Network Visualization (PyTorch)

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 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.

In this notebook we will explore three techniques for image generation:

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

This notebook uses **PyTorch**

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

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

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

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

In [2]:
```
def preprocess(img, size=224):
    transform = T.Compose([
        T.Scale(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
```

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 ease 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

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

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

```
/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-
packages/torchvision/models/squeezenet.py:94: UserWarning: nn.init.ka
iming_uniform is now deprecated in favor of nn.init.kaiming_uniform_.
  init.kaiming_uniform(m.weight.data)
/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-
packages/torchvision/models/squeezenet.py:92: UserWarning: nn.init.no
rmal is now deprecated in favor of nn.init.normal_.
  init.normal(m.weight.data, mean=0.0, std=0.01)
Downloading: "https://download.pytorch.org/models/squeezenet1_1-f364a
a15.pth" to /Users/rishipuri/.torch/models/squeezenet1_1-f364aa15.pth
100%|██████████| 4966400/4966400 [00:00<00:00, 7367112.41it/s]
```

Load some ImageNet images

We have provided a few example images from the validation set of the ImageNet ILSVRC 2012 Classification dataset. To download these images, change to `deeplearning/datasets/` and run `get_imagenet_val.sh` .

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

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

In [4]:
```
from deeplearning.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()
```

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

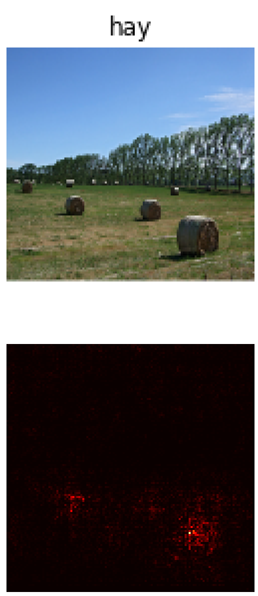
```
tensor([[ 0.4913,  1.1891, -1.1489,  0.0116, -0.4131],
        [-0.1069, -0.0172,  0.3661, -0.7271,  1.0016],
        [-1.7497, -0.4531, -0.3137, -0.3702, -0.8342],
        [-0.0438, -0.0393, -0.3222,  0.0648,  0.6691]])
tensor([1, 2, 1, 3])
tensor([ 1.1891,  0.3661, -0.4531,  0.0648])
```

In [ ]:
```
```

In [31]:
```
def compute_saliency_maps(X, y, model):
    """
    Compute a class saliency map using the model for images X and lab
els 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 salien
cy 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)
    saliency = None

    score=model(X_var)
    criterion=torch.nn.CrossEntropyLoss()
    loss=criterion(score,y_var)
    loss.backward()

    saliency=X_var.grad

    saliency,_=torch.max(abs(saliency),0)

    return saliency
```

In [32]:
```
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)
    y_pred=model(X_tensor)

    # 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 s
how 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)
```

```
/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-
packages/torchvision/transforms/transforms.py:188: UserWarning: The u
se of the transforms.Scale transform is deprecated, please use transf
orms.Resize instead.
  "please use transforms.Resize instead.")
```



Fooling Images

In [144]:
```
def make_fooling_image(X, target_y, model):
    """
    Generate a fooling image that is close to X, but that the model c
lassifies
    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 a
s target_y
    by the model.
    """
    # 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)

    learning_rate = 1

#     zeros=torch.zeros(1,1000)
#     target_y_var=zeros.long()
#     target_y_var[0,target_y]=1.0
    #target_y_var=target_y_var.reshape(1000)
    target_yhold=torch.zeros(1).long()
    target_yhold[0]=target_y
    target_y=target_yhold
    #print(target_y)
    for i in range(100):

        scores=model(X_fooling_var)
        #scores=torch.softmax(scores,1)
        #scores=scores.reshape(1000)
        #print(scores.data.max(1)[1][0])
        criterion=torch.nn.CrossEntropyLoss()
        loss=criterion(scores,target_y)

        loss.backward()
        g=X_fooling_var.grad
        g=g.detach()
        dx=learning_rate*g/torch.norm(g)
        #print(dx)
        X_fooling-=dx
        X_fooling_var=Variable(X_fooling, requires_grad=True)


    return X_fooling_var
```

