Deep learning

9.4. Optimizing inputs

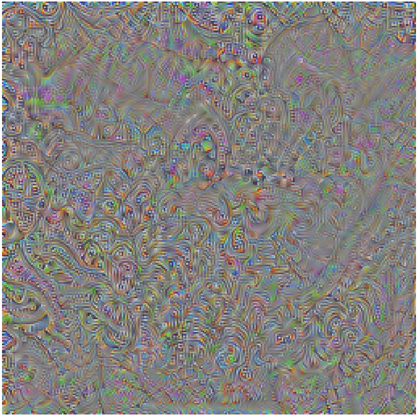
François Fleuret

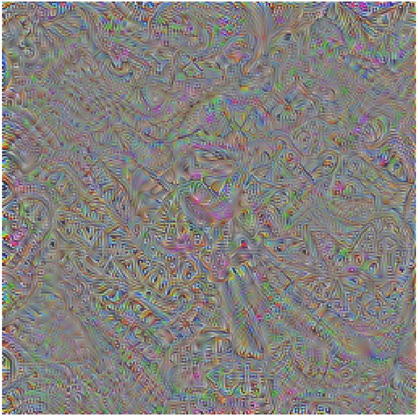
https://fleuret.org/dlc/

UNIVERSITÉ DE GENÈVE

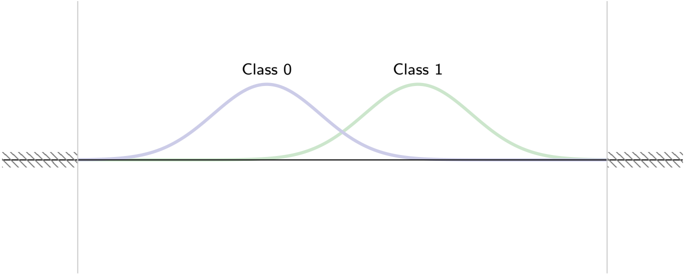
A strategy to get an intuition of the information actually encoded in the weights of a convnet consists of optimizing from scratch a sample to maximize the activation f of a chosen unit, or the sum over an activation map.

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

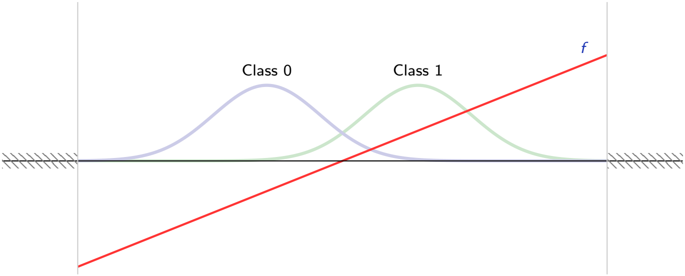




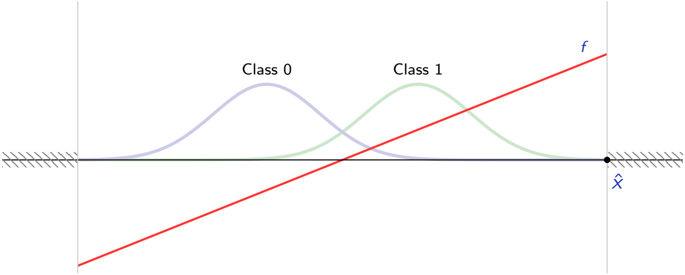
Since f is trained in a discriminative manner, a sample $\hat{x}$ maximizing it has no reason to be "realistic".



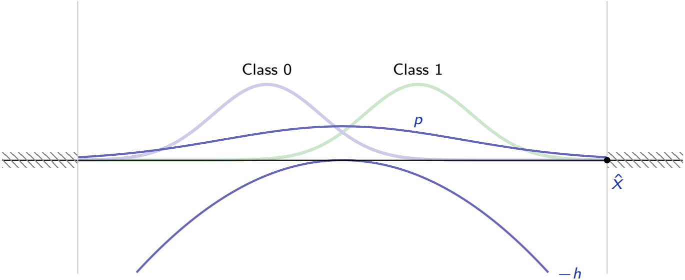
Since f is trained in a discriminative manner, a sample $\hat{x}$ maximizing it has no reason to be "realistic".



