Deep learning

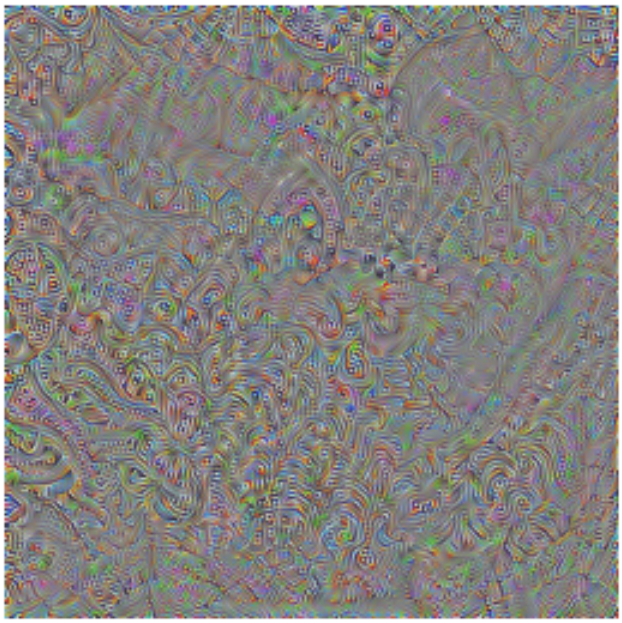
9.4. Optimizing inputs

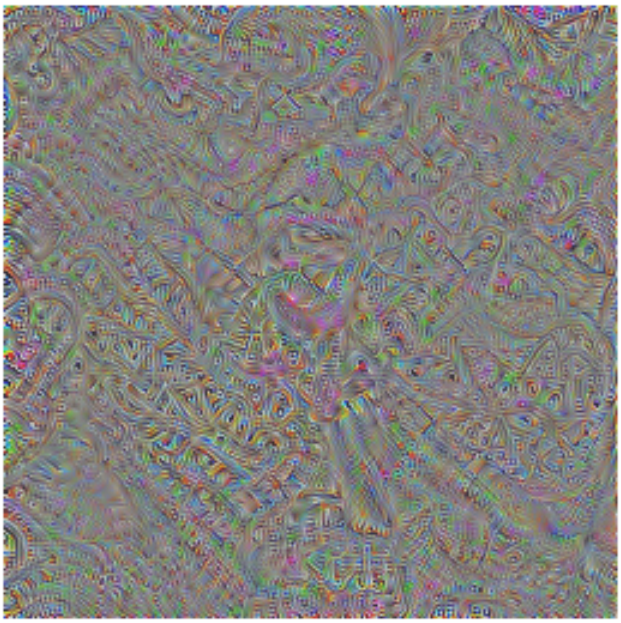
François Fleuret

https://fleuret.org/dlc/

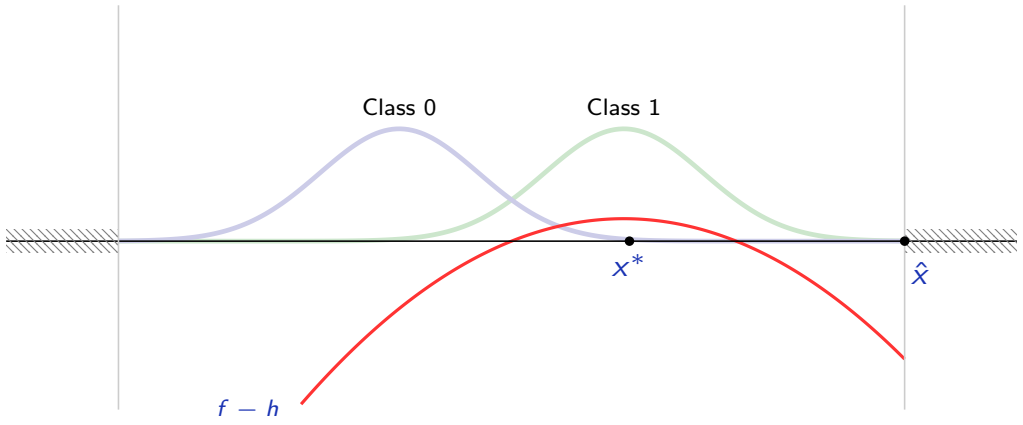
A strategy to get an intuition of the information actually encoded in the weights of a convnet consists of optimizing from scratch a sample to maximize the activation f of a chosen unit, or the sum over an activation map.

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".



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

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

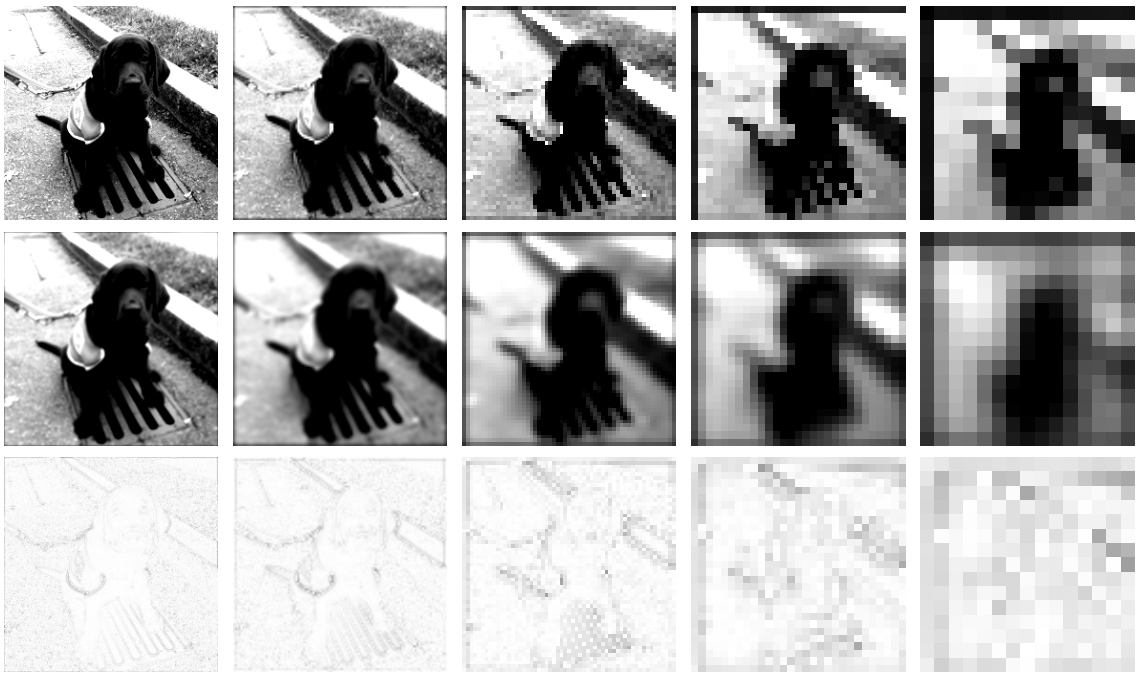
by iterating a standard gradient update:

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

Notes

The key issue is that given a target y, a sample $\hat{x}$ maximizing $P(Y = y \mid X = \hat{x})$ can be extremely unlikely, that is can lead to an arbitrarily small $P(X = \hat{x} \mid Y = y)$.
For instance, under a Gaussian model of the men and women heights, although it is very unlikely that a man would be 3m tall, a 3m tall human would be far more likely to be a man than a woman than a 2m tall human.

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



Notes

We saw that the baseline procedure generates too much high frequencies. The goal is to design a penalty which spreads the energy across frequencies.
The images of the top row here shows the same original image at different resolutions. The second row shows blurred versions of them. And the third row is the difference between the two.
The more non-zero pixels there is in the left images of that third row, the greater the energy in the high frequencies.

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.

Notes

The penalty is the sum across scales of the square distance between a downscaled image $\delta^s(x)$ and a blurred version of it $g \circledast \delta^s(x)$. Each term in the sum is one of the images of the third row of the previous slide.
The quadratic form of this penalty makes it lower when the energy is spread-out across terms.

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

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:

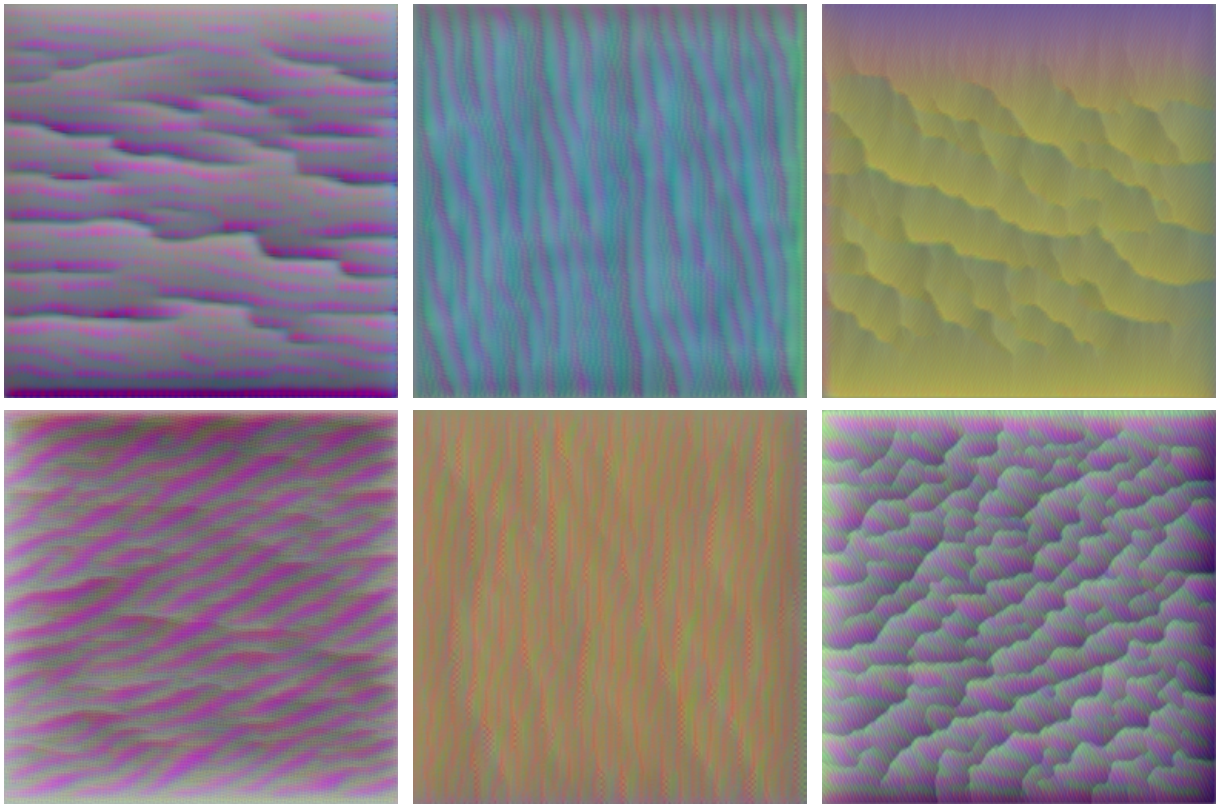
```
model = models.vgg16(weights = 'IMAGENET1K_V1')
model.eval()
edge_energy = MultiScaleEdgeEnergy()
input = torch.empty(1, 3, 224, 224).normal_(0, 0.01)

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)

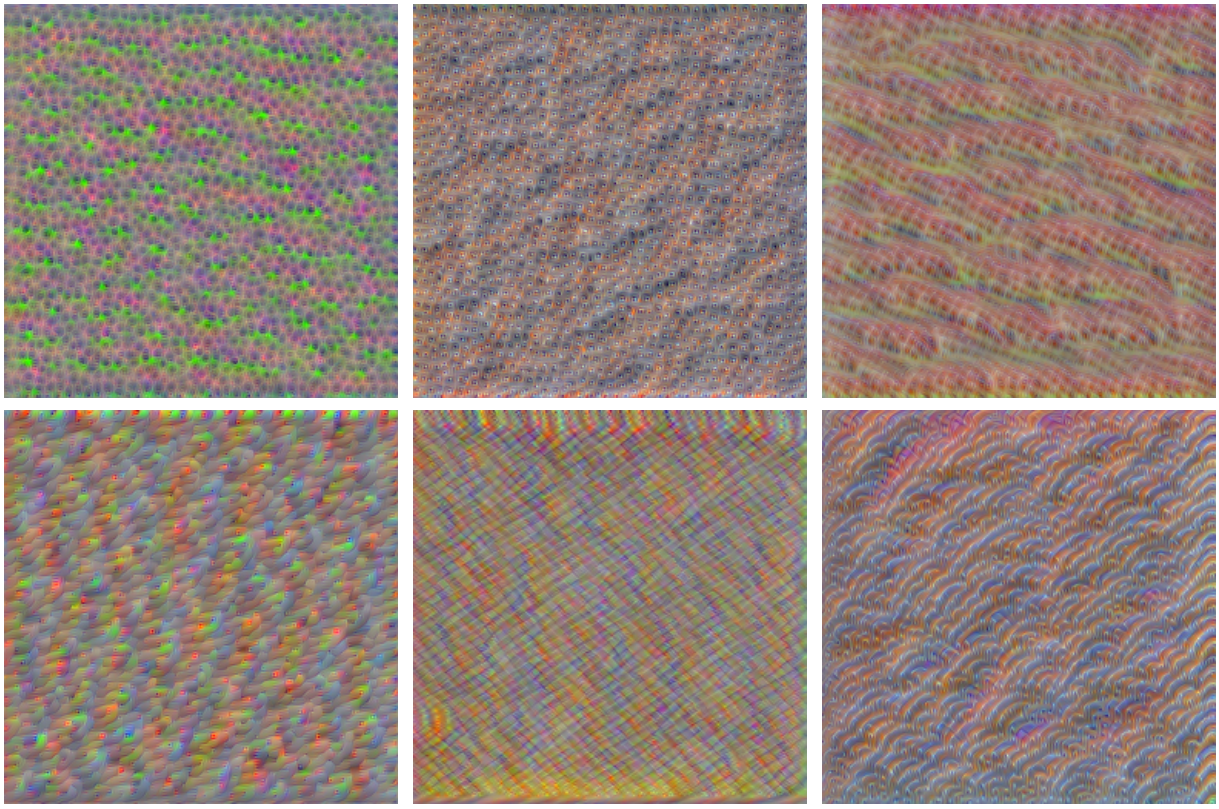
Notes

This code shows how to optimize an image to make a specific unit have a high response, here the output unit of index 700 corresponding to the class "paper towel"
The image to optimize is initialized with Gaussian noise of standard deviation 0.01, and with the default input size of VGG nets $3 \times 224 \times 224$.
`requires_grad_()` is called on the corresponding tensor because the derivative of the loss w.r.t. it will be needed.

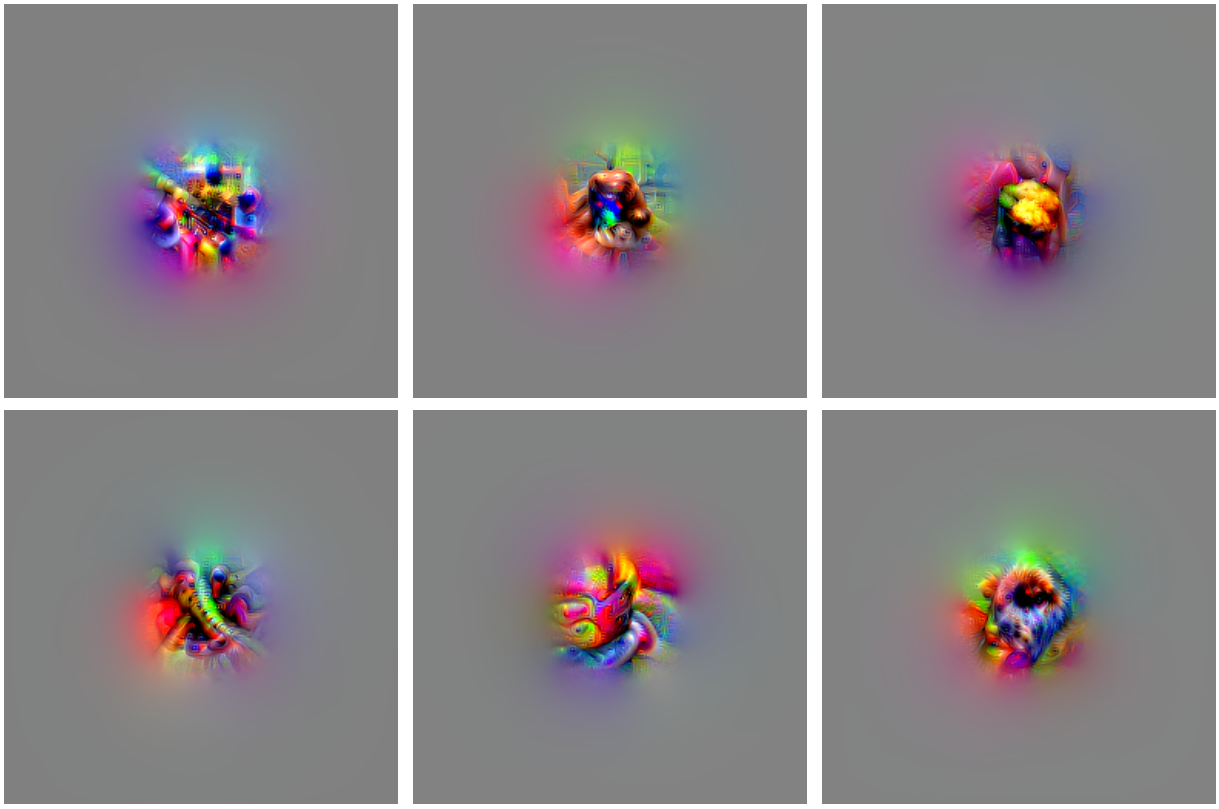
VGG16, maximizing a channel of the 4th convolution layer



VGG16, maximizing a channel of the 7th convolution layer



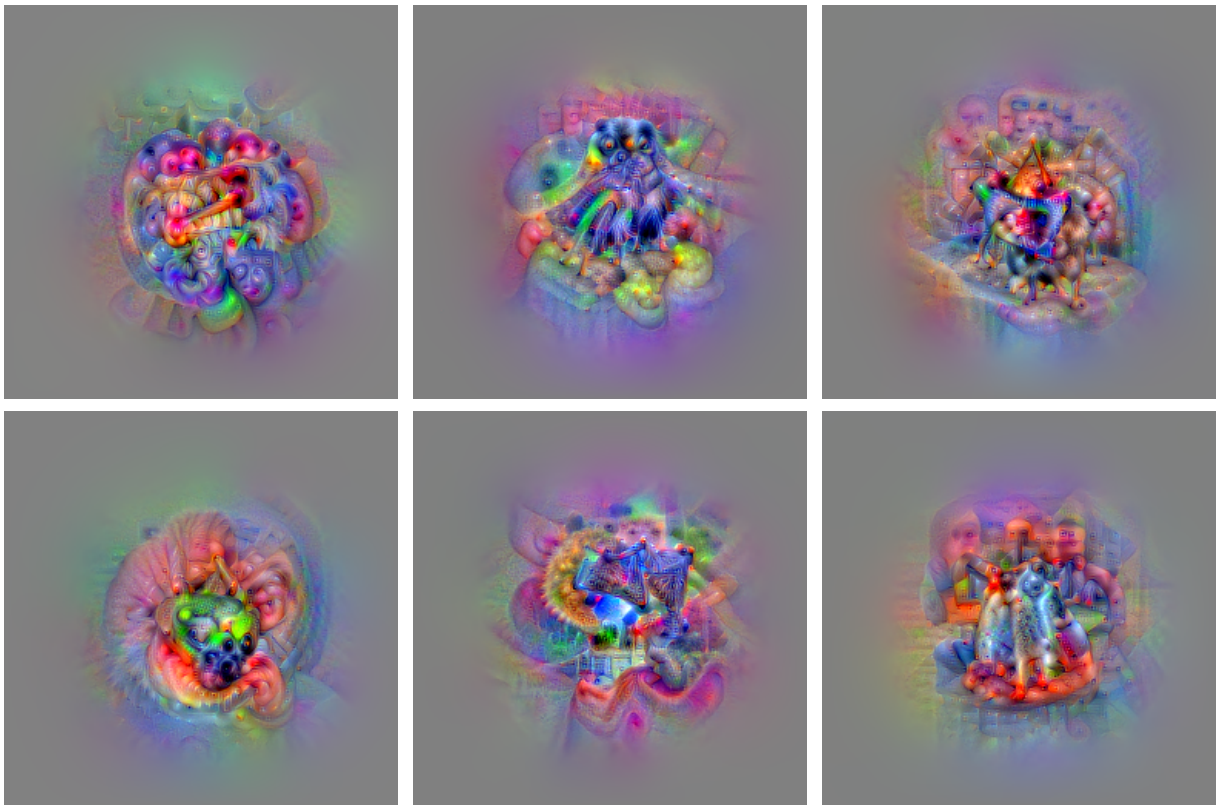
VGG16, maximizing a unit of the 10th convolution layer



Notes

In the 10th layer, we optimize a single unit in the center. Therefore, only its receptive field in the input image can be optimized.

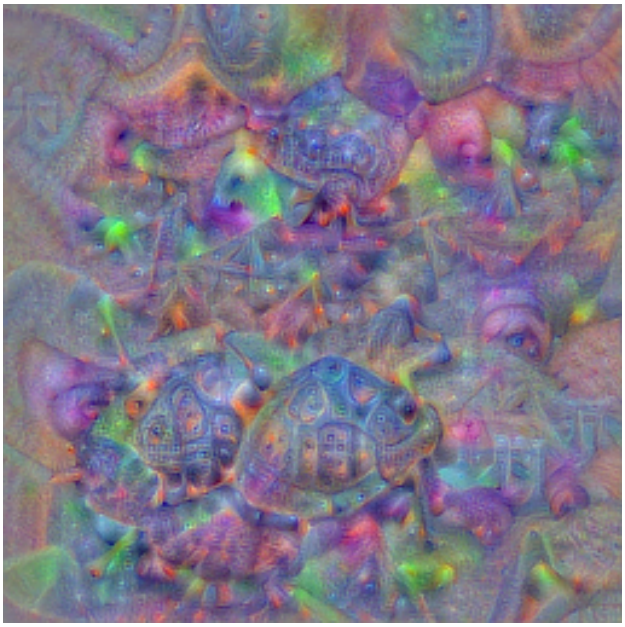
VGG16, maximizing a unit of the 13th (and last) convolution layer



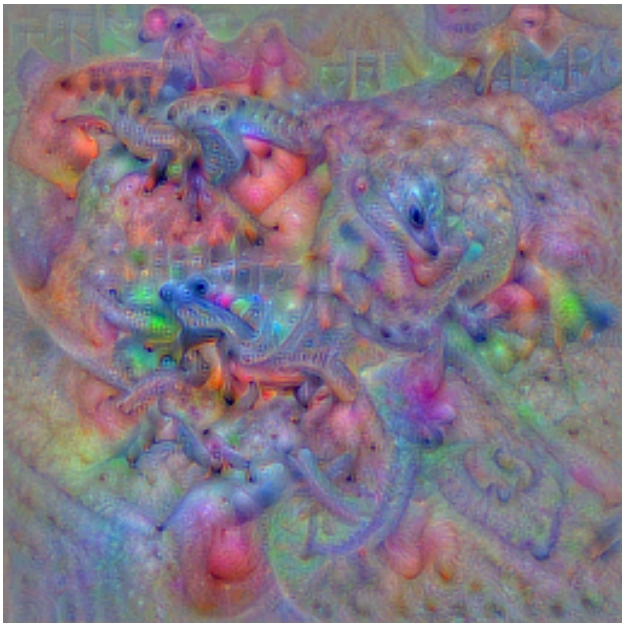
Notes

In the 13th layer, we optimize a single unit in the center. The receptive field is larger than in the 10th layer, causing the pattern to be larger than before. Here, pieces of objects are emerging.

VGG16, maximizing a unit of the output layer



"Box turtle"

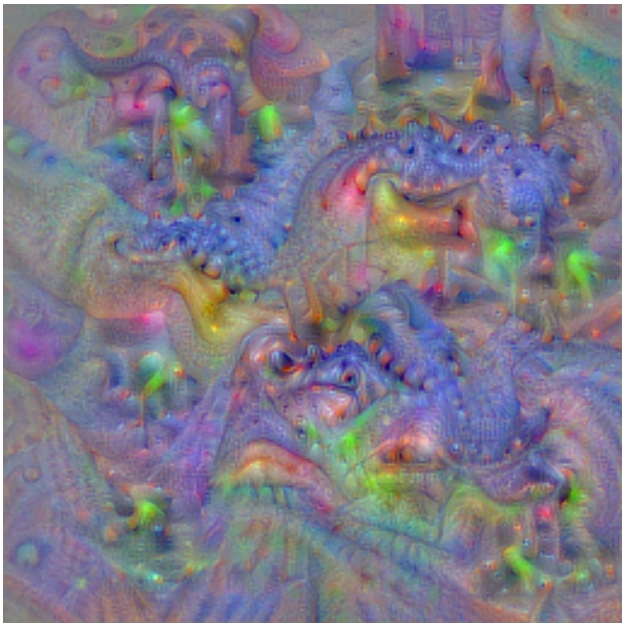


"Whiptail lizard"

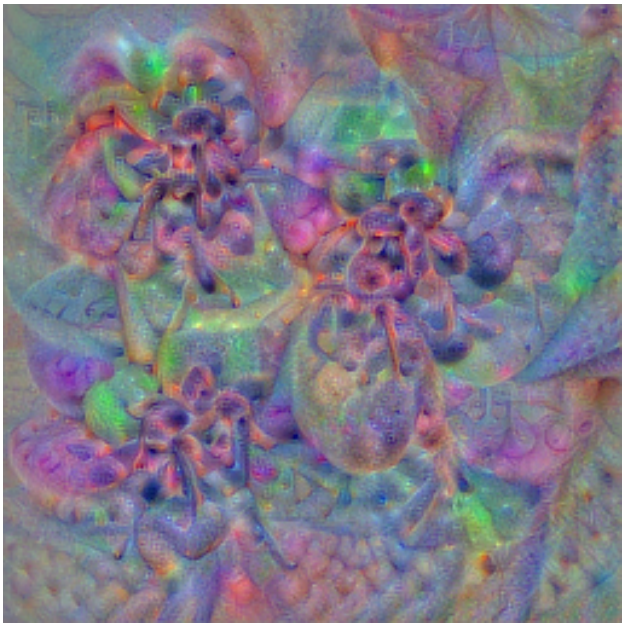
Notes

Here, the output unit corresponding to an actual class is maximized.
Despite many parts of the class object being present in the generated image, one major shortcoming is that the numbers of parts is wrong and they are the overall consistency (symmetry, relative positions) is not enforced. Big chunks are present but the general structure is incorrect.
We can also see patterns emerging which are not directly the class itself, but other objects often present in the context of the picture: chairs or tables for class "king crab", castle or domes for "geyser", etc.

VGG16, maximizing a unit of the output layer

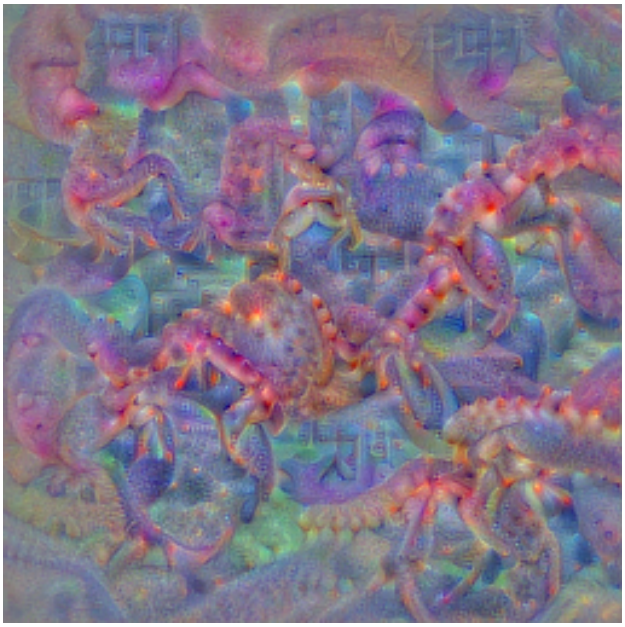


"African chameleon"

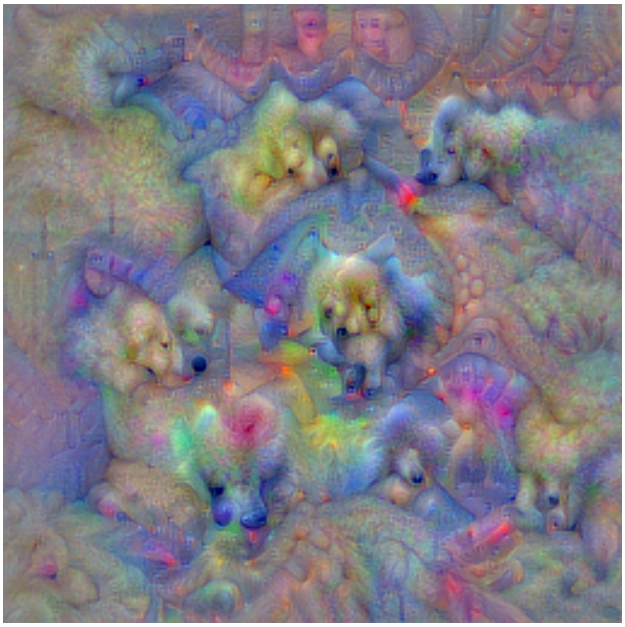


"Wolf spider"

VGG16, maximizing a unit of the output layer



"King crab"



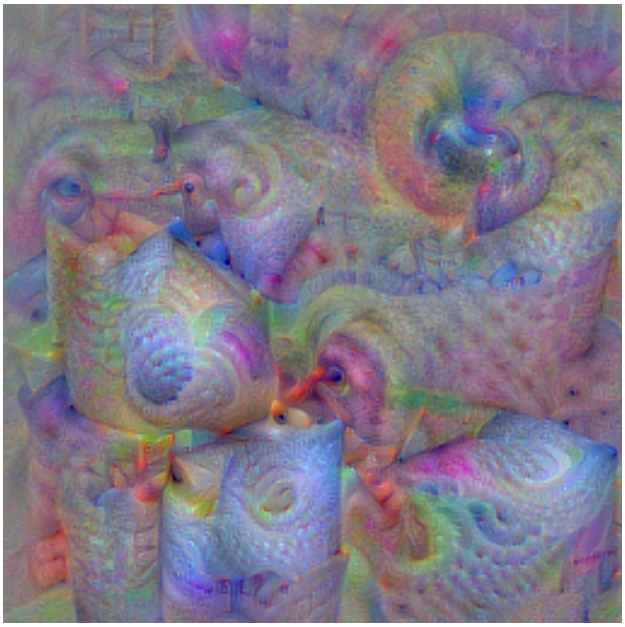
"Samoyed" (that's a fluffy dog)

Notes

For the crab, we can recognize legs. There are other rectangular patterns which are probably due to crabs being usually photographed on plates in restaurants, and these pattern may come from the corners of chairs or tables.

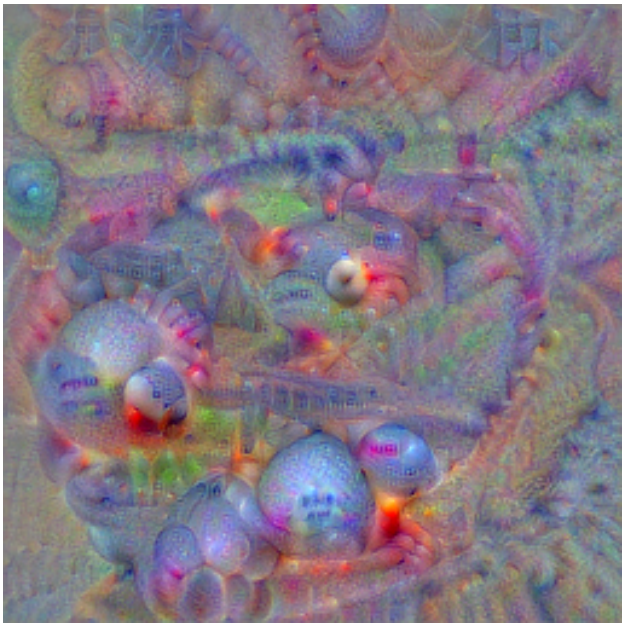
VGG16, maximizing a unit of the output layer

"Hourglass"

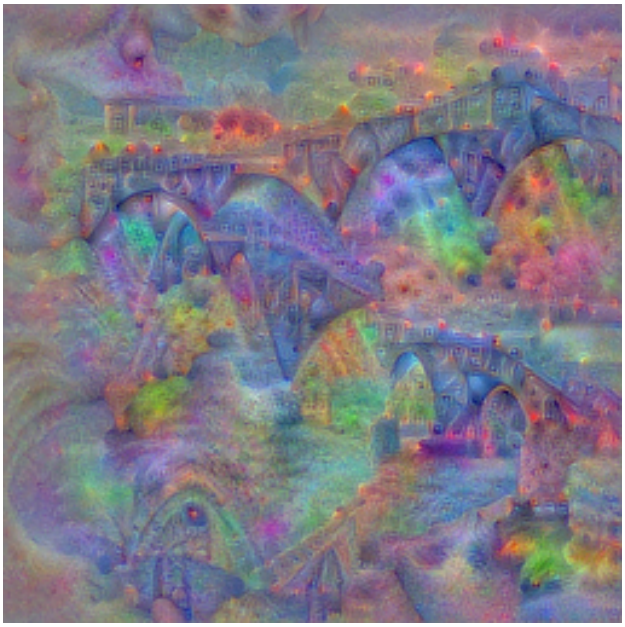


"Paper towel"

VGG16, maximizing a unit of the output layer

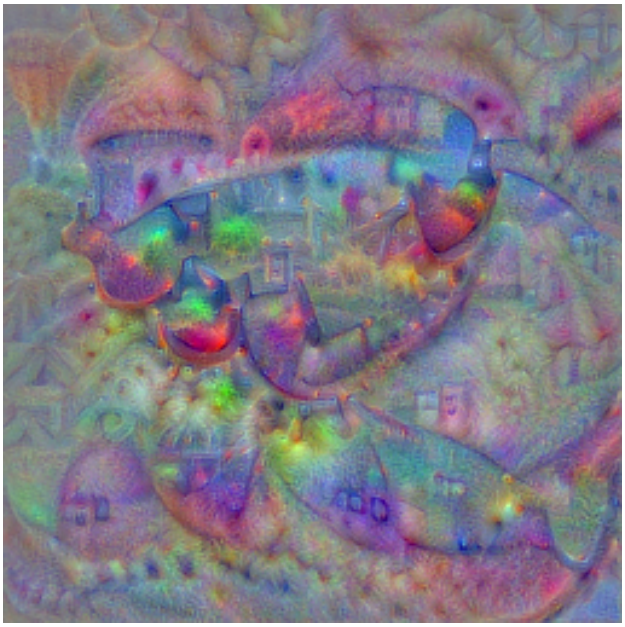


"Ping-pong ball"

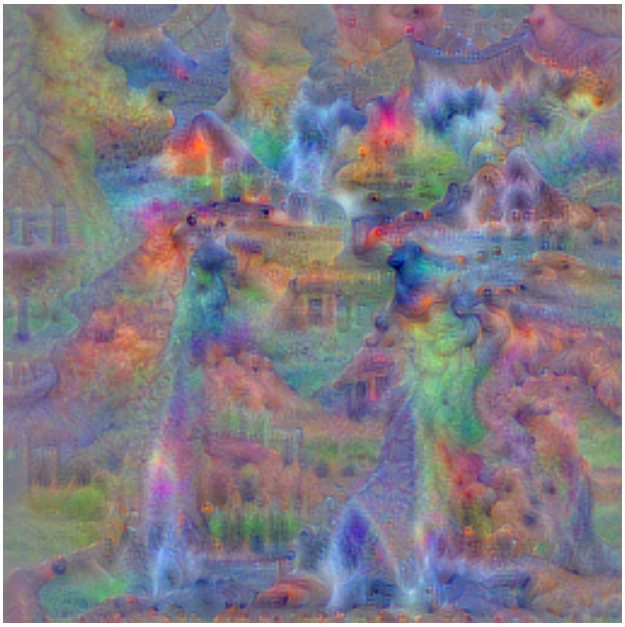


"Steel arch bridge"

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.

Notes

Usually what is done during backpropagation, is to optimize the parameters w of a model to minimize the loss. Here, to generate an adversarial sample, we optimize the input sample x to maximize the loss.

```
model = torchvision.models.alexnet(weights = 'IMAGENET1K_V1')
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()
```

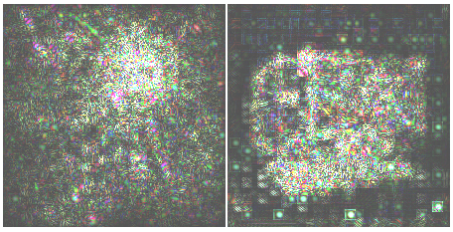
Notes

To perform gradient ascent, we do gradient descent on the opposite of the loss, hence the minus sign.

Original

Adversarial

Differences (magnified)



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

Notes

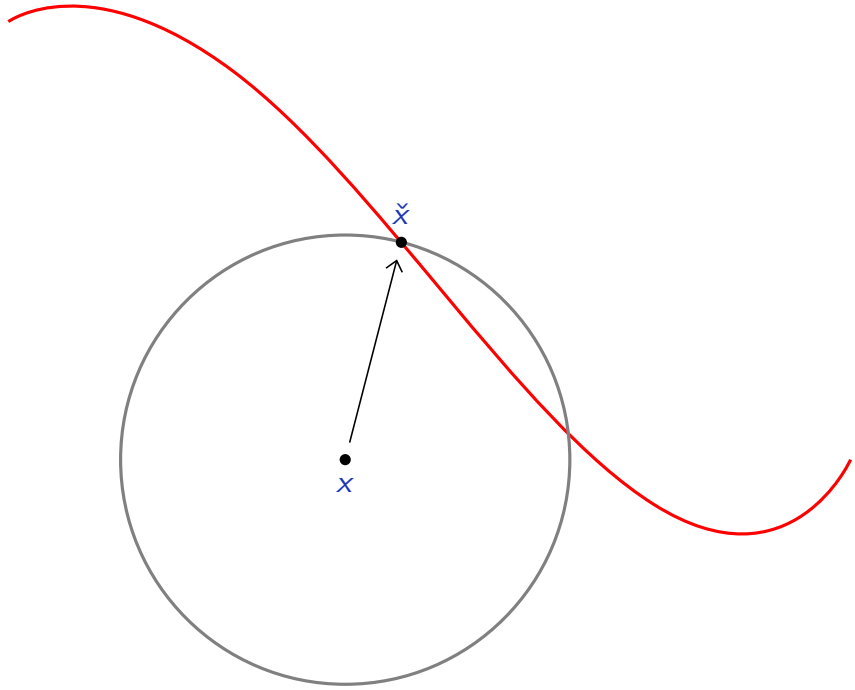
The images in the top row are the originals, respectively classified (correctly) as "Weimaraner" and "desktop computer". The images of the second row are the adversarial examples obtained with the optimization presented in the previous slide, which are classified respectively as "sundial" and "desk".
The surprising and counter-intuitive observation is that these latter images do not differ substantially from the original images to the human eye.

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

Notes

We monitor here the classification of the modified images after each step of the adversarial optimization. We see that the dog is initially [miss-]classified as other animals before being [miss-]classified as "sundial". This mistake may be due to the dark bars in the sewer grid that resemble elongated shadows.

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.



Notes

- x represents a correctly classified image by the network.
- $\check{x}$ is the adversarial sample generated by gradient ascent: it is misclassified by the network.
- The red line shows the boundary between the correct class and the class the adversarial sample is classified in.
- The gray circle shows the samples which are equidistant to x, at the same distance between x and $\check{x}$: they have a perturbation of the same magnitude.
- A large part of the circle is on the same size as x: thus corresponds to the set of samples correctly classified by the model.
- A small portion of the circle is on the other side of the boundary: these are the samples misclassified by the network.

When we randomly select $1{,}000$ samples on the circle, that is samples as perturbed as $\check{x}$ is from x, none of them are misclassified: a random perturbation of the same magnitude is statistically very unlikely to actually fool the network.

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.