Run the following cell to generate a fooling image:

In [ ]:

In [145]:
```
idx = 0
target_y = 6

X_tensor = torch.cat([preprocess(Image.fromarray(x)) for x in X], dim
=0)
scores = model(Variable(X_tensor[idx:idx+1]))
print('initial clasif'+str(scores.data.max(1)[1][0]))
X_fooling = make_fooling_image(X_tensor[idx:idx+1], target_y, model)

scores = model(Variable(X_fooling))
assert target_y == scores.data.max(1)[1][0], 'The model is not foole
d!'
```

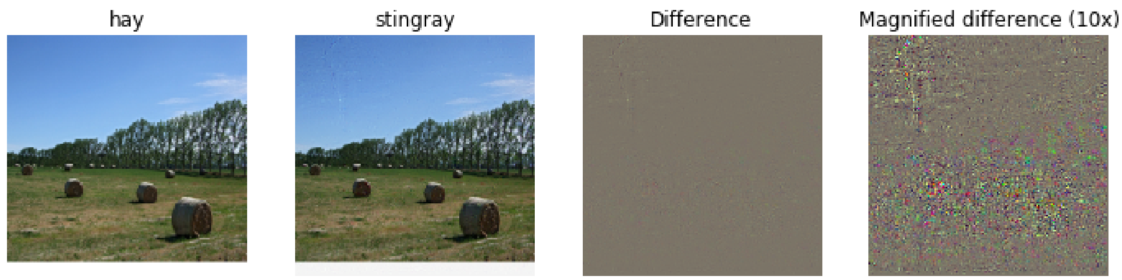
```
/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-
packages/torchvision/transforms/transforms.py:188: UserWarning: The u
se of the transforms.Scale transform is deprecated, please use transf
orms.Resize instead.
  "please use transforms.Resize instead.")
```

```
initial clasiftensor(958)
```

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

In [146]:
```
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()
```

```
/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-
packages/torchvision/transforms/transforms.py:188: UserWarning: The u
se of the transforms.Scale transform is deprecated, please use transf
orms.Resize instead.
  "please use transforms.Resize instead.")
```



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 [147]:
```
def jitter(X, ox, oy):
    """
    Helper function to randomly jitter an image.

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

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

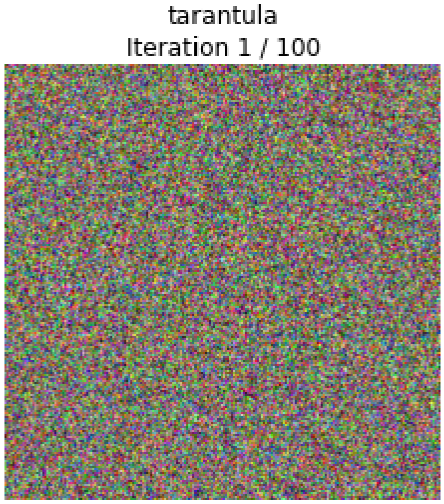
In [164]:
```
def create_class_visualization(target_y, model, dtype, **kwargs):
    """
    Generate an image to maximize the score of target_y under a pretr
ained model.

    Inputs:
    - target_y: Integer in the range [0, 1000) giving the index of th
e 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 regulari
zer
    - max_jitter: How much to gjitter the image as an implicit regula
rizer
    - 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 also wra
p it in
    # a PyTorch Variable.
    img = torch.randn(1, 3, 224, 224).mul_(1.0).type(dtype)
    img_var = Variable(img, requires_grad=True)
    target_yhold=torch.zeros(1).long()
    target_yhold[0]=target_y
    target_y=target_yhold
    for t in range(num_iterations):
        # Randomly jitter the image a bit; this gives slightly nicer
 results
        ox, oy = random.randint(0, max_jitter), random.randint(0, max
_jitter)
        img.copy_(jitter(img, ox, oy))



