CS4803/7643: Deep Learning HW2

Problem 1-1

1.1 $X\_padding = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & X_{00} & X_{01} & X_{02} & 0 \\ 0 & X_{10} & X_{11} & X & 0 \\ 0 & X_{20} & X_{21} & X_{22} & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$, with stride = 2, pad = 1,

Using Formula: size = $[(\text{original size} - \text{kernel size} + 2P)/S] + 1$

$= (3 - 3 + 2 \times 1)/2 + 1 = 2$

$Y_{\text{before flatten}} = \begin{bmatrix} W_{11}X_{00} + W_{12}X_{01} + W_{21}X_{10} + W_{22}X_{11}, & X_{01}W_{10} + W_{11}X_{02} + W_{20}X_{11} + W_{21}X_{12} \\ W_{01}X_{10} + W_{22}X_{11} + W_{11} + X_{20} + W_{22}X_{21}, & W_{00}X_{11} + W_{01}X_{22} + W_{10}X_{21} + W_{11}X_{22} \end{bmatrix}$

$X = [X_{00}, X_{01}, X_{02}, X_{10}, X_{11}, X_{12}, X_{20}, X_{21}, X_{22}]$.

$A = \begin{bmatrix} W_{11} & W_{12} & 0 & W_{21} & W_{22} & 0 & 0 & 0 & 0 \\ 0 & W_{10} & W_{11} & 0 & W_{20} & W_{21} & 0 & 0 & 0 \\ 0 & 0 & 0 & W_{01} & W_{02} & 0 & W_{22} & W_{12} & 0 \\ 0 & 0 & 0 & 0 & W_{00} & W_{01} & 0 & W_{10} & W_{11} \end{bmatrix}$

$Y_{\text{flatten}} = \begin{bmatrix} W_{11}X_{00} + W_{12}X_{01} + W_{21}X_{10} + W_{22}X_{11}, \\ W_{10}X_{01} + W_{11}X_{02} + W_{20}X_{11} + W_{21}X_{12}, \\ W_{01}X_{10} + W_{22}X_{11} + W_{11} + X_{20} + W_{22}X_{11}, \\ W_{00}X_{11} + W_{01}X_{22} + W_{10}X_{21} + W_{11}X_{22} \end{bmatrix}$

Problem 1-2

1.2 stride=2, padding=0, W=2, size after convolution = 4

$$Y = \begin{bmatrix} W_{00}X_{00} & W_{01}X_{00} & W_{00}X_{01} & W_{01}X_{01} \\ W_{10}X_{00} & W_{11}X_{00} & W_{10}X_{01} & W_{11}X_{01} \\ W_{00}X_{10} & W_{01}X_{10} & W_{00}X_{11} & W_{01}X_{11} \\ W_{10}X_{10} & W_{11}X_{10} & W_{10}X_{11} & W_{11}X_{11} \end{bmatrix}$$

$$A = \begin{bmatrix} W_{00} & 0 & 0 & 0 \\ W_{01} & 0 & 0 & 0 \\ 0 & W_{00} & 0 & 0 \\ 0 & W_{01} & 0 & 0 \\ W_{10} & 0 & 0 & 0 \\ W_{11} & 0 & 0 & 0 \\ 0 & W_{10} & 0 & 0 \\ 0 & W_{11} & 0 & 0 \\ 0 & 0 & W_{00} & 0 \\ 0 & 0 & W_{01} & 0 \\ 0 & 0 & 0 & W_{00} \\ 0 & 0 & 0 & W_{01} \\ 0 & 0 & W_{10} & 0 \\ 0 & 0 & W_{11} & 0 \\ 0 & 0 & 0 & W_{10} \\ 0 & 0 & 0 & W_{11} \end{bmatrix}$$

Problem2-1

2.

2.1 Assume $W=[W_1\ W_2]$, provided with $\begin{cases} b<0 \\ W_2+b<0 \\ W_1+b<0 \\ W_1+W_2+b\geq 0 \end{cases}$

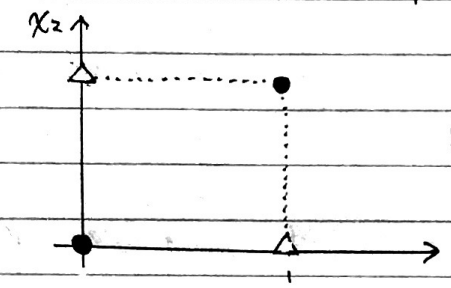
From truth table of OR

We can deduce $\begin{cases} W_1, W_2, b>0 \\ W_1+W_2>|b|,\ |b|>W_1, W_2 \end{cases}$

Since we set $b_{AND}=-0.5$, $W_{AND}=[0.6, 0.6]$

Problem 2-2

2.2 If we give the visual representation, as follows



Since the △ and ● are linear inseparable, ⇔ cannot be represented by linear model.

Problem3-1

3.

3.1 $W_{ij}^{(1)} = \begin{bmatrix} 2 & 0 & \cdots & 0 \\ 0 & 2 & & \\ \vdots & & \ddots & \\ 0 & & & 2 \end{bmatrix}_{d \times d}$ $f_1(x) = \begin{bmatrix} |2x_1 - 1| \\ |2x_2 - 1| \\ \vdots \\ |2x_d - 1| \end{bmatrix}_{d \times 1}$

For each dimension, We could find $R = \{(0, \frac{1}{2}), (\frac{1}{2}, 1)\}$ onto $O = (0, 1)$,
so, For d dimension, we have 2^d output regions.

Problem3-2

3.2

With combined function, we get $n_g \times n_f$ regions onto $O(0,1)^d$
because each region in n_g create n_f region

Problem3-3

3.3 With the results of 3.1, when range = (0,1), each layer identifies 2^d regions, combined with the result of 3.2, we know if we ranges, region of original function will also scaling, so the answer should be

$$\Rightarrow \prod_{i=1}^{L} 2^d = 2^{dL} \quad \#$$

Problem5-1

In my conjectures, the reason that deep learning is robust to the noise is that between the deep structure, activation function may filter out the ambiguous sample which with noisy labels. Since the ambiguous samples are filtering out, the gradient of state out art model won't be affected and could predict the actual class regardless the noise. Andrew Ng also observed the phenomenon in the CheXNet that the Densenet152 process the ability to optimize the wrong label.

Ref: CheXNet: Radiologist-Level Pneumonia Detection on Chest X-Rays with Deep Learning

Problem5-2

If we regard the problem as the optimization question, take convolutional neural network as example, the similar categories usually share low level features like geometry and texture, which makes their gradient similar to each other when passing by the low level layer. Hence, in my opinion, the optimal point of similar categories may close to each other in high dimensional spaces which makes generalization easier.

NetworkVisualization-PyTorch

February 26, 2020

1 Network Visualization (45 Points)

In the first part of the notebook we will explore the use of different type of attribution algorithms - both gradient and perturbation - for images, and understand their differences using the Captum model interpretability tool for PyTorch.

Link to the Website: https://captum.ai/ Link to the Github page: https://github.com/pytorch/captum

As an exercise you'll be also asked to implement Saliency Maps from scratch.

- Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps", ICLR Workshop 2014.
- Mukund Sundararajan, Ankur Taly, Qiqi Yan, "Axiomatic Attribution for Deep Networks", ICML, 2017
- Matthew D Zeiler, Rob Fergus, "Visualizing and Understanding Convolutional Networks", Visualizing and Understanding Convolutional Networks, 2013.
- Ramprasaath R. Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, Dhruv Batra, Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization, 2016

For the full list of available attribution algorithms please check out: https://captum.ai/api/

In the second and third parts we will focus on generating new images, by studying and implementing key components in two papers: * 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 papers, 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 homework 2. We will explore four different techniques:

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. **GradCAM**: GradCAM is a way to show the focus area on an image for a given label.
3. **Fooling Images**: We can perturb an input image so that it appears the same to humans, but will be misclassified by the pretrained network.
4. **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 1.4** 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
```

- To install captum (please install the latest version 0.2.0 from source - for more information see https://github.com/pytorch/captum),

```
git clone https://github.com/pytorch/captum.git
cd captum
pip install -e .
```

- The total credit for this notebook is 45 points - 10 points for each section, and 5 points for the captum attribution calls.
- Although we will run your notebook in grading, you still need to **submit the notebook with all the outputs you generated, in pdf form**. Sometimes it will inform us if we get any inconsitent results with respect to yours.

[1]:
```
import torch
from torch.autograd import Variable
from torch.autograd import Function as TorchFunc
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
import matplotlib
from cs7643.image_utils import SQUEEZENET_MEAN, SQUEEZENET_STD
from PIL import Image
import captum

%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'
print(captum.__version__)
print(torch.__version__)
```

```
0.2.0
1.3.1
```

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.

[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
```

```
def visualize_attr_maps(attributions, titles, attr_preprocess=lambda attr: attr.
↪permute(1, 2, 0).detach().numpy(),
                        cmap='viridis', alpha=0.7):
    '''
    A helper function to visualize captum attributions for a list of captum␣
↪attribution algorithms.

    attributions(A list of torch tensors): Each element in the attributions␣
↪list corresponds to an
                        attribution algorithm, such an Saliency, Integrated␣
↪Gradient, Perturbation, etc.
                        Each row in the attribution tensor contains
    titles(A list of strings): A list of strings, names of the attribution␣
↪algorithms corresponding to each element in
                        the `attributions` list. len(attributions) == len(titles)
    '''
    N = attributions[0].shape[0]
    plt.figure()
    for i in range(N):
        axs = plt.subplot(len(attributions) + 1, N + 1, i+1)
        plt.imshow(X[i])
        plt.axis('off')
        plt.title(class_names[y[i]])

    plt.subplot(len(attributions) + 1, N + 1 , N + 1)
    plt.text(0.0, 0.5, 'Original Image', fontsize=14)
    plt.axis('off')
    for j in range(len(attributions)):
        for i in range(N):
            plt.subplot(len(attributions) + 1, N + 1 , (N + 1) * (j + 1) + i +␣
↪1)
            attr = np.array(attr_preprocess(attributions[j][i]))
            attr = (attr - np.mean(attr)) / np.std(attr).clip(1e-20)
            attr = attr * 0.2 + 0.5
            attr = attr.clip(0.0, 1.0)
            plt.imshow(attr, cmap=cmap, alpha=alpha)
            plt.axis('off')
        plt.subplot(len(attributions) + 1, N + 1 , (N + 1) * (j + 1) + N + 1)
        plt.text(0.0, 0.5, titles[j], fontsize=14)
        plt.axis('off')

    plt.gcf().set_size_inches(20, 13)
    plt.show()

def compute_attributions(algo, inputs, **kwargs):
    '''
```

```
    A common function for computing captum attributions
    '''
    return algo.attribute(inputs, **kwargs)
```

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

[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
```

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 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.

[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.

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.

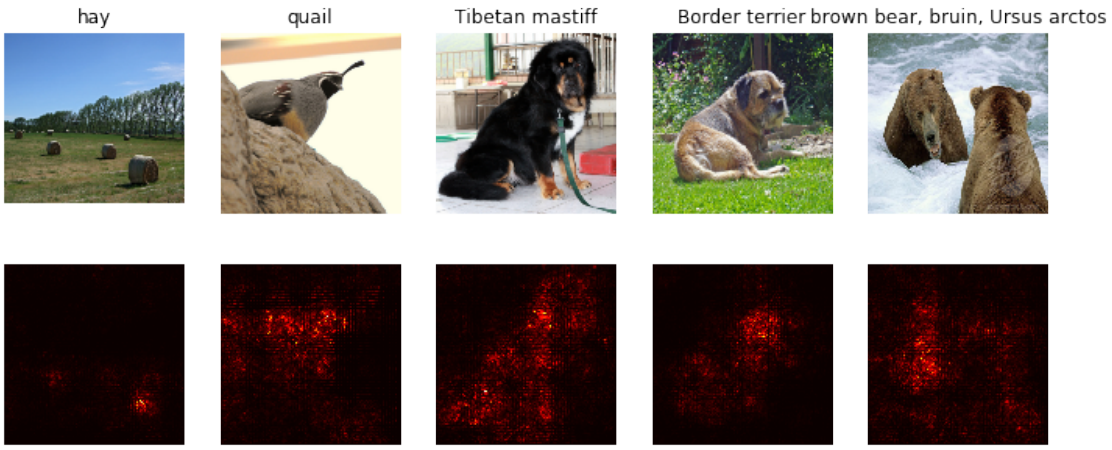
[5]:
```
# Example of using gather to select one entry from each row in PyTorch
def gather_example():
    N, C = 4, 5
```

