CS7643: Deep Learning
Fall 2019
HW2 Solutions

Nicolas Six

October 10, 2019

1 Convolution Basics

1.1 Convolution

If we include padding in X:

$$A^T = \begin{bmatrix}
w_{(0,0)} & 0 & 0 & 0 \\
w_{(0,1)} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & w_{(0,0)} & 0 & 0 \\
0 & w_{(0,1)} & 0 & 0 \\
w_{(1,0)} & 0 & 0 & 0 \\
w_{(1,1)} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & w_{(1,0)} & 0 & 0 \\
0 & w_{(1,1)} & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & w_{(0,0)} & 0 \\
0 & 0 & w_{(0,1)} & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & w_{(0,0)} \\
0 & 0 & 0 & w_{(0,1)} \\
0 & 0 & w_{(1,0)} & 0 \\
0 & 0 & w_{(1,1)} & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & w_{(1,0)} \\
0 & 0 & 0 & w_{(1,1)}
\end{bmatrix}$$

If we include padding don't as mentioned on Piazza, but not clearly on the subject:

$$A = \begin{bmatrix} w_{(1,1)} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & w_{(1,0)} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & w_{(0,1)} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & w_{(1,1)} \end{bmatrix}$$

In both cases, with a results with or without padding:

$$A \cdot X = Y$$

1.2 Transpose convolution

$$A = \begin{bmatrix} w_{(0,0)} & 0 & 0 & 0 \\ w_{(0,1)} & 0 & 0 & 0 \\ 0 & w_{(0,0)} & 0 & 0 \\ 0 & w_{(0,1)} & 0 & 0 \\ w_{(1,0)} & 0 & 0 & 0 \\ w_{(1,1)} & 0 & 0 & 0 \\ 0 & w_{(1,0)} & 0 & 0 \\ 0 & w_{(1,1)} & 0 & 0 \\ 0 & 0 & w_{(0,0)} & 0 \\ 0 & 0 & w_{(0,1)} & 0 \\ 0 & 0 & 0 & w_{(0,0)} \\ 0 & 0 & 0 & w_{(0,1)} \\ 0 & 0 & w_{(1,0)} & 0 \\ 0 & 0 & w_{(1,1)} & 0 \\ 0 & 0 & 0 & w_{(1,0)} \\ 0 & 0 & 0 & w_{(1,1)} \end{bmatrix}$$

1.3 Computation link

1.3.1 Convolution

We are here dealing with 3 dimensional matrix, as such data are hard to represent on paper we are going to divise the operation for each filters. So for a final Y of dimension $(4, 2, 2)$ we would have Y_0, Y_1, Y_2, Y_3 each of dimension $(2, 2)$.
With that in mind we can write Y as:

$$
\begin{aligned}
Y_0 &= w_{(0,0)}X \\
Y_1 &= w_{(0,1)}X \\
Y_2 &= w_{(1,0)}X \\
Y_3 &= w_{(1,1)}X
\end{aligned}
$$

which give a final row major flattened version of Y equal to the following:

$$
Y = \begin{bmatrix}
w_{(0,0)}x_{(0,0)} \\
w_{(0,0)}x_{(0,1)} \\
w_{(0,0)}x_{(1,0)} \\
w_{(0,0)}x_{(1,1)} \\
w_{(0,1)}x_{(0,0)} \\
w_{(0,1)}x_{(0,1)} \\
w_{(0,1)}x_{(1,0)} \\
w_{(0,1)}x_{(1,1)} \\
w_{(1,0)}x_{(0,0)} \\
w_{(1,0)}x_{(0,1)} \\
w_{(1,0)}x_{(1,0)} \\
w_{(1,0)}x_{(1,1)} \\
w_{(1,1)}x_{(0,0)} \\
w_{(1,1)}x_{(0,1)} \\
w_{(1,1)}x_{(1,0)} \\
w_{(1,1)}x_{(1,1)}
\end{bmatrix}
$$

1.3.2 Transpose convolution

Here the computation is very similar to the one given in Subsection 1.2. So we have:

$$A = \begin{bmatrix} w_{(0,0)} & 0 & 0 & 0 \\ w_{(0,1)} & 0 & 0 & 0 \\ 0 & w_{(0,0)} & 0 & 0 \\ 0 & w_{(0,1)} & 0 & 0 \\ w_{(1,0)} & 0 & 0 & 0 \\ w_{(1,1)} & 0 & 0 & 0 \\ 0 & w_{(1,0)} & 0 & 0 \\ 0 & w_{(1,1)} & 0 & 0 \\ 0 & 0 & w_{(0,0)} & 0 \\ 0 & 0 & w_{(0,1)} & 0 \\ 0 & 0 & 0 & w_{(0,0)} \\ 0 & 0 & 0 & w_{(0,1)} \\ 0 & 0 & w_{(1,0)} & 0 \\ 0 & 0 & w_{(1,1)} & 0 \\ 0 & 0 & 0 & w_{(1,0)} \\ 0 & 0 & 0 & w_{(1,1)} \end{bmatrix}$$

$$Y = \begin{bmatrix} x_{(0,0)}w_{(0,0)} \\ x_{(0,0)}w_{(0,1)} \\ x_{(0,1)}w_{(0,0)} \\ x_{(0,1)}w_{(0,1)} \\ x_{(0,0)}w_{(1,0)} \\ x_{(0,0)}w_{(1,1)} \\ x_{(0,1)}w_{(1,0)} \\ x_{(0,1)}w_{(1,1)} \\ x_{(1,0)}w_{(0,0)} \\ x_{(1,0)}w_{(0,1)} \\ x_{(1,1)}w_{(0,0)} \\ x_{(1,1)}w_{(0,1)} \\ x_{(1,0)}w_{(1,0)} \\ x_{(1,0)}w_{(1,1)} \\ x_{(1,1)}w_{(1,0)} \\ x_{(1,1)}w_{(1,1)} \end{bmatrix}$$

So in both case we have a matrix of all the different product possible between the weights and the input, but in a different ordering. Following the definition given in this question, the two operations are then identical.

2 Logic and XOR

2.1 AND and OR

2.1.1 AND

$$W_{AND} = \begin{bmatrix} 1 & 1 \end{bmatrix}$$
$$b_{AND} = -1.5$$

2.1.2 OR

$$W_{OR} = \begin{bmatrix} 1 & 1 \end{bmatrix}$$
$$b_{OR} = -0.5$$

2.2 XOR

Our current model with three different weights value can only represent linear separation of the data. With W representing the slope of the slope of this line in the plan formed by the two variable x and y and the bias b representing the shift regarding to the origin.

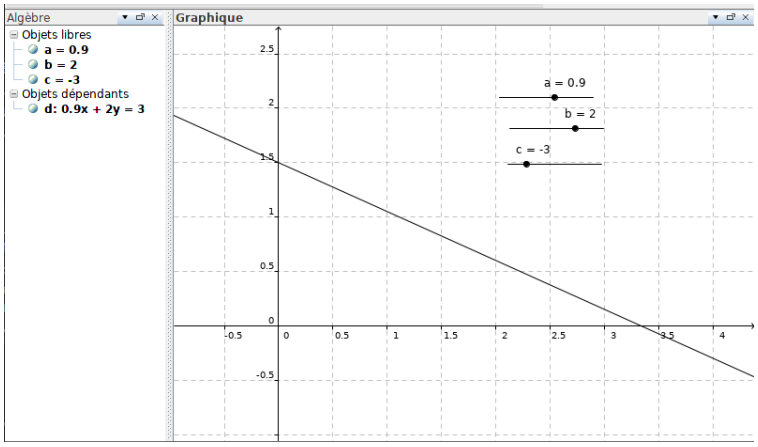


Figure 1: Representation of the the frontier depending on the three weights

But to get an XOR we need to satisfy the following conditions:

$$
\begin{aligned}
f(0,0) &< 0 \\
f(0,1) &\geq 0 \\
f(1,0) &\geq 0 \\
f(1,1) &< 0
\end{aligned}
$$

Which is strictly impossible with only one linear boundary.
As a side not it's important to note that:

$$XOR(x,y) = (\overline{x} \cdot y) + (x \cdot \overline{y})$$

So it's possible to have XOR with linear classifier as we already showed how to get AND and OR operations in subsection 2.1. The NOT operation being just a simple one input classifier with one weight of negative value and a small positive bias.

3 Piecewise linearity

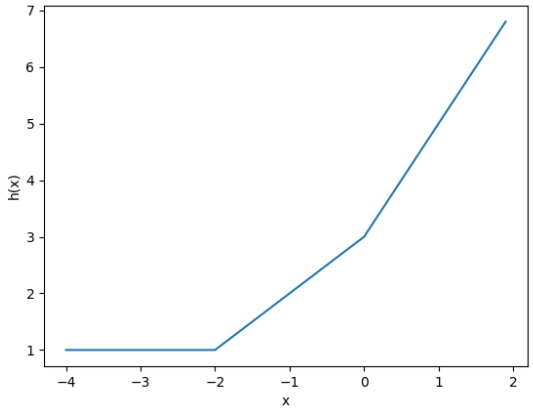


Figure 2: Evolution of the output for different value of the input

3.1 $x = 1$

In the neighborhood of $x = 1$

$$
\begin{aligned}
h(1) &= 5 \\
h(x) &= 2 \cdot x + 3 \\
\frac{\partial h}{\partial x} &= 2
\end{aligned}
$$

3.2 $x = -1$

In the neighborhood of $x = -1$

$$\begin{aligned} h(-1) &= 2 \\ h(x) &= 1 \cdot x + 3 \\ \frac{\partial h}{\partial x} &= 1 \end{aligned}$$

3.3 $x = -0.5$

In the neighborhood of $x = -0.5$

$$\begin{aligned} h(-0.5) &= 5 \\ h(x) &= 1 \cdot x + 3 \\ \frac{\partial h}{\partial x} &= 1 \end{aligned}$$

4 Depth - Composing Linear Pieces

4.1 Basic region split

According to the definition we have:

$$W^{(1)} = 2 \cdot I$$

Which let us simplify the expression of f_1:

$$\begin{aligned} f_1(x) &= |W^{(1)}x + b| \\ &= |2 \cdot I \cdot x + b| \\ &= |2x + b| \end{aligned}$$

In other word the transformation is the same on every dimension and is the function $|2x - 1|$. This function get to zero for $x = 0.5$ and as value of 1 in 0 and 1 and is linear between those points.
In conclusion there is 2^d input region on this interval.

$$R = \left\{ [0, 0.5]^d, [0.5, 1]^d \right\}$$

4.2 Regions and composition

If we only consider the path that seams to be wanted by the question, x in $]0,1[^d$, $g(x)$ in $]0,1[^d$ and $f \circ g(x)$ in $]0,1[^d$. Then it easily appear that each identified regions by g would be split into n_f regions. Leading to the fact that $f \circ g(x)$ identify onto $]0,1[^d$ $n_g \cdot n_f$ regions. Which look like the expected answer.
It's important to note that this implication is only guarantee if each of the n_g regions of g identify to $]0,1[^d$ and no one of its subset like $]0,0.1[^d$. Such behaviour would lead to some input regions of f not being hit, and so, some output regions may be missed, leading us to the fact that under this current set of constraint on f and g and the current definition of identification, the $n_g \cdot n_f$ number of regions is an upper bound of the final results.

In addition, counting the number of regions on which $f \circ g(\cdot)$ identify onto $]0,1[^d$, doesn't imply any assumptions on the intermediate results of function g. For example function g may identify n_{g2} regions of $]0,1[^d$ onto $]1,2[^d$ and f n_{f2} regions of $]1,2[^d$ onto $]0,1[^d$. This would increase the final results by $n_{g2} \cdot n_{f2}$.
We could apply such a reasoning on every none overlapping intervals where f inputs and g outputs are defined. However such definition of f and g are not given here, so we can't assume that such path doesn't exists. This show that the final result may be higher than the previously given $n_g \cdot n_f$.

In conclusion as stated previously, the answer to this question is the product of the regions of each sub function, in this case $n_g \cdot n_f$. But this imply a set large number of assumption that are not defined, and so may not hold true. Depending on the assumptions you take this number can be a lower bound or a upper bound, showing that we can't get a precise number for the general case.

4.3 Regions on a multi layers network

Each layers having an absolute value activation function, they have one discontinuity leading to only two possible identifications per dimensions of the output. So the final output of a given layer identify onto 2^d regions of it's input (d split in two of the original input region).
As shown in the previous question, staking layers (i.e. combining functions) lead to the multiplication of the regions. Here all layers behave in the same way and identify onto 2^d regions. Staking all of this layers together give us the expected 2^{Ld} regions results.

However, we are here also forgetting about a large set of assumptions. Due to the limited input space, and the large set of value possible for the weights and bias this result may not hold true.
On way to see that is to pick the weights such as:

$$W_1 = W_2 = W_3 = \cdots = W_L = I$$

And the bias such as:

$$b_1 = b_2 = b_3 = \cdots = b_L = \vec{0}$$

Using this we quickly see that:

$$f(x) = |x|$$

and as every every value of x are bigger than 0 ($x \in]0, 1[^d$):

$$f(x) = x$$

Which in this special case clearly give us that $f(x)$ identify to 1 region of it's input. However in this special case we didn't break any assumption given in the subject, showing that $f(x)$ may not identifies to 2^{Ld} regions of it's input.

In conclusion, The general results of this question as showed above is clearly 2^{Ld}. But, as demonstrated in the example above, this results is get by a large set of underlying assumption that are never given and may not hold true in real cases (even if it will on most of the case).

5 Conclusion to Theory Part

6 Coding: Uses of Gradients With Respect to Input

6.1 Gradient on images for visualisation

NetworkVisualization-PyTorch

October 7, 2019

1 Network Visualization (30 Points)

In this notebook we will explore the use of *image gradients* for generating new images, by studying and implementing key components in three papers:

- Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps", ICLR Workshop 2014.
- Szegedy et al, "Intriguing properties of neural networks", ICLR 2014
- Yosinski et al, "Understanding Neural Networks Through Deep Visualization", ICML 2015 Deep Learning Workshop

You will need to first read the paper, and then we will guide you to understand them deeper with some problems.

When training a model, we define a loss function which measures our current unhappiness with the model's performance; we then use backpropagation to compute the gradient of the loss with respect to the model parameters, and perform gradient descent on the model parameters to minimize the loss.

In this homework, we will do something slightly different. We will start from a convolutional neural network model which has been pretrained to perform image classification on the ImageNet dataset. We will use this model to define a loss function which quantifies our current unhappiness with our image, then use backpropagation to compute the gradient of this loss with respect to the pixels of the image. We will then keep the model fixed, and perform gradient descent *on the image* to synthesize a new image which minimizes the loss.

This notebook is the first part of homwwork 2. We will explore three techniques for image generation:

1. **Saliency Maps**: Saliency maps are a quick way to tell which part of the image influenced the classification decision made by the network.
2. **Fooling Images**: We can perturb an input image so that it appears the same to humans, but will be misclassified by the pretrained network.
3. **Class Visualization**: We can synthesize an image to maximize the classification score of a particular class; this can give us some sense of what the network is looking for when it classifies images of that class.

We will use **PyTorch 1.1** to finish the problems in this notebook, which has been tested with Python3.6 on Linux and Mac.

Suppose you have already installed the dependencies in the last homework. **Before you start this one, here are some preparation work you need to do**:

- Download the imagenet_val_25 dataset

```
cd cs7643/datasets
bash get_imagenet_val.sh
```

** Note for grading**:

- The total credits for this notebook are 30 points, which are equally distributed in the three problems.
- Although we will run your notebook in grading, but you still need to **submit the notebook with all the outputs you generated**. Sometimes it will inform us if we get any inconsitent results with respect to yours.

```
In [1]: import torch
        from torch.autograd import Variable
        import torchvision
        import torchvision.transforms as T
        import random

        import numpy as np
        from scipy.ndimage.filters import gaussian_filter1d
        import matplotlib.pyplot as plt
        from cs7643.image_utils import SQUEEZENET_MEAN, SQUEEZENET_STD
        from PIL import Image

        %matplotlib inline
        plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
        plt.rcParams['image.interpolation'] = 'nearest'
        plt.rcParams['image.cmap'] = 'gray'
```

1.0.1 Helper Functions

Our pretrained model was trained on images that had been preprocessed by subtracting the per-color mean and dividing by the per-color standard deviation. We define a few helper functions for performing and undoing this preprocessing.

You don't need to do anything in this cell. Just run it.

```
In [2]: def preprocess(img, size=224):
            transform = T.Compose([
                T.Resize(size),
                T.ToTensor(),
                T.Normalize(mean=SQUEEZENET_MEAN.tolist(),
                            std=SQUEEZENET_STD.tolist()),
                T.Lambda(lambda x: x[None]),
            ])
            return transform(img)

        def deprocess(img, should_rescale=True):
```

```
        transform = T.Compose([
            T.Lambda(lambda x: x[0]),
            T.Normalize(mean=[0, 0, 0], std=(1.0 / SQUEEZENET_STD).tolist()),
            T.Normalize(mean=(-SQUEEZENET_MEAN).tolist(), std=[1, 1, 1]),
            T.Lambda(rescale) if should_rescale else T.Lambda(lambda x: x),
            T.ToPILImage(),
        ])
        return transform(img)

    def rescale(x):
        low, high = x.min(), x.max()
        x_rescaled = (x - low) / (high - low)
        return x_rescaled

    def blur_image(X, sigma=1):
        X_np = X.cpu().clone().numpy()
        X_np = gaussian_filter1d(X_np, sigma, axis=2)
        X_np = gaussian_filter1d(X_np, sigma, axis=3)
        X.copy_(torch.Tensor(X_np).type_as(X))
        return X
```

2 Pretrained Model

For all of our image generation experiments, we will start with a convolutional neural network which was pretrained to perform image classification on ImageNet. We can use any model here, but for the purposes of this assignment we will use SqueezeNet, which achieves accuracies comparable to AlexNet but with a significantly reduced parameter count and computational complexity.

Using SqueezeNet rather than AlexNet or VGG or ResNet means that we can easily perform all the experiments in this notebook on a CPU machine. You are encouraged to use a larger model to finish the rest of the experiments if GPU resouces are not a problem for you, but please highlight the backbone network you use in your implementation if you do it.

Switching a backbone network is quite easy in pytorch. You can refer to torchvision model zoos for more information.

- Iandola et al, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and < 0.5MB model size", arXiv 2016

In [3]:
```
# Download and load the pretrained SqueezeNet model.
model = torchvision.models.squeezenet1_1(pretrained=True)

# We don't want to train the model, so tell PyTorch not to compute gradients
# with respect to model parameters.
for param in model.parameters():
    param.requires_grad = False
```

```
Downloading: "https://download.pytorch.org/models/squeezenet1_1-f364aa15.pth" to /home/nicolas/
100%|| 4.74M/4.74M [00:00<00:00, 44.7MB/s]
```

2.1 Load some ImageNet images

If you have not execute the downloading script. Here is a reminder that you have to do it now. We have provided a few example images from the validation set of the ImageNet ILSVRC 2012 Classification dataset.

To download these images and run

```
cd cs7643/datasets/
bash get_imagenet_val.sh
```

Since they come from the validation set, our pretrained model did not see these images during training.

Run the following cell to visualize some of these images, along with their ground-truth labels.

```
In [4]: from cs7643.data_utils import load_imagenet_val
        X, y, class_names = load_imagenet_val(num=5)

        plt.figure(figsize=(12, 6))
        for i in range(5):
            plt.subplot(1, 5, i + 1)
            plt.imshow(X[i])
            plt.title(class_names[y[i]])
            plt.axis('off')
        plt.gcf().tight_layout()
```

3 Saliency Maps (10 pts)

Using this pretrained model, we will compute class saliency maps as described in the paper:

[1] [Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps", ICLR Workshop 2014.](https://arxiv.org/abs/1312.6034)

We will also review this paper in the paper presentation.

A **saliency map** tells us the degree to which each pixel in the image affects the classification score for that image. To compute it, we compute the gradient of the unnormalized score corresponding to the correct class (which is a scalar) with respect to the pixels of the image. If the image has shape `(3, H, W)` then this gradient will also have shape `(3, H, W)`; for each pixel in the image, this gradient tells us the amount by which the classification score will change if the pixel changes by a small amount. To compute the saliency map, we take the absolute value of this gradient, then take the maximum value over the 3 input channels; the final saliency map thus has shape `(H, W)` and all entries are nonnegative.

3.0.1 Hint: PyTorch `gather` method

Recall when you need to select one element from each row of a matrix; if `s` is an numpy array of shape `(N, C)` and `y` is a numpy array of shape `(N,)` containing integers `0 <= y[i] < C`, then `s[np.arange(N), y]` is a numpy array of shape `(N,)` which selects one element from each element in `s` using the indices in `y`.

In PyTorch you can perform the same operation using the `gather()` method. If `s` is a PyTorch Tensor or Variable of shape `(N, C)` and `y` is a PyTorch Tensor or Variable of shape `(N,)` containing longs in the range `0 <= y[i] < C`, then

`s.gather(1, y.view(-1, 1)).squeeze()`

will be a PyTorch Tensor (or Variable) of shape `(N,)` containing one entry from each row of `s`, selected according to the indices in `y`.

run the following cell to see an example.

You can also read the documentation for the gather method and the squeeze method.

```
In [5]: # Example of using gather to select one entry from each row in PyTorch
        def gather_example():
            N, C = 4, 5
            s = torch.randn(N, C)
            y = torch.LongTensor([1, 2, 1, 3])
            print(s)
            print(y)
            print(s.gather(1, y.view(-1, 1)).squeeze())
        gather_example()
```

```
tensor([[-0.0333, -1.4277, -2.5025, -1.5945, -0.5335],
        [ 0.5817,  0.8416, -0.0904, -1.0548,  0.5112],
        [ 1.5504,  0.3712,  0.9924,  0.6339,  1.1652],
        [ 0.7294, -2.0734,  1.9746,  0.4835,  0.1123]])
tensor([1, 2, 1, 3])
tensor([-1.4277, -0.0904,  0.3712,  0.4835])
```

```
In [16]: def compute_saliency_maps(X, y, model):
             """
             Compute a class saliency map using the model for images X and labels y.

             Input:
             - X: Input images; Tensor of shape (N, 3, H, W)
             - y: Labels for X; LongTensor of shape (N,)
             - model: A pretrained CNN that will be used to compute the saliency map.

             Returns:
             - saliency: A Tensor of shape (N, H, W) giving the saliency maps for the input
             images.
             """
             # Make sure the model is in "test" mode
             model.eval()
```

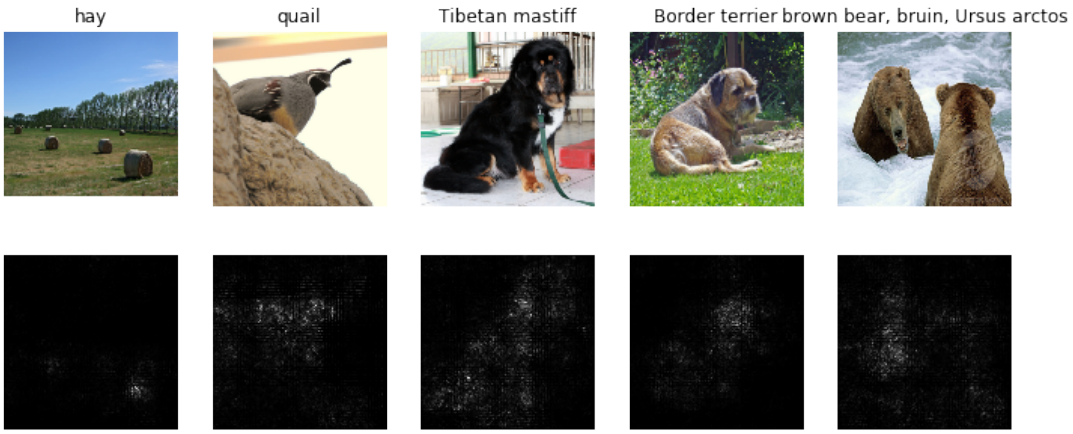
```
    # Wrap the input tensors in Variables
    X_var = Variable(X, requires_grad=True)
    y_var = Variable(y, requires_grad=False)
    saliency = None

    lam = 1e3 # This is the regularization parameter when you need it

    ##############################################################################
    # TODO: Implement this function. Perform a forward and backward pass through #
    # the model to compute the gradient of the correct class score with respect  #
    # to each input image. You first want to compute the loss over the correct   #
    # scores, and then compute the gradients with a backward pass.               #
    ##############################################################################
    outputs = model(X_var)
    criterion = torch.nn.functional.nll_loss
    loss = criterion(outputs, y_var)
    loss.backward()
    saliency = X_var.grad
    saliency=torch.max(saliency.abs(), 1, keepdim=True)[0].data
    saliency=saliency.reshape((-1,224,224))
    ##############################################################################
    #                               END OF YOUR CODE                             #
    ##############################################################################
    return saliency
```

Once you have completed the implementation in the cell above, run the following to visualize some class saliency maps on our example images from the ImageNet validation set. You can compare to the figure 2 in the refered paper as a comparison for your results.

```
In [17]: def show_saliency_maps(X, y):
    # Convert X and y from numpy arrays to Torch Tensors
    X_tensor = torch.cat([preprocess(Image.fromarray(x)) for x in X], dim=0)
    y_tensor = torch.LongTensor(y)

    # Compute saliency maps for images in X
    saliency = compute_saliency_maps(X_tensor, y_tensor, model)
    # Convert the saliency map from Torch Tensor to numpy array and show images
    # and saliency maps together.
    saliency = saliency.numpy()

    N = X.shape[0]
    for i in range(N):
        plt.subplot(2, N, i + 1)
        plt.imshow(X[i])
        plt.axis('off')
        plt.title(class_names[y[i]])
        plt.subplot(2, N, N + i + 1)
        plt.imshow(saliency[i], cmap=plt.cm.gray)
```

```
            plt.axis('off')
            plt.gcf().set_size_inches(12, 5)
        plt.show()

    show_saliency_maps(X, y)
```

```
torch.Size([5, 3, 224, 224])
torch.Size([5, 224, 224])
```



4 Fooling Images (10 pts)

We can also use the similar concept of image gradients to study the stability of the network. Consider a state-of-the-art deep neural network that generalizes well on an object recognition task. We expect such network to be robust to small perturbations of its input, because small perturbation cannot change the object category of an image. However, [2] find that applying an imperceptible non-random perturbation to a test image, it is possible to arbitrarily change the network's prediction.

[2] [Szegedy et al, "Intriguing properties of neural networks", ICLR 2014](https://arxiv.org/abs/1312.6199)

Given an image and a target class, we can perform **gradient ascent** over the image to maximize the target class, stopping when the network classifies the image as the target class. We term the so perturbed examples "adversarial examples".

Read the paper, and then implement the following function to generate fooling images.

```
In [25]: def make_fooling_image(X, target_y, model):
             """
             Generate a fooling image that is close to X, but that the model classifies
             as target_y.

             Inputs:
```

```
- X: Input image; Tensor of shape (1, 3, 224, 224)
- target_y: An integer in the range [0, 1000)
- model: A pretrained CNN

Returns:
- X_fooling: An image that is close to X, but that is classifed as target_y
by the model.
"""

model.eval()

# Initialize our fooling image to the input image, and wrap it in a Variable.
X_fooling = X.clone()
X_fooling_var = Variable(X_fooling, requires_grad=True)

# We will fix these parameters for everyone so that there will be
# comparable outputs

learning_rate = 10 # learning rate is 1
max_iter = 100 # maximum number of iterations

for it in range(max_iter):
    ##############################################################################
    # TODO: Generate a fooling image X_fooling that the model will classify as   #
    # the class target_y. You should perform gradient ascent on the score of the #
    # target class, stopping when the model is fooled.                           #
    # When computing an update step, first normalize the gradient:               #
    #   dX = learning_rate * g / ||g||_2                                         #
    #                                                                            #
    # Inside of this loop, write the update rule.                                #
    #                                                                            #
    # HINT:                                                                      #
    # You can print your progress (current prediction and its confidence score)  #
    # over iterations to check your gradient ascent progress.                    #
    ##############################################################################
    model.zero_grad()
    outputs = model(X_fooling_var)
    _, predict = torch.max(outputs, 1)
    if (predict.data[0] == target_y):
        break
    target = outputs[0,target_y]
    print("Current target confidence: {}".format(target))

    target.backward()

    dx = X_fooling_var.grad
    dx /= dx.norm()
    X_fooling_var.data.add_(dx.data * learning_rate )
```

```
            ##############################################################################
            #                                END OF YOUR CODE                            #
            ##############################################################################

        X_fooling = X_fooling_var.data

        return X_fooling
```

Now you can run the following cell to **generate a fooling image**. You will see the message 'Fooled the model' when you succeed.

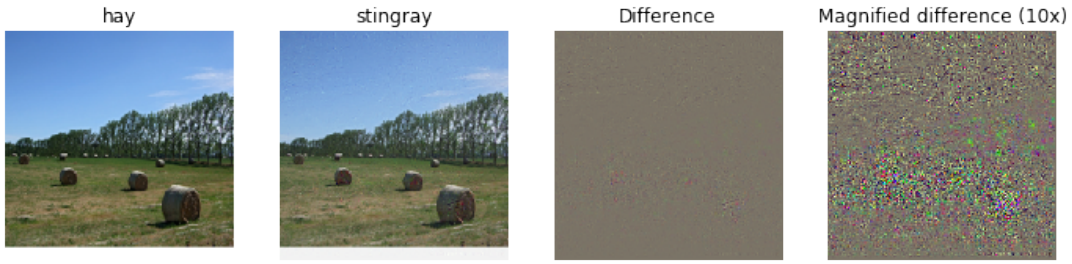
```
In [26]: idx = 0
         target_y = 6 # target label. Change to a different label to see the difference.

         X_tensor = torch.cat([preprocess(Image.fromarray(x)) for x in X], dim=0)
         X_fooling = make_fooling_image(X_tensor[idx:idx+1], target_y, model)

         scores = model(Variable(X_fooling))

         if target_y == scores.data.max(1)[1][0]:
             print('Fooled the model!')
         else:
             print('The model is not fooled!')
```

```
Current target confidence: 5.213543891906738
Current target confidence: 12.341626167297363
Current target confidence: 22.990808486938477
Fooled the model!
```

After generating a fooling image, run the following cell to visualize the original image, the fooling image, as well as the difference between them.

```
In [27]: X_fooling_np = deprocess(X_fooling.clone())
         X_fooling_np = np.asarray(X_fooling_np).astype(np.uint8)

         plt.subplot(1, 4, 1)
         plt.imshow(X[idx])
         plt.title(class_names[y[idx]])
         plt.axis('off')

         plt.subplot(1, 4, 2)
         plt.imshow(X_fooling_np)
         plt.title(class_names[target_y])
         plt.axis('off')

         plt.subplot(1, 4, 3)
         X_pre = preprocess(Image.fromarray(X[idx]))
         diff = np.asarray(deprocess(X_fooling - X_pre, should_rescale=False))
```

```
plt.imshow(diff)
plt.title('Difference')
plt.axis('off')

plt.subplot(1, 4, 4)
diff = np.asarray(deprocess(10 * (X_fooling - X_pre), should_rescale=False))
plt.imshow(diff)
plt.title('Magnified difference (10x)')
plt.axis('off')

plt.gcf().set_size_inches(12, 5)
plt.show()
```



5 Class visualization (10 pts)

By starting with a random noise image and performing gradient ascent on a target class, we can generate an image that the network will recognize as the target class. This idea was first presented in [1]; [3] extended this idea by suggesting several regularization techniques that can improve the quality of the generated image.

Concretely, let I be an image and let y be a target class. Let $s_y(I)$ be the score that a convolutional network assigns to the image I for class y; note that these are raw unnormalized scores, not class probabilities. We wish to generate an image I^* that achieves a high score for the class y by solving the problem

$$I^* = \arg\max_I s_y(I) - R(I)$$

where R is a (possibly implicit) regularizer (note the sign of $R(I)$ in the argmax: we want to minimize this regularization term). We can solve this optimization problem using gradient ascent, computing gradients with respect to the generated image. We will use (explicit) L2 regularization of the form

$$R(I) = \lambda \|I\|_2^2$$

and implicit regularization as suggested by [3] by periodically blurring the generated image. We can solve this problem using gradient ascent on the generated image.

[1] [Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps", ICLR Workshop 2014](https://arxiv.org/abs/1312.6034)

[3] [Yosinski et al, "Understanding Neural Networks Through Deep Visualization", ICML 2015 Deep Learning Workshop](http://yosinski.com/media/papers/Yosinski__2015__ICML_DL__Understanding_N

In the cell below, complete the implementation of the `create_class_visualization` function.

In [28]:
```
def jitter(X, ox, oy):
    """
    Helper function to randomly jitter an image.

    Inputs
    - X: PyTorch Tensor of shape (N, C, H, W)
    - ox, oy: Integers giving number of pixels to jitter along W and H axes

    Returns: A new PyTorch Tensor of shape (N, C, H, W)
    """
    if ox != 0:
        left = X[:, :, :, :-ox]
        right = X[:, :, :, -ox:]
        X = torch.cat([right, left], dim=3)
    if oy != 0:
        top = X[:, :, :-oy]
        bottom = X[:, :, -oy:]
        X = torch.cat([bottom, top], dim=2)
    return X
```

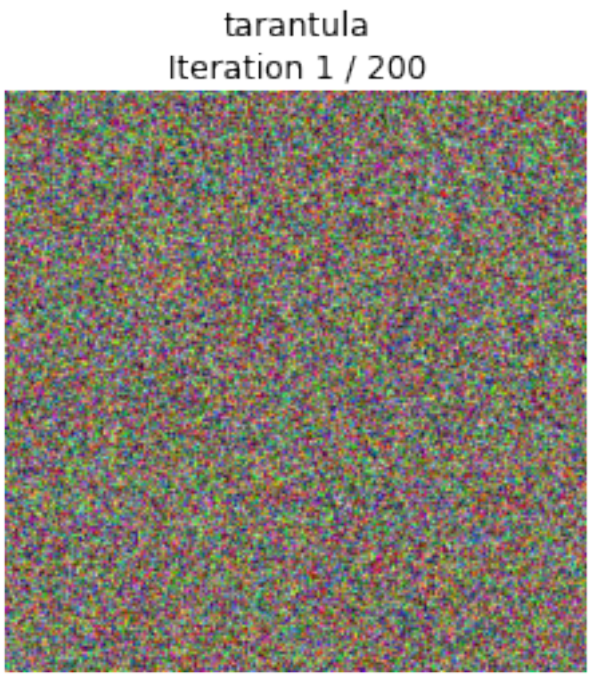
In [34]:
```
def create_class_visualization(target_y, model, dtype, **kwargs):
    """
    Generate an image to maximize the score of target_y under a pretrained model.

    Inputs:
    - target_y: Integer in the range [0, 1000) giving the index of the class
    - model: A pretrained CNN that will be used to generate the image
    - dtype: Torch datatype to use for computations

    Keyword arguments:
    - l2_reg: Strength of L2 regularization on the image
    - learning_rate: How big of a step to take
    - num_iterations: How many iterations to use
    - blur_every: How often to blur the image as an implicit regularizer
    - max_jitter: How much to gjitter the image as an implicit regularizer
    - show_every: How often to show the intermediate result
    """

    model.eval()
```

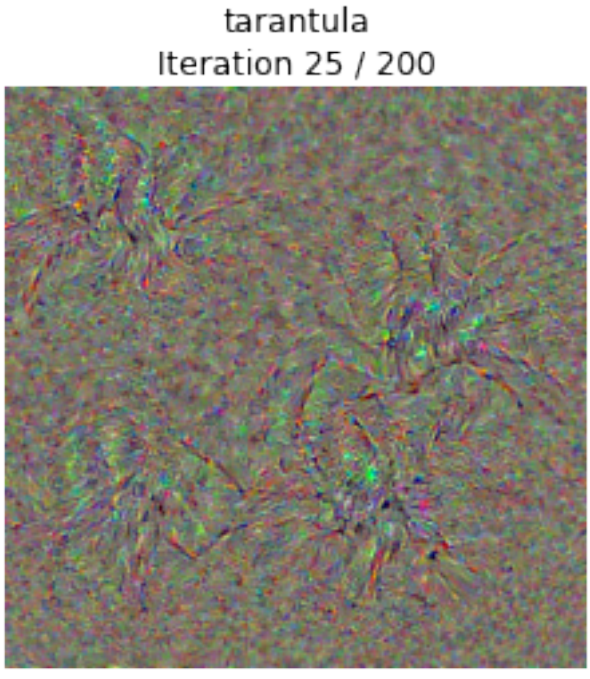
```
    model.type(dtype)
    l2_reg = kwargs.pop('l2_reg', 1e-3)
    learning_rate = kwargs.pop('learning_rate', 25)
    num_iterations = kwargs.pop('num_iterations', 100)
    blur_every = kwargs.pop('blur_every', 10)
    max_jitter = kwargs.pop('max_jitter', 16)
    show_every = kwargs.pop('show_every', 25)

    # Randomly initialize the image as a PyTorch Tensor, and also wrap it in
    # a PyTorch Variable.
    img = torch.randn(1, 3, 224, 224).mul_(1.0).type(dtype)
    img_var = Variable(img, requires_grad=True)

    for t in range(num_iterations):
        # Randomly jitter the image a bit; this gives slightly nicer results
        ox, oy = random.randint(0, max_jitter), random.randint(0, max_jitter)
        img.copy_(jitter(img, ox, oy))

        ########################################################################
        # TODO: Use the model to compute the gradient of the score for the     #
        # class target_y with respect to the pixels of the image, and make a   #
        # gradient step on the image using the learning rate. Don't forget the #
        # L2 regularization term!                                              #
        # Be very careful about the signs of elements in your code.            #
        ########################################################################
        model.zero_grad()
        outputs = model(img_var)
        target = outputs[0,target_y]
#         print("Current target confidence: {}".format(target))

        target.backward()

        dx = img_var.grad - 2 * l2_reg * img_var
        dx /= dx.norm()
        img_var.data.add_(dx.data * learning_rate)
        img_var.grad.zero_()
        ########################################################################
        #                             END OF YOUR CODE                         #
        ########################################################################

        # Undo the random jitter
        img.copy_(jitter(img, -ox, -oy))

        # As regularizer, clamp and periodically blur the image
        for c in range(3):
            lo = float(-SQUEEZENET_MEAN[c] / SQUEEZENET_STD[c])
            hi = float((1.0 - SQUEEZENET_MEAN[c]) / SQUEEZENET_STD[c])
            img[:, c].clamp_(min=lo, max=hi)
```

```
            if t % blur_every == 0:
                blur_image(img, sigma=0.5)

            # Periodically show the image
            if t == 0 or (t + 1) % show_every == 0 or t == num_iterations - 1:
                plt.imshow(deprocess(img.clone().cpu()))
                class_name = class_names[target_y]
                plt.title('%s\nIteration %d / %d' % (class_name, t + 1, num_iterations))
                plt.gcf().set_size_inches(4, 4)
                plt.axis('off')
                plt.show()

        return deprocess(img.cpu())
```

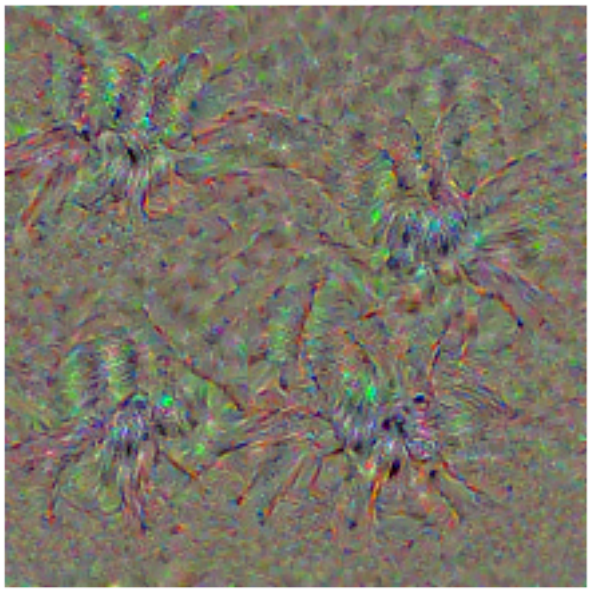
Once you have completed the implementation in the cell above, run the following cell to generate images of several classes. Show the generated images when you submitted your notebook.

In [35]:
```
dtype = torch.FloatTensor
# dtype = torch.cuda.FloatTensor # Uncomment this to use GPU
model.type(dtype)

# You can use a single class during your debugging session,
# but please show all the generated outputs in your submitted notebook

# target_y = 76 # Tarantula
# target_y = 78 # Tick
# target_y = 187 # Yorkshire Terrier
# target_y = 683 # Oboe
# target_y = 366 # Gorilla
# target_y = 604 # Hourglass

targets = [76, 78, 187, 683, 366, 604]

for target in targets:
    out = create_class_visualization(target, model, dtype, num_iterations=200)
```

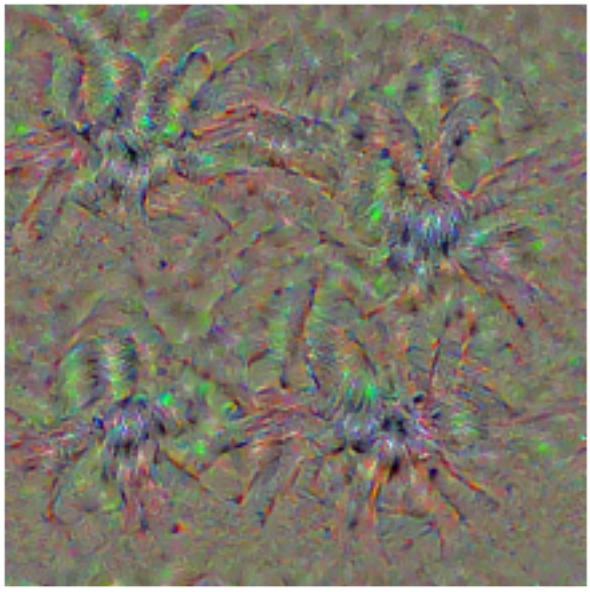

tarantula
Iteration 1 / 200


tarantula
Iteration 25 / 200

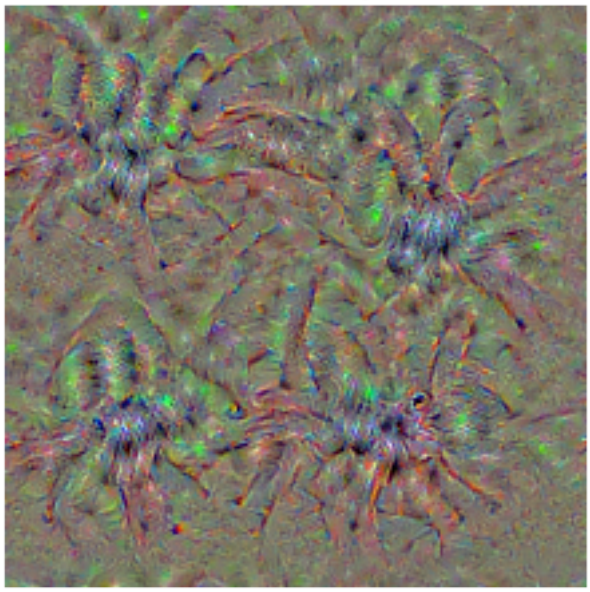
tarantula
Iteration 50 / 200



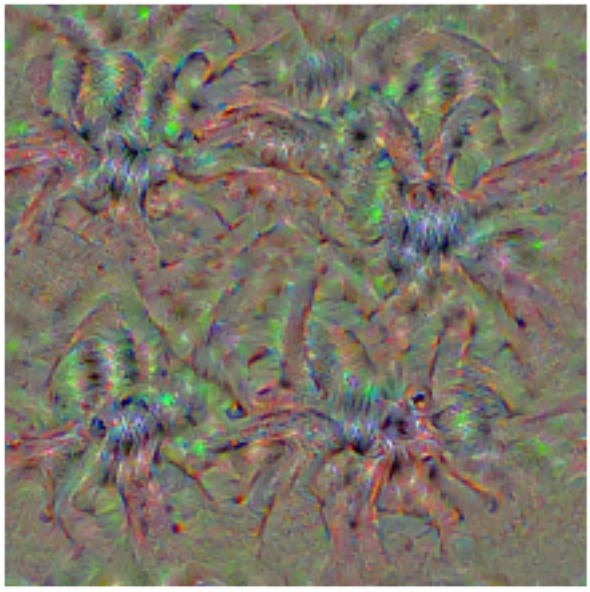
tarantula
Iteration 75 / 200



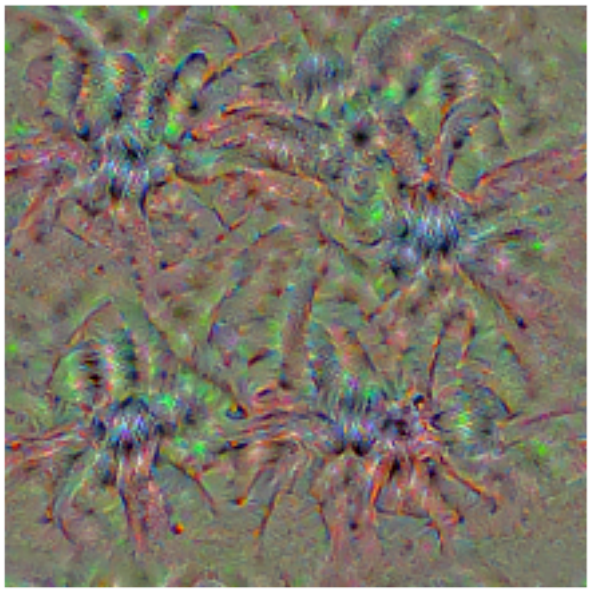
tarantula
Iteration 100 / 200



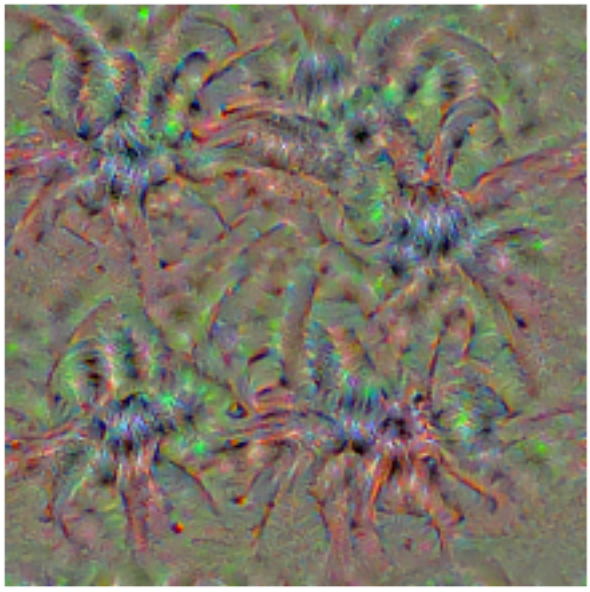
tarantula
Iteration 125 / 200



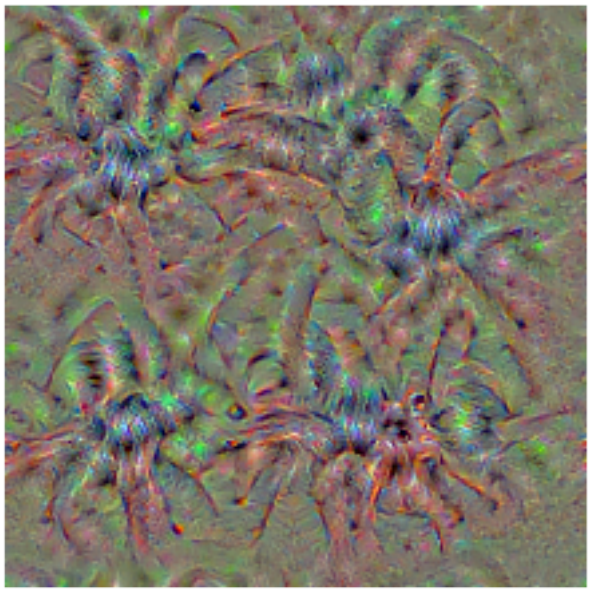
tarantula
Iteration 150 / 200



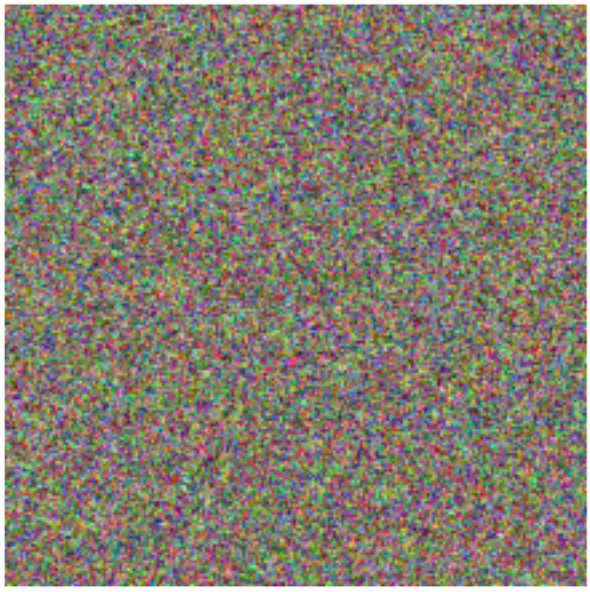
tarantula
Iteration 175 / 200



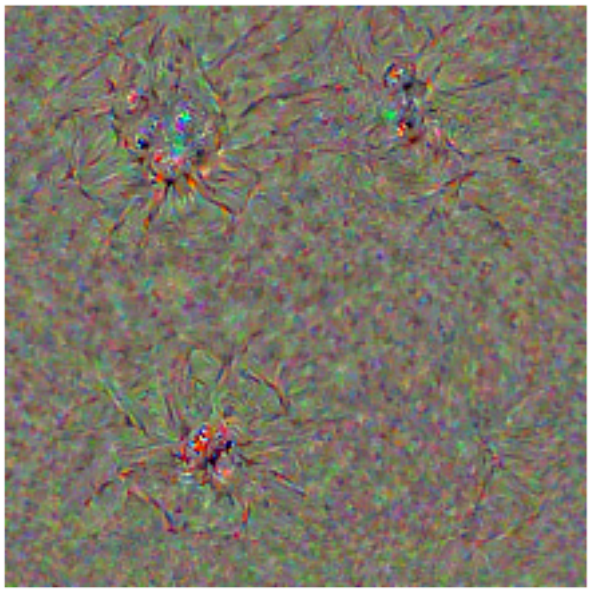
tarantula
Iteration 200 / 200



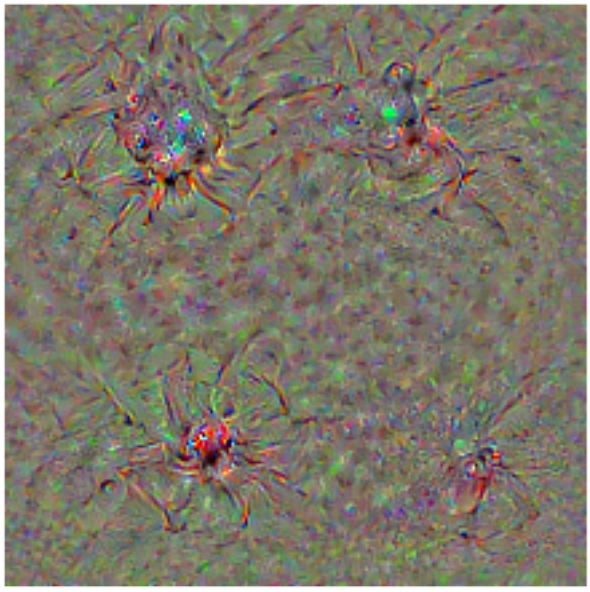
tick
Iteration 1 / 200



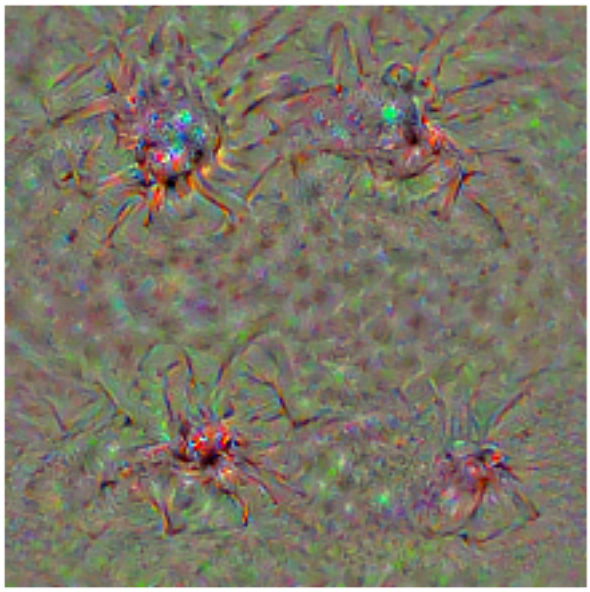
tick
Iteration 25 / 200



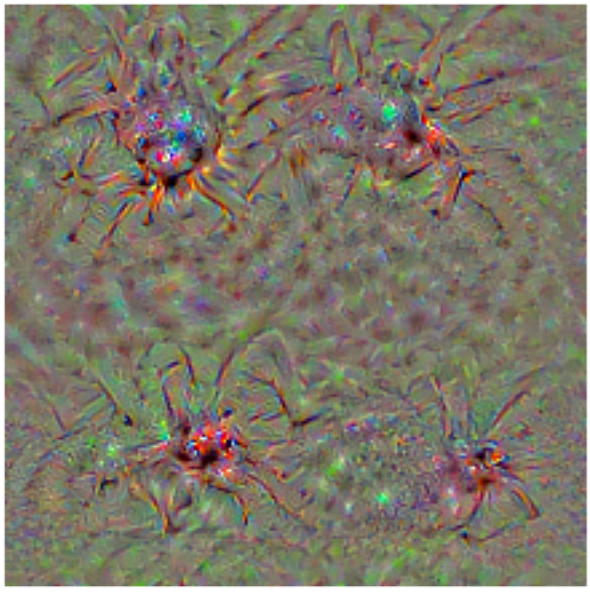
tick
Iteration 50 / 200



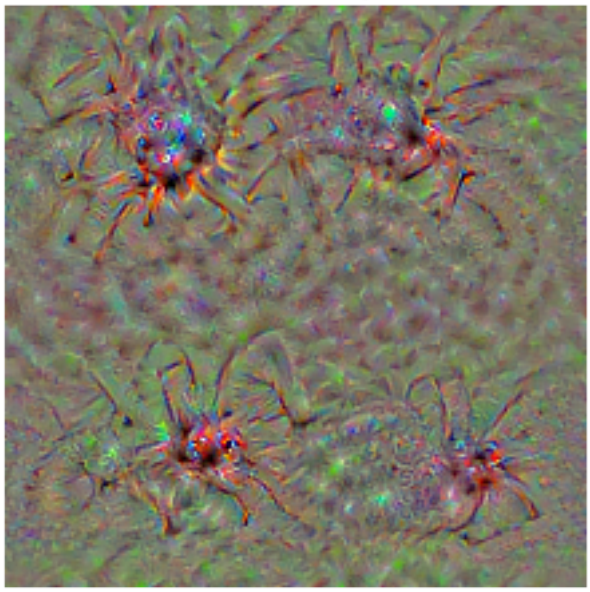
tick
Iteration 75 / 200



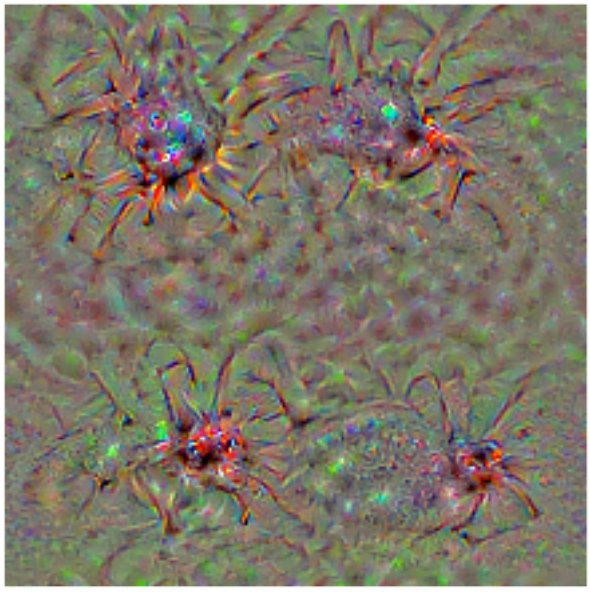
tick
Iteration 100 / 200



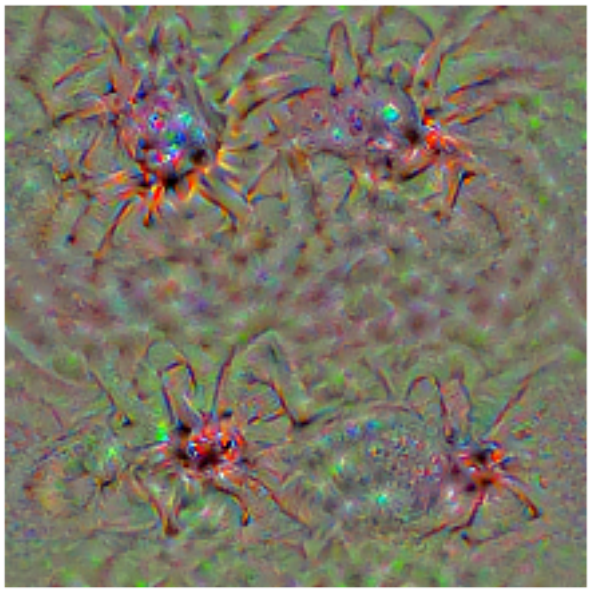
tick
Iteration 125 / 200



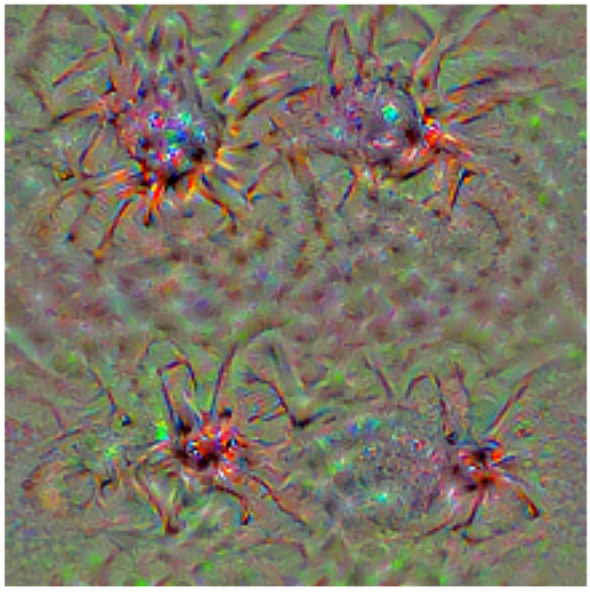
tick
Iteration 150 / 200



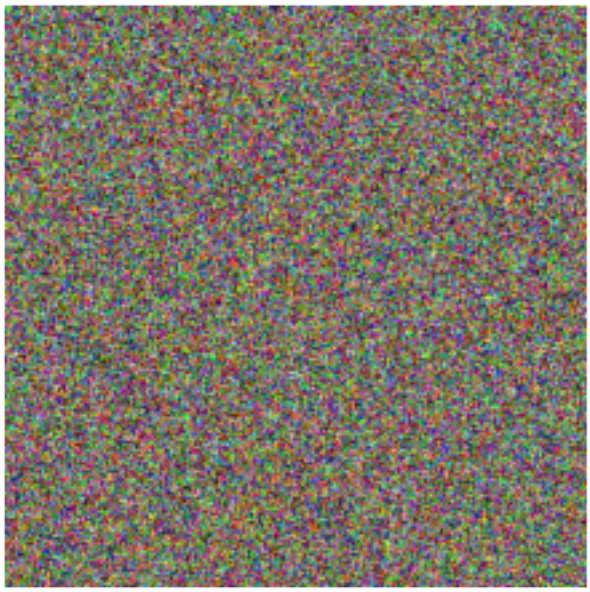
tick
Iteration 175 / 200



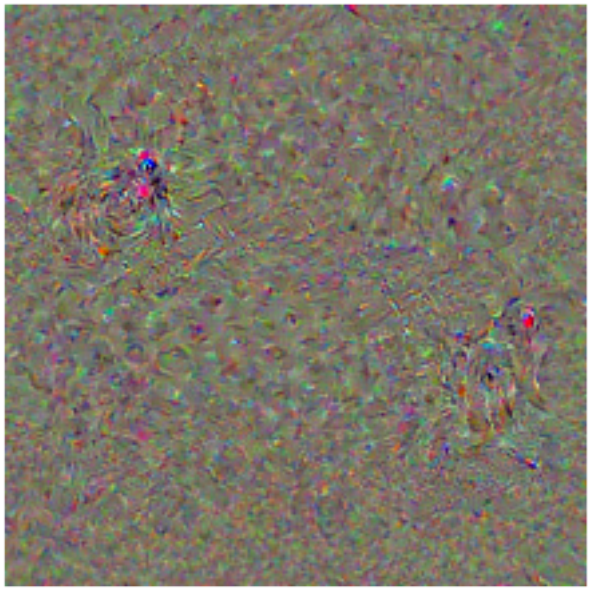
tick
Iteration 200 / 200



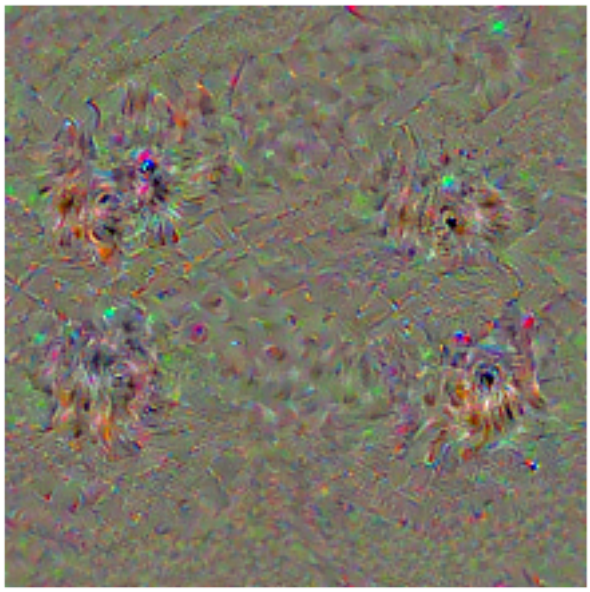
Yorkshire terrier
Iteration 1 / 200



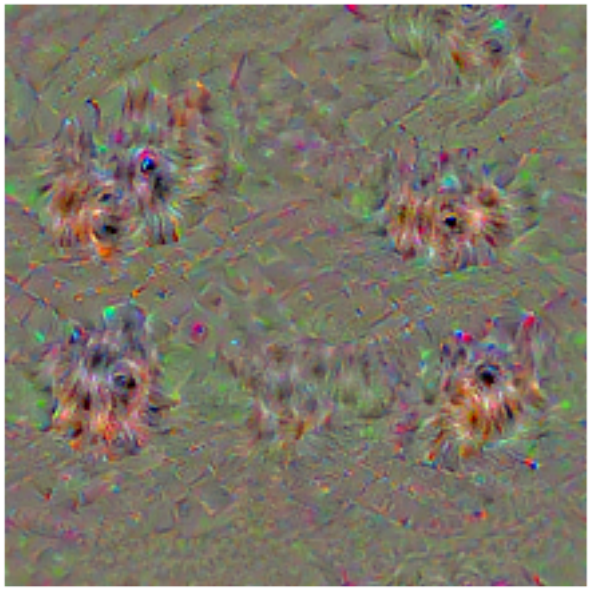
Yorkshire terrier
Iteration 25 / 200



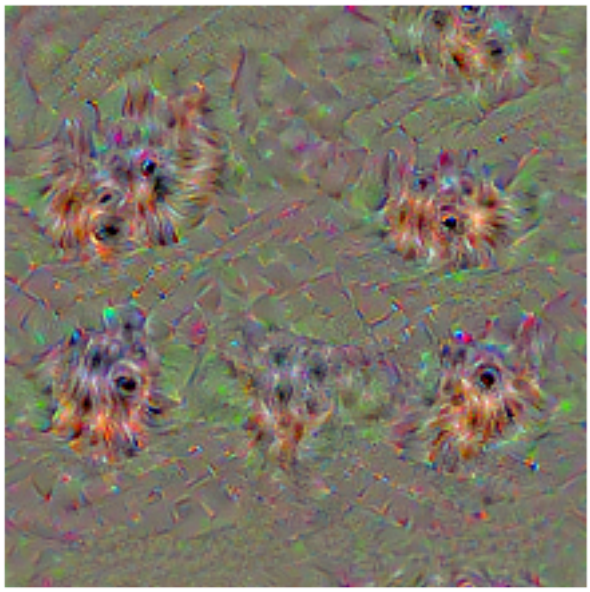
Yorkshire terrier
Iteration 50 / 200



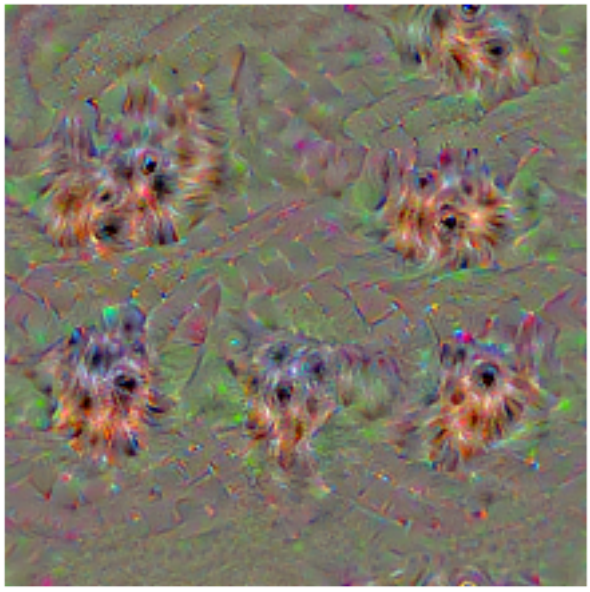
Yorkshire terrier
Iteration 75 / 200



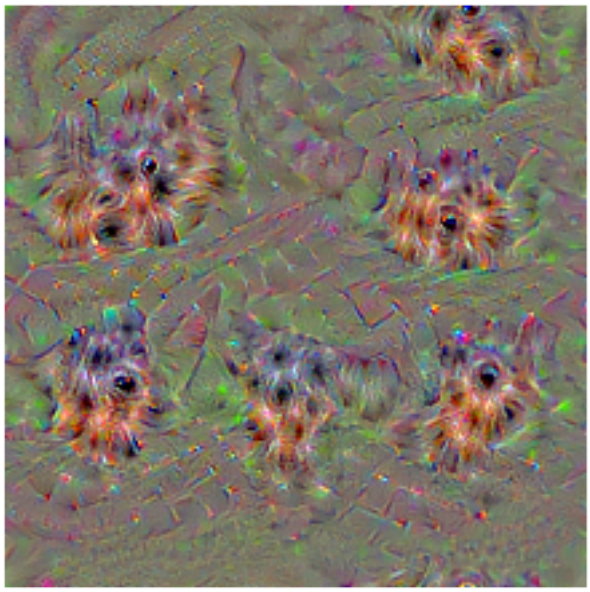
Yorkshire terrier
Iteration 100 / 200



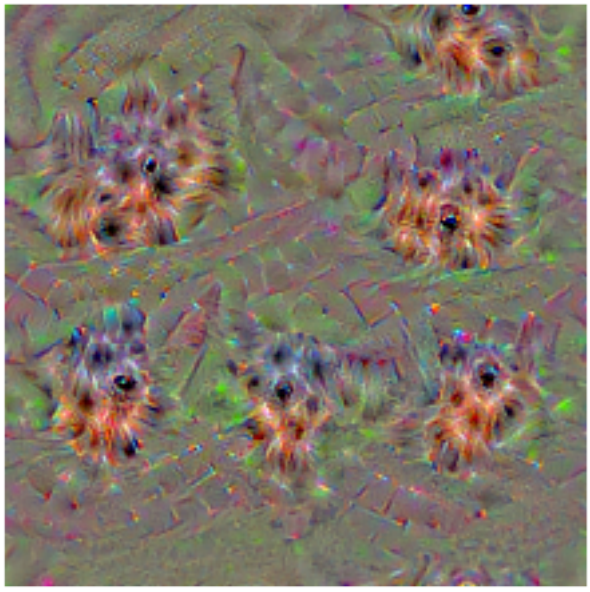
Yorkshire terrier
Iteration 125 / 200



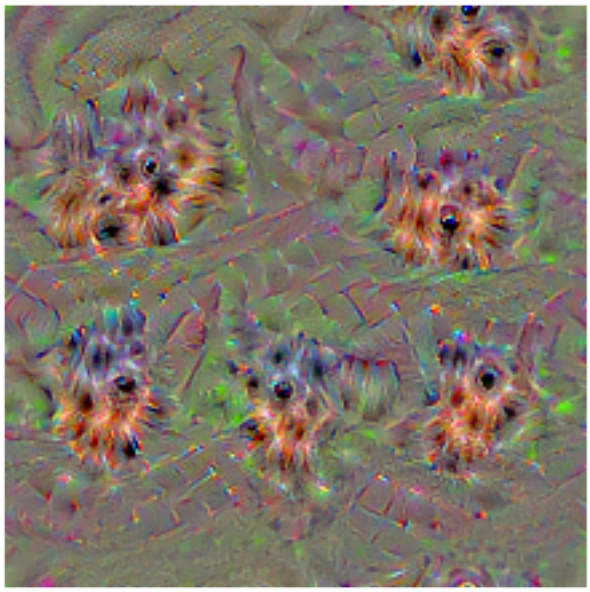
Yorkshire terrier
Iteration 150 / 200



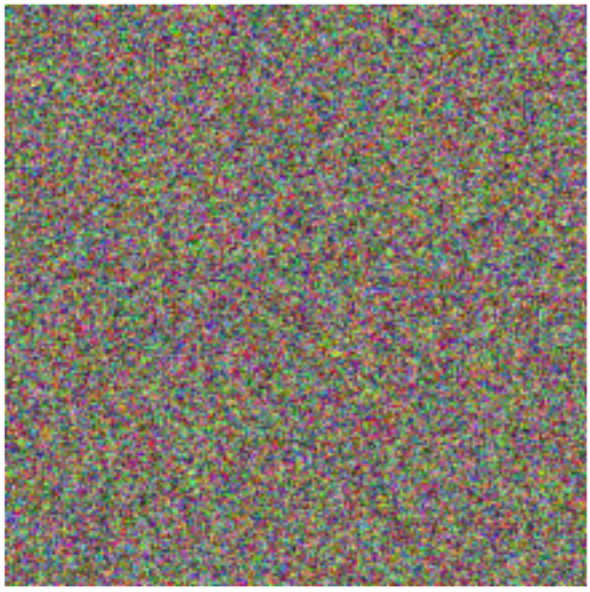
Yorkshire terrier
Iteration 175 / 200



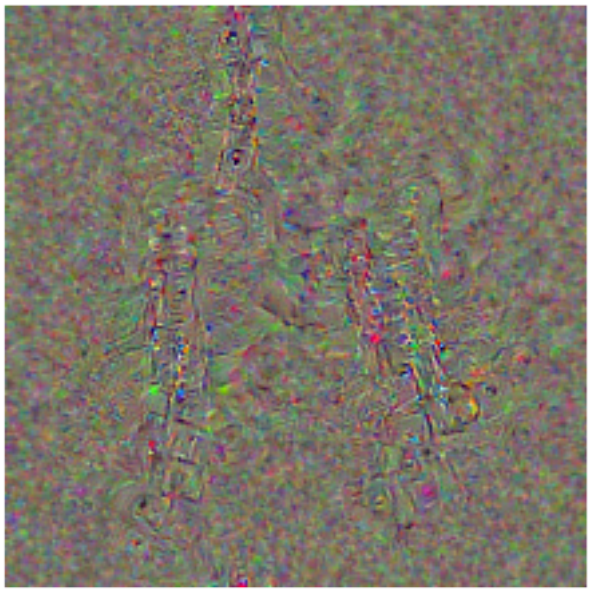
Yorkshire terrier
Iteration 200 / 200



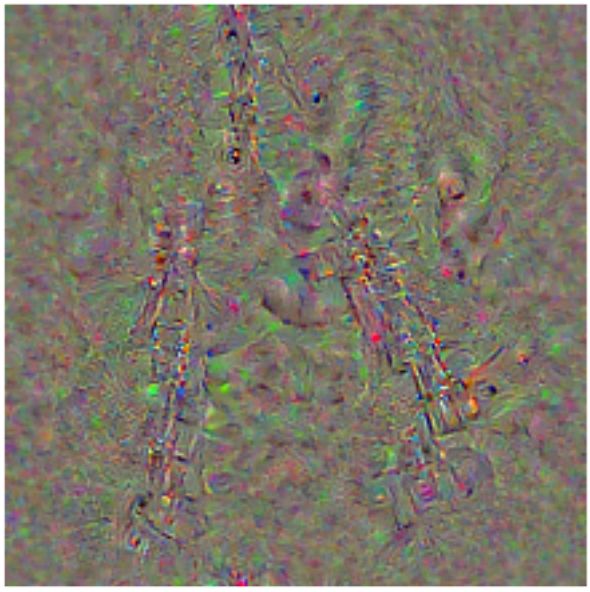
oboe, hautboy, hautbois
Iteration 1 / 200



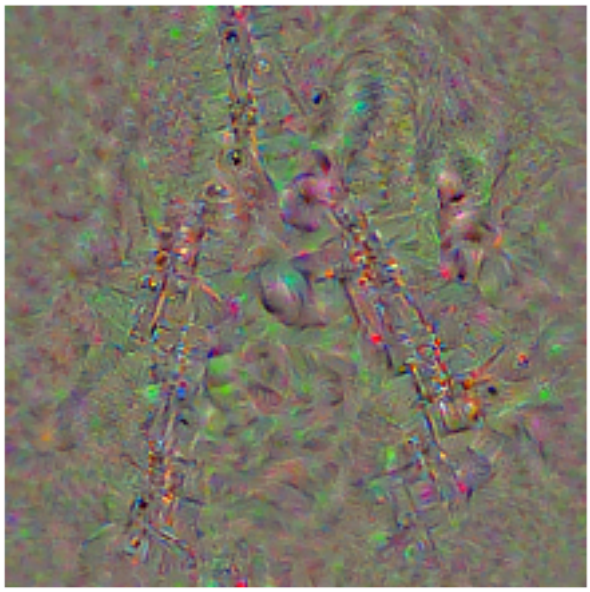
oboe, hautboy, hautbois
Iteration 25 / 200



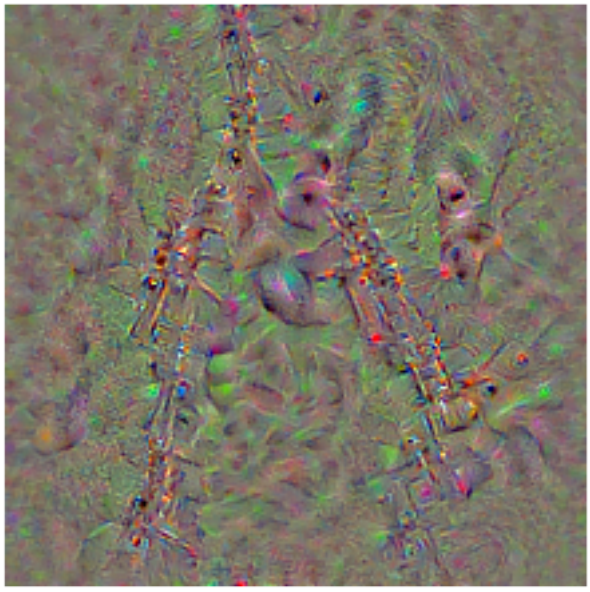
oboe, hautboy, hautbois
Iteration 50 / 200



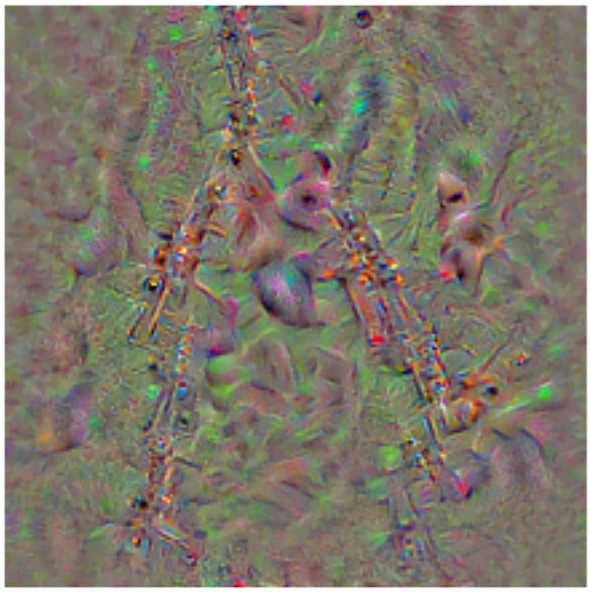
oboe, hautboy, hautbois
Iteration 75 / 200



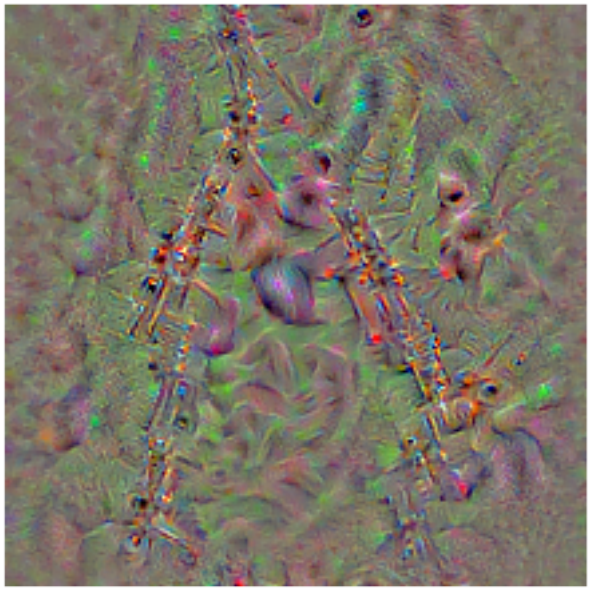
oboe, hautboy, hautbois
Iteration 100 / 200



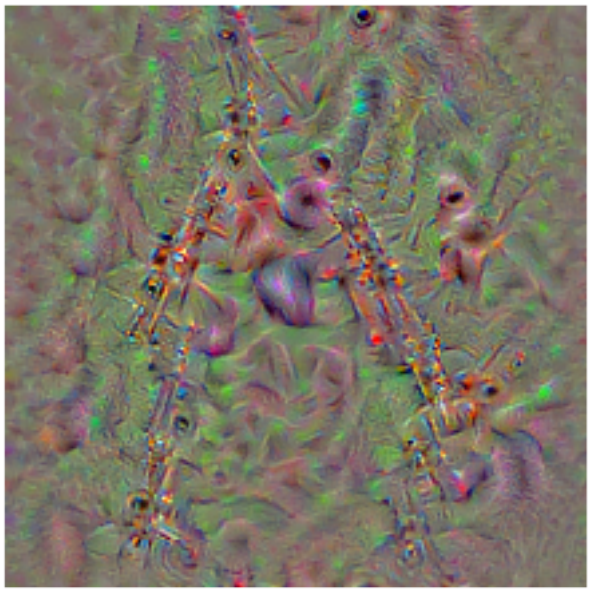
oboe, hautboy, hautbois
Iteration 125 / 200



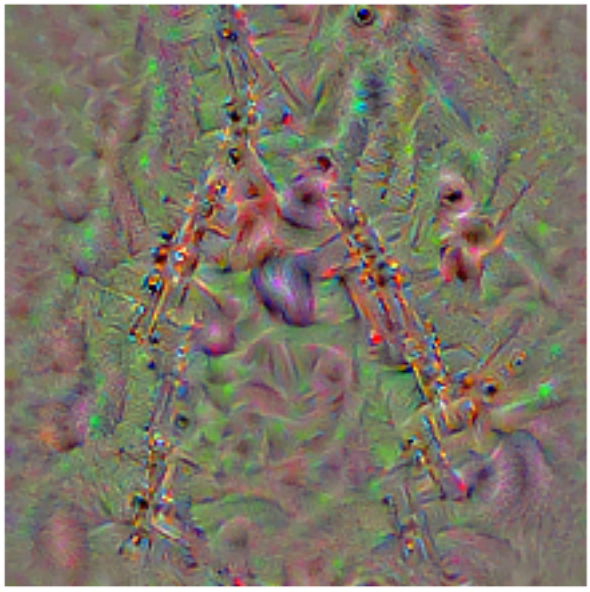
oboe, hautboy, hautbois
Iteration 150 / 200



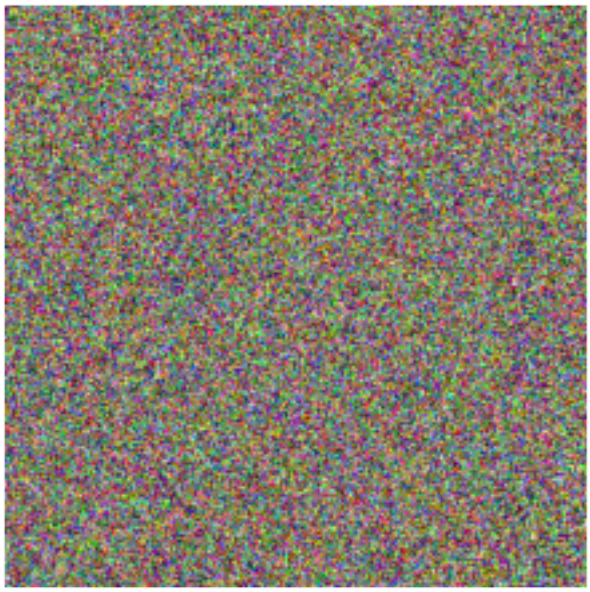
oboe, hautboy, hautbois
Iteration 175 / 200



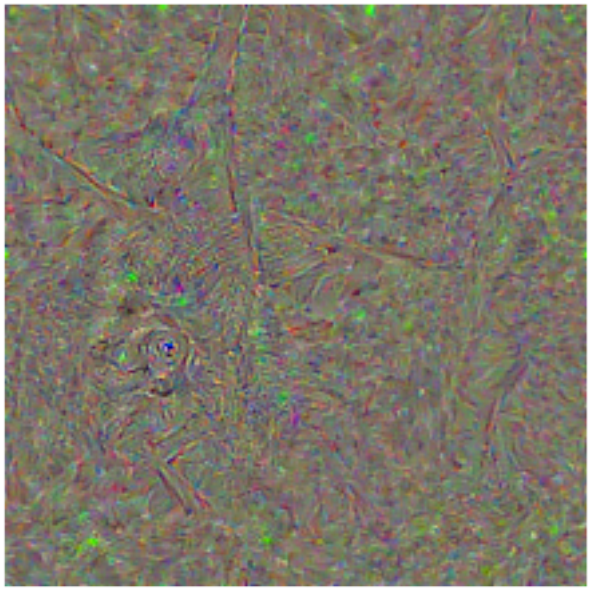
oboe, hautboy, hautbois
Iteration 200 / 200



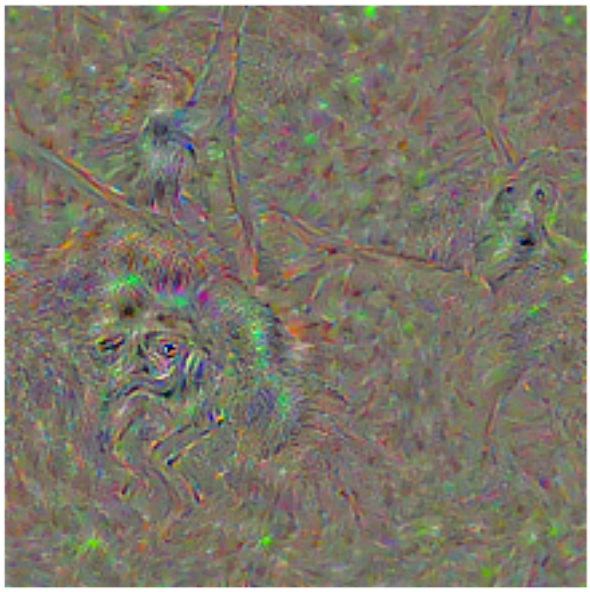
gorilla, Gorilla gorilla
Iteration 1 / 200



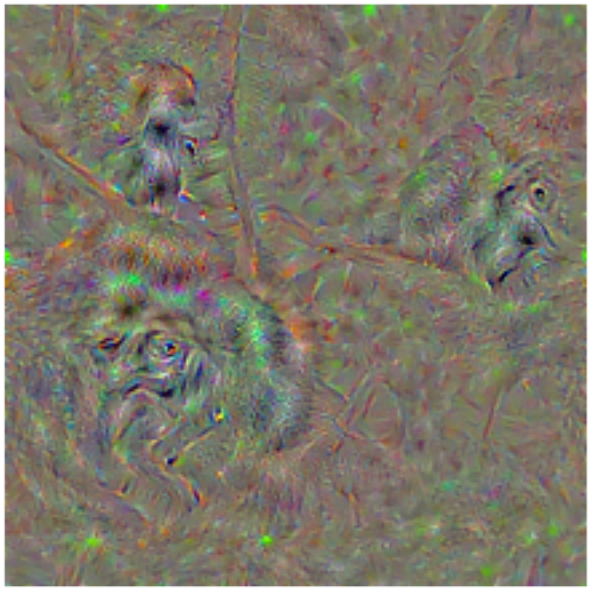
gorilla, Gorilla gorilla
Iteration 25 / 200



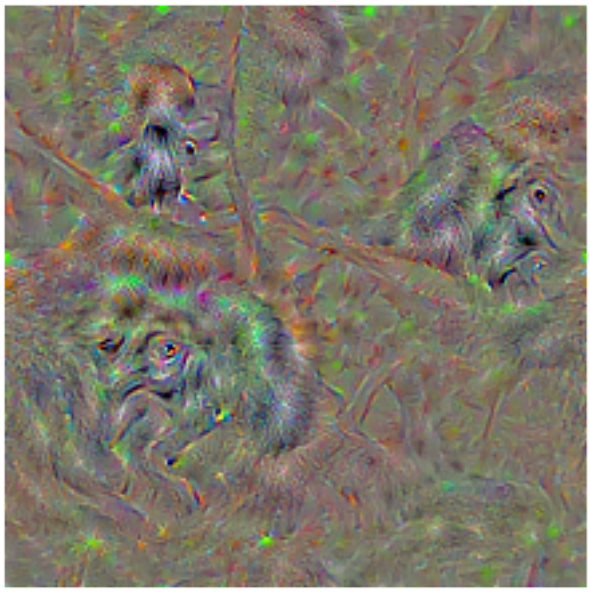
gorilla, Gorilla gorilla
Iteration 50 / 200



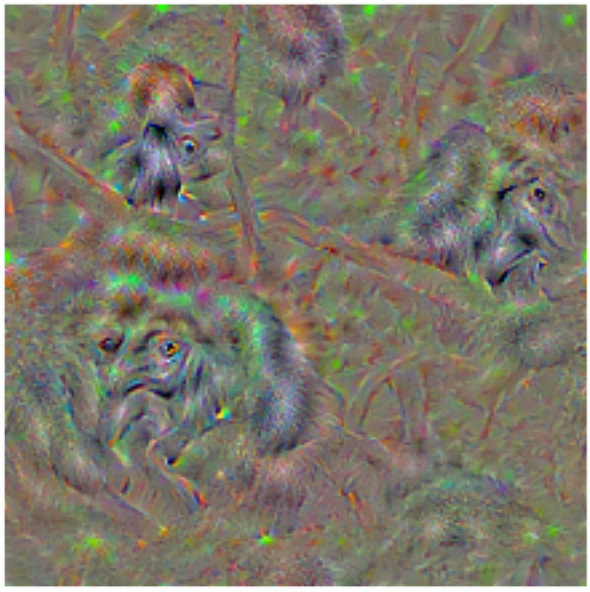
gorilla, Gorilla gorilla
Iteration 75 / 200



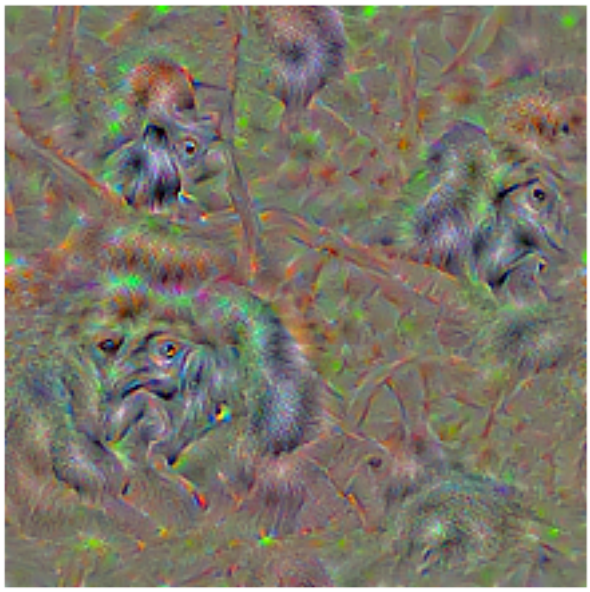
gorilla, Gorilla gorilla
Iteration 100 / 200



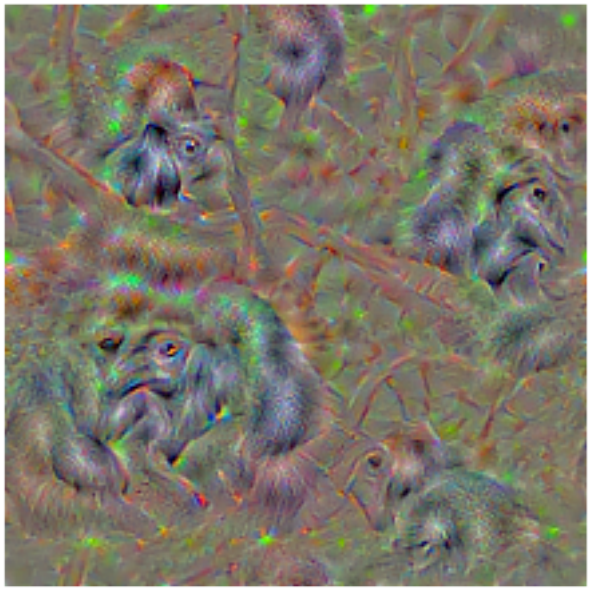
gorilla, Gorilla gorilla
Iteration 125 / 200



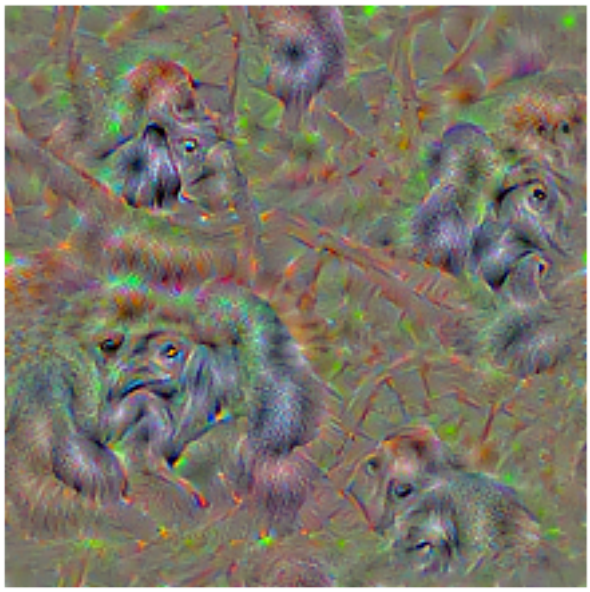
gorilla, Gorilla gorilla
Iteration 150 / 200



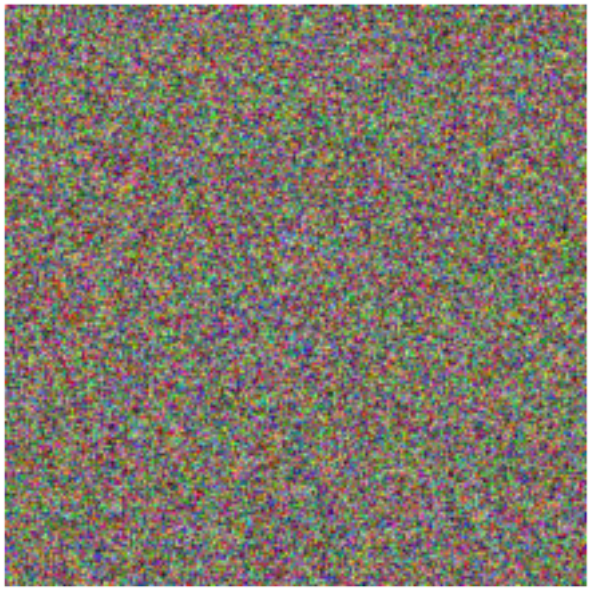
gorilla, Gorilla gorilla
Iteration 175 / 200



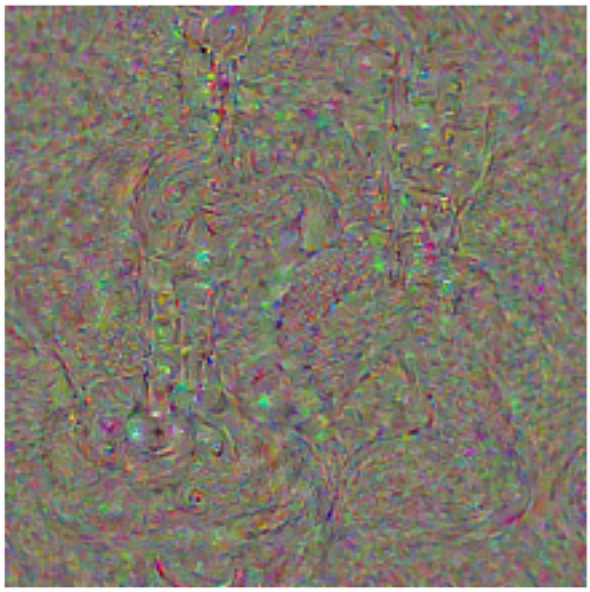
gorilla, Gorilla gorilla
Iteration 200 / 200



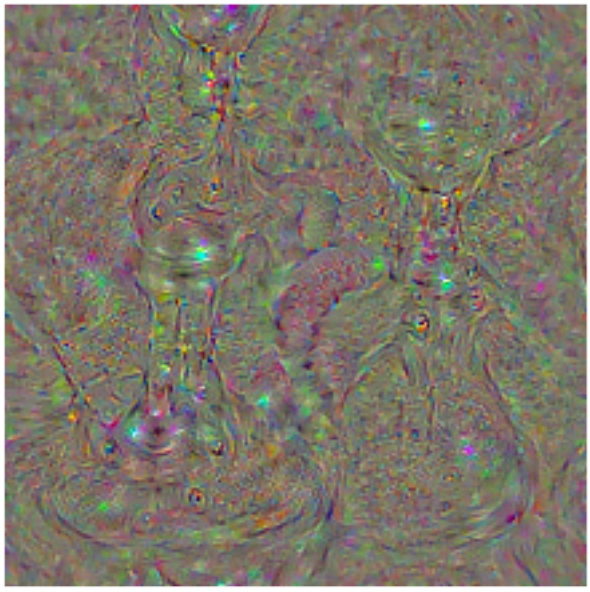
hourglass
Iteration 1 / 200



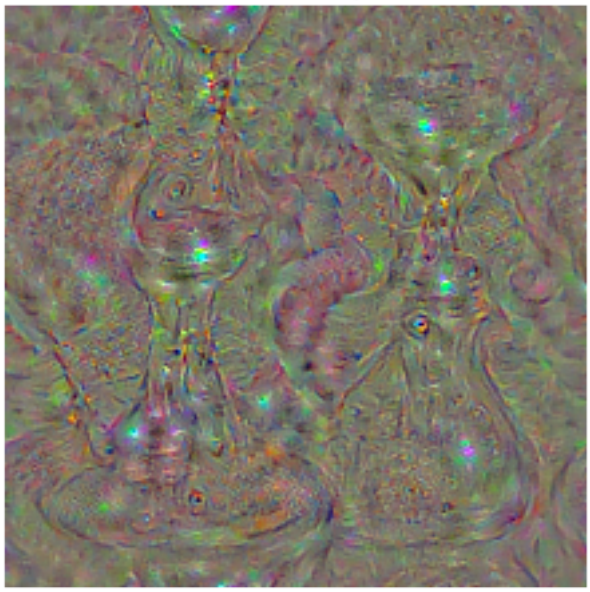
hourglass
Iteration 25 / 200



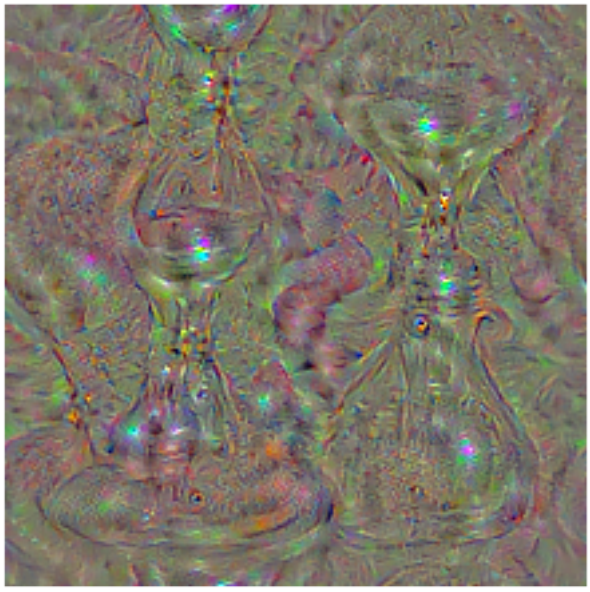
hourglass
Iteration 50 / 200



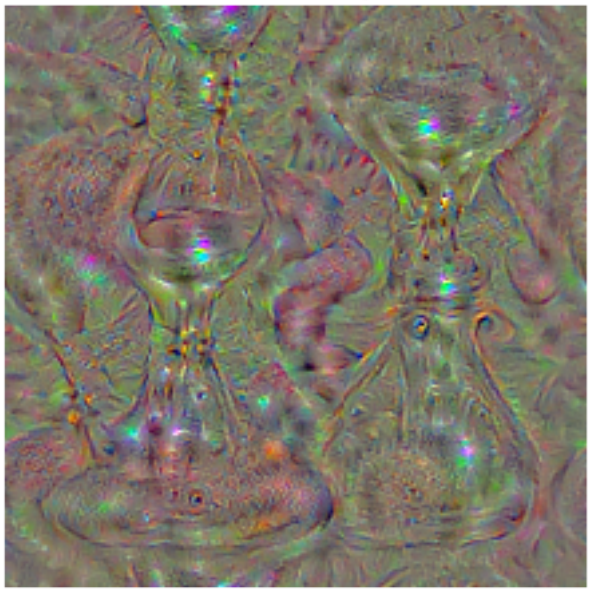
hourglass
Iteration 75 / 200



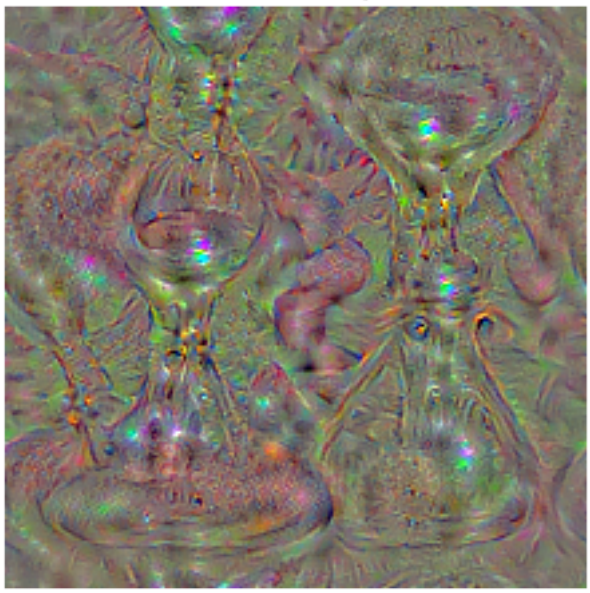
hourglass
Iteration 100 / 200



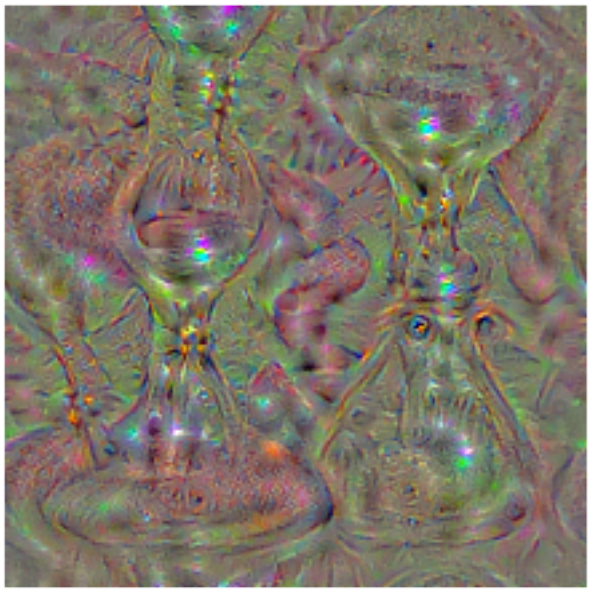
hourglass
Iteration 125 / 200



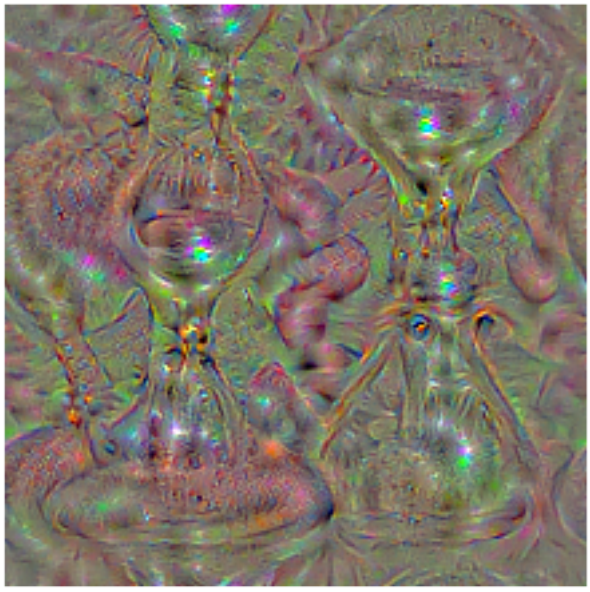
hourglass
Iteration 150 / 200



hourglass
Iteration 175 / 200



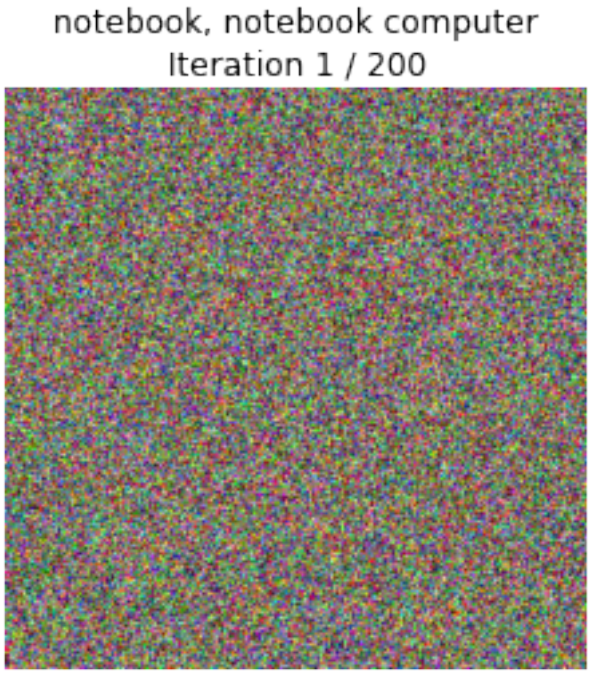
hourglass
Iteration 200 / 200



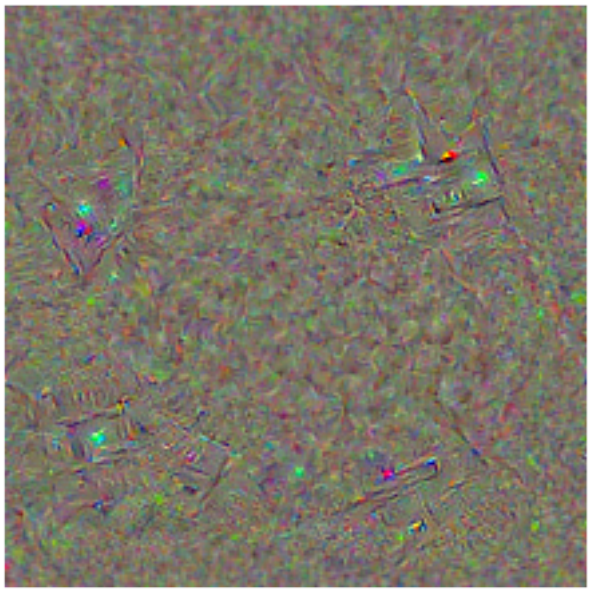
Try out your class visualization on other classes! You should also feel free to play with various hyperparameters to try and improve the quality of the generated image, but this is not required.

```
In [36]: # target_y = 78 # Tick
         # target_y = 187 # Yorkshire Terrier
         # target_y = 683 # Oboe
         # target_y = 366 # Gorilla
         # target_y = 604 # Hourglass
         target_y = np.random.randint(1000)
         print(class_names[target_y])
         X = create_class_visualization(target_y, model, dtype, num_iterations=200)
```

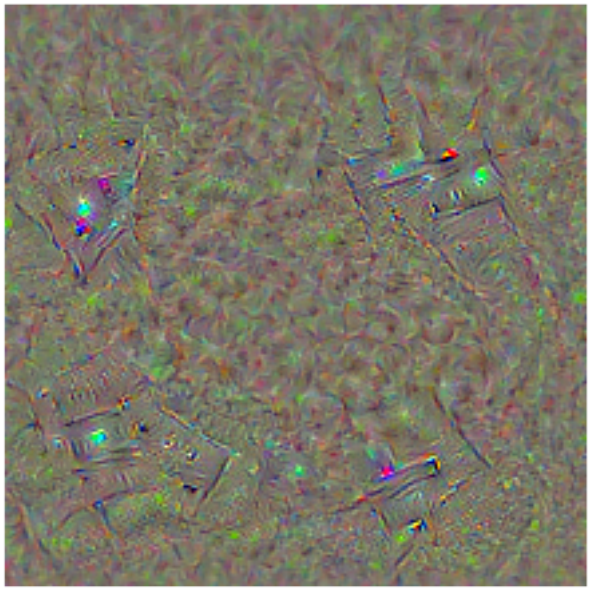
```
notebook, notebook computer
```



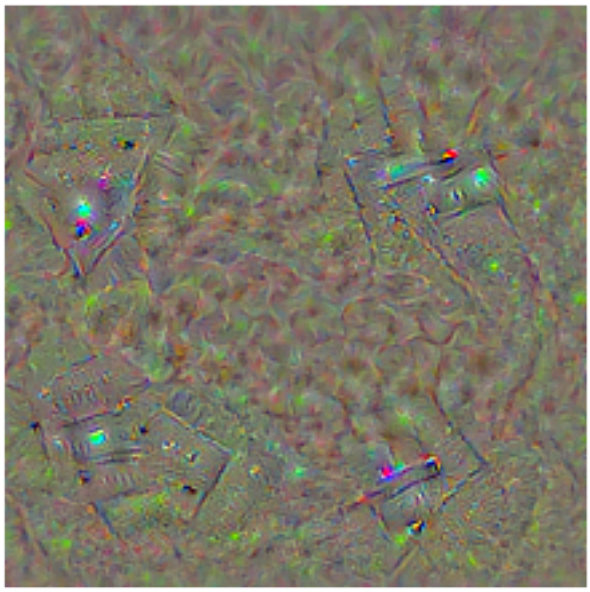
notebook, notebook computer
Iteration 25 / 200



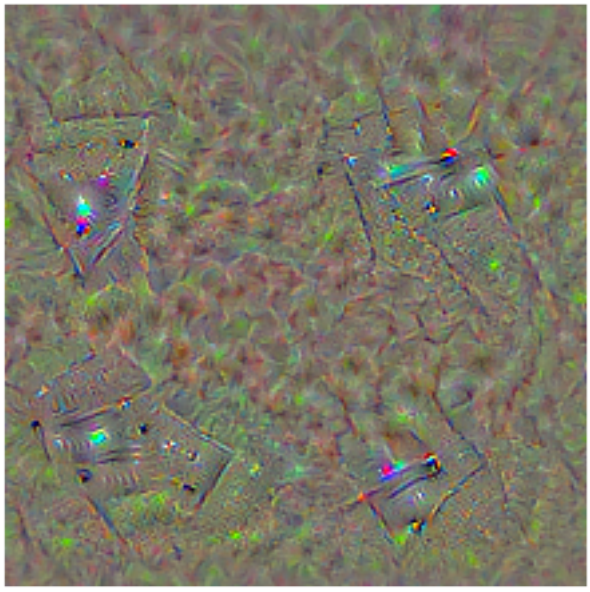
notebook, notebook computer
Iteration 50 / 200



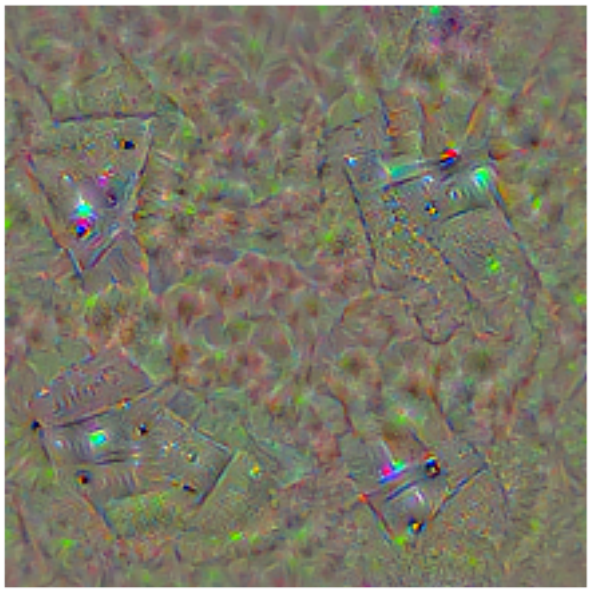
notebook, notebook computer
Iteration 75 / 200



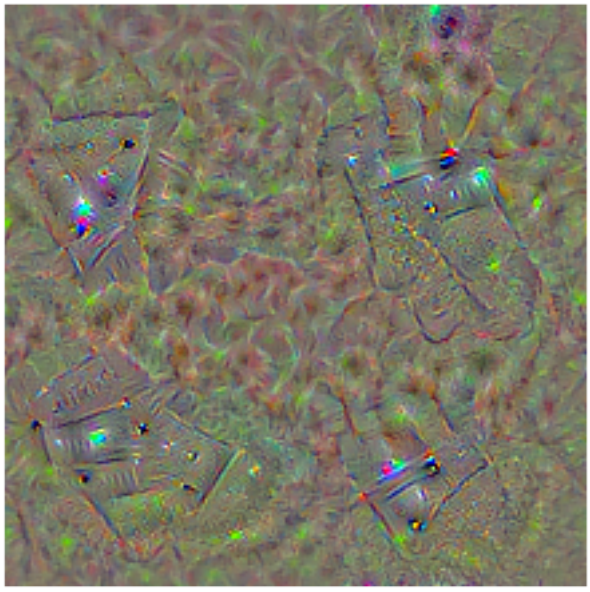
notebook, notebook computer
Iteration 100 / 200



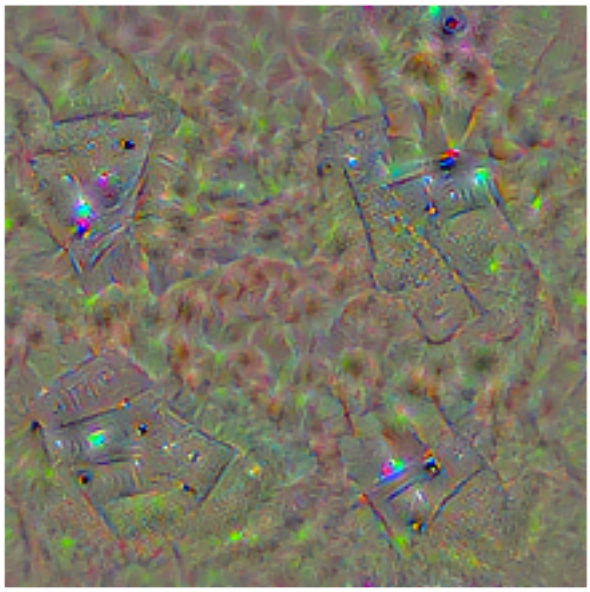
notebook, notebook computer
Iteration 125 / 200



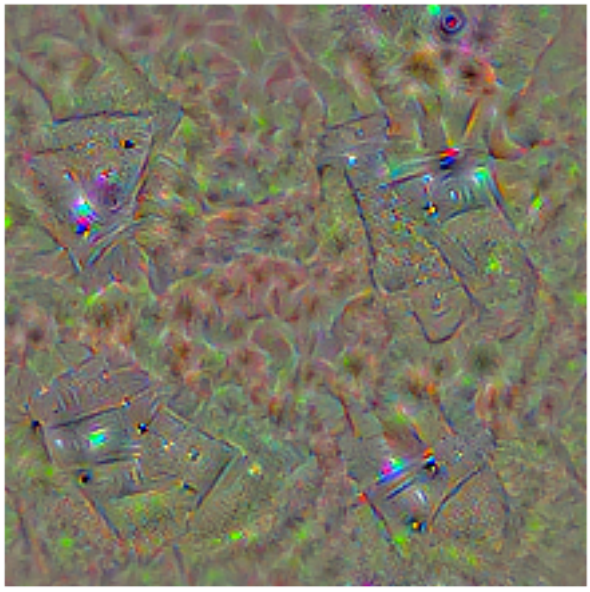
notebook, notebook computer
Iteration 150 / 200



notebook, notebook computer
Iteration 175 / 200



notebook, notebook computer
Iteration 200 / 200



6.2 Style transfert

StyleTransfer-PyTorch

October 7, 2019

1 Style Transfer (20 Points)

Another task closely related to image gradient is style transfer. This has become a cool application in deep learning with computer vision. In this notebook we will study and implement the style transfer technique from:

- "Image Style Transfer Using Convolutional Neural Networks" (Gatys et al., CVPR 2015).

The general idea is to take two images (a content image and a style image), and produce a new image that reflects the content of one but the artistic "style" of the other. We will do this by first formulating a loss function that matches the content and style of each respective image in the feature space of a deep network, and then performing gradient descent on the pixels of the image itself.

In this notebook, we will also use SqueezeNet as our feature extractor which can easily work on a CPU machine. Similarly, if computational resources are not any problem for you, you are encouraged to try a larger network, which may give you benefits in the visual output in this homework.

** Note for grading**:

- The total credits for this notebook are 20 points. For each of the loss function, **you will need to pass the unit test to receive full credits, otherwise it will be 0.** For the final output you will be expected to generate the images similar to the output to receive the full credits.
- Although we will not run your notebook in grading, you still need to **submit the notebook with all the outputs you generated**. Sometimes it will inform us if we get any inconsitent results with respect to yours.

Here's an example of the images you'll be able to produce by the end of this notebook:
Excited? Let's get started!
First, run the setup cells which provide the utility functions you will need later.

```
In [1]: import torch
        import torch.nn as nn
        from torch.autograd import Variable
        import torchvision
        import torchvision.transforms as T
        import PIL
```

caption

```
import numpy as np

from scipy.misc import imread
from collections import namedtuple
import matplotlib.pyplot as plt

from cs7643.image_utils import SQUEEZENET_MEAN, SQUEEZENET_STD
%matplotlib inline
```

We provide you with some helper functions to deal with images, since for this part of the assignment we're dealing with real JPEGs, not CIFAR-10 data.

```
In [2]: def preprocess(img, size=512):
            transform = T.Compose([
                T.Resize(size),
                T.ToTensor(),
                T.Normalize(mean=SQUEEZENET_MEAN.tolist(),
                            std=SQUEEZENET_STD.tolist()),
                T.Lambda(lambda x: x[None]),
            ])
            return transform(img)

        def deprocess(img):
            transform = T.Compose([
                T.Lambda(lambda x: x[0]),
                T.Normalize(mean=[0, 0, 0], std=[1.0 / s for s in SQUEEZENET_STD.tolist()]),
                T.Normalize(mean=[-m for m in SQUEEZENET_MEAN.tolist()], std=[1, 1, 1]),
                T.Lambda(rescale),
                T.ToPILImage(),
            ])
            return transform(img)

        def rescale(x):
            low, high = x.min(), x.max()
            x_rescaled = (x - low) / (high - low)
            return x_rescaled

        def rel_error(x,y):
```

```
        return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))

    def features_from_img(imgpath, imgsize):
        img = preprocess(PIL.Image.open(imgpath), size=imgsize)
        img_var = Variable(img.type(dtype))
        return extract_features(img_var, cnn), img_var

    # Older versions of scipy.misc.imresize yield different results
    # from newer versions, so we check to make sure scipy is up to date.
    def check_scipy():
        import scipy
        vnums = list(map(int, scipy.__version__.split('.')))
        assert vnums[1] >= 16 or vnums[0] >= 1, "You must install SciPy >= 0.16.0 to compl

    check_scipy()

    answers = np.load('style-transfer-checks.npz')
```

As in the last notebook, we need to set the dtype to select either the CPU or the GPU

```
In [9]: dtype = torch.FloatTensor
        # Uncomment out the following line if you're on a machine with a GPU set up for PyTorc
        # dtype = torch.cuda.FloatTensor

In [10]: # Load the pre-trained SqueezeNet model.
         cnn = torchvision.models.squeezenet1_1(pretrained=True).features
         cnn.type(dtype)

         # Fix the weights of the pretrained network
         for param in cnn.parameters():
             param.requires_grad = False

         # We provide this helper code which takes an image, a model (cnn), and returns a list
         # feature maps, one per layer.
         def extract_features(x, cnn):
             """
             Use the CNN to extract features from the input image x.

             Inputs:
             - x: A PyTorch Variable of shape (N, C, H, W) holding a minibatch of images that
               will be fed to the CNN.
             - cnn: A PyTorch model that we will use to extract features.

             Returns:
             - features: A list of feature for the input images x extracted using the cnn mode
               features[i] is a PyTorch Variable of shape (N, C_i, H_i, W_i); recall that feat
               from different layers of the network may have different numbers of channels (C_
               spatial dimensions (H_i, W_i).
```

```
    """
    features = []
    prev_feat = x
    for i, module in enumerate(cnn._modules.values()):
        next_feat = module(prev_feat)
        features.append(next_feat)
        prev_feat = next_feat
    return features
```

1.1 Implementation: Computing Loss

We're going to compute the three components of our loss function now. The loss function is a weighted sum of three terms: content loss + style loss + total variation loss. You'll fill in the functions that compute these weighted terms below.

1.2 Content loss (3 pts)

We can generate an image that reflects the content of one image and the style of another by incorporating both in our loss function. We want to penalize deviations from the content of the content image and deviations from the style of the style image. We can then use this hybrid loss function to perform gradient descent **not on the parameters** of the model, but instead **on the pixel values** of our original image.

Let's first write the content loss function. Content loss measures how much the feature map of the generated image differs from the feature map of the source image. We only care about the content representation of one layer of the network (say, layer ℓ), that has feature maps $A^\ell \in \mathbb{R}^{1\times C_\ell \times H_\ell \times W_\ell}$. C_ℓ is the number of filters/channels in layer ℓ, H_ℓ and W_ℓ are the height and width. We will work with reshaped versions of these feature maps that combine all spatial positions into one dimension. Let $F^\ell \in \mathbb{R}^{N_\ell \times M_\ell}$ be the feature map for the current image and $P^\ell \in \mathbb{R}^{N_\ell \times M_\ell}$ be the feature map for the content source image where $M_\ell = H_\ell \times W_\ell$ is the number of elements in each feature map. Each row of F^ℓ or P^ℓ represents the vectorized activations of a particular filter, convolved over all positions of the image. Finally, let w_c be the weight of the content loss term in the loss function.

Then the content loss is given by:
$L_c = w_c \times \sum_{i,j}(F_{ij}^\ell - P_{ij}^\ell)^2$

```
In [13]: def content_loss(content_weight, content_current, content_original):
             """
             Compute the content loss for style transfer.

             Inputs:
             - content_weight: Scalar giving the weighting for the content loss.
             - content_current: features of the current image; this is a PyTorch Tensor of sha
               (1, C_l, H_l, W_l).
             - content_target: features of the content image, Tensor with shape (1, C_l, H_l,

             Returns:
             - scalar content loss
             """
```

```
        ##############################################################################
        # TODO: Implement content loss function                                      #
        # Please pay attention to use torch tensor math function to finish it.       #
        # Otherwise, you may run into the issues later that dynamic graph is broken  #
        # and gradient can not be derived.                                           #
        ##############################################################################
        loss = content_weight * (content_current - content_original).pow(2).sum()
        return loss
        ##############################################################################
        #                               END OF YOUR CODE                             #
        ##############################################################################
```

Test your content loss function. You should see errors less than 0.001 (normally it should be exactly 0).

```
In [14]: def content_loss_test(correct):
             content_image = 'styles/tubingen.jpg'
             image_size =  192
             content_layer = 3
             content_weight = 6e-2

             c_feats, content_img_var = features_from_img(content_image, image_size)

             bad_img = Variable(torch.zeros(*content_img_var.data.size()))
             feats = extract_features(bad_img, cnn)

             student_output = content_loss(content_weight, c_feats[content_layer], feats[conte
             error = rel_error(correct, student_output)
             print('Maximum error is {:.3f}'.format(error))

         content_loss_test(answers['cl_out'])
```

```
Maximum error is 0.000
```

1.3 Style loss (3 pts for Gram matrix + 3 pts for loss)

Now we can tackle the style loss. For a given layer ℓ, the style loss is defined as follows:

First, compute the Gram matrix G which represents the correlations between the responses of each filter, where F is as above. The Gram matrix is an approximation to the covariance matrix -- we want the activation statistics of our generated image to match the activation statistics of our style image, and matching the (approximate) covariance is one way to do that. There are a variety of ways you could do this, but the Gram matrix is nice because it's easy to compute and in practice shows good results.

Given a feature map F^ℓ of shape $(1, C_\ell, M_\ell)$, the Gram matrix has shape $(1, C_\ell, C_\ell)$ and its elements are given by:

$$G^\ell_{ij} = \sum_k F^\ell_{ik} F^\ell_{jk}$$

Assuming G^ℓ is the Gram matrix from the feature map of the current image, A^ℓ is the Gram Matrix from the feature map of the source style image, and w_ℓ a scalar weight term, then the style loss for the layer ℓ is simply the weighted Euclidean distance between the two Gram matrices:

$$L_s^\ell = w_\ell \sum_{i,j} \left(G_{ij}^\ell - A_{ij}^\ell \right)^2$$

In practice we usually compute the style loss at a set of layers $\mathcal{L}$ rather than just a single layer ℓ; then the total style loss is the sum of style losses at each layer:

$$L_s = \sum_{\ell \in \mathcal{L}} L_s^\ell$$

Begin by implementing the Gram matrix computation below:

```
In [33]: def gram_matrix(features, normalize=True):
             """
             Compute the Gram matrix from features.

             Inputs:
             - features: PyTorch Variable of shape (N, C, H, W) giving features for
               a batch of N images.
             - normalize: optional, whether to normalize the Gram matrix
                 If True, divide the Gram matrix by the number of neurons (H * W * C)

             Returns:
             - gram: PyTorch Variable of shape (N, C, C) giving the
               (optionally normalized) Gram matrices for the N input images.
             """
             ################################################################################
             # TODO: Implement content loss function                                        #
             # Please pay attention to use torch tensor math function to finish it.         #
             # Otherwise, you may run into the issues later that dynamic graph is broken    #
             # and gradient can not be derived.                                             #
             #                                                                              #
             # HINT: you may find torch.bmm() function is handy when it comes to process    #
             # matrix product in a batch. Please check the document about how to use it.    #
             ################################################################################
             features = features.reshape((features.shape[0], features.shape[1], -1))
             gram = torch.bmm(features, features.transpose(1, 2))
             if normalize:
                 gram /= features.reshape((features.shape[0], -1)).shape[1]
             return gram
             ################################################################################
             #                               END OF YOUR CODE                               #
             ################################################################################
```

Test your Gram matrix code. You should see errors less than 0.001 (normally it should be exactly 0).

```
In [34]: def gram_matrix_test(correct):
             style_image = 'styles/starry_night.jpg'
             style_size = 192
             feats, _ = features_from_img(style_image, style_size)
             student_output = gram_matrix(feats[5].clone()).data.numpy()
             error = rel_error(correct, student_output)
             print('Maximum error is {:.3f}'.format(error))

         gram_matrix_test(answers['gm_out'])
```

```
Maximum error is 0.000
```

Next, implement the style loss:

```
In [64]: # Now put it together in the style_loss function...
         def style_loss(feats, style_layers, style_targets, style_weights):
             """
             Computes the style loss at a set of layers.

             Inputs:
             - feats: list of the features at every layer of the current image, as produced by
               the extract_features function.
             - style_layers: List of layer indices into feats giving the layers to include in
               style loss.
             - style_targets: List of the same length as style_layers, where style_targets[i]
               a PyTorch Variable giving the Gram matrix the source style image computed at
               layer style_layers[i].
             - style_weights: List of the same length as style_layers, where style_weights[i]
               is a scalar giving the weight for the style loss at layer style_layers[i].

             Returns:
             - style_loss: A PyTorch Variable holding a scalar giving the style loss.
             """

             ################################################################################
             # TODO: Implement content loss function                                        #
             # Please pay attention to use torch tensor math function to finish it.         #
             # Otherwise, you may run into the issues later that dynamic graph is broken    #
             # and gradient can not be derived.                                             #
             #                                                                              #
             # Hint:                                                                        #
             # you can do this with one for loop over the style layers, and should not be   #
             # very much code (~5 lines). Please refer to the 'style_loss_test' for the     #
             # actual data structure.                                                       #
             #                                                                              #
             # You will need to use your gram_matrix function.                              #
             ################################################################################
```

```
#     feat_gram = gram_matrix(feats)
    loss = torch.tensor(0.0)
    for l, target_gram_layer, w in zip(style_layers, style_targets, style_weights):
        loss += (gram_matrix(feats[l]) - target_gram_layer).pow(2).sum() * w
    return loss
    ##############################################################################
    #                               END OF YOUR CODE                             #
    ##############################################################################
```

Test your style loss implementation. The error should be less than 0.001 (normally it should be exactly 0).

```
In [65]: def style_loss_test(correct):
    content_image = 'styles/tubingen.jpg'
    style_image = 'styles/starry_night.jpg'
    image_size =  192
    style_size = 192
    style_layers = [1, 4, 6, 7]
    style_weights = [300000, 1000, 15, 3]

    c_feats, _ = features_from_img(content_image, image_size)
    feats, _ = features_from_img(style_image, style_size)
    style_targets = []
    for idx in style_layers:
        style_targets.append(gram_matrix(feats[idx].clone()))

    student_output = style_loss(c_feats, style_layers, style_targets, style_weights).
    error = rel_error(correct, student_output)
    print('Error is {:.3f}'.format(error))


style_loss_test(answers['sl_out'])
```

```
Error is 0.000
```

1.4 Total-variation regularization (3 pts)

It turns out that it's helpful to also encourage smoothness in the image. We can do this by adding another term to our loss that penalizes wiggles or **total variation** in the pixel values. This concept is widely used in many computer vision task as a regularization term.

You can compute the "total variation" as the sum of the squares of differences in the pixel values for all pairs of pixels that are next to each other (horizontally or vertically). Here we sum the total-variation regualarization for each of the 3 input channels (RGB), and weight the total summed loss by the total variation weight, w_t:

$L_{tv} = w_t \times \sum_{c=1}^{3} \sum_{i=1}^{H-1} \sum_{j=1}^{W-1} \left((x_{i,j+1,c} - x_{i,j,c})^2 + (x_{i+1,j,c} - x_{i,j,c})^2 \right)$

You may not see this loss function in this particular reference paper, but you should be able to implement it based on this equation. In the next cell, fill in the definition for the TV loss term.

You need to provide an efficient vectorized implementation to receive the full credit, your implementation should not have any loops. Otherwise, penalities will be given according to the actual implementation.

```
In [66]: def tv_loss(img, tv_weight):
             """
             Compute total variation loss.

             Inputs:
             - img: PyTorch Variable of shape (1, 3, H, W) holding an input image.
             - tv_weight: Scalar giving the weight w_t to use for the TV loss.

             Returns:
             - loss: PyTorch Variable holding a scalar giving the total variation loss
               for img weighted by tv_weight.
             """

             ################################################################################
             # TODO: Implement content loss function                                        #
             # Please pay attention to use torch tensor math function to finish it.         #
             # Otherwise, you may run into the issues later that dynamic graph is broken    #
             # and gradient can not be derived.                                             #
             ################################################################################
             loss = (img[:,:,1:,:] - img[:,:,:-1,:]).pow(2).sum()
             loss += (img[:,:,:,1:] - img[:,:,:,:-1]).pow(2).sum()
             loss *= tv_weight
             return loss
             ################################################################################
             #                              END OF YOUR CODE                                #
             ################################################################################
```

Test your TV loss implementation. Error should be less than 0.001 (normally it should be exactly 0).

```
In [67]: def tv_loss_test(correct):
             content_image = 'styles/tubingen.jpg'
             image_size =  192
             tv_weight = 2e-2

             content_img = preprocess(PIL.Image.open(content_image), size=image_size)
             content_img_var = Variable(content_img.type(dtype))

             student_output = tv_loss(content_img_var, tv_weight).data.numpy()
             error = rel_error(correct, student_output)
             print('Error is {:.3f}'.format(error))

         tv_loss_test(answers['tv_out'])
```

```
Error is 0.000
```

1.5 Implement style transfer (6 pts)

You have implemented all the loss functions in the paper. Now we're ready to string it all together. Please read the entire function: figure out what are all the parameters, inputs, solvers, etc. **The update rule in the following block is hold out for you to finish.**

```
In [89]: def style_transfer(content_image, style_image, image_size, style_size, content_layer,
                            style_layers, style_weights, tv_weight, init_random = False):
             """
             Run style transfer!

             Inputs:
             - content_image: filename of content image
             - style_image: filename of style image
             - image_size: size of smallest image dimension (used for content loss and generat
             - style_size: size of smallest style image dimension
             - content_layer: layer to use for content loss
             - content_weight: weighting on content loss
             - style_layers: list of layers to use for style loss
             - style_weights: list of weights to use for each layer in style_layers
             - tv_weight: weight of total variation regularization term
             - init_random: initialize the starting image to uniform random noise
             """

             # Extract features for the content image
             content_img = preprocess(PIL.Image.open(content_image), size=image_size)
             content_img_var = Variable(content_img.type(dtype))
             feats = extract_features(content_img_var, cnn)
             content_target = feats[content_layer].clone()

             # Extract features for the style image
             style_img = preprocess(PIL.Image.open(style_image), size=style_size)
             style_img_var = Variable(style_img.type(dtype))
             feats = extract_features(style_img_var, cnn)
             style_targets = []
             for idx in style_layers:
                 style_targets.append(gram_matrix(feats[idx].clone()))

             # Initialize output image to content image or nois
             if init_random:
                 img = torch.Tensor(content_img.size()).uniform_(0, 1)
             else:
                 img = content_img.clone().type(dtype)

             # We do want the gradient computed on our image!
             img_var = Variable(img, requires_grad=True)

             # Set up optimization hyperparameters
             initial_lr = 3.0
```

```
decayed_lr = 0.1
decay_lr_at = 180

# Note that we are optimizing the pixel values of the image by passing
# in the img_var Torch variable, whose requires_grad flag is set to True
optimizer = torch.optim.Adam([img_var], lr=initial_lr)

f, axarr = plt.subplots(1,2)
axarr[0].axis('off')
axarr[1].axis('off')
axarr[0].set_title('Content Source Img.')
axarr[1].set_title('Style Source Img.')
axarr[0].imshow(deprocess(content_img.cpu()))
axarr[1].imshow(deprocess(style_img.cpu()))
plt.show()
plt.figure()

for t in range(200):
    if t < 190:
        img.clamp_(-1.5, 1.5)
    feats = extract_features(img_var, cnn)

    ################################################################################
    # TODO: Implement this update rule with by forwarding it to criterion          #
    # functions and perform the backward update.                                   #
    #                                                                              #
    # HINTS: all the weights, loss functions are defined. You don't need to add    #
    # any other extra weights for the three loss terms.                            #
    # The optimizer needs to clear its grad before backward in every step.         #
    ################################################################################
    if t == decay_lr_at:
        for param_group in optimizer.param_groups:
            param_group['lr'] = decayed_lr
    loss = tv_loss(img_var, tv_weight) + style_loss(feats, style_layers, style_ta
    optimizer.zero_grad()
    loss.backward()

    optimizer.step()
    ################################################################################
    #                               END OF YOUR CODE                               #
    ################################################################################

    if t % 100 == 0:
        print('Iteration {}'.format(t))
        plt.axis('off')
        plt.imshow(deprocess(img.cpu()))
        plt.show()
print('Iteration {}'.format(t))
```

```
    plt.axis('off')
    plt.imshow(deprocess(img.cpu()))
    plt.show()
```

1.6 Generate some pretty pictures!

Try out `style_transfer` on the three different parameter sets below. Make sure to run all three cells. Feel free to add your own, but make sure to include the results of style transfer on the third parameter set (starry night) in your submitted notebook.

- The `content_image` is the filename of content image.
- The `style_image` is the filename of style image.
- The `image_size` is the size of smallest image dimension of the content image (used for content loss and generated image).
- The `style_size` is the size of smallest style image dimension.
- The `content_layer` specifies which layer to use for content loss.
- The `content_weight` gives weighting on content loss in the overall loss function. Increasing the value of this parameter will make the final image look more realistic (closer to the original content).
- `style_layers` specifies a list of which layers to use for style loss.
- `style_weights` specifies a list of weights to use for each layer in style_layers (each of which will contribute a term to the overall style loss). We generally use higher weights for the earlier style layers because they describe more local/smaller scale features, which are more important to texture than features over larger receptive fields. In general, increasing these weights will make the resulting image look less like the original content and more distorted towards the appearance of the style image.
- `tv_weight` specifies the weighting of total variation regularization in the overall loss function. Increasing this value makes the resulting image look smoother and less jagged, at the cost of lower fidelity to style and content.

Below the next three cells of code (in which you shouldn't change the hyperparameters), feel free to copy and paste the parameters to play around them and see how the resulting image changes.

```
In [90]: # Composition VII + Tubingen
         params1 = {
             'content_image' : 'styles/tubingen.jpg',
             'style_image' : 'styles/composition_vii.jpg',
             'image_size' : 192,
             'style_size' : 512,
             'content_layer' : 3,
             'content_weight' : 5e-2,
             'style_layers' : (1, 4, 6, 7),
             'style_weights' : (20000, 500, 12, 1),
             'tv_weight' : 5e-2
         }

         style_transfer(**params1)
```

Content Source Img.

Style Source Img.

```
Iteration 0
```

```
Iteration 100
```

Iteration 199

In [91]:
```
# Scream + Tubingen
params2 = {
    'content_image':'styles/tubingen.jpg',
    'style_image':'styles/the_scream.jpg',
    'image_size':192,
    'style_size':224,
    'content_layer':3,
    'content_weight':3e-2,
    'style_layers':[1, 4, 6, 7],
    'style_weights':[200000, 800, 12, 1],
    'tv_weight':2e-2
}

style_transfer(**params2)
```

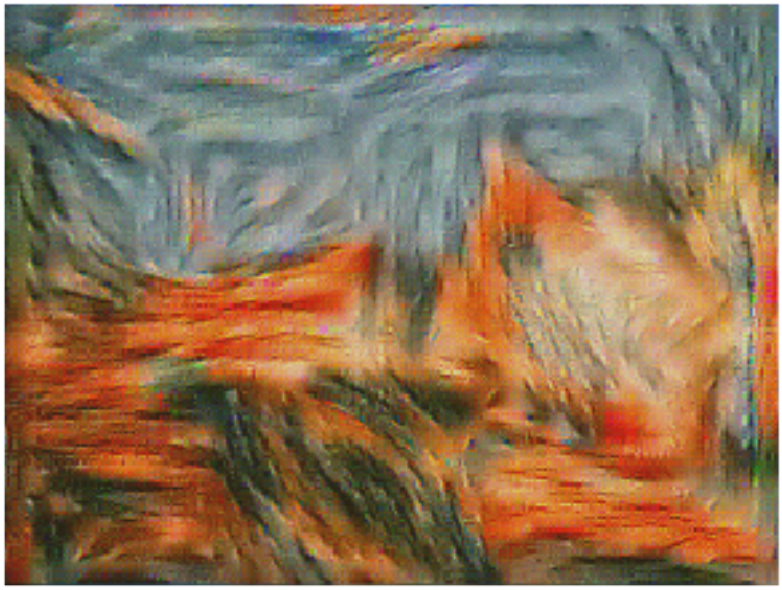
Content Source Img.

Style Source Img.

```
Iteration 0
```

Iteration 100

```
Iteration 199
```

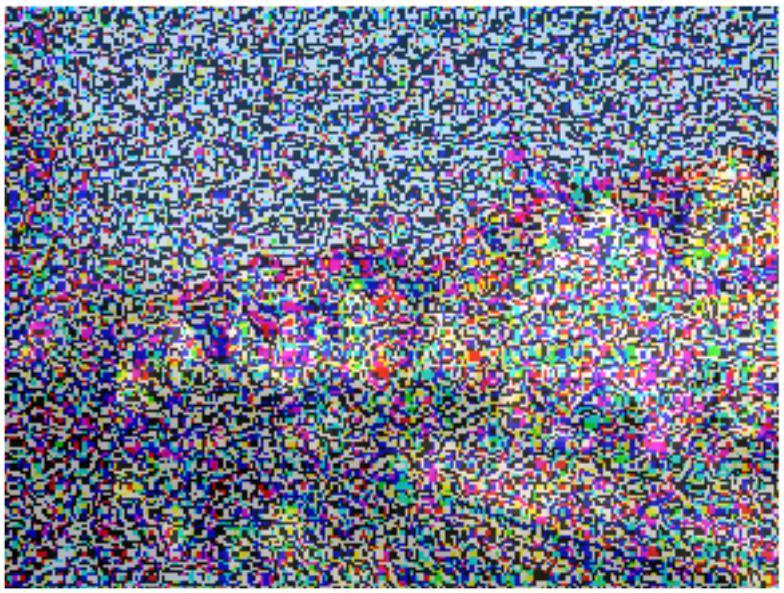


In [92]:
```
# Starry Night + Tubingen
params3 = {
    'content_image' : 'styles/tubingen.jpg',
    'style_image' : 'styles/starry_night.jpg',
    'image_size' : 192,
    'style_size' : 192,
    'content_layer' : 3,
    'content_weight' : 6e-2,
    'style_layers' : [1, 4, 6, 7],
    'style_weights' : [300000, 1000, 15, 3],
    'tv_weight' : 2e-2
}

style_transfer(**params3)
```

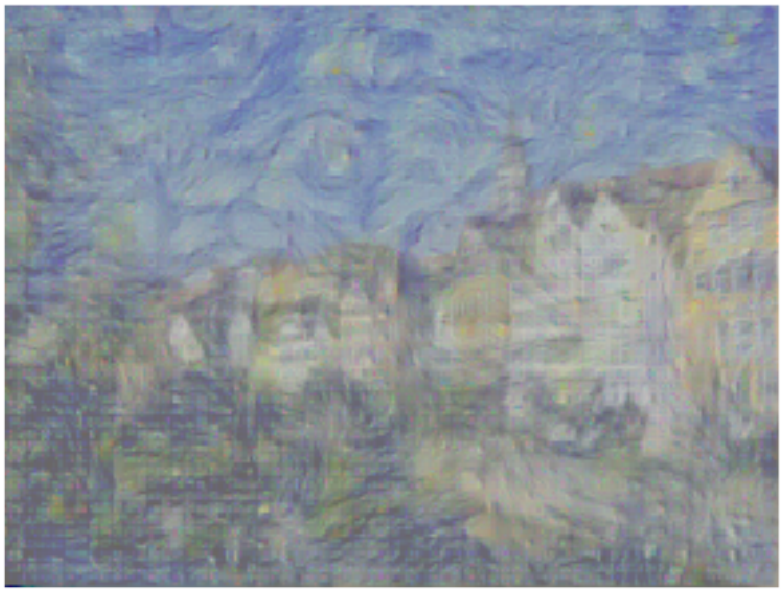
Content Source Img.

Style Source Img.

```
Iteration 0
```



```
Iteration 100
```



Iteration 199

1.7 Feature Inversion (Just run it, 2 pts)

The code you've written can do another cool thing. In an attempt to understand the types of features that convolutional networks learn to recognize, a recent paper [2] attempts to reconstruct an image from its feature representation. We can easily implement this idea using image gradients from the pretrained network, which is exactly what we did above (but with two different feature representations).

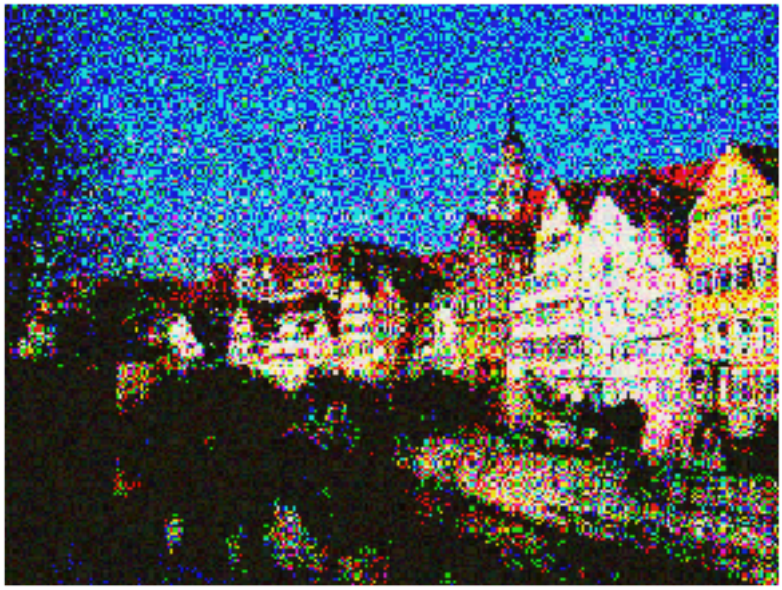
Now, if you set the style weights to all be 0 and initialize the starting image to random noise instead of the content source image, you'll reconstruct an image from the feature representation of the content source image. You're starting with total noise, but you should end up with something that looks quite a bit like your original image.

(Similarly, you could do "texture synthesis" from scratch if you set the content weight to 0 and initialize the starting image to random noise, but we won't ask you to do that here.)

[2] Aravindh Mahendran, Andrea Vedaldi, "Understanding Deep Image Representations by Inverting them", CVPR 2015

```
In [93]: # Feature Inversion -- Starry Night + Tubingen
         params_inv = {
             'content_image' : 'styles/tubingen.jpg',
             'style_image' : 'styles/starry_night.jpg',
             'image_size' : 192,
             'style_size' : 192,
             'content_layer' : 3,
             'content_weight' : 6e-2,
             'style_layers' : [1, 4, 6, 7],
             'style_weights' : [0, 0, 0, 0], # we discard any contributions from style to the
             'tv_weight' : 2e-2,
             'init_random': True # we want to initialize our image to be random
         }

         style_transfer(**params_inv)
```

Iteration 0



Iteration 100

Iteration 199