        scores=model(img_var)
        if t%show_every==0:
            print(scores.data.max(1)[1][0])
        criterion=torch.nn.CrossEntropyLoss()
        loss=criterion(scores,target_y)
        loss.backward()
        g=img_var.grad.detach()
        g==(g-2*l2_reg*(img))
        dIm=learning_rate*g/torch.norm(g)
        img-=dIm
```

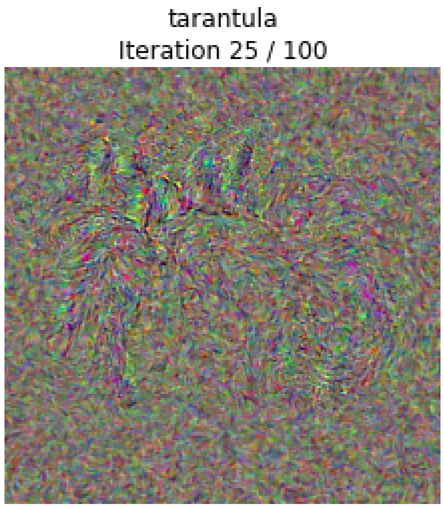
```
        img_var=Variable(img, requires_grad=True)


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

        # As regularizer, clamp and periodically blur the image
        for c in range(3):
            lo = float(-SQUEEZENET_MEAN[c] / SQUEEZENET_STD[c])
            hi = float((1.0 - SQUEEZENET_MEAN[c]) / SQUEEZENET_STD[c
])
            img[:, c].clamp_(min=lo, max=hi)
        if t % blur_every == 0:
            blur_image(img, sigma=0.5)

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

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

In [165]:

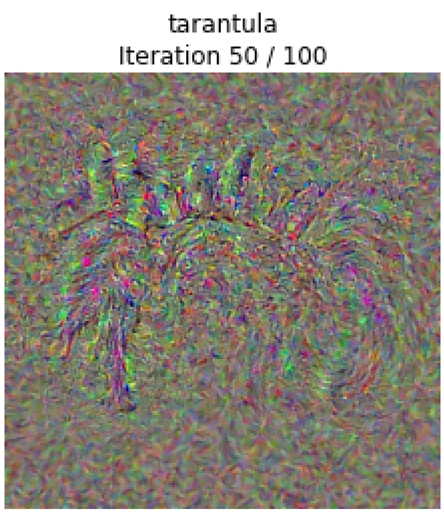
```
dtype = torch.FloatTensor
# dtype = torch.cuda.FloatTensor # Uncomment this to use GPU
model.type(dtype)

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)
```

```
tensor(539)
```



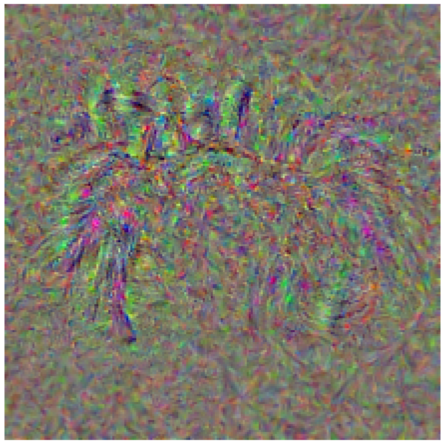



```
tensor(76)
```



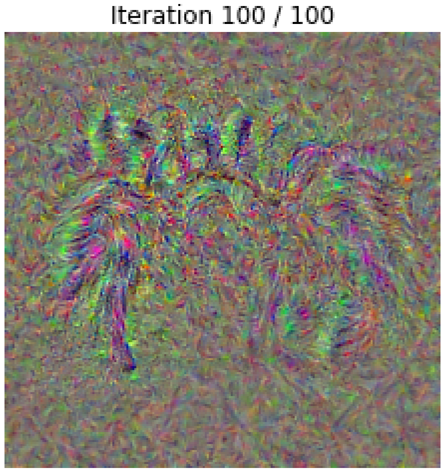

```
tensor(76)
```

tarantula
Iteration 75 / 100



```
tensor(76)
```

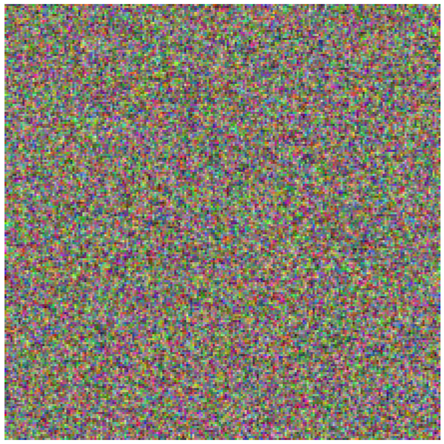
tarantula
Iteration 100 / 100



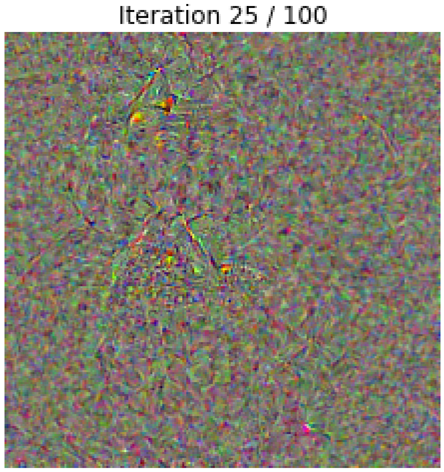
In [172]:
```
# target_y = 78 # Tick
#target_y = 187 # Yorkshire Terrier
# target_y = 683 # Oboe
#target_y = 366 # Gorilla
# target_y = 604 # Hourglass
target_y = np.random.randint(1000)
print(class_names[target_y])
X = create_class_visualization(target_y, model, dtype)
```

```
sulphur butterfly, sulfur butterfly
tensor(539)
```

sulphur butterfly, sulfur butterfly
Iteration 1 / 100

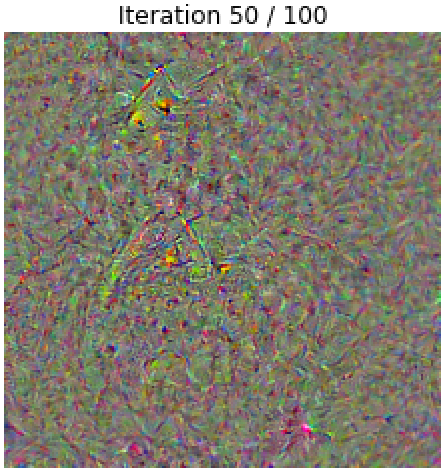


sulphur butterfly, sulfur butterfly
Iteration 25 / 100

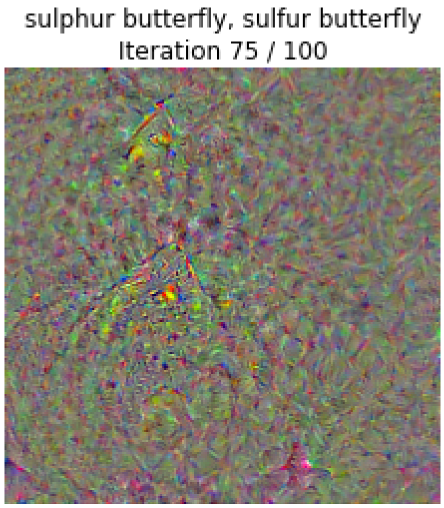


```
tensor(325)
```

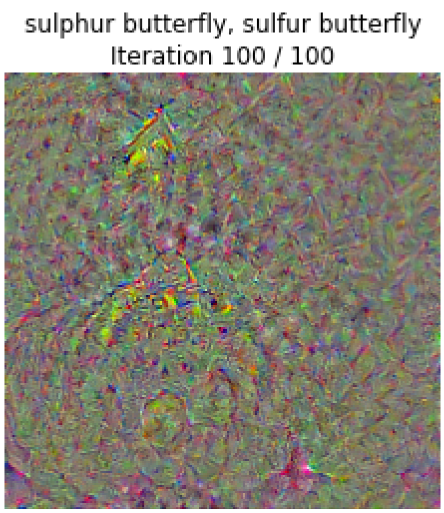
sulphur butterfly, sulfur butterfly
Iteration 50 / 100



```
tensor(325)
```




```
tensor(325)
```



In [ ]: