EE-559 – Deep learning

8. Under the hood

François Fleuret

https://fleuret.org/dlc/

[version of: June 5, 2018]

idiap RESEARCH INSTITUTE

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Understanding a network's behavior

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 as images,
- distributions of activations on a population of samples,
- derivatives of the response(s) wrt the input,
- maximum-response synthetic samples,
- adversarial samples.

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 represent each of its internal units as a line corresponding to

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

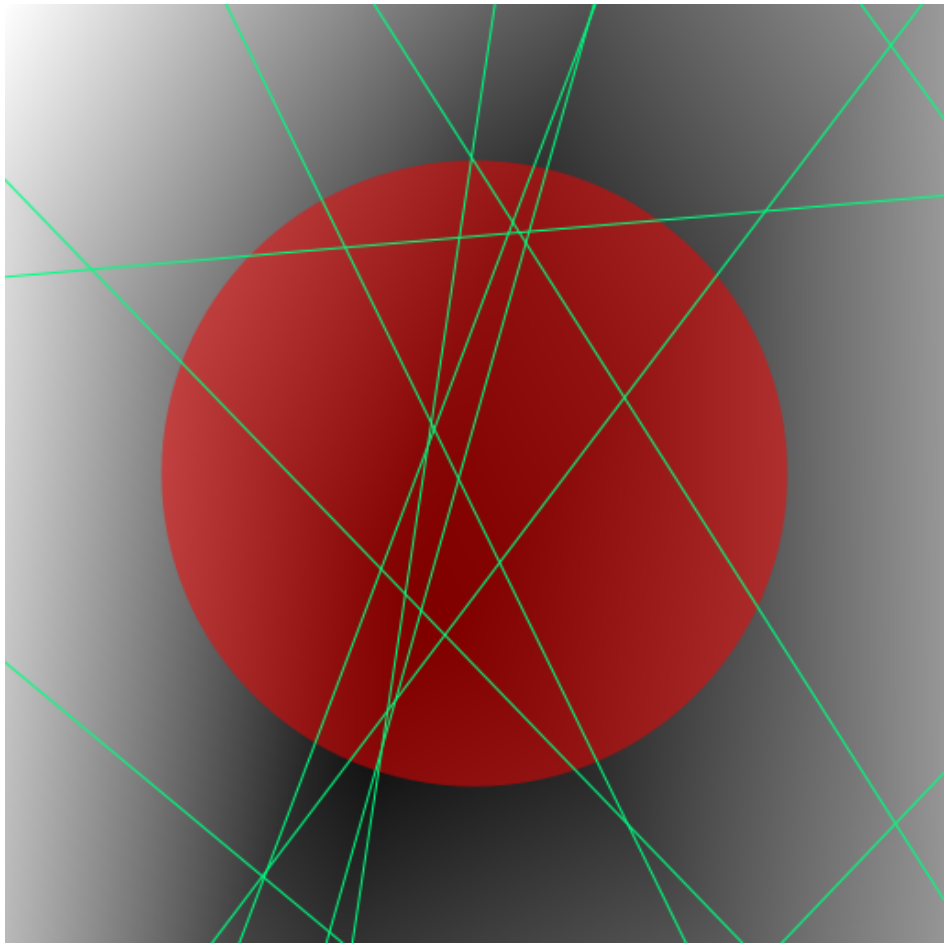
Given a one-hidden layer fully connected network $\mathbb{R}^2 \to \mathbb{R}^2$

```
nb_hidden = 20

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

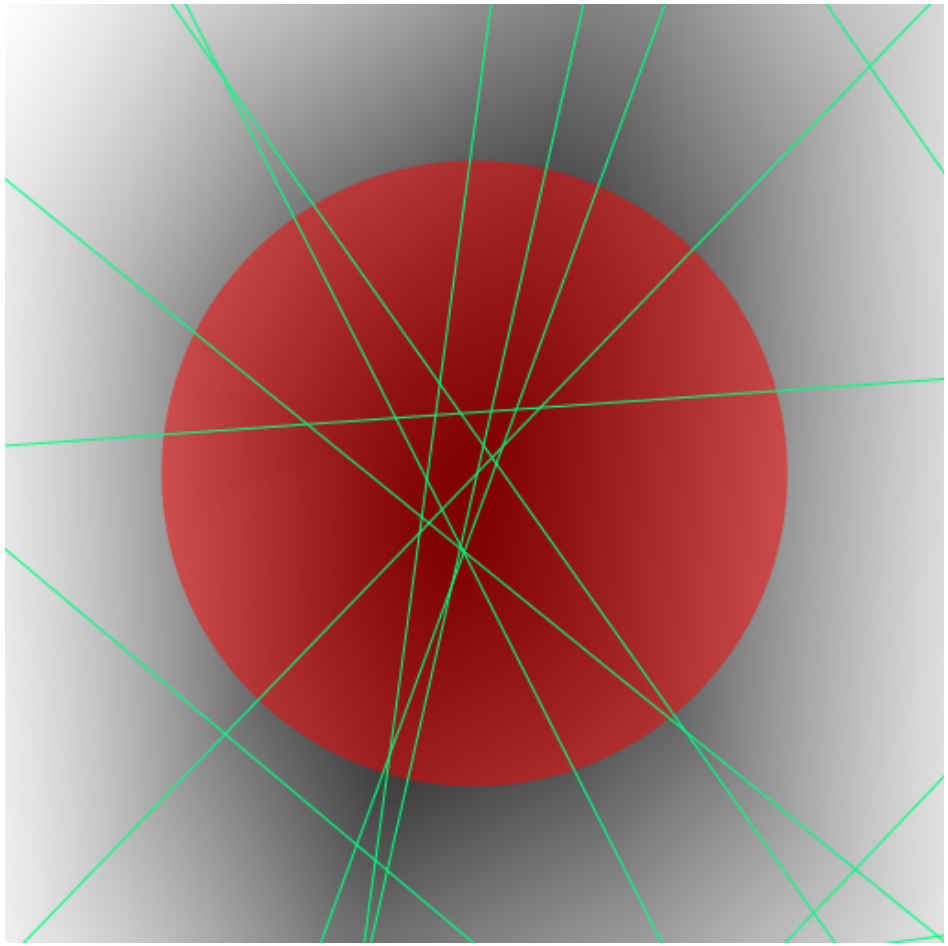
we can represent each of its internal units as a line corresponding to

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

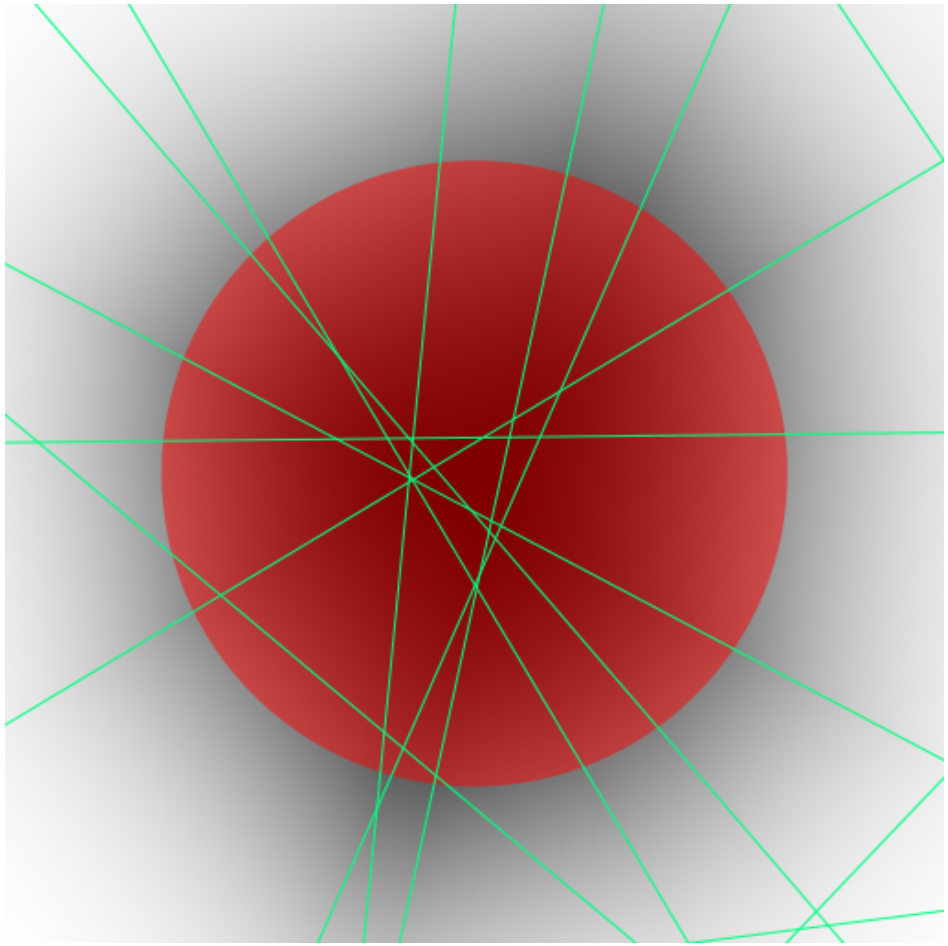
During training, these separations get organized so that their combination delimitates properly the different value domains in the signal space.



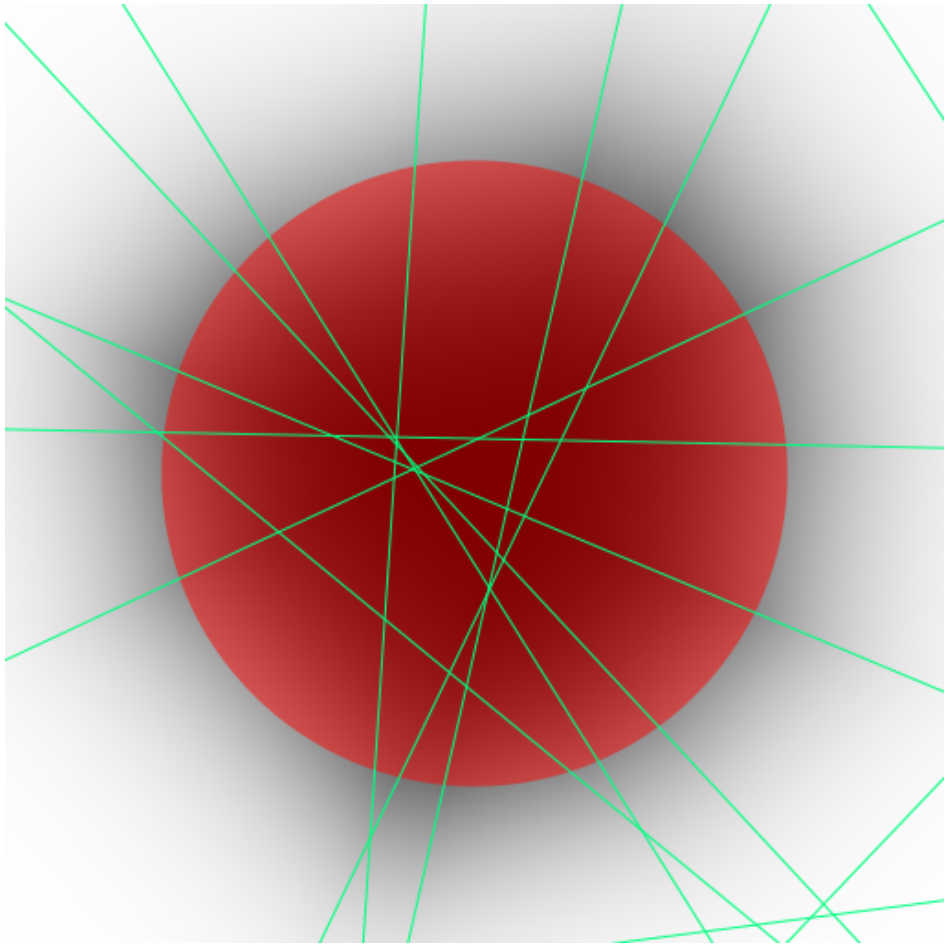
Iteration 1



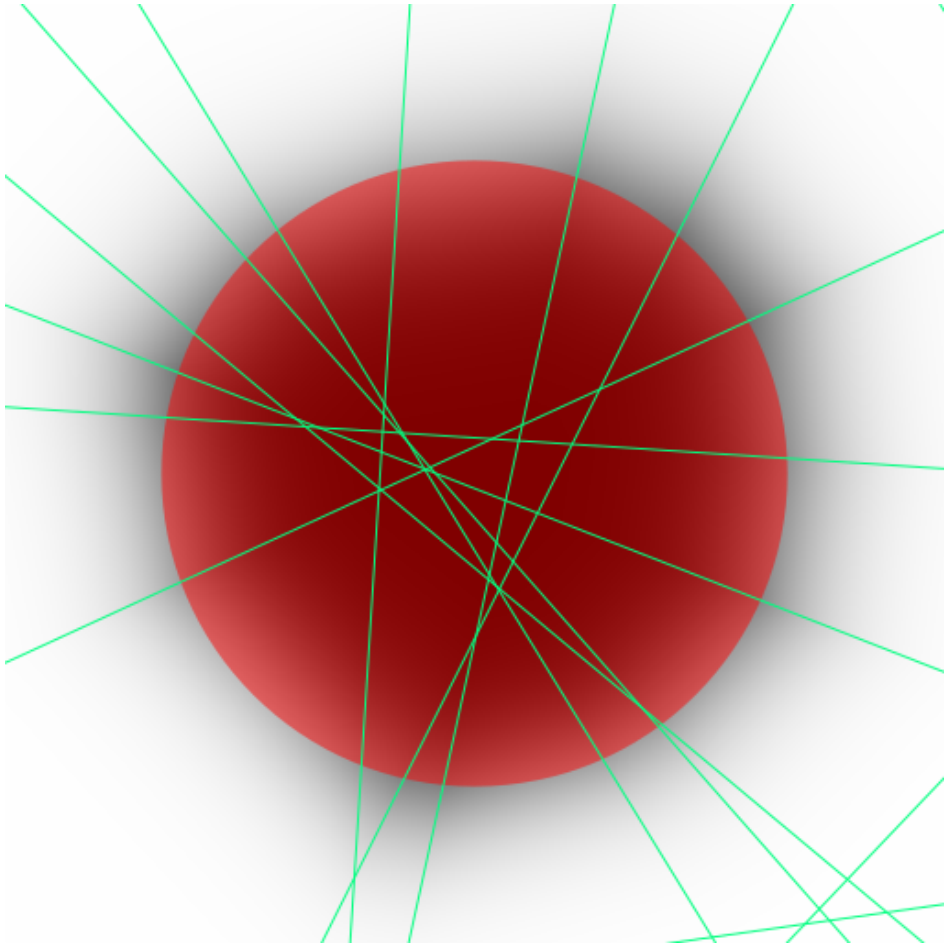
Iteration 4



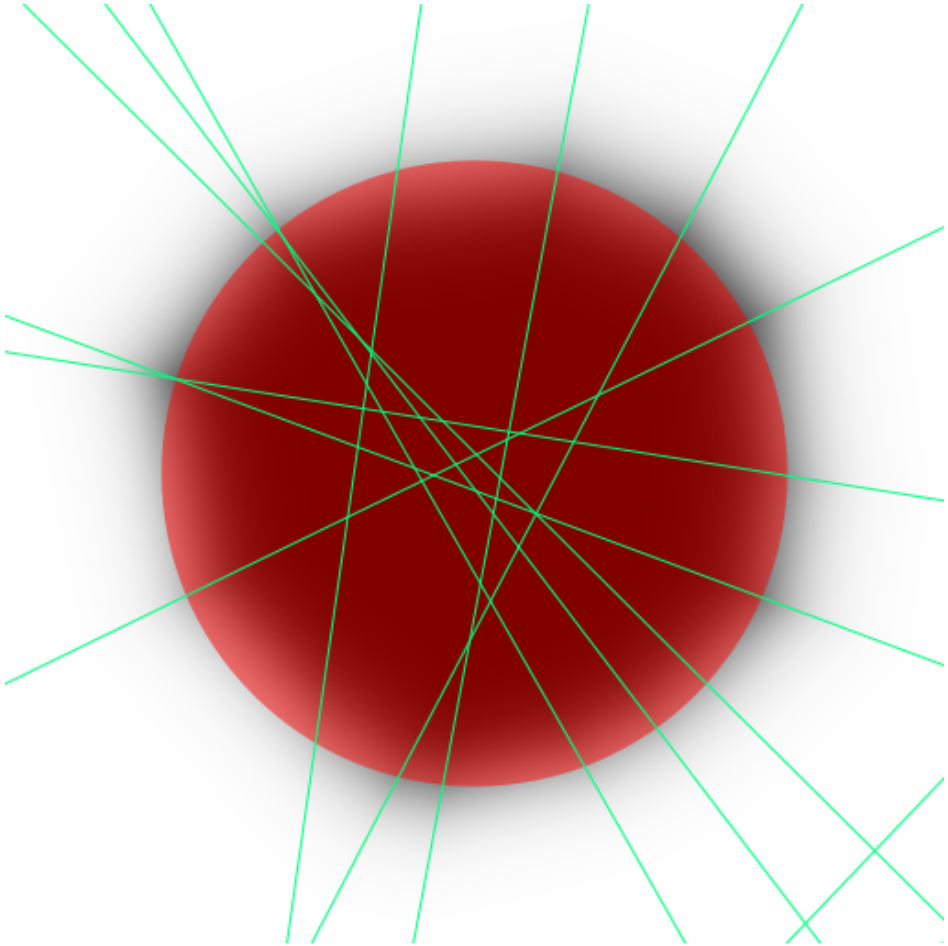
Iteration 7



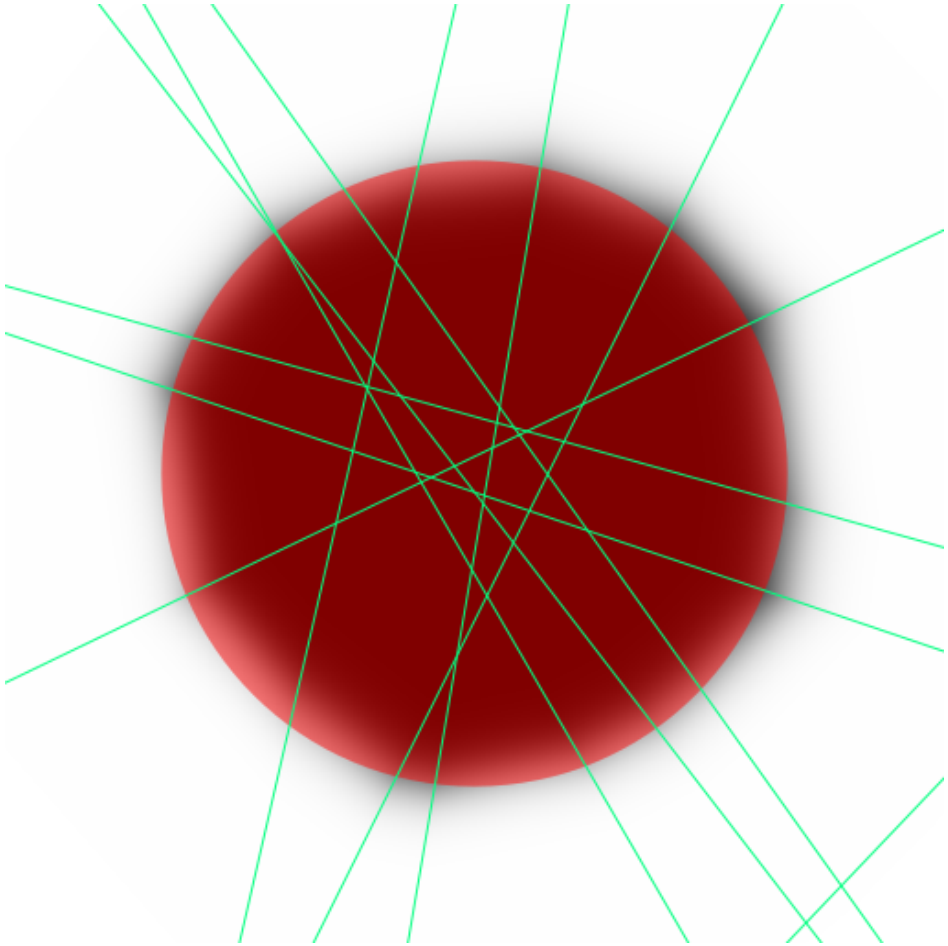
Iteration 10



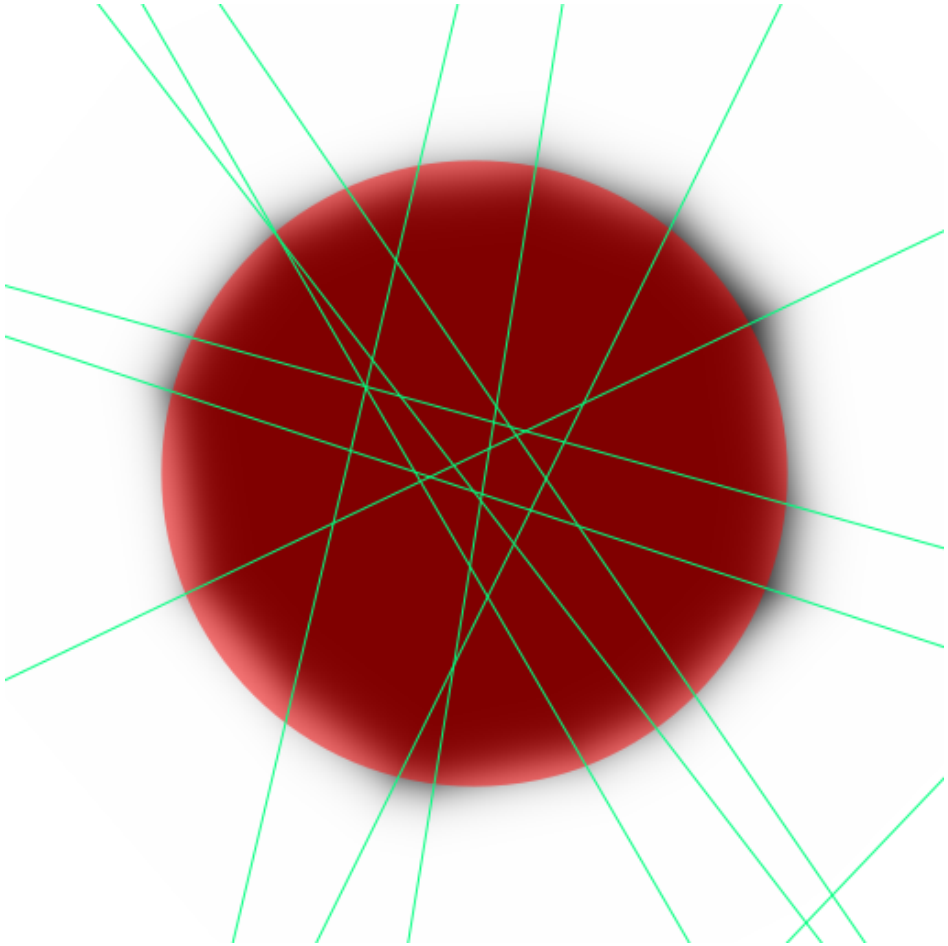
Iteration 16



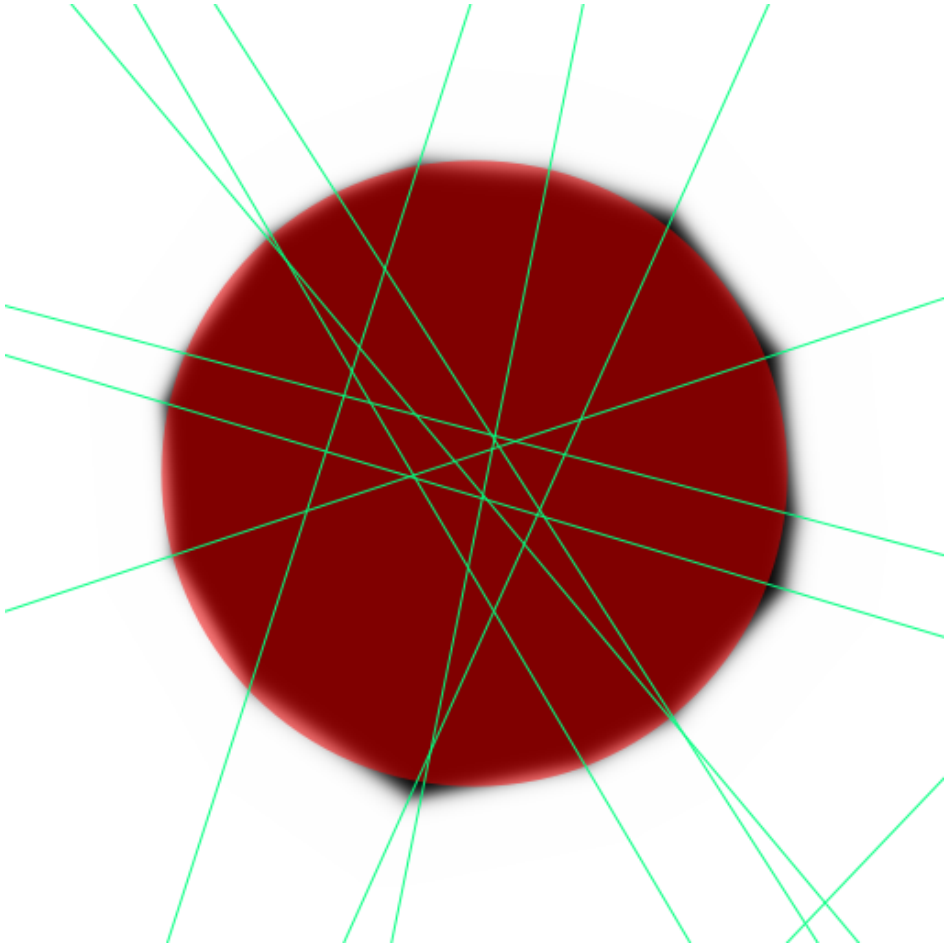
Iteration 34



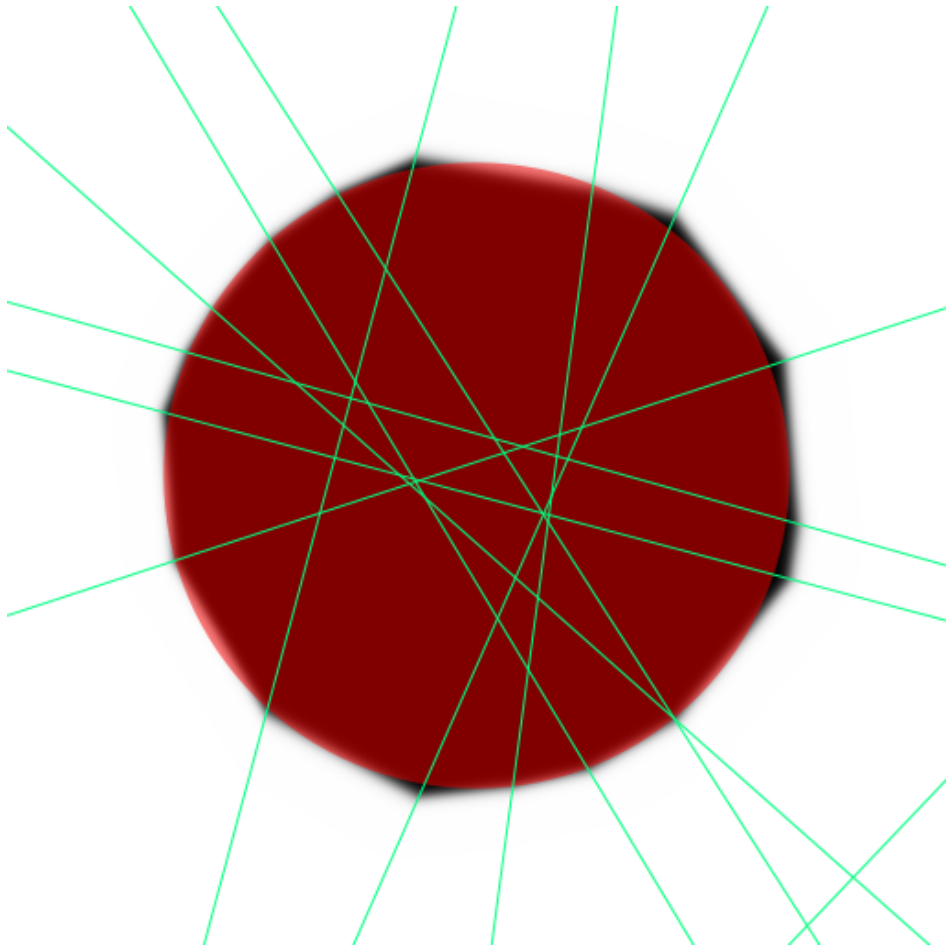
Iteration 77



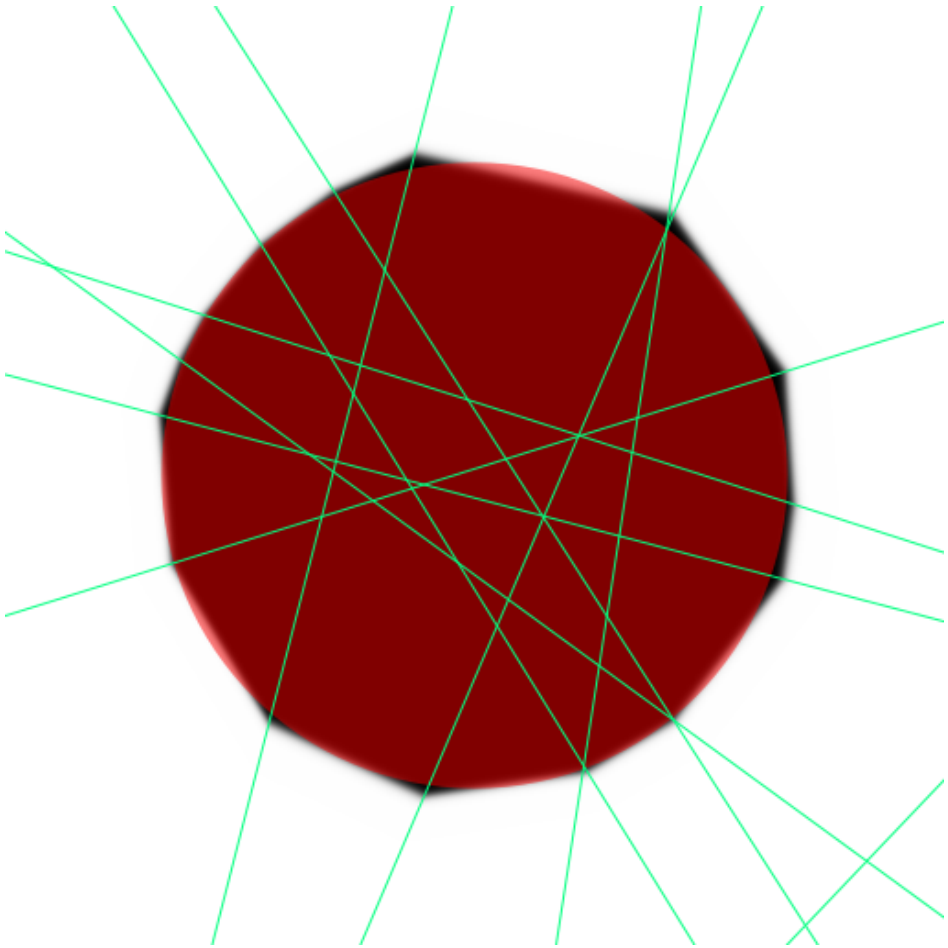
Iteration 100



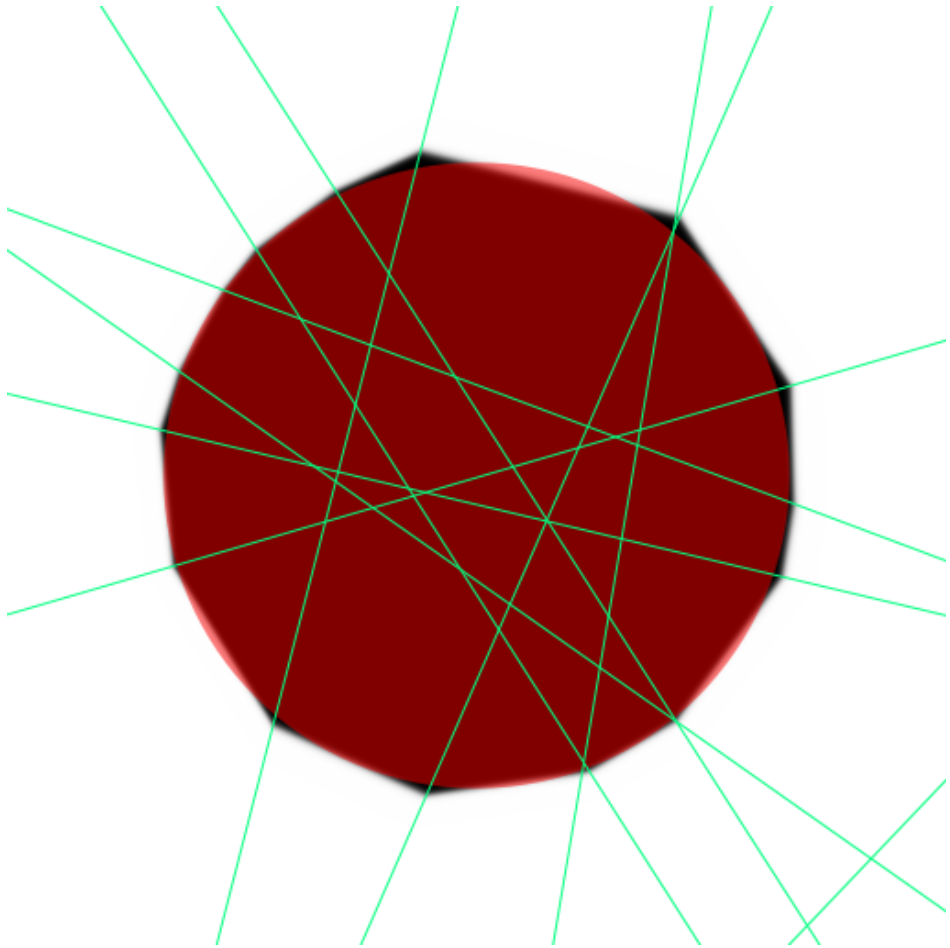
Iteration 703



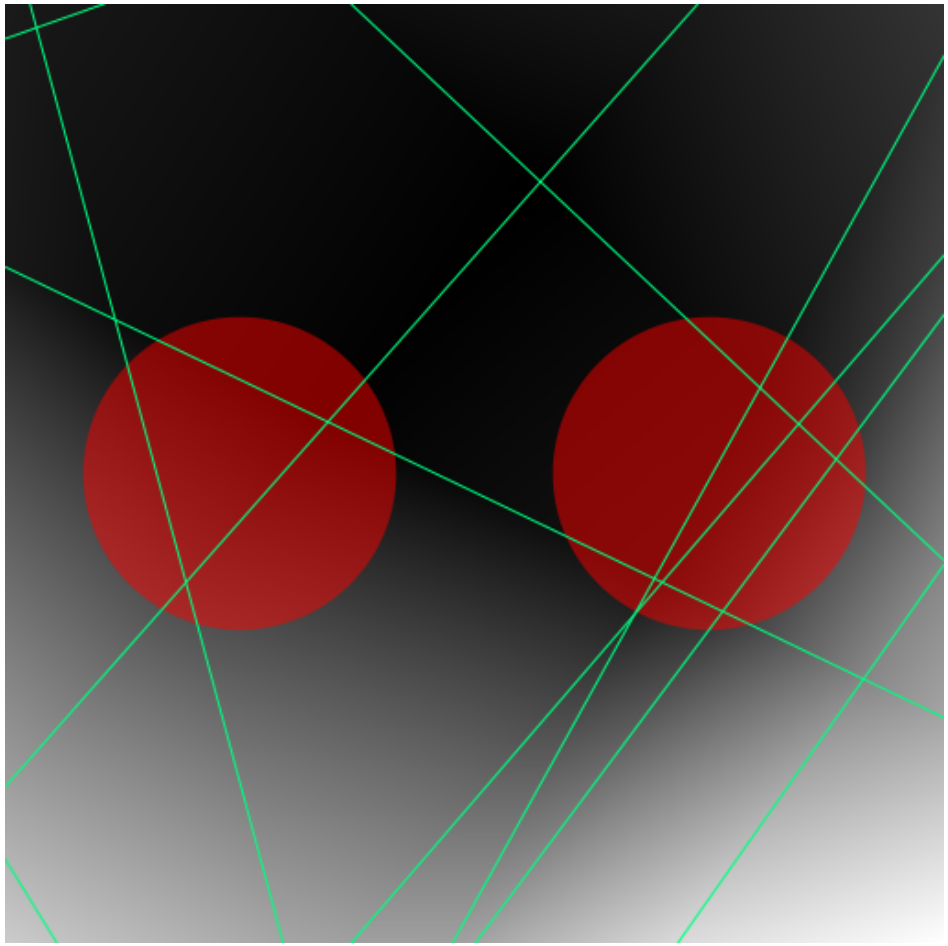
Iteration 1407



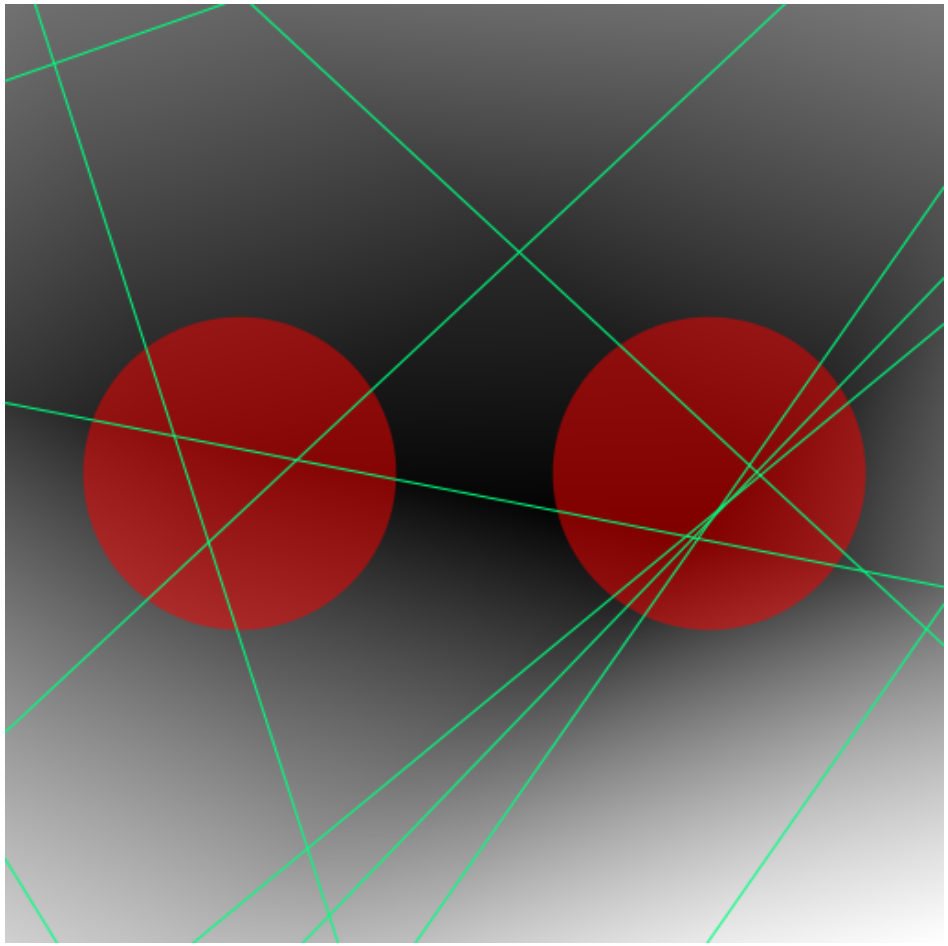
Iteration 2789



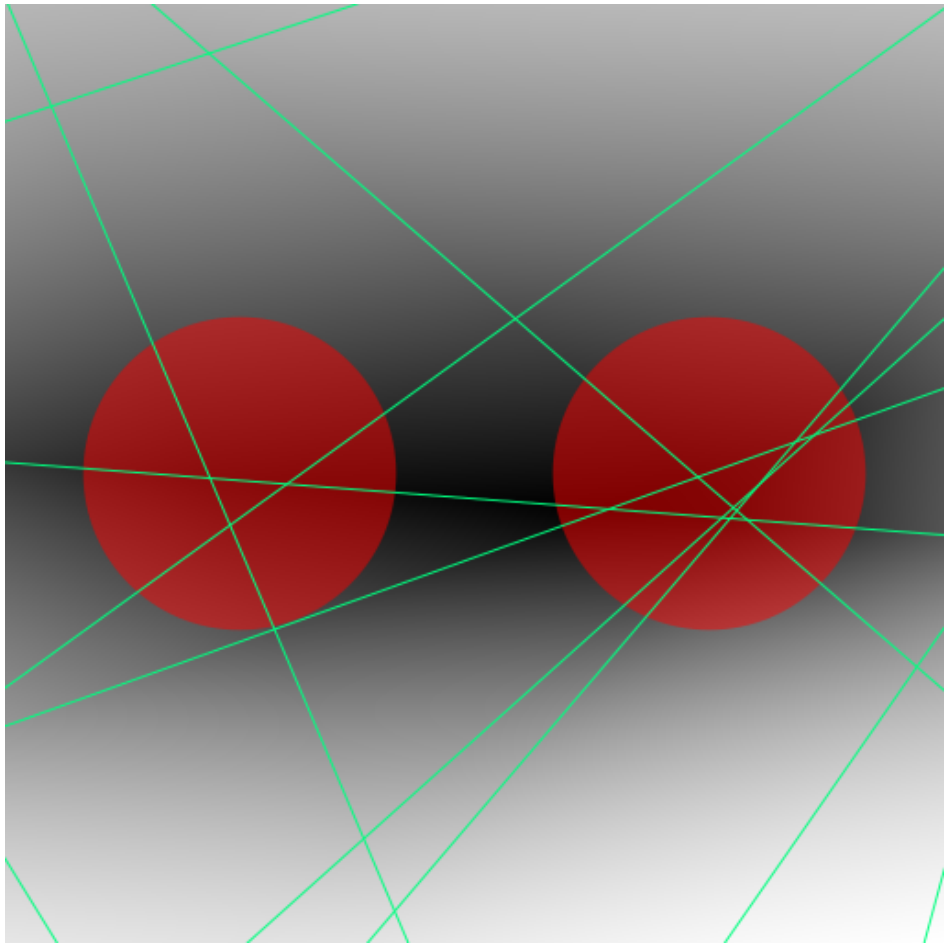
Iteration 4999



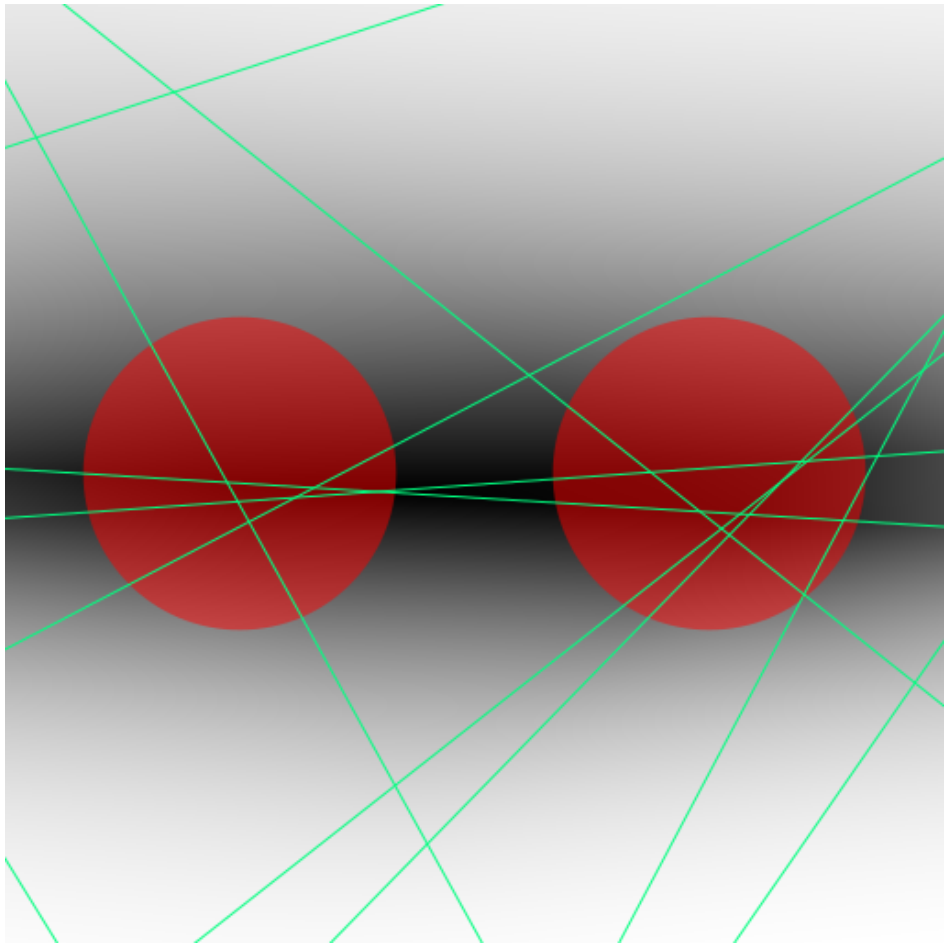
Iteration 1



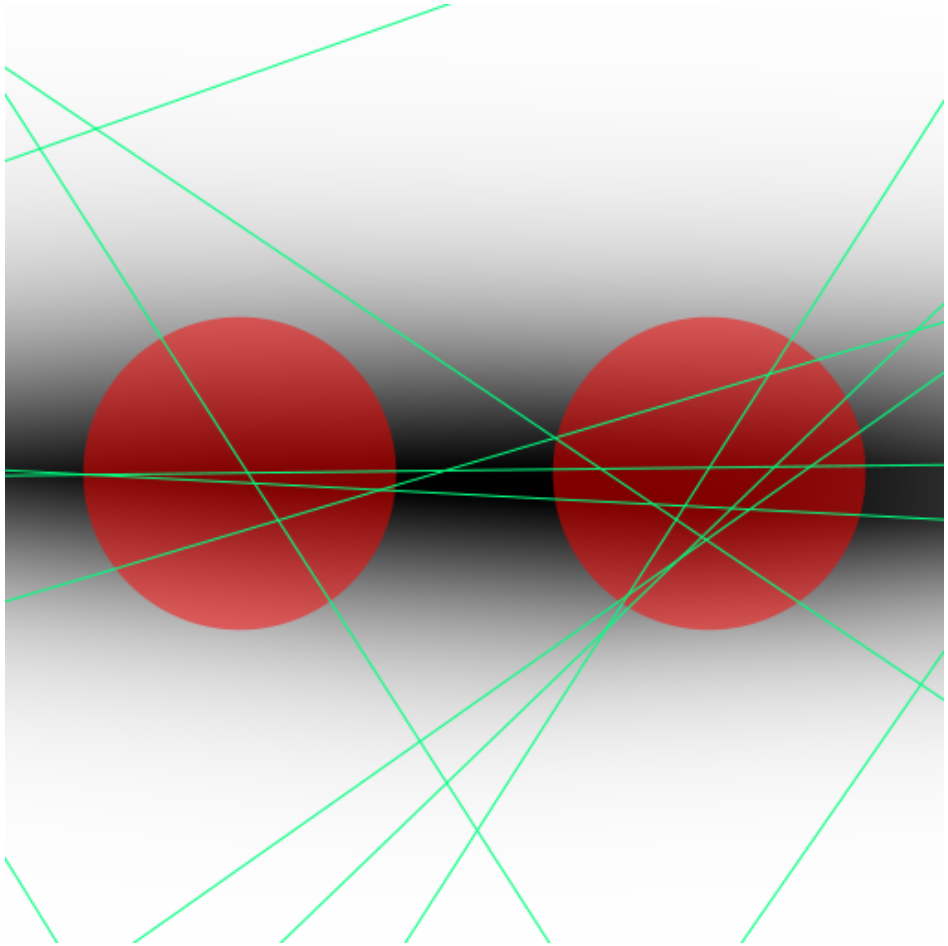
Iteration 4



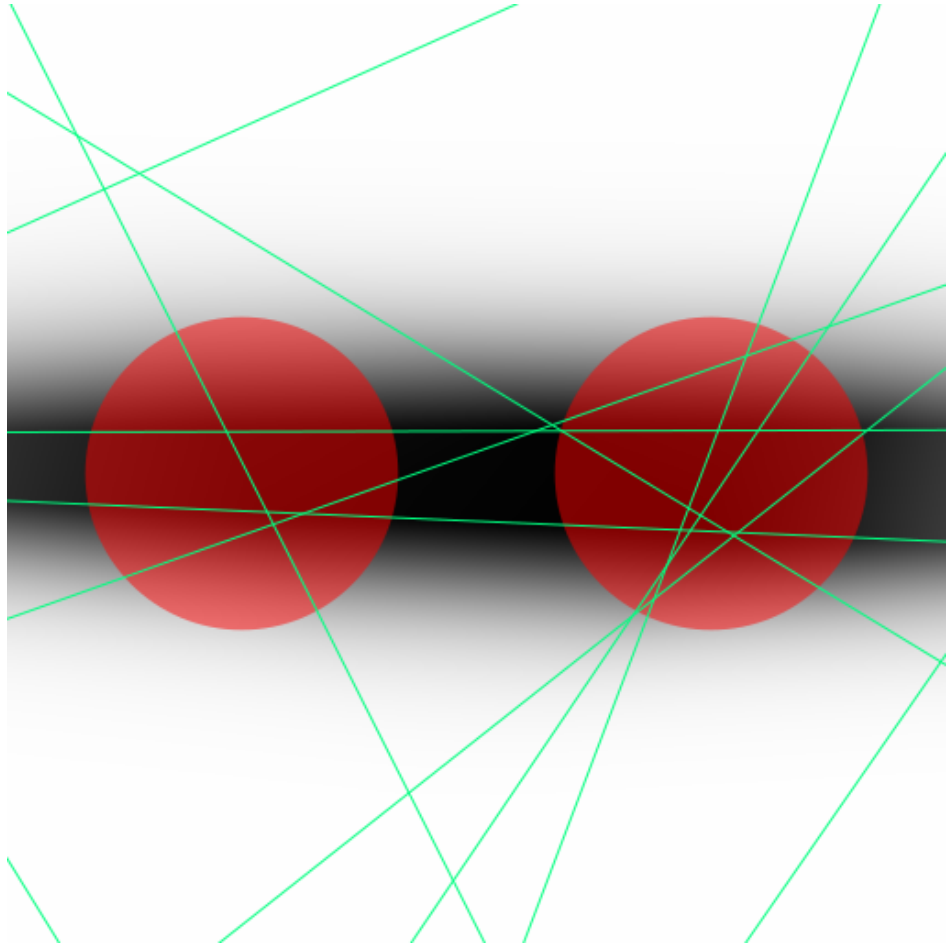
Iteration 7



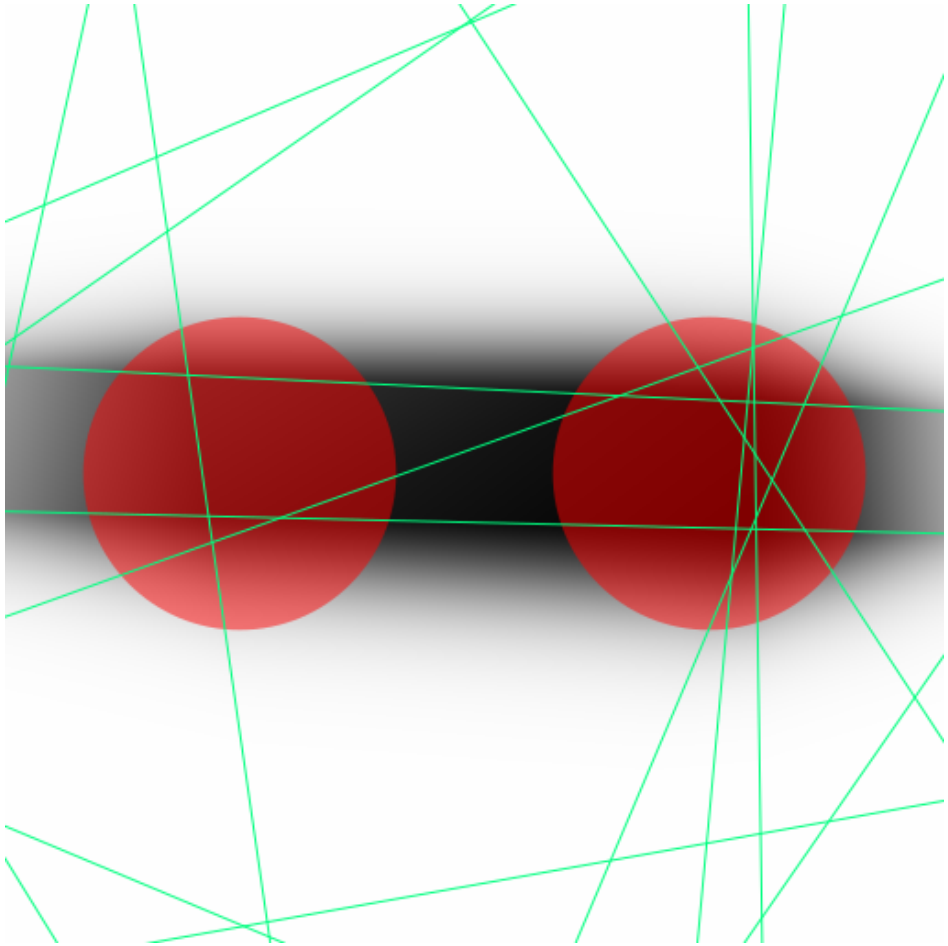
Iteration 10



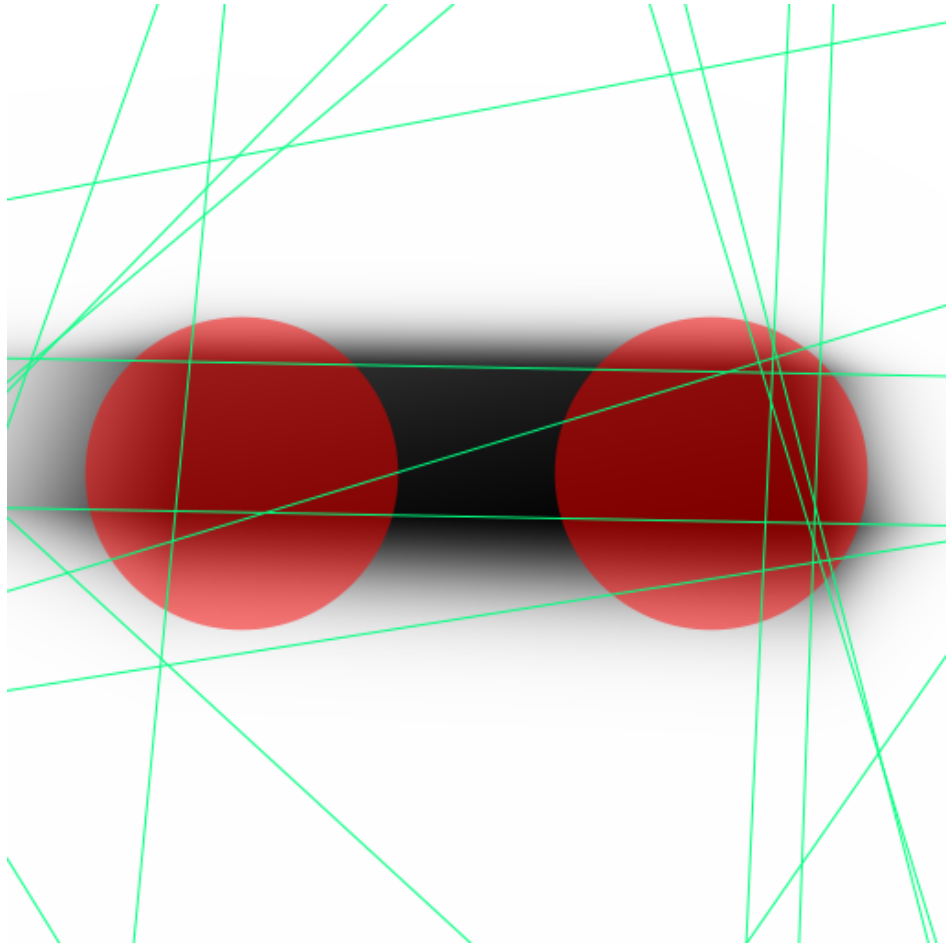
Iteration 16



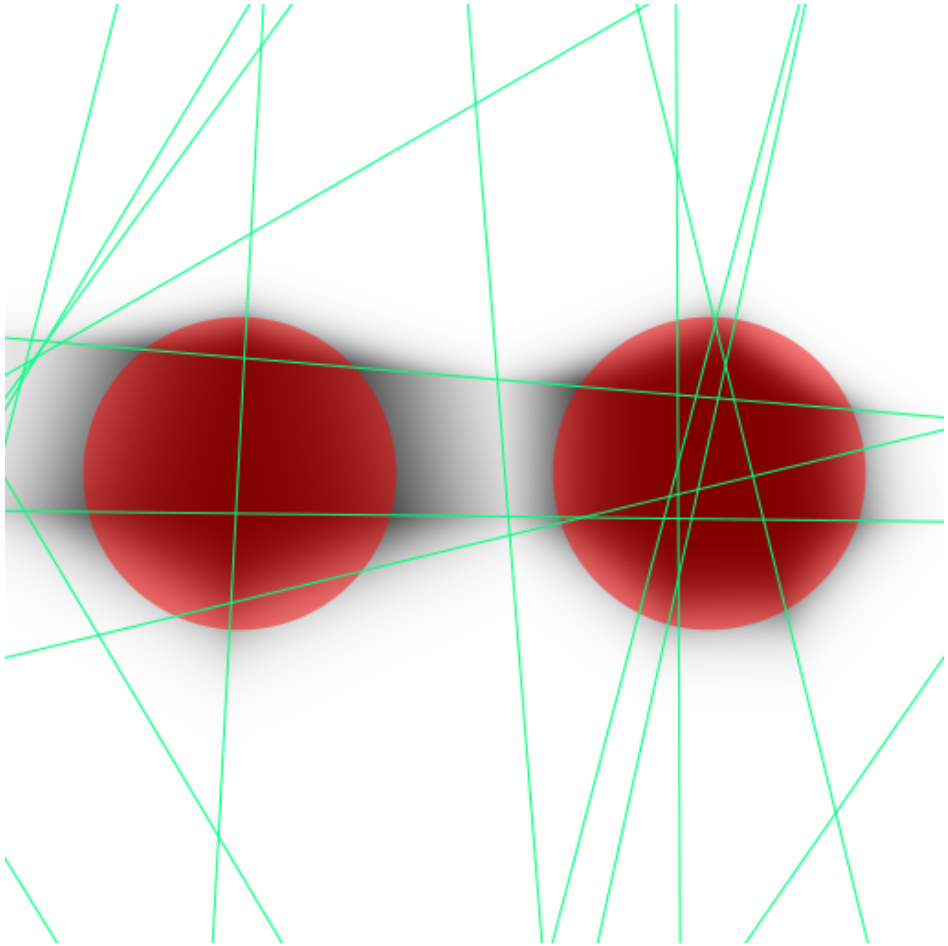
Iteration 34



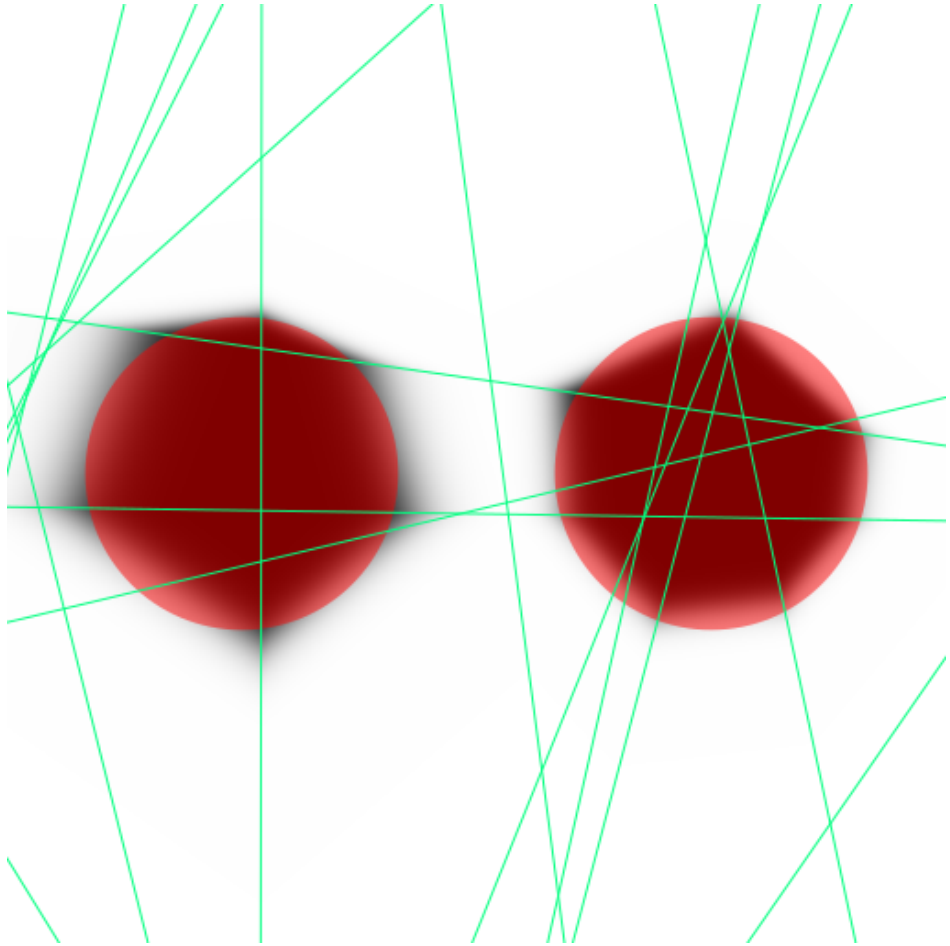
Iteration 100



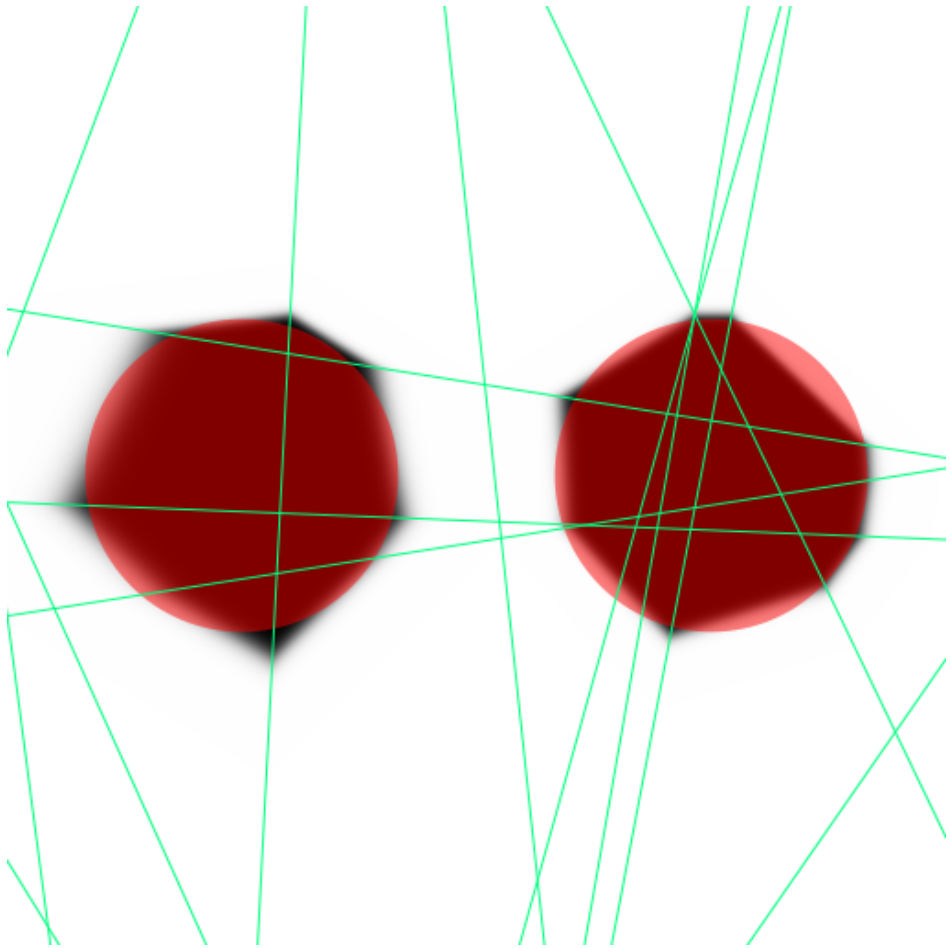
Iteration 272



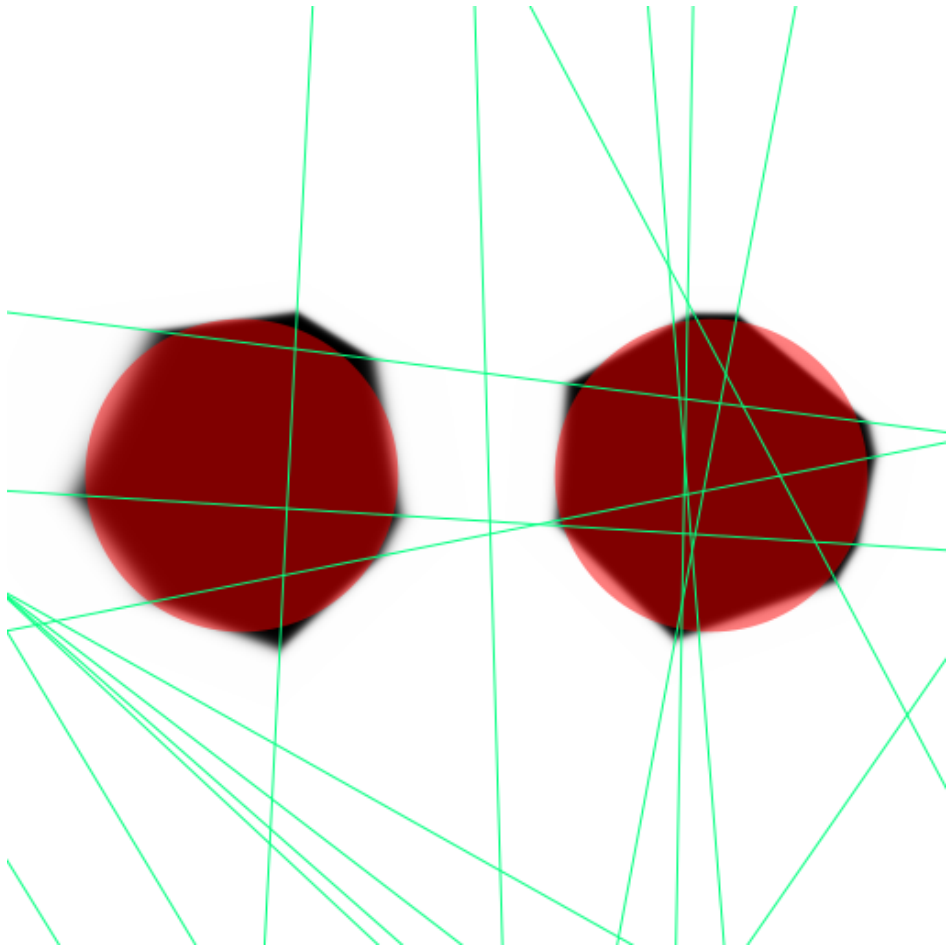
Iteration 556



Iteration 887



Iteration 2222



Iteration 4999

Convnet filters

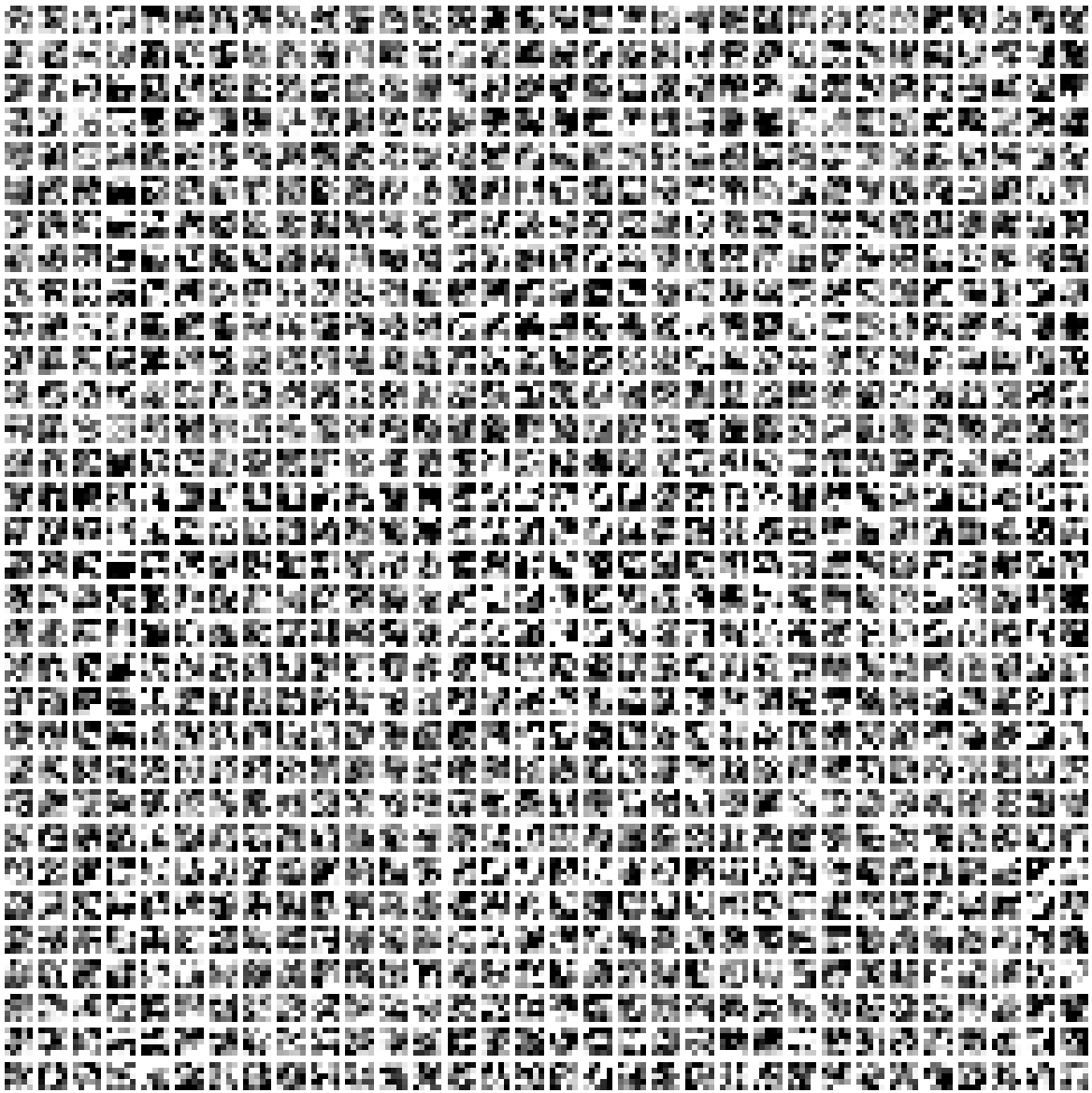
A similar analysis is complicated to conduct with real-life networks given the high dimension of the signal.

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

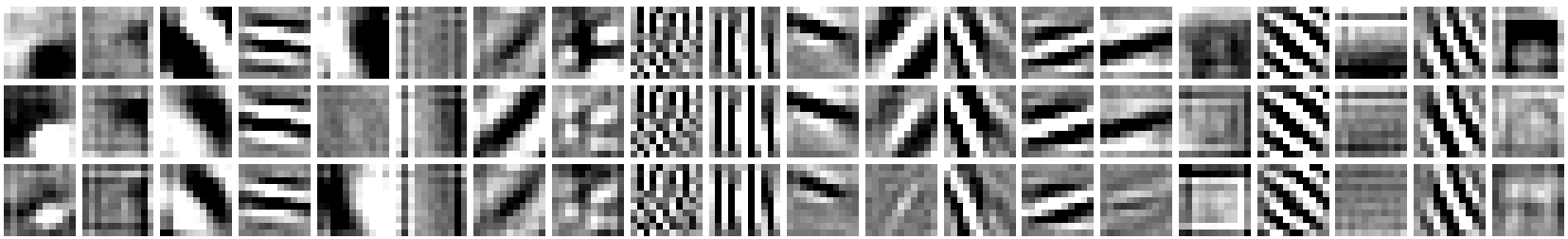
While it is quite reasonable in the first layer, since the filters are indeed consistent with the image input, it is far less so in the subsequent layers.

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

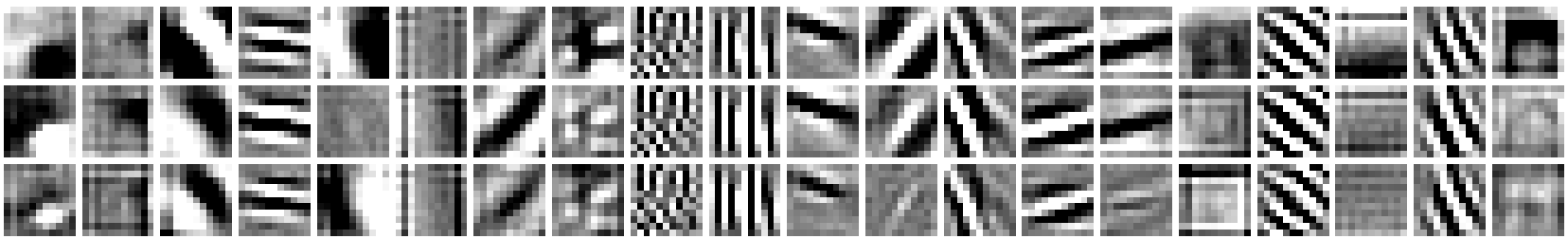
