Network_Visualization

February 19, 2023

[1]:

```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive')

# TODO: Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment2/'
FOLDERNAME = 'cs231n/assignments/assignment2/'
assert FOLDERNAME is not None, "[!] Enter the foldername."

# Now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# This downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
%cd /content/drive/My\ Drive/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/cs231n/assignments/assignment2/cs231n/datasets
/content/drive/My Drive/cs231n/assignments/assignment2
```

1 Network Visualization

In this notebook, we will explore the use of *image gradients* for generating new images.

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.

Here we will do something slightly different. We will start from a CNN 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 we will 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.

We will explore three techniques for image generation.

Saliency Maps. We can use saliency maps to tell which part of the image influenced the classification decision made by the network.

Fooling Images. We can perturb an input image so that it appears the same to humans but will be misclassified by the pretrained network.

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.

[2]:

```
# Setup cell.
import torch
import torchvision
import numpy as np
import random
import matplotlib.pyplot as plt
from PIL import Image
from cs231n.image_utils import SQUEEZENET_MEAN, SQUEEZENET_STD
from cs231n.net_visualization_pytorch import *

%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'

%load_ext autoreload
%autoreload 2
```

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 [1], 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 image generation experiments on CPU.

[1] 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
```

```python
# with respect to model parameters.
for param in model.parameters():
    param.requires_grad = False
```

```
/usr/local/lib/python3.8/dist-packages/torchvision/models/_utils.py:208:
UserWarning: The parameter 'pretrained' is deprecated since 0.13 and may be
removed in the future, please use 'weights' instead.
  warnings.warn(
/usr/local/lib/python3.8/dist-packages/torchvision/models/_utils.py:223:
UserWarning: Arguments other than a weight enum or `None` for 'weights' are
deprecated since 0.13 and may be removed in the future. The current behavior is
equivalent to passing `weights=SqueezeNet1_1_Weights.IMAGENET1K_V1`. You can
also use `weights=SqueezeNet1_1_Weights.DEFAULT` to get the most up-to-date
weights.
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/squeezenet1_1-b8a52dc0.pth" to
/root/.cache/torch/hub/checkpoints/squeezenet1_1-b8a52dc0.pth
```

```
  0%|          | 0.00/4.73M [00:00<?, ?B/s]
```

2.1 Loading ImageNet Validation Images

We have provided a few example images from the validation set of the ImageNet ILSVRC 2012 Classification dataset. 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.

```python
[4]: from cs231n.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

Using this pretrained model, we will compute class saliency maps as described in Section 3.1 of [2].

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.

[2] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps", ICLR Workshop 2014.

3.0.1 Hint: PyTorch `gather` method

Recall in Assignment 1 you needed 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 of shape `(N, C)` and `y` is a PyTorch Tensor 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 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]:

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

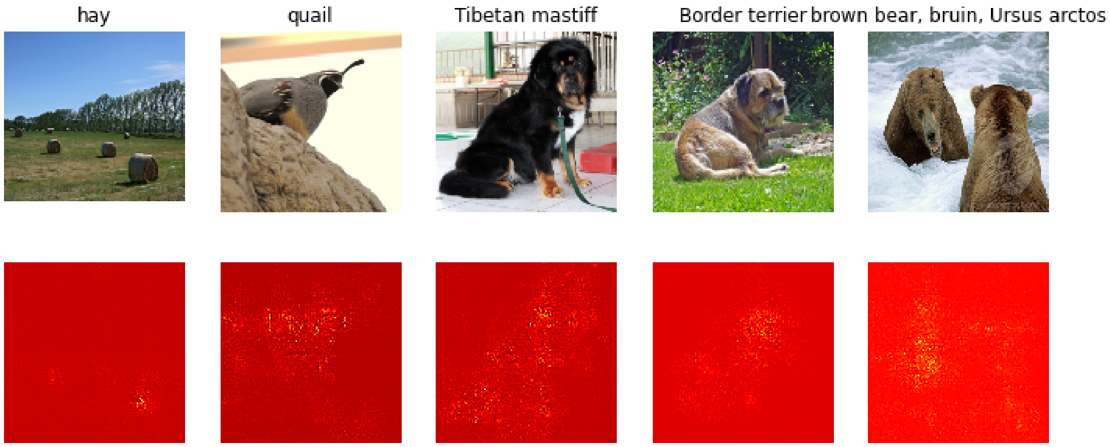
```
tensor([[ 1.0058e+00, -7.6487e-01,  5.9185e-01, -1.6828e-01, -1.7958e+00],
        [ 1.9185e+00, -2.7419e-03, -5.0717e-01, -2.7804e-01, -1.4080e+00],
        [ 1.6661e-01,  1.1826e+00, -6.1828e-01, -1.7337e-02,  1.3515e+00],
        [ 4.8192e-01,  1.3348e+00,  1.1010e-03, -7.6283e-01, -1.4448e+00]])
tensor([1, 2, 1, 3])
tensor([-0.7649, -0.5072,  1.1826, -0.7628])
```