```
    s = torch.randn(N, C)
    y = torch.LongTensor([1, 2, 1, 3])
gather_example()
```

[6]:
```
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.             ␣
↪ #
  ␣
↪##############################################################################
    scores = model.forward(X_var)
    scores = scores.gather(1, y_var.view(-1,1)).squeeze()
    grad = scores.backward(torch.Tensor([1., 1., 1., 1., 1.])) # tensor_grad␣
↪means the weight of the current gradient

    saliency = abs(X_var.grad.data)
    saliency, _ = torch.max(saliency, dim=1)
```

```
␣
↪#############################################################################
    #                                END OF YOUR CODE                          ␣
↪ #
␣
↪#############################################################################
    return saliency
```

[7]:
```
def show_saliency_maps(X, y):

    X_tensor = torch.cat([preprocess(Image.fromarray(x)) for x in X], dim=0)
    y_tensor = torch.LongTensor(y)

    saliency = compute_saliency_maps(X_tensor, y_tensor, model)
    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.hot)
        plt.axis('off')
        plt.gcf().set_size_inches(12, 5)
    plt.show()

show_saliency_maps(X, y)
```



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.

3.1 Captum (1 pt)

As a final step, we will show you how simple it is to use Saliency Maps using Captum instead.

Captum offers a number of attribution algorithms for PyTorch models that are very easy to use. Let's apply those algorithms to our model and the five images that we loaded from imagenet.

We have created a generic helper visualization function that will allow you to visualize and compare the attributions of different algorithms next to each other for each image. (`compute_attributions` and `visualize_attr_maps`, found in the helper functions section)

Let's apply the saliency maps attribution algorithm on the images and observe how the attributions differ.

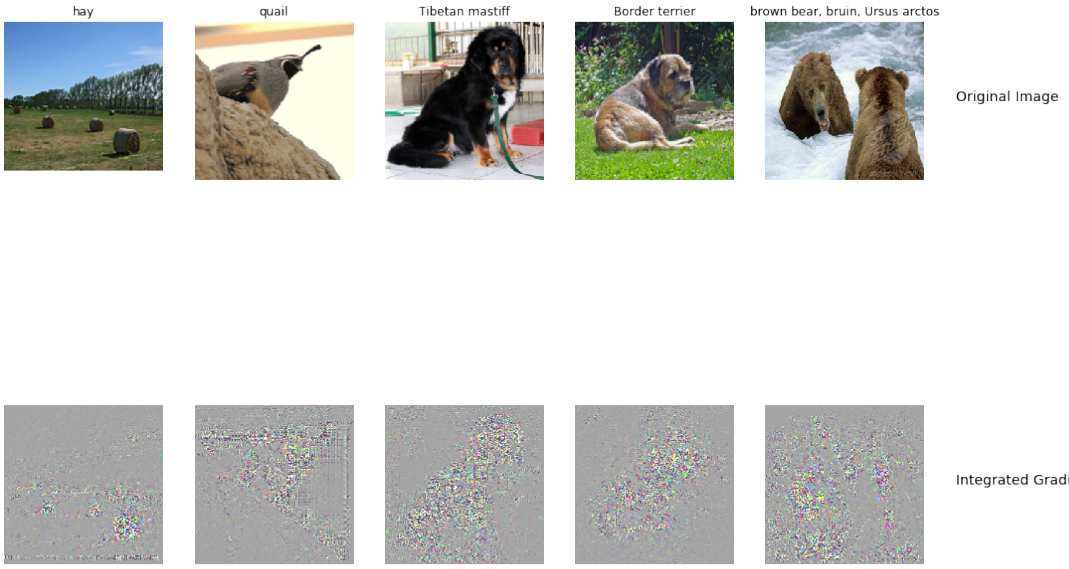
To do so we need to import those algorithms from the Captum library, create instances of the corresponding algorithms, call our compute_attribute function on these instances, and finally visualize using our helper function.

We have included an example of how we can apply a number of attribution algorithms on our model and images.

Feel free to try other algorithms and compare their results.

Please, be aware that some of the algorithms, such as perturbation algorithms, might take longer to execute and might have higher memory requirments. In case you run into OOM issues for integrated gradients you can try to reduce the number of integral approximation steps (n_steps) or set the value of internal_batch_size input argument to a small number. The value of `ablations_per_eval` or `perturbation_per_eval` can be adjusted also for all perturbation algorithms in order to reduce memory footprint. This might lead to a slower execution runtime, however it will help to avoid OOM.

[8]:
```
from captum.attr import IntegratedGradients, Saliency


# 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)


# Computing Integrated Gradient
int_grads = IntegratedGradients(model)
attr_ig = compute_attributions(int_grads, X_tensor, target=y_tensor, n_steps=10)

################################################################################
# TODO: Compute/Visualize Saliency using captum.                               #
################################################################################
visualize_attr_maps([attr_ig], ['Integrated Gradients'])
################################################################################
#                               END OF YOUR CODE                               #
################################################################################
```



4 GradCAM (10 pts)

GradCAM (which stands for Gradient Class Activation Mapping) is a technique that tells us where a convolutional network is looking when it is making a decision on a given input image. There are three main stages to it:

- **Guided Backprop (Changing ReLU Backprop Layer, https://arxiv.org/abs/1412.6806)**
- **GradCAM (Manipulating gradients at the last convolutional layer, https://arxiv.org/abs/1610.02391)**
- **Guided GradCAM (Pointwise multiplication of above stages)**

In this section, you will be implementing these three stages to recreate the full GradCAM pipeline. At each stage, you can visualize what the current output is to see exactly what is happening.

We begin with Guided Backprop. We encourage you to read the paper above first, to gain an understanding of what Guided Backprop is trying to do. From the paper, we have that: * The 'deconvolution' is equivalent to a backward pass through the network, except that when propagating through a nonlinearity, its gradient is solely computed based on the top gradient signal, ignoring the bottom input. In case of the ReLU nonlinearity this amounts to setting to zero certain entries based on the top gradient. We propose to combine these two methods: rather than masking out values corresponding to negative entries of the top gradient ('deconvnet') or bottom data (backpropagation), we mask out the values for which at least one of these values is negative.

[9]:
```
# FOR THIS SECTION ONLY, we need to use gradients. We introduce a new model we␣
↪will use explicitly for GradCAM for this.
gc_model = torchvision.models.squeezenet1_1(pretrained=True)
for param in gc_model.parameters():
    param.requires_grad = True
```

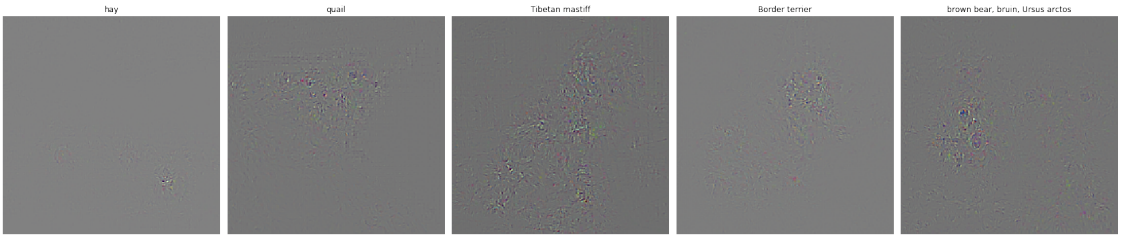
[10]:
```
class CustomReLU(TorchFunc):
    """
    Define the custom change to the standard ReLU function necessary to perform␣
↪guided backpropagation.
    We have already implemented the forward pass for you, as this is the same␣
↪as a normal ReLU function.
    """

    @staticmethod
    def forward(self, x):
        output = torch.addcmul(torch.zeros(x.size()), x, (x > 0).type_as(x))
        self.save_for_backward(x, output)
        return output

    @staticmethod
    def backward(self, y):
        ␣
↪#############################################################################
        # TODO: Implement this function. Perform a backwards pass as described␣
↪in    #
        # paper above. Note: torch.addcmul might be useful, and you can access ␣
↪     #
        # the input/output from the forward pass with self.saved_tensors.      ␣
↪     #
        ␣
↪#############################################################################
        x, output = self.saved_tensors
        gradient = torch.addcmul(torch.zeros(y.size()), y, (y>0).type_as(y))
        gradient[x <= 0] = 0

        return gradient
        ␣
↪#############################################################################
        #                                END OF YOUR CODE                      ␣
↪     #
        ␣
↪#############################################################################
```

To test your implementation, run the code below.

[11]:
```
for idx, module in gc_model.features._modules.items():
    if module.__class__.__name__ == 'ReLU':
```

```
        gc_model.features._modules[idx] = CustomReLU.apply

def guided_backprop(X_tensor,y_tensor):
    ␣
 ↪#############################################################################
    # TODO: Implement guided backprop as described in paper.                  ␣
 ↪ #
    # (Hint): Now that you have implemented the custom ReLU function, this    ␣
 ↪ #
    # method will be similar to a single training iteration.                  ␣
 ↪ #
    ␣
 ↪#############################################################################
    X_var = Variable(X_tensor, requires_grad=True)
    y_var = Variable(y_tensor, requires_grad=False)
    score = gc_model(X_var)
    gradient_weight = torch.FloatTensor(score.size()[0], score.size()[-1]).
 ↪zero_()
    # Only the gradient of the target class are left
    gradient_weight[np.arange(score.size()[0]), y_var] = 1

    model.zero_grad()
    score.backward(gradient_weight)

    guided_backprop = X_var.grad.data.numpy()
    for i in range(y_var.size()[0]):
        guided_backprop[i] -= guided_backprop[i].min()
        guided_backprop[i] /= guided_backprop[i].max()
    guided_backprop = np.transpose(guided_backprop, (0,2,3,1))

    return guided_backprop

    ␣
 ↪#############################################################################
    #                               END OF YOUR CODE                          ␣
 ↪ #
    ␣
 ↪#############################################################################

X_tensor = torch.cat([preprocess(Image.fromarray(x)) for x in X], dim=0).
 ↪requires_grad_(True)
y_tensor = torch.LongTensor(y)
gbp_result = guided_backprop(X_tensor,y_tensor)

plt.figure(figsize=(24, 24))
for i in range(gbp_result.shape[0]):
```

```
    plt.subplot(1, 5, i + 1)
    plt.imshow(gbp_result[i])
    plt.title(class_names[y[i]])
    plt.axis('off')
plt.gcf().tight_layout()
```



Next, we can implement GradCAM. We have given you which module(=layer) that we need to capture gradients from, which you can see in `conv_module` variable below - feel free to play around with this to see visualizations for different layers, but in your final submission keep it to what we gave you. We have already provided a gradient and activation hook for you, and the gradient value of the module you choose will be stored in the `gradient_value` variable (similarly `activation_value` will hold the layer activation). The rest of the implementation of GradCAM is up to you.

[12]:
```
# Reset model from any changes made during Guided Backprop
gc_model = torchvision.models.squeezenet1_1(pretrained=True)
for param in gc_model.parameters():
    param.requires_grad = True
```

[13]:
```
conv_module = gc_model.features[12]

gradient_value = None # Stores gradient of the module you chose above during a␣
 ↪backwards pass.
activation_value = None # Stores the activation of the module you chose above␣
 ↪during a forwards pass.

def gradient_hook(a,b,gradient):
    global gradient_value
    gradient_value = gradient[0]

def activation_hook(a,b,activation):
    global activation_value
    activation_value = activation

conv_module.register_forward_hook(activation_hook)
conv_module.register_backward_hook(gradient_hook)
```

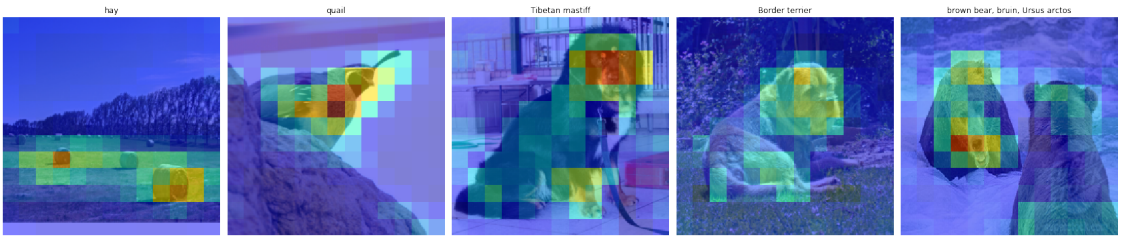
[13]:
```
<torch.utils.hooks.RemovableHandle at 0x7fedd7313240>
```

[20]:
```
def grad_cam(X_tensor, y_tensor):
    ␣
↪#############################################################################
    # TODO: Implement GradCam as described in paper.                          ␣
↪ #
    ␣
↪#############################################################################
    X_var = Variable(X_tensor, requires_grad=True)
    y_var = Variable(y_tensor, requires_grad=False)
    score = gc_model(X_var)

    gradient_weight = torch.FloatTensor(score.size()[0], score.size()[-1]).
↪zero_()
    # Only the gradient of the target class are left
    gradient_weight[np.arange(score.size()[0]), y_var] = 1

    model.zero_grad()
    score.backward(gradient_weight, retain_graph=True)


    # global averaging gradient as weight of each activation map
    guided_gradients = gradient_value.data.numpy()
    activations = activation_value.data.numpy()
    weights = np.mean(guided_gradients, axis=(2,3))

    cam = np.ones((guided_gradients.shape[0],guided_gradients.shape[2],␣
↪guided_gradients.shape[3]), dtype=np.float32)
    # weight X activation map
    for i in range(score.shape[0]):
        weight = weights[i]
        c = cam[i]
        act = activations[i]
        for j, w in enumerate(weight):
            c += w * act[j,:,:]
    # relu
    cam = np.maximum(cam,0)
    ␣
↪#############################################################################
    #                              END OF YOUR CODE                           ␣
↪ #
    ␣
↪#############################################################################

    # Rescale GradCam output to fit image.
    cam_scaled = []
    for i in range(cam.shape[0]):
```

```
        cam_scaled.append(np.array(Image.fromarray(cam[i]).resize(X_tensor[i,0,:
↪,:].shape)))
    cam = np.array(cam_scaled)
    cam -= np.min(cam)
    cam /= np.max(cam)
    return cam
```

To test your implementation, run the code below.

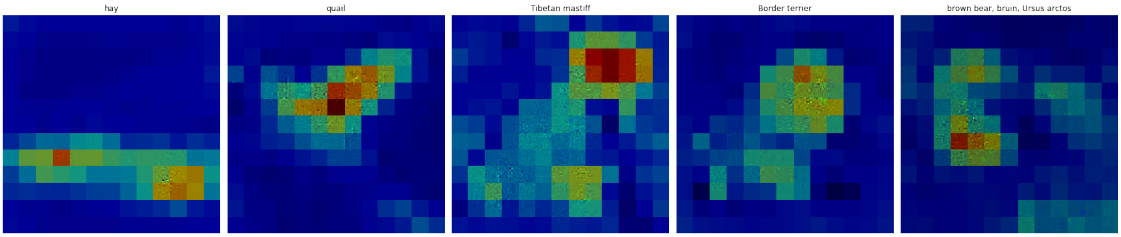
[21]:
```
X_tensor = torch.cat([preprocess(Image.fromarray(x)) for x in X], dim=0).
↪requires_grad_(True)
y_tensor = torch.LongTensor(y)
gradcam_result = grad_cam(X_tensor, y_tensor)

plt.figure(figsize=(24, 24))
for i in range(gradcam_result.shape[0]):
    gradcam_val = gradcam_result[i]
    img = X[i] + (matplotlib.cm.jet(gradcam_val)[:,:,:3]*255)
    img = img / np.max(img)
    plt.subplot(1, 5, i + 1)
    plt.imshow(img)
    plt.title(class_names[y[i]])
    plt.axis('off')
plt.gcf().tight_layout()
```



As a final step, we can combine GradCam and Guided Backprop to get Guided GradCam.

[22]:
```
X_tensor = torch.cat([preprocess(Image.fromarray(x)) for x in X], dim=0).
↪requires_grad_(True)
y_tensor = torch.LongTensor(y)
gradcam_result = grad_cam(X_tensor, y_tensor)
gbp_result = guided_backprop(X_tensor,y_tensor)

plt.figure(figsize=(24, 24))
for i in range(gradcam_result.shape[0]):
    gbp_val = gbp_result[i]
    gbp_val /= np.max(gbp_val)
    gradcam_val = (matplotlib.cm.jet(gradcam_result[i])[:,:,:3]*255)
```

```
  ␣
↪##############################################################################
    # TODO: Pointwise multiplication and normalization of the gradcam and␣
↪guided #
    #       backprop results (2 lines)                                         ␣
↪ #
  ␣
↪##############################################################################
    img = gradcam_val * gbp_val
    img /= np.max(ggc)
  ␣
↪##############################################################################
    #                               END OF YOUR CODE                           ␣
↪ #
  ␣
↪##############################################################################

    img = np.expand_dims(img.transpose(2,0,1),axis=0)
    img = np.float32(img)
    img = torch.from_numpy(img)
    img = deprocess(img)
    plt.subplot(1, 5, i + 1)
    plt.imshow(img)
    plt.title(class_names[y[i]])
    plt.axis('off')
plt.gcf().tight_layout()
```



4.1 Captum (1 pt)

As a final step, implement GradCam and GuidedBackprop exactly as you did for saliency maps above and compare your visualizations with the ones using Captum (note: **These visualization will look significantly different, as Captum has different pre/post processing steps for images**).

[26]:
```
from captum.attr import GuidedGradCam, GuidedBackprop


# 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)

################################################################################
# TODO: Compute/Visualize GuidedBackprop and GradCAM as well.                  #
################################################################################
X_var = Variable(X_tensor, requires_grad=True)
conv = model.features[9]
guided_bp = GuidedBackprop(model)
attr_gbp = compute_attributions(guided_bp, X_var, target=y_tensor)
ggc = GuidedGradCam(model, layer=conv)
attr_ggc = compute_attributions(ggc, X_var, target=y_tensor)
visualize_attr_maps([attr_gbp, attr_ggc], ['GuidedBackprop', 'GuidedGradCam'])
################################################################################
#                              END OF YOUR CODE                                #
################################################################################
```

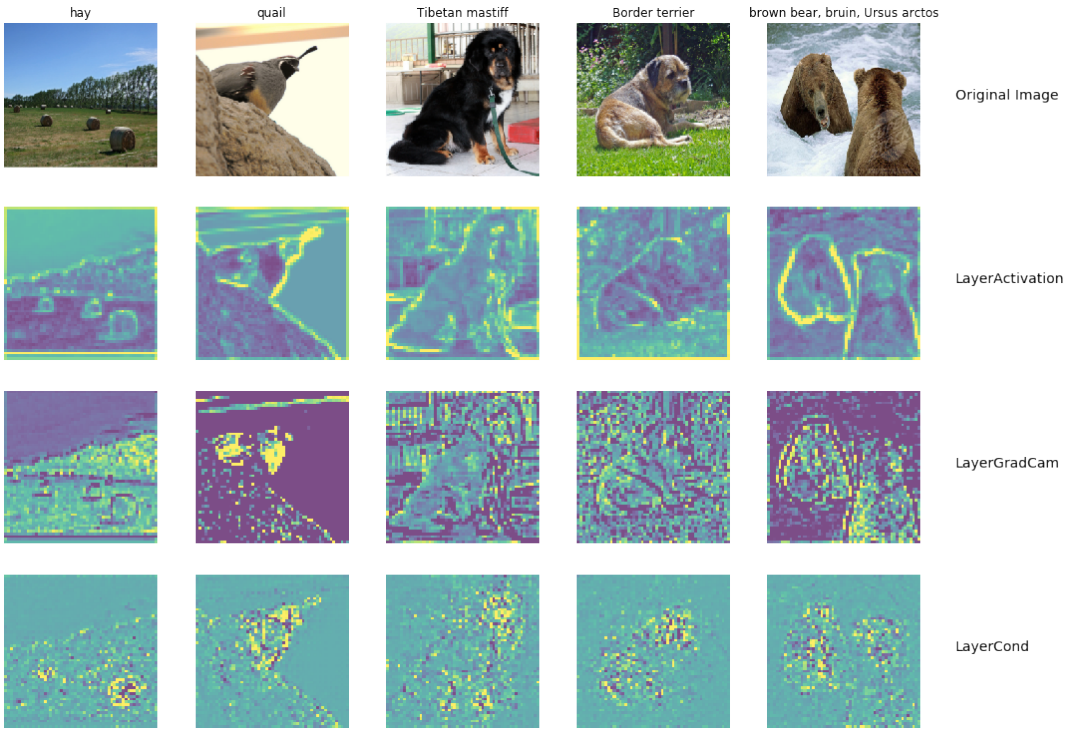
4.2 Visualizaing layers and neurons using Captum (3 pts)

Let's try to attribute to a selected layer and visualize the attribution for any selected channel. We can choose to change the layers and channels and observe how the attribution changes.

Consider also using https://captum.ai/api/_modules/captum/attr/_utils/attribution.html#LayerAttributio to interpolate layer dimensions to given input dimensions.

Please, be aware of the memory limitation and how you can overcome those using the techinques described in the previous section.

[33]:
```
from captum.attr import LayerActivation, LayerConductance, LayerGradCam,␣
↪LayerAttribution

# Try out different layers and see observe how the attributions change
layer = model.features[3]

layer_act = LayerActivation(model, layer)
layer_act_attr = compute_attributions(layer_act, X_tensor)
layer_act_attr_sum = layer_act_attr.mean(axis=1, keepdim=True)

##############################################################################
# TODO: Visualize Individual Layer Gradcam and Layer Conductance (similar    #
# to what we did for the other captum sections, using our helper methods),   #
# but with some preprocessing calculations.                                  #
##############################################################################

X_var = Variable(X_tensor, requires_grad=True)
gc_per_layer = LayerGradCam(model, layer)
attr_gradcam = gc_per_layer.attribute(X_var, target=y_tensor,␣
↪relu_attributions=True)
attr_gradcam = LayerAttribution.interpolate(attr_gc, (X_tensor.shape[2],␣
↪X_tensor.shape[3]))

layer_conductance = LayerConductance(model, layer)
conductance = layer_conductance.attribute(X_var, target=y_tensor)
conductance = conductance.mean(axis=1, keepdim=True)
conductance = LayerAttribution.interpolate(conductance, (X_tensor.shape[2],␣
↪X_tensor.shape[3]))
visualize_attr_maps([layer_act_attr_sum, attr_gradcam, conductance],
                    ['LayerActivation', 'LayerGradCam', 'LayerCond'],
                    attr_preprocess=lambda attr: attr.permute(1, 2, 0).detach().
↪numpy().squeeze())
##############################################################################
#                              END OF YOUR CODE                              #
##############################################################################
```



5 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

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.

[35]:
```
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.                   ␣
↪ #
```

```
  ␣
↪##############################################################################␣
↪
        scores = model(X_fooling_var)
        _, pred = torch.max(scores, dim = 1)

        if pred == target_y:
            break
        scores[:, target_y].backward()
        grad = X_fooling_var.grad.data
        #create noise
        leaf = learning_rate * grad / torch.norm(grad)
        X_fooling_var.data += leaf.data
        X_fooling_var.grad.data.zero_()



  ␣
↪##############################################################################
    #                                END OF YOUR CODE                            ␣
↪ #
  ␣
↪##############################################################################

    X_fooling = X_fooling_var.data

    return X_fooling
```

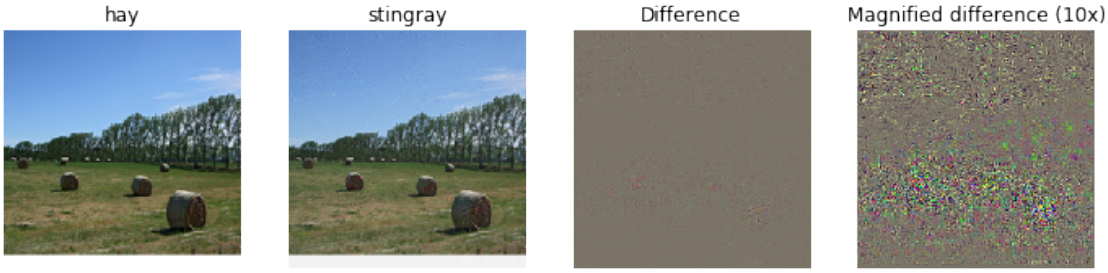
Now you can run the following cell to **generate a fooling image**. You will see the message 'Fooled the model' when you succeed.

[36]:
```
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!')
```

```
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.

[37]:
```
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()
```



6 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

[3] Yosinski et al, "Understanding Neural Networks Through Deep Visualization", ICML 2015 Deep Learning Workshop

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

[38]:
```
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
```

[49]:
```
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
```

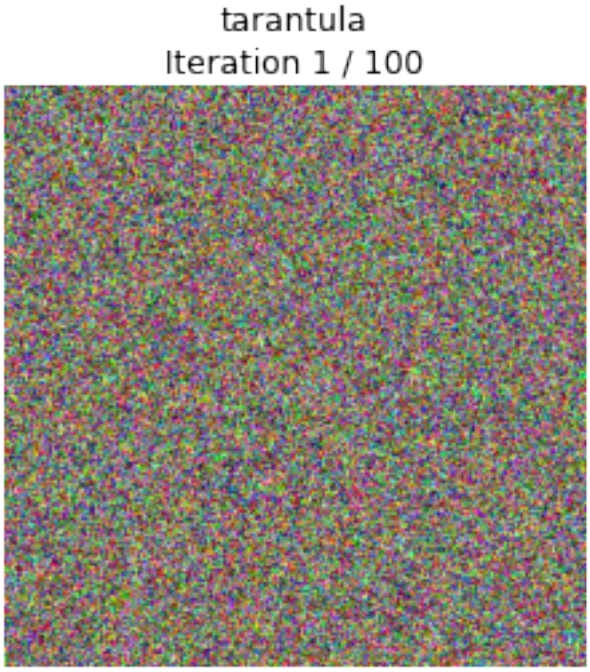
```
    - model: A pretrained CNN that will be used to generate the image
    - dtype: Torch datatype to use for computations

    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.data.copy_(jitter(img.data, 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.            ␣
↪#
        ␣
↪##############################################################################
```

```
        img.requires_grad = True
        preds = model(img)
        s_y = preds[:, target_y]
        pred = s_y - (l2_reg * torch.norm(img, p=2))

        pred.backward()

        grad = img.grad.data
        leaf = learning_rate * grad / torch.norm(grad, p=2)
        img.data += leaf.data
        img.grad.data.zero_()

        img.requires_grad = False

      ␣
↪##############################################################################
        #                                END OF YOUR CODE                             ␣
↪#
      ␣
↪##############################################################################

        # Undo the random jitter
        img.data.copy_(jitter(img.data, -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())
```

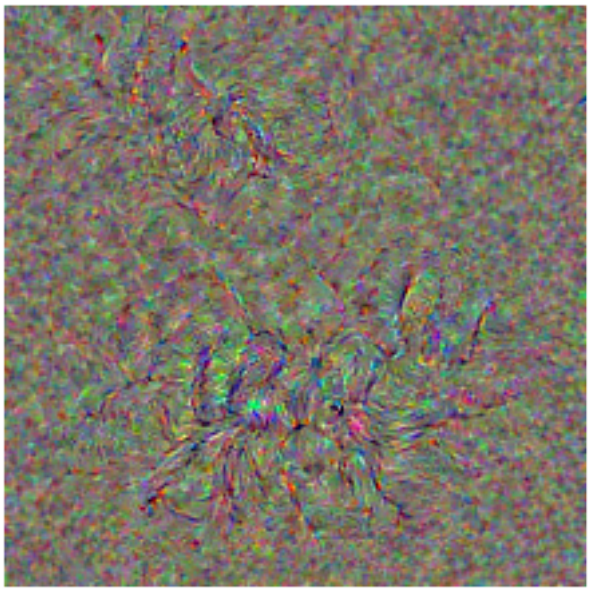
Once you have completed the implementation in the cell above, run the following cell to generate images of several classes. Show the generated images when you submitted your notebook.

[50]:
```
dtype = torch.FloatTensor
# dtype = torch.cuda.FloatTensor # Uncomment this to use GPU
```

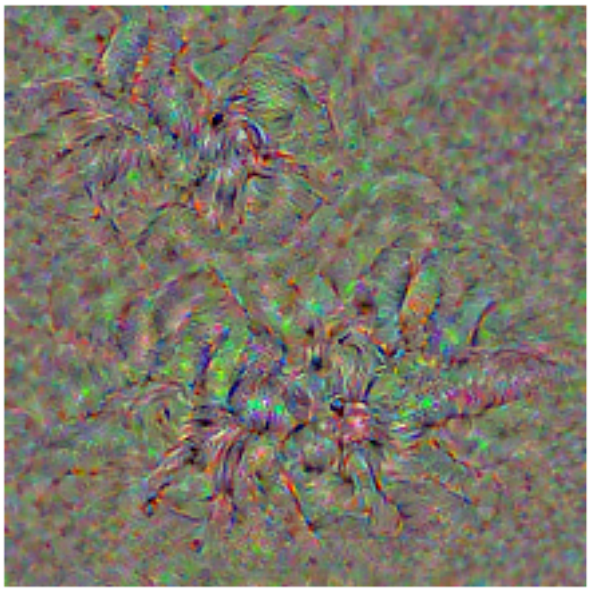
```
model.type(dtype)

# 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)
```



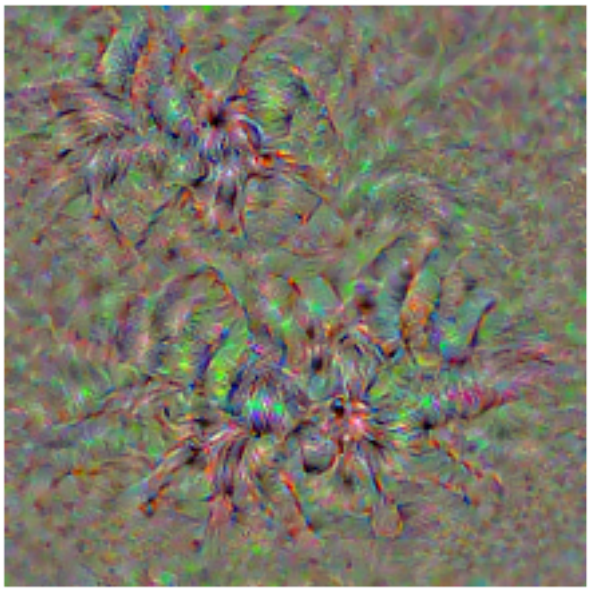
tarantula
Iteration 25 / 100



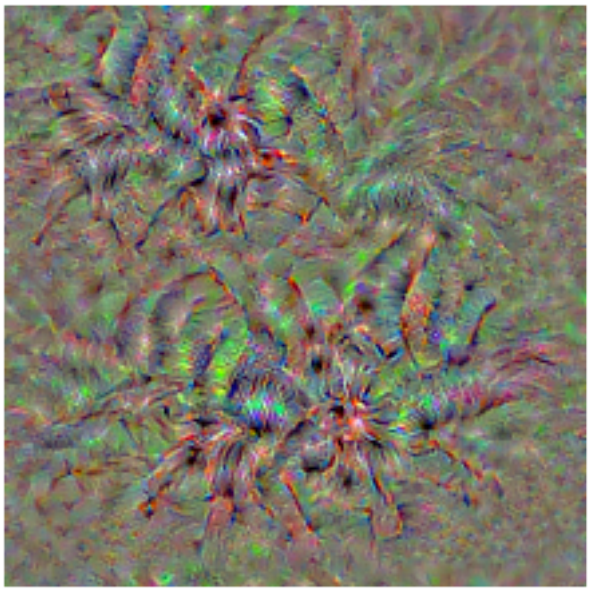
tarantula
Iteration 50 / 100



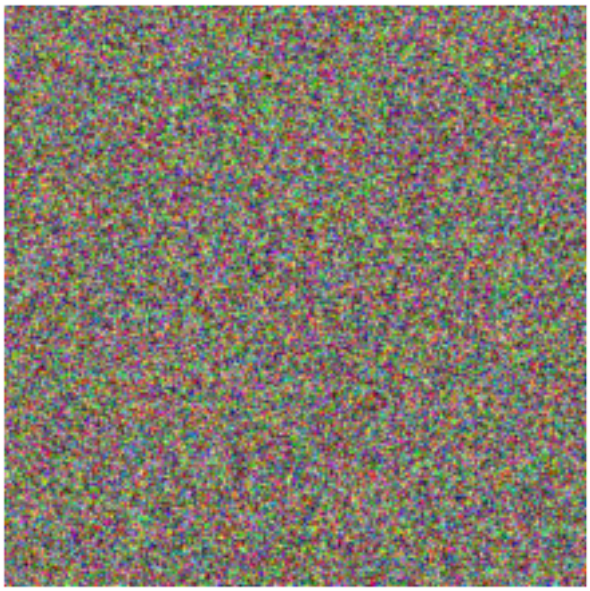
tarantula
Iteration 75 / 100



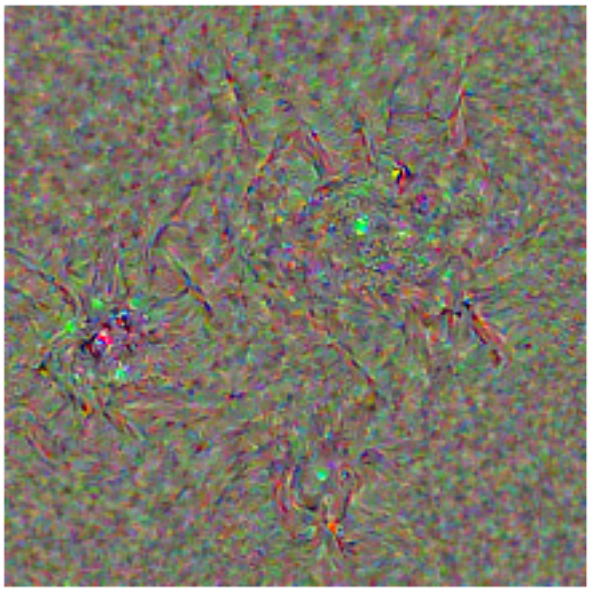
tarantula
Iteration 100 / 100



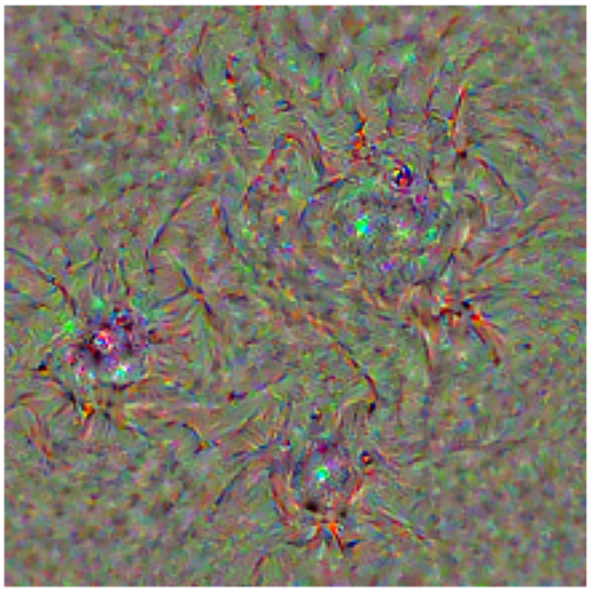
tick
Iteration 1 / 100



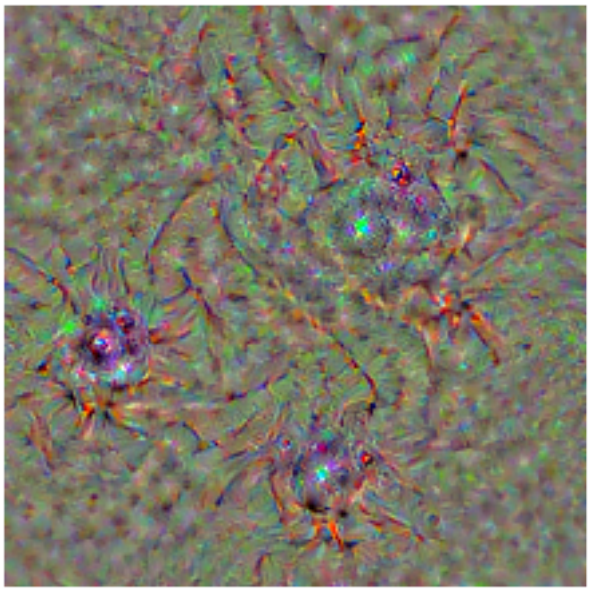
tick
Iteration 25 / 100



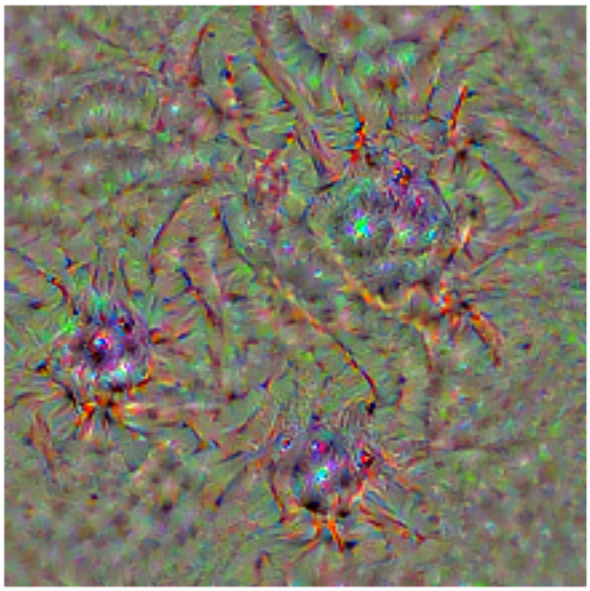
tick
Iteration 50 / 100



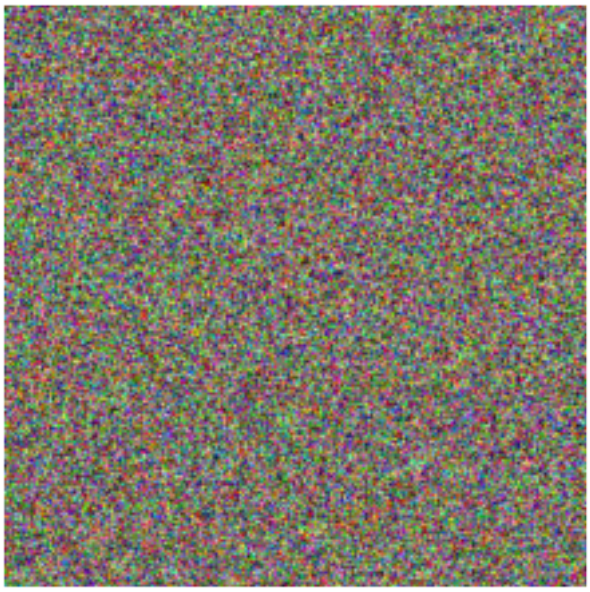
tick
Iteration 75 / 100



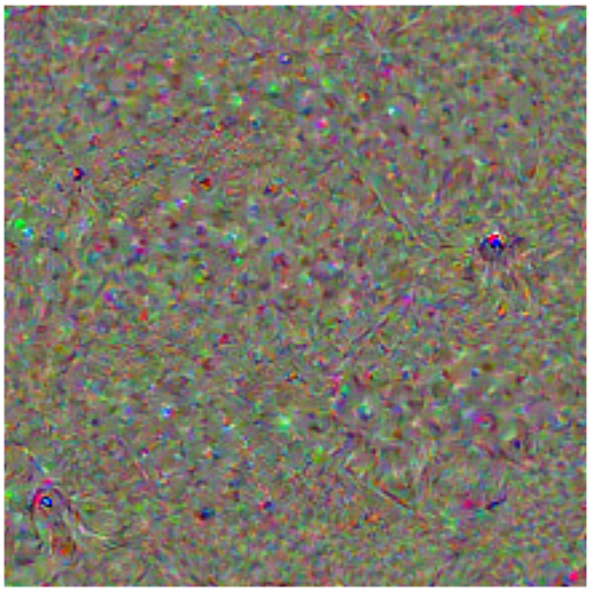
tick
Iteration 100 / 100



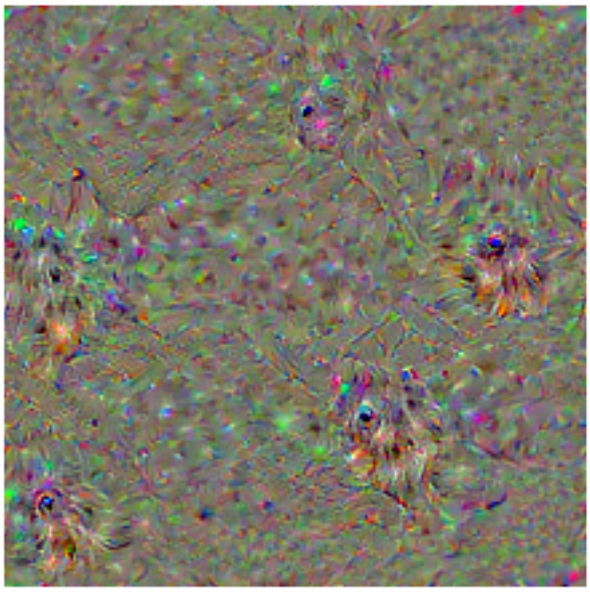
Yorkshire terrier
Iteration 1 / 100



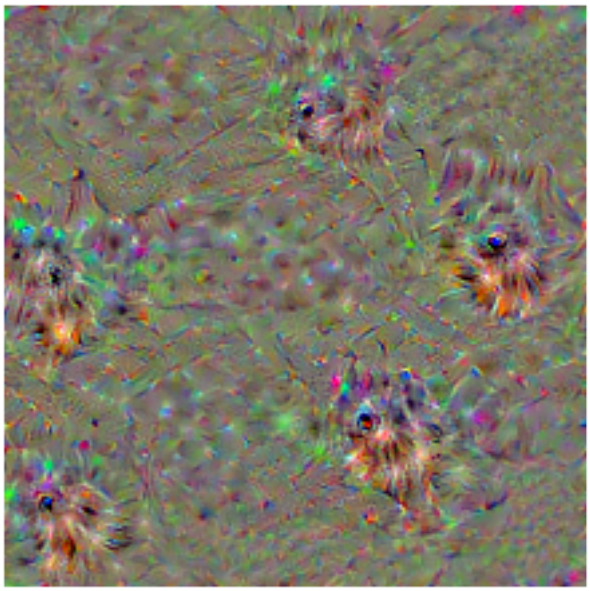
Yorkshire terrier
Iteration 25 / 100



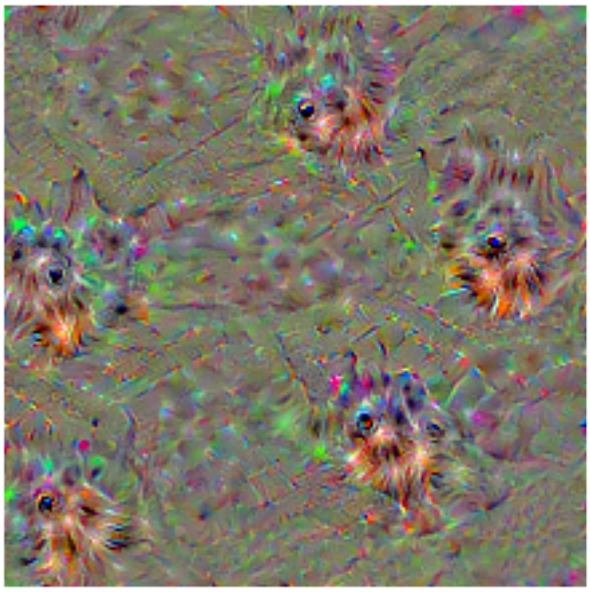
Yorkshire terrier
Iteration 50 / 100



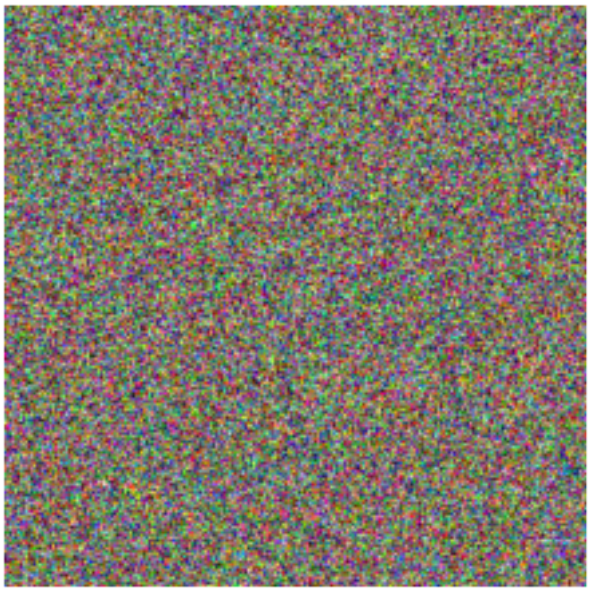
Yorkshire terrier
Iteration 75 / 100



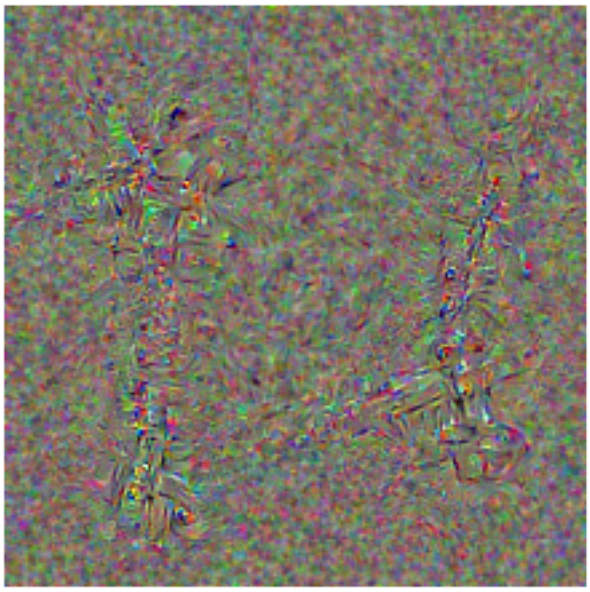
Yorkshire terrier
Iteration 100 / 100



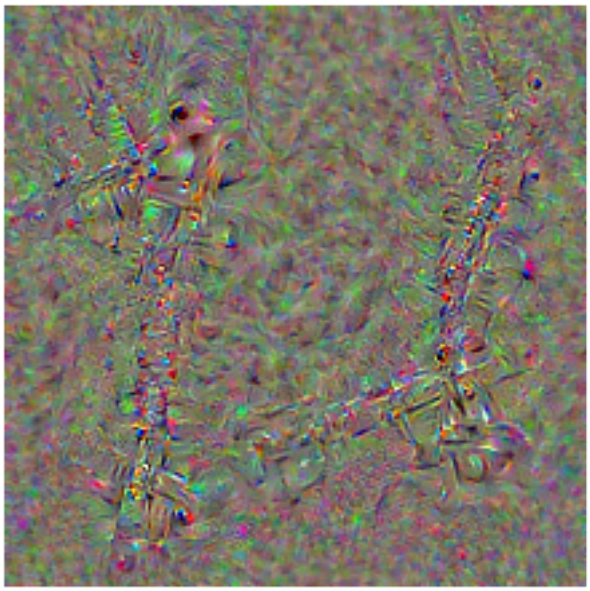
oboe, hautboy, hautbois
Iteration 1 / 100



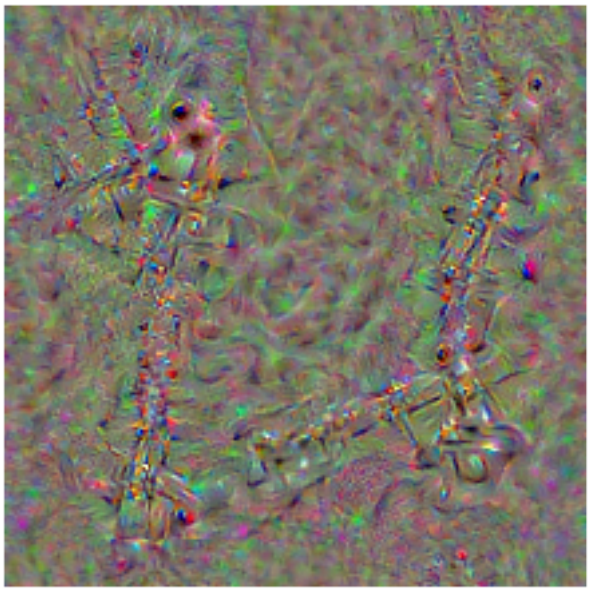
oboe, hautboy, hautbois
Iteration 25 / 100



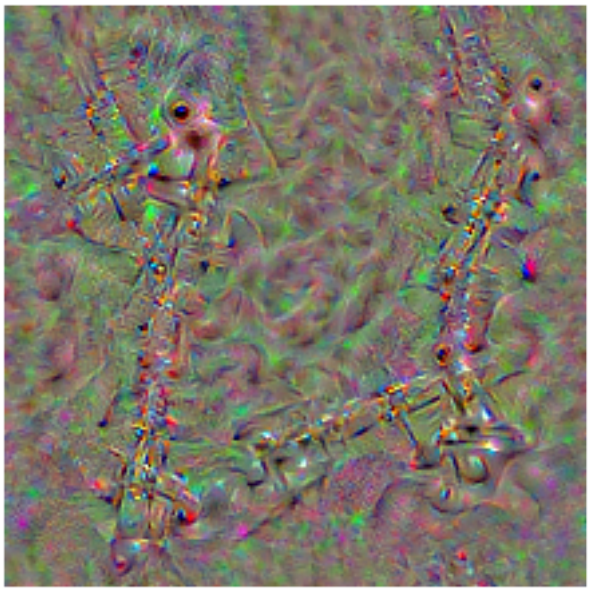
oboe, hautboy, hautbois
Iteration 50 / 100



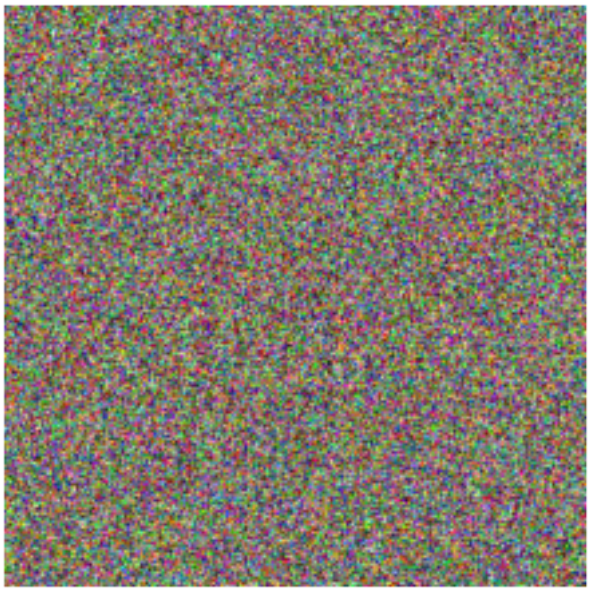
oboe, hautboy, hautbois
Iteration 75 / 100



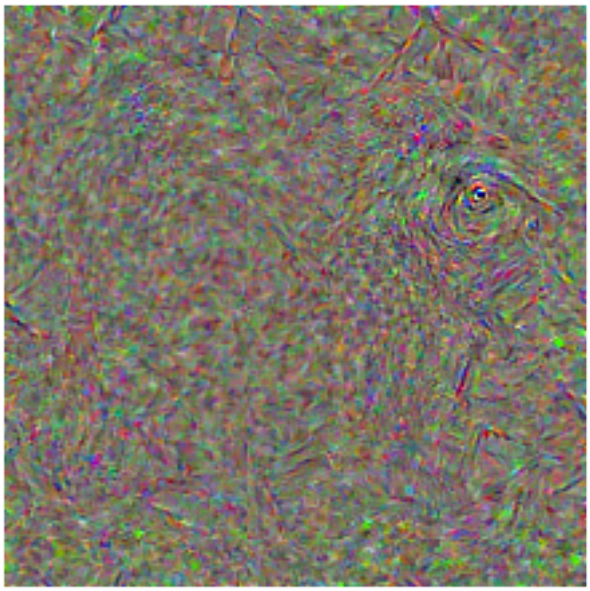
oboe, hautboy, hautbois
Iteration 100 / 100



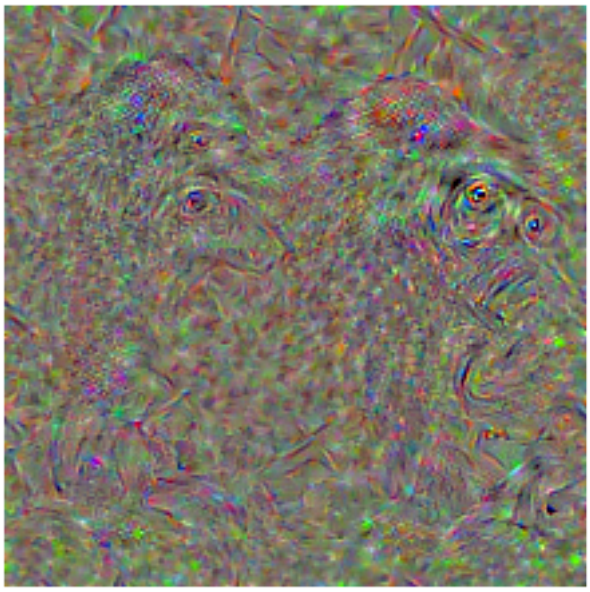
gorilla, Gorilla gorilla
Iteration 1 / 100



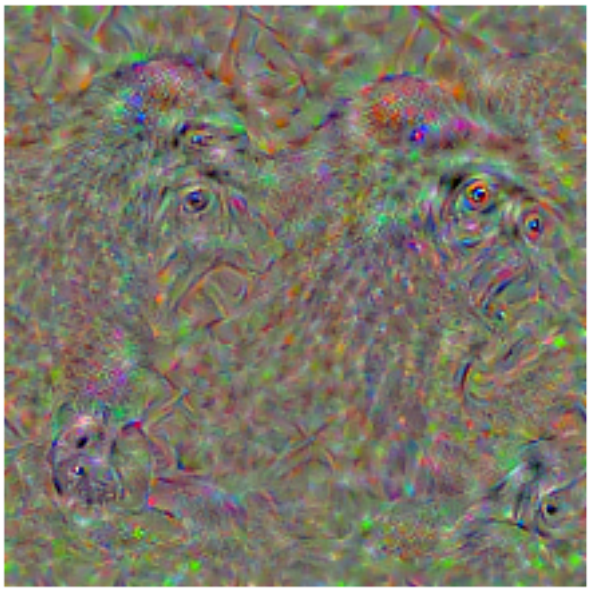
gorilla, Gorilla gorilla
Iteration 25 / 100



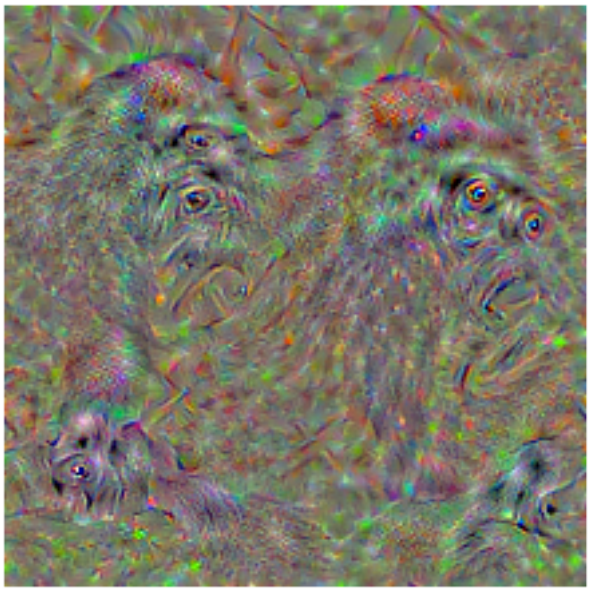
gorilla, Gorilla gorilla
Iteration 50 / 100



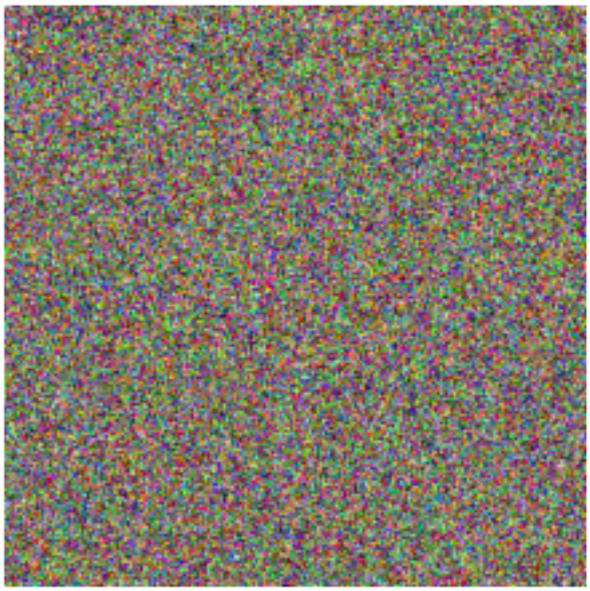
gorilla, Gorilla gorilla
Iteration 75 / 100



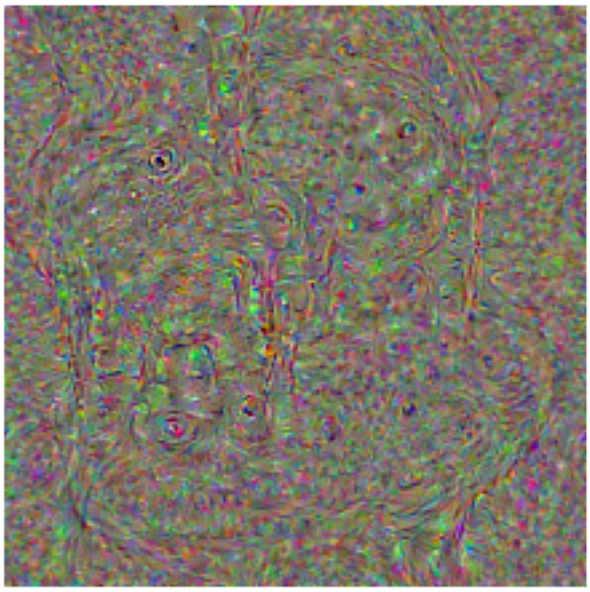
gorilla, Gorilla gorilla
Iteration 100 / 100



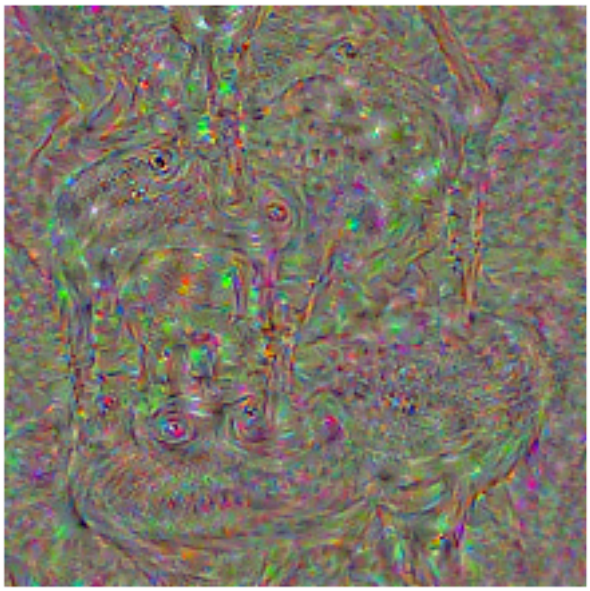
hourglass
Iteration 1 / 100



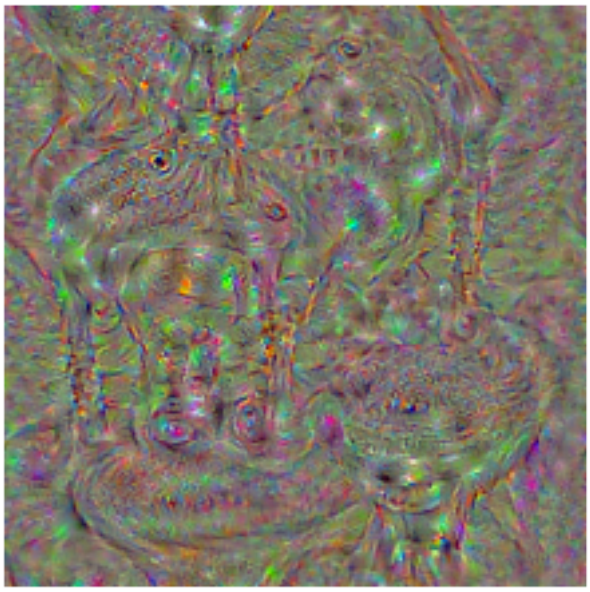
hourglass
Iteration 25 / 100

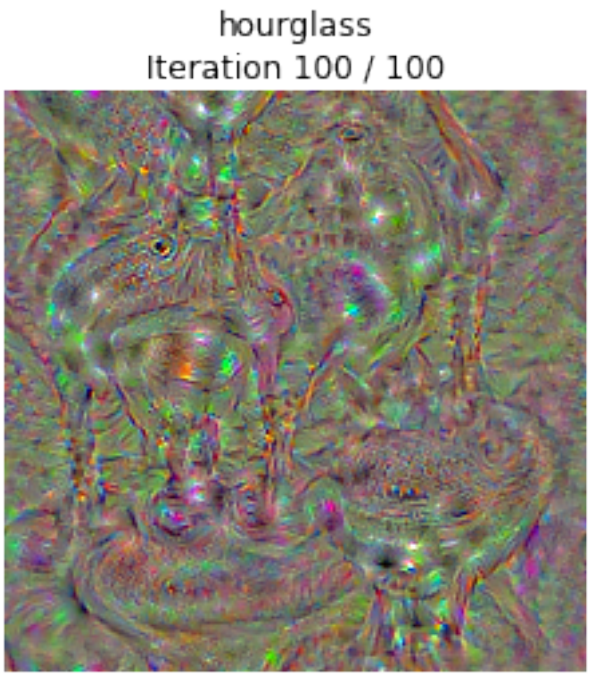


hourglass
Iteration 50 / 100



hourglass
Iteration 75 / 100



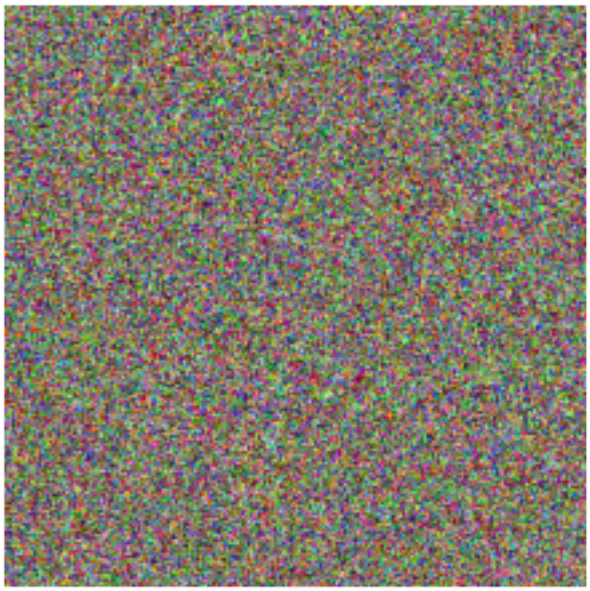


Try out your class visualization on other classes! You should also feel free to play with various hyperparameters to try and improve the quality of the generated image, but this is not required.

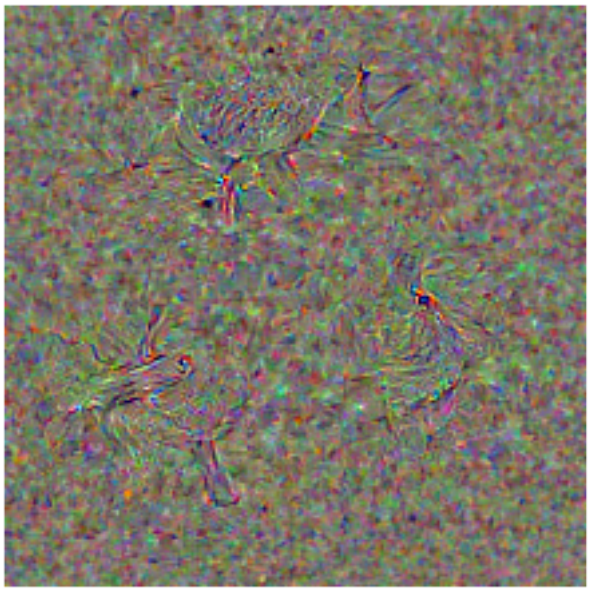
[51]:
```
# target_y = 78 # Tick
# target_y = 187 # Yorkshire Terrier
# target_y = 683 # Oboe
# target_y = 366 # Gorilla
# target_y = 604 # Hourglass
target_y = np.random.randint(1000)
print(class_names[target_y])
X = create_class_visualization(target_y, model, dtype)
```

```
red-backed sandpiper, dunlin, Erolia alpina
```

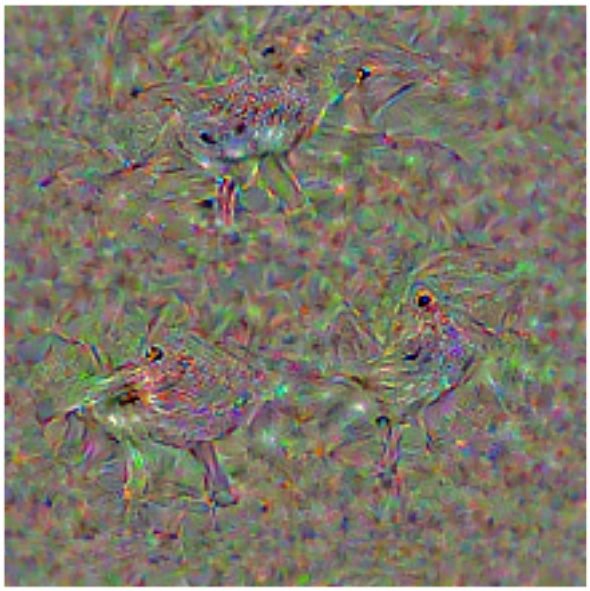
red-backed sandpiper, dunlin, Erolia alpina
Iteration 1 / 100



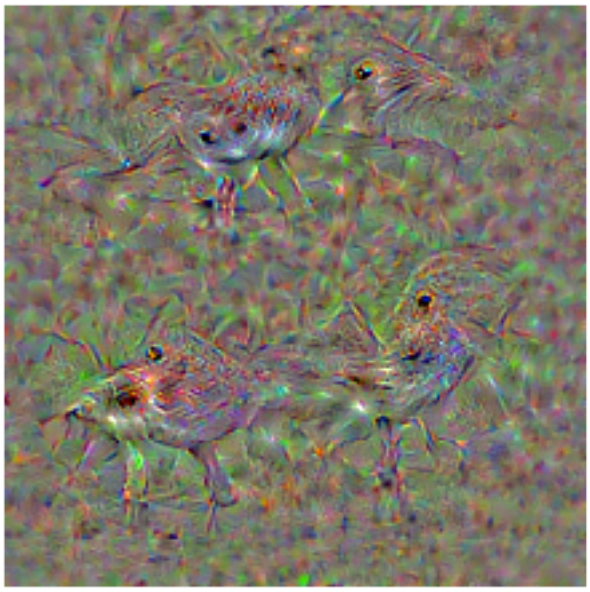
red-backed sandpiper, dunlin, Erolia alpina
Iteration 25 / 100

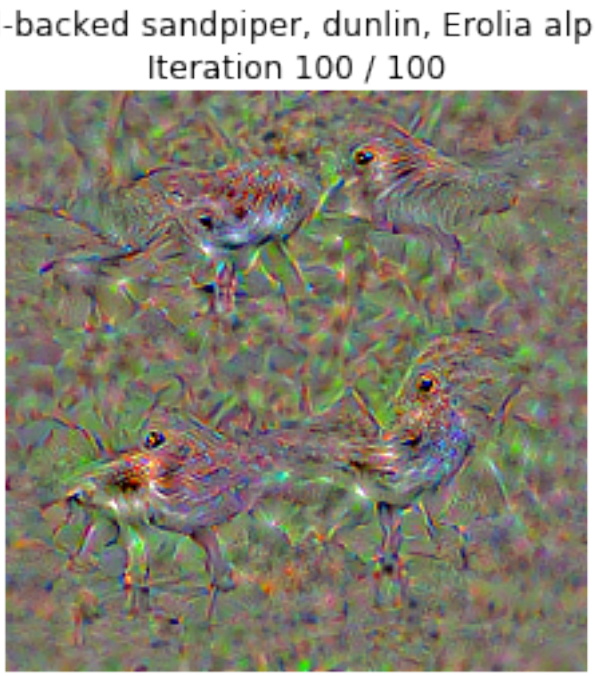


red-backed sandpiper, dunlin, Erolia alpina
Iteration 50 / 100



red-backed sandpiper, dunlin, Erolia alpina
Iteration 75 / 100




red-backed sandpiper, dunlin, Erolia alpina
Iteration 100 / 100

[ ]:

StyleTransfer-PyTorch

February 26, 2020

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:

caption

Excited? Let's get started!

First, run the setup cells which provide the utility functions you will need later.

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

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.

[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 complete this notebook."

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

[3]:
```
dtype = torch.FloatTensor
# Uncomment out the following line if you're on a machine with a GPU set up for␣
 ↪PyTorch!
# dtype = torch.cuda.FloatTensor
```

[4]:
```
# 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 of
# 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 model.
      features[i] is a PyTorch Variable of shape (N, C_i, H_i, W_i); recall␣
↪that features
      from different layers of the network may have different numbers of␣
↪channels (C_i) and
      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:
$L_c = w_c \times \sum_{i,j}(F_{ij}^\ell - P_{ij}^\ell)^2$

[5]:
```
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 shape
        (1, C_l, H_l, W_l).
    - content_target: features of the content image, Tensor with shape (1, C_l,␣
↪H_l, W_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.                                         ␣
↪ #
  ␣
↪##############################################################################
    l_c = content_weight * torch.sum((content_current - content_original).
↪pow(2))
    return l_c
  ␣
↪##############################################################################
    #                               END OF YOUR CODE                           ␣
↪ #
  ␣
↪##############################################################################
```

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

[6]:
```
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[content_layer]).data.numpy()
    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:

$$L^\ell_s = w_\ell \sum_{i,j} \left(G^\ell_{ij} - A^\ell_{ij} \right)^2$$

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:

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

Begin by implementing the Gram matrix computation below:

[12]:
```
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.␣
↪ #
    ␣
↪#############################################################################
    N, C, H, W = features.size()
    features = features.view(N * C, H * W)
    # Let each feature map go through all feature maps
    g_m = torch.mm(features, features.t())

    return g_m.div(N * C * H * W)
    ␣
↪#############################################################################
    #                               END OF YOUR CODE                           ␣
↪ #
    ␣
↪#############################################################################
```

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

[13]:
```
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:

[14]:
```
# 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 the
      style loss.
    - style_targets: List of the same length as style_layers, where␣
↪style_targets[i] is
      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.                       ␣
↪ #
  ␣
↪################################################################################
    style_loss = 0
    for i in range(len(style_layers)):
        G = gram_matrix(feats[style_layers[i]])
        style_loss += style_weights[i] * torch.sum((G - style_targets[i]).
↪pow(2))
    return style_loss
  ␣
↪################################################################################
    #                               END OF YOUR CODE                        ␣
↪ #
  ␣
↪################################################################################
```

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

[15]:
```
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).data.numpy()
    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.

[16]:
```
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 = tv_weight * (torch.sum((img[:,:,:,1:] - img[:,:,:,:-1]).pow(2))
                        + torch.sum((img[:,:,1:,:] - img[:,:,:-1,:]).pow(2)))
    return loss


    ################################################################################
```

```
    #                             END OF YOUR CODE                                 ␣
↪ #
  ␣
↪###############################################################################
```

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

[17]:
```
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.**

[18]:
```
def style_transfer(content_image, style_image, image_size, style_size,␣
↪content_layer, content_weight,
                   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␣
↪generated image)
    - 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. ␣
↪       #
        ␣
↪#############################################################################
        loss_content = content_loss(content_weight, feats[content_layer],␣
↪content_target)
        loss_style = style_loss(feats, style_layers, style_targets,␣
↪style_weights)
        loss_tv = tv_loss(img_var, tv_weight)

        total_loss = loss_content + loss_tv + loss_style

        optimizer.zero_grad()
        total_loss.backward()

        if t > decay_lr_at:
            for g in optimizer.param_groups:
                g['lr'] = 0.001
        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.

[19]:
```
# 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)
```

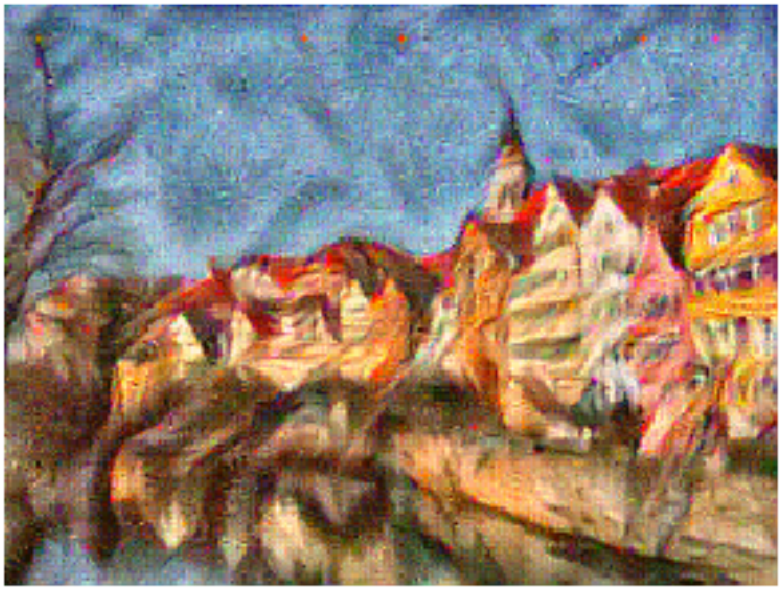
Content Source Img.

Style Source Img.

```
Iteration 0
```

```
Iteration 100
```

```
Iteration 199
```

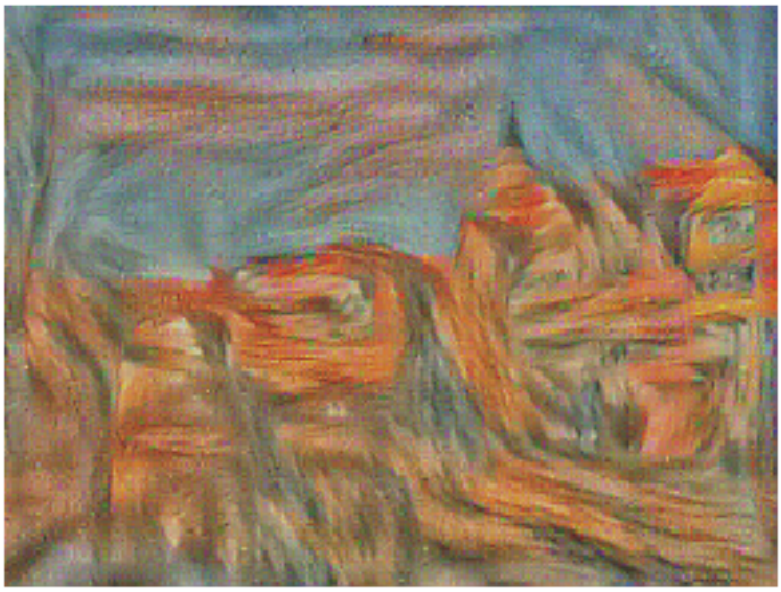


[20]:
```
# Scream + Tubingen
params2 = {
    'content_image':'styles/tubingen.jpg',
    'style_image':'styles/the_scream.jpg',
    'image_size':192,
```

```
    '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)
```


```
Iteration 0
```

Iteration 100



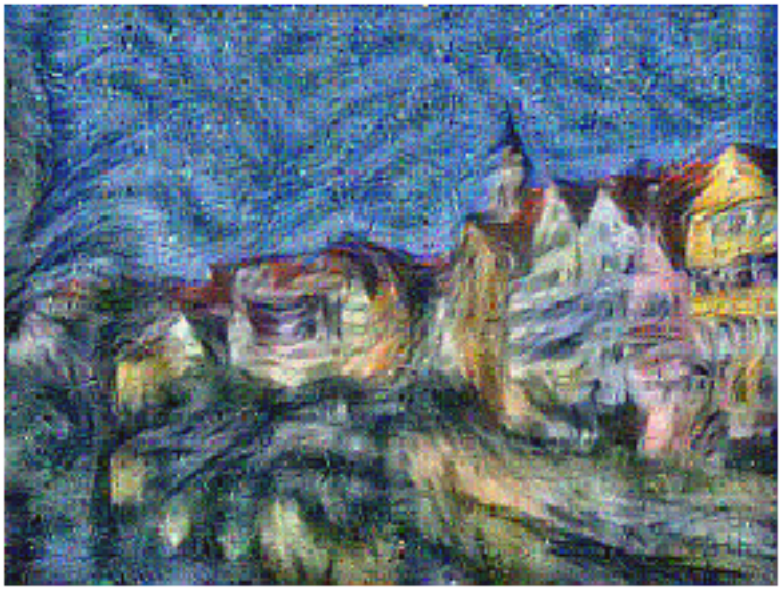
Iteration 199

[21]:
```
# 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)
```

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.)

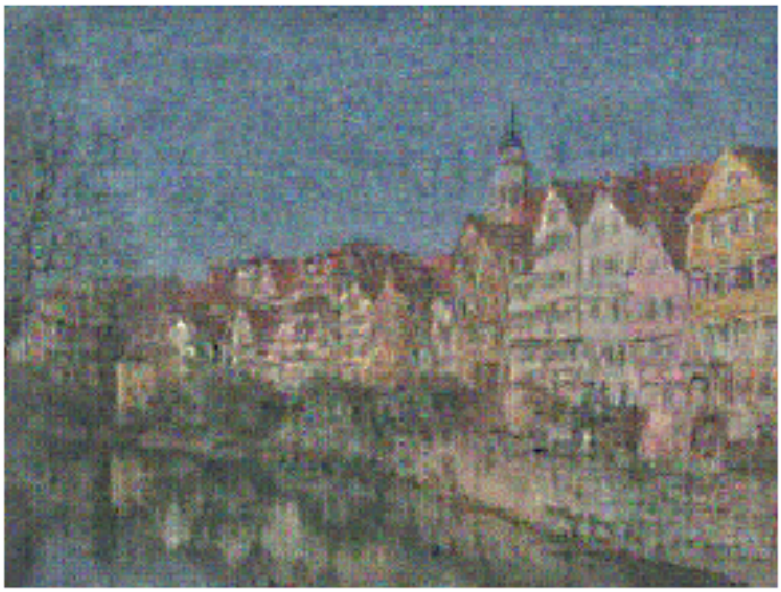
[2] Aravindh Mahendran, Andrea Vedaldi, "Understanding Deep Image Representations by Inverting them", CVPR 2015

[22]:
```
# 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 loss
    '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
```

[ ]:

[ ]: