Deep Dreams (with Caffe)

This notebook demonstrates how to use the Caffe neural network framework to produce "dream" visuals shown in the Google Research blog post.

It'll be interesting to see what imagery people are able to generate using the described technique. If you post images to Google+, Facebook, or Twitter, be sure to tag them with **#deepdream** so other researchers can check them out too.

Dependencies

This notebook is designed to have as few dependencies as possible:

- Standard Python scientific stack: NumPy, SciPy, PIL, IPython. Those libraries can also be installed as a part of one of the scientific packages for Python, such as Anaconda or Canopy.
- Caffe deep learning framework (installation instructions).
- Google protobuf library that is used for Caffe model manipulation.

In [1]:

```
# imports and basic notebook setup
from io import StringIO
from io import BytesIO
import numpy as np
import scipy.ndimage as nd
import PIL.Image
from IPython.display import clear_output, Image, display
from google.protobuf import text_format
import matplotlib.pyplot as plt

import caffe

# If your GPU supports CUDA and Caffe was built with CUDA support,
# uncomment the following to run Caffe operations on the GPU.
# caffe.set_mode_gpu()
# caffe.set_device(0) # select GPU device if multiple devices exist

def showarray(a, fmt='jpeg'):
    a = np.uint8(np.clip(a, 0, 255))
    f = BytesIO()
    PIL.Image.fromarray(a).save(f, fmt)
    display(Image(data=f.getvalue()))
```

Loading DNN model

In this notebook we are going to use a GoogLeNet model trained on ImageNet dataset. Feel free to experiment with other models from Caffe Model Zoo. One particularly interesting model was trained in MIT Places dataset. It produced many visuals from the original blog post.

In [2]:

```
model_path = '/home/sarala/Downloads/bvlc_googlenet/' # substitute your pat
h here
net_fn   = model_path + 'deploy.prototxt'
param_fn = model_path + 'bvlc_googlenet.caffemodel'

# Patching model to be able to compute gradients.
# Note that you can also manually add "force_backward: true" line to
"deploy.prototxt".
model = caffe.io.caffe_pb2.NetParameter()
text_format.Merge(open(net_fn).read(), model)
model.force_backward = True
open('tmp.prototxt', 'w').write(str(model))

net = caffe.Classifier('tmp.prototxt', param_fn,
                       mean = np.float32([104.0, 116.0, 122.0]), # ImageNet
mean, training set dependent
                       channel_swap = (2,1,0)) # the reference model has cha
nnels in BGR order instead of RGB

# a couple of utility functions for converting to and from Caffe's input im
age layout
def preprocess(net, img):
    return np.float32(np.rollaxis(img, 2)[::-1]) - net.transformer.mean['dat
a']
def deprocess(net, img):
    return np.dstack((img + net.transformer.mean['data'])[::-1])
```

Producing dreams

Making the "dream" images is very simple. Essentially it is just a gradient ascent process that tries to maximize the L2 norm of activations of a particular DNN layer. Here are a few simple tricks that we found useful for getting good images:

- offset image by a random jitter
- normalize the magnitude of gradient ascent steps
- apply ascent across multiple scales (octaves)

First we implement a basic gradient ascent step function, applying the first two tricks:

In [3]:

```
def objective_L2(dst):
    dst.diff[:] = dst.data

def make_step(net, step_size=1, end='conv2/3x3_reduce',
              jitter=32, clip=True, objective=objective_L2):
    '''Basic gradient ascent step.'''

    src = net.blobs['data'] # input image is stored in Net's 'data' blob
    dst = net.blobs[end]

    ox, oy = np.random.randint(-jitter, jitter+1, 2)
    src.data[0] = np.roll(np.roll(src.data[0], ox, -1), oy, -2) # apply jitt
er shift

    net.forward(end=end)
```

```
    net.forward(end=end)
    objective(dst)  # specify the optimization objective
    net.backward(start=end)
    g = src.diff[0]
    # apply normalized ascent step to the input image
    src.data[:] += step_size/np.abs(g).mean() * g

    src.data[0] = np.roll(np.roll(src.data[0], -ox, -1), -oy, -2) # unshift
image

    if clip:
        bias = net.transformer.mean['data']
        src.data[:] = np.clip(src.data, -bias, 255-bias)
```

Next we implement an ascent through different scales. We call these scales "octaves".