[5]:
```
torch.return_types.max(
values=tensor([2.1174, 1.2155, 0.5030, 1.0872]),
indices=tensor([0, 1, 3, 2]))
```

Implement `compute_saliency_maps` function inside `cs231n/net_visualization_pytorch.py`

Once you have completed the implementation above, run the following to visualize some class saliency maps on our example images from the ImageNet validation set:

[6]:
```python
def show_saliency_maps(X, y):
    # Convert X and y from numpy arrays to Torch Tensors
    X_tensor = torch.cat([preprocess(Image.fromarray(x)) for x in X], dim=0)
    y_tensor = torch.LongTensor(y)

    # Compute saliency maps for images in X
    saliency = compute_saliency_maps(X_tensor, y_tensor, model)

    # Convert the saliency map from Torch Tensor to numpy array and show images
    # and saliency maps together.
    saliency = saliency.numpy()
    N = X.shape[0]
    for i in range(N):
        plt.subplot(2, N, i + 1)
        plt.imshow(X[i])
        plt.axis('off')
        plt.title(class_names[y[i]])
        plt.subplot(2, N, N + i + 1)
        plt.imshow(saliency[i], cmap=plt.cm.hot)
        plt.axis('off')
        plt.gcf().set_size_inches(12, 5)
    plt.show()

show_saliency_maps(X, y)
```



4 Inline Question 1

A friend of yours suggests that in order to find an image that maximizes the correct score, we can perform gradient ascent on the input image, but instead of the gradient we can actually use the saliency map in each step to update the image. Is this assertion true? Why or why not?

Your Answer: I think the saliency image is only one channel, but the shape of the gradient of image is (3, H, W), so this assertion is not True.

5 Fooling Images

We can also use image gradients to generate "fooling images" as discussed in [3]. 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. Implement the following function to generate fooling images.

[3] Szegedy et al, "Intriguing properties of neural networks", ICLR 2014

Implement `make_fooling_image` function inside `cs231n/net_visualization_pytorch.py`

Run the following cell to generate a fooling image. You should ideally see at first glance no major difference between the original and fooling images, and the network should now make an incorrect prediction on the fooling one. However you should see a bit of random noise if you look at the 10x magnified difference between the original and fooling images. Feel free to change the `idx` variable to explore other images.

[50]:

```
idx = 0
target_y = 6

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(X_fooling)
```

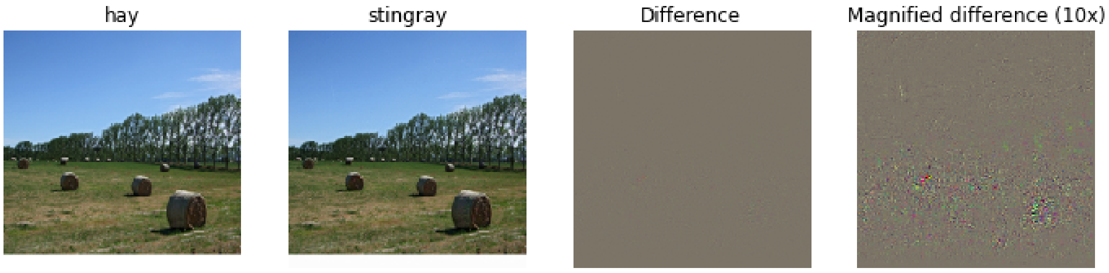
```python
assert target_y == scores.data.max(1)[1][0].item(), 'The model is not fooled!'
```

```
tensor(5.2135, grad_fn=<SelectBackward0>)
tensor(5.3097, grad_fn=<SelectBackward0>)
tensor(5.3865, grad_fn=<SelectBackward0>)
tensor(5.4650, grad_fn=<SelectBackward0>)
tensor(5.5583, grad_fn=<SelectBackward0>)
tensor(5.6737, grad_fn=<SelectBackward0>)
tensor(5.8031, grad_fn=<SelectBackward0>)
tensor(5.9385, grad_fn=<SelectBackward0>)
tensor(6.0813, grad_fn=<SelectBackward0>)
tensor(6.2441, grad_fn=<SelectBackward0>)
tensor(6.4013, grad_fn=<SelectBackward0>)
tensor(6.5437, grad_fn=<SelectBackward0>)
tensor(6.6869, grad_fn=<SelectBackward0>)
tensor(6.8144, grad_fn=<SelectBackward0>)
tensor(6.9290, grad_fn=<SelectBackward0>)
tensor(7.0425, grad_fn=<SelectBackward0>)
tensor(7.1518, grad_fn=<SelectBackward0>)
tensor(7.2599, grad_fn=<SelectBackward0>)
tensor(7.3713, grad_fn=<SelectBackward0>)
tensor(7.4849, grad_fn=<SelectBackward0>)
tensor(7.5994, grad_fn=<SelectBackward0>)
tensor(7.7194, grad_fn=<SelectBackward0>)
tensor(7.8357, grad_fn=<SelectBackward0>)
tensor(7.9493, grad_fn=<SelectBackward0>)
tensor(8.0626, grad_fn=<SelectBackward0>)
tensor(8.1755, grad_fn=<SelectBackward0>)
tensor(8.2889, grad_fn=<SelectBackward0>)
tensor(8.4015, grad_fn=<SelectBackward0>)
tensor(8.5119, grad_fn=<SelectBackward0>)
tensor(8.6192, grad_fn=<SelectBackward0>)
tensor(8.7231, grad_fn=<SelectBackward0>)
tensor(8.8260, grad_fn=<SelectBackward0>)
tensor(8.9270, grad_fn=<SelectBackward0>)
tensor(9.0250, grad_fn=<SelectBackward0>)
tensor(9.1224, grad_fn=<SelectBackward0>)
tensor(9.2185, grad_fn=<SelectBackward0>)
tensor(9.3140, grad_fn=<SelectBackward0>)
tensor(9.4117, grad_fn=<SelectBackward0>)
tensor(9.5094, grad_fn=<SelectBackward0>)
tensor(9.6058, grad_fn=<SelectBackward0>)
tensor(9.7000, grad_fn=<SelectBackward0>)
tensor(9.7930, grad_fn=<SelectBackward0>)
tensor(9.8869, grad_fn=<SelectBackward0>)
tensor(9.9796, grad_fn=<SelectBackward0>)
tensor(10.0717, grad_fn=<SelectBackward0>)
tensor(10.1632, grad_fn=<SelectBackward0>)
tensor(10.2550, grad_fn=<SelectBackward0>)
tensor(10.3465, grad_fn=<SelectBackward0>)
tensor(10.4361, grad_fn=<SelectBackward0>)
tensor(10.5239, grad_fn=<SelectBackward0>)
tensor(10.6136, grad_fn=<SelectBackward0>)
tensor(10.7025, grad_fn=<SelectBackward0>)
tensor(10.7911, grad_fn=<SelectBackward0>)
tensor(10.8778, grad_fn=<SelectBackward0>)
tensor(10.9608, grad_fn=<SelectBackward0>)
tensor(11.0438, grad_fn=<SelectBackward0>)
tensor(11.1264, grad_fn=<SelectBackward0>)
tensor(11.2083, grad_fn=<SelectBackward0>)
tensor(11.2893, grad_fn=<SelectBackward0>)
tensor(11.3692, grad_fn=<SelectBackward0>)
tensor(11.4482, grad_fn=<SelectBackward0>)
tensor(11.5272, grad_fn=<SelectBackward0>)
tensor(11.6063, grad_fn=<SelectBackward0>)
tensor(11.6846, grad_fn=<SelectBackward0>)
tensor(11.7629, grad_fn=<SelectBackward0>)
tensor(11.8400, grad_fn=<SelectBackward0>)
tensor(11.9160, grad_fn=<SelectBackward0>)
tensor(11.9907, grad_fn=<SelectBackward0>)
tensor(12.0652, grad_fn=<SelectBackward0>)
tensor(12.1402, grad_fn=<SelectBackward0>)
tensor(12.2154, grad_fn=<SelectBackward0>)
tensor(12.2899, grad_fn=<SelectBackward0>)
tensor(12.3644, grad_fn=<SelectBackward0>)
tensor(12.4387, grad_fn=<SelectBackward0>)
tensor(12.5129, grad_fn=<SelectBackward0>)
tensor(12.5855, grad_fn=<SelectBackward0>)
tensor(12.6561, grad_fn=<SelectBackward0>)
tensor(12.7248, grad_fn=<SelectBackward0>)
tensor(12.7936, grad_fn=<SelectBackward0>)
tensor(12.8623, grad_fn=<SelectBackward0>)
tensor(12.9292, grad_fn=<SelectBackward0>)
tensor(12.9952, grad_fn=<SelectBackward0>)
tensor(13.0602, grad_fn=<SelectBackward0>)
tensor(13.1249, grad_fn=<SelectBackward0>)
tensor(13.1891, grad_fn=<SelectBackward0>)
tensor(13.2534, grad_fn=<SelectBackward0>)
tensor(13.3184, grad_fn=<SelectBackward0>)
tensor(13.3833, grad_fn=<SelectBackward0>)
tensor(13.4475, grad_fn=<SelectBackward0>)
tensor(13.5115, grad_fn=<SelectBackward0>)
tensor(13.5743, grad_fn=<SelectBackward0>)
tensor(13.6372, grad_fn=<SelectBackward0>)
tensor(13.7007, grad_fn=<SelectBackward0>)
tensor(13.7630, grad_fn=<SelectBackward0>)
tensor(13.8252, grad_fn=<SelectBackward0>)
tensor(13.8873, grad_fn=<SelectBackward0>)
tensor(13.9489, grad_fn=<SelectBackward0>)
tensor(14.0104, grad_fn=<SelectBackward0>)
tensor(14.0716, grad_fn=<SelectBackward0>)
tensor(14.1323, grad_fn=<SelectBackward0>)
```

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

[51]:

```
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

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 [2]; [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.

[2] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. “Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps”, ICLR Workshop 2014.

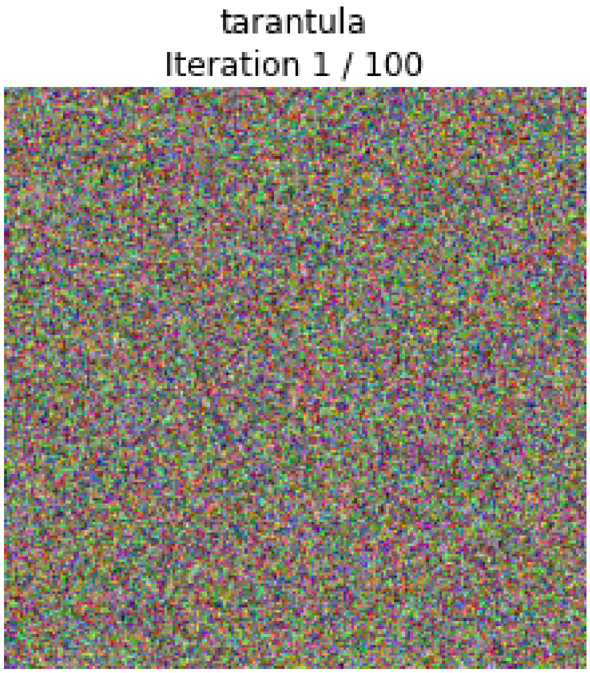
[3] Yosinski et al, “Understanding Neural Networks Through Deep Visualization”, ICML 2015 Deep Learning Workshop

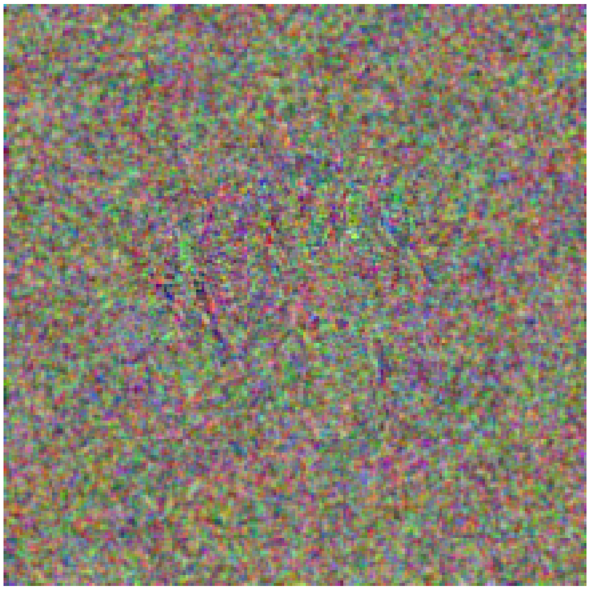
In `cs231n/net_visualization_pytorch.py` complete the implementation of the `class_visualization_update_step` used in the `create_class_visualization` function below. Once you have completed that implementation, run the following cells to generate an image of a Tarantula:

[52]:
```python
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.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 make it requires␣
↪gradient.
    img = torch.randn(1, 3, 224, 224).mul_(1.0).type(dtype).requires_grad_()

    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))
        class_visualization_update_step(img, model, target_y, l2_reg,␣
↪learning_rate)
        # 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.data[:, c].clamp_(min=lo, max=hi)
        if t % blur_every == 0:
            blur_image(img.data, 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.data.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.data.cpu())
```

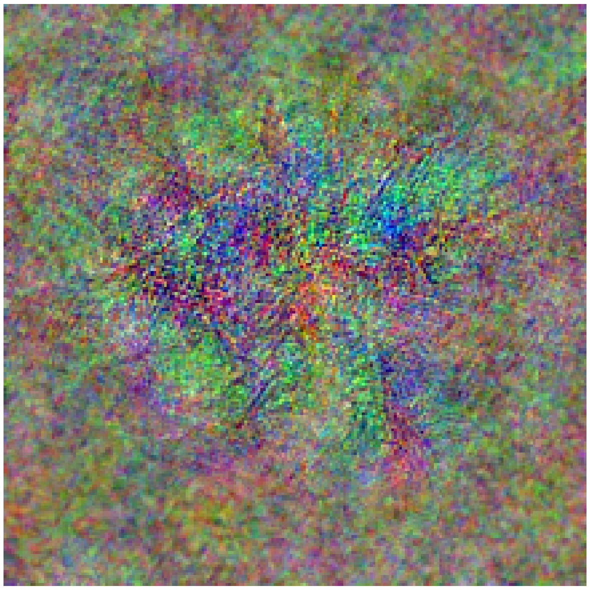
[56]:

```
dtype = torch.FloatTensor
model.type(dtype)

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
out = create_class_visualization(target_y, model, dtype)
```

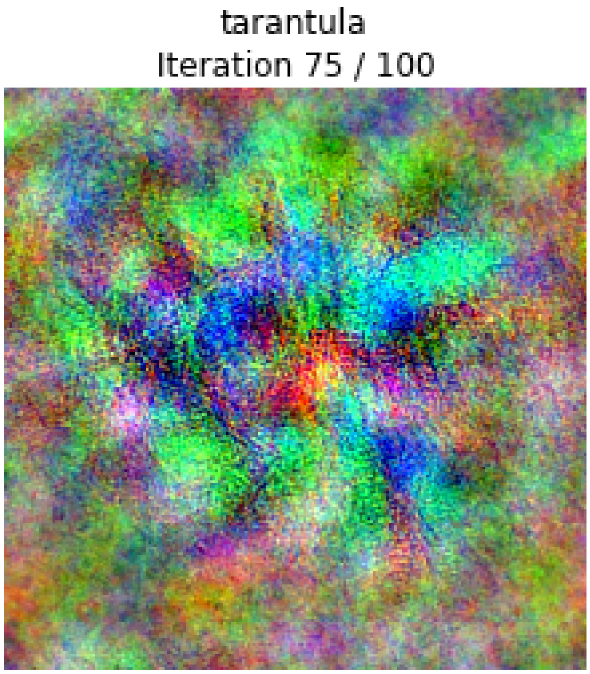


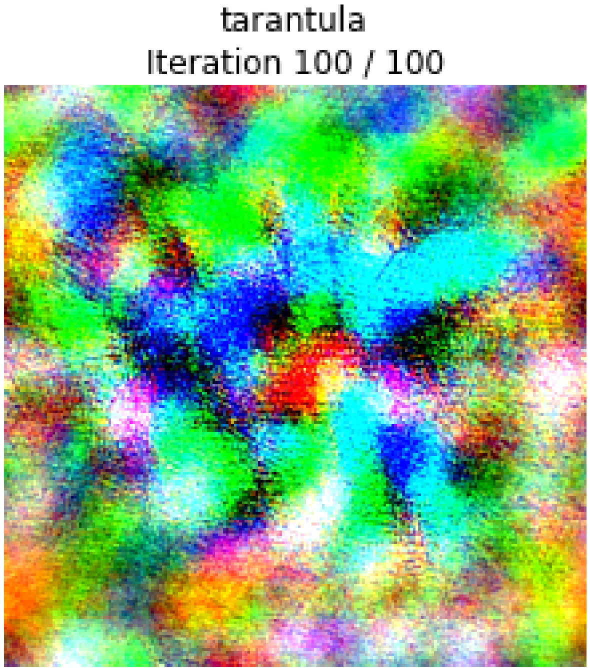


tarantula
Iteration 25 / 100



tarantula
Iteration 50 / 100



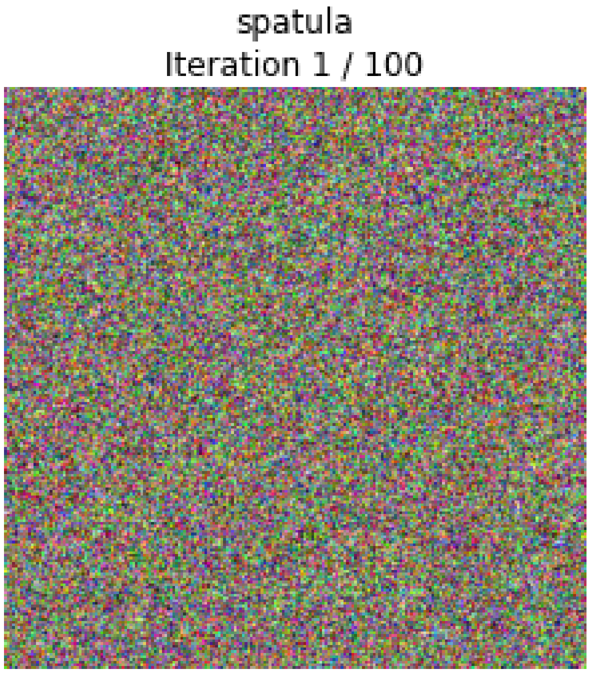


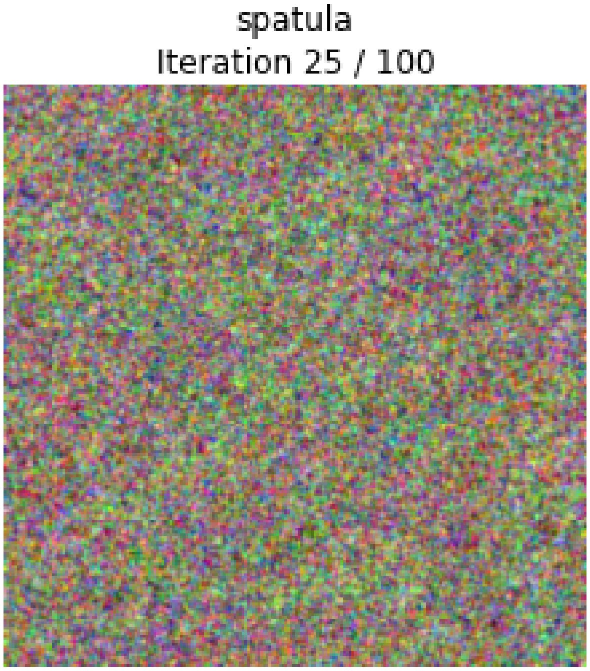
Try out your class visualization on other classes! You should also feel free to play with various hyperparameters to try and improve the quality of the generated image, but this is not required.

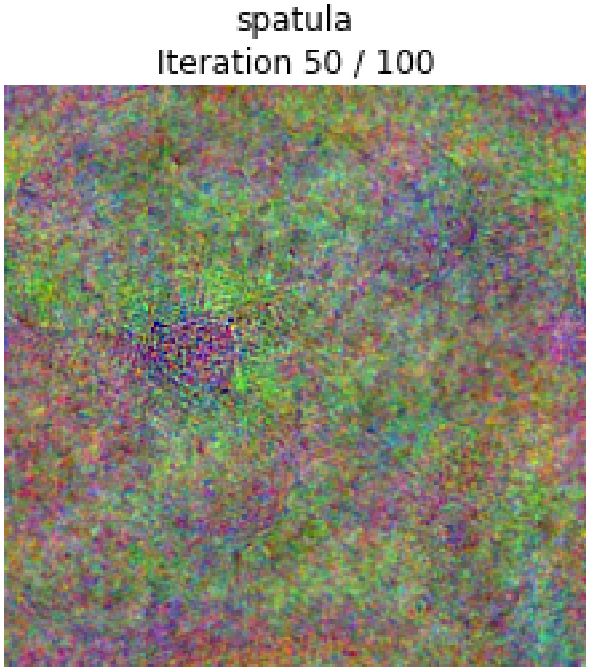
[57]:

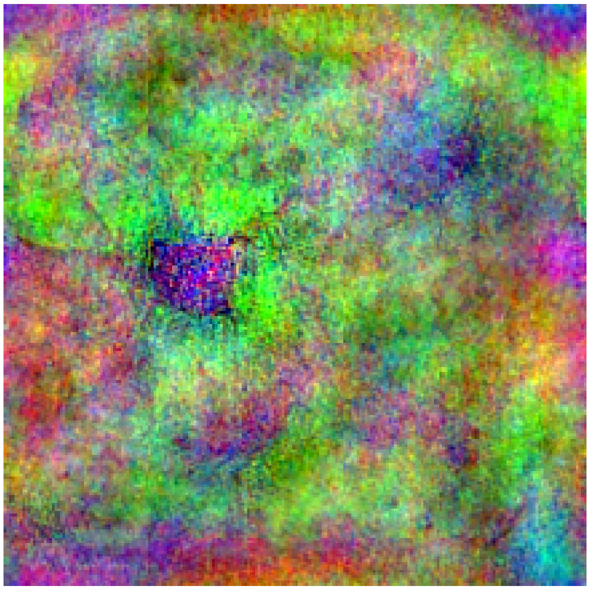
```
# target_y = 78 # Tick
# target_y = 187 # Yorkshire Terrier
# target_y = 683 # Oboe
# target_y = 366 # Gorilla
# target_y = 604 # Hourglass
target_y = np.random.randint(1000)
print(class_names[target_y])
X = create_class_visualization(target_y, model, dtype)
```

```
spatula
```

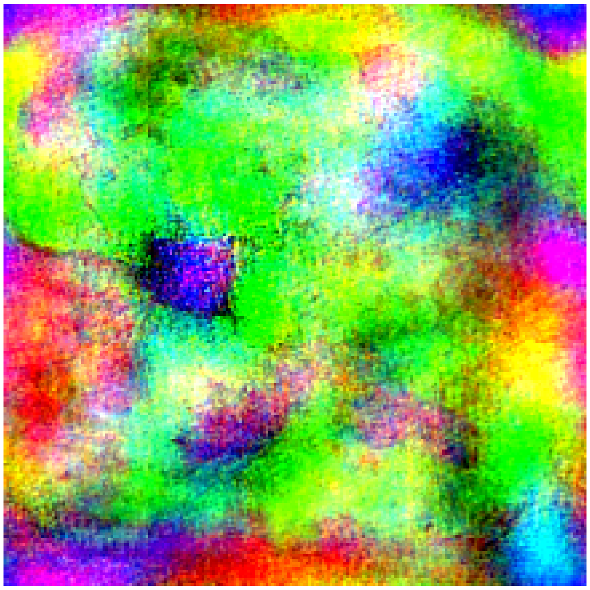



spatula
Iteration 25 / 100


spatula
Iteration 50 / 100



spatula
Iteration 75 / 100



spatula
Iteration 100 / 100