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



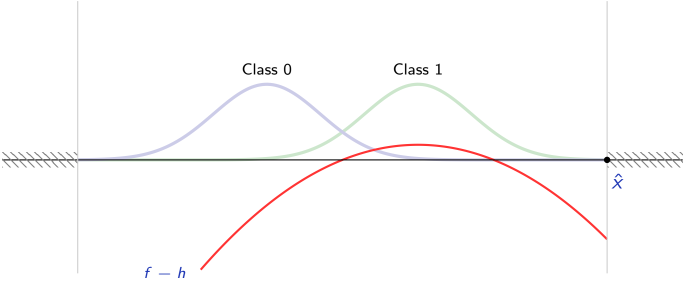
Since f is trained in a discriminative manner, a sample $\hat{x}$ maximizing it has no reason to be "realistic".



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

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

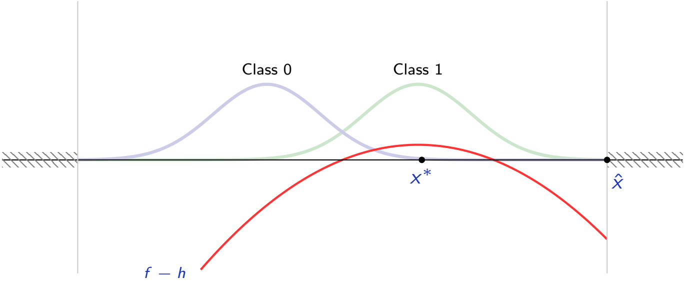
Since f is trained in a discriminative manner, a sample $\hat{x}$ maximizing it has no reason to be "realistic".



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

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

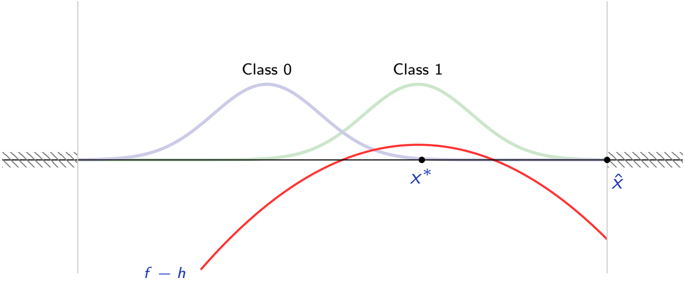
Since f is trained in a discriminative manner, a sample $\hat{x}$ maximizing it has no reason to be "realistic".



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

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

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



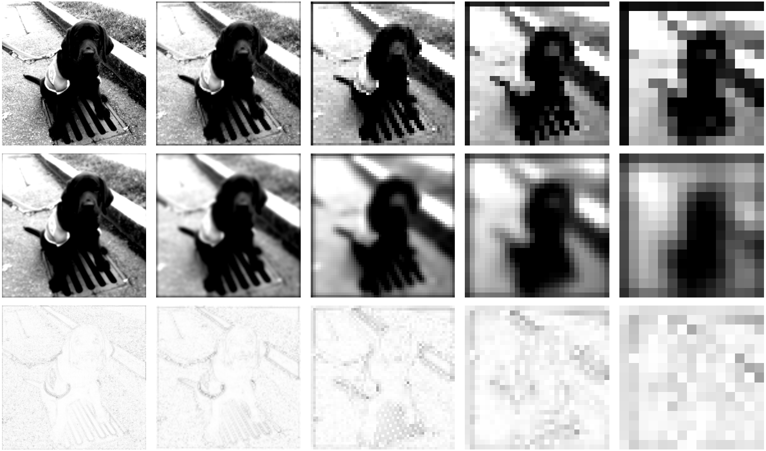
We can mitigate this by adding a penalty h corresponding to a "realistic" prior, that is compute

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

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 downscale-by-two operator.

