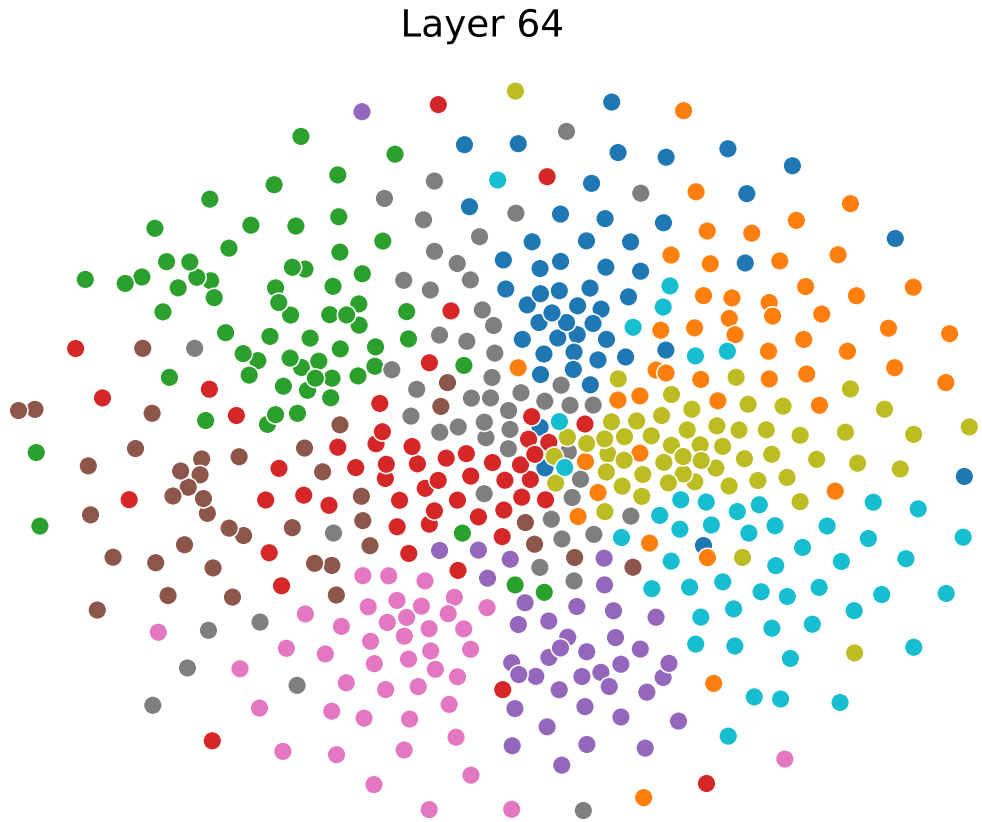
t-SNE for a home-baked ResNet (no pooling, 66 layers) CIFAR10



t-SNE for a home-baked ResNet (no pooling, 66 layers) CIFAR10

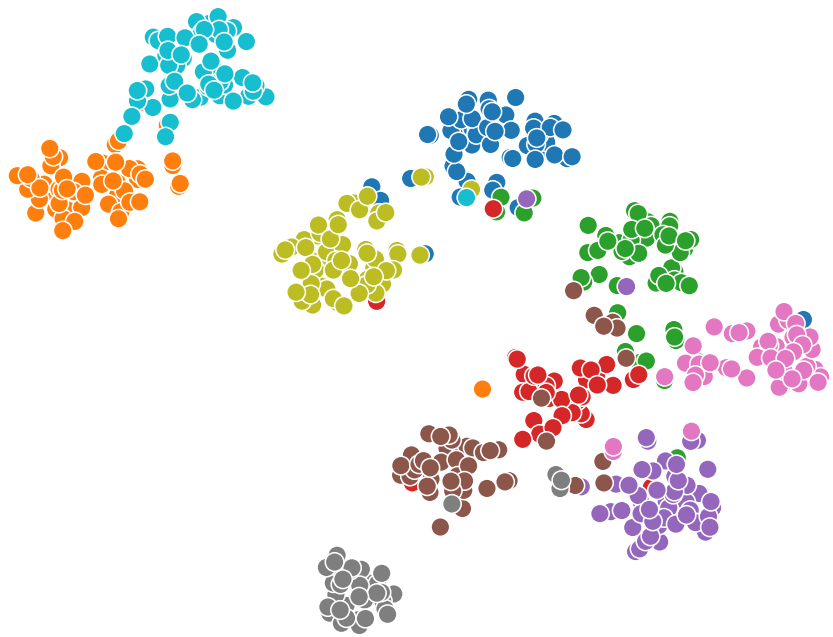


t-SNE for a home-baked ResNet (no pooling, 66 layers) CIFAR10



t-SNE for a home-baked ResNet (no pooling, 66 layers) CIFAR10

Layer 65



t-SNE for a home-baked ResNet (no pooling, 66 layers) CIFAR10

References

F. Fleuret. **Predicting the dynamics of 2d objects with a deep residual network**. CoRR, abs/1610.04032, 2016.

K. He, X. Zhang, S. Ren, and J. Sun. **Deep residual learning for image recognition**. CoRR, abs/1512.03385, 2015.

A. Krizhevsky, I. Sutskever, and G. Hinton. **Imagenet classification with deep convolutional neural networks**. In Neural Information Processing Systems (NIPS), 2012.

Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. **Gradient-based learning applied to document recognition**. Proceedings of the IEEE, 86(11):2278–2324, 1998.

L. van der Maaten and G. Hinton. **Visualizing high-dimensional data using t-SNE**. Journal of Machine Learning Research (JMLR), 9:2579–2605, 2008.

J. Yosinski, J. Clune, A. Nguyen, T. Fuchs, and H. Lipson. **Understanding neural networks through deep visualization**. In Deep Learning Workshop, International Conference on Machine Learning (WS/ICML), 2015.

Deep learning

9.3. Visualizing the processing in the input

François Fleuret

https://fleuret.org/dlc/

Occlusion sensitivity

Another approach to understanding the functioning of a network is to look at the behavior of the network "around" an image.

For instance, we can get a simple estimate of the importance of a part of the input image for a given output by computing the difference between:

1. the value of that output on the original image, and
2. the value of the same output with that part occluded.

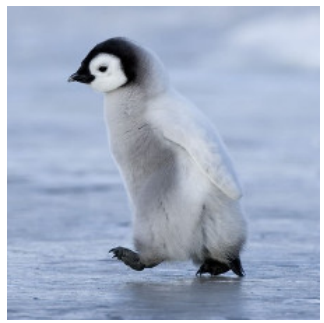
This is computationally intensive since it requires as many forward passes as there are locations of the occlusion mask, ideally the number of pixels.

Original images

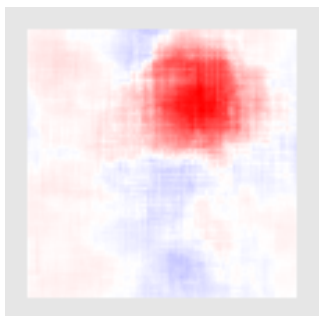
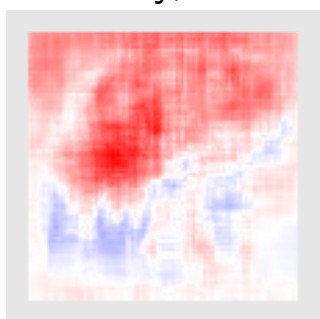
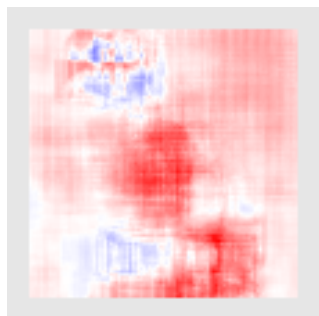
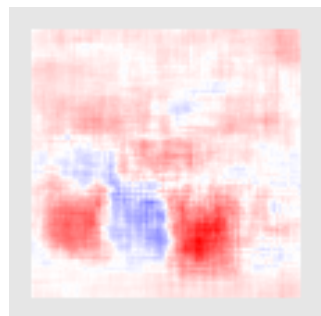
Occlusion mask 32×32

Original images



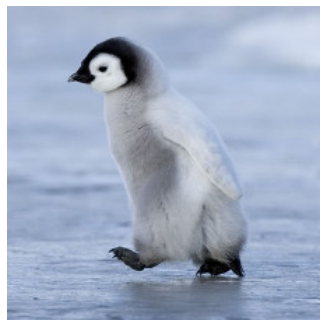



Occlusion sensitivity, mask 32×32, stride of 2, AlexNet

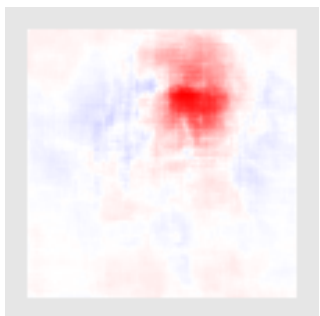
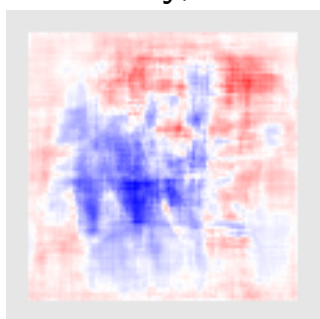
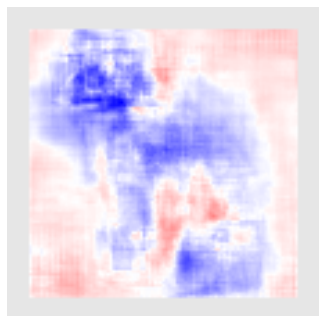
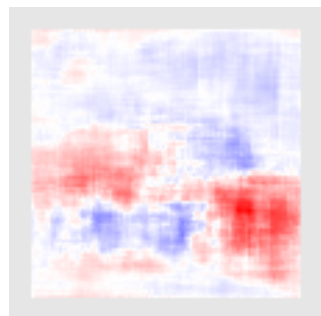





Original images






Occlusion sensitivity, mask 32×32, stride of 2, VGG19

Saliency maps

An alternative is to compute the gradient of an output with respect to the input (Erhan et al., 2009; Simonyan et al., 2013), e.g.

$$\nabla_{|x} f_c(x; w)$$

where $|x$ stresses that the gradient is computed with respect to the input x and not as usual with respect to the parameters w.

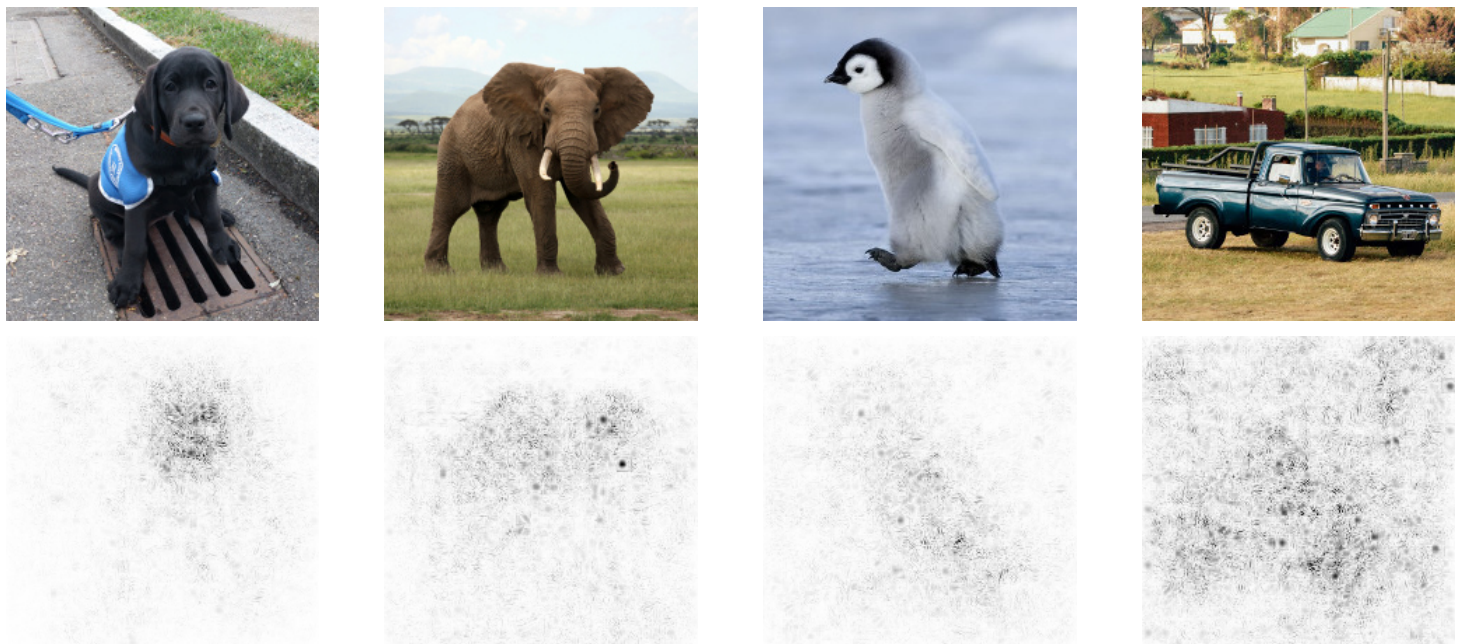
This can be implemented by specifying that we need the gradient with respect to the input.

Using `torch.autograd.grad` to compute the gradient w.r.t. the input image instead of `torch.autograd.backward` has the advantage of not changing the model's parameter gradients.

```
input.requires_grad_()
output = model(input)
grad_input, = torch.autograd.grad(output[0, c], input)
```

Note that since `torch.autograd.grad` computes the gradient of a function with possibly multiple inputs, the returned result is a tuple.

The resulting maps are quite noisy. For instance with AlexNet:



This is due to the local irregularity of the network's response as a function of the input.

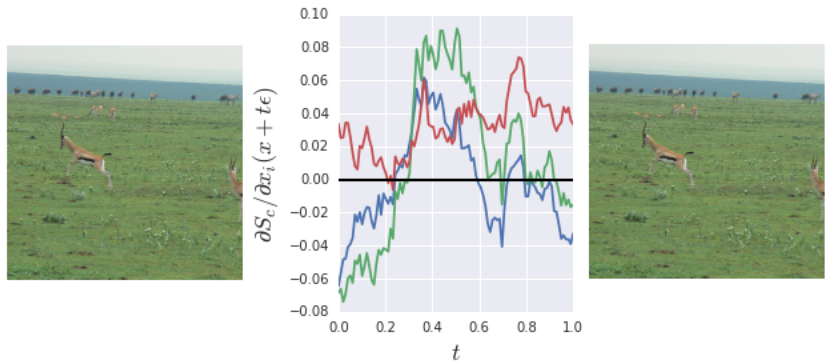


Figure 2. The partial derivative of S_c with respect to the RGB values of a single pixel as a fraction of the maximum entry in the gradient vector, $\max_i \frac{\partial S_c}{\partial x_i}(t)$, (middle plot) as one slowly moves away from a baseline image x (left plot) to a fixed location $x + \epsilon$ (right plot). ϵ is one random sample from $\mathcal{N}(0,\, 0.01^2)$. The final image $(x + \epsilon)$ is indistinguishable to a human from the origin image x.

(Smilkov et al., 2017)

Smilkov et al. (2017) proposed to smooth the gradient with respect to the input image by averaging over slightly perturbed versions of the latter.

$$\tilde{\nabla}_{|x} f_y(x; w) = \frac{1}{N} \sum_{n=1}^{N} \nabla_{|x} f_y(x + \epsilon_n; w)$$

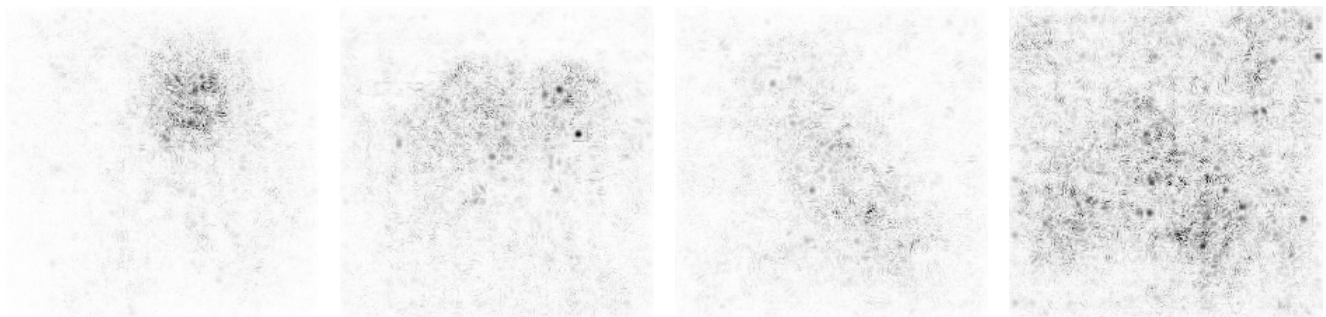
where $\epsilon_1, \ldots, \epsilon_N$ are i.i.d of distribution $\mathcal{N}(0, \sigma^2 \mathbf{I})$, and σ is a fraction of the gap Δ between the maximum and the minimum of the pixel values.

A simple version of this "SmoothGrad" approach can be implemented as follows

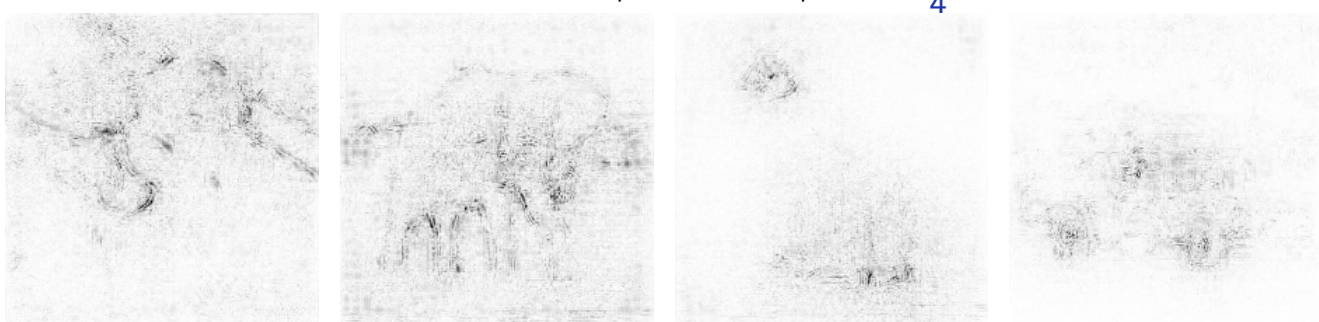
```
std = std_fraction * (img.max() - img.min())
acc_grad = img.new_zeros(img.size())

for q in range(nb_smooth): # This should be done with mini-batches ...
    noisy_input = img + img.new(img.size()).normal_(0, std)
    noisy_input.requires_grad_()
    output = model(noisy_input)
    grad_input, = torch.autograd.grad(output[0, c], noisy_input)
    acc_grad += grad_input

acc_grad = acc_grad.abs().sum(1) # sum across channels
```

Original images

Gradient, AlexNet

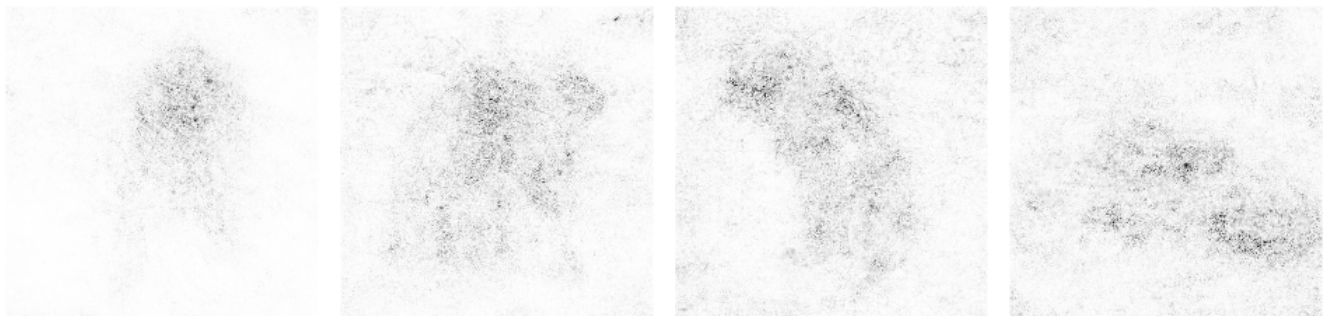


SmoothGrad, AlexNet, $\sigma = \frac{\Delta}{4}$

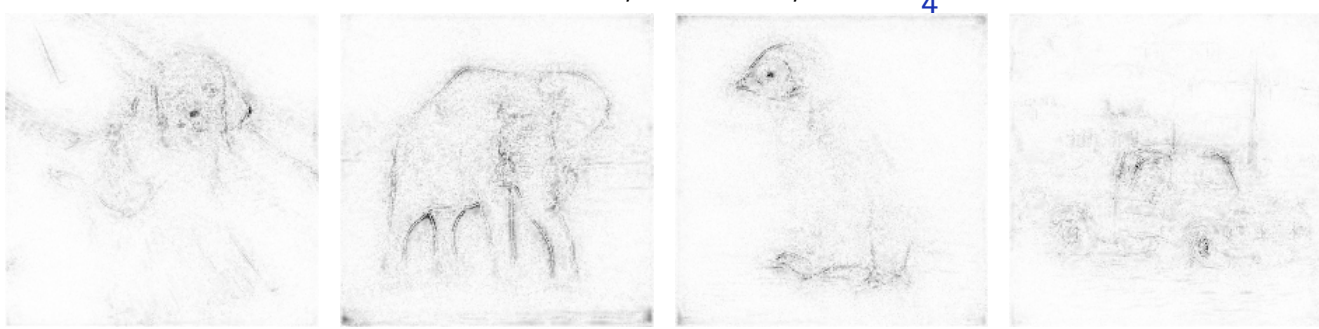


Original images

Gradient, VGG19



SmoothGrad, VGG19, $\sigma = \frac{\Delta}{4}$



Deconvolution and guided back-propagation

Zeiler and Fergus (2014) proposed to invert the processing flow of a convolutional network by constructing a corresponding **deconvolutional network** to compute the "activating pattern" of a sample.

As they point out, the resulting processing is identical to a standard backward pass, except when going through the ReLU layers.

Remember that if s is one of the input to a ReLU layer, and x the corresponding output, we have for the forward pass

$$x = \max(0, s),$$

and for the backward

$$\frac{\partial \ell}{\partial s} = \mathbf{1}_{\{s>0\}} \frac{\partial \ell}{\partial x}.$$

Zeiler and Fergus's deconvolution can be seen as a backward pass where we propagate back through ReLU layers the quantity

$$\max\left(0, \frac{\partial \ell}{\partial x}\right) = \mathbf{1}_{\{\frac{\partial \ell}{\partial x} > 0\}} \frac{\partial \ell}{\partial x},$$

instead of the usual

$$\frac{\partial \ell}{\partial s} = \mathbf{1}_{\{s>0\}} \frac{\partial \ell}{\partial x}.$$

This quantity is positive for units whose output has a positive contribution to the response, kills the others, and is not modulated by the pre-layer activation s.

Springenberg et al. (2014) improved upon the deconvolution with the **guided back-propagation**, which aims at the best of both worlds: Discarding structures which would not contribute positively to the final response, and discarding structures which are not already present.

It back-propagates through the ReLU layers the quantity

$$\mathbf{1}_{\{s>0\}}\mathbf{1}_{\left\{\frac{\partial \ell}{\partial x}>0\right\}}\frac{\partial \ell}{\partial x}$$

which keeps only units which have a positive contribution and activation.

So these three visualization methods differ only in the quantities propagated through ReLU layers during the back-pass:

- back-propagation (Erhan et al., 2009; Simonyan et al., 2013):

$$\mathbf{1}_{\{s>0\}} \frac{\partial \ell}{\partial x},$$

- deconvolution (Zeiler and Fergus, 2014):

$$\mathbf{1}_{\{\frac{\partial \ell}{\partial x}>0\}} \frac{\partial \ell}{\partial x},$$

- guided back-propagation (Springenberg et al., 2014):

$$\mathbf{1}_{\{s>0\}}\mathbf{1}_{\{\frac{\partial \ell}{\partial x}>0\}} \frac{\partial \ell}{\partial x}.$$

These procedures can be implemented simply in PyTorch by changing the `nn.ReLU`'s backward pass.

The class `nn.Module` provides methods to register "hook" functions that are called during the forward or the backward pass, and can implement a different computation for the latter.

For instance

```
>>> x = torch.tensor([ 1.23, -4.56 ])
>>> m = nn.ReLU()
>>> m(x)
tensor([ 1.2300,  0.0000])

>>> def my_hook(m, input, output):
...     print(str(m) + ' got ' + str(input[0].size()))
...
>>> handle = m.register_forward_hook(my_hook)
>>> m(x)
ReLU() got torch.Size([2])
tensor([ 1.2300,  0.0000])

>>> handle.remove()
>>> m(x)
tensor([ 1.2300,  0.0000])
```

Using hooks, we can implement the deconvolution as follows:

```
def relu_backward_deconv_hook(module, grad_input, grad_output):
    return F.relu(grad_output[0]),

def equip_model_deconv(model):
    for m in model.modules():
        if isinstance(m, nn.ReLU):
            m.register_backward_hook(relu_backward_deconv_hook)
```

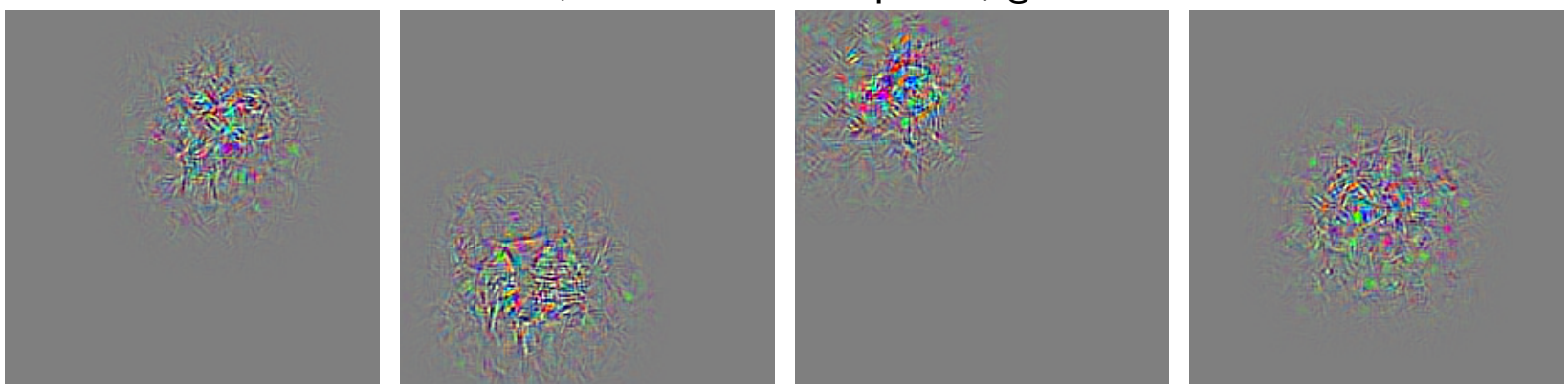
```
def grad_view(model, image_name):
    to_tensor = transforms.ToTensor()
    img = to_tensor(PIL.Image.open(image_name))
    img = 0.5 + 0.5 * (img - img.mean()) / img.std()

    model.to(device)
    img = img.to(device)

    input = img.view(1, img.size(0), img.size(1), img.size(2)).requires_grad_()
    output = model(input)
    result, = torch.autograd.grad(output.max(), input)

    result = result / result.max() + 0.5

    return result

model = models.vgg16(pretrained = True)
model.eval()
model = model.features
equip_model_deconv(model)
result = grad_view(model, 'blacklab.jpg')
utils.save_image(result, 'blacklab-vgg16-deconv.png')
```

The code is the same for the guided back-propagation, except the hooks themselves:

```
def relu_forward_gbackprop_hook(module, input, output):
    module.input_kept = input[0]

def relu_backward_gbackprop_hook(module, grad_input, grad_output):
    return F.relu(grad_output[0]) * F.relu(module.input_kept).sign(),

def equip_model_gbackprop(model):
    for m in model.modules():
        if isinstance(m, nn.ReLU):
            m.register_forward_hook(relu_forward_gbackprop_hook)
            m.register_backward_hook(relu_backward_gbackprop_hook)
```

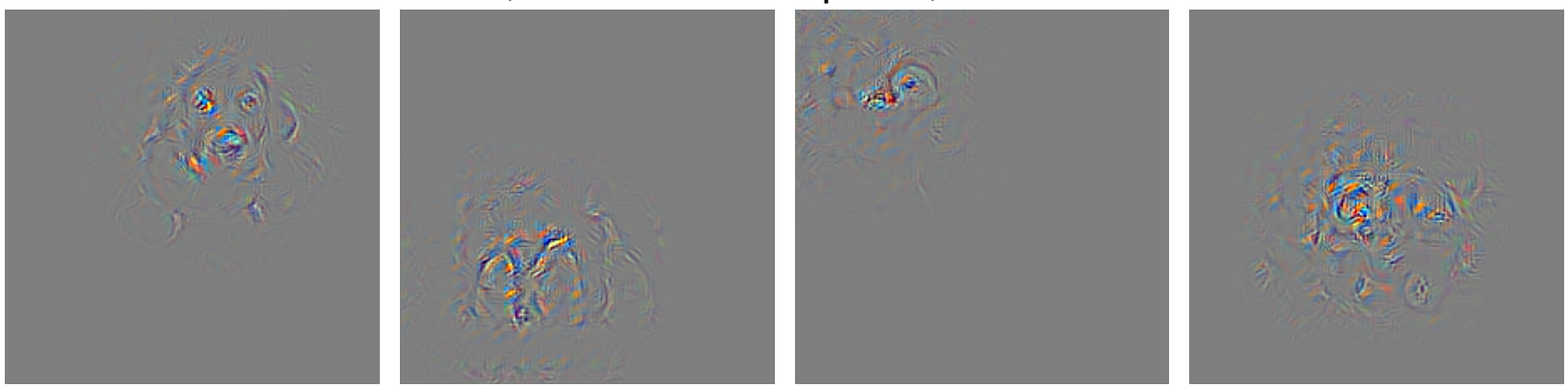
Original images

AlexNet, max feature response, gradient



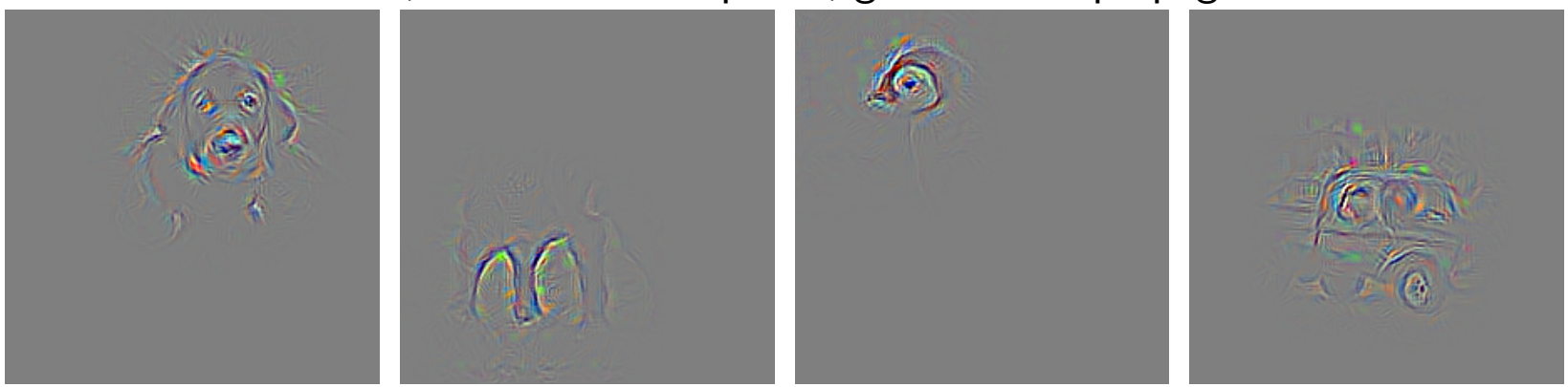
Original images

AlexNet, max feature response, deconvolution



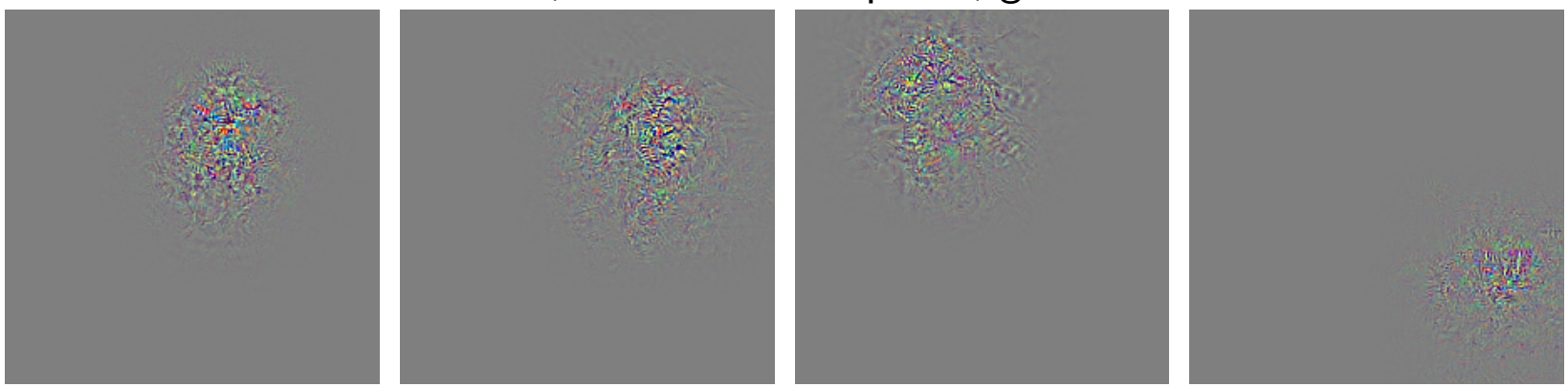
Original images

AlexNet, max feature response, guided back-propagation



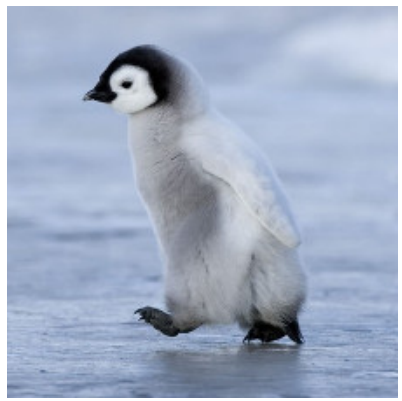
Original images

VGG16, max feature response, gradient

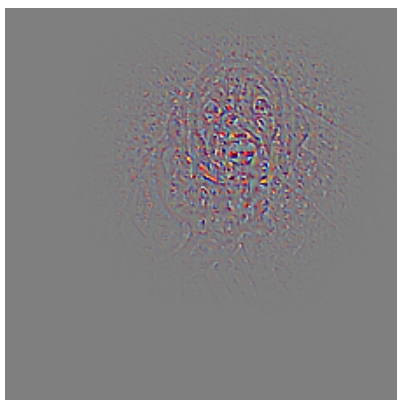
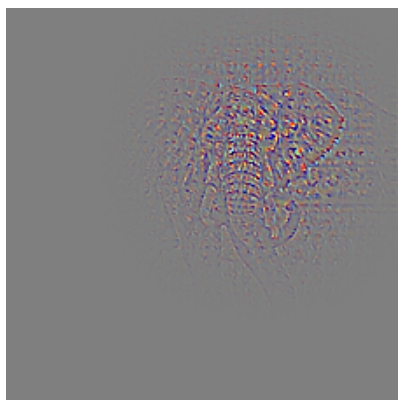
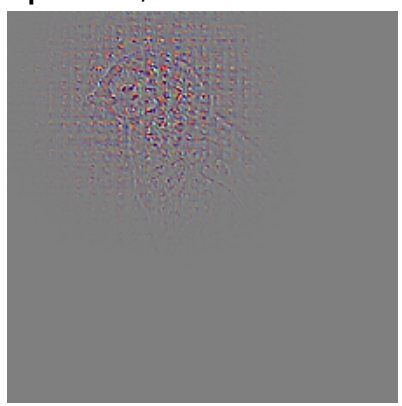
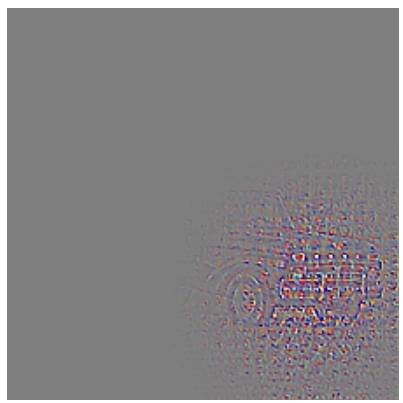


Original images






VGG16, max feature response, deconvolution

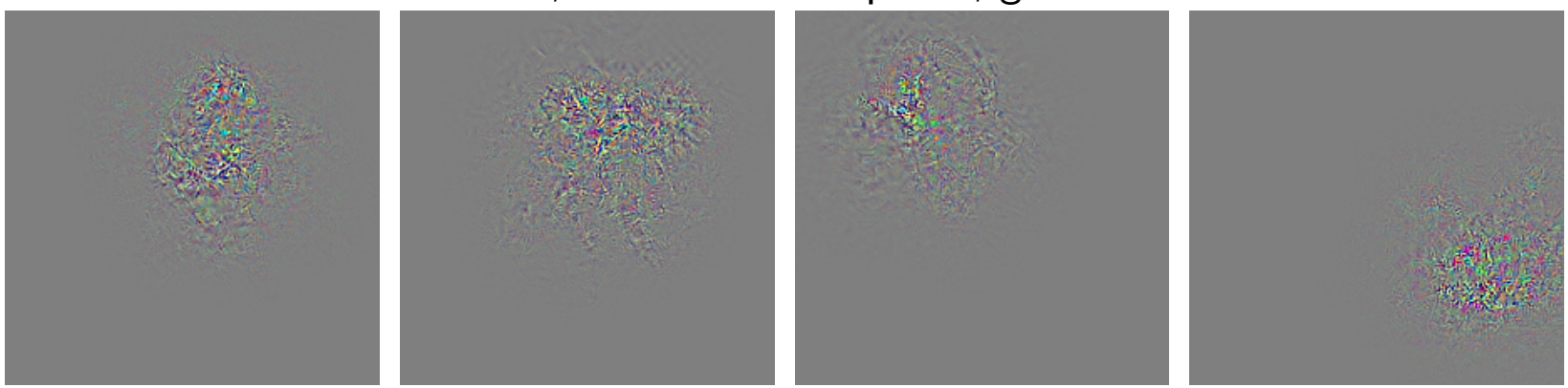





Original images

VGG16, max feature response, guided back-propagation

Original images

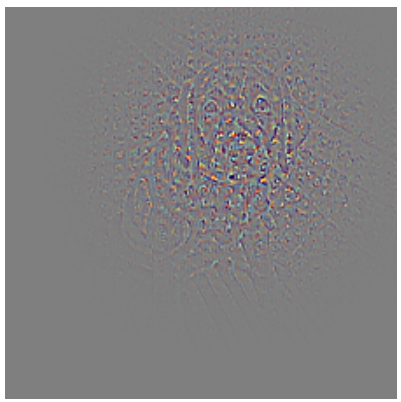
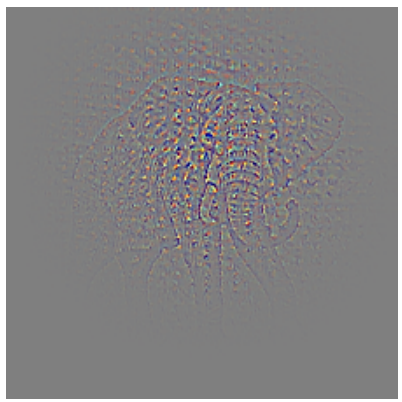
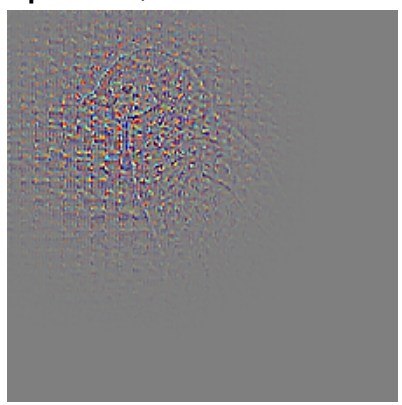
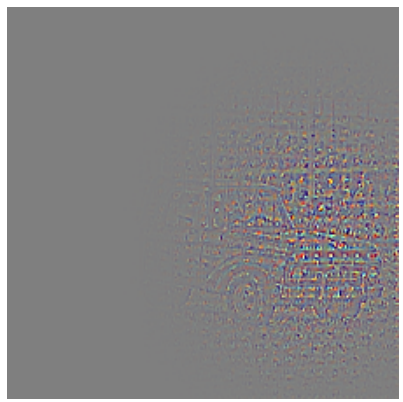
VGG19, max feature response, gradient



Original images



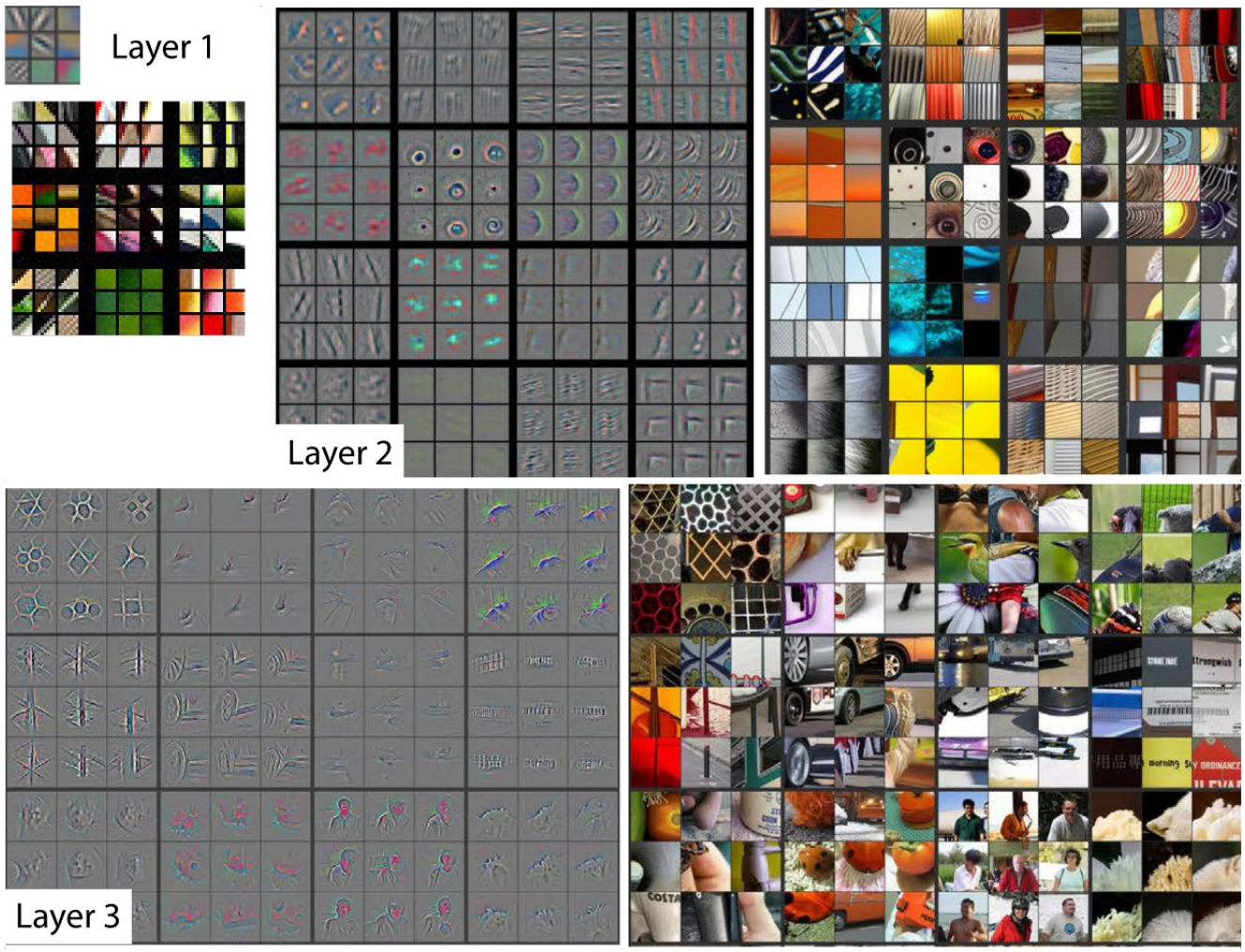

VGG19, max feature response, deconvolution

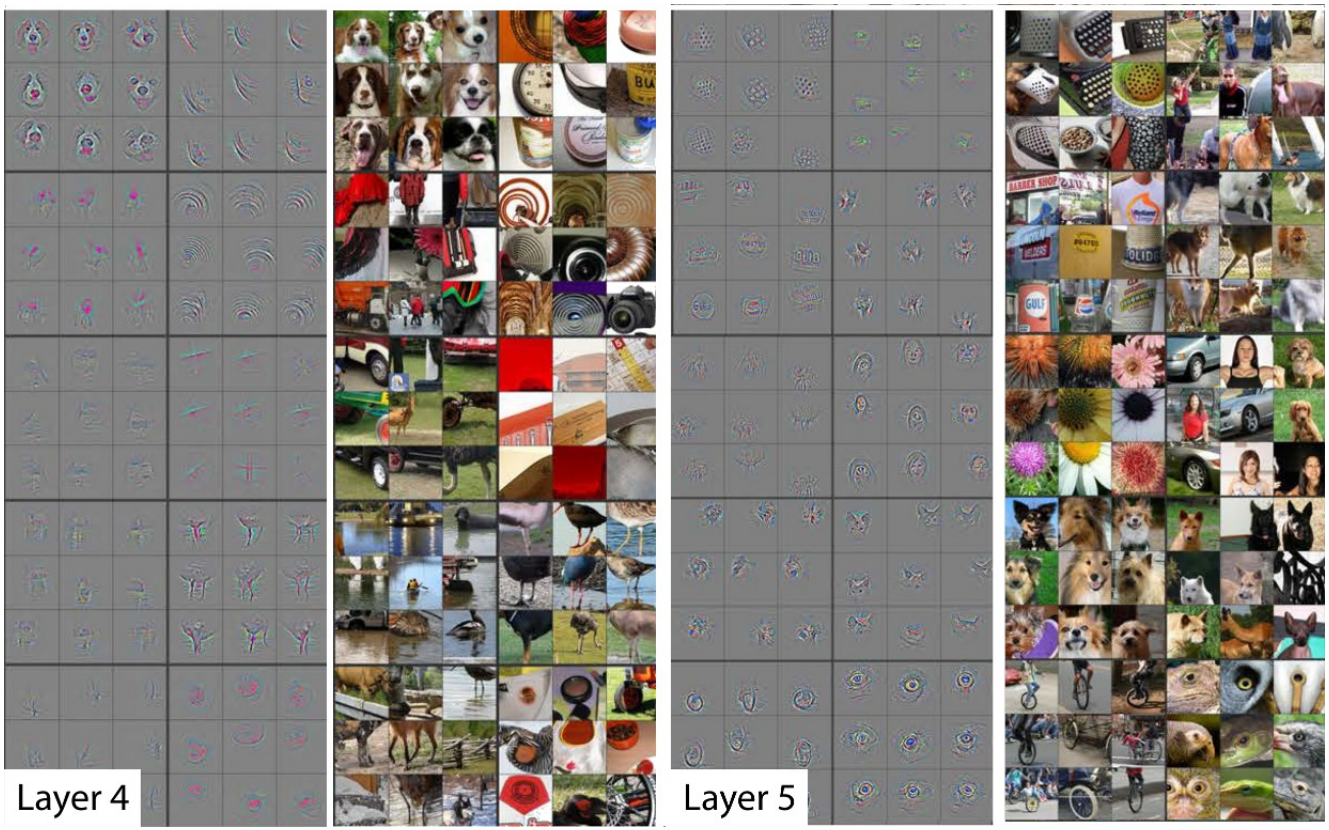
Original images

VGG19, max feature response, guided back-propagation

Experiments with an AlexNet-like network. Original images + deconvolution (or filters) for the top-9 activations for channels picked randomly.



(Zeiler and Fergus, 2014)



(Zeiler and Fergus, 2014)

Grad-CAM

Gradient-weighted Class Activation Mapping (Grad-CAM) proposed by Selvaraju et al. (2016) visualizes the importance of the input sub-parts according to the activations in a specific layer.

It computes a sum of the activations weighted by the average gradient of the output of interest w.r.t. individual channels.

Formally, let $k \in \{1, \ldots, C\}$ be a channel number, $A^k \in \mathbb{R}^{H \times W}$ the output feature map k of the selected layer, c a class number, and y^c the network's logit for that class.

The channel weights are

$$\alpha_k^c = \frac{1}{HW} \sum_{i=1}^{H} \sum_{j=1}^{W} \frac{\partial y^c}{\partial A_{i,j}^k}.$$

And the final localization map is

$$L_{\text{Grad-CAM}}^c = \text{ReLU}\left(\sum_{k=1}^{C} \alpha_k^c A^k \right).$$

We are going to test it with VGG19.

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    /.../
    (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (35): ReLU(inplace=True)
    (36): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

To implement Grad-CAM, first define hooks to store the feature maps in the forward pass, and the gradient w.r.t. them in the backward:

```
def hook_store_A(module, input, output):
    module.A = output[0]

def hook_store_dydA(module, grad_input, grad_output):
    module.dydA = grad_output[0]
```

Then, load a pre-trained VGG19, and install the hooks in the last ReLU layer of the convolutional part:

```
model = torchvision.models.vgg19(pretrained = True)
model.eval()

layer = model.features[35] # Last ReLU of the conv layers

layer.register_forward_hook(hook_store_A)
layer.register_backward_hook(hook_store_dydA)
```

Load an image and make it a one sample batch:

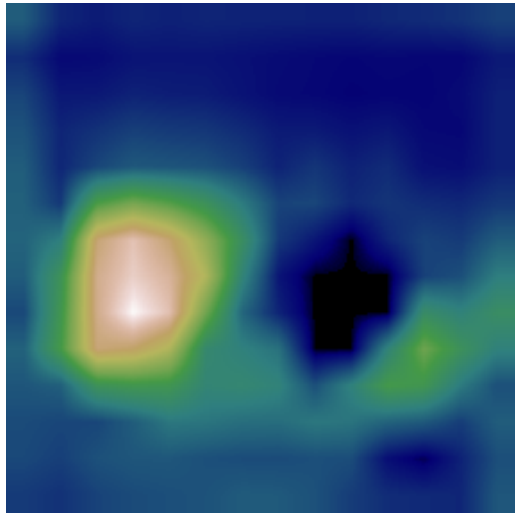
```
to_tensor = torchvision.transforms.ToTensor()
input = to_tensor(PIL.Image.open('example_images/elephant_hippo.png')).unsqueeze(0)
```

Compute the network's output, the gradient, and $L^c_{\text{Grad-CAM}}$:

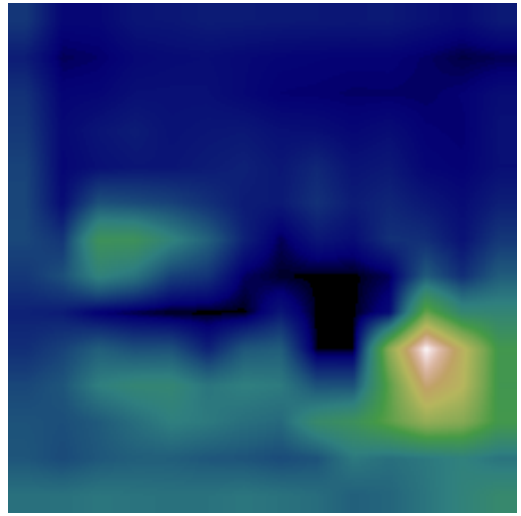
```
output = model(input)

c = 386 # African elephant
output[0, c].backward()

alpha = layer.dydA.mean((2, 3), keepdim = True)
L = torch.relu((alpha * layer.A).sum(1, keepdim = True))
```

Save it as a resized colored heat-map:

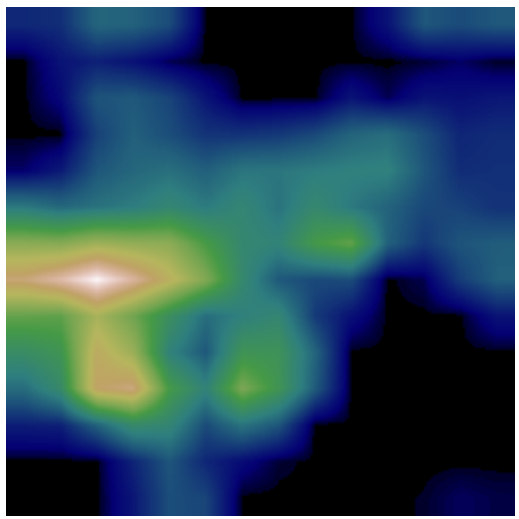
```
L = L / L.max()
L = F.interpolate(L, size = (input.size(2), input.size(3)),
                  mode = 'bilinear', align_corners = False)

l = L.view(L.size(2), L.size(3)).detach().numpy()
PIL.Image.fromarray(numpy.uint8(cm.gist_earth(l) * 255)).save('result.png')
```



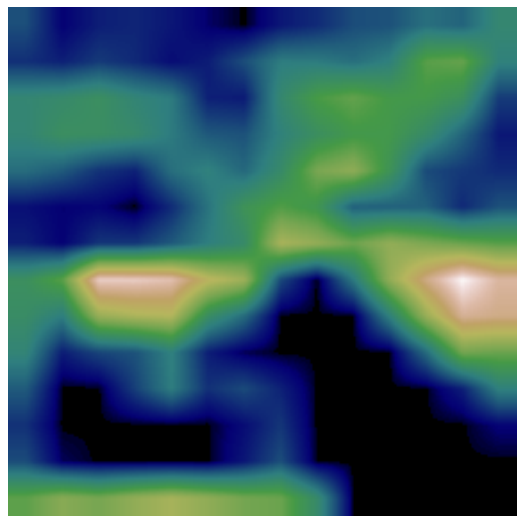
African elephant



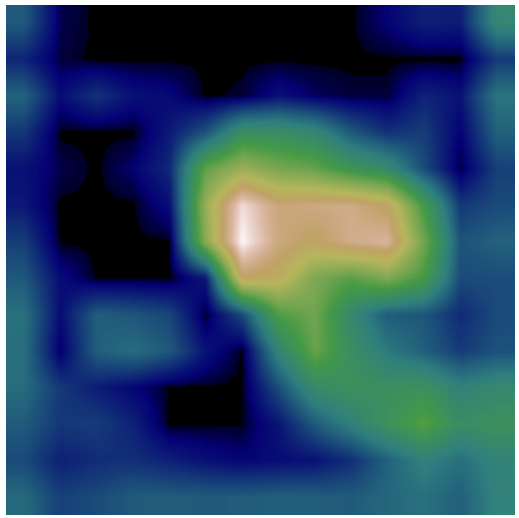
Hippopotamus



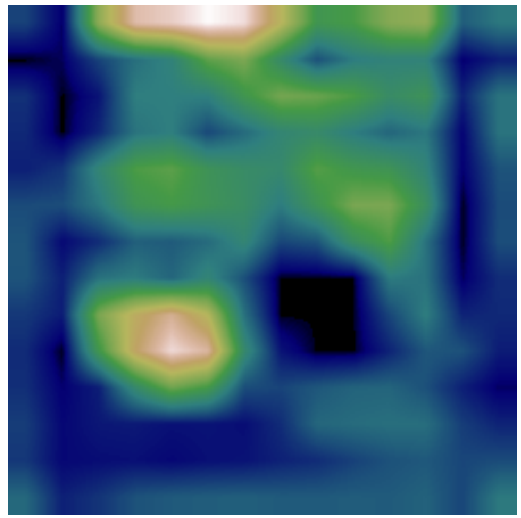
Ox



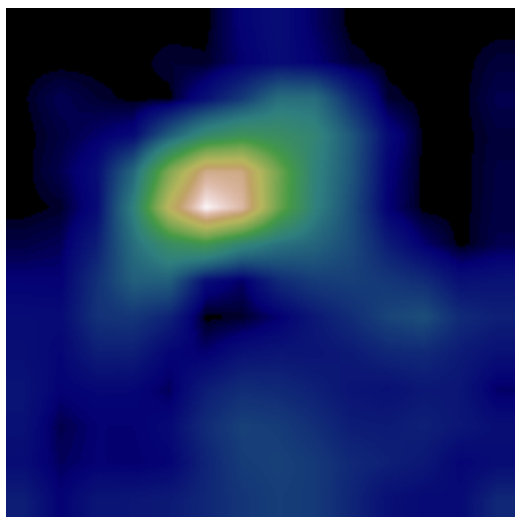
Fountain



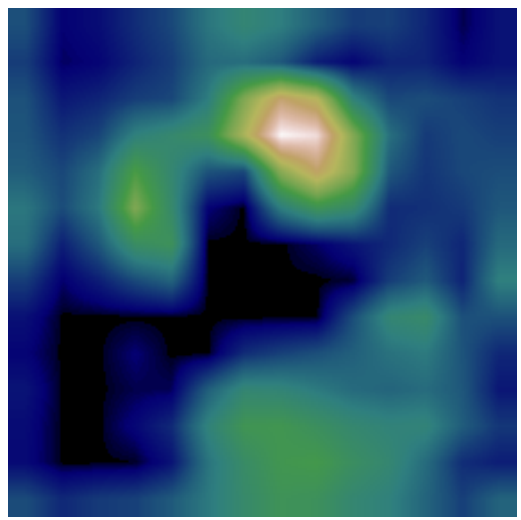
Coffee mug



Bagel



Bee



Daisy

References

D. Erhan, Y. Bengio, A. Courville, and P. Vincent. **Visualizing higher-layer features of a deep network**. Technical Report 1341, Departement IRO, Université de Montréal, 2009.

R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra. **Grad-cam: Visual explanations from deep networks via gradient-based localization**. CoRR, abs/1610.02391, 2016.

K. Simonyan, A. Vedaldi, and A. Zisserman. **Deep inside convolutional networks: Visualising image classification models and saliency maps**. CoRR, abs/1312.6034, 2013.

D. Smilkov, N. Thorat, B. Kim, F. Viegas, and M. Wattenberg. **Smoothgrad: removing noise by adding noise**. CoRR, abs/1706.03825, 2017.

J. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller. **Striving for simplicity: The all convolutional net**. CoRR, abs/1412.6806, 2014.

M. D. Zeiler and R. Fergus. **Visualizing and understanding convolutional networks**. In European Conference on Computer Vision (ECCV), 2014.

Deep learning

9.4. Optimizing inputs

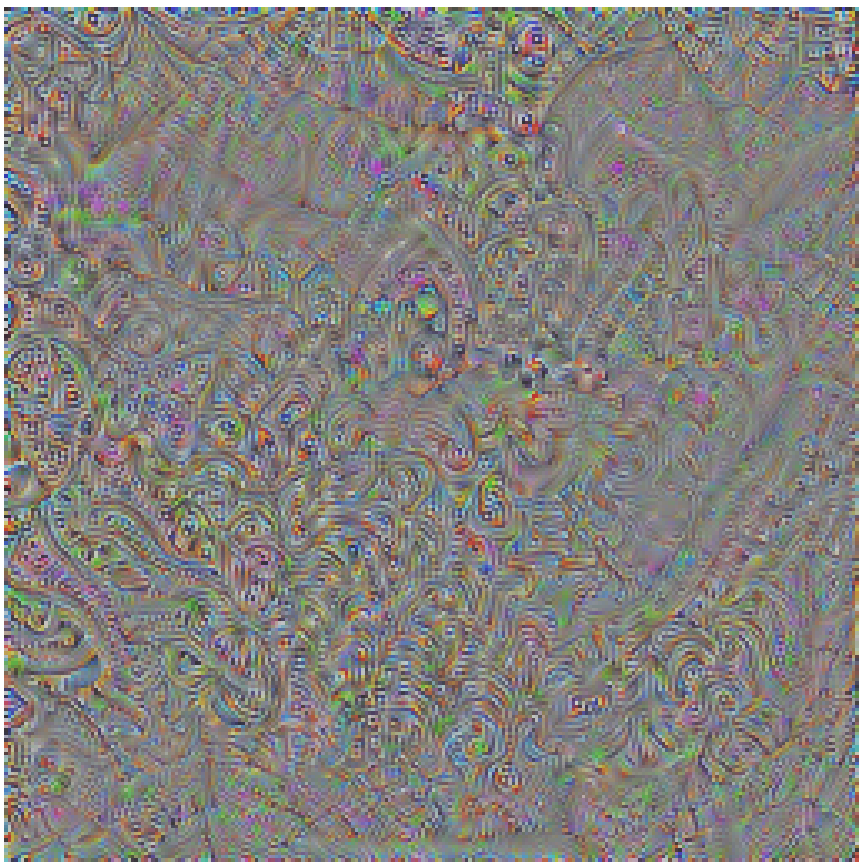
François Fleuret

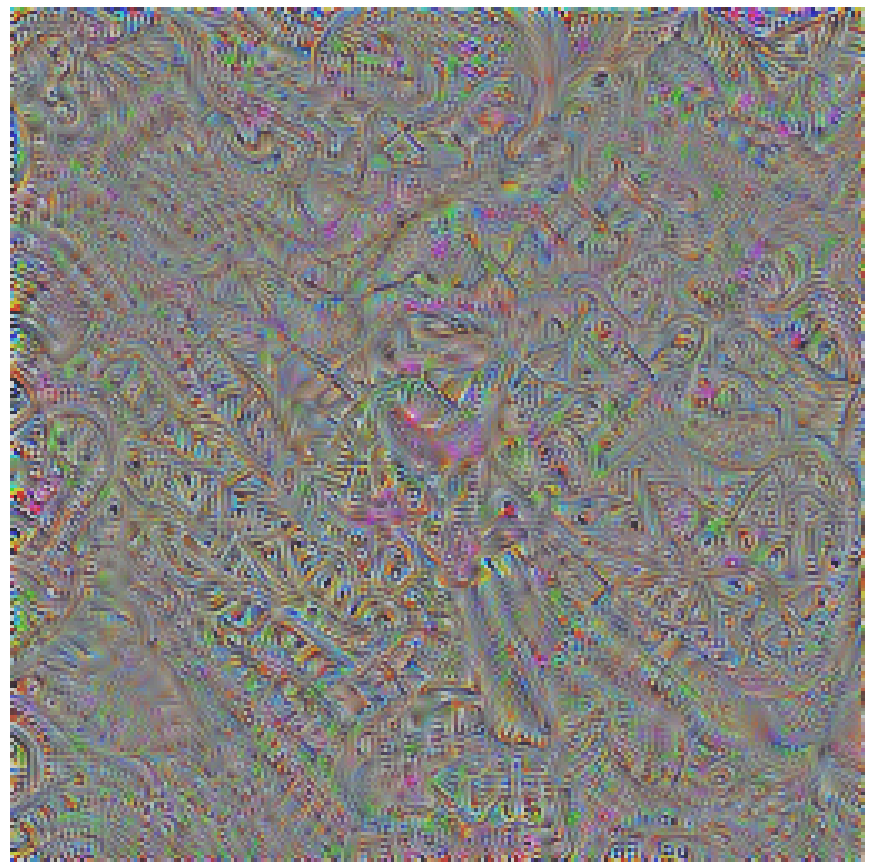
https://fleuret.org/dlc/

UNIVERSITÉ
DE GENÈVE

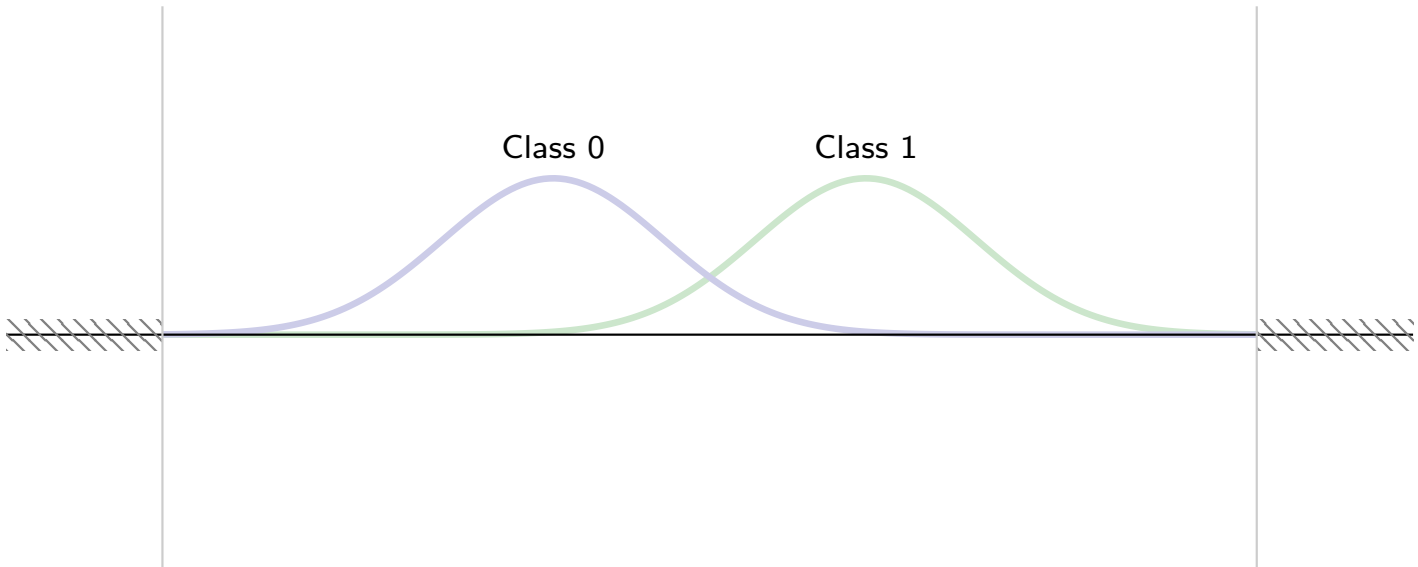
A strategy to get an intuition of the information actually encoded in the weights of a convnet consists of optimizing from scratch a sample to maximize the activation f of a chosen unit, or the sum over an activation map.

Doing so generates images with high frequencies, which tend to activate units a lot. For instance these images maximize the responses of the units "bathtub" and "lipstick" respectively (yes, this is strange, we will come back to it).

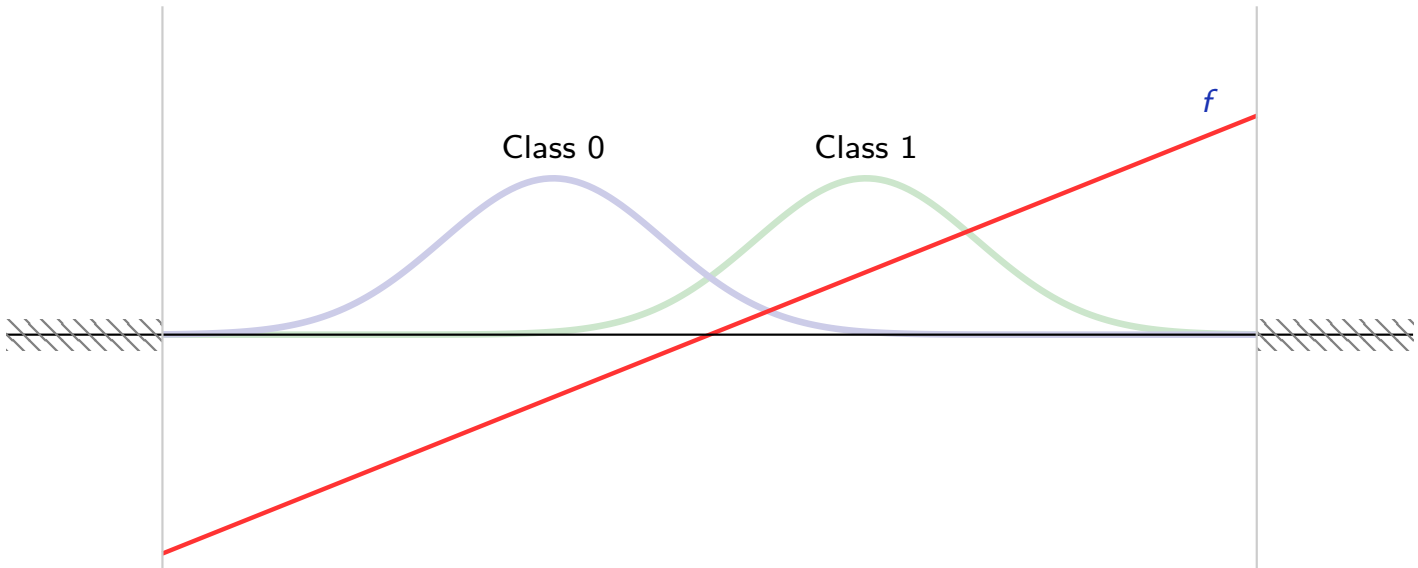




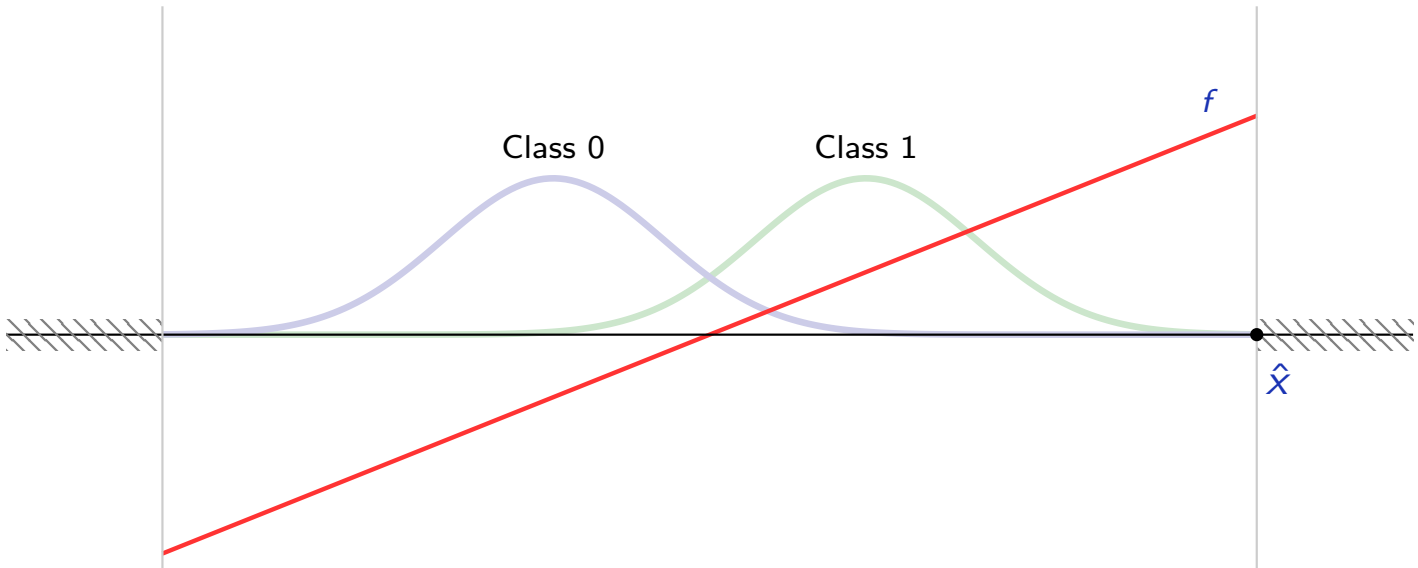
Since f is trained in a discriminative manner, a sample $\hat{x}$ maximizing it has no reason to be "realistic".



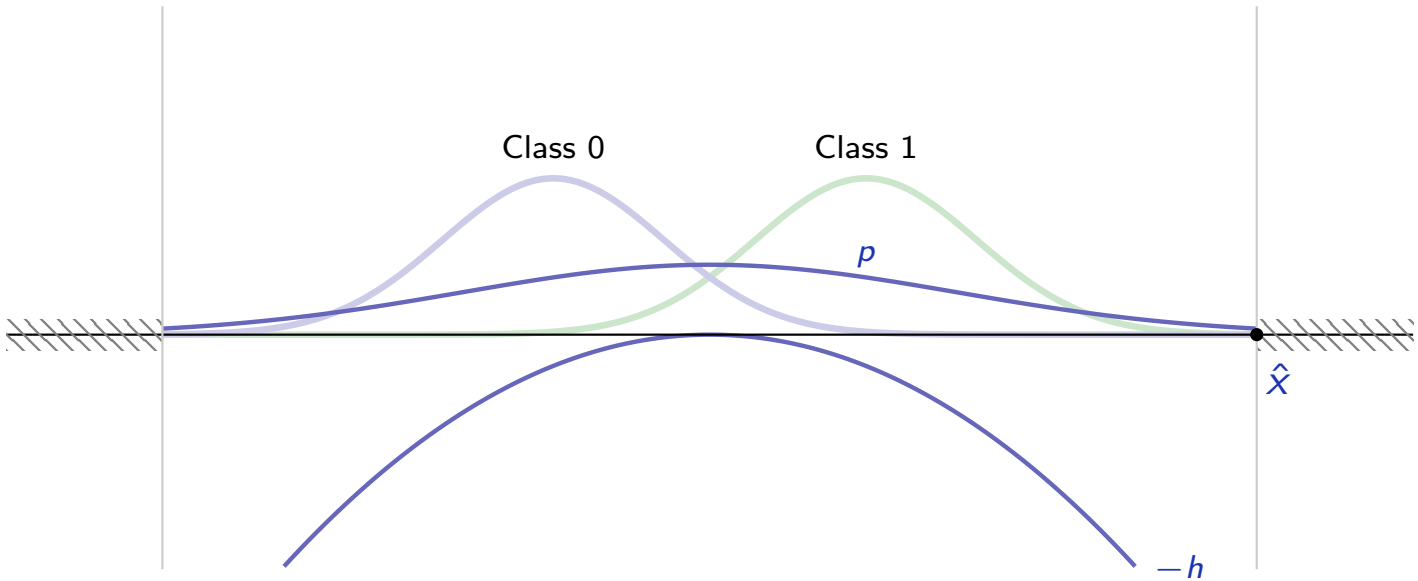
Since f is trained in a discriminative manner, a sample $\hat{x}$ maximizing it has no reason to be "realistic".



Since f is trained in a discriminative manner, a sample $\hat{x}$ maximizing it has no reason to be "realistic".



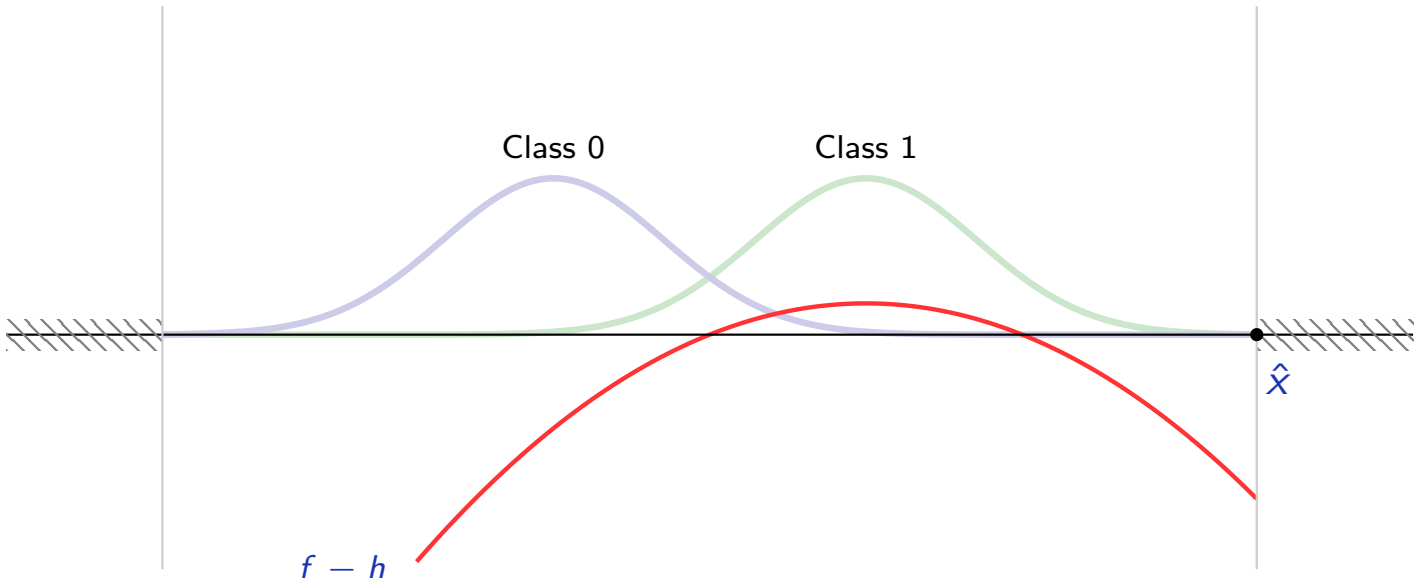
Since f is trained in a discriminative manner, a sample $\hat{x}$ maximizing it has no reason to be "realistic".



We can mitigate this by adding a penalty h corresponding to a "realistic" prior, that is compute

$$x^* = \underset{x}{\text{argmax}} \; f(x; w) - h(x)$$

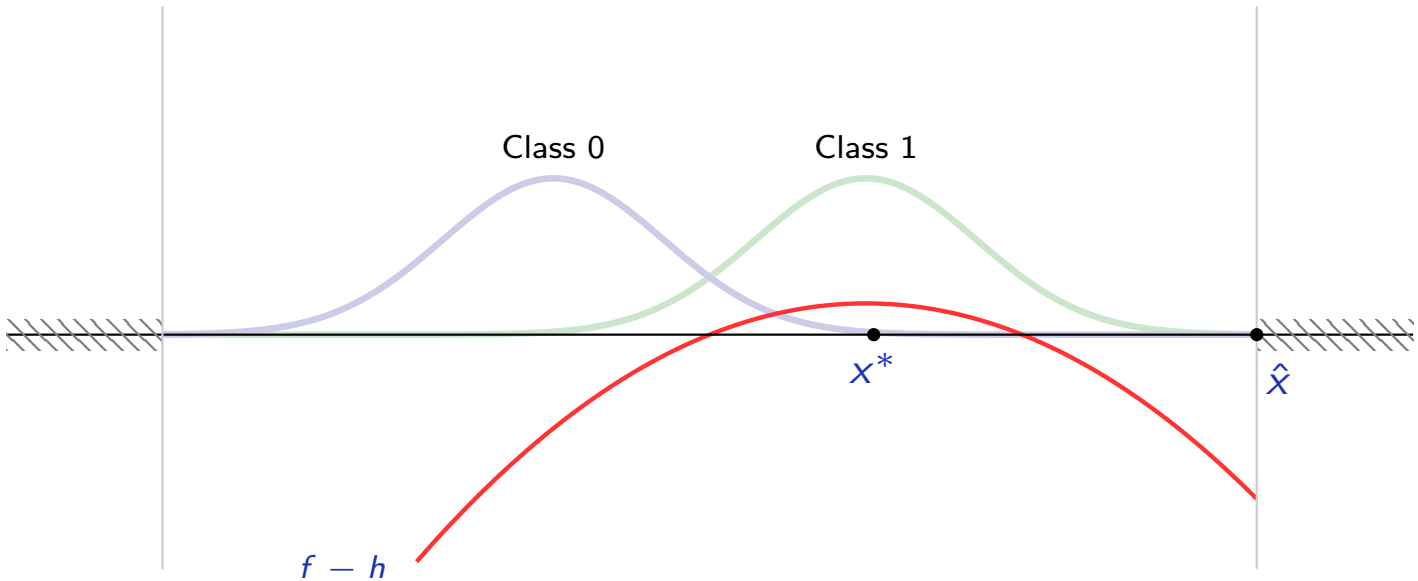
Since f is trained in a discriminative manner, a sample $\hat{x}$ maximizing it has no reason to be "realistic".



We can mitigate this by adding a penalty h corresponding to a "realistic" prior, that is compute

$$x^* = \underset{x}{\text{argmax}} \; f(x; w) - h(x)$$

Since f is trained in a discriminative manner, a sample $\hat{x}$ maximizing it has no reason to be "realistic".



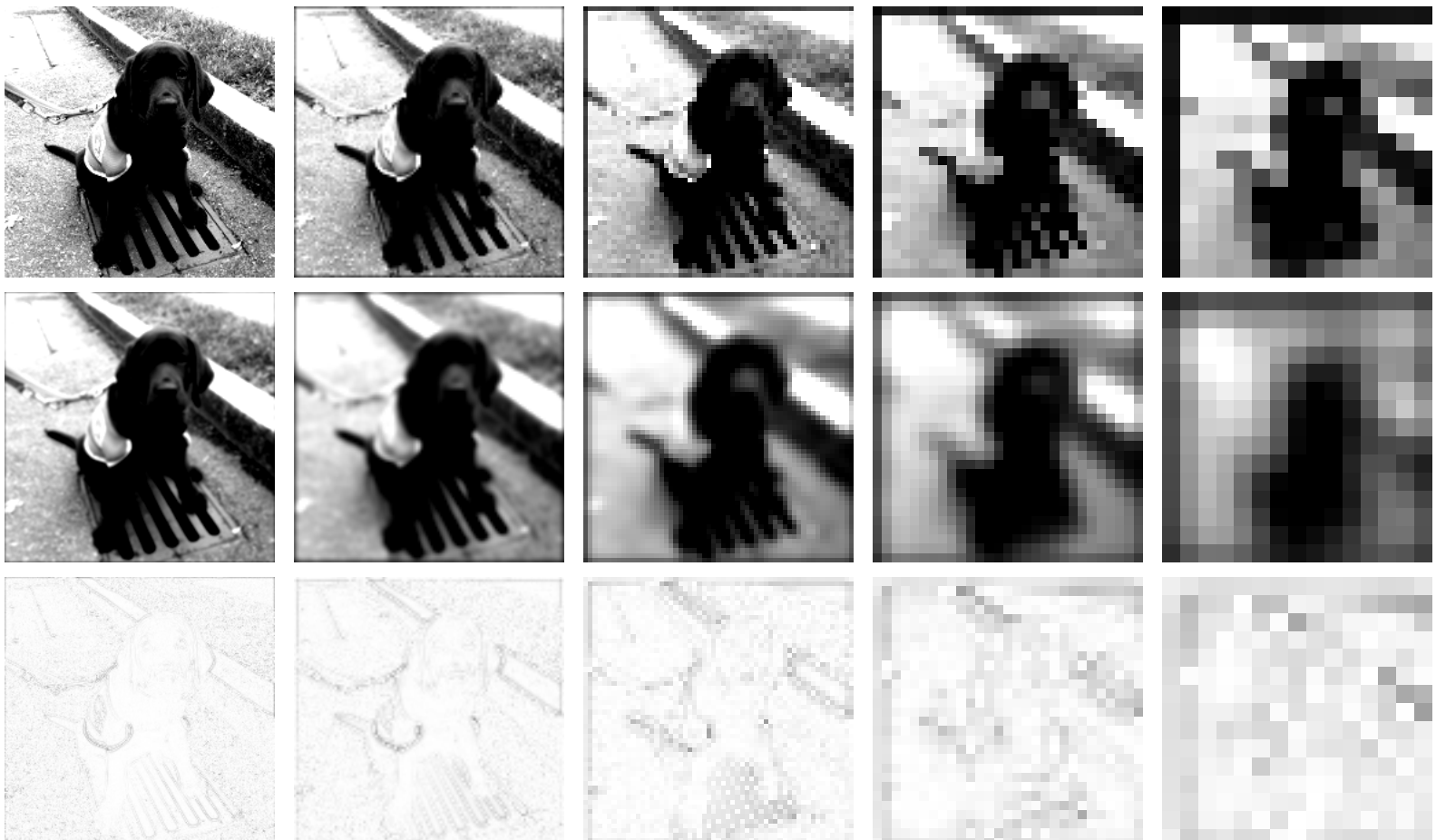
We can mitigate this by adding a penalty h corresponding to a "realistic" prior, that is compute

$$x^* = \underset{x}{\text{argmax}} \; f(x; w) - h(x)$$

by iterating a standard gradient update:

$$x_{k+1} = x_k - \eta \nabla_{|x}(h(x_k) - f(x_k; w)).$$

A reasonable h penalizes too much energy in the high frequencies by integrating edge amplitude at multiple scales.



This can be formalized as a penalty function h of the form

$$h(x) = \sum_{s \geq 0} \|\delta^s(x) - g \circledast \delta^s(x)\|^2$$

where g is a Gaussian kernel, and δ is a downscale-by-two operator.

$$h(x) = \sum_{s \geq 0} \|\delta^s(x) - g \circledast \delta^s(x)\|^2$$

We process channels as separate images, and sum across channels in the end.

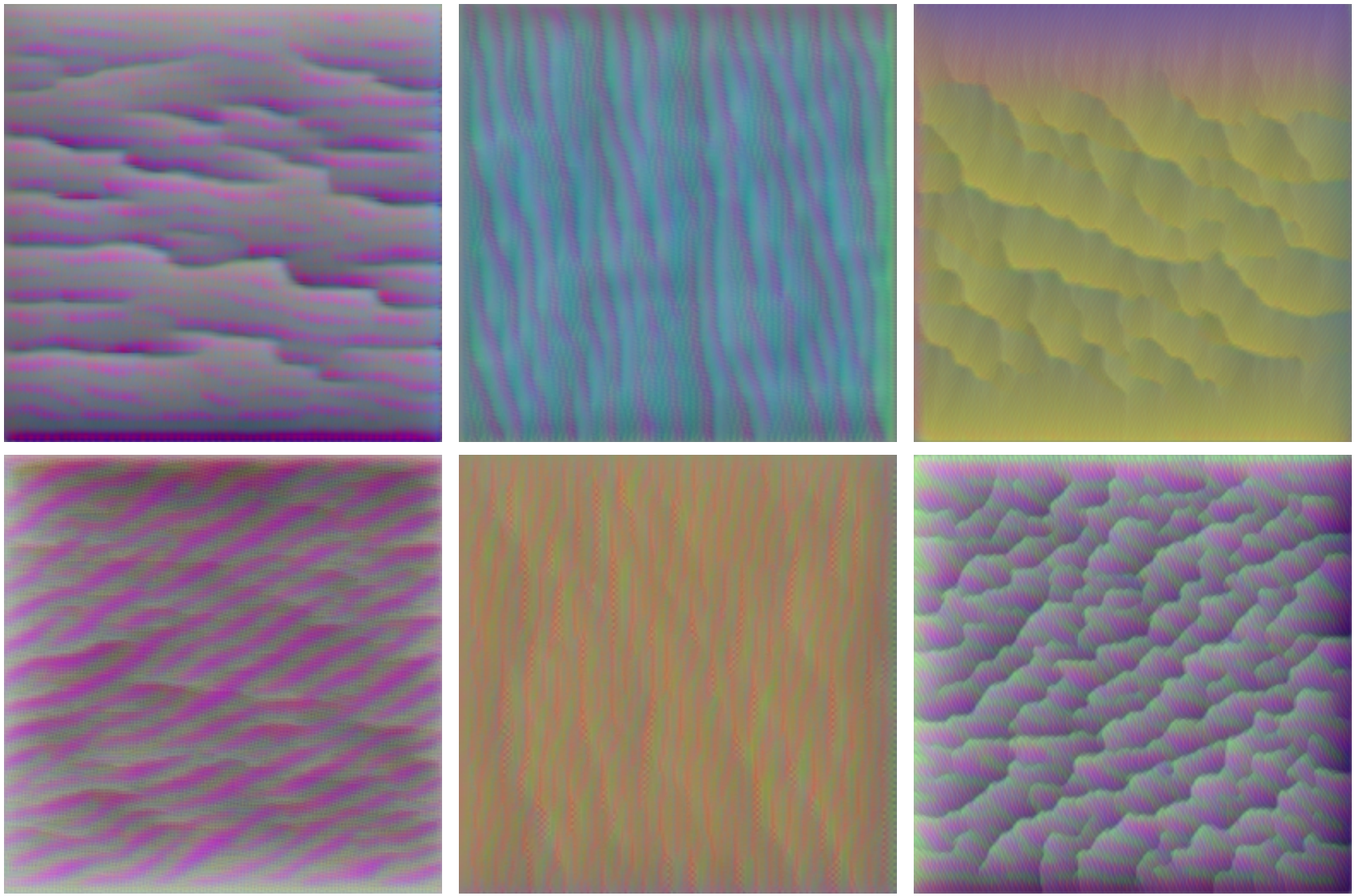
```
class MultiScaleEdgeEnergy(nn.Module):
    def __init__(self):
        super().__init__()
        k = torch.exp(- torch.tensor([[-2., -1., 0., 1., 2.]])**2 / 2)
        k = (k.t() @ k).view(1, 1, 5, 5)
        self.register_buffer('gaussian_5x5', k / k.sum())

    def forward(self, x):
        u = x.view(-1, 1, x.size(2), x.size(3))
        result = 0.0
        while min(u.size(2), u.size(3)) > 5:
            blurry  = F.conv2d(u, self.gaussian_5x5, padding = 2)
            result += (u - blurry).view(u.size(0), -1).pow(2).sum(1)
            u = F.avg_pool2d(u, kernel_size = 2, padding = 1)
        result = result.view(x.size(0), -1).sum(1)
        return result
```

Then, the optimization of the image *per se* is straightforward:

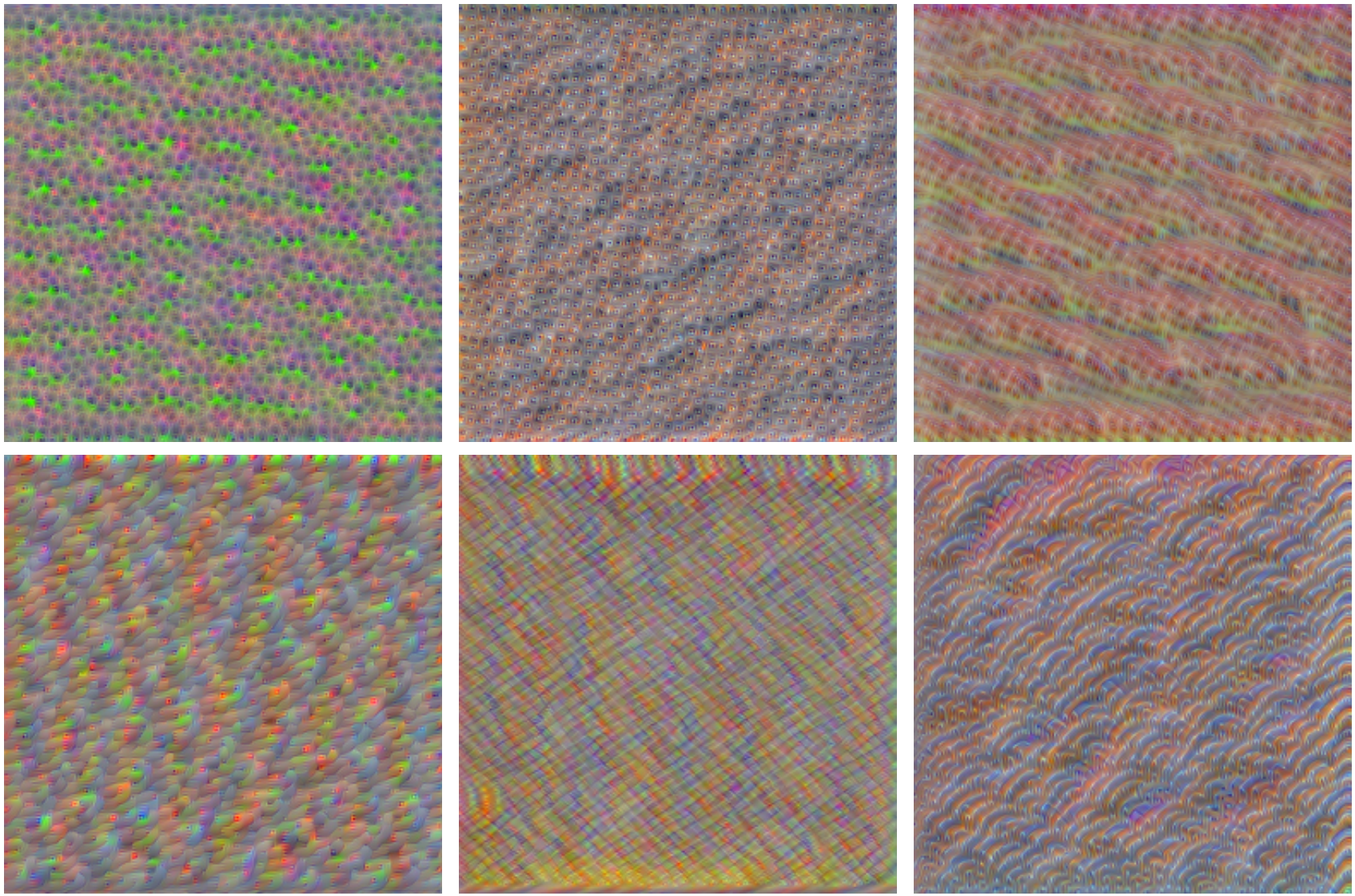
```
model = models.vgg16(pretrained = True)
model.eval()
edge_energy = MultiScaleEdgeEnergy()
input = torch.empty(1, 3, 224, 224).normal_(0, 0.01)

input.requires_grad_()
optimizer = optim.Adam([input], lr = 1e-1)

for k in range(250):
    output = model(input)
    score = edge_energy(input) - output[0, 700] # paper towel
    optimizer.zero_grad()
    score.backward()
    optimizer.step()

result = 0.5 + 0.1 * (input - input.mean()) / input.std()
torchvision.utils.save_image(result, 'dream-course-example.png')
```

(take a second to think about the beauty of autograd)

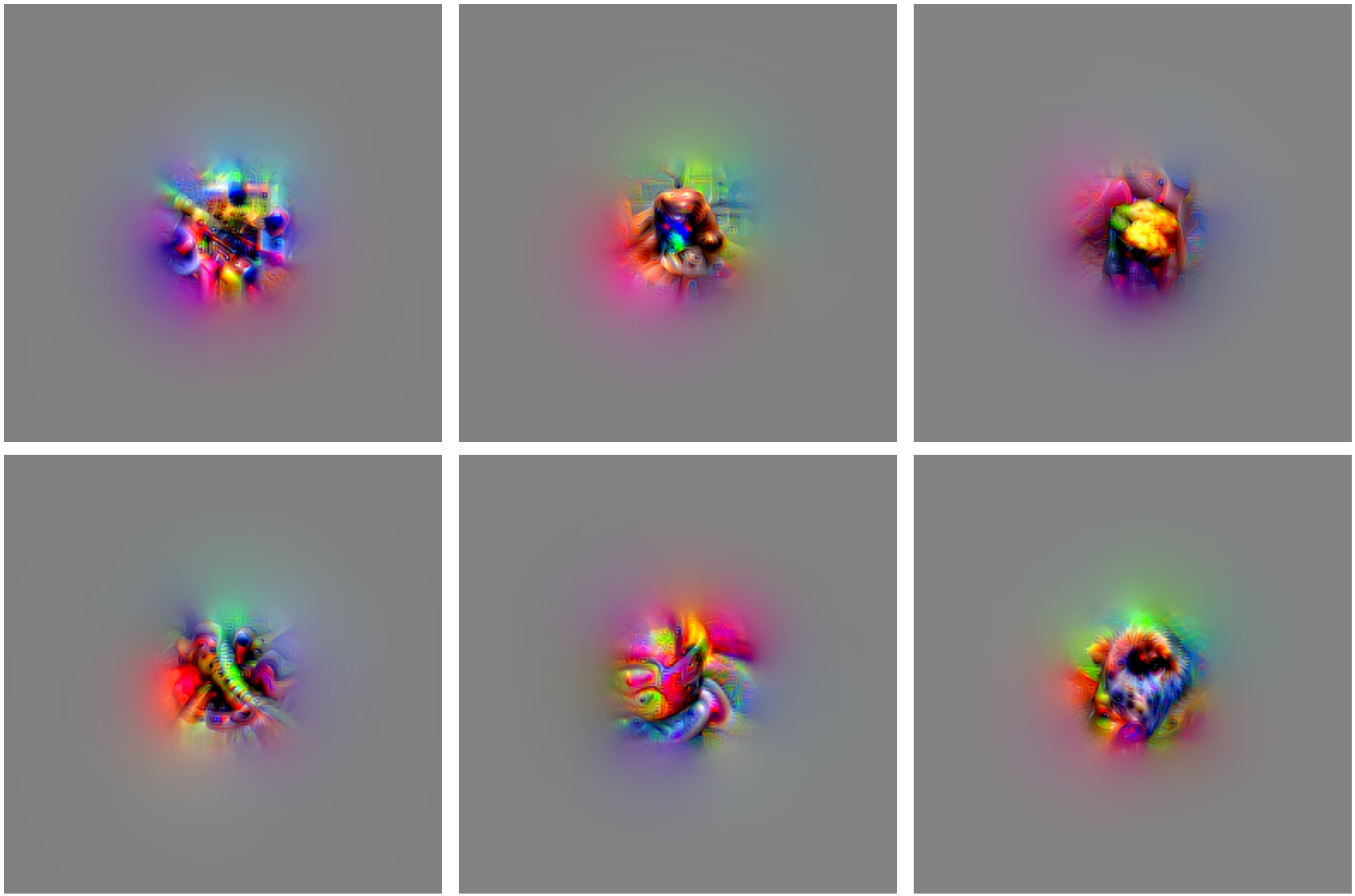
VGG16, maximizing a channel of the 4th convolution layer



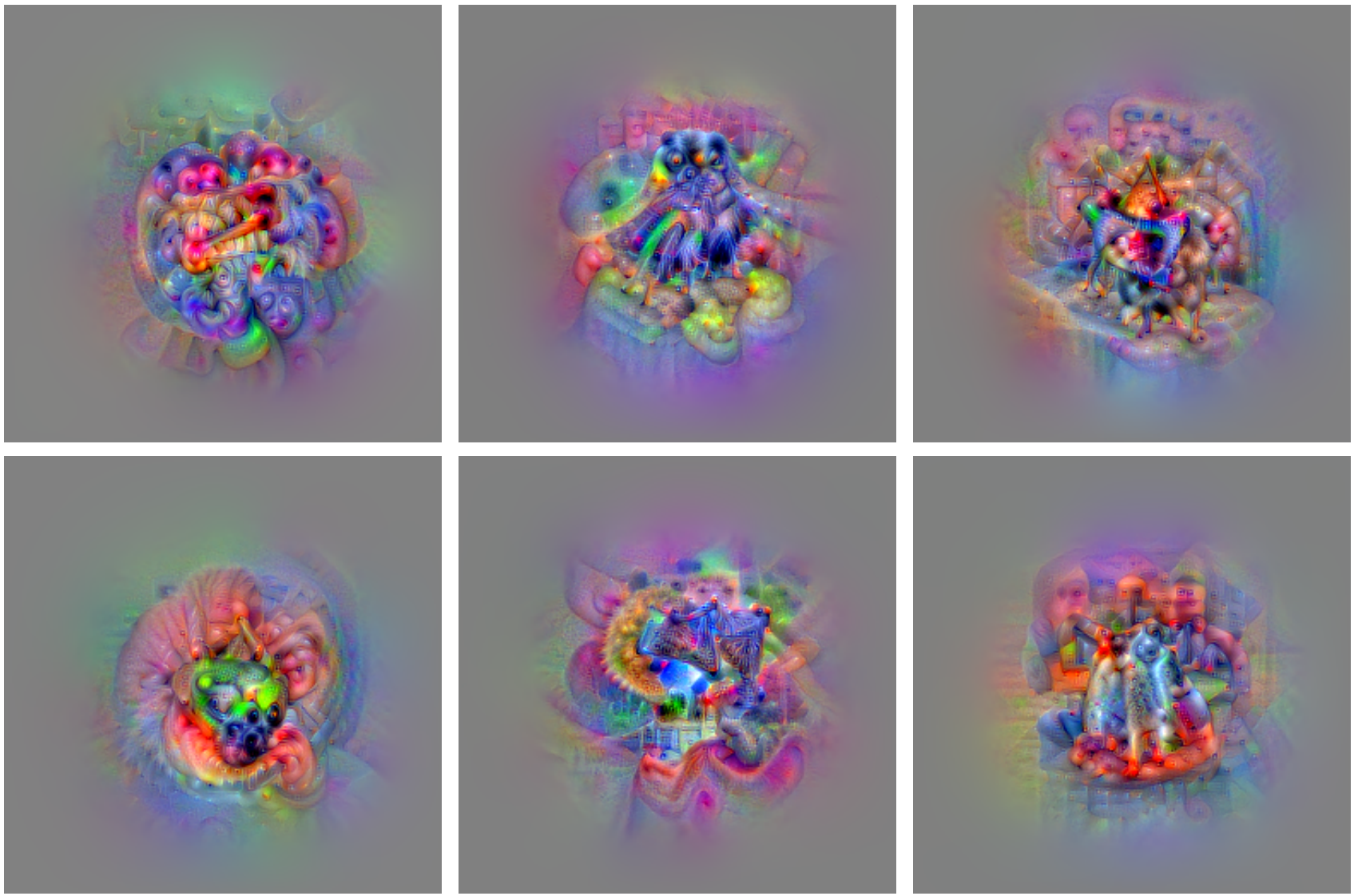
VGG16, maximizing a channel of the 7th convolution layer



VGG16, maximizing a unit of the 10th convolution layer

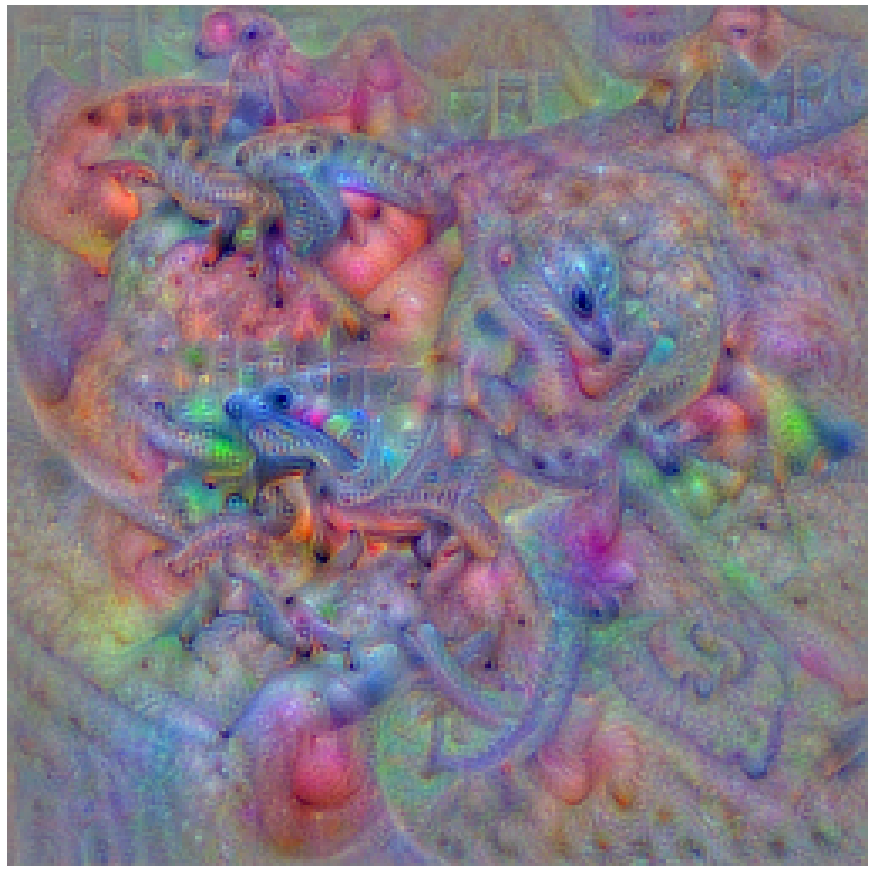


VGG16, maximizing a unit of the 13th (and last) convolution layer



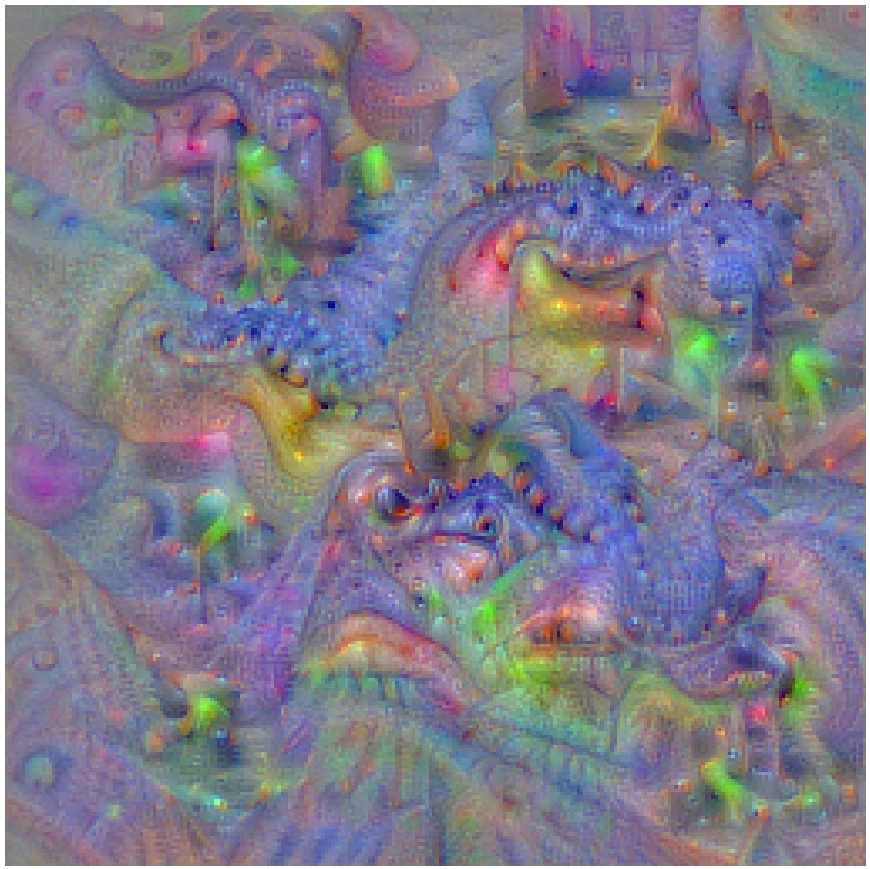
VGG16, maximizing a unit of the output layer

"Box turtle"



"Whiptail lizard"

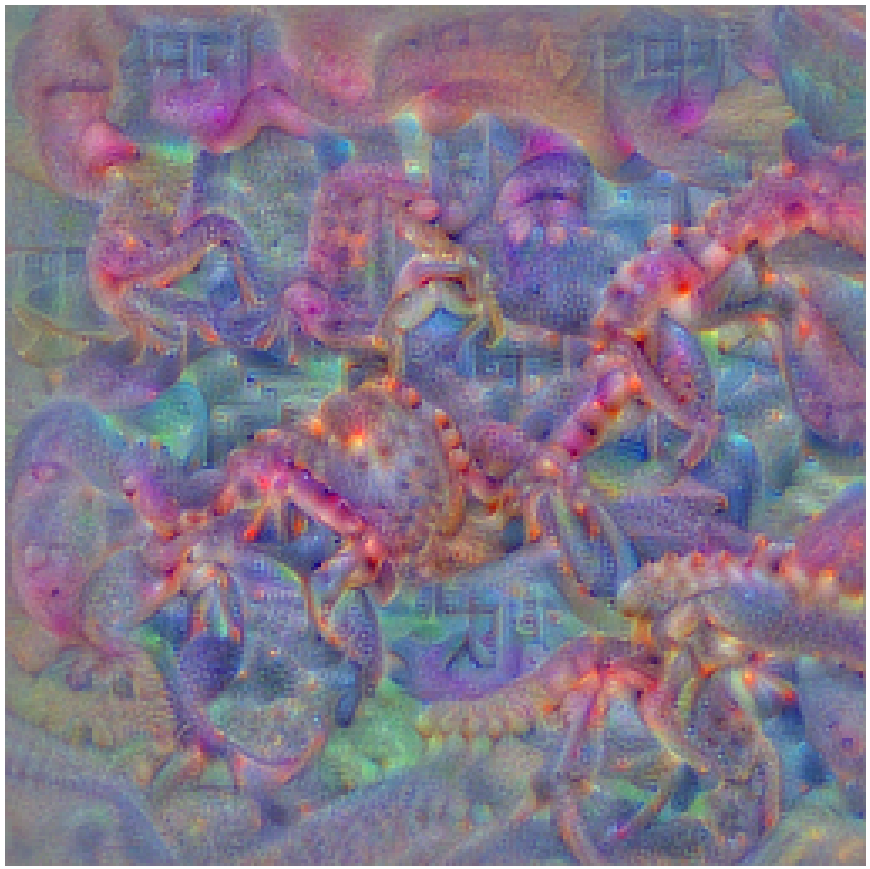
VGG16, maximizing a unit of the output layer



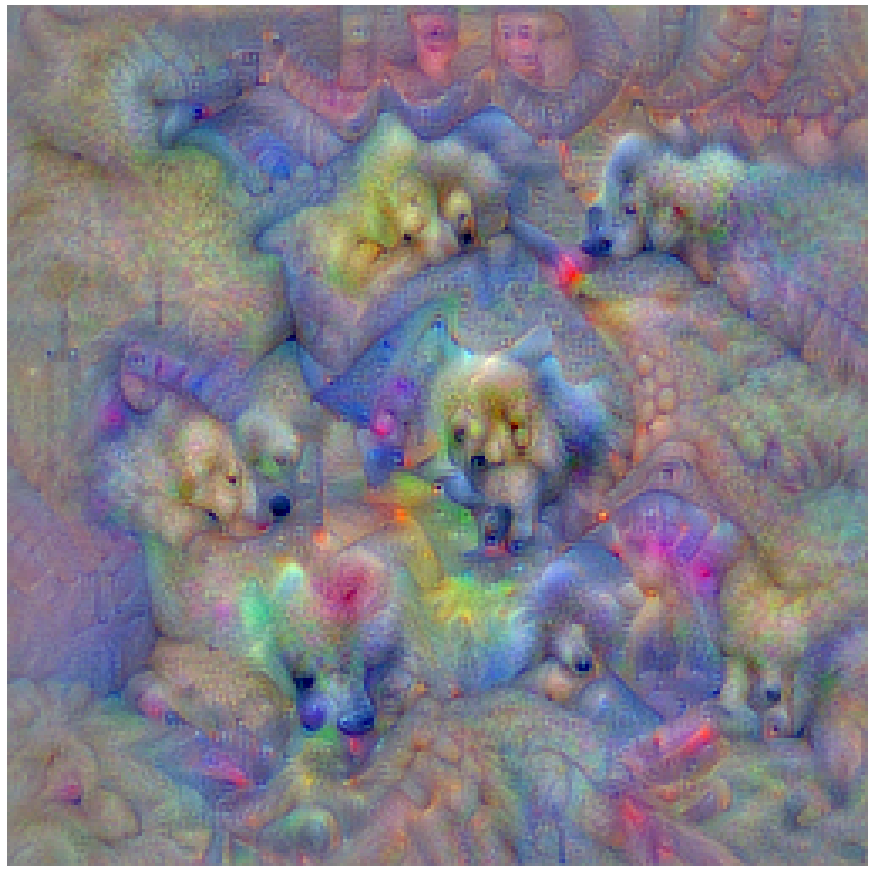
"African chameleon"

"Wolf spider"

VGG16, maximizing a unit of the output layer



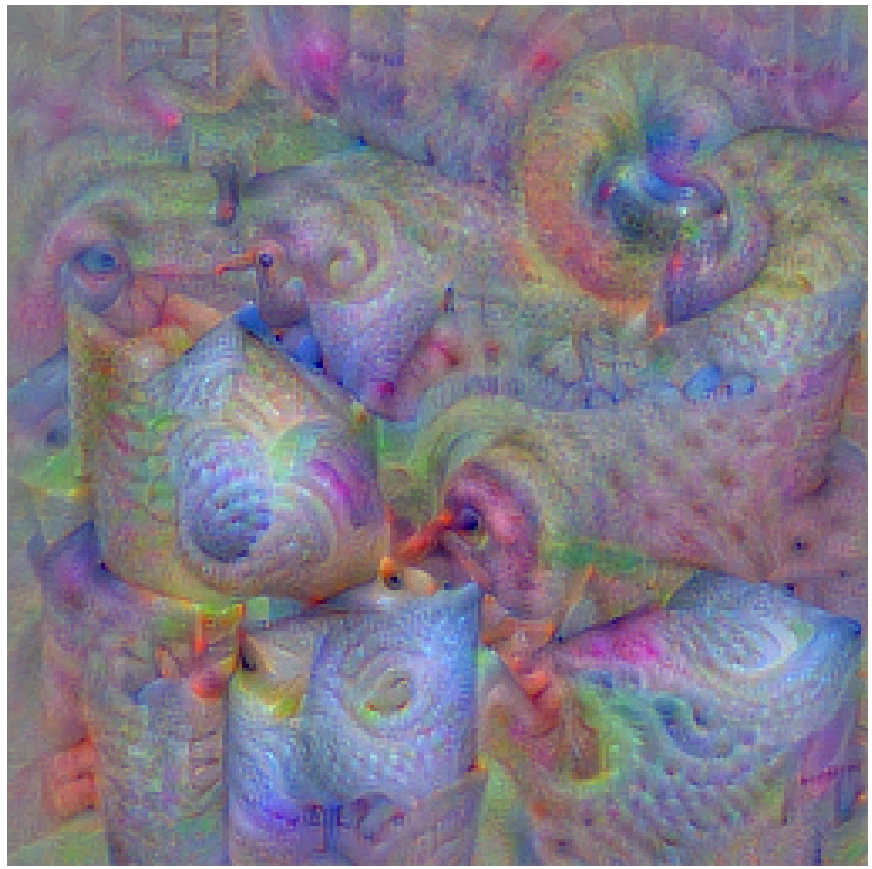
"King crab"



"Samoyed" (that's a fluffy dog)