LeNet's second convolutional layer ($32 \rightarrow 64$), first 32 filters out of 64



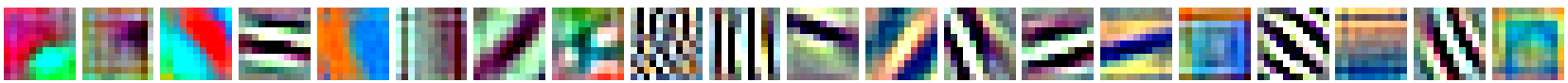
AlexNet's first convolutional layer ($3 \rightarrow 64$), first 20 filters out of 64



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).

Convnet internal layer activations

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.

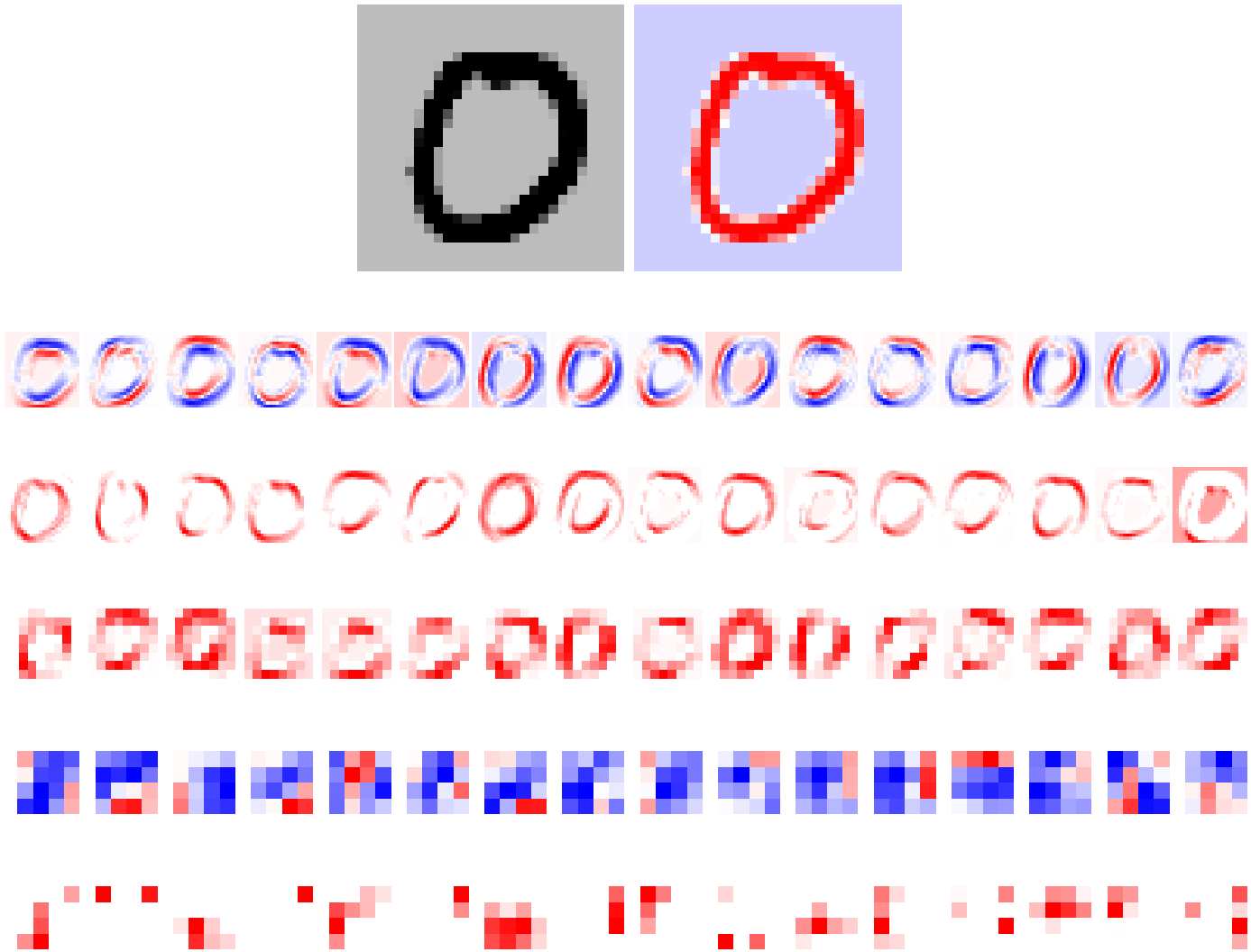
An alternative approach is to look at the activations themselves.

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.

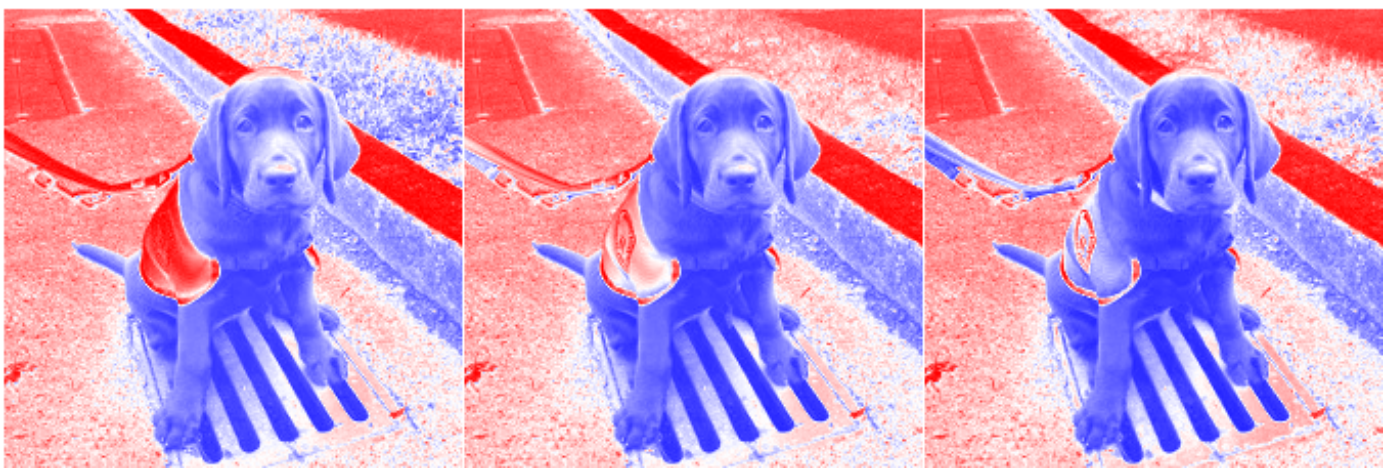
Given the large number of channels, we have to pick a few at random.

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

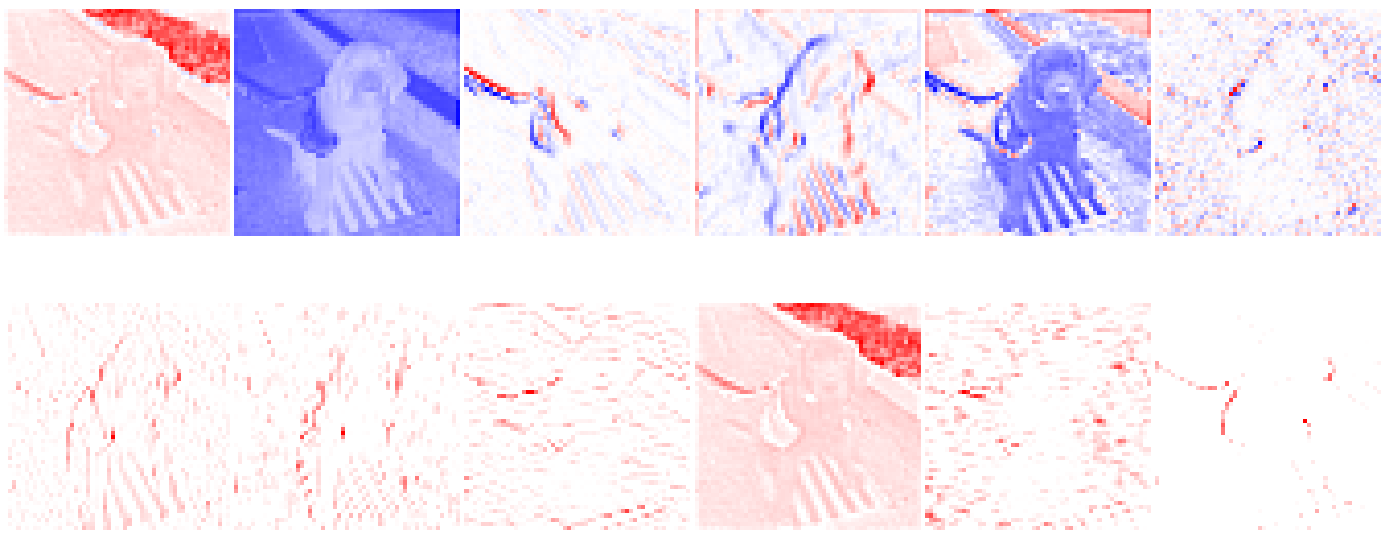
A MNIST character with LeNet (leCun et al., 1998).



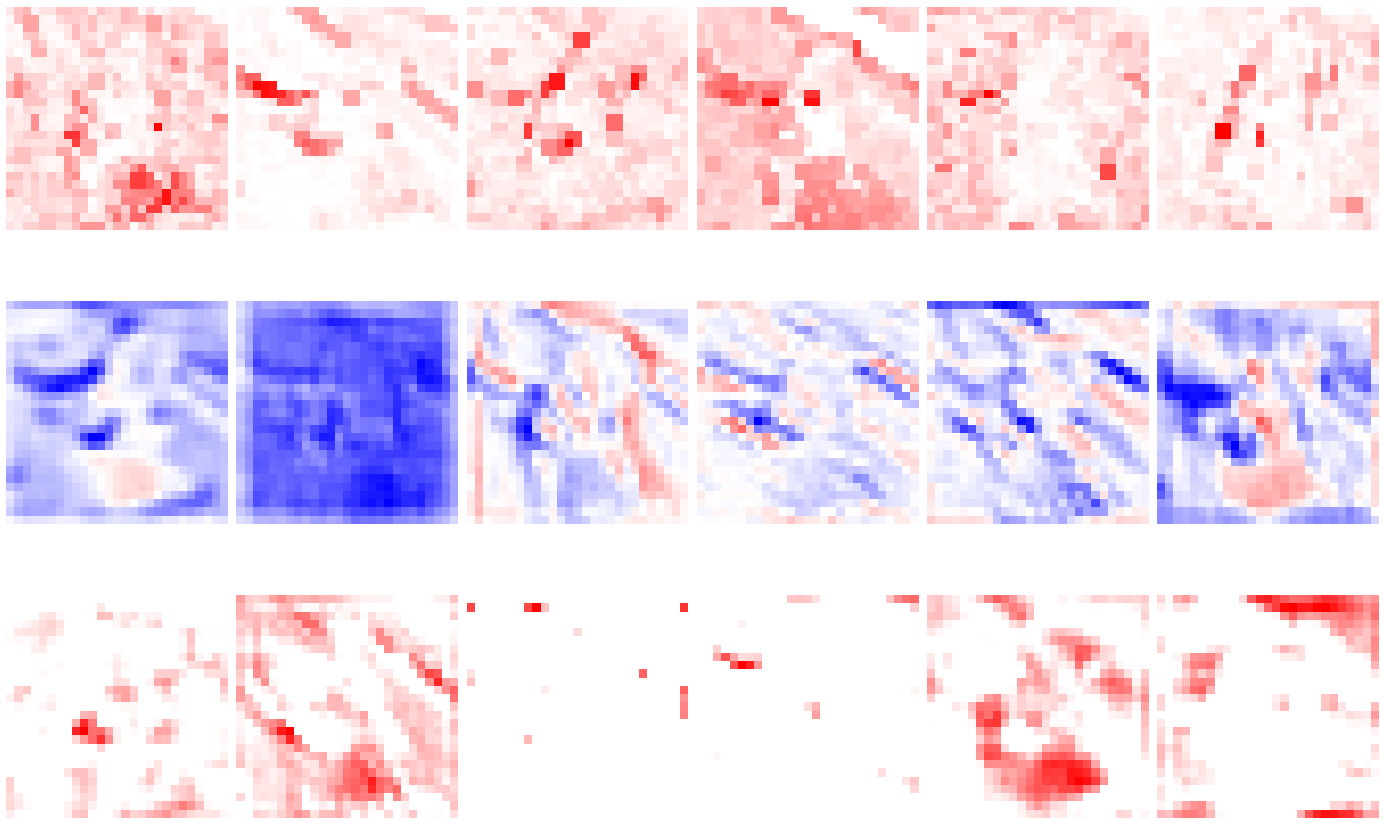
An RGB image with AlexNet (Krizhevsky et al., 2012).



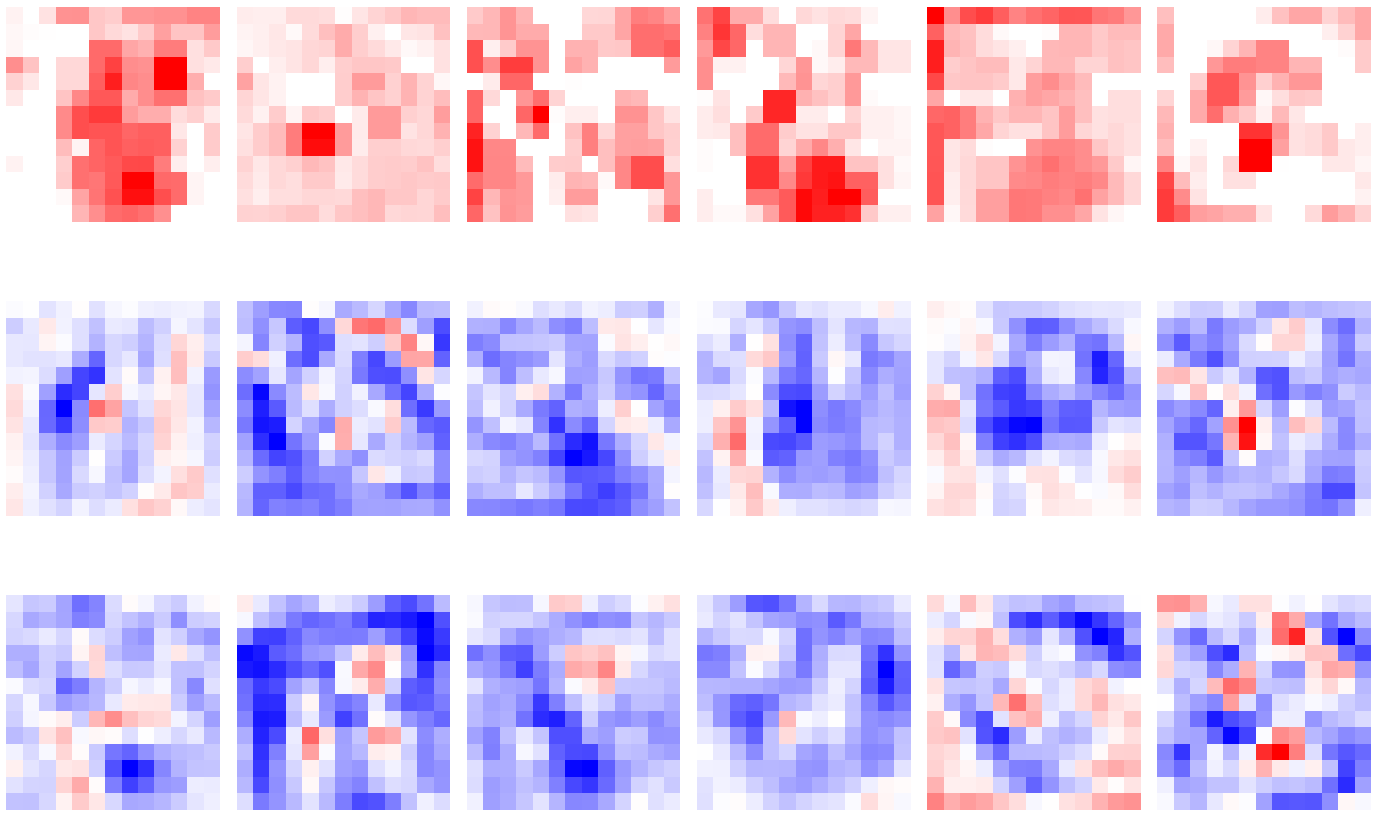
An RGB image with AlexNet (Krizhevsky et al., 2012).



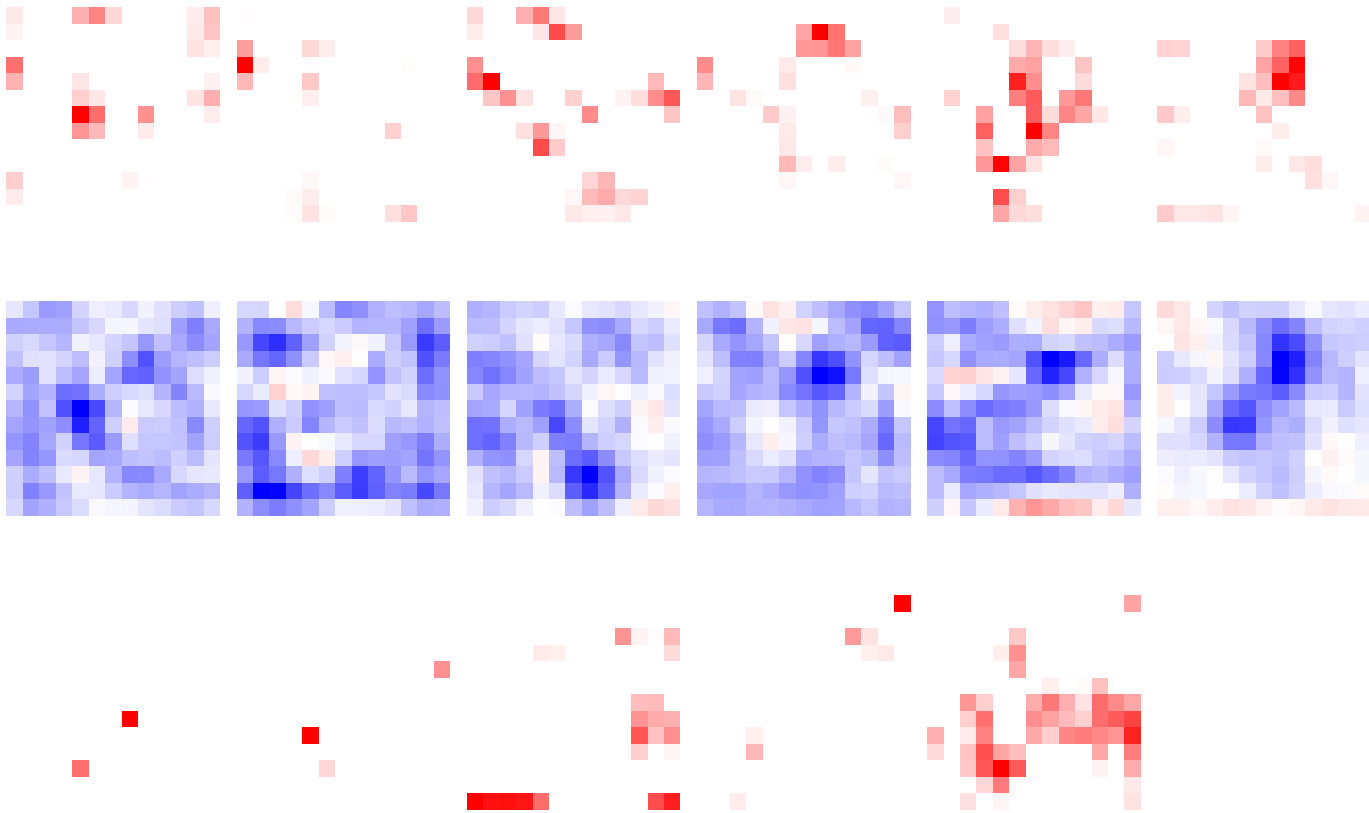
An RGB image with AlexNet (Krizhevsky et al., 2012).



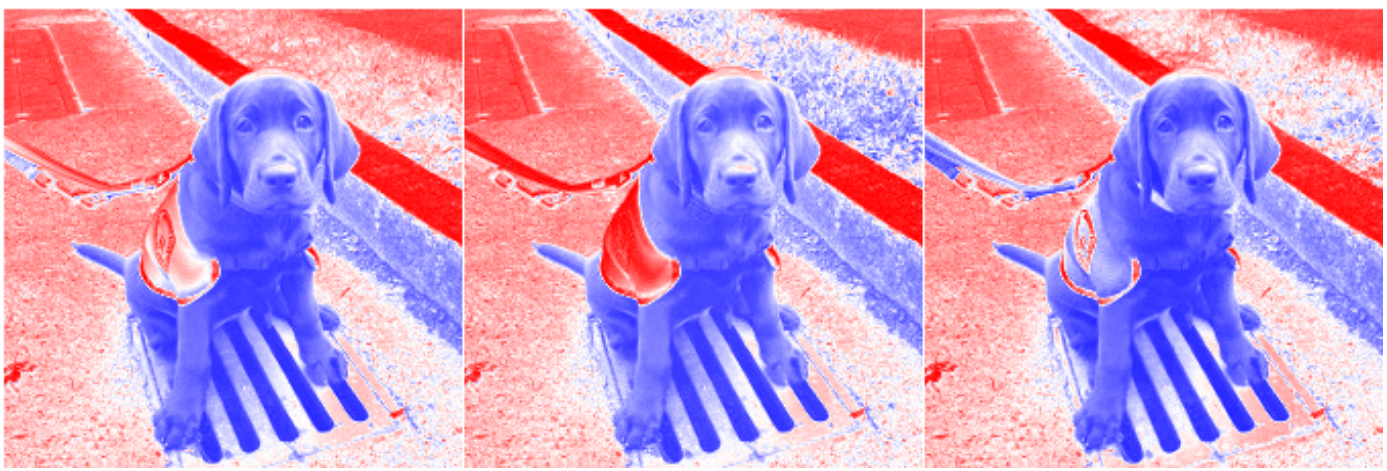
An RGB image with AlexNet (Krizhevsky et al., 2012).



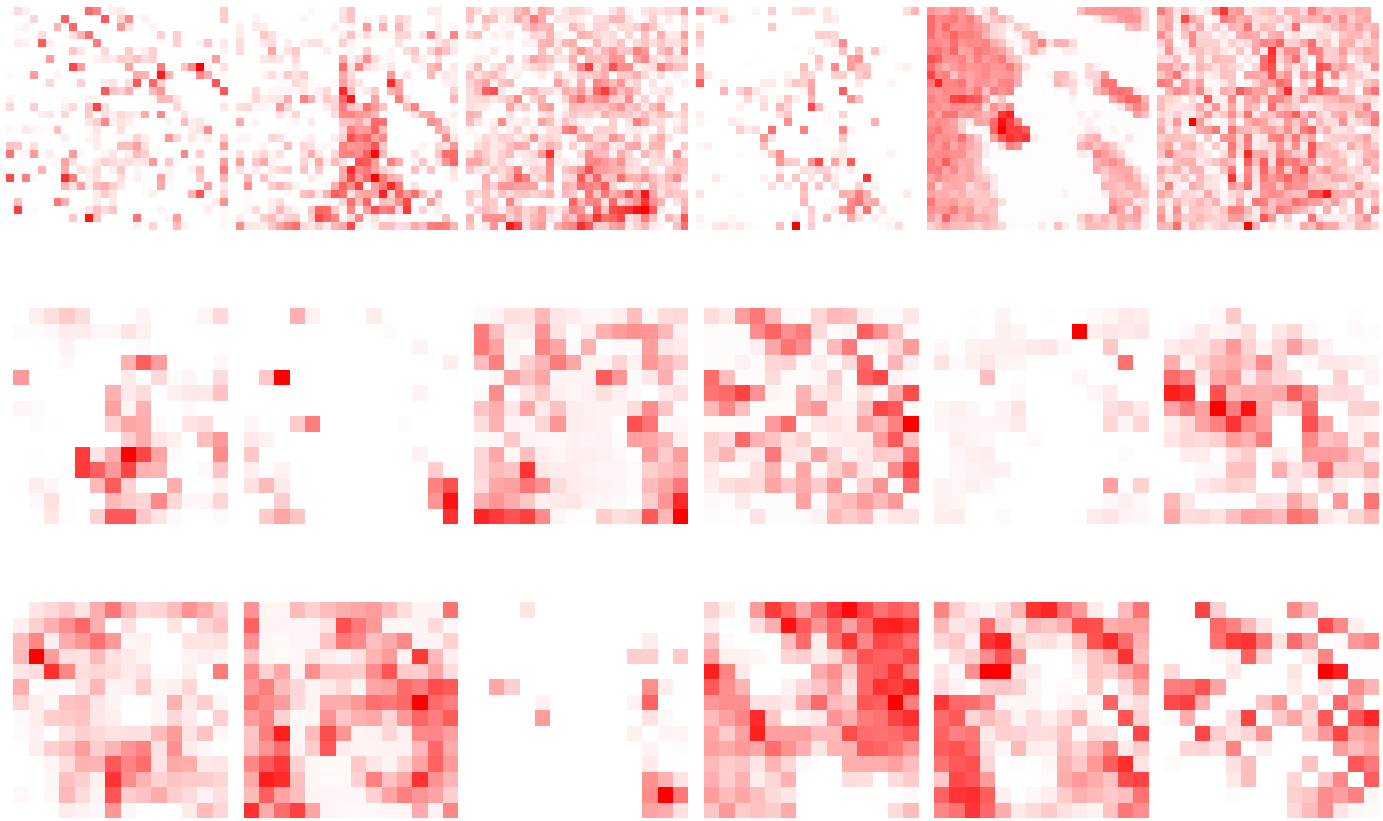
An RGB image with AlexNet (Krizhevsky et al., 2012).



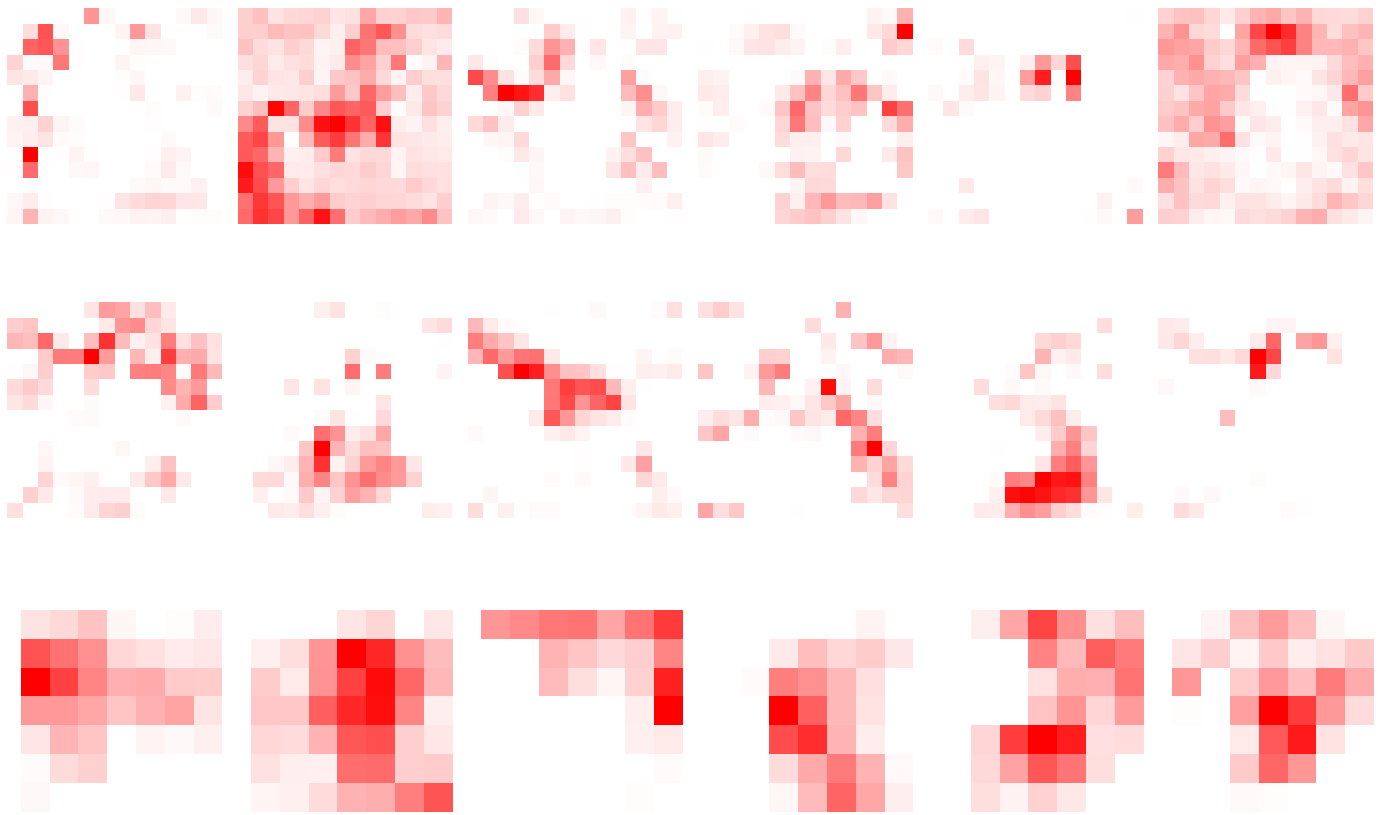
ILSVRC12 with ResNet152 (He et al., 2015).



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.

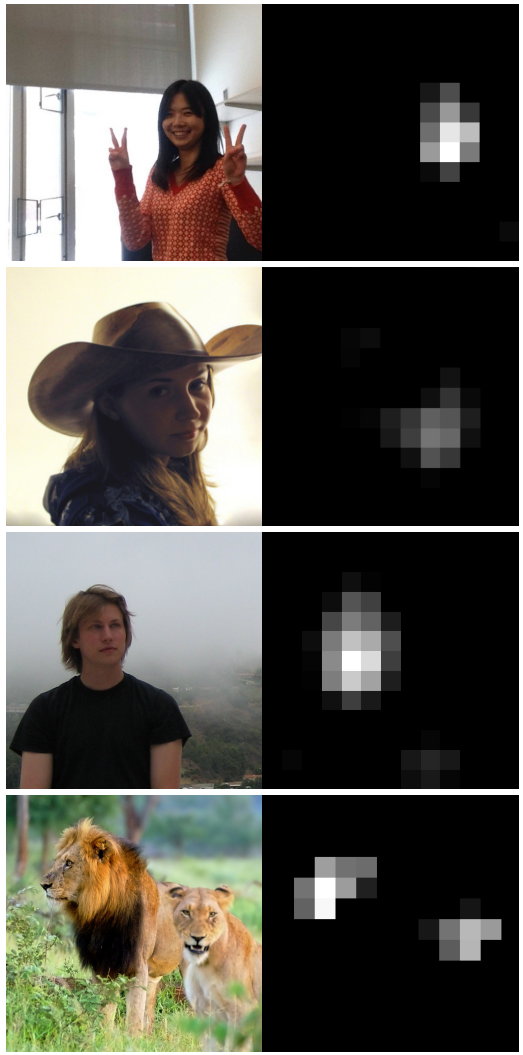
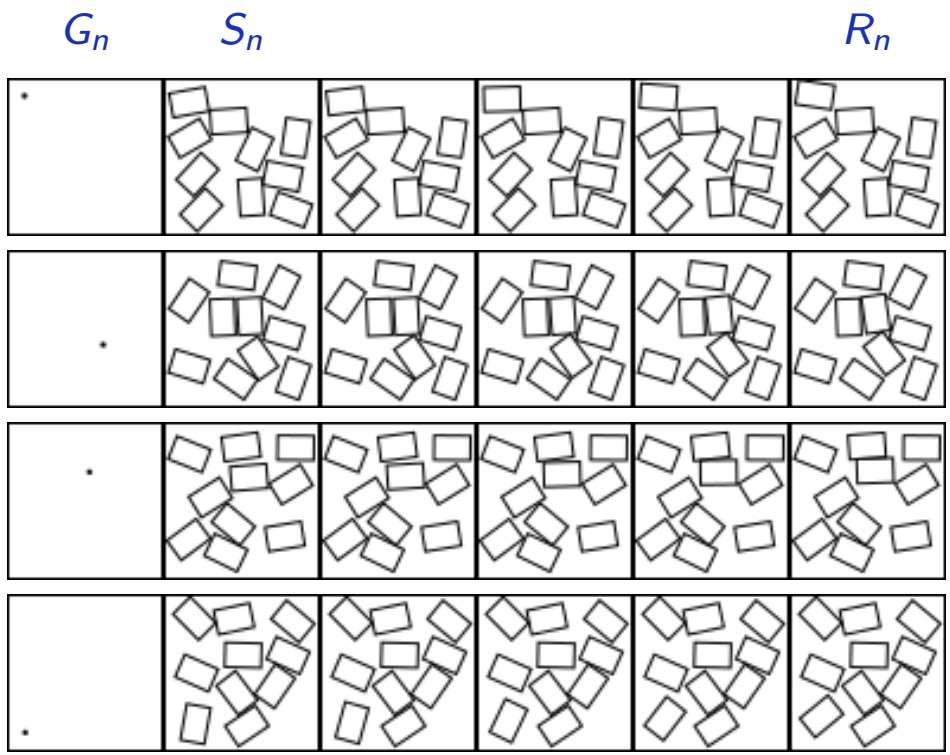


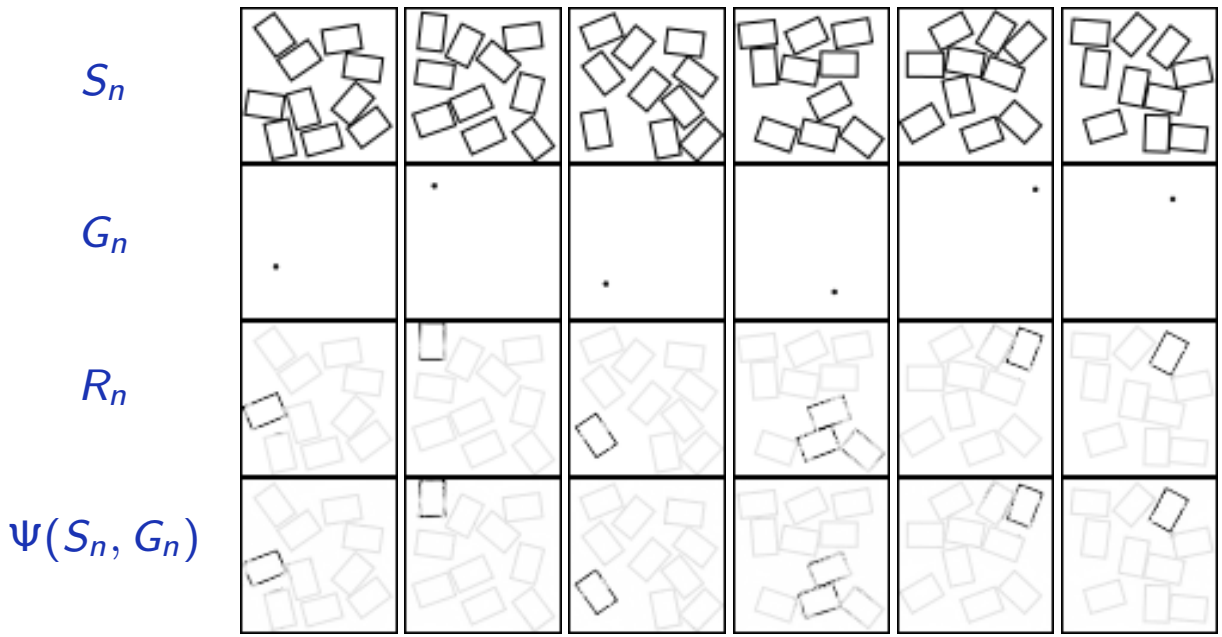
Figure 2. A view of the 13×13 activations of the 151st channel on the conv5 layer of a deep neural network trained on ImageNet, a dataset that does not contain a face class, but does contain many images with faces. The channel responds to human and animal faces and is robust to changes in scale, pose, 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)

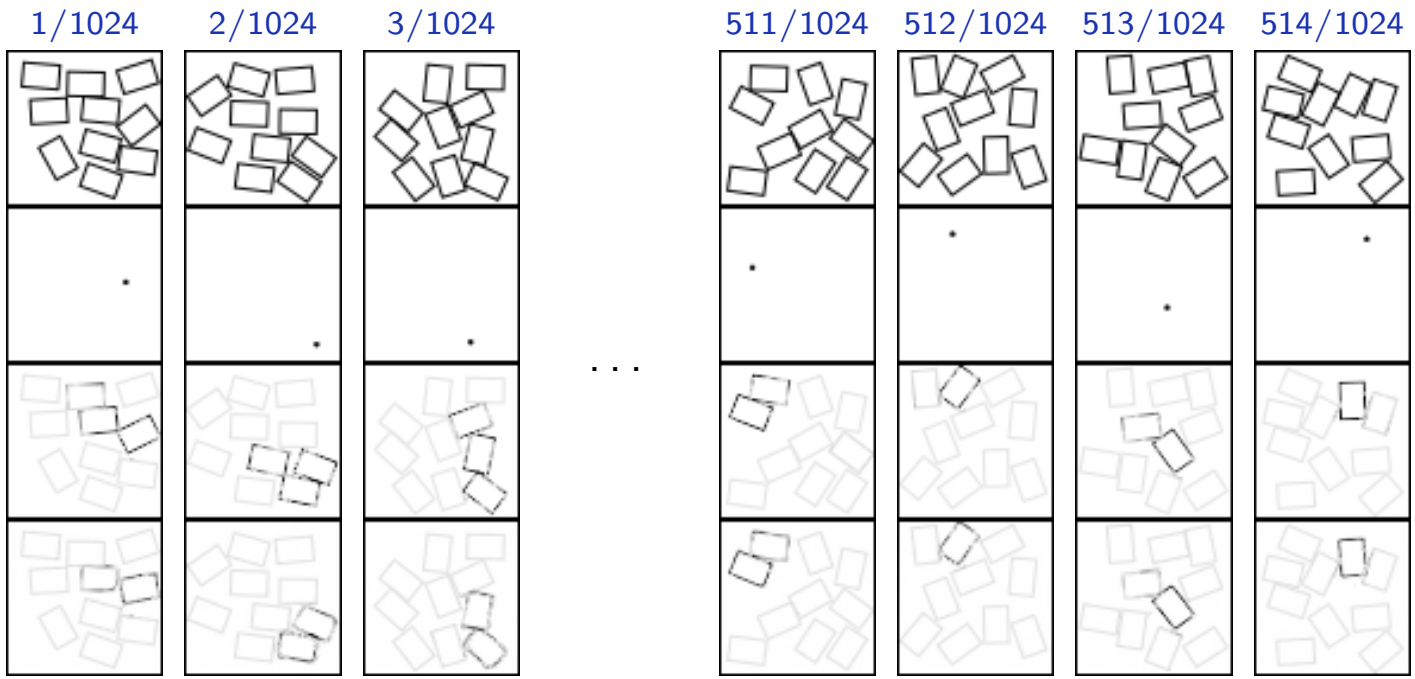
Prediction of 2d dynamics with a 18 layer residual network.



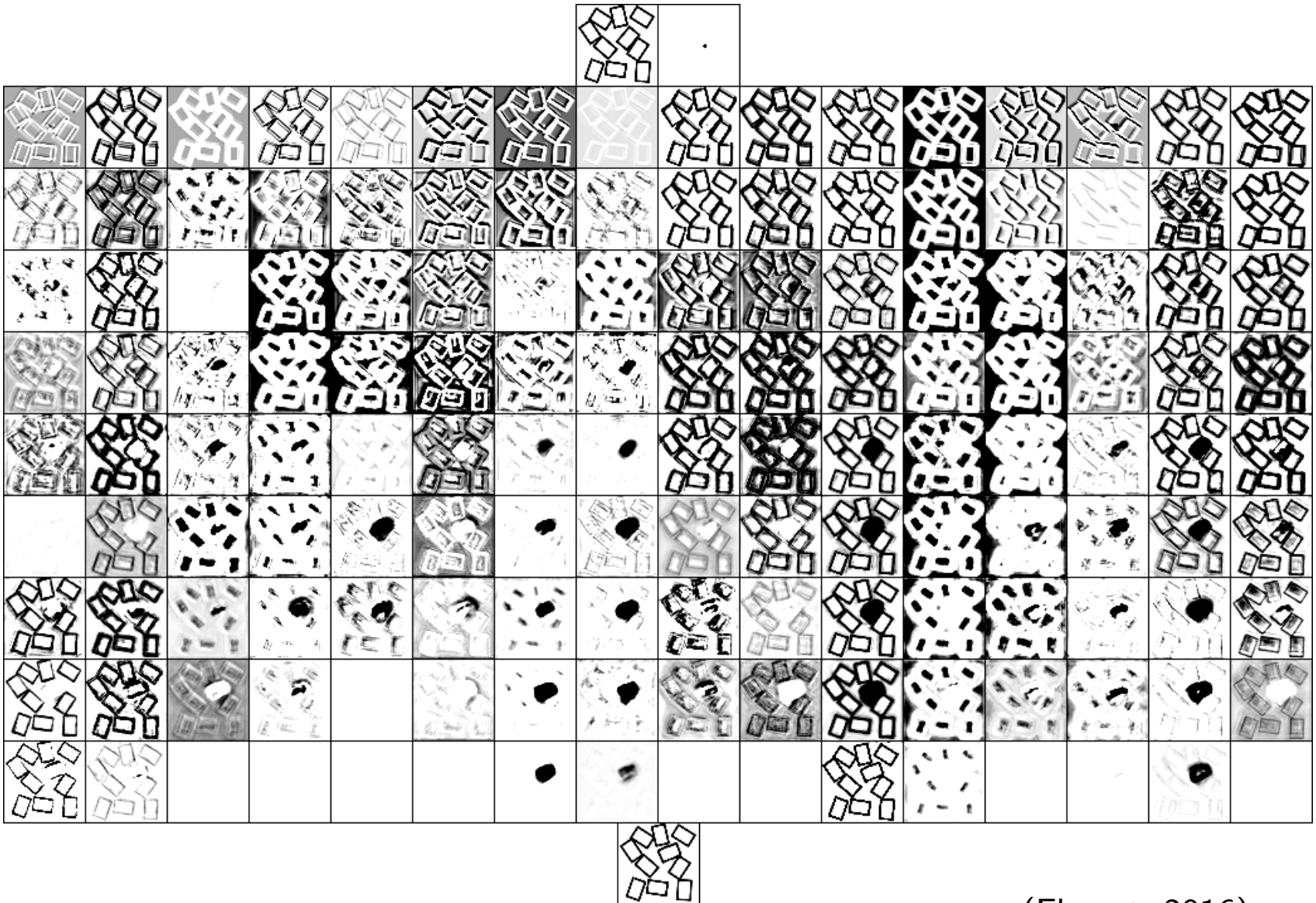
(Fleuret, 2016)



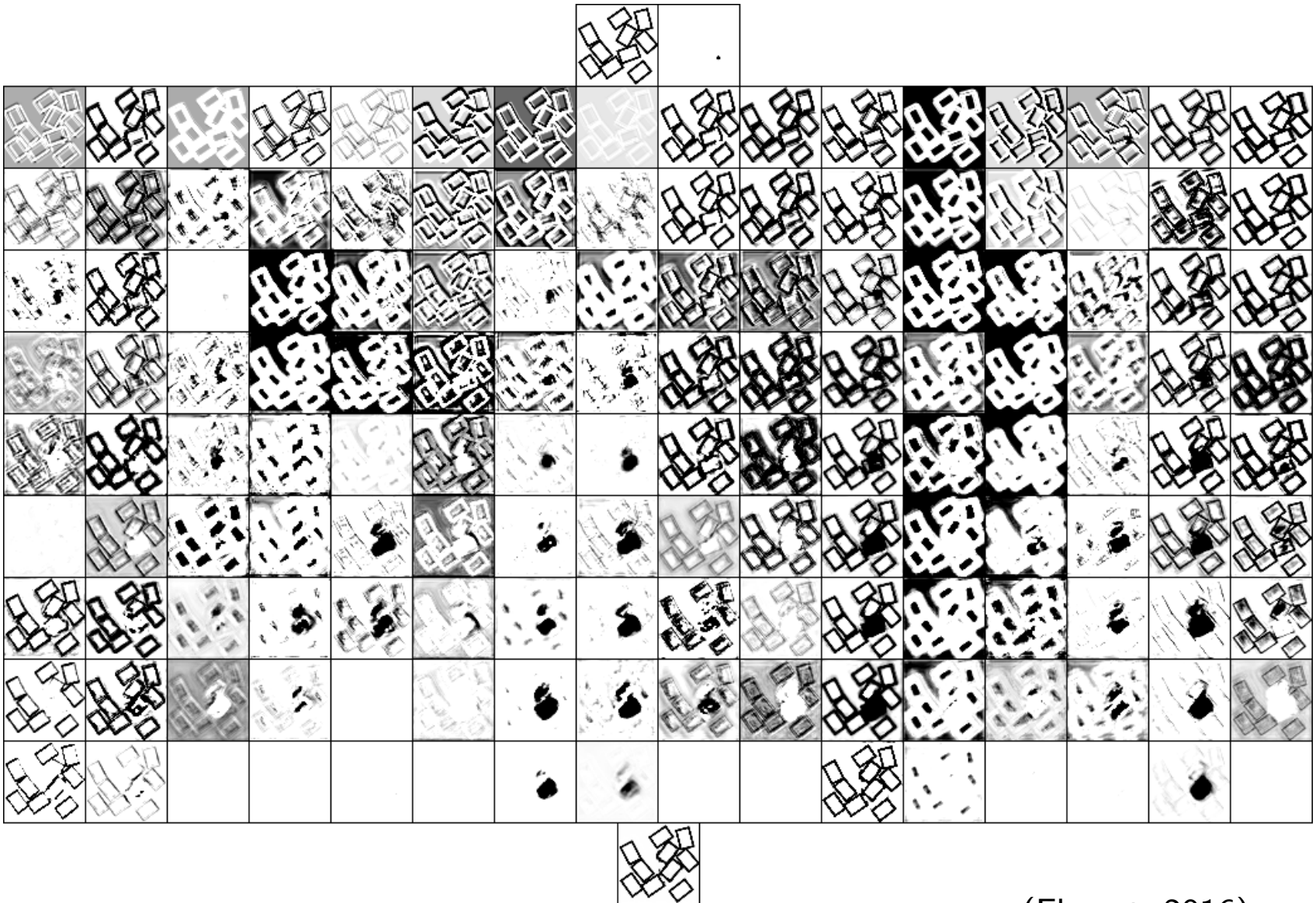
(Fleuret, 2016)



(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 require 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.

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. A popular one is t-SNE developed by van der Maaten and Hinton (2008).

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 distances to close neighbors of each point are preserved.

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 distances to close neighbors of each point are preserved.

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

The scikit-learn toolbox

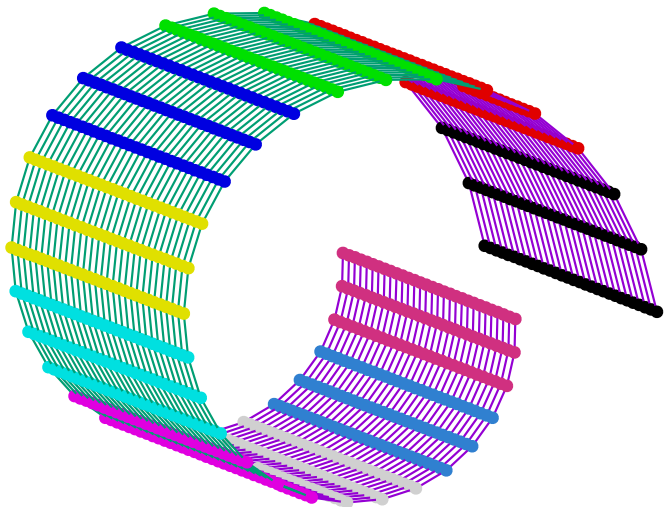
http://scikit-learn.org/

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

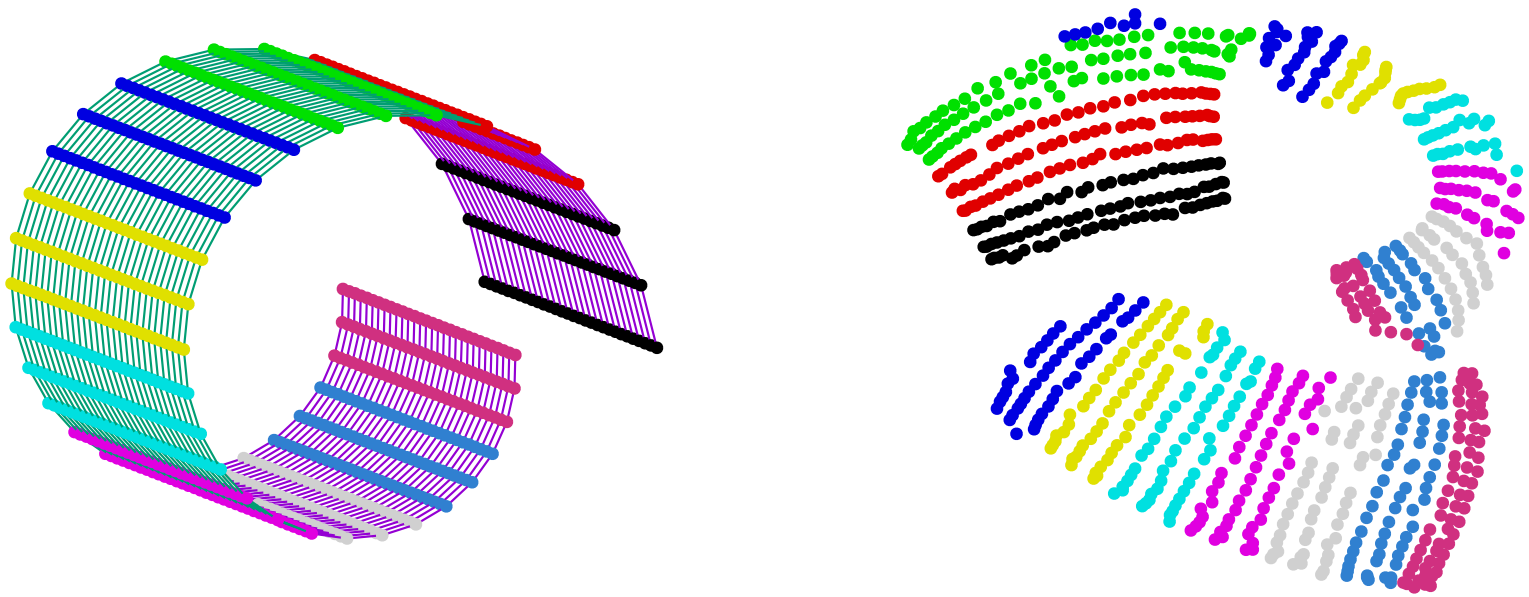
The only catch to use it in PyTorch is the conversions to and from numpy arrays.

```
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 = torch.from_numpy(y_np)
```

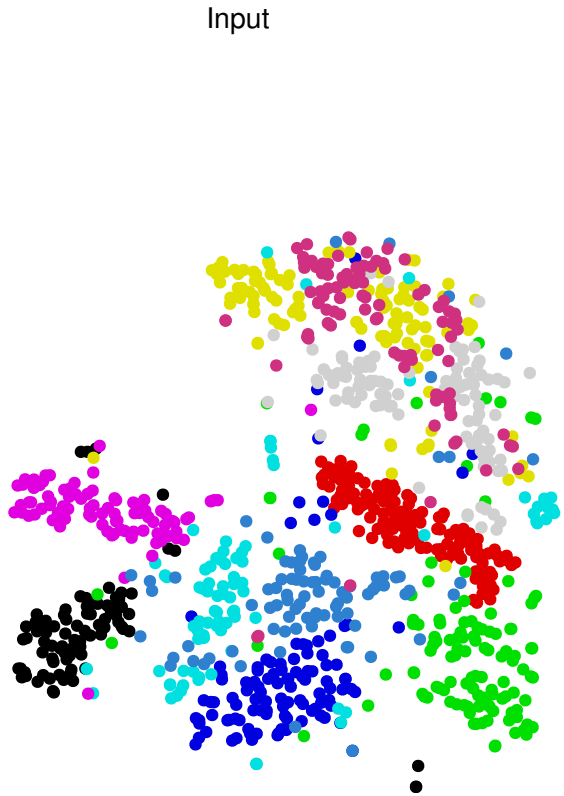
`n_components` specifies the embedding dimension and `perplexity` states [crudely] how many points are considered neighbors of each point.



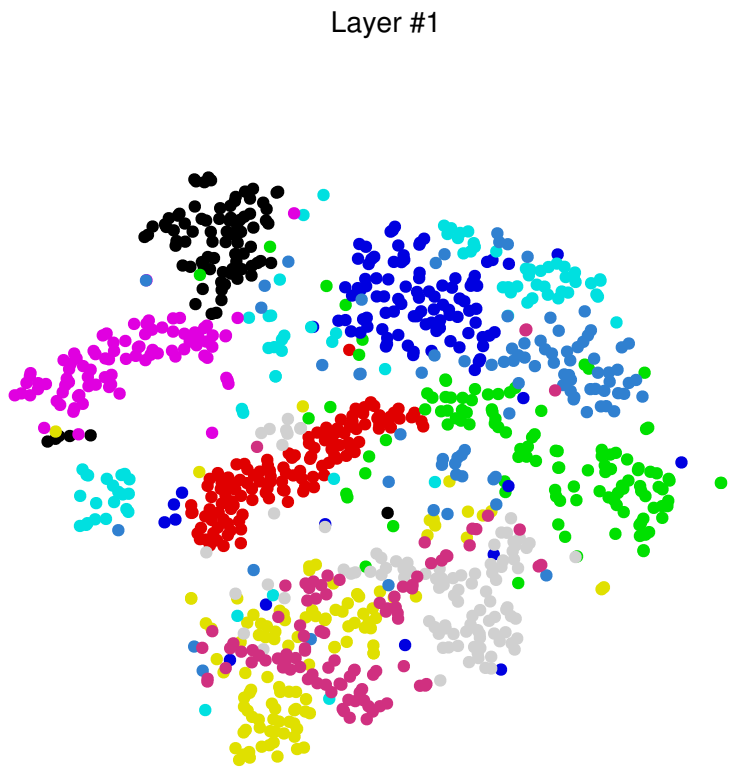
t-SNE unrolling of the swiss roll (with one noise dimension)



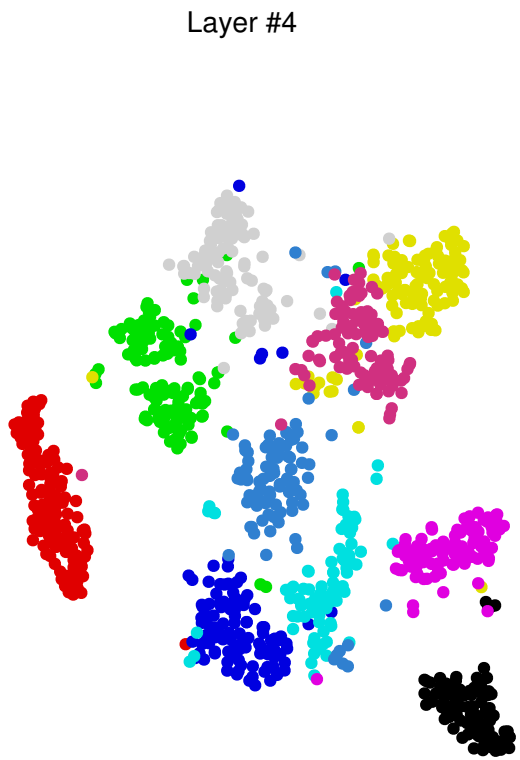
t-SNE unrolling of the swiss roll (with one noise dimension)



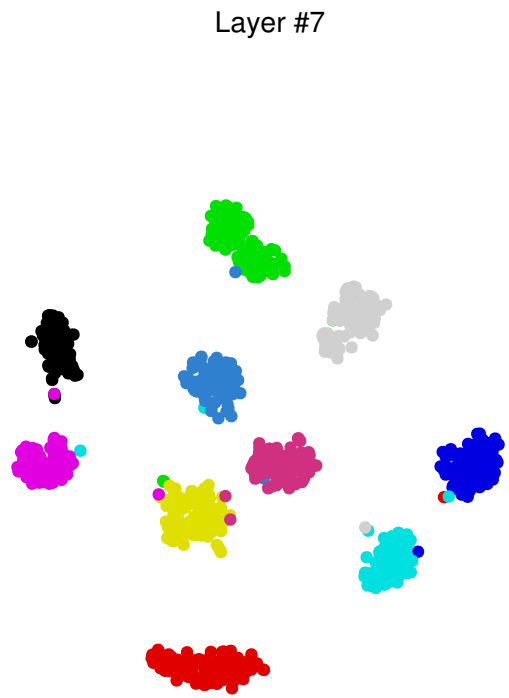
t-SNE for LeNet on MNIST



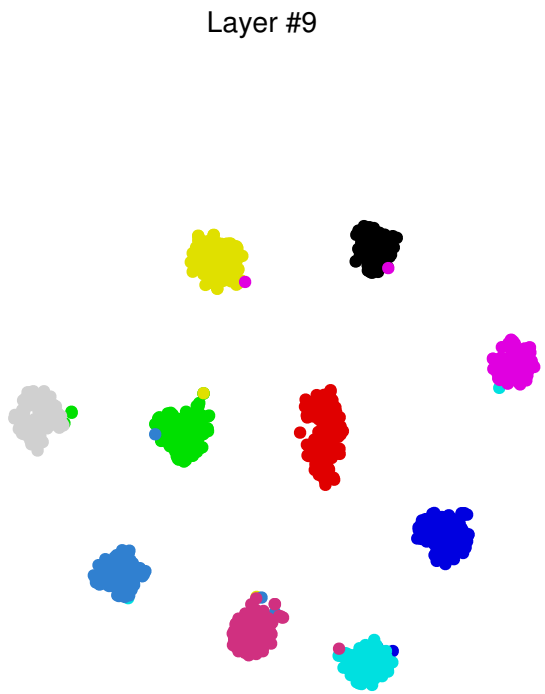
t-SNE for LeNet on MNIST



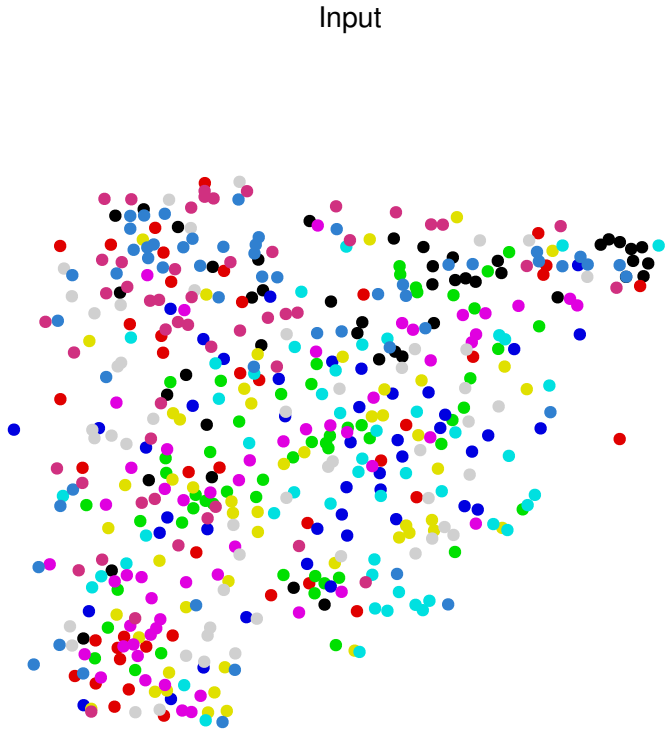
t-SNE for LeNet on MNIST



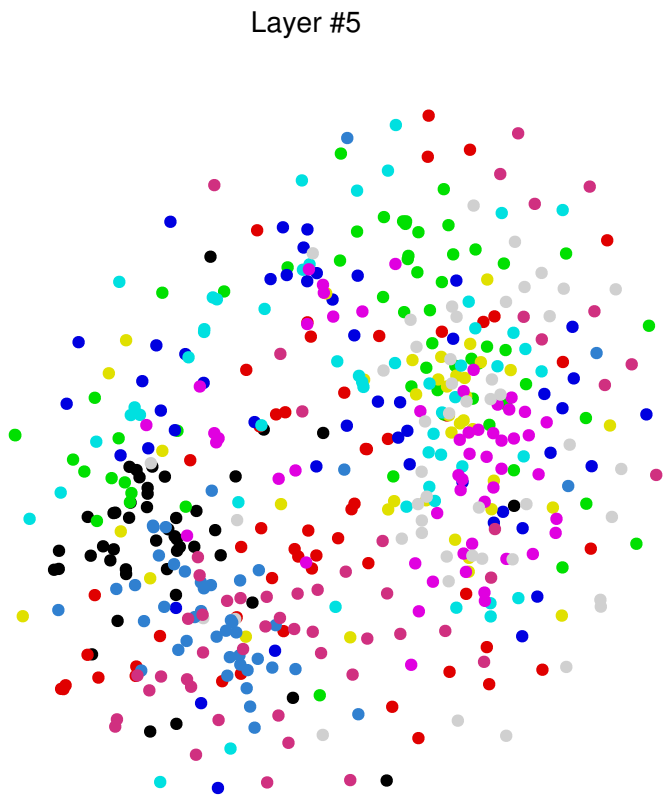
t-SNE for LeNet on MNIST



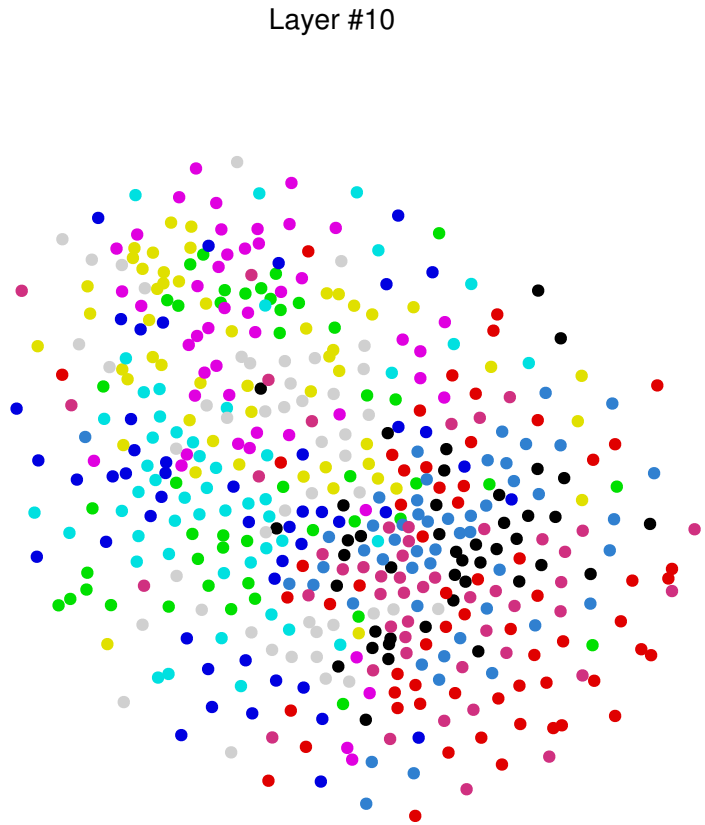
t-SNE for LeNet on MNIST



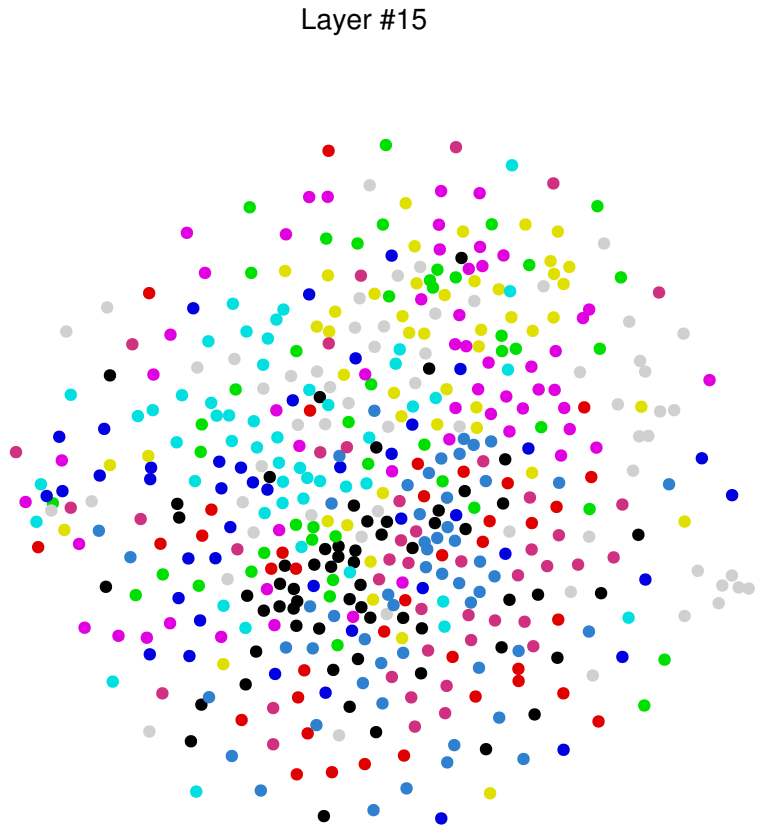
t-SNE for an home-baked resnet (no pooling, 66 layers) CIFAR10



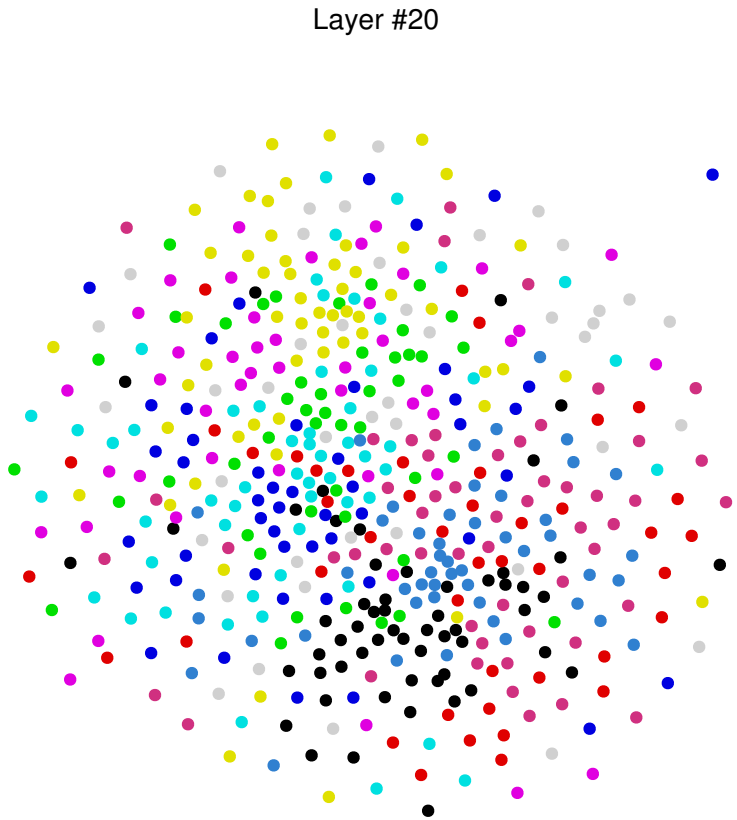
t-SNE for an home-baked resnet (no pooling, 66 layers) CIFAR10



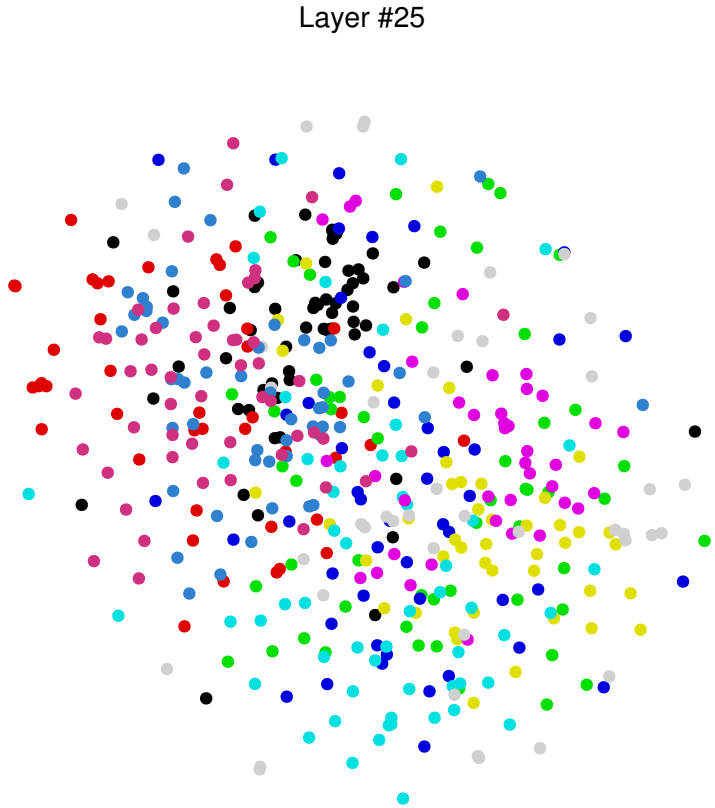
t-SNE for an home-baked resnet (no pooling, 66 layers) CIFAR10



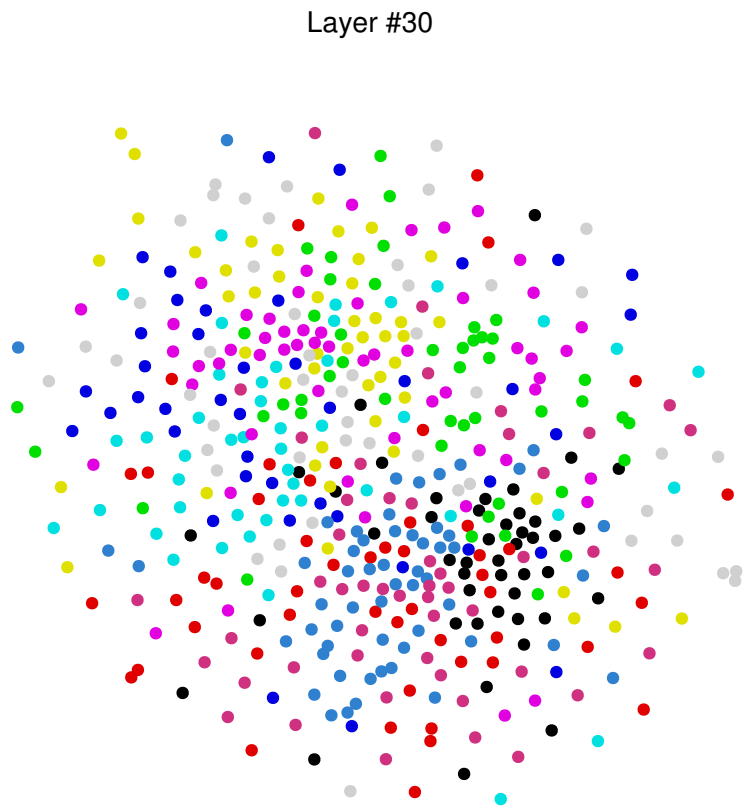
t-SNE for an home-baked resnet (no pooling, 66 layers) CIFAR10



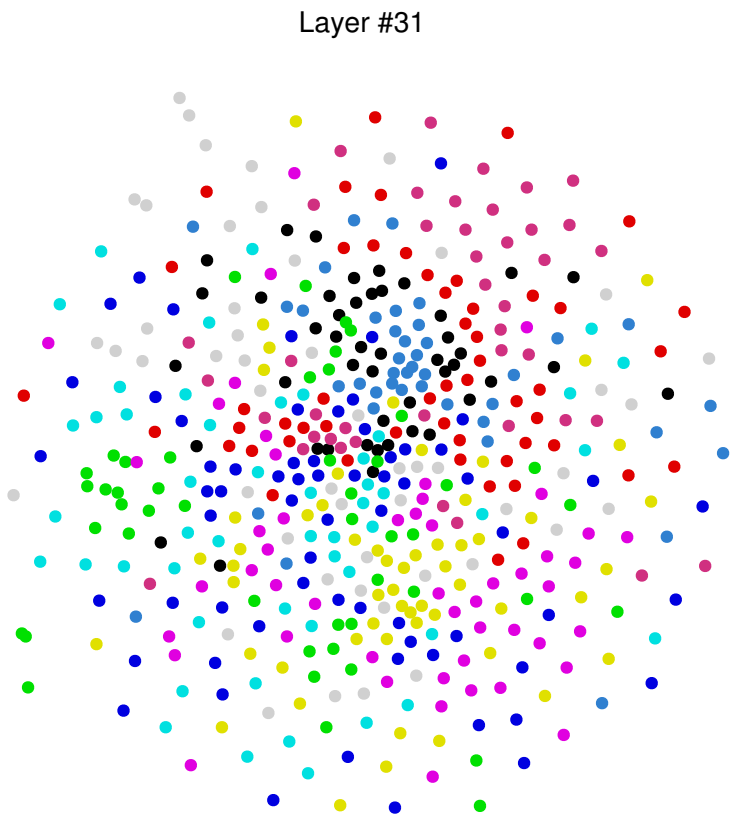
t-SNE for an home-baked resnet (no pooling, 66 layers) CIFAR10



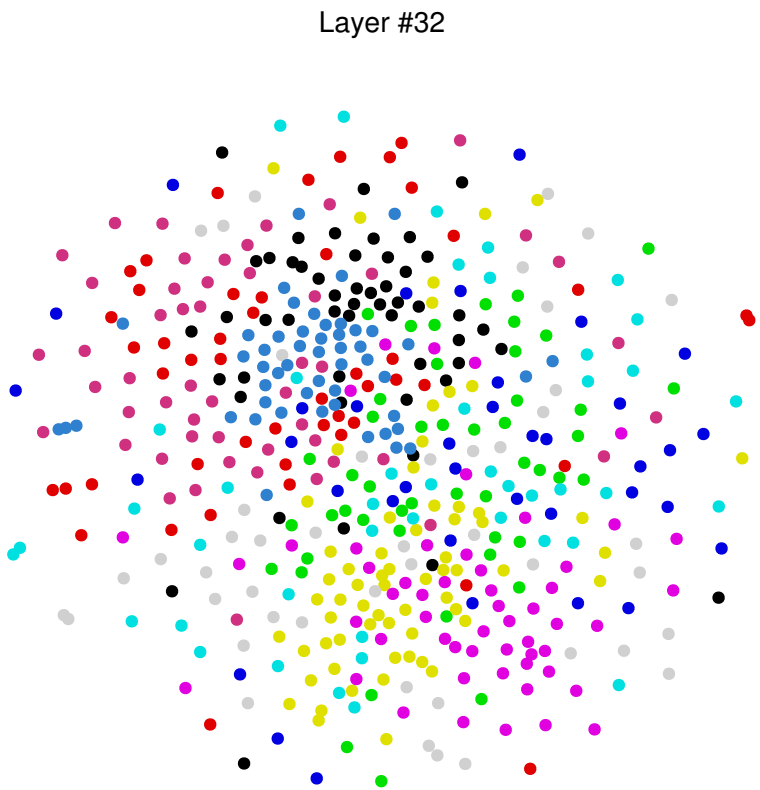
t-SNE for an home-baked resnet (no pooling, 66 layers) CIFAR10



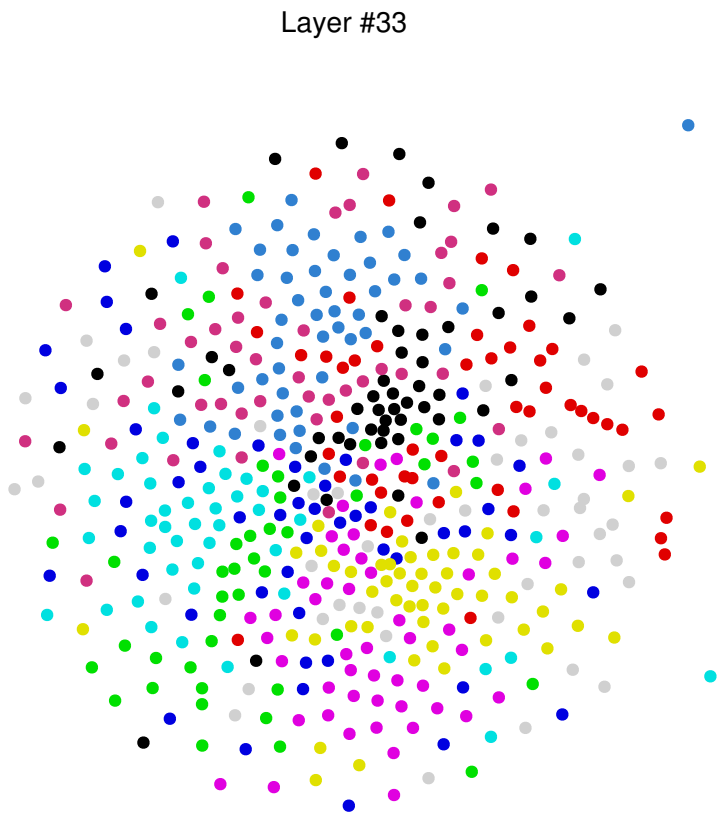
t-SNE for an home-baked resnet (no pooling, 66 layers) CIFAR10



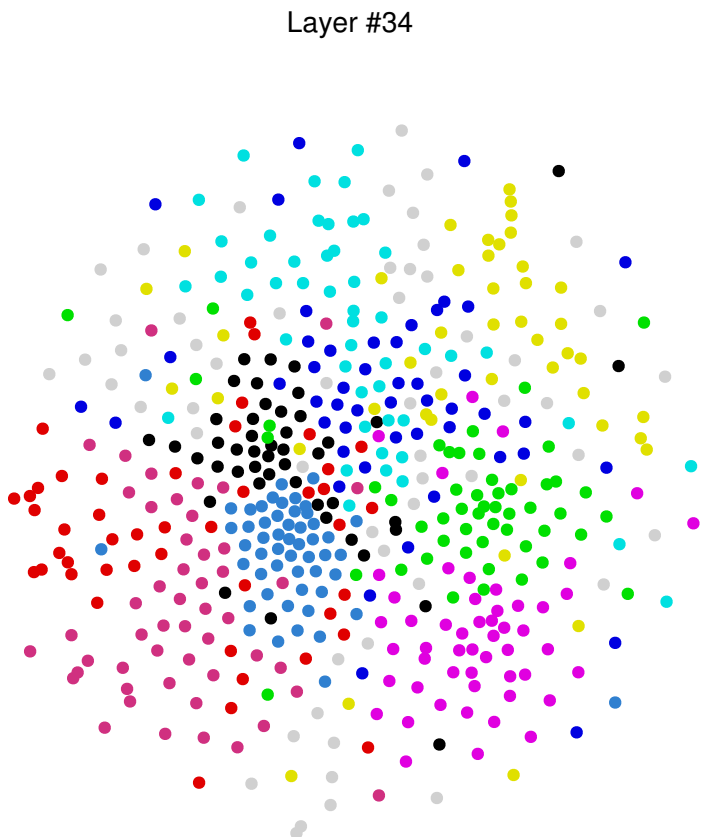
t-SNE for an home-baked resnet (no pooling, 66 layers) CIFAR10



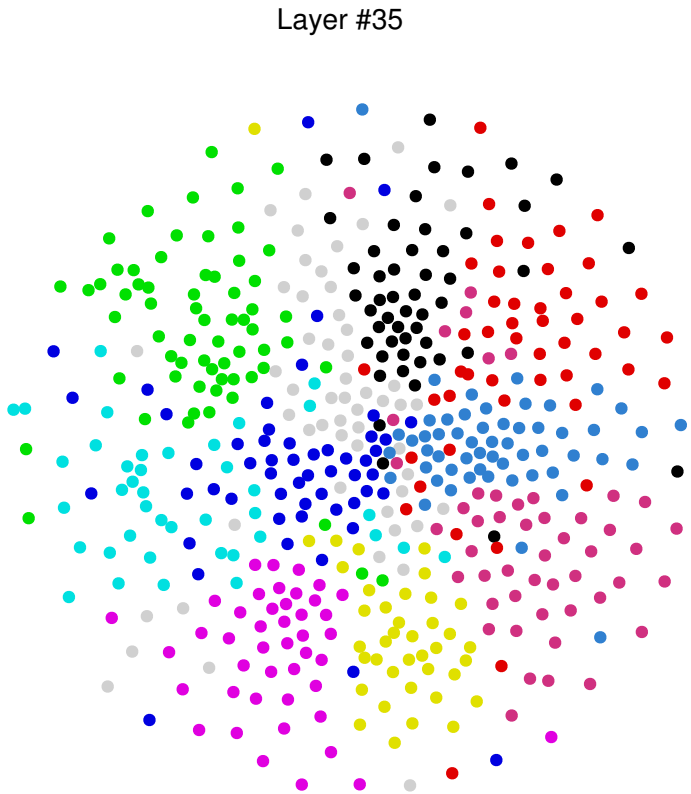
t-SNE for an home-baked resnet (no pooling, 66 layers) CIFAR10



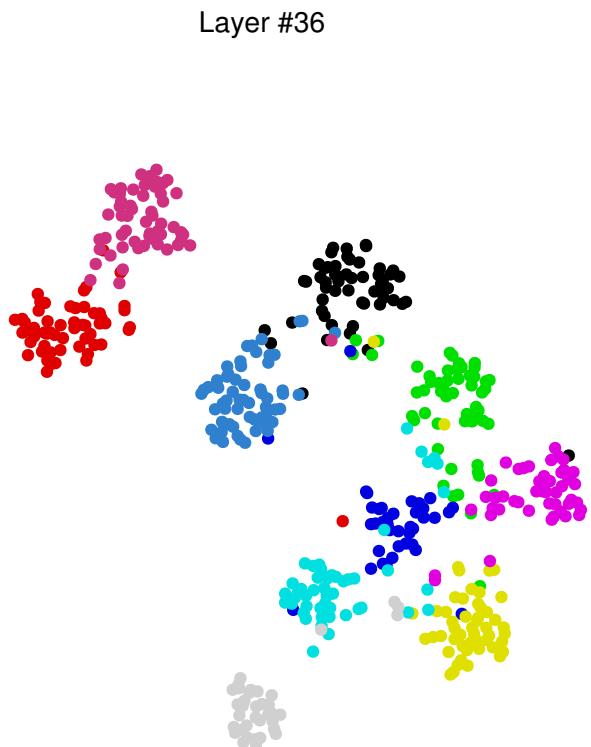
t-SNE for an home-baked resnet (no pooling, 66 layers) CIFAR10



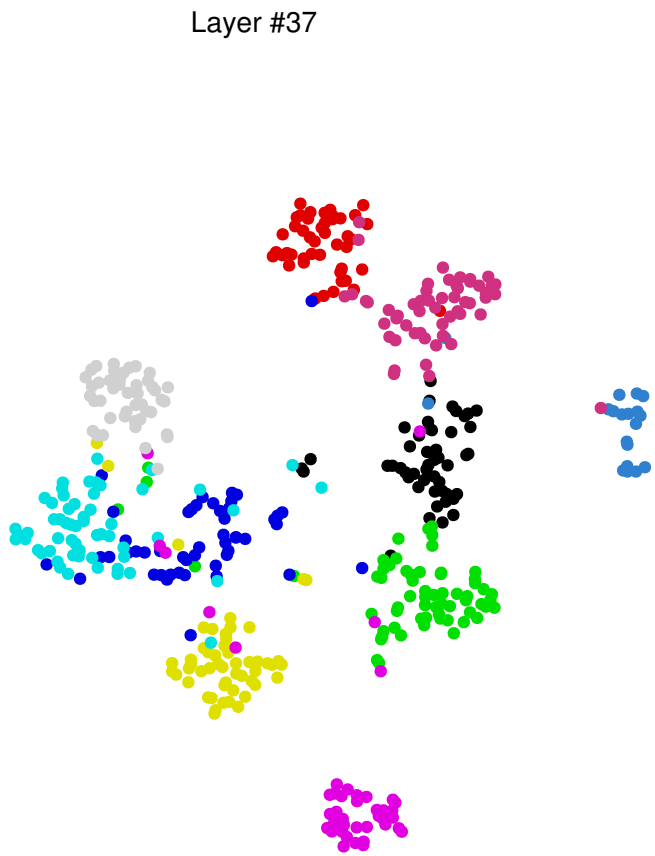
t-SNE for an home-baked resnet (no pooling, 66 layers) CIFAR10



t-SNE for an home-baked resnet (no pooling, 66 layers) CIFAR10



t-SNE for an home-baked resnet (no pooling, 66 layers) CIFAR10



t-SNE for an home-baked resnet (no pooling, 66 layers) CIFAR10

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 by computing the difference between:

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

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 by computing the difference between:

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

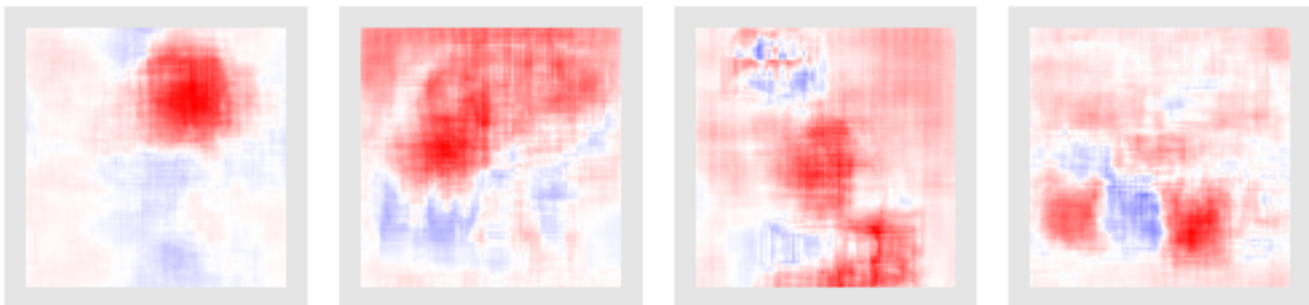
This is computationally intensive since it requires as many forward passes as there are locations of the occlusion mask, ideally the number of pixels.

Original images

Occlusion mask 32×32

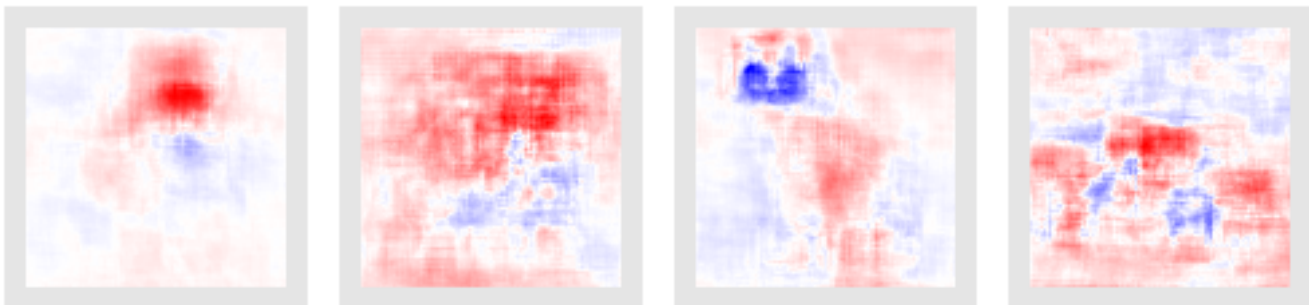
Original images

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



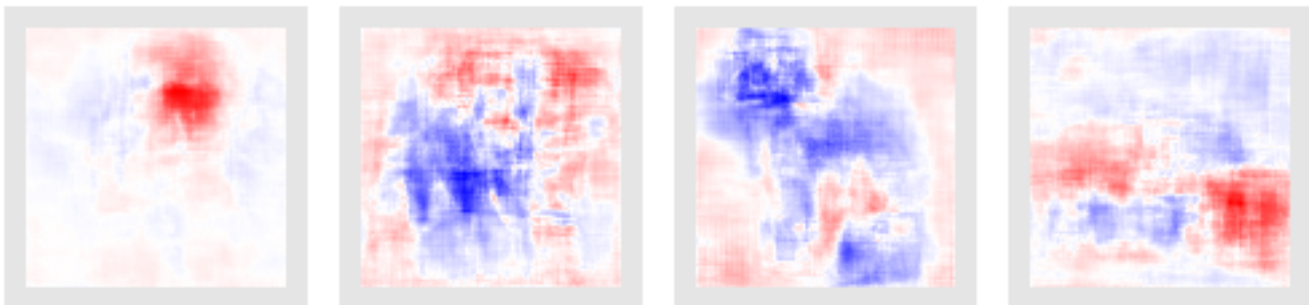
Original images

Occlusion sensitivity, mask 32×32, stride of 2, VGG16



Original images

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



Saliency maps

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

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

where f is the activation of the output unit with maximum response, and $|x$ stresses that the gradient is computed with respect to the input x and not as usual with respect to the parameters w.

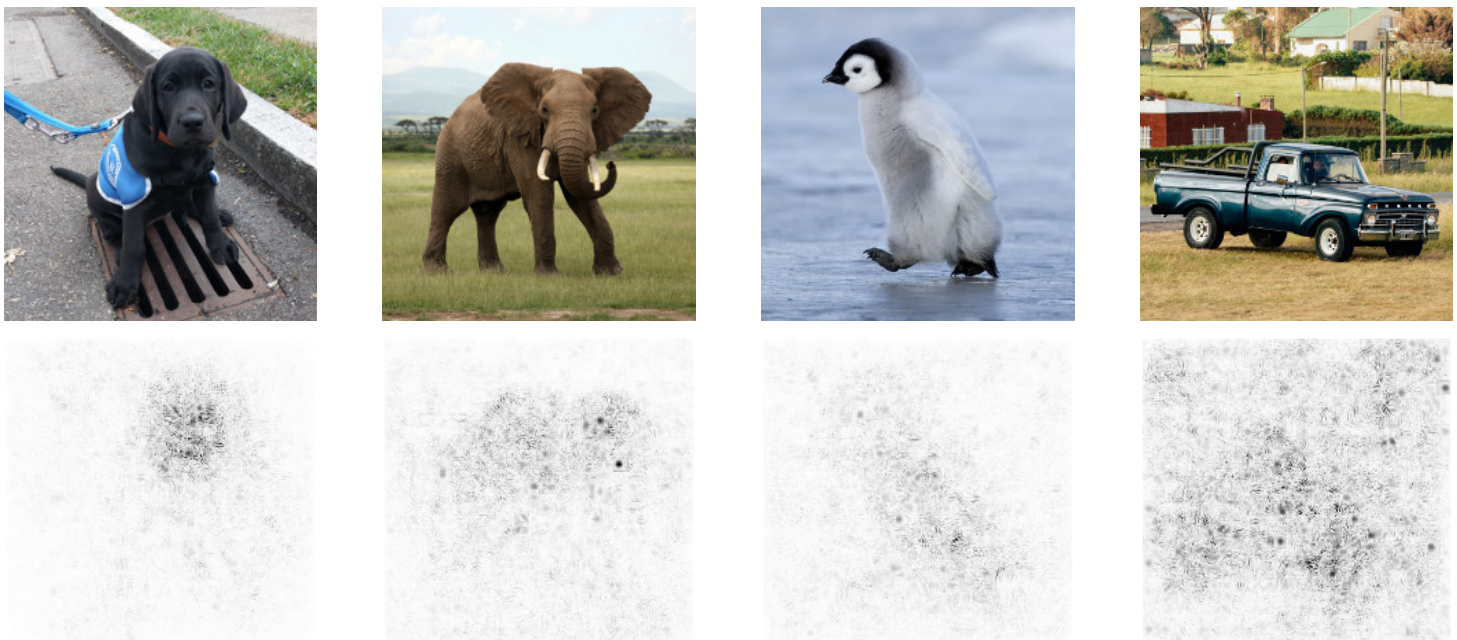
This can be implemented by specifying that we need the gradient with respect to the input. We use here the correct unit, not the maximum response one.

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

```
input = Variable(img, requires_grad = True)
output = model(input)
loss = nllloss(output, target)
grad_input, = torch.autograd.grad(loss, 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.

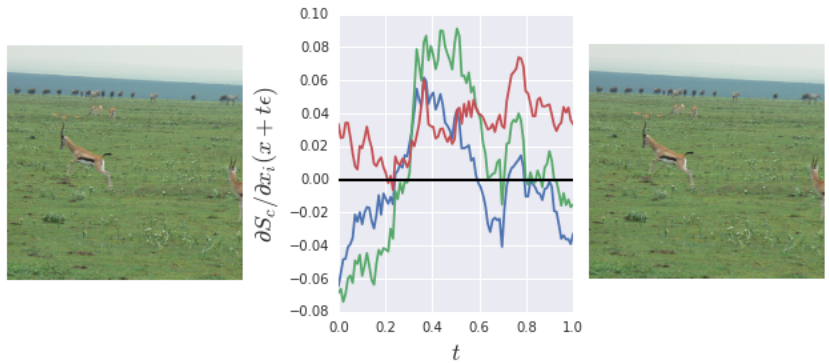


Figure 2. The partial derivative of S_c with respect to the RGB values of a single pixel as a fraction of the maximum entry in the gradient vector, $\max_i \frac{\partial S_c}{\partial x_i}(t)$, (middle plot) as one slowly moves away from a baseline image x (left plot) to a fixed location $x + \epsilon$ (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.

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

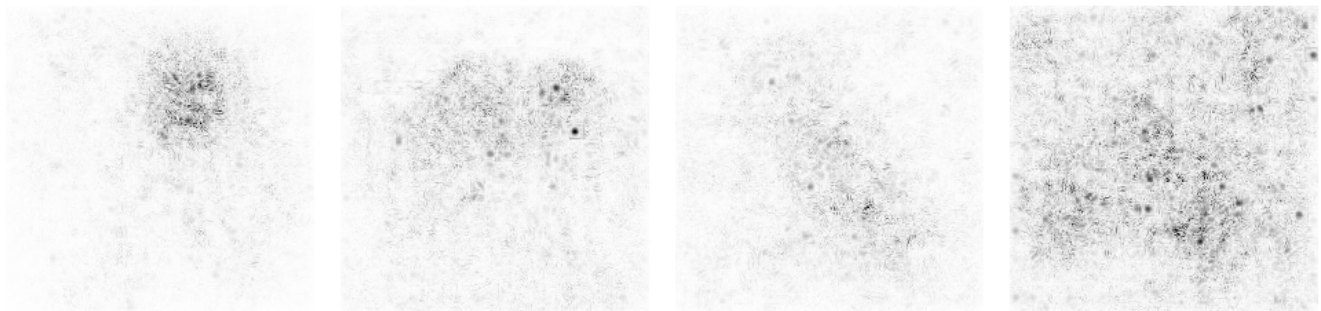
where $\epsilon_1, \ldots, \epsilon_N$ are i.i.d of distribution $\mathcal{N}(0, \sigma^2 \mathbf{I})$, and σ is a fraction of the gap Δ between the maximum and the minimum of the pixel values.

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

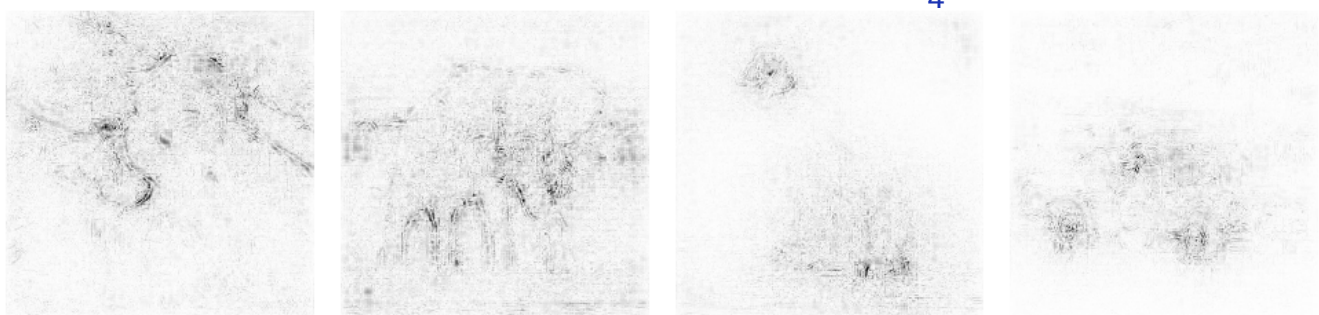
```
nb_smooth = 100
std = smooth_std * (img.max() - img.min())
acc_grad = img.new(img.size()).zero_()

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 = Variable(noisy_input, requires_grad = True)
    output = model(noisy_input)
    loss = nllloss(output, target)
    grad_input, = torch.autograd.grad(loss, noisy_input)
    acc_grad += grad_input.data

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

Original images

Gradient, AlexNet

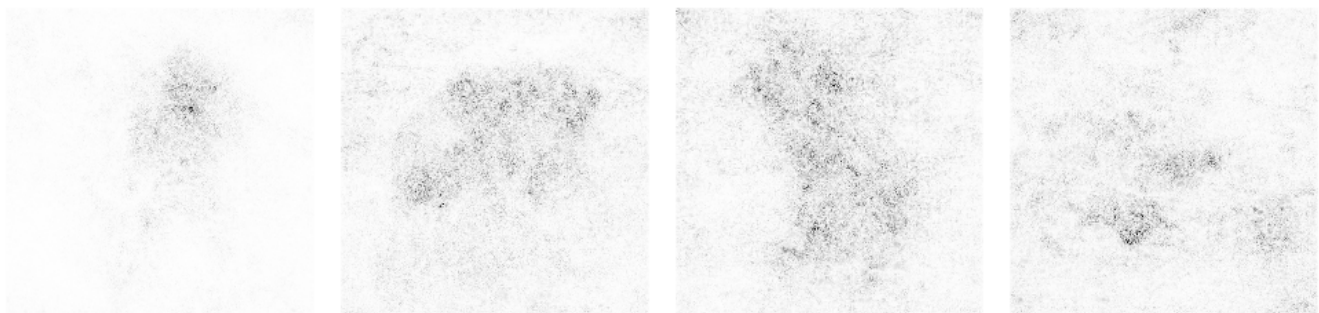


SmoothGrad, AlexNet, $\sigma = \frac{\Delta}{4}$

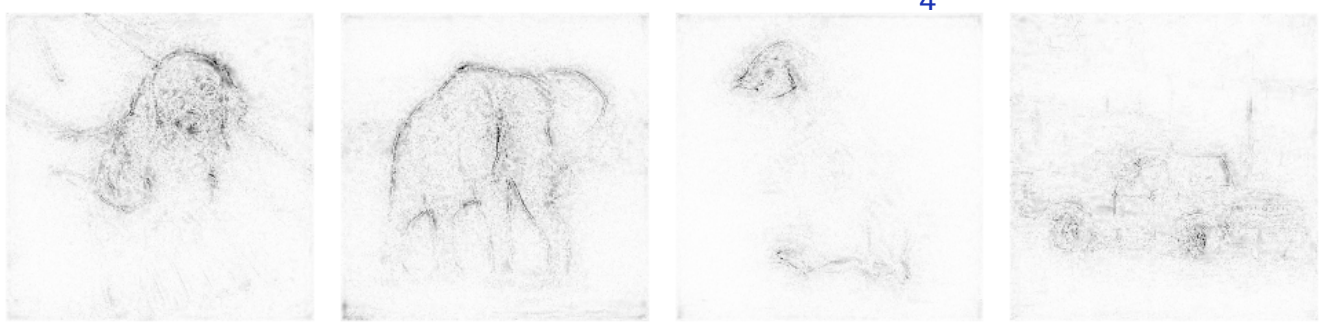


Original images

Gradient, VGG16

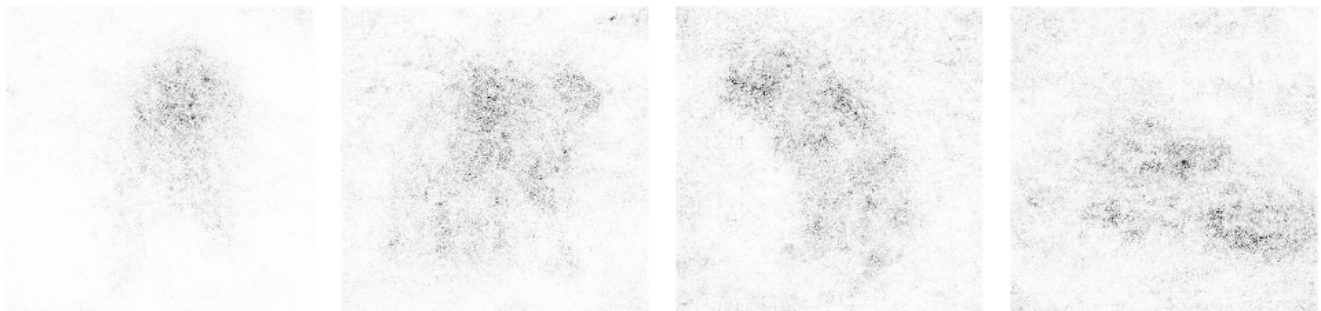


SmoothGrad, VGG16, $\sigma = \frac{\Delta}{4}$

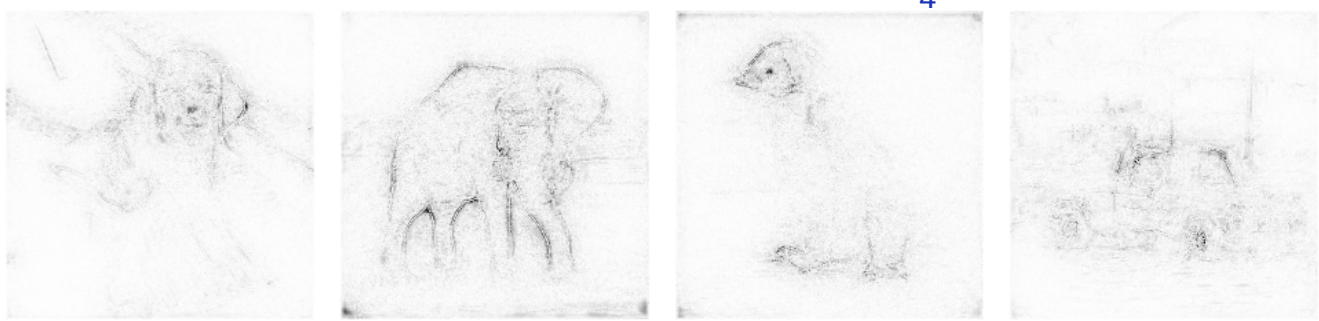


Original images

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

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

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

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

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 = Variable(Tensor([ 1.23, -4.56 ]))
>>> m = nn.ReLU()
>>> m(x)
Variable containing:
 1.2300
 0.0000
[torch.FloatTensor of size 2]
```

For instance

```
>>> x = Variable(Tensor([ 1.23, -4.56 ]))
>>> m = nn.ReLU()
>>> m(x)
Variable containing:
 1.2300
 0.0000
[torch.FloatTensor of size 2]

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

For instance

```
>>> x = Variable(Tensor([ 1.23, -4.56 ]))
>>> m = nn.ReLU()
>>> m(x)
Variable containing:
 1.2300
 0.0000
[torch.FloatTensor of size 2]

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

>>> handle.remove()
>>> m(x)
Variable containing:
 1.2300
 0.0000
[torch.FloatTensor of size 2]
```

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)
```

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()

    if torch.cuda.is_available(): img = img.cuda()

    input = Variable(img.view(1, img.size(0), img.size(1), img.size(2)), \
                     requires_grad = True)

    output = model(input)
    result, = torch.autograd.grad(output.max(), input)

    result = result.data / result.data.max() + 0.5

    return result
```

Using hooks, we can implement the deconvolution as follows:

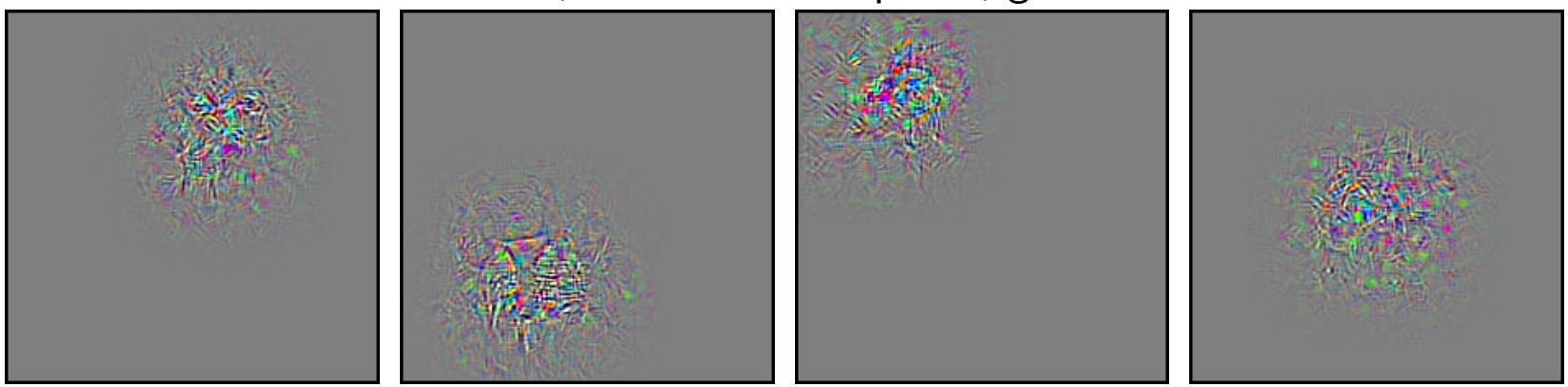
```
def relu_backward_deconv_hook(module, grad_input, grad_output):
    return F.relu(grad_output[0]),

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()

    if torch.cuda.is_available(): img = img.cuda()

    input = Variable(img.view(1, img.size(0), img.size(1), img.size(2)), \
                     requires_grad = True)

    output = model(input)
    result, = torch.autograd.grad(output.max(), input)

    result = result.data / result.data.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)
```

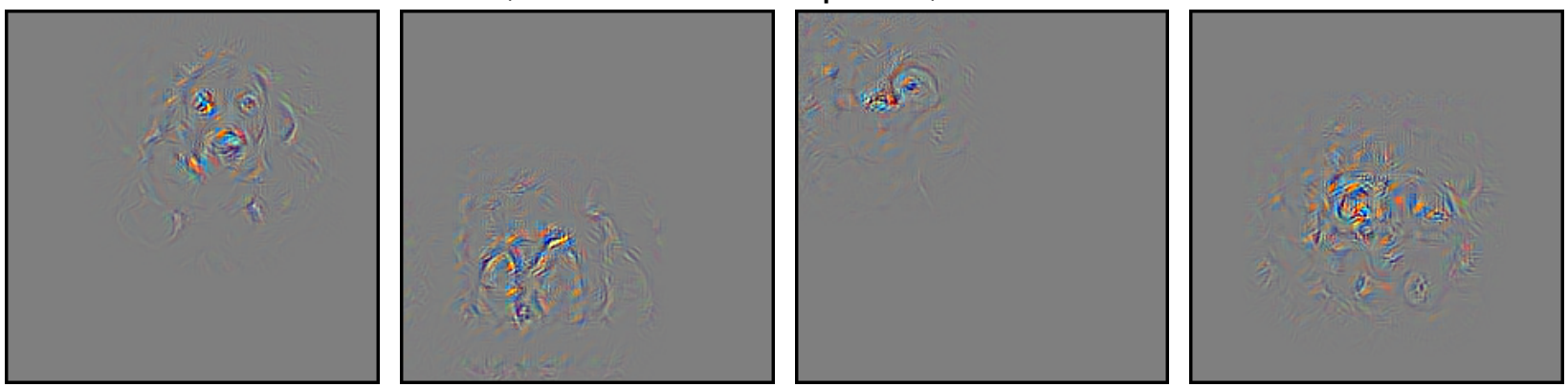
Original images

AlexNet, max feature response, gradient



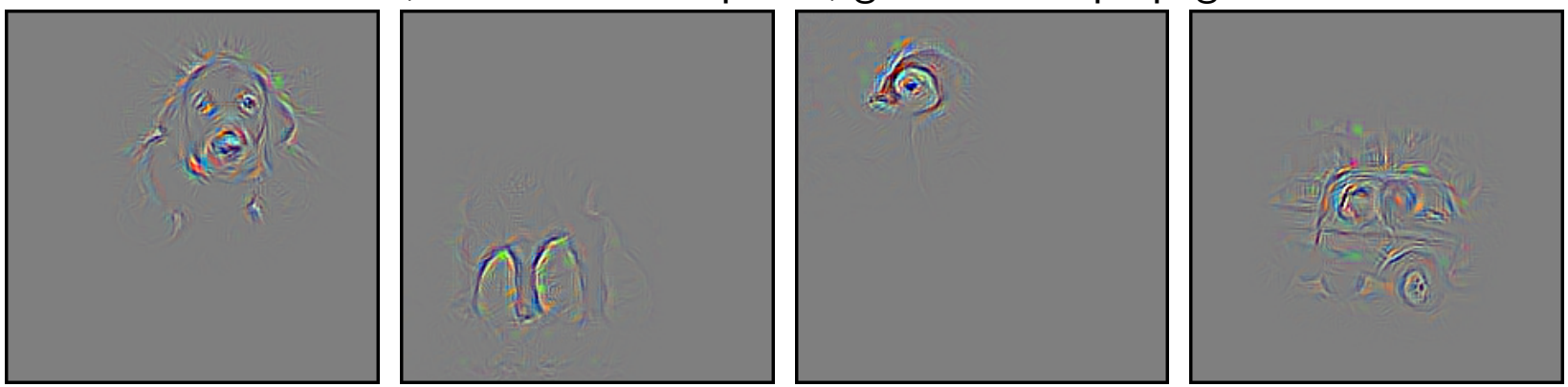
Original images

AlexNet, max feature response, deconvolution



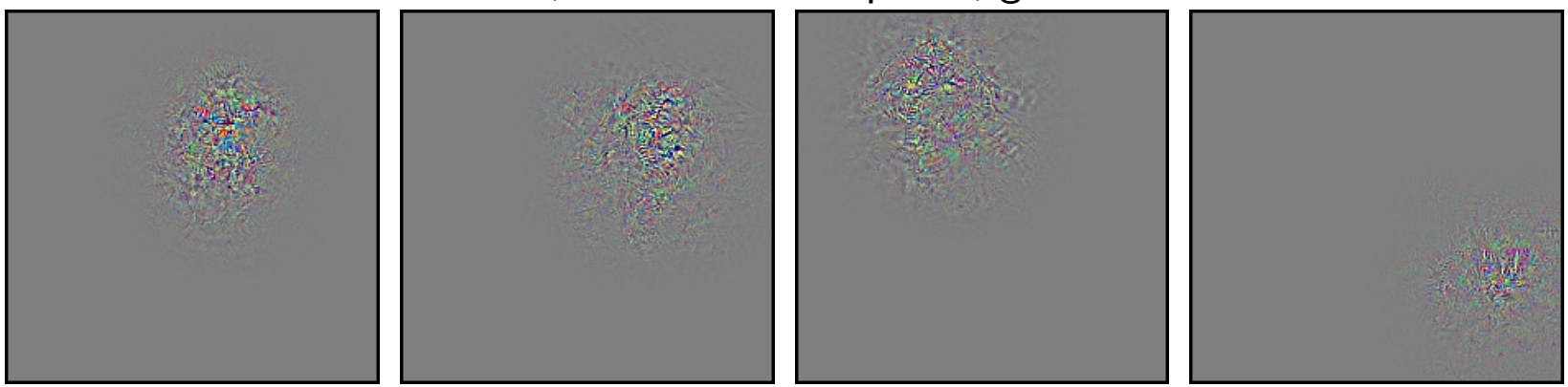
Original images

AlexNet, max feature response, guided back-propagation



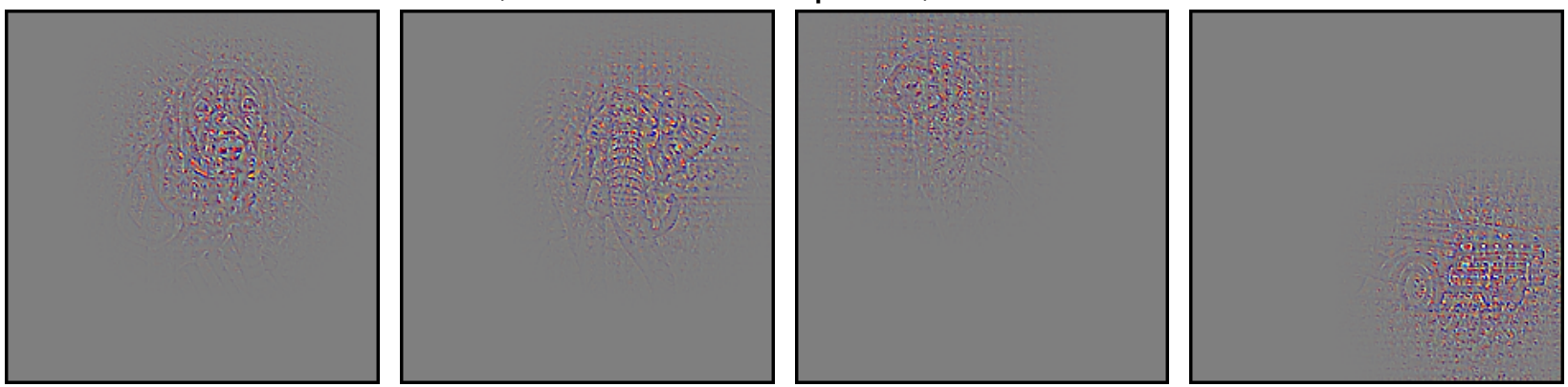
Original images

VGG16, max feature response, gradient



Original images

VGG16, max feature response, deconvolution

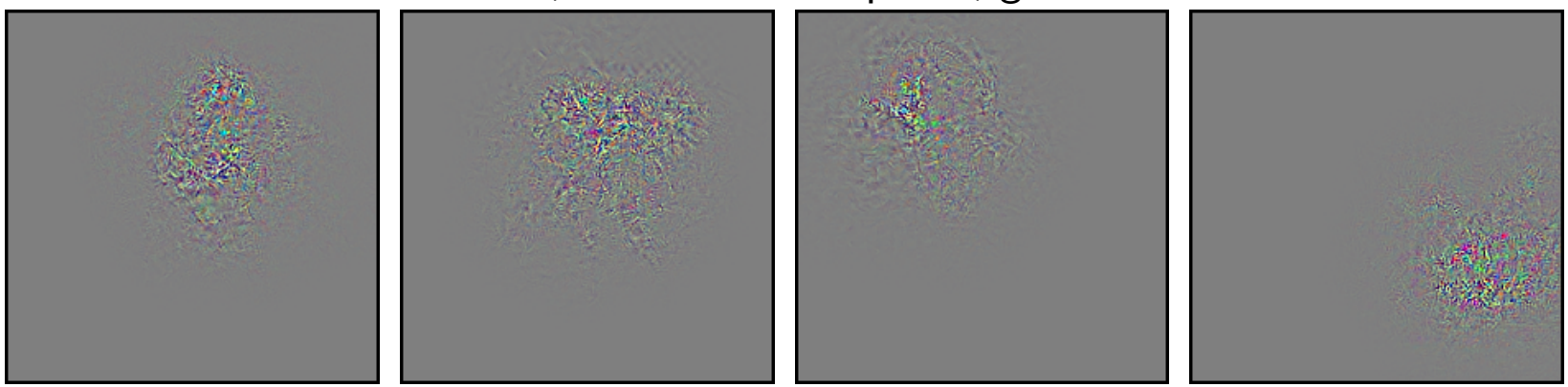


Original images

VGG16, max feature response, guided back-propagation

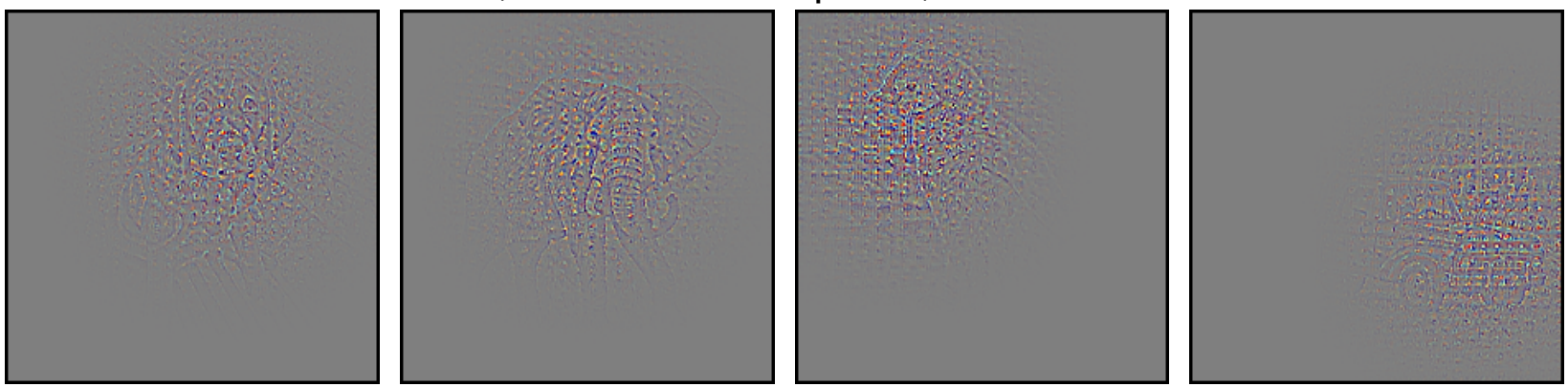
Original images

VGG19, max feature response, gradient



Original images

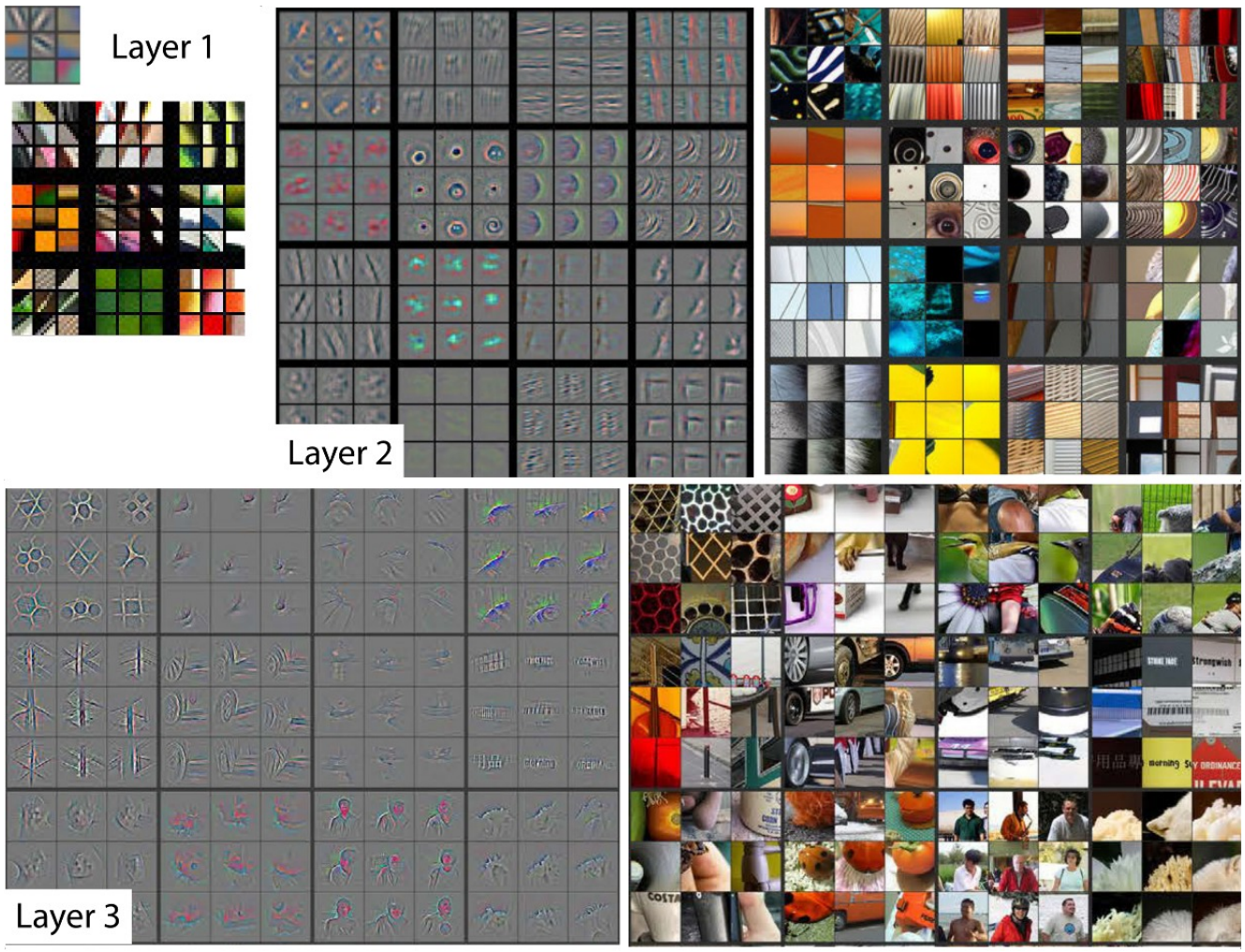
VGG19, max feature response, deconvolution



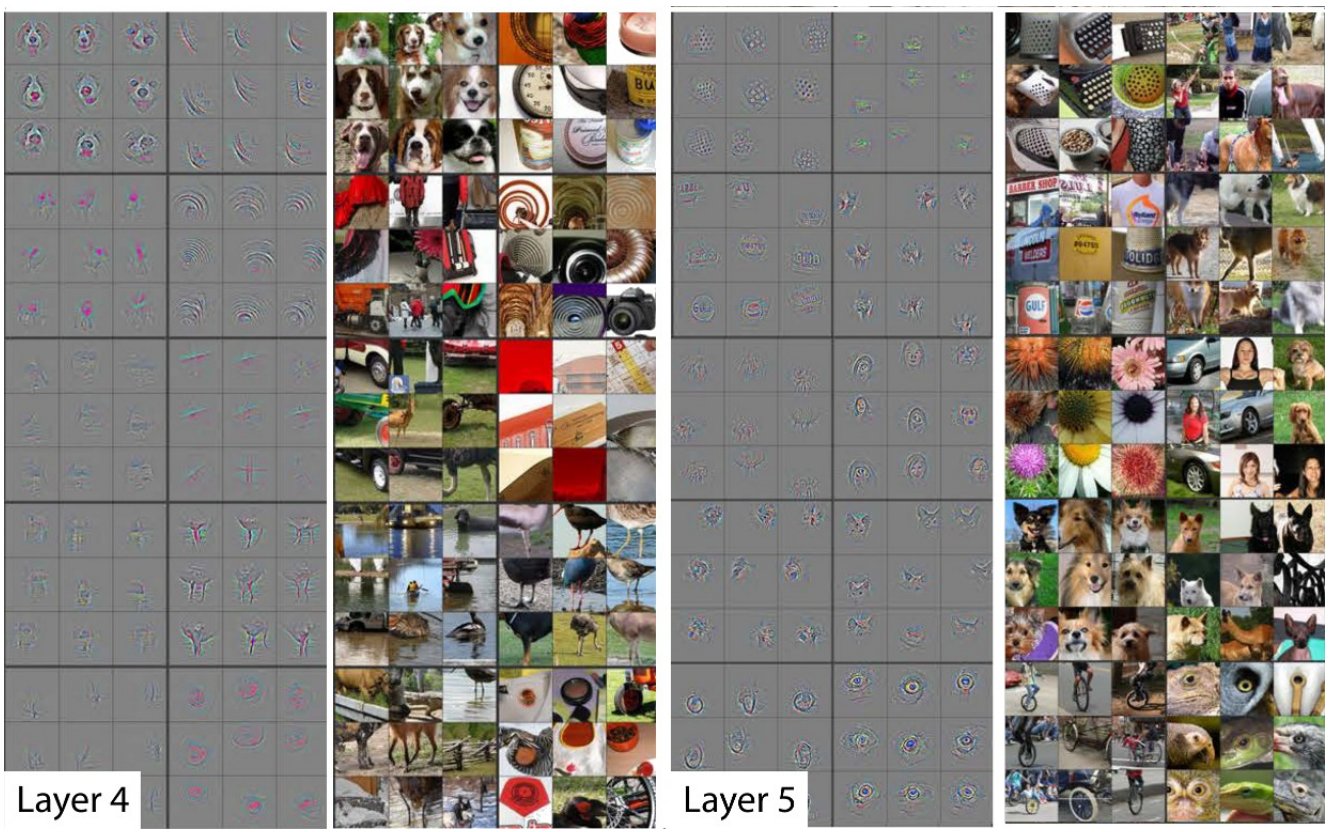
Original images

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)

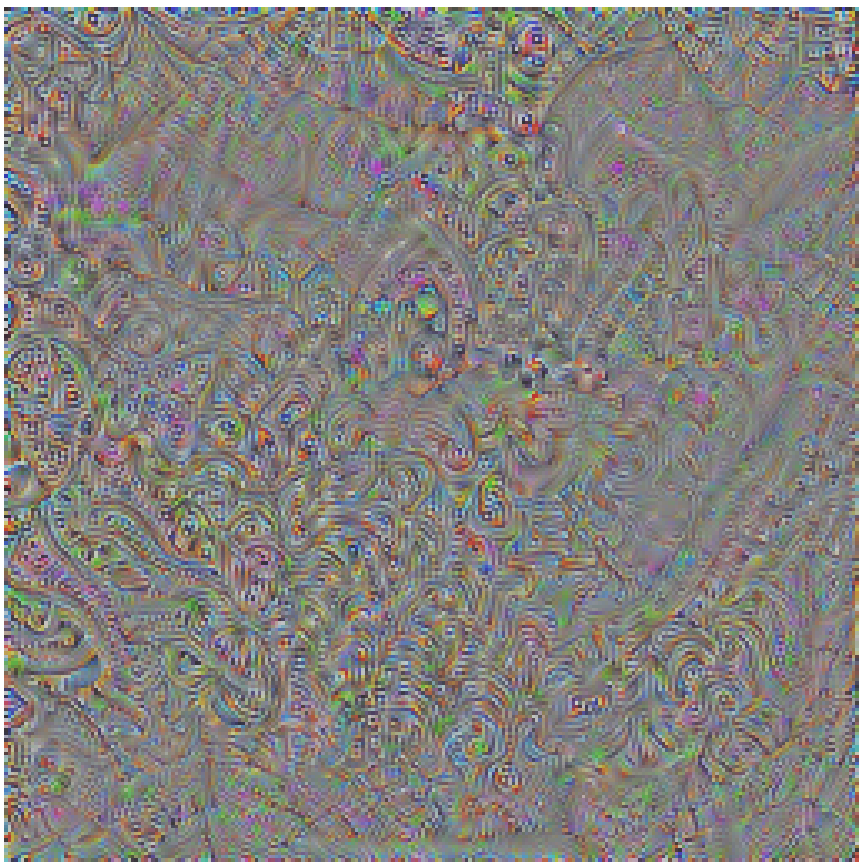


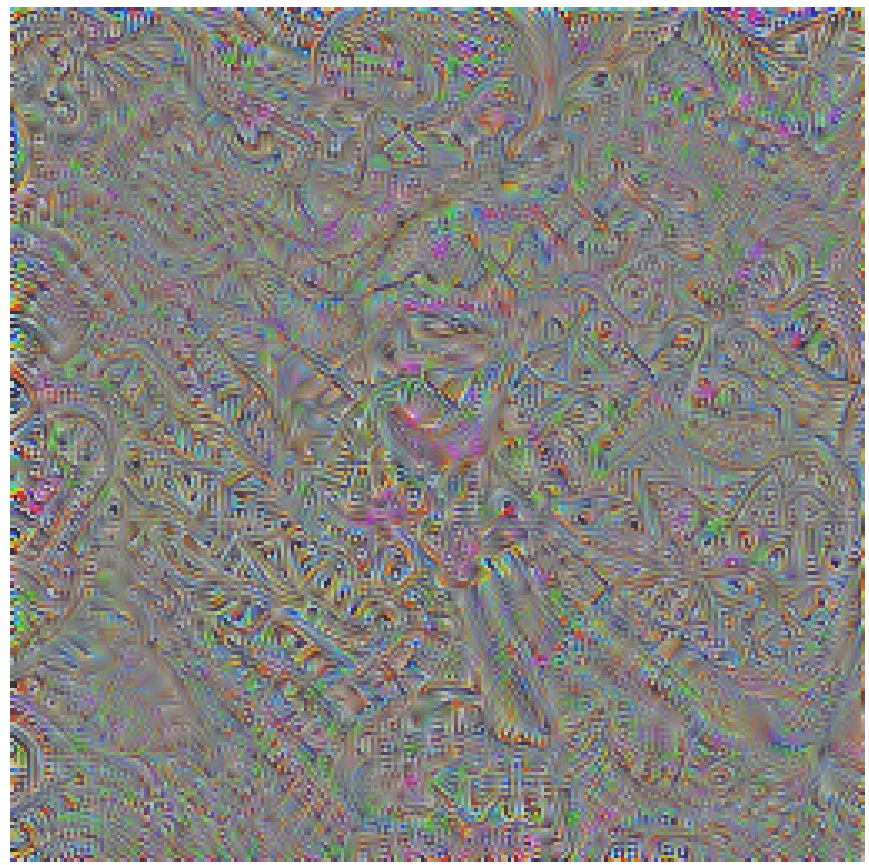
(Zeiler and Fergus, 2014)

Maximum response samples

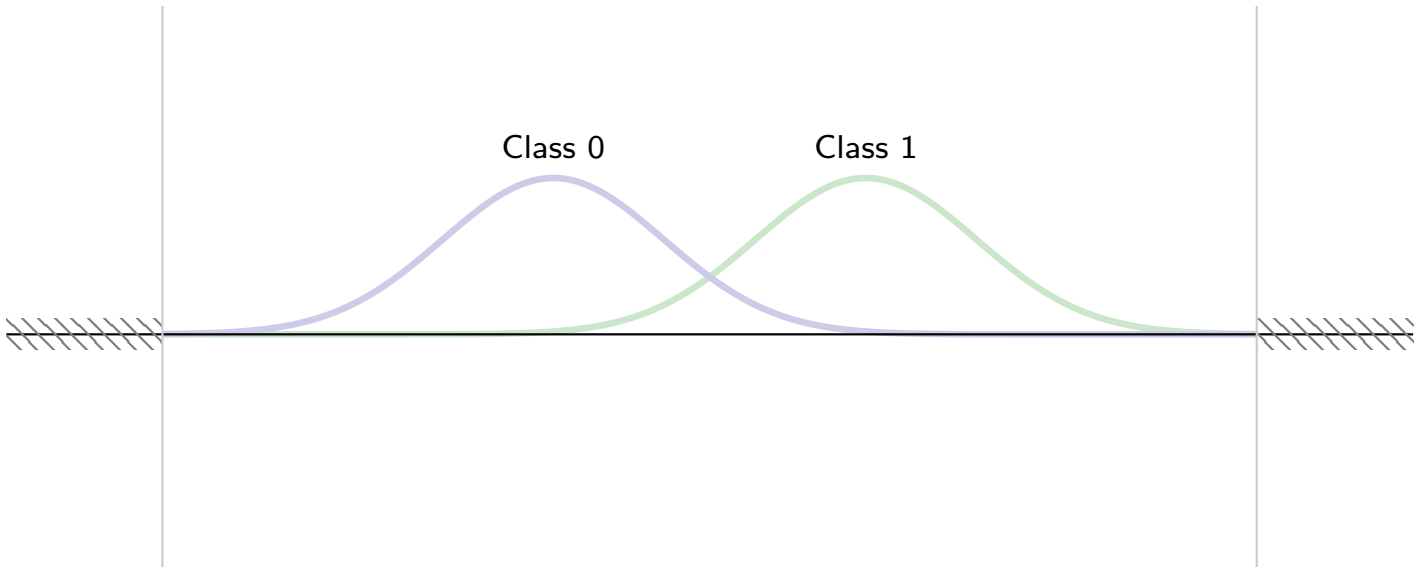
Another approach to get an intuition of the information actually encoded in the weights of a convnet consists of optimizing from scratch a sample to maximize the activation f of a chosen unit, or the sum over an activation map.

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).

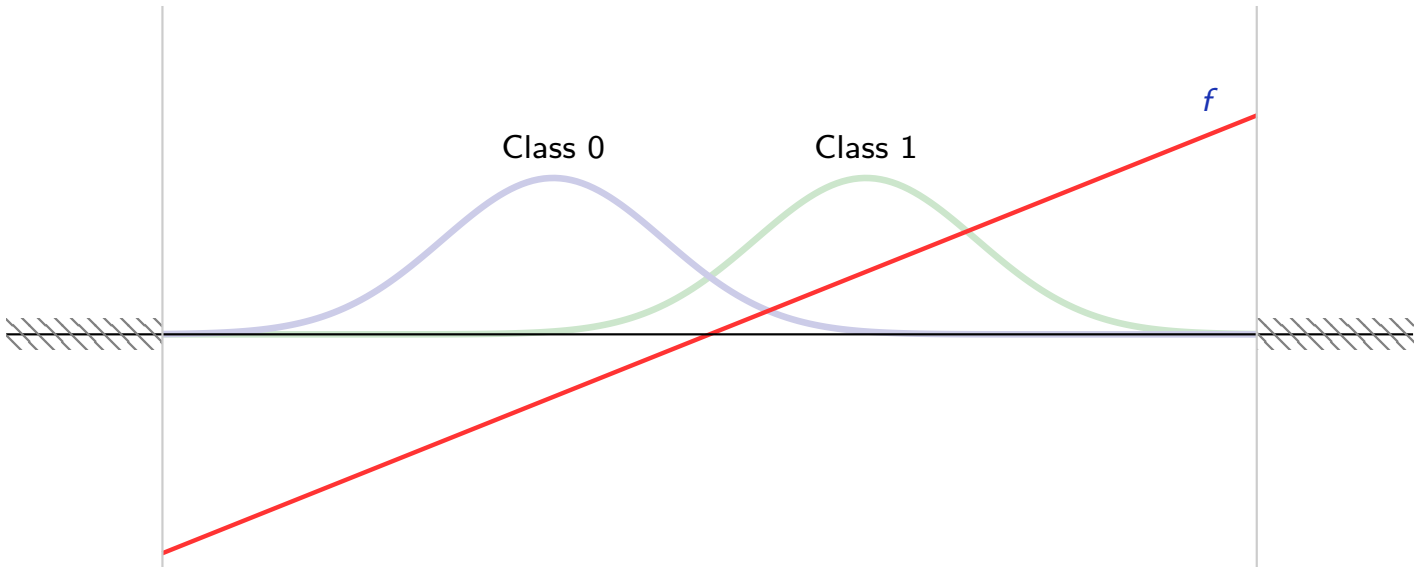




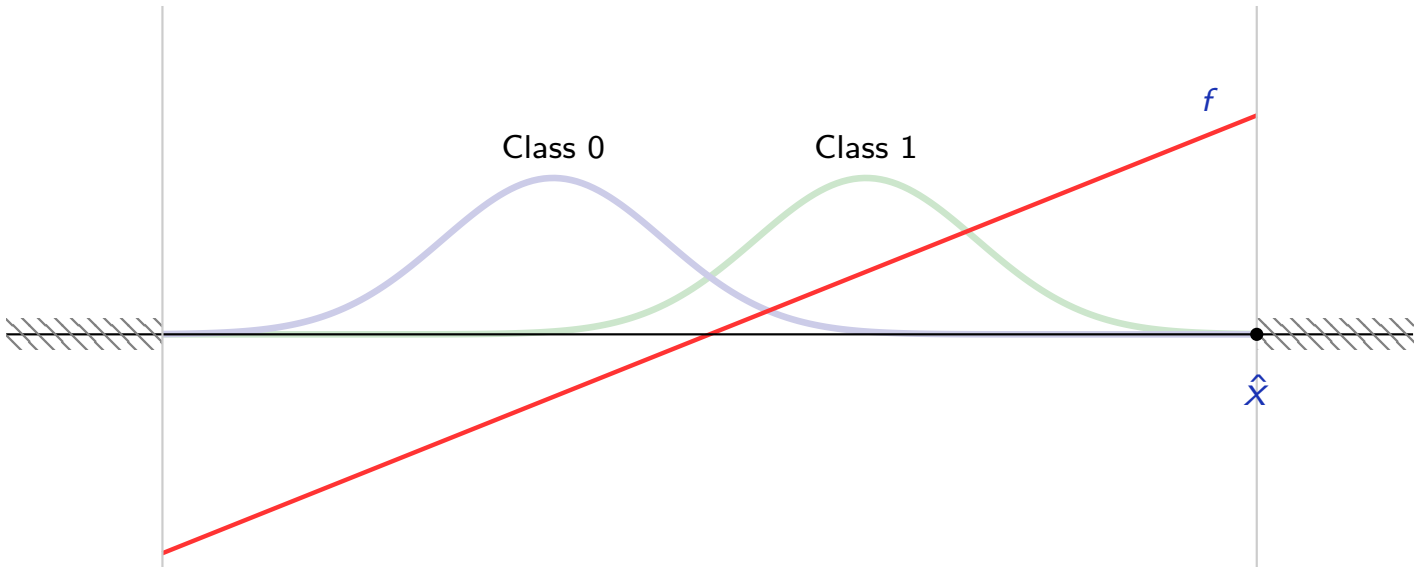
Since f is trained in a discriminative manner, there is no reason that a sample maximizing its response would be "realistic".



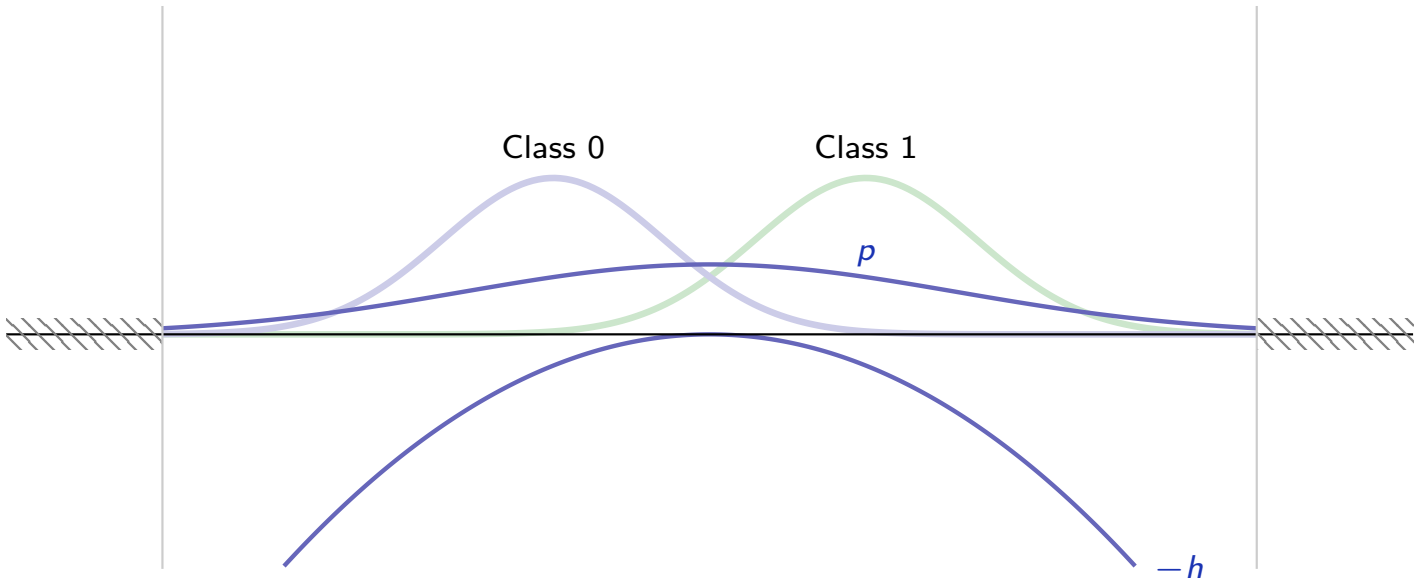
Since f is trained in a discriminative manner, there is no reason that a sample maximizing its response would be "realistic".



Since f is trained in a discriminative manner, there is no reason that a sample maximizing its response would be "realistic".



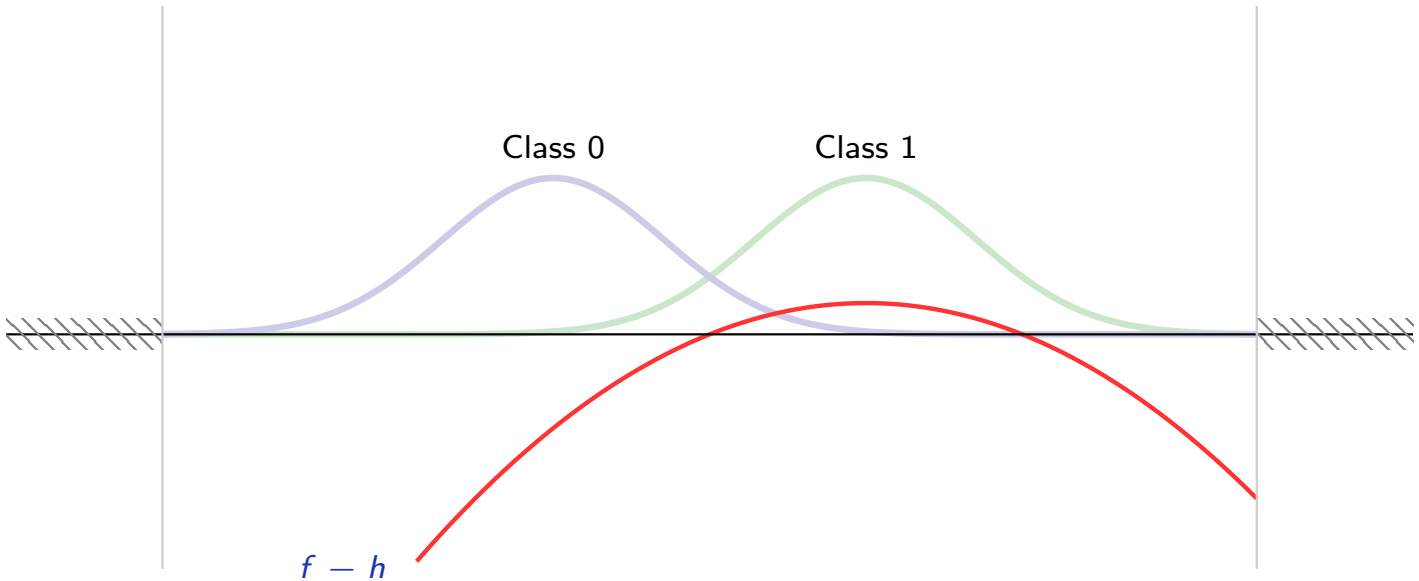
Since f is trained in a discriminative manner, there is no reason that a sample maximizing its response would be "realistic".



We can mitigate this by adding a penalty h corresponding to a "realistic" prior and compute in the end

$$\underset{x}{\operatorname{argmax}}\, f(x; w) - h(x)$$

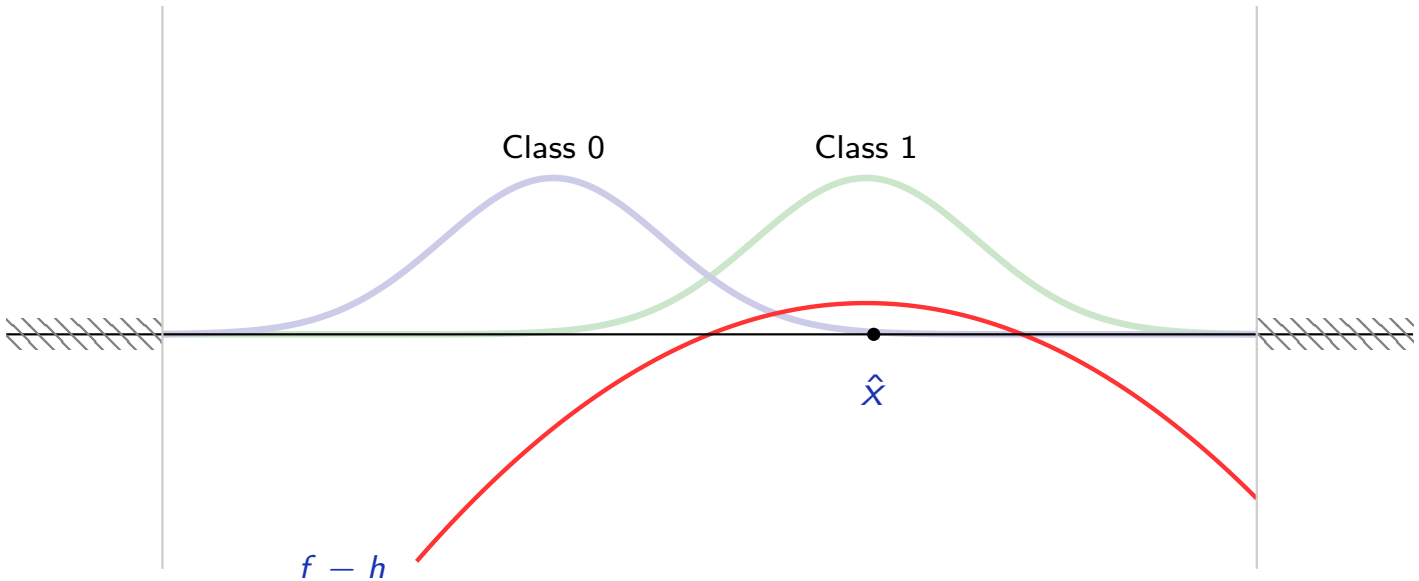
Since f is trained in a discriminative manner, there is no reason that a sample maximizing its response would be "realistic".



We can mitigate this by adding a penalty h corresponding to a "realistic" prior and compute in the end

$$\underset{x}{\operatorname{argmax}}\, f(x; w) - h(x)$$

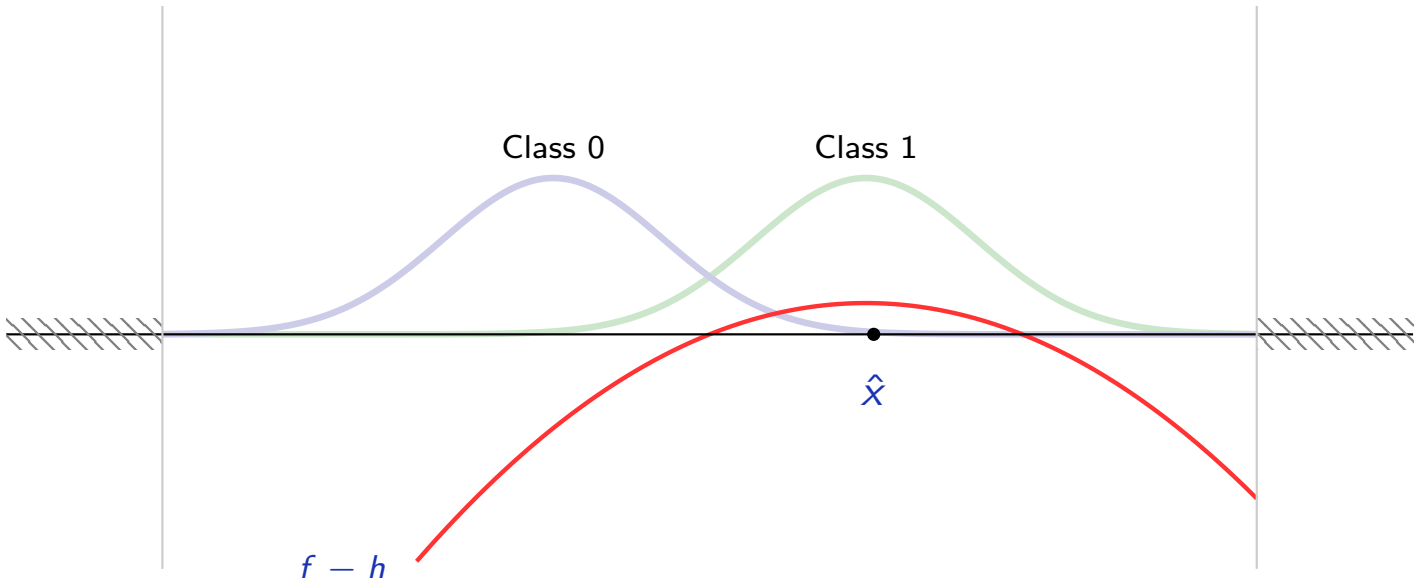
Since f is trained in a discriminative manner, there is no reason that a sample maximizing its response would be "realistic".



We can mitigate this by adding a penalty h corresponding to a "realistic" prior and compute in the end

$$\underset{x}{\operatorname{argmax}}\, f(x; w) - h(x)$$

Since f is trained in a discriminative manner, there is no reason that a sample maximizing its response would be "realistic".



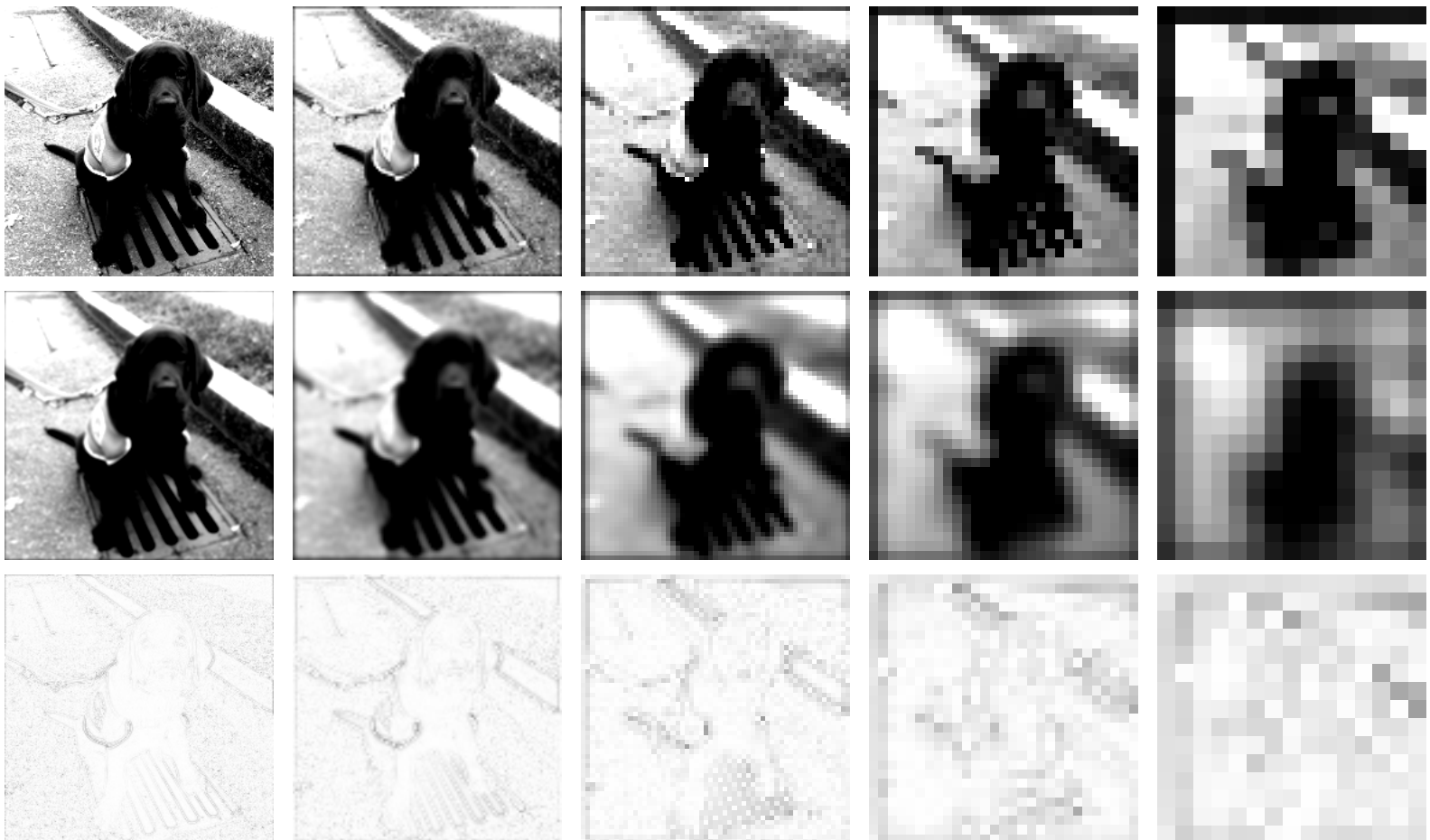
We can mitigate this by adding a penalty h corresponding to a "realistic" prior and compute in the end

$$\underset{x}{\operatorname{argmax}}\, f(x; w) - h(x)$$

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 factor-2 downscale operator.

We first implement

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

as a module:

We first implement

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

as a module:

```
class MultiScaleEdgeEnergy(nn.Module):
    def __init__(self):
        super(MultiScaleEdgeEnergy, self).__init__()
        k = Tensor([[1, 4, 6, 4, 1]])
        k_5x5_pseudo_gaussian = k.t().mm(k).view(1, 1, 5, 5)
        k_5x5_pseudo_gaussian /= k_5x5_pseudo_gaussian.sum()
        self.k_5x5_pseudo_gaussian = Parameter(k_5x5_pseudo_gaussian)
        k_2x2_uniform = Tensor([[0.25, 0.25], [0.25, 0.25]]).view(1, 1, 2, 2)
        self.k_2x2_uniform = Parameter(k_2x2_uniform)

    def forward(self, x):
        if x.size(1) > 1:
            # dealing with multiple channels by unfolding them as as
            # many one channel images
            result = self(x.view(x.size(0) * x.size(1), 1, x.size(2), x.size(3)))
            result = result.view(x.size(0), -1).sum(1)
        else:
            result = 0.0
            while x.size(2) > 5 and x.size(3) > 5:
                blurry  = F.conv2d(x, self.k_5x5_pseudo_gaussian, padding = 2)
                result += (x - blurry).view(x.size(0), -1).pow(2).sum(1)
                x = F.conv2d(x, self.k_2x2_uniform, stride = 2, padding = 1)

        return result
```

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

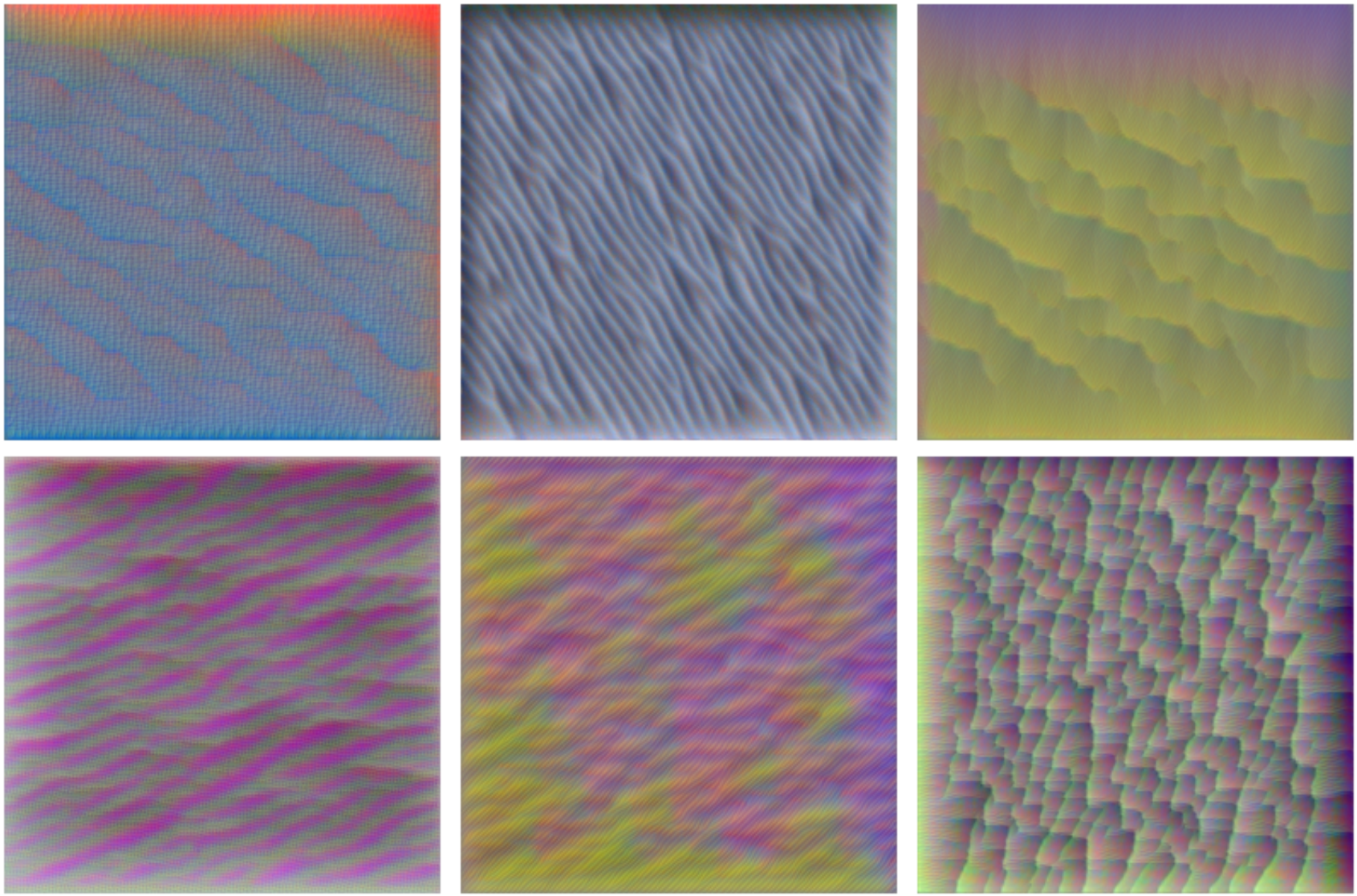
```
model = models.vgg16(pretrained = True)
model.eval()
edge_energy = MultiScaleEdgeEnergy()
input = Tensor(1, 3, 224, 224).normal_(0, 0.01)

if torch.cuda.is_available():
    model.cuda()
    edge_energy.cuda()
    input = input.cuda()

input = Variable(input, requires_grad = True)
optimizer = optim.Adam([input], lr = 1e-1)

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

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

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

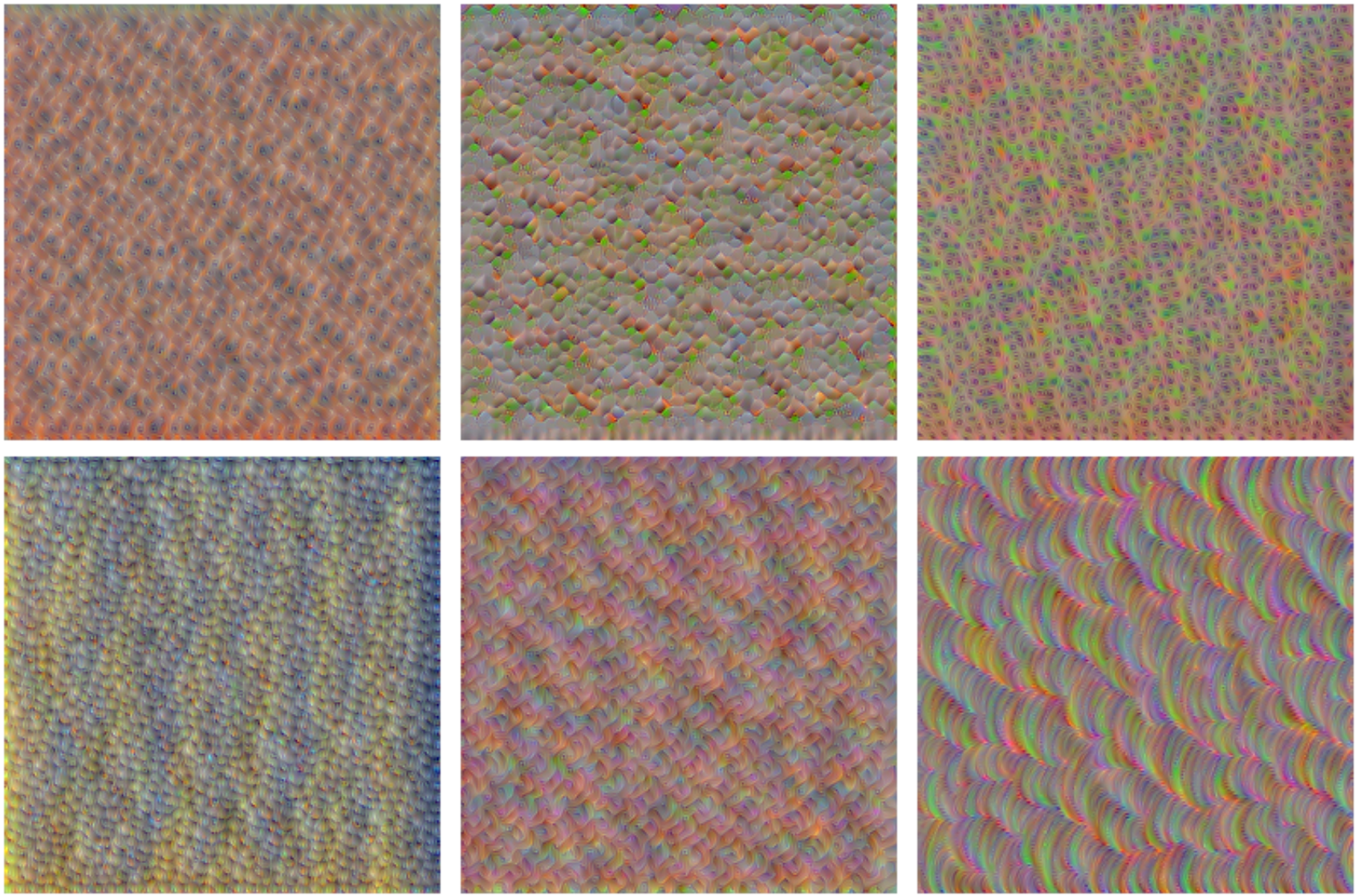
```
model = models.vgg16(pretrained = True)
model.eval()
edge_energy = MultiScaleEdgeEnergy()
input = Tensor(1, 3, 224, 224).normal_(0, 0.01)

if torch.cuda.is_available():
    model.cuda()
    edge_energy.cuda()
    input = input.cuda()

input = Variable(input, requires_grad = True)
optimizer = optim.Adam([input], lr = 1e-1)

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

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

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

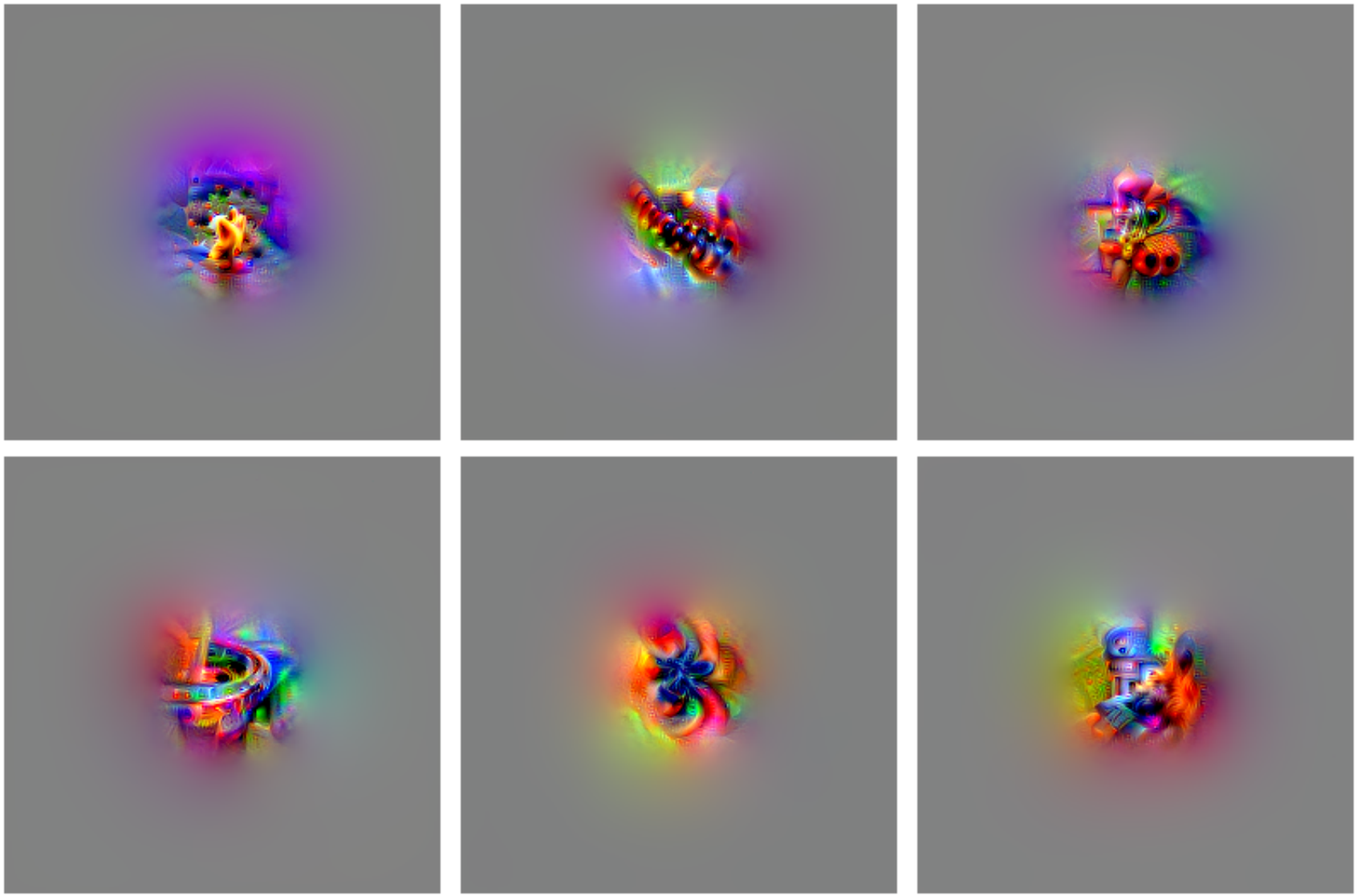
VGG16, maximizing a channel of the 4th convolution layer



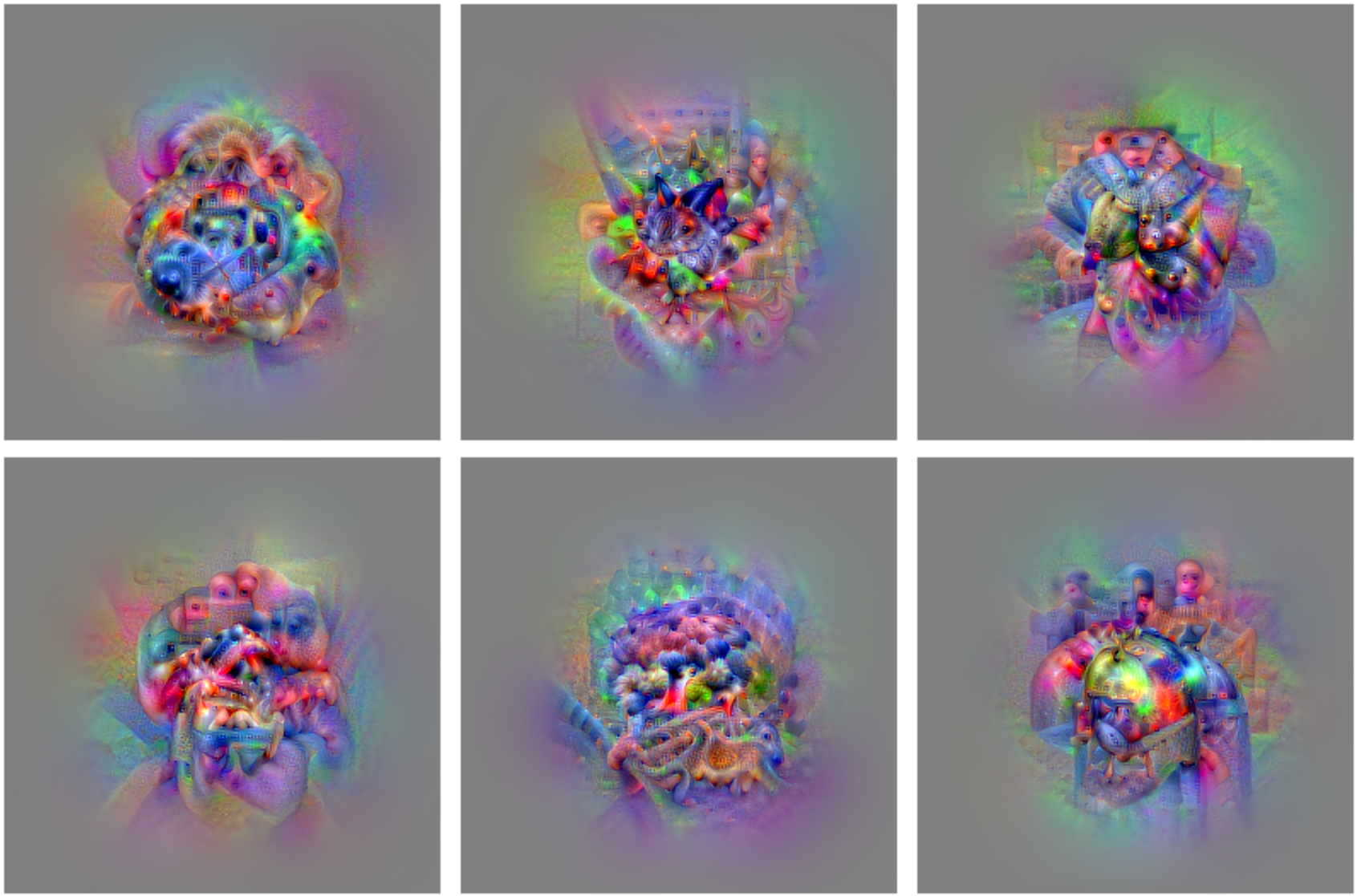
VGG16, maximizing a channel of the 7th convolution layer



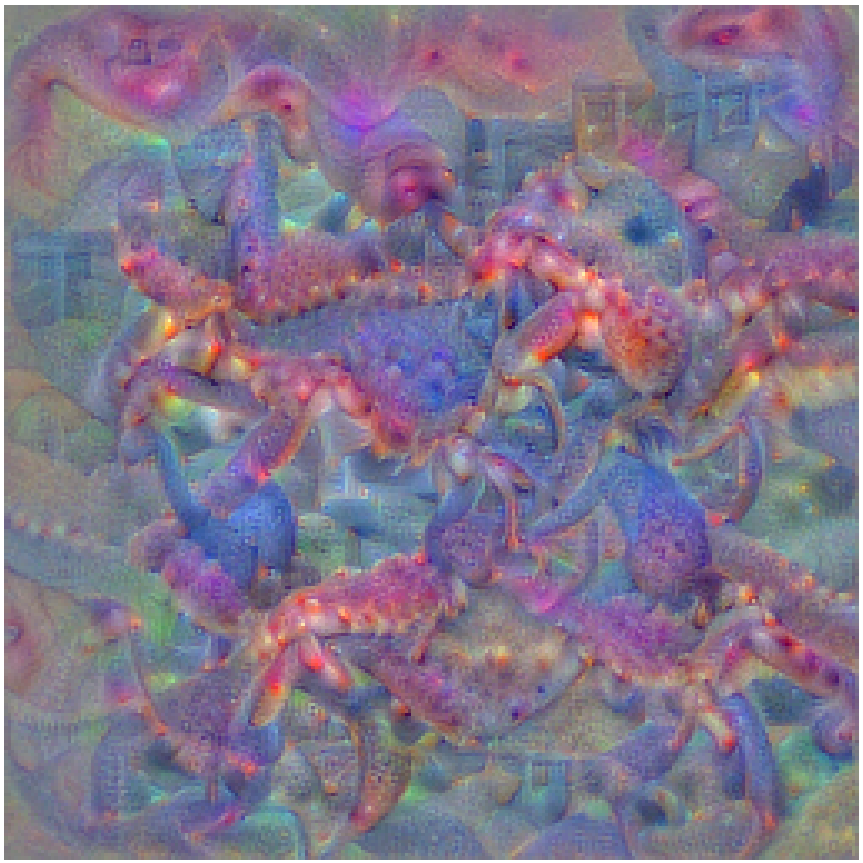
VGG16, maximizing a unit of the 10th convolution layer

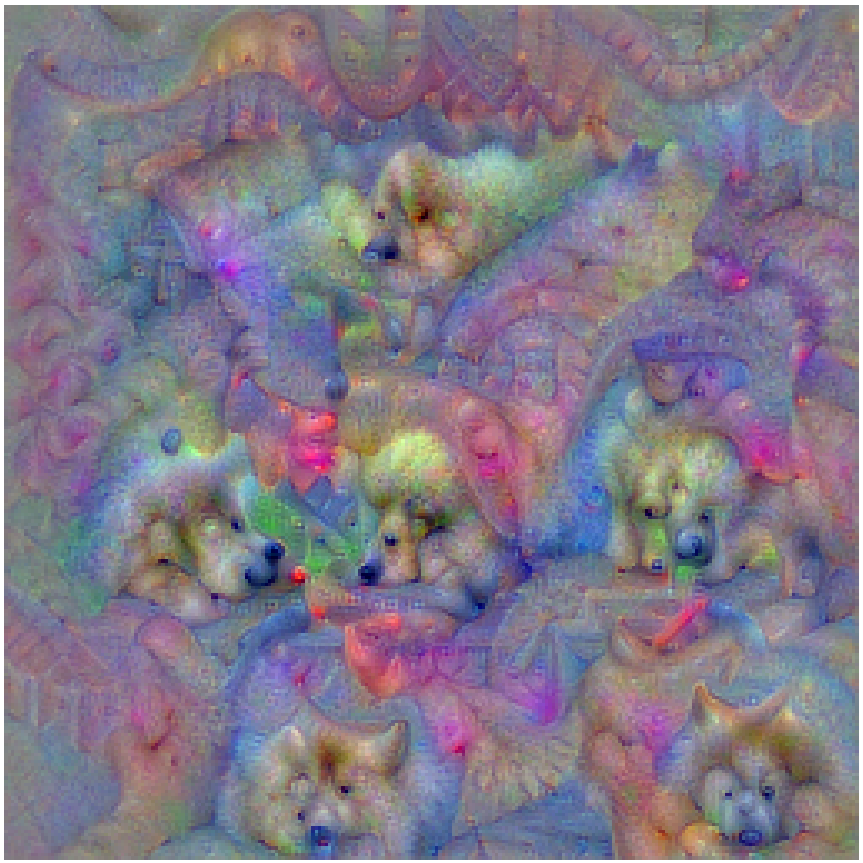


VGG16, maximizing a unit of the 13th (and last) convolution layer

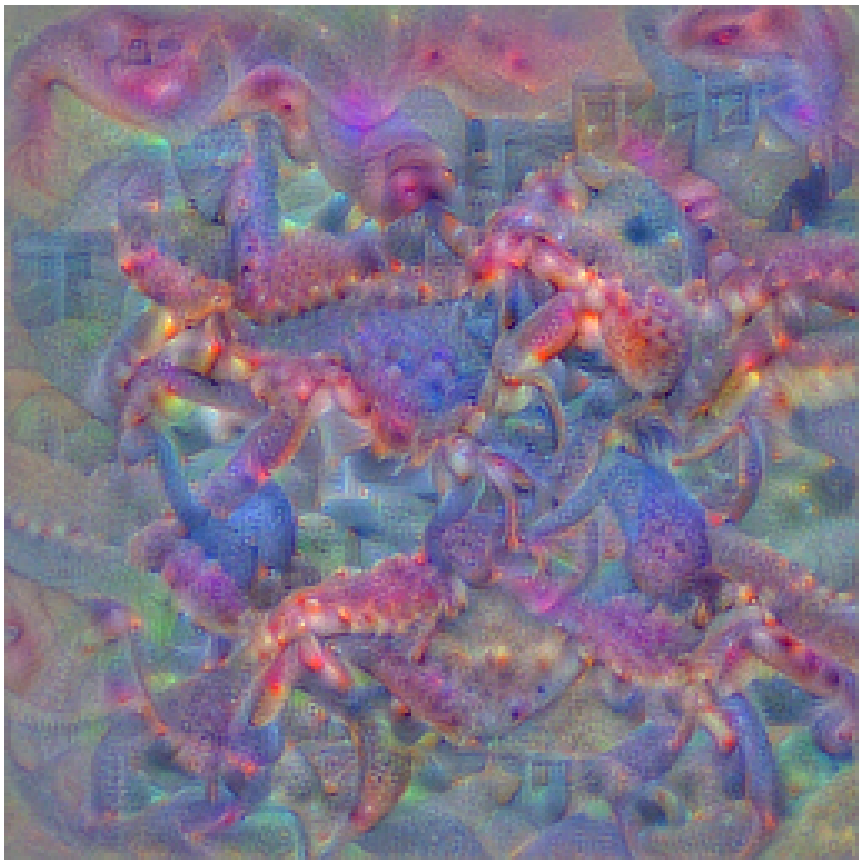


VGG16, maximizing a unit of the output layer

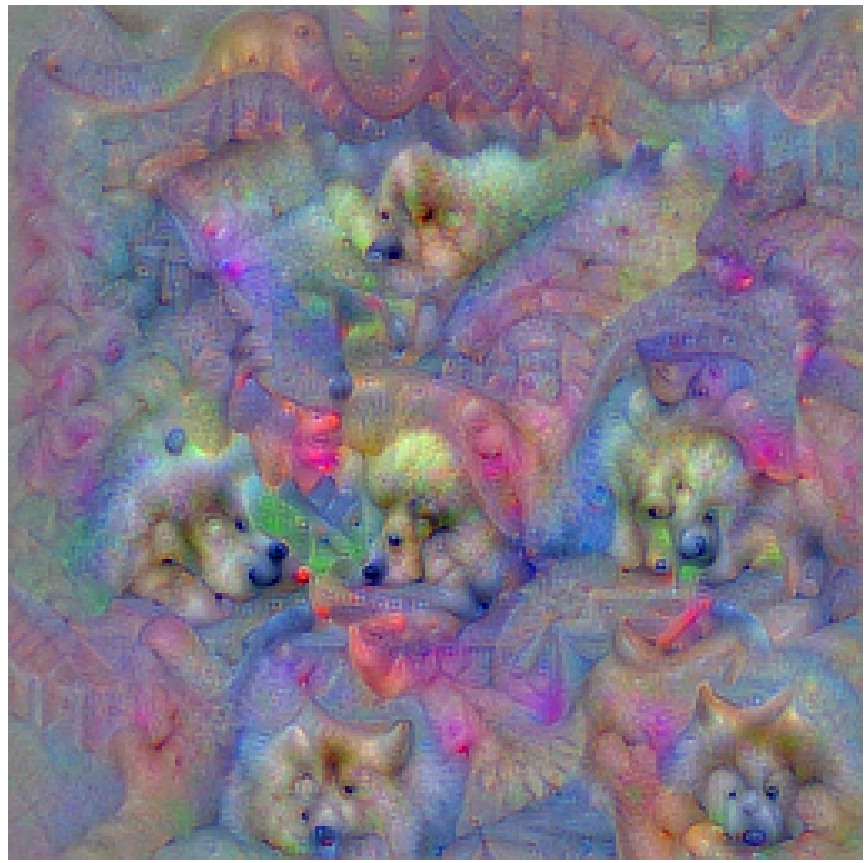




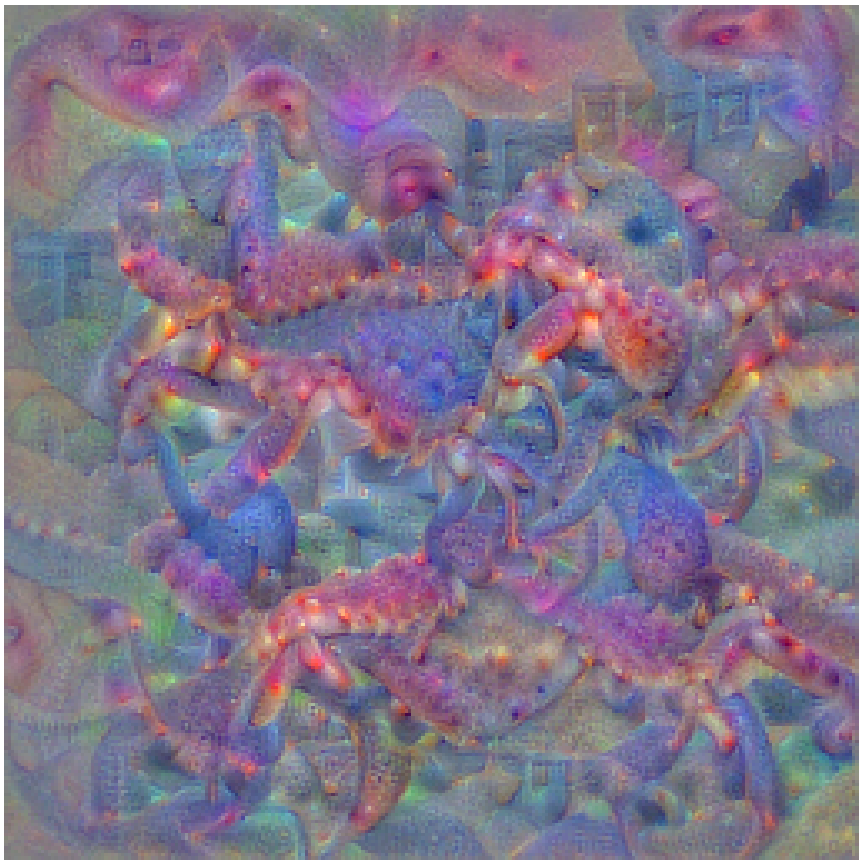
VGG16, maximizing a unit of the output layer



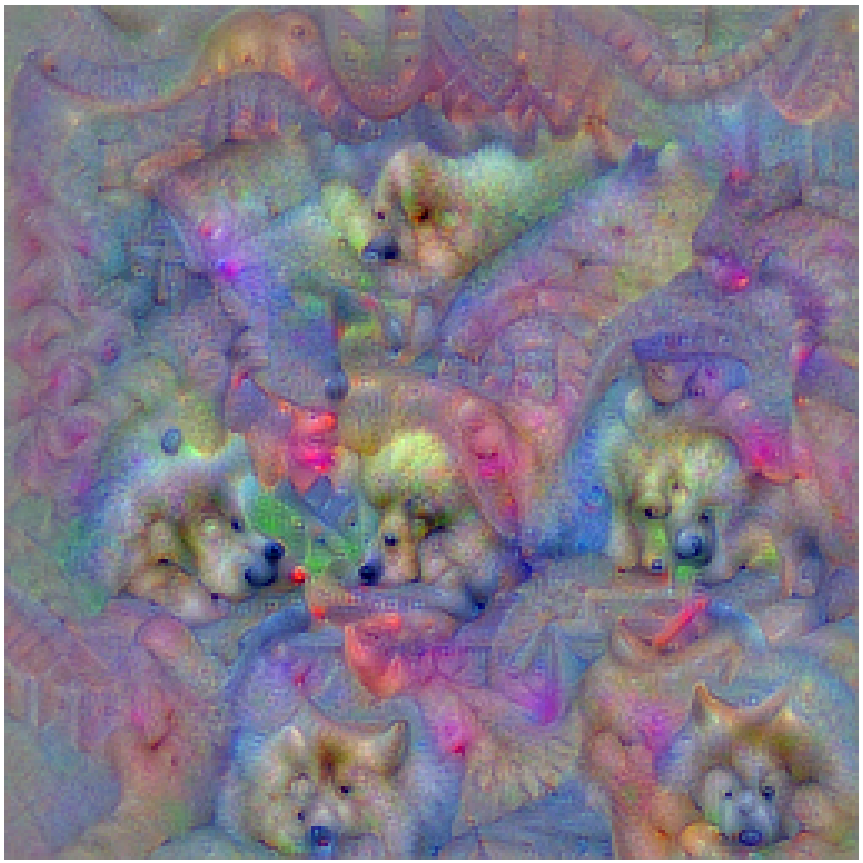
"King crab"



VGG16, maximizing a unit of the output layer

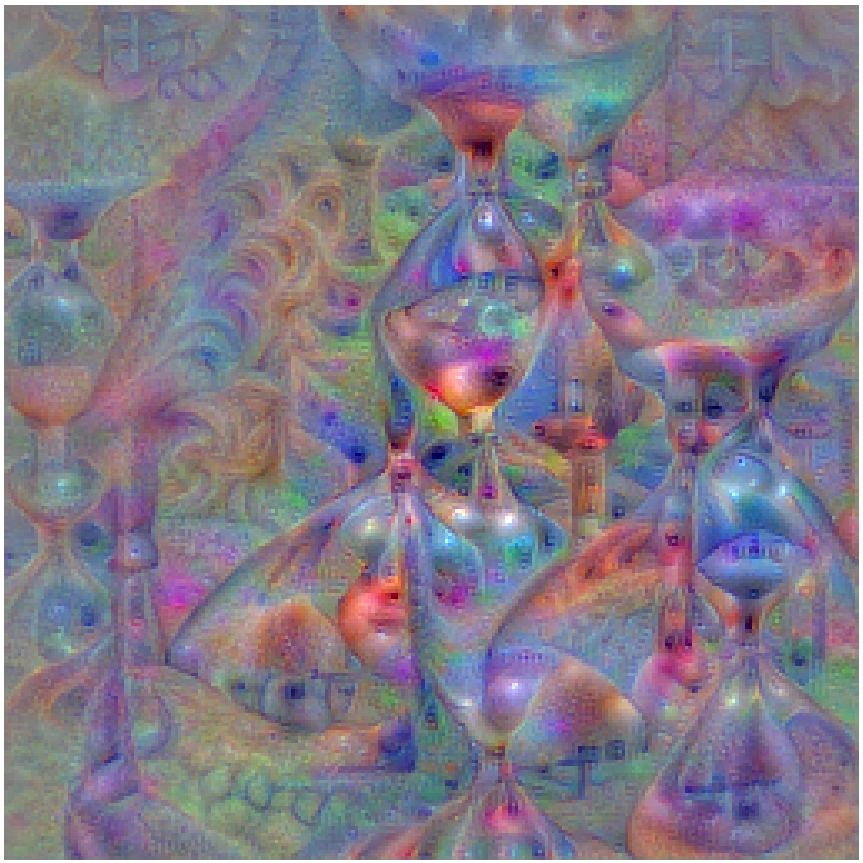


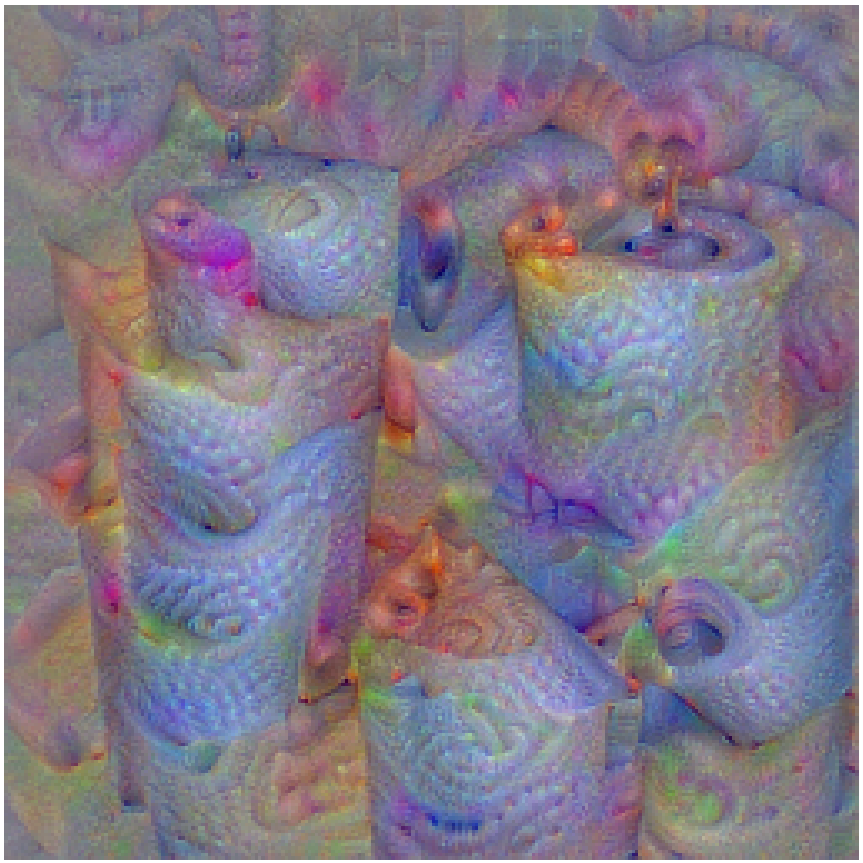
"King crab"



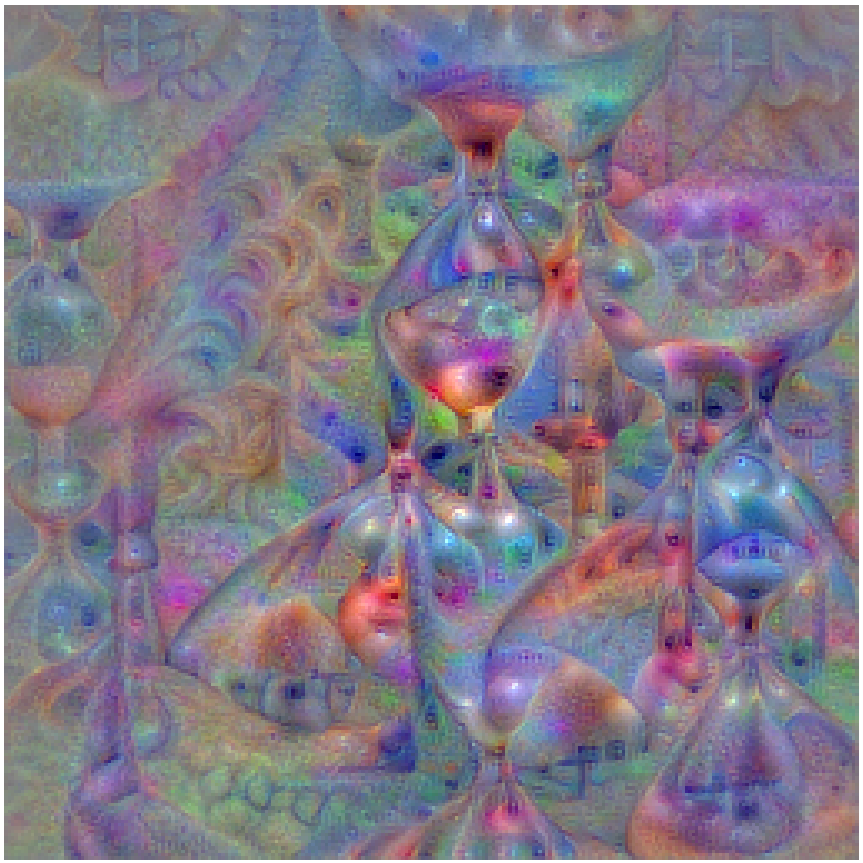
"Samoyed" (that's a fluffy dog)

VGG16, maximizing a unit of the output layer

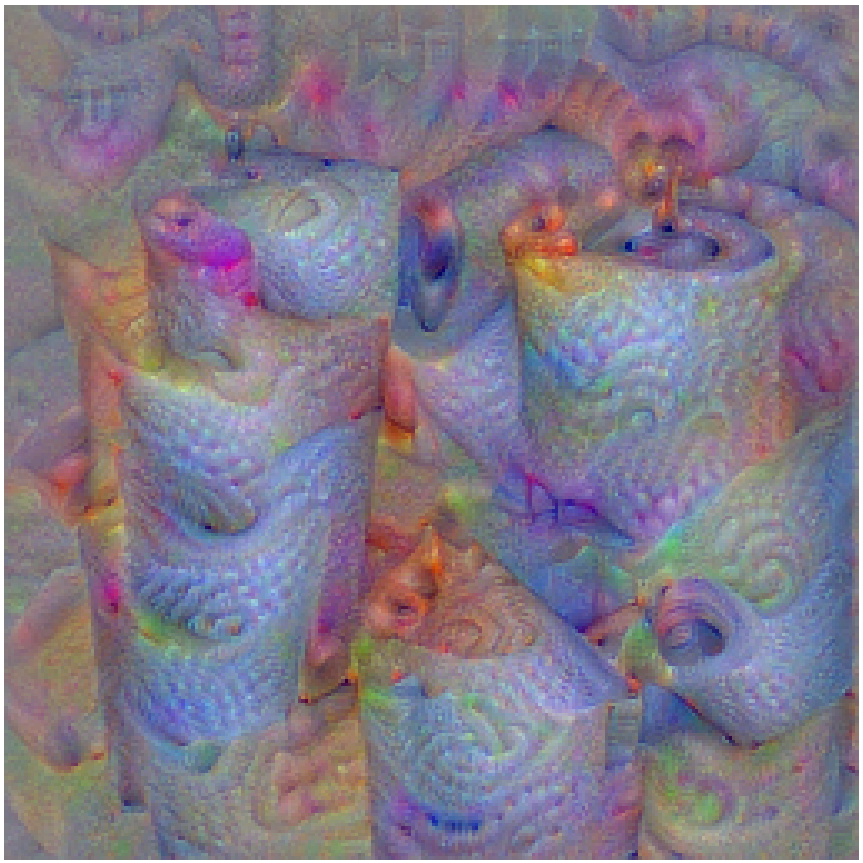




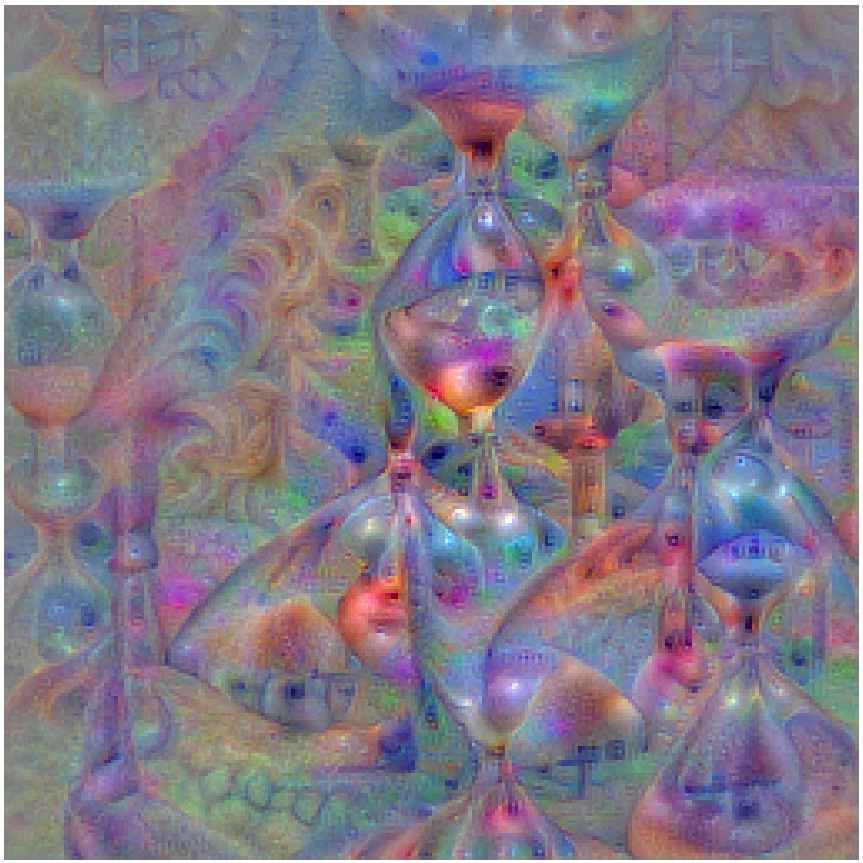
VGG16, maximizing a unit of the output layer



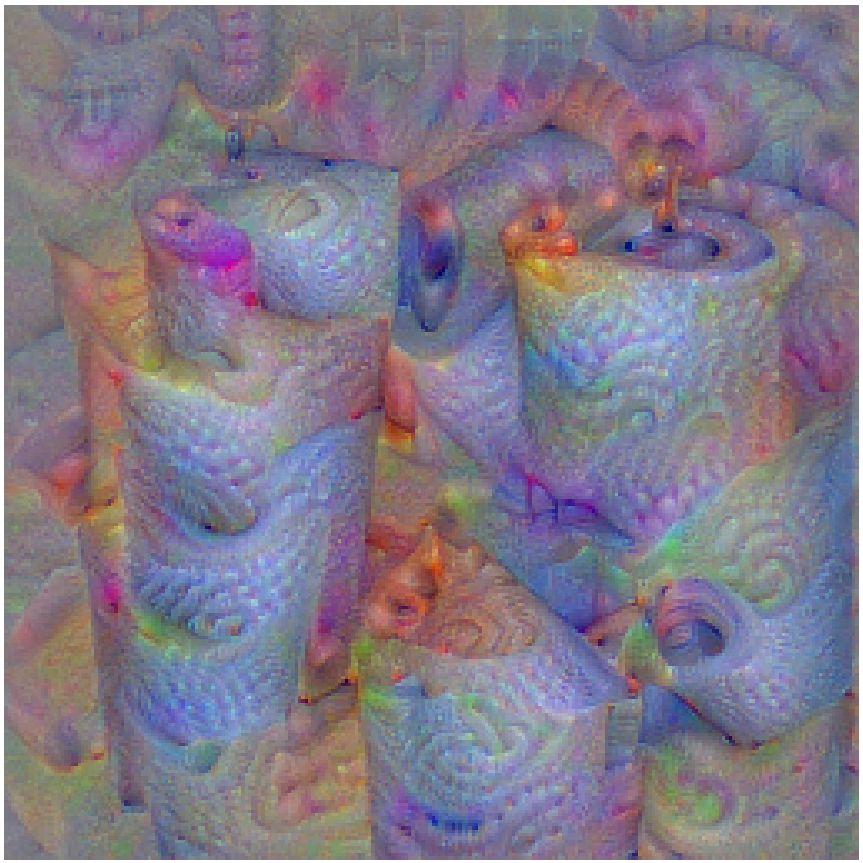
"Hourglass"



VGG16, maximizing a unit of the output layer

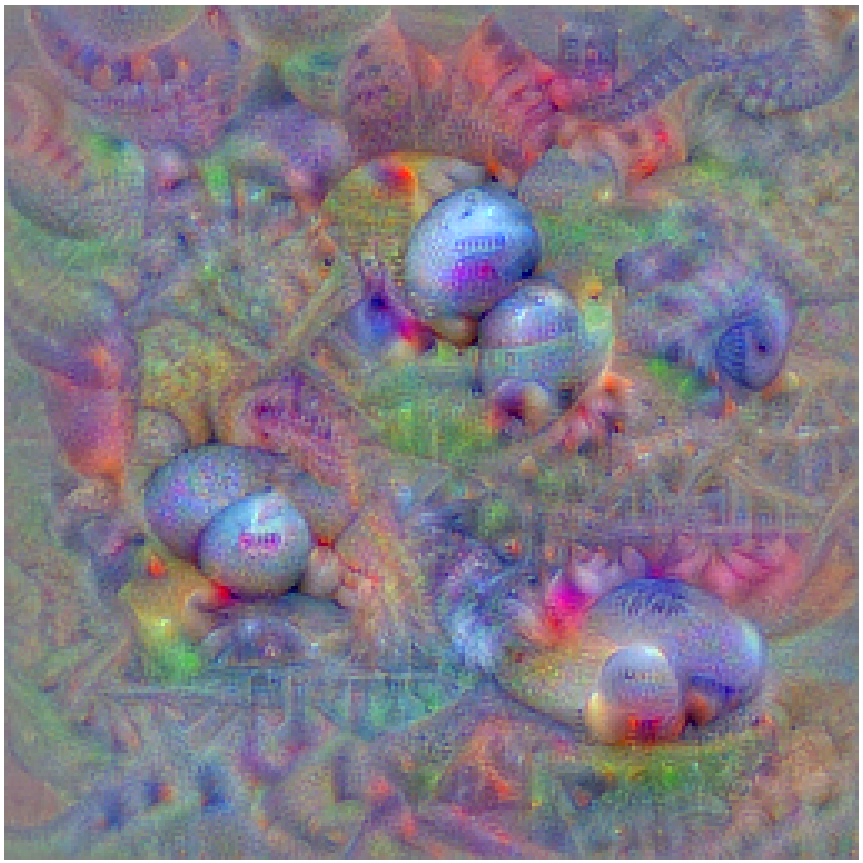


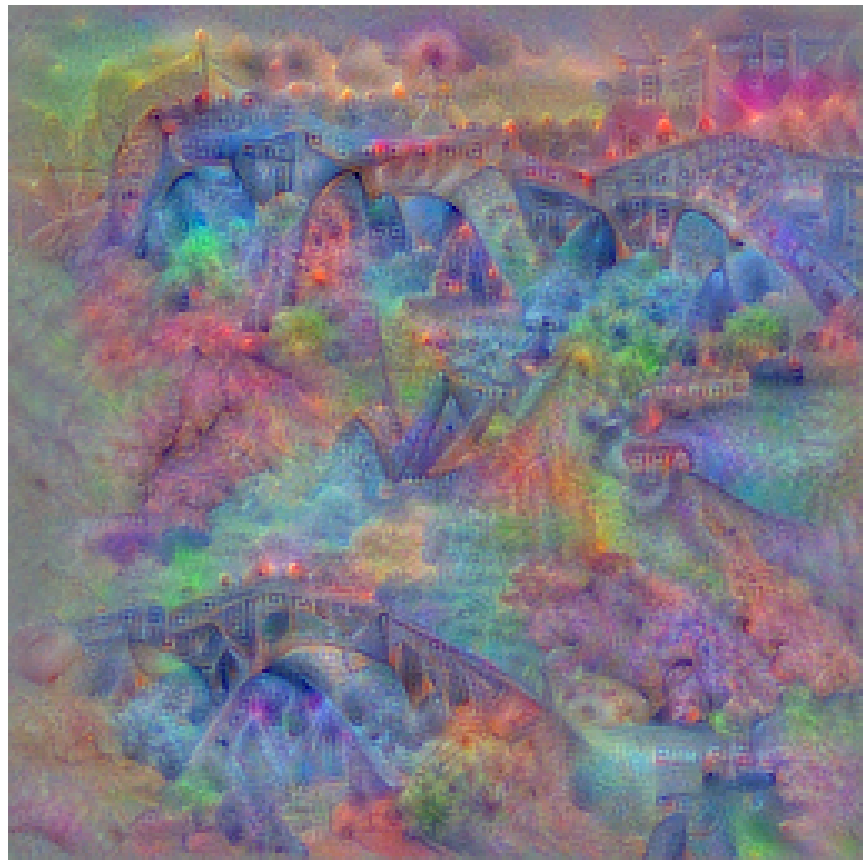
"Hourglass"



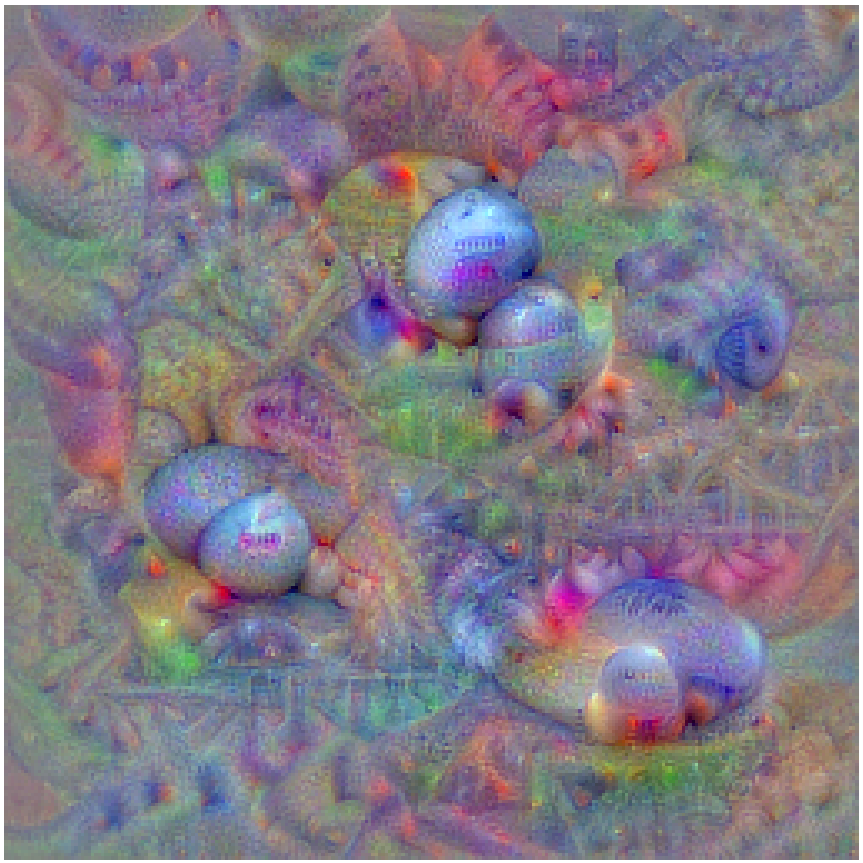
"Paper towel"

VGG16, maximizing a unit of the output layer

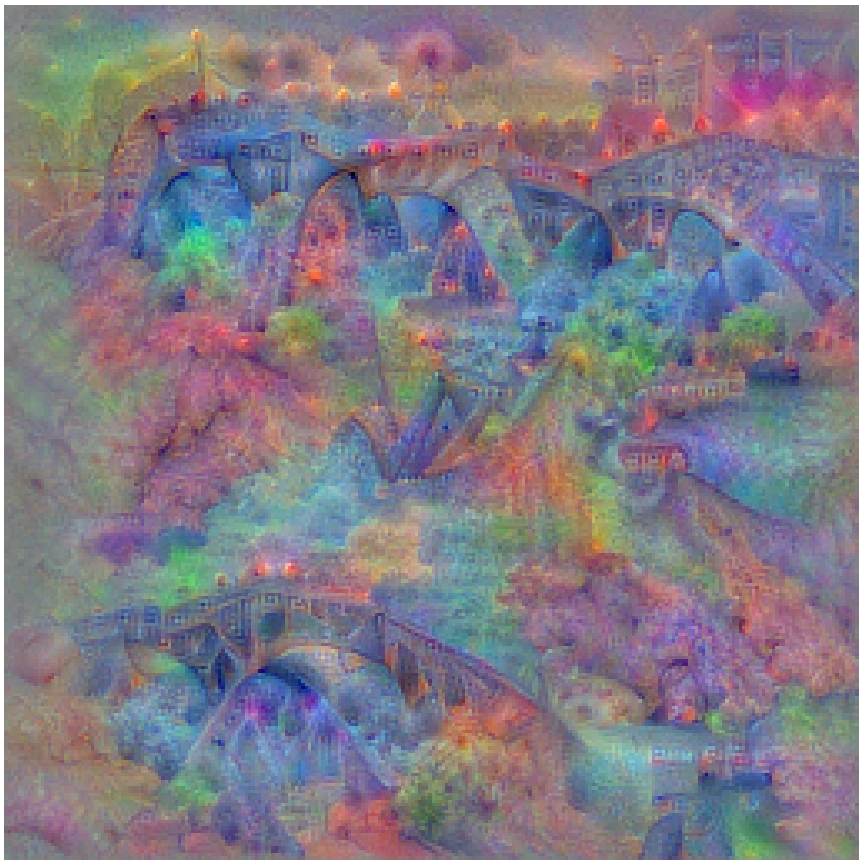




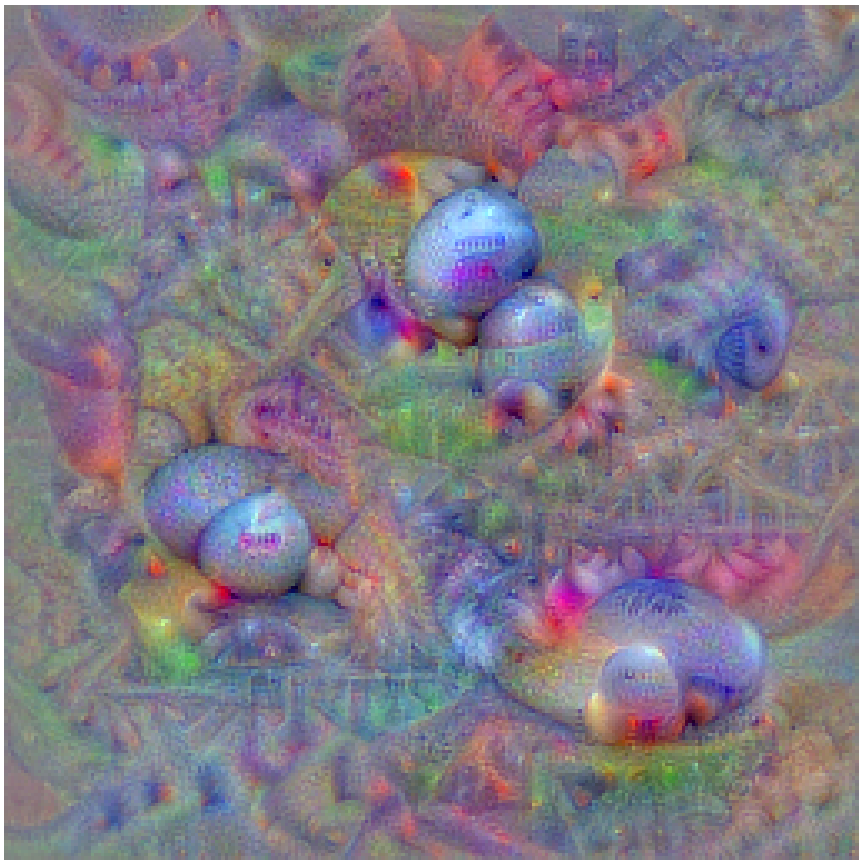
VGG16, maximizing a unit of the output layer



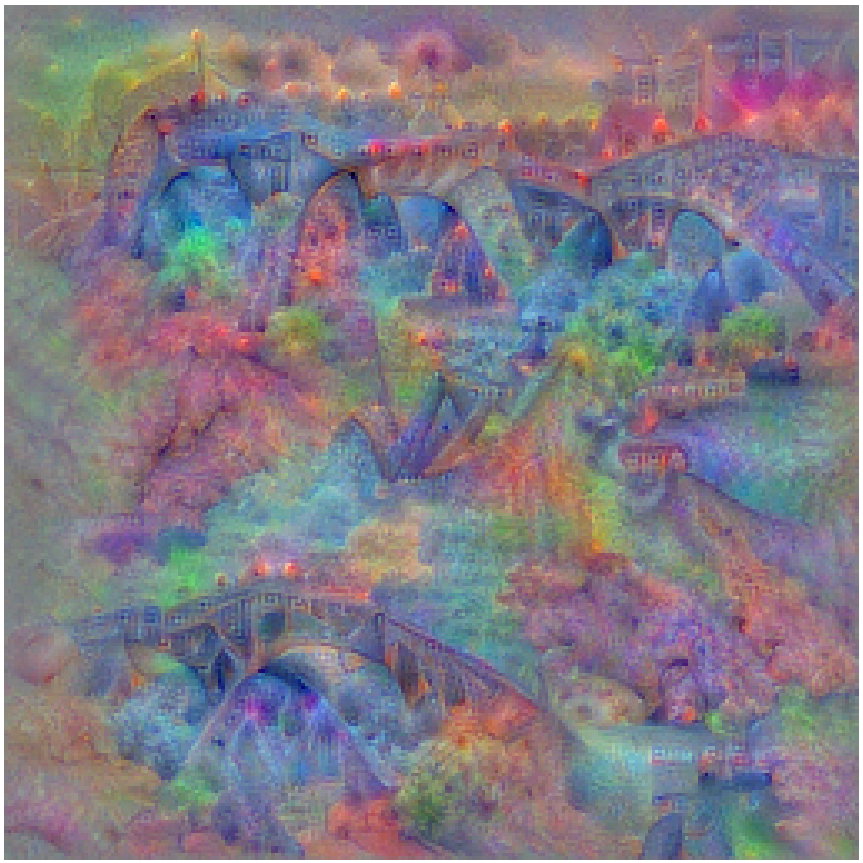
"Ping-pong ball"



VGG16, maximizing a unit of the output layer

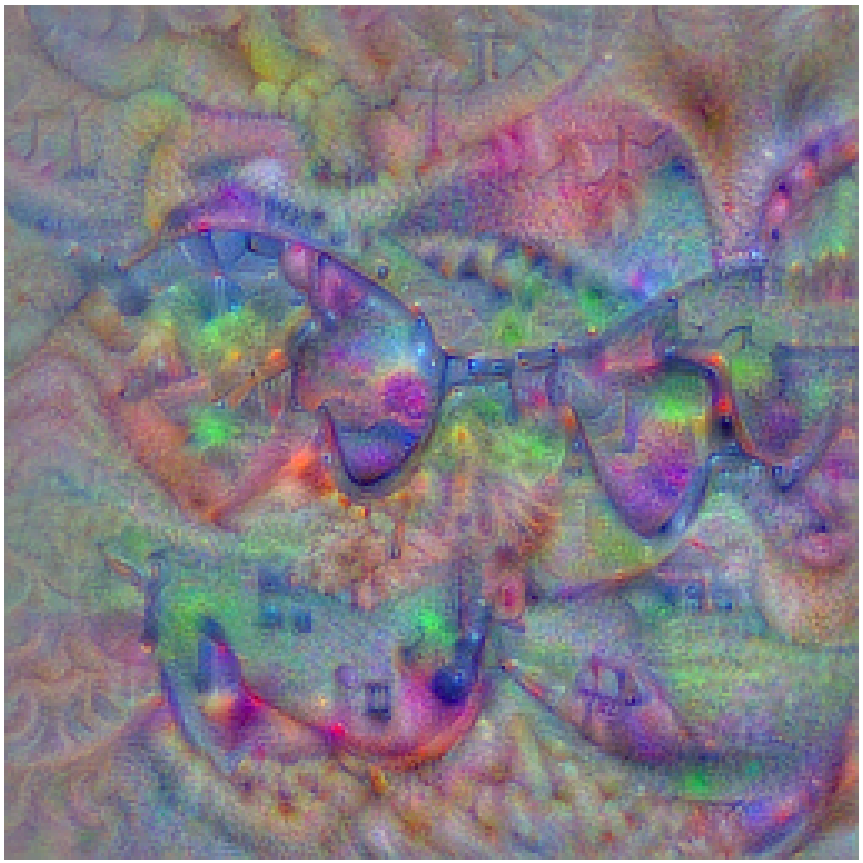


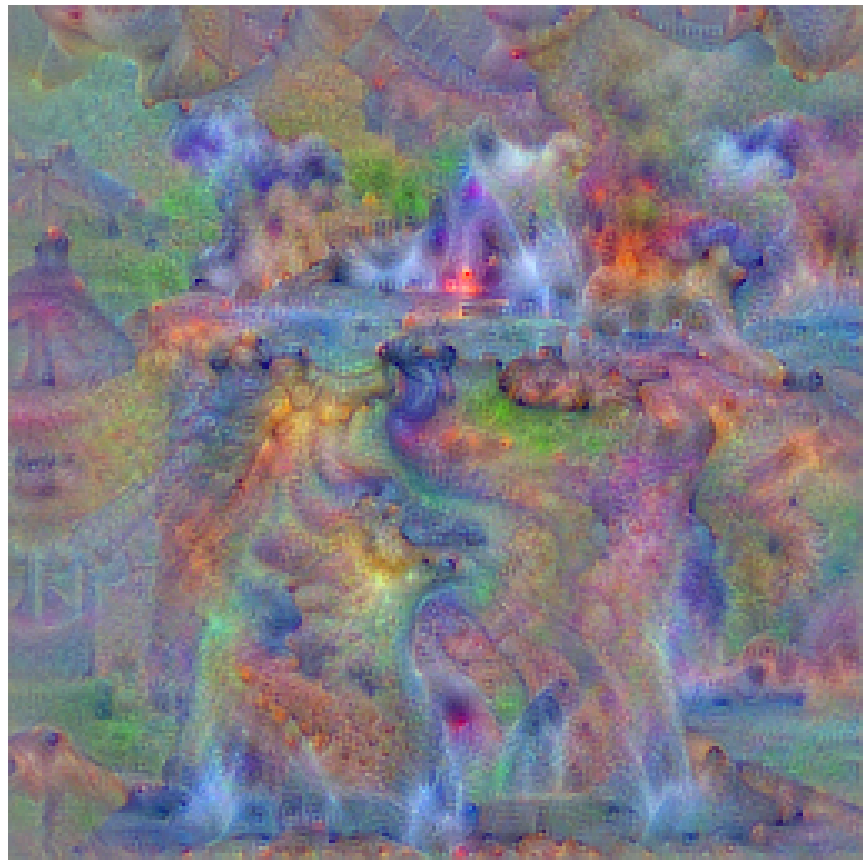
"Ping-pong ball"



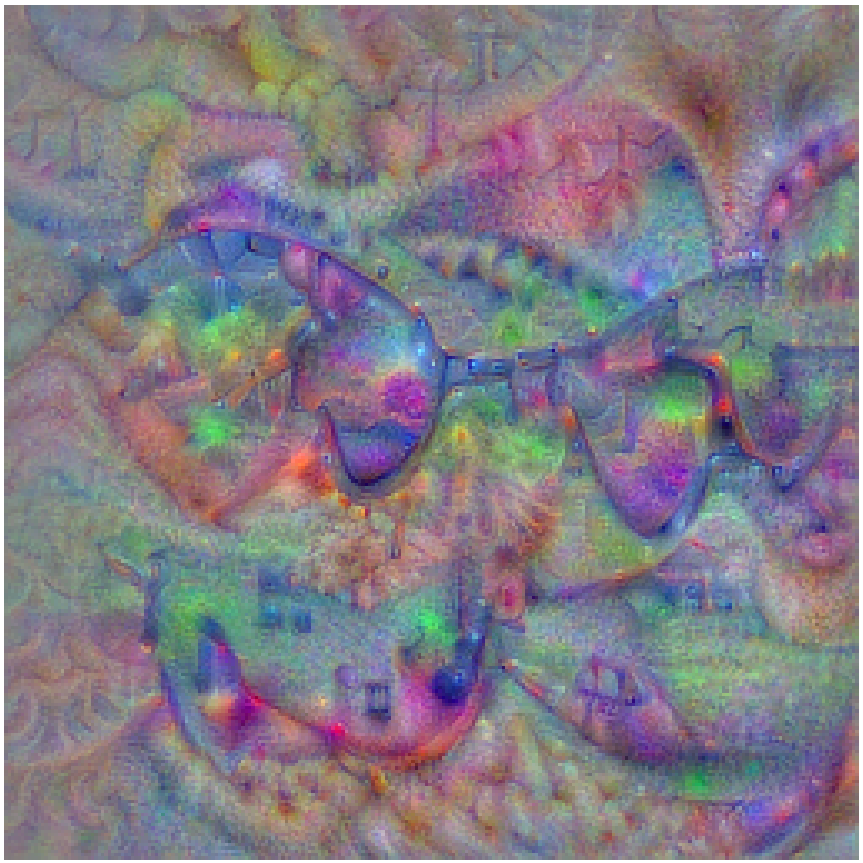
"Steel arch bridge"

VGG16, maximizing a unit of the output layer

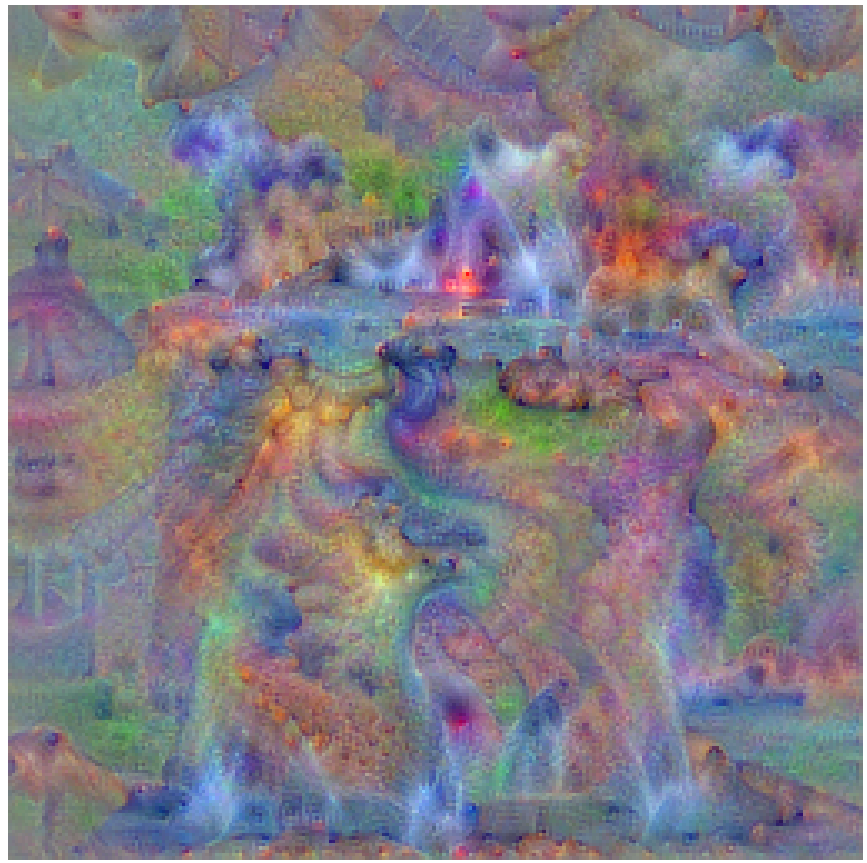




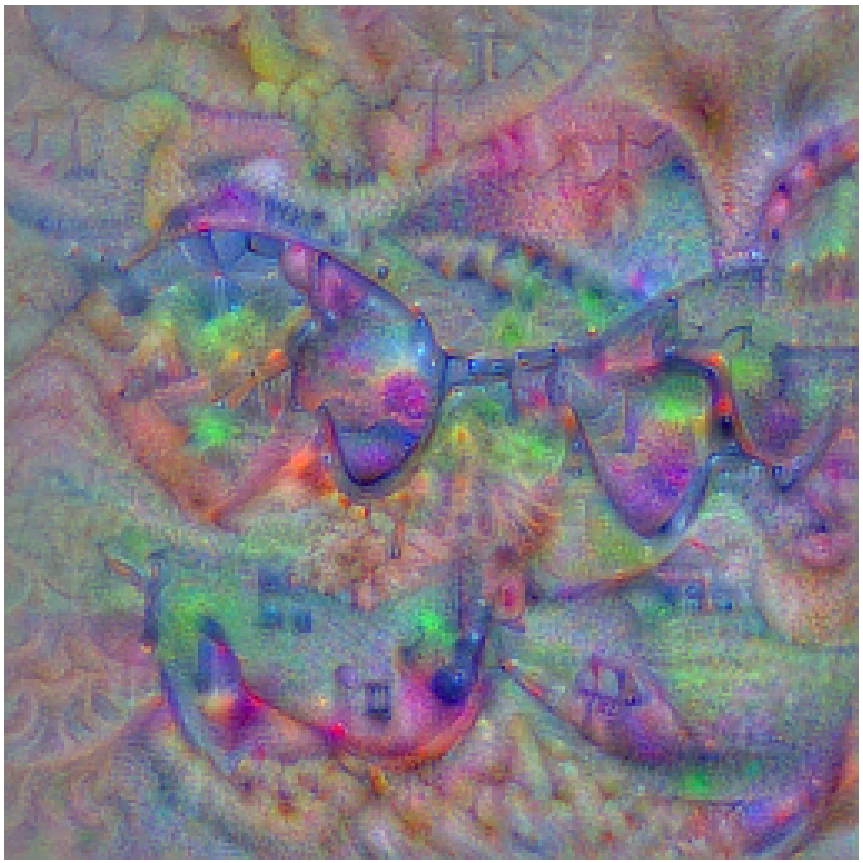
VGG16, maximizing a unit of the output layer



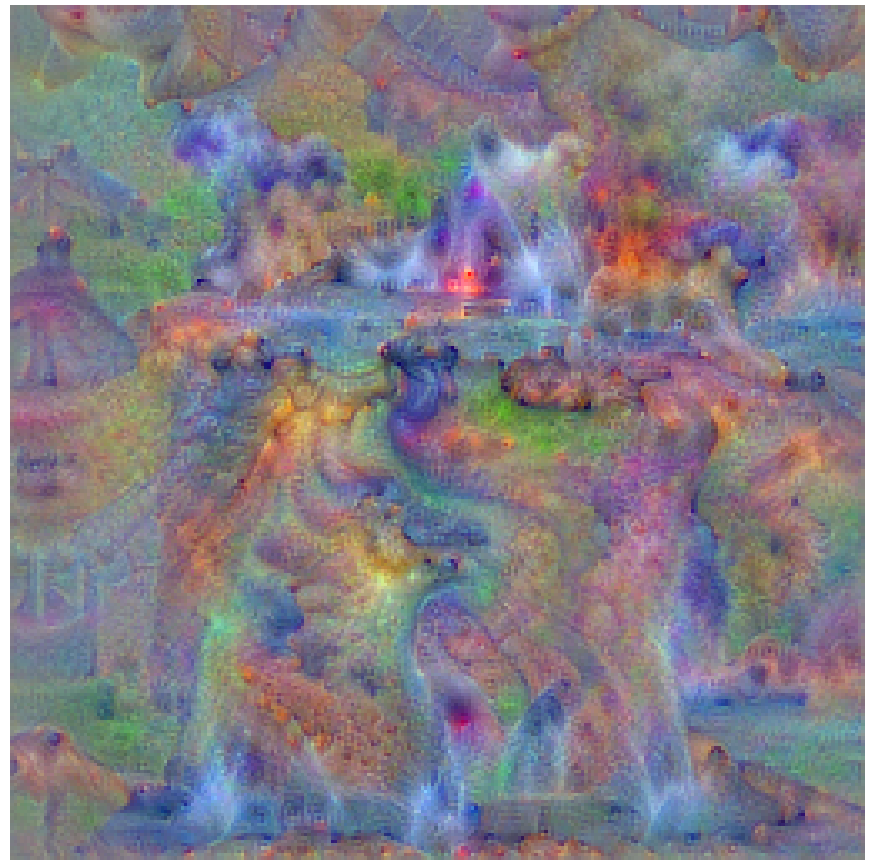
"Sunglass"



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).

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.

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.

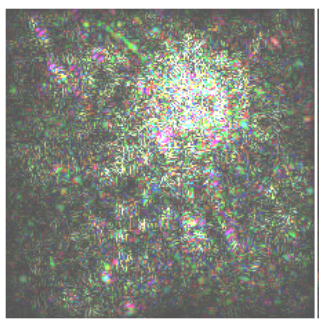
The counter-intuitive result is that the resulting miss-classified images are indistinguishable from the original ones to a human eye.

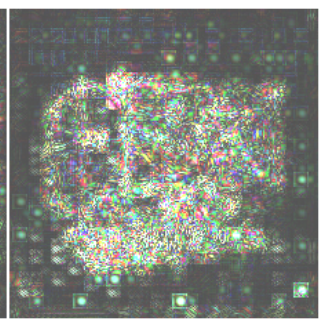
```
input = Variable(input, requires_grad = True)

model = torchvision.models.alexnet(pretrained = True)
cross_entropy = nn.CrossEntropyLoss()
optimizer = optim.SGD([input], lr = 1e-1)

if torch.cuda.is_available():
    model.cuda()
    cross_entropy.cuda()

target = model(input).data.max(1)[1].view(-1)
if torch.cuda.is_available(): target = target.cuda()
target = Variable(target)

for k in range(15):
    output = model(input)
    loss = - cross_entropy(output, target)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Original

Adversarial

Differences (magnified)

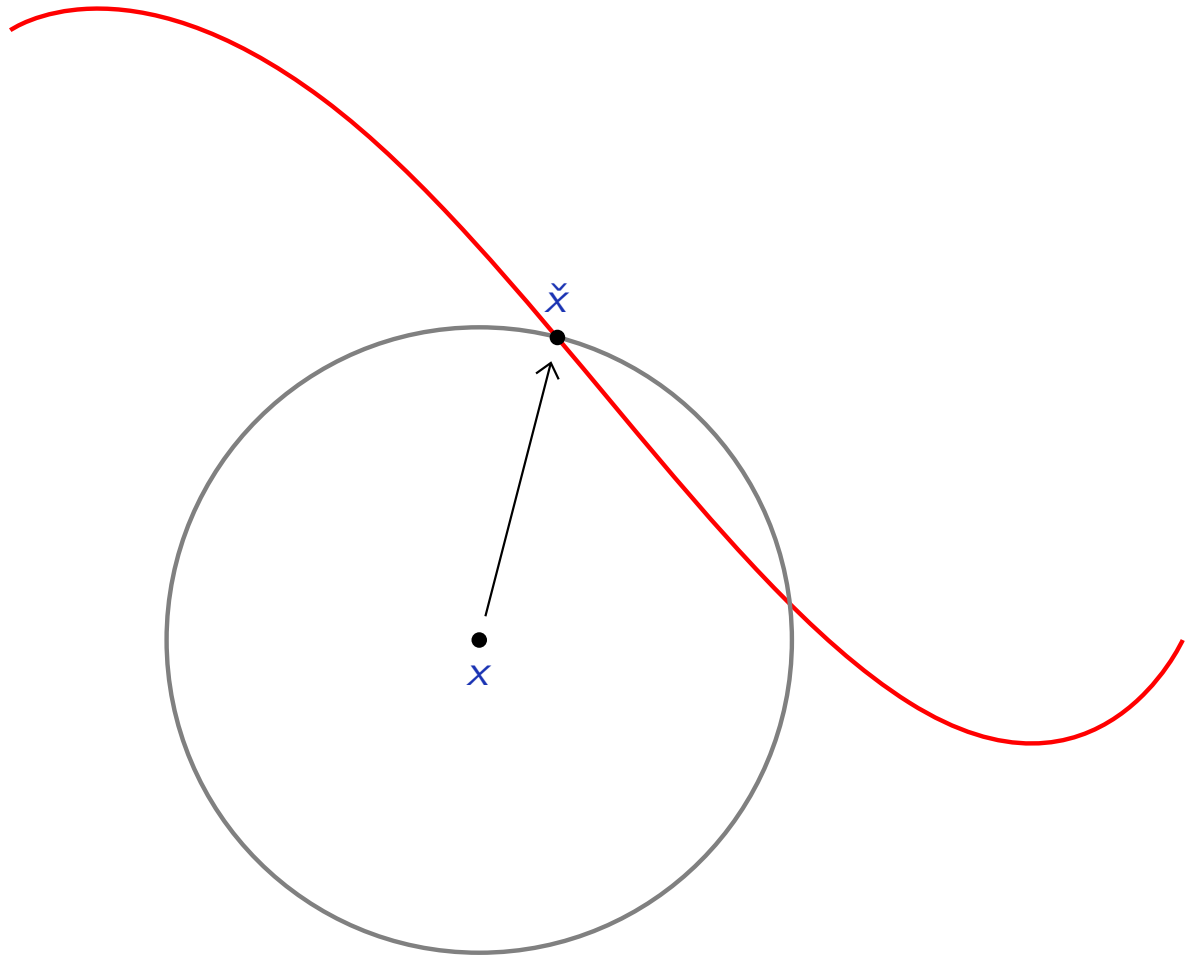




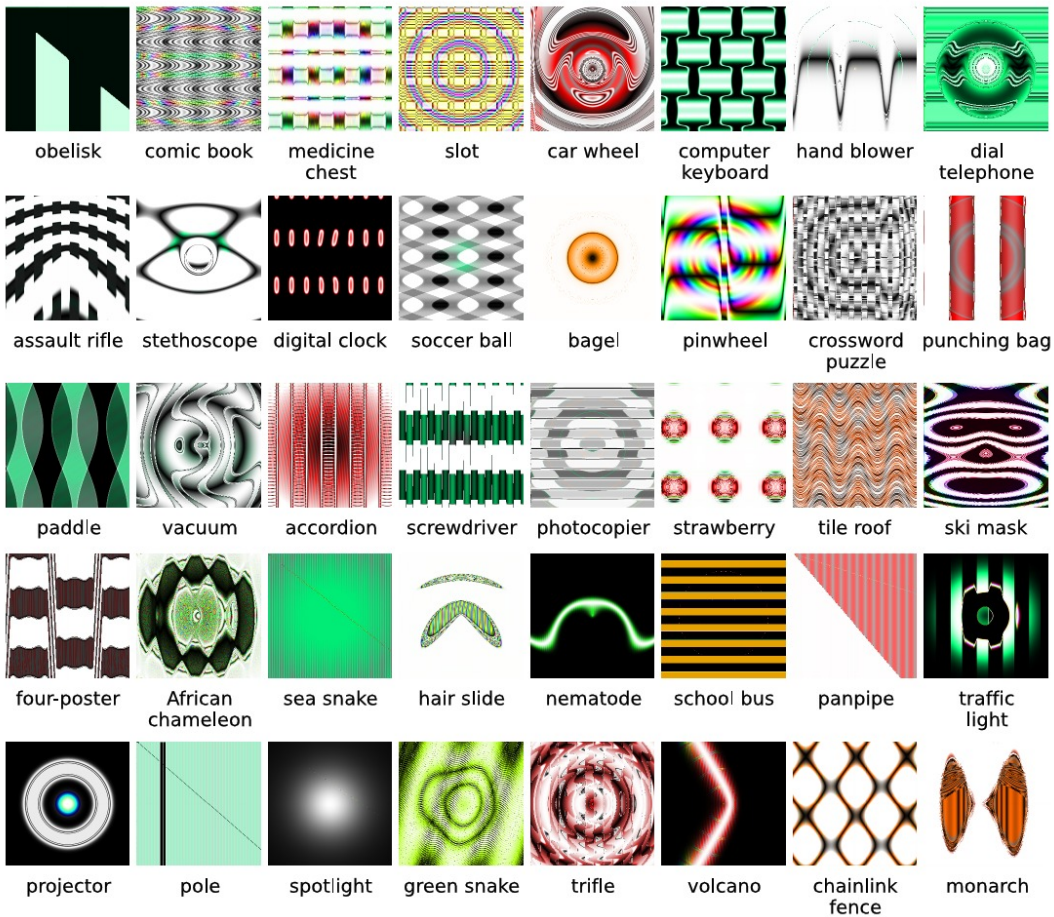
$\frac{\|x-\check{x}\|}{\|x\|}$ 1.02% 0.27%

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.



Adversarial images can be pushed one step further by optimizing images from scratch with genetic optimization to maximize the network's response



(Nguyen et al., 2015)

Dilated convolution

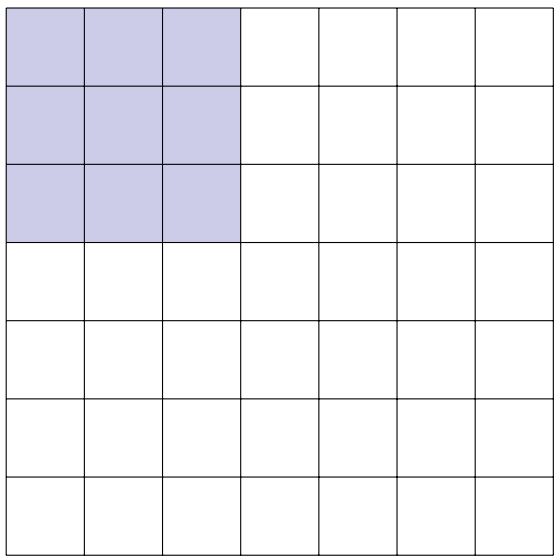
Convolution operations admit one more standard parameter that we have not discussed yet: The dilation, which modulates the expansion of the filter support (Yu and Koltun, 2015).

It is 1 for standard convolutions, but can be greater, in which case the resulting operation can be envisioned as a convolution with a regularly sparsified filter.

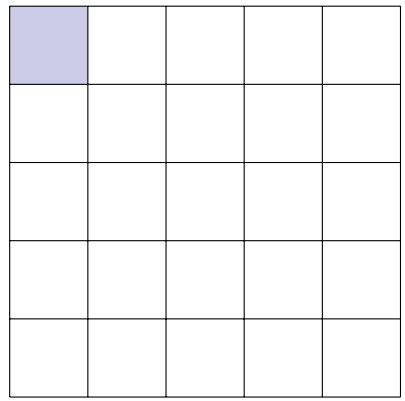
Convolution operations admit one more standard parameter that we have not discussed yet: The dilation, which modulates the expansion of the filter support (Yu and Koltun, 2015).

It is 1 for standard convolutions, but can be greater, in which case the resulting operation can be envisioned as a convolution with a regularly sparsified filter.

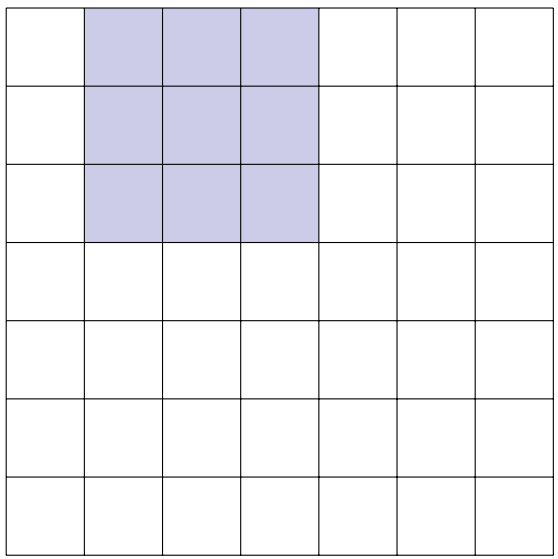
This notion comes from signal processing, where it is referred to as *algorithme à trous*, hence the term sometime used of "convolution à trous".



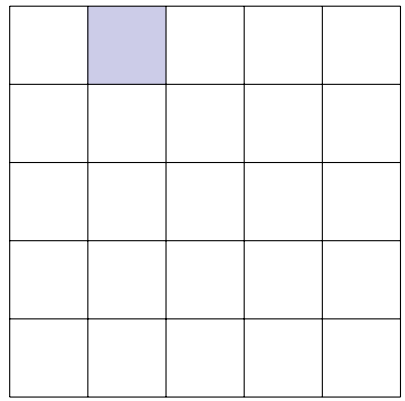
Input



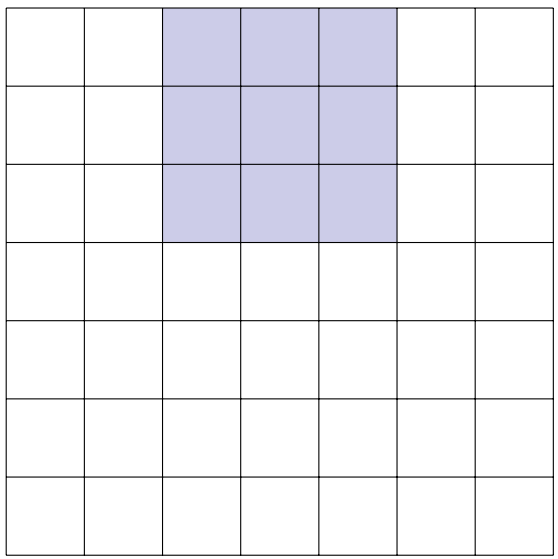
Output



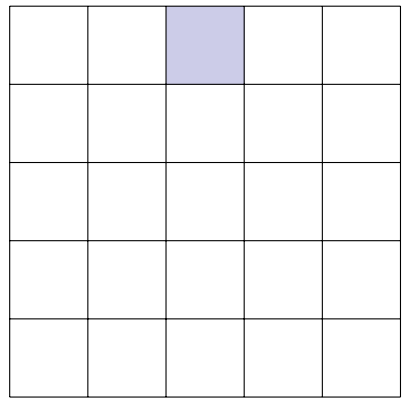
Input



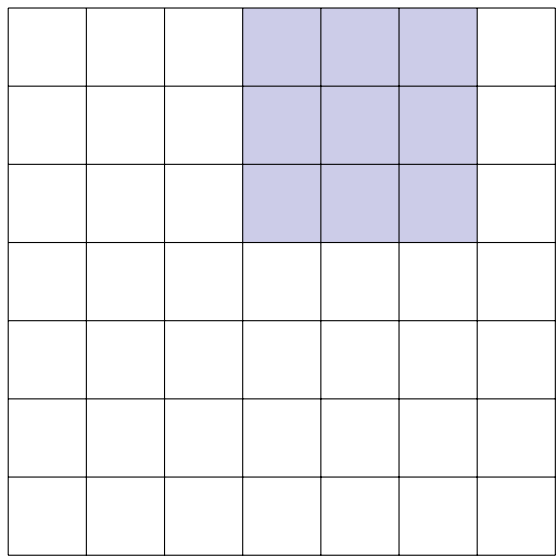
Output



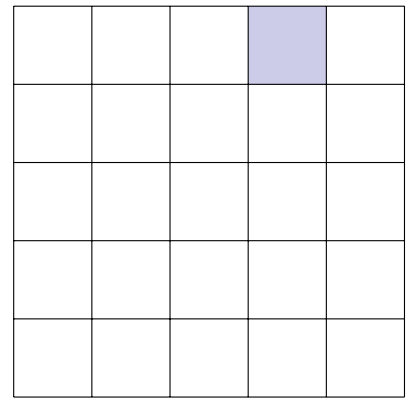
Input



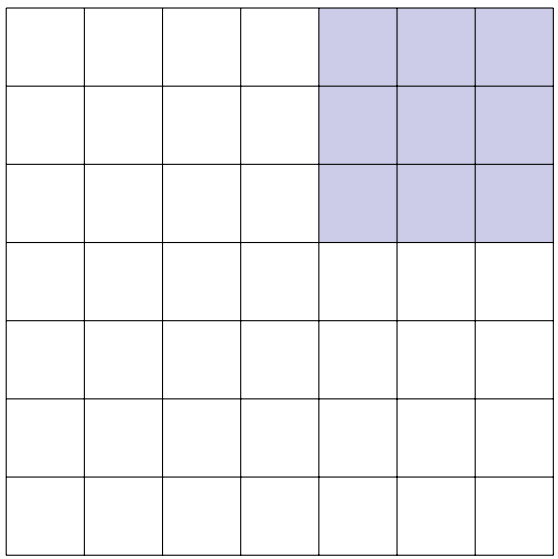
Output



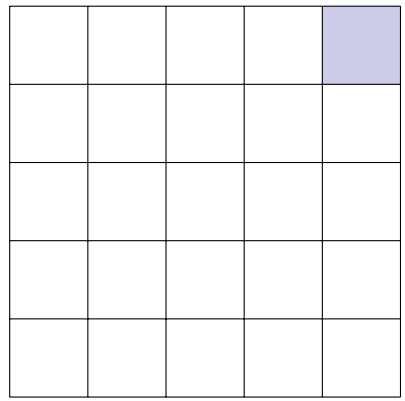
Input



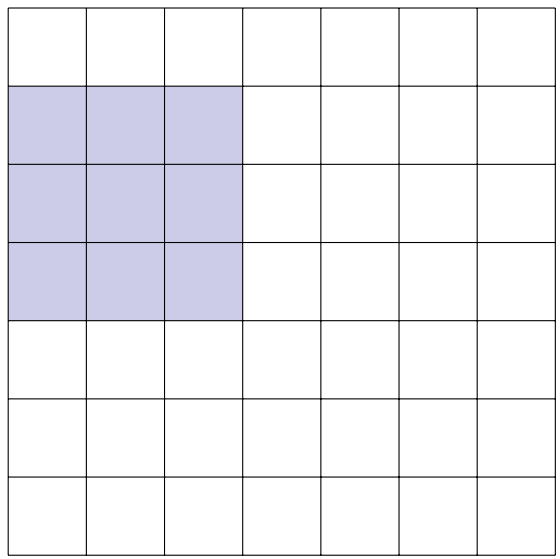
Output



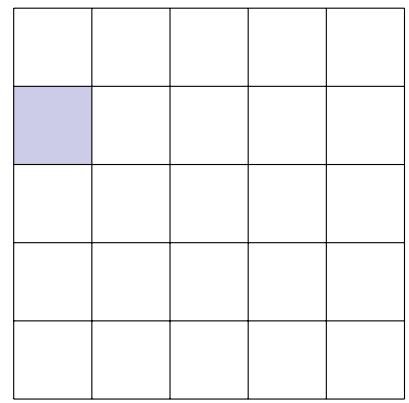
Input



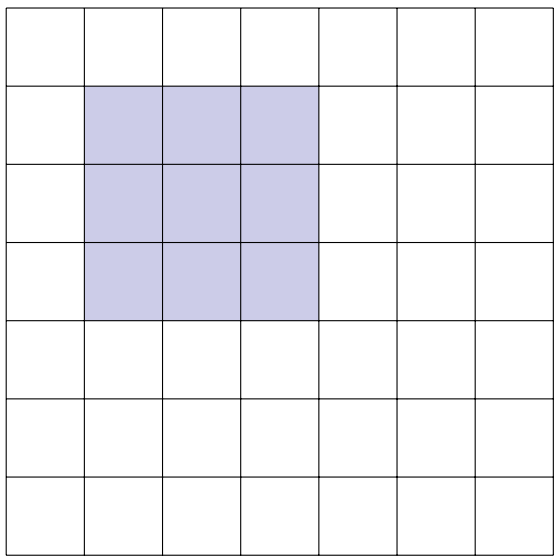
Output



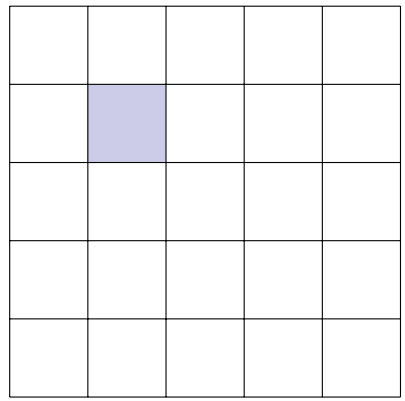
Input



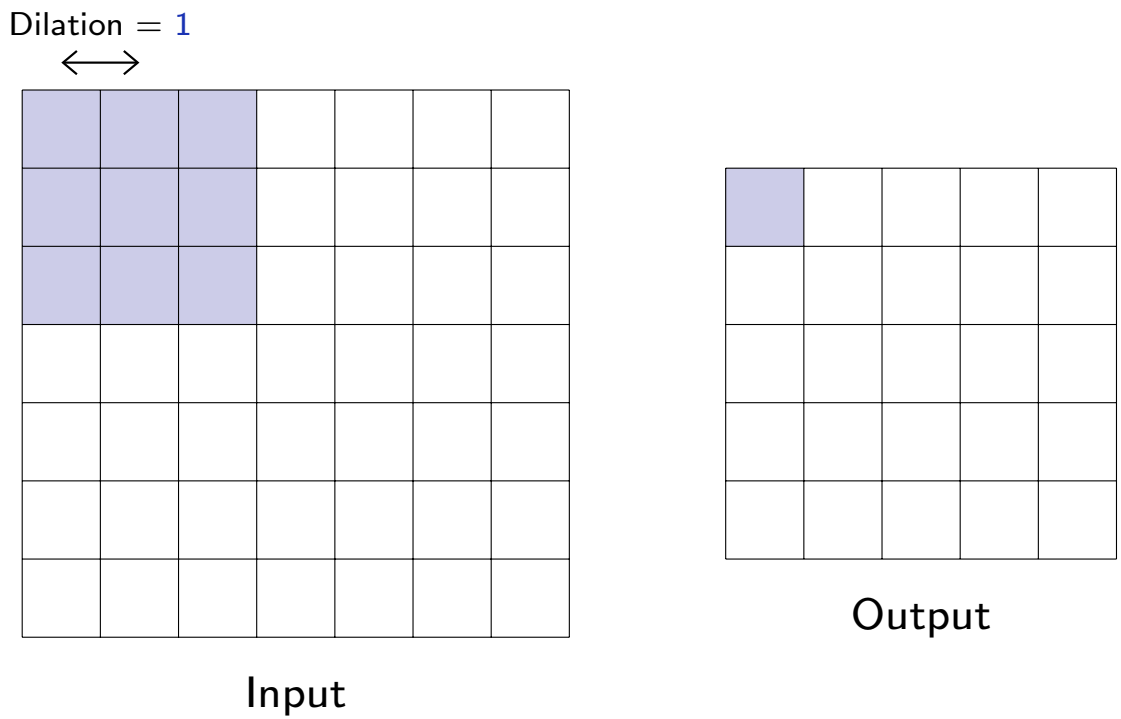
Output

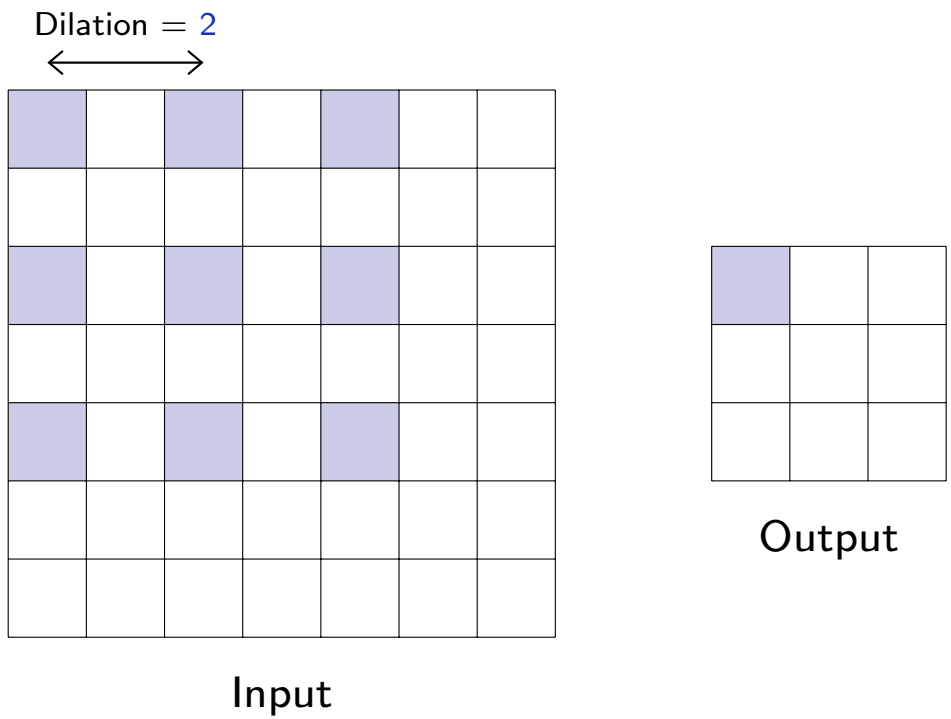


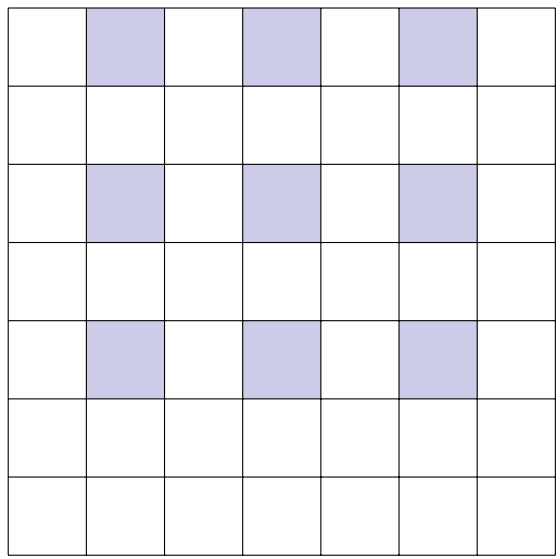
Input



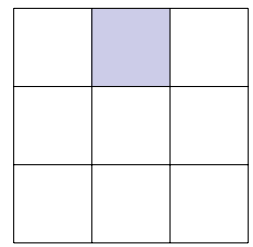
Output


Dilation = 1
Input
Output

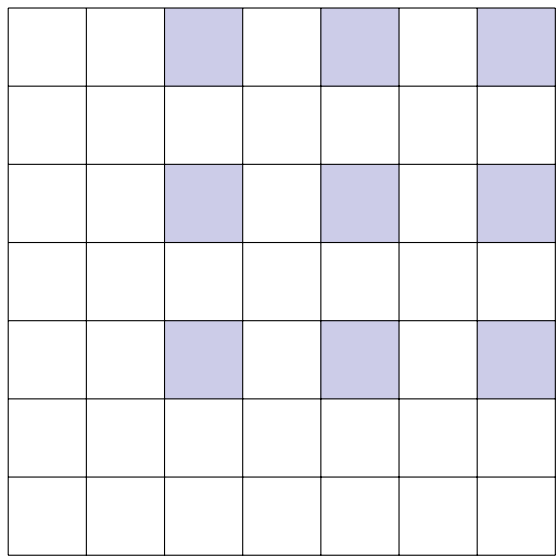

Dilation = 2
Input
Output



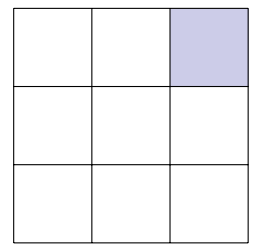
Input



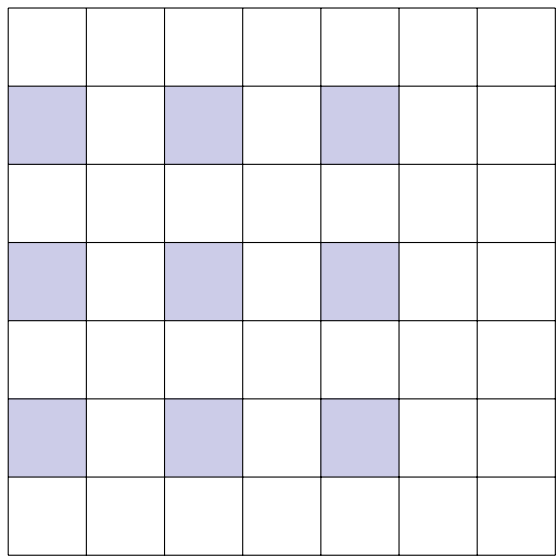
Output



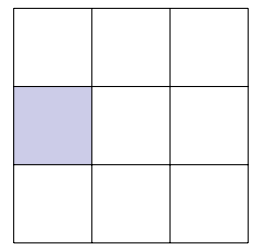
Input



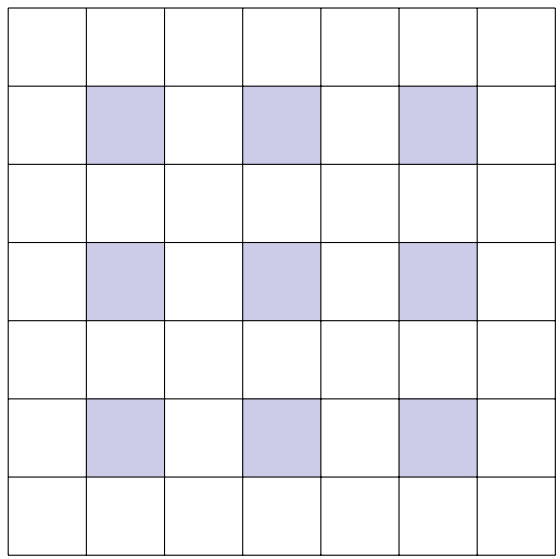
Output



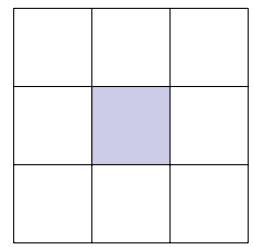
Input



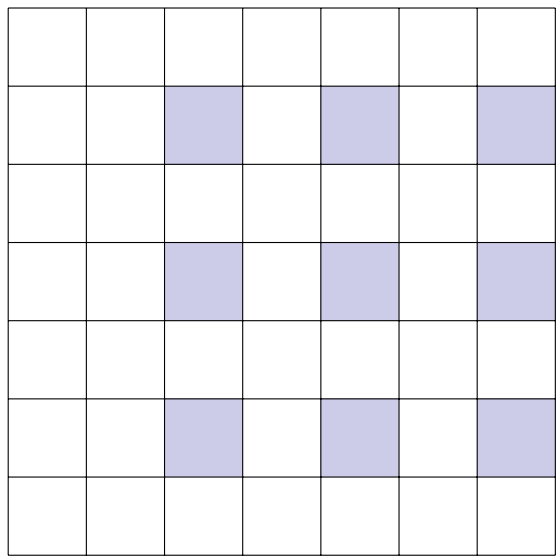
Output



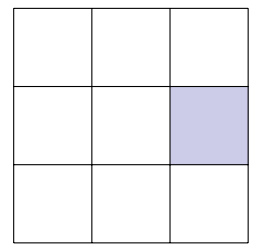
Input



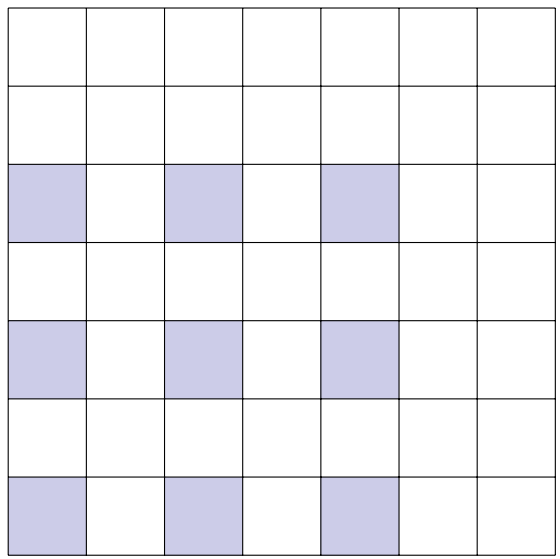
Output



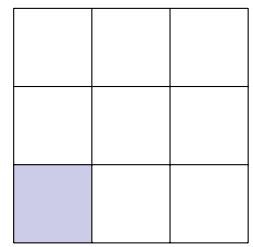
Input



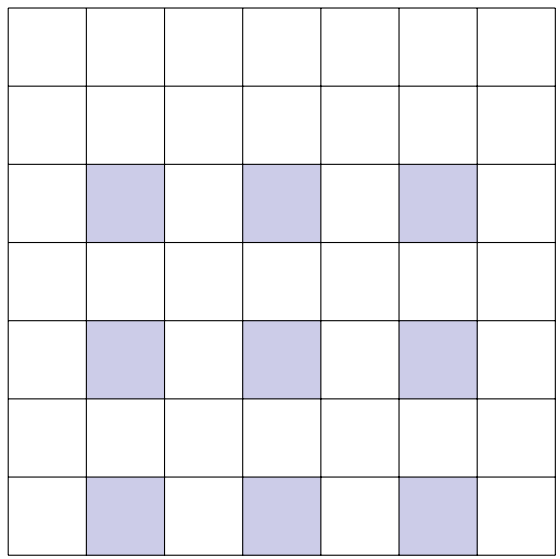
Output



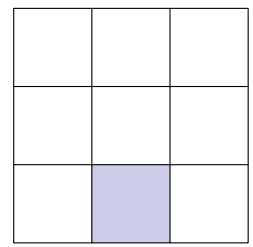
Input



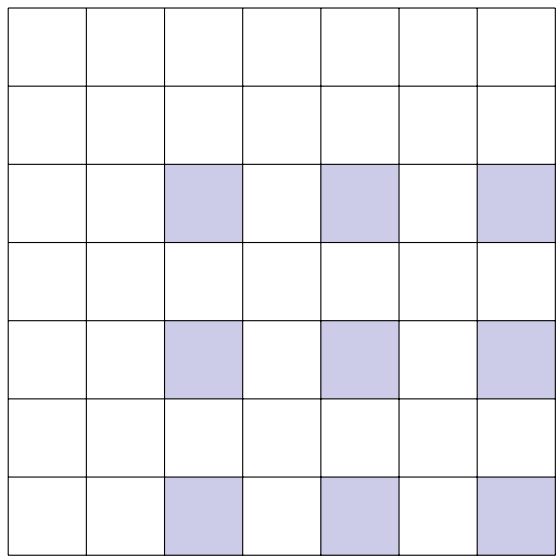
Output



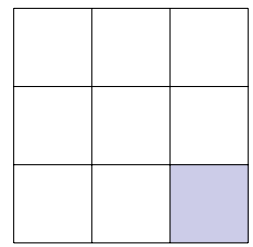
Input



Output



Input



Output

A convolution with a 1d kernel of size k and dilation d can be interpreted as a convolution with a filter of size $1 + (k - 1)d$ with only k non-zero coefficients.

For with $k = 3$ and $d = 4$, the difference between the input map size and the output map size is $1 + (3 - 1)4 - 1 = 8$.

```
>>> from torch import nn, Tensor
>>> from torch.autograd import Variable
>>> x = Variable(Tensor(1, 1, 20, 30).normal_())
>>> l = nn.Conv2d(1, 1, kernel_size = 3, dilation = 4)
>>> l(x).size()
torch.Size([1, 1, 12, 22])
```

Having a dilation greater than one increases the units' receptive field size without increasing the number of parameters.

Convolutions with stride or dilation strictly greater than one reduce the activation map size, for instance to make a final classification decision, without employing pooling operators.

Having a dilation greater than one increases the units' receptive field size without increasing the number of parameters.

Convolutions with stride or dilation strictly greater than one reduce the activation map size, for instance to make a final classification decision, without employing pooling operators.

Such networks have the advantage of simplicity:

- non-linear operations are only in the activation function,
- joint operations (combining multiple activations to produce one) are only in the convolutional layers.

The end

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.

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.

A. M. Nguyen, J. Yosinski, and J. Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

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.

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.

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.

F. Yu and V. Koltun. Multi-scale context aggregation by dilated convolutions. *CoRR*, abs/1511.07122v3, 2015.

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