In [4]:

```
def deepdream(net, base_img, iter_n=10, octave_n=4, octave_scale=1.4,
              end='inception_4c/output', clip=True, **step_params):
    # prepare base images for all octaves
    octaves = [preprocess(net, base_img)]
    for i in range(octave_n-1):
        octaves.append(nd.zoom(octaves[-1], (1, 1.0/octave_scale,1.0/octave_
scale), order=1))

    src = net.blobs['data']
    detail = np.zeros_like(octaves[-1]) # allocate image for network-produc
ed details
    for octave, octave_base in enumerate(octaves[::-1]):
        h, w = octave_base.shape[-2:]
        if octave > 0:
            # upscale details from the previous octave
            h1, w1 = detail.shape[-2:]
            detail = nd.zoom(detail, (1, 1.0*h/h1,1.0*w/w1), order=1)

        src.reshape(1,3,h,w) # resize the network's input image size
        src.data[0] = octave_base+detail
        for i in range(iter_n):
            make_step(net, end=end, clip=clip, **step_params)

            # visualization
            vis = deprocess(net, src.data[0])
            if not clip: # adjust image contrast if clipping is disabled
                vis = vis*(255.0/np.percentile(vis, 99.98))
            showarray(vis)
            print(octave, i, end, vis.shape)
            clear_output(wait=True)

        # extract details produced on the current octave
        detail = src.data[0]-octave_base
    # returning the resulting image
    return deprocess(net, src.data[0])
```

Now we are ready to let the neural network reveal its dreams! Let's take a cloud image as a starting point:

In [8]:

```
img = np.float32(PIL.Image.open('IMG_9292.jpg'))
showarray(img)
```

Running the next code cell starts the detail generation process. You may see how new patterns start to form, iteration by iteration, octave by octave.

In [15]:

```
_=deepdream(net, img)
```

```
3 9 inception_4c/output (575, 766, 3)
```

The complexity of the details generated depends on which layer's activations we try to maximize. Higher layers produce complex features, while lower ones enhance edges and textures, giving the image an impressionist feeling:

In [16]:

```
_=deepdream(net, img, end='conv1/7x7_s2')
# inception_4b/pool_proj
```

```
3 9 conv1/7x7_s2 (575, 766, 3)
```

We encourage readers to experiment with layer selection to see how it affects the results. Execute the next code cell to see the list of different layers. You can modify the `make_step` function to make it follow some different objective, say to select a subset of activations to maximize, or to maximize multiple layers at once. There is a huge design space to explore!

In [26]:

```
net.blobs.keys()
```

Out[26]:

```
odict_keys(['data', 'conv1/7x7_s2', 'pool1/3x3_s2', 'pool1/norm1', 'conv2/3
x3_reduce', 'conv2/3x3', 'conv2/norm2', 'pool2/3x3_s2',
'pool2/3x3_s2_pool2/3x3_s2_0_split_0',
'pool2/3x3_s2_pool2/3x3_s2_0_split_1',
'pool2/3x3_s2_pool2/3x3_s2_0_split_2',
'pool2/3x3_s2_pool2/3x3_s2_0_split_3', 'inception_3a/1x1',
'inception_3a/3x3_reduce', 'inception_3a/3x3', 'inception_3a/5x5_reduce', '
inception_3a/5x5', 'inception_3a/pool', 'inception_3a/pool_proj', 'inceptio
n_3a/output', 'inception_3a/output_inception_3a/output_0_split_0', 'incepti
on_3a/output_inception_3a/output_0_split_1',
'inception_3a/output_inception_3a/output_0_split_2',
'inception_3a/output_inception_3a/output_0_split_3', 'inception_3b/1x1', 'i
nception_3b/3x3_reduce', 'inception_3b/3x3', 'inception_3b/5x5_reduce', 'in
ception_3b/5x5', 'inception_3b/pool', 'inception_3b/pool_proj',
'inception_3b/output', 'pool3/3x3_s2',
'pool3/3x3_s2_pool3/3x3_s2_0_split_0',
'pool3/3x3_s2_pool3/3x3_s2_0_split_1',
'pool3/3x3_s2_pool3/3x3_s2_0_split_2',
'pool3/3x3_s2_pool3/3x3_s2_0_split_3', 'inception_4a/1x1',
'inception_4a/3x3_reduce', 'inception_4a/3x3', 'inception_4a/5x5_reduce', '
inception_4a/5x5', 'inception_4a/pool', 'inception_4a/pool_proj', 'inceptio
n_4a/output', 'inception_4a/output_inception_4a/output_0_split_0', 'incepti
on_4a/output_inception_4a/output_0_split_1',
'inception_4a/output_inception_4a/output_0_split_2',
'inception_4a/output_inception_4a/output_0_split_3', 'inception_4b/1x1', 'i
nception_4b/3x3_reduce', 'inception_4b/3x3', 'inception_4b/5x5_reduce', 'in
ception_4b/5x5', 'inception_4b/pool', 'inception_4b/pool_proj',
'inception_4b/output', 'inception_4b/output_inception_4b/output_0_split_0',
'inception_4b/output_inception_4b/output_0_split_1',
'inception_4b/output_inception_4b/output_0_split_2',
'inception_4b/output_inception_4b/output_0_split_3', 'inception_4c/1x1', 'i
nception_4c/3x3_reduce', 'inception_4c/3x3', 'inception_4c/5x5_reduce', 'in
ception_4c/5x5', 'inception_4c/pool', 'inception_4c/pool_proj',
'inception_4c/output', 'inception_4c/output_inception_4c/output_0_split_0',
'inception_4c/output_inception_4c/output_0_split_1',
'inception_4c/output_inception_4c/output_0_split_2',
'inception_4c/output_inception_4c/output_0_split_3', 'inception_4d/1x1', 'i
nception_4d/3x3_reduce', 'inception_4d/3x3', 'inception_4d/5x5_reduce', 'in
ception_4d/5x5', 'inception_4d/pool', 'inception_4d/pool_proj',
'inception_4d/output', 'inception_4d/output_inception_4d/output_0_split_0',
'inception_4d/output_inception_4d/output_0_split_1',
'inception_4d/output_inception_4d/output_0_split_2',
'inception_4d/output_inception_4d/output_0_split_3', 'inception_4e/1x1', 'i
nception_4e/3x3_reduce', 'inception_4e/3x3', 'inception_4e/5x5_reduce', 'in
ception_4e/5x5', 'inception_4e/pool', 'inception_4e/pool_proj',
'inception_4e/output', 'pool4/3x3_s2',
'pool4/3x3_s2_pool4/3x3_s2_0_split_0',
'pool4/3x3_s2_pool4/3x3_s2_0_split_1',
'pool4/3x3_s2_pool4/3x3_s2_0_split_2',
'pool4/3x3_s2_pool4/3x3_s2_0_split_3', 'inception_5a/1x1',
'inception_5a/3x3_reduce', 'inception_5a/3x3', 'inception_5a/5x5_reduce', '
inception_5a/5x5', 'inception_5a/pool', 'inception_5a/pool_proj', 'inceptio
n_5a/output', 'inception_5a/output_inception_5a/output_0_split_0', 'incepti
on_5a/output_inception_5a/output_0_split_1',
'inception_5a/output_inception_5a/output_0_split_2',
'inception_5a/output_inception_5a/output_0_split_3', 'inception_5b/1x1', 'i
nception_5b/3x3_reduce', 'inception_5b/3x3', 'inception_5b/5x5_reduce', 'in
```

```
ception_5b/5x5', 'inception_5b/pool', 'inception_5b/pool_proj',
'inception_5b/output', 'pool5/7x7_s1', 'loss3/classifier', 'prob'])
```

What if we feed the `deepdream` function its own output, after applying a little zoom to it? It turns out that this leads to an endless stream of impressions of the things that the network saw during training. Some patterns fire more often than others, suggestive of basins of attraction.

We will start the process from the same sky image as above, but after some iteration the original image becomes irrelevant; even random noise can be used as the starting point.

In [24]:

```
!mkdir frames
frame = img
frame_i = 0
```

In [28]:

```
h, w = frame.shape[:2]
s = 0.05 # scale coefficient
for i in range(15):
    frame = deepdream(net, frame)
    PIL.Image.fromarray(np.uint8(frame)).save("frames/%04d.jpg"%frame_i)
    frame = nd.affine_transform(frame, [1-s,1-s,1], [h*s/2,w*s/2,0], order=1
)
    frame_i += 1
```

```
3 9 inception_4c/output (575, 766, 3)
```

Be careful running the code above, it can bring you into very strange realms!

In [32]:

```
Image(filename='frames/0007.jpg')
```

Out[32]:

Controlling dreams

The image detail generation method described above tends to produce some patterns more often the others. One easy way to improve the generated image diversity is to tweak the optimization objective. Here we show just one of many ways to do that. Let's use one more input image. We'd call it a "*guide*".

In [5]:

```
guide = np.float32(PIL.Image.open('flowers.jpg'))
showarray(guide)
```

Note that the neural network we use was trained on images downscaled to 224x224 size. So high resolution images might have to be downscaled, so that the network could pick up their features. The image we use here is already small enough.

Now we pick some target layer and extract guide image features.

In [6]:

```
end = 'inception_4a/1x1'
h, w = guide.shape[:2]
src, dst = net.blobs['data'], net.blobs[end]
src.reshape(1,3,h,w)
src.data[0] = preprocess(net, guide)
net.forward(end=end)
guide_features = dst.data[0].copy()
```

Instead of maximizing the L2-norm of current image activations, we try to maximize the dot-products between activations of current image, and their best matching correspondences from the guide image.

In [9]:

```
def objective_guide(dst):
    x = dst.data[0].copy()
    y = guide_features
    ch = x.shape[0]
    x = x.reshape(ch,-1)
    y = y.reshape(ch,-1)
    A = x.T.dot(y) # compute the matrix of dot-products with guide features
    dst.diff[0].reshape(ch,-1)[:] = y[:,A.argmax(1)] # select ones that matc
h best

_=deepdream(net, img, end=end, objective=objective_guide)
```

```
3 9 inception_4a/1x1 (575, 766, 3)
```

This way we can affect the style of generated images without using a different training set.