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

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:

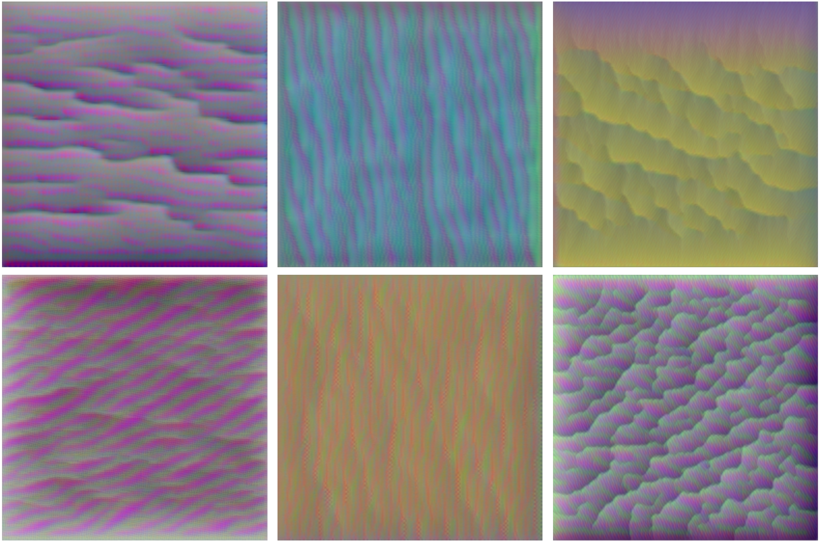
```
model = models.vgg16(weights = 'IMAGENET1K_V1')
model.eval()
edge_energy = MultiScaleEdgeEnergy()
input = torch.empty(1, 3, 224, 224).normal_(0, 0.01)

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')
```

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

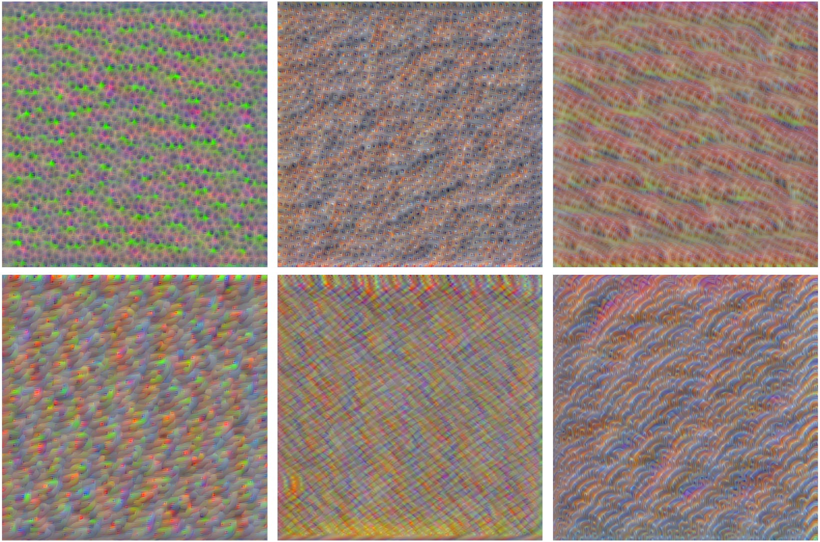
```
model = models.vgg16(weights = 'IMAGENET1K_V1')
model.eval()
edge_energy = MultiScaleEdgeEnergy()
input = torch.empty(1, 3, 224, 224).normal_(0, 0.01)

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)

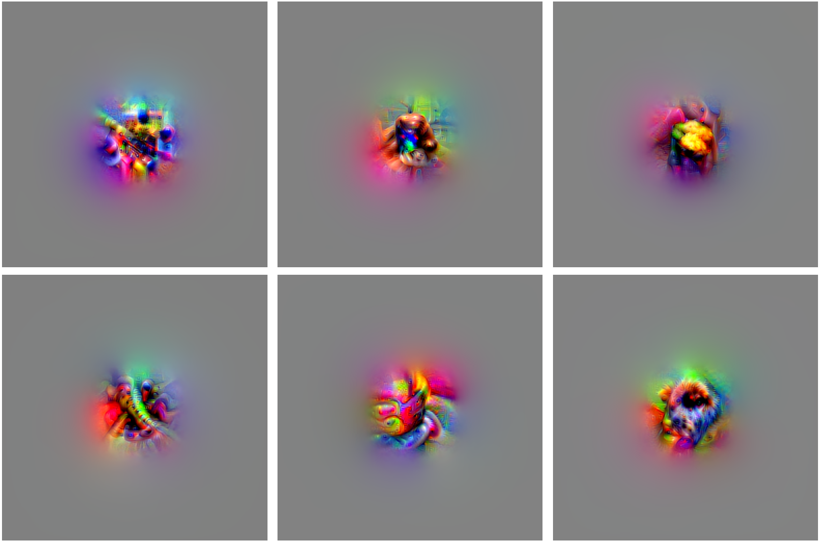
VGG16, maximizing a channel of the 4th convolution layer



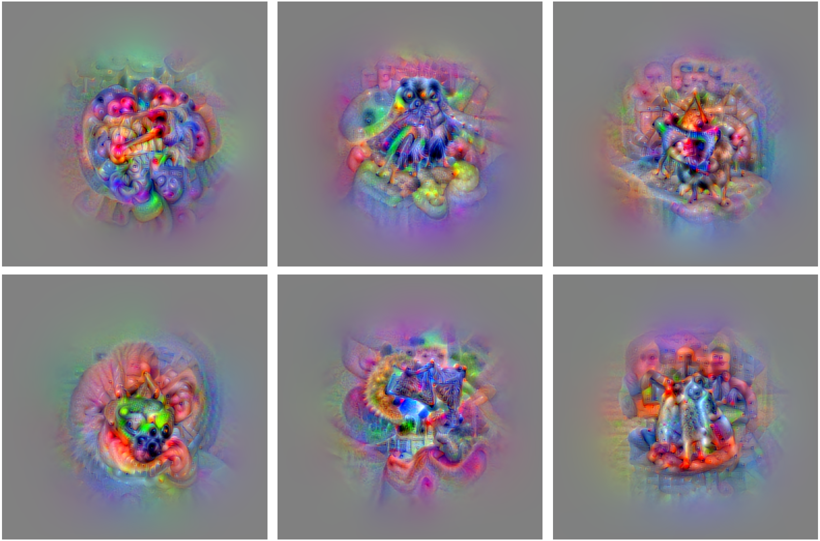
VGG16, maximizing a channel of the 7th convolution layer



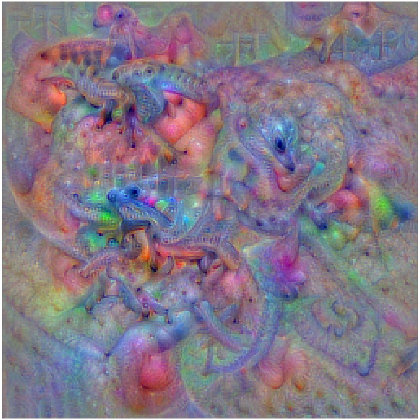
VGG16, maximizing a unit of the 10th convolution layer



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

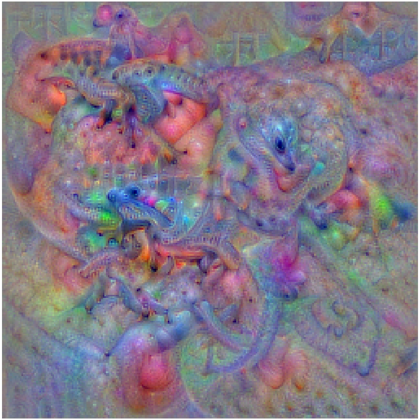


VGG16, maximizing a unit of the output layer



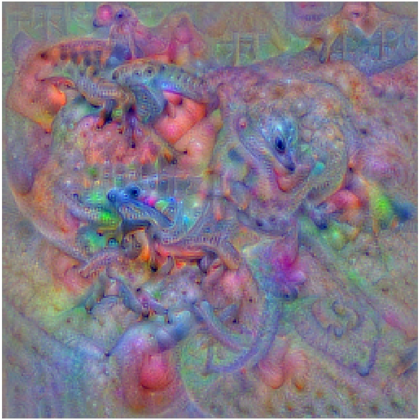
VGG16, maximizing a unit of the output layer

"Box turtle"



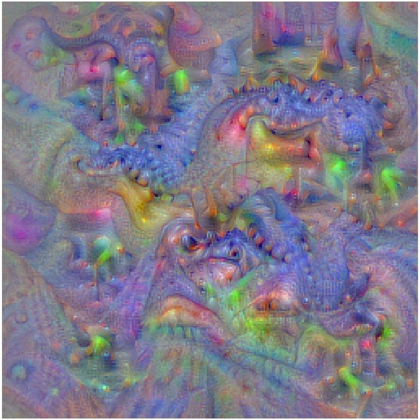
VGG16, maximizing a unit of the output layer

"Box turtle"

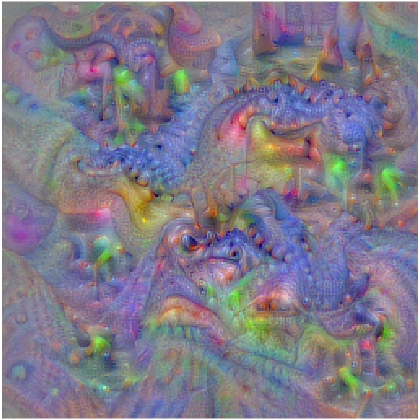


"Whiptail lizard"

VGG16, maximizing a unit of the output layer

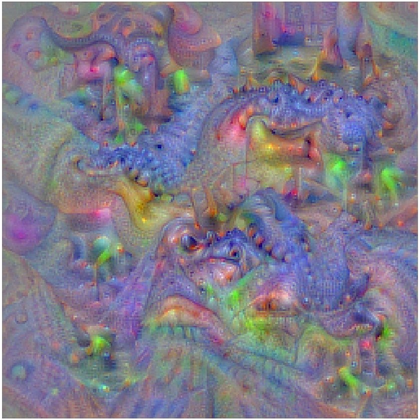


VGG16, maximizing a unit of the output layer



"African chameleon"

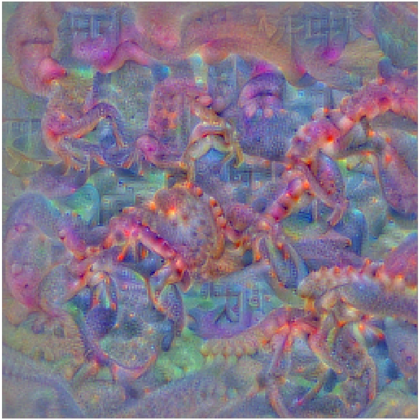
VGG16, maximizing a unit of the output layer

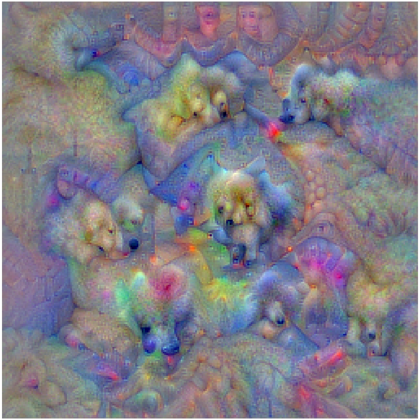


"African chameleon"

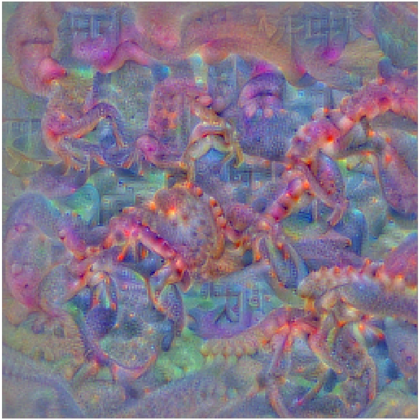
"Wolf spider"

VGG16, maximizing a unit of the output layer

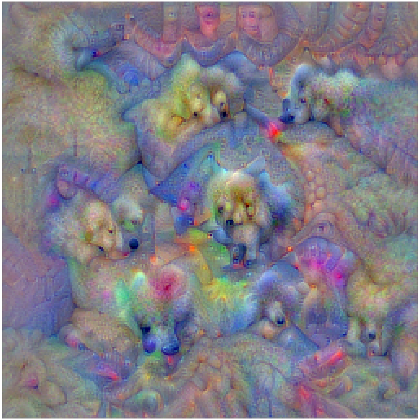




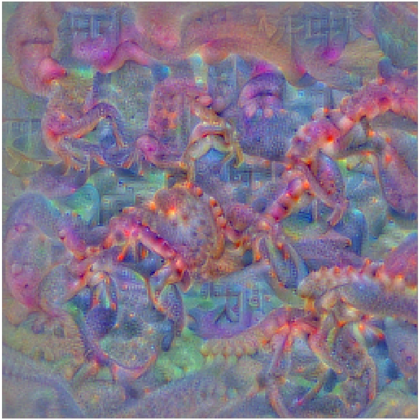
VGG16, maximizing a unit of the output layer



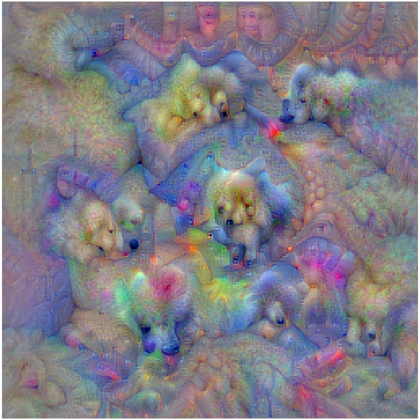
"King crab"



VGG16, maximizing a unit of the output layer

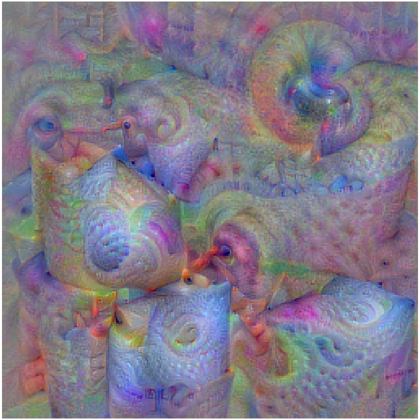


"King crab"



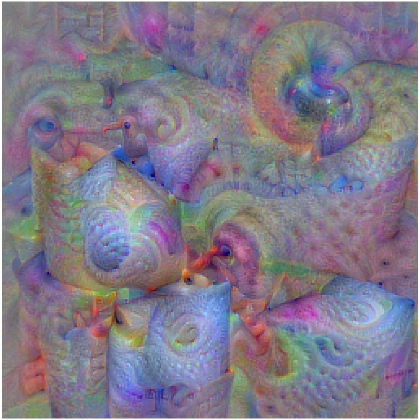
"Samoyed" (that's a fluffy dog)

VGG16, maximizing a unit of the output layer



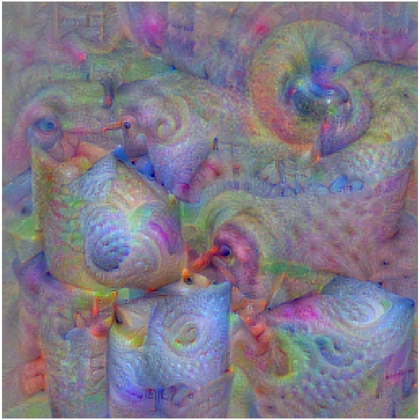
VGG16, maximizing a unit of the output layer

"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

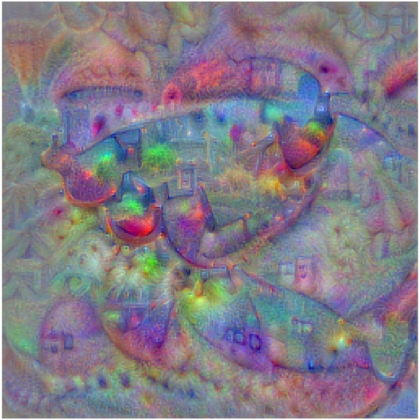
"Ping-pong ball"

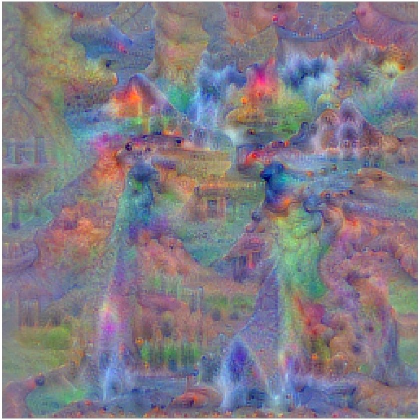
VGG16, maximizing a unit of the output layer

"Ping-pong ball"

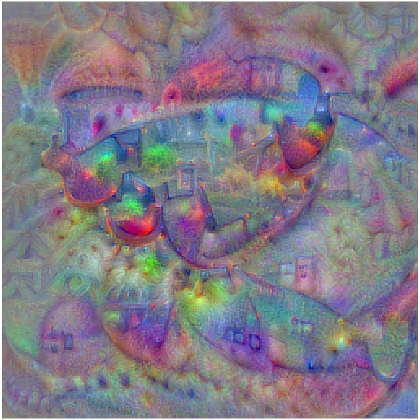
"Steel arch bridge"

VGG16, maximizing a unit of the output layer

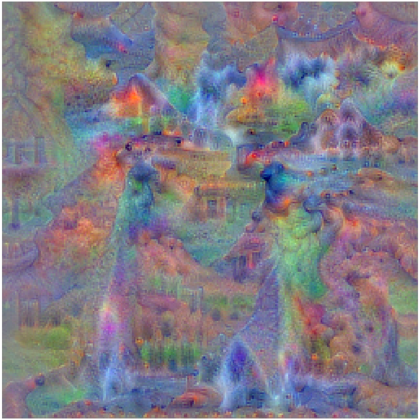




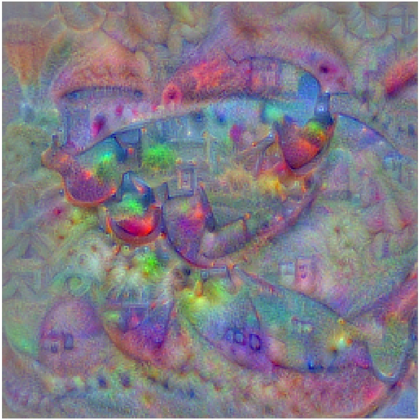
VGG16, maximizing a unit of the output layer



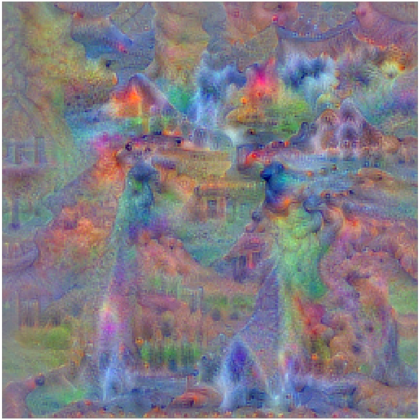
"Sunglass"



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.

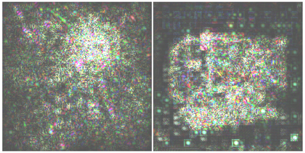
The counter-intuitive result is that the resulting miss-classified images are indistinguishable from the original ones to a human eye.

```
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()
```

Original

Adversarial

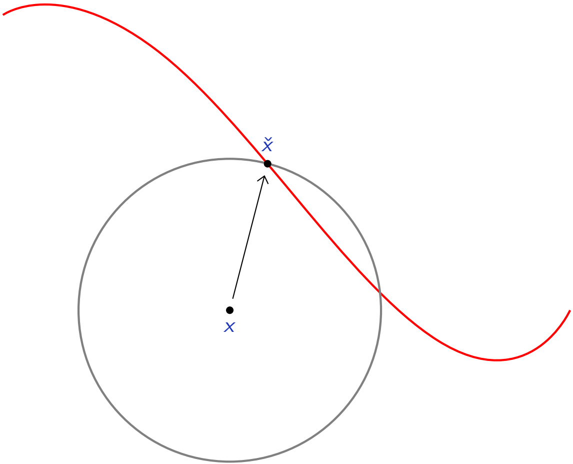
Differences
(magnified)



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

	Predicted classes	
Nb. iterations	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.



The end

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.