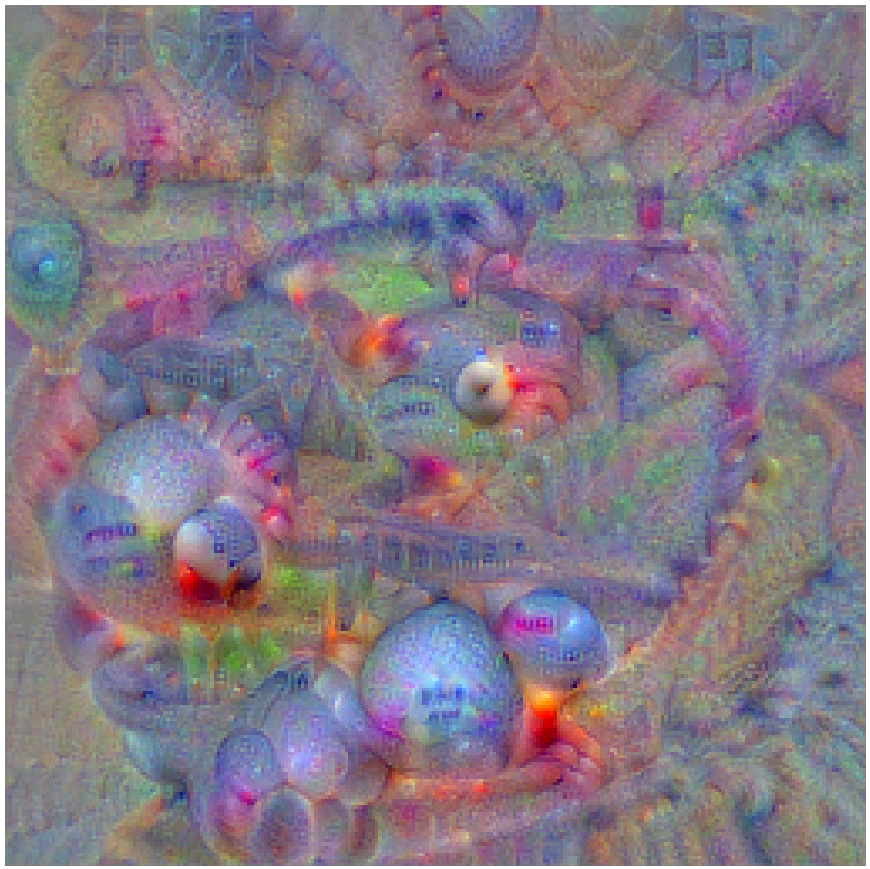
VGG16, maximizing a unit of the output layer

"Hourglass"

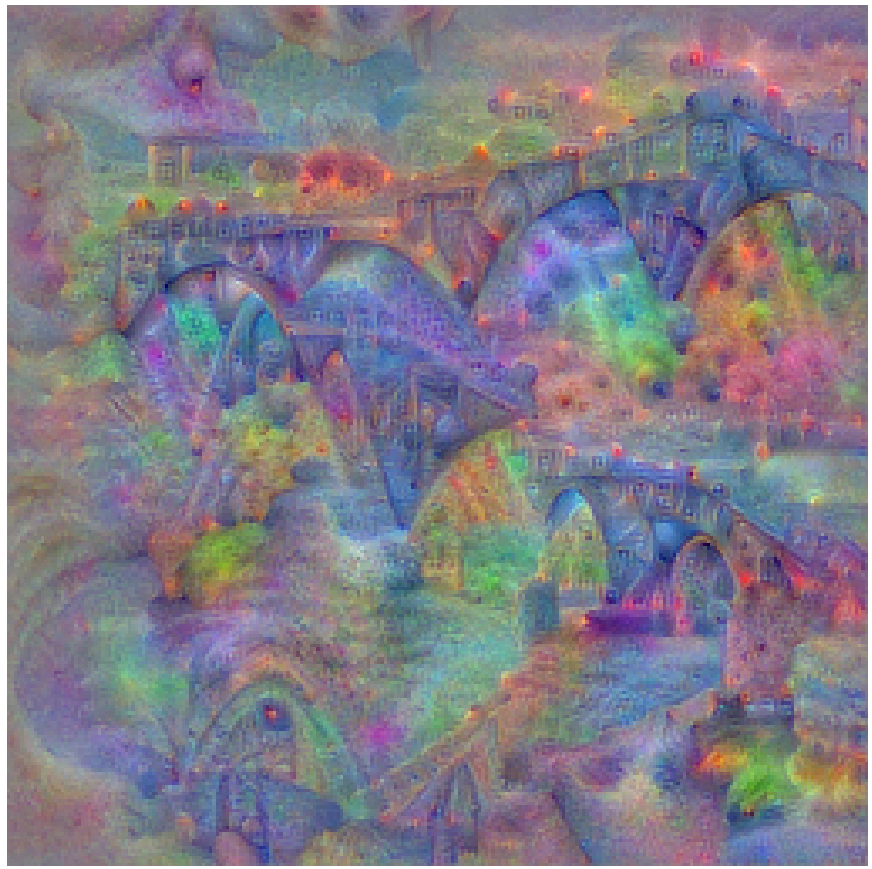


"Paper towel"

VGG16, maximizing a unit of the output layer

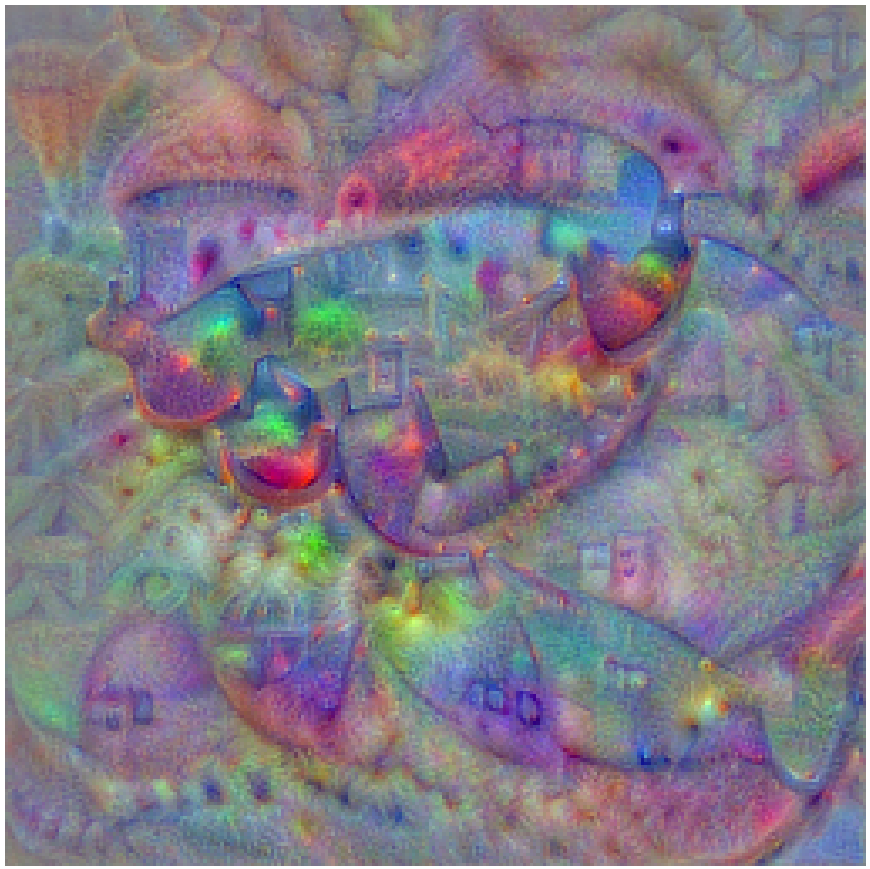


"Ping-pong ball"

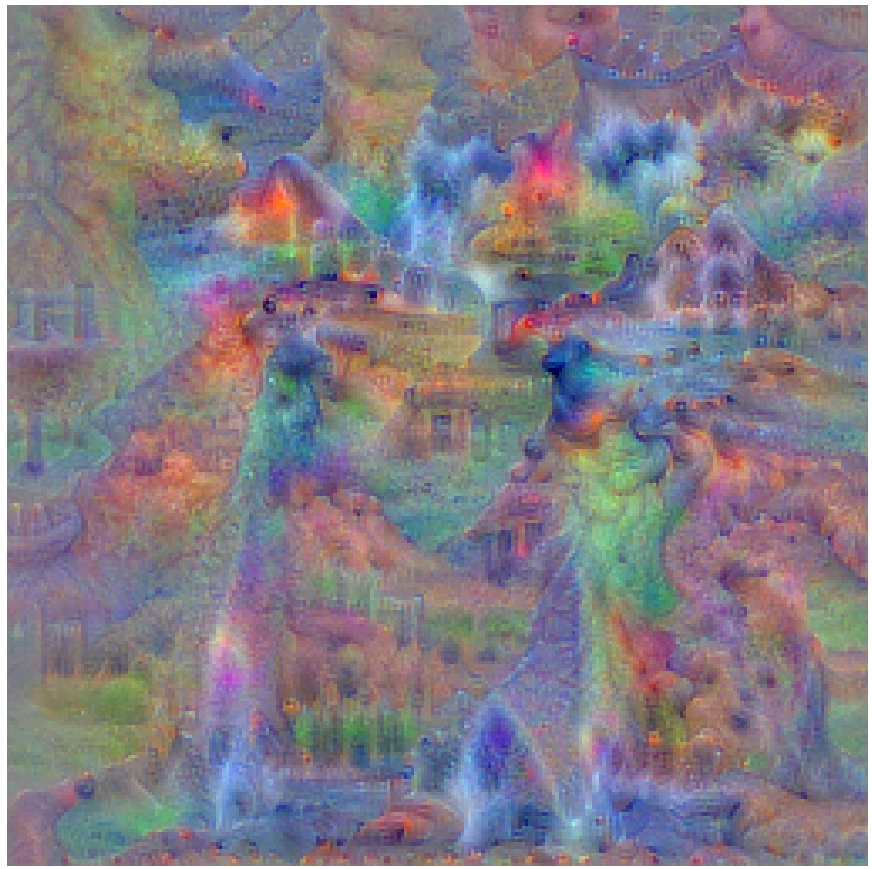


"Steel arch bridge"

VGG16, maximizing a unit of the output layer



"Sunglass"



"Geyser"

These results show that the parameters of a network trained for classification carry enough information to generate identifiable large-scale structures.

Although the training is discriminative, the resulting model has strong generative capabilities.

It also gives an intuition of the accuracy and shortcomings of the resulting global compositional model.

Adversarial examples

In spite of their good predictive capabilities, deep neural networks are quite sensitive to adversarial inputs, that is to inputs crafted to make them behave incorrectly (Szegedy et al., 2014).

The simplest strategy to exhibit such behavior is to **optimize the input to maximize the loss**.

Let x be an image, y its proper label, $f(x; w)$ the network's prediction, and $\mathscr{L}$ the cross-entropy loss. We can construct an adversarial example by maximizing the loss. To do so, we iterate a "gradient ascent" step:

$$x_{k+1} = x_k + \eta \nabla_{|x} \mathscr{L}(f(x_k; w), y).$$

After a few iterations, this procedure will reach a sample $\check{x}$ whose class is not y.

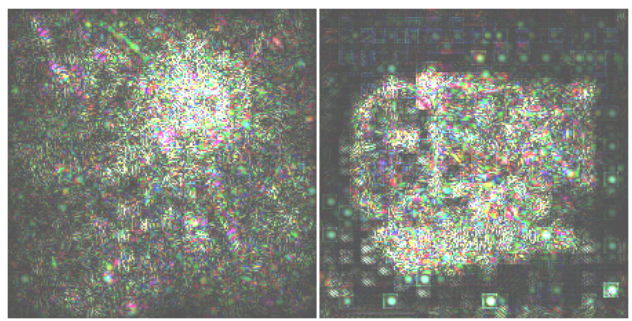
The counter-intuitive result is that the resulting miss-classified images are indistinguishable from the original ones to a human eye.

```
model = torchvision.models.alexnet(pretrained = True)
target = model(input).argmax(1).view(-1)

cross_entropy = nn.CrossEntropyLoss()
optimizer = optim.SGD([input], lr = 1e-1)
nb_steps = 15

for k in range(nb_steps):
    output = model(input)
    loss = - cross_entropy(output, target)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Original

Adversarial

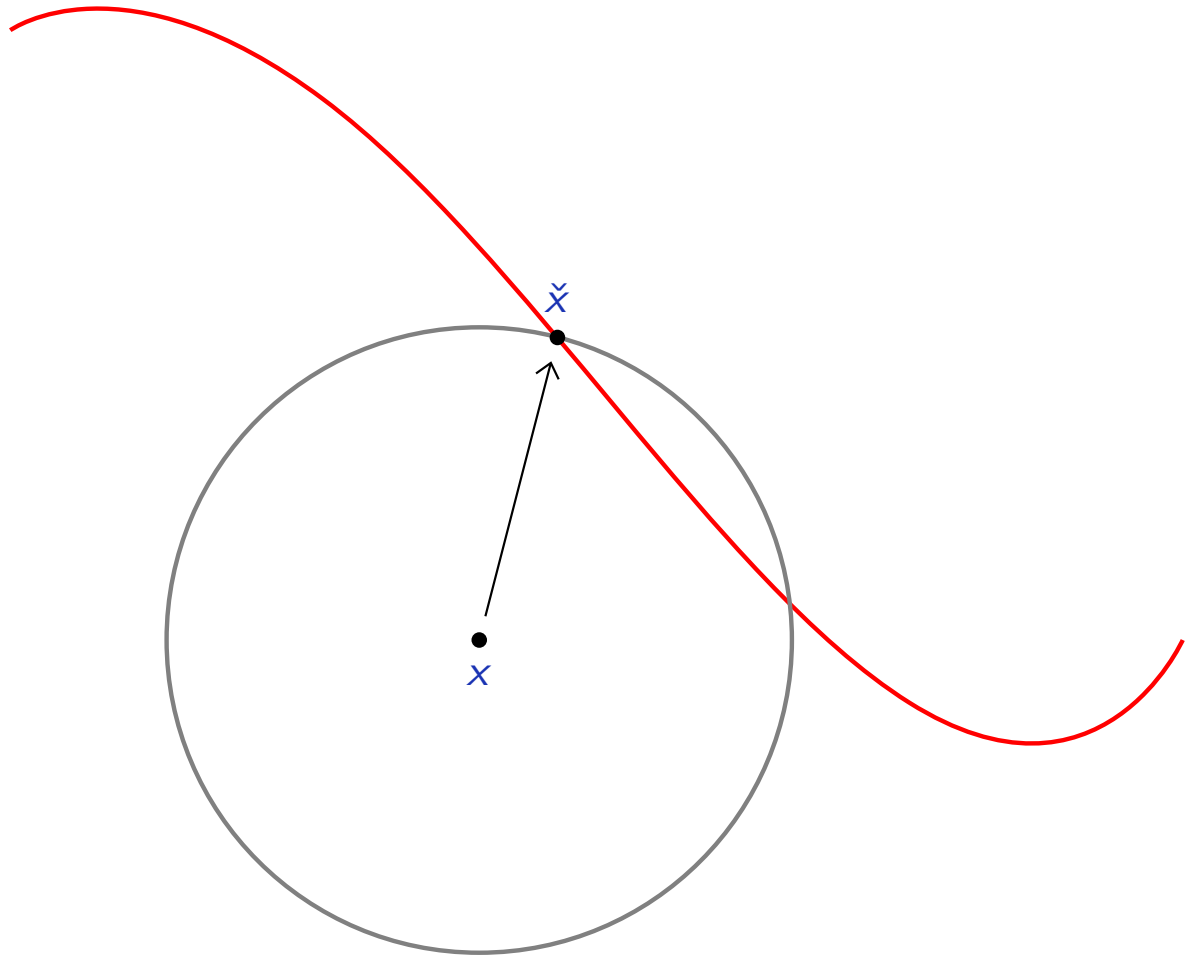
Differences
(magnified)



$\frac{\|x-\check{x}\|}{\|x\|}$ 1.02% 0.27%

Nb. iterations	Predicted classes Image #1	Image #2
0	Weimaraner	desktop computer
1	Weimaraner	desktop computer
2	Labrador retriever	desktop computer
3	Labrador retriever	desktop computer
4	Labrador retriever	desktop computer
5	brush kangaroo	desktop computer
6	brush kangaroo	desktop computer
7	sundial	desktop computer
8	sundial	desktop computer
9	sundial	desktop computer
10	sundial	desktop computer
11	sundial	desktop computer
12	sundial	desktop computer
13	sundial	desktop computer
14	sundial	desk

Another counter-intuitive result is that if we sample $1,000$ images on the sphere centered on x of radius $2\|x - \check{x}\|$, we do not observe any change of label.



References

C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. **Intriguing properties of neural networks**. In International Conference on Learning Representations (ICLR), 2014.

The end