风格迁移

这份作业里面我们会实现风格迁移的技巧，来自于这篇文章 "Image Style Transfer Using Convolutional Neural Networks" (Gatys et al., CVPR 2015).

主要的想法就是拿两张图，然后生成一张新图片来反应了一张图的内容和另一张图的艺术风格。我们首先要定义一个损失函数来在深度网络的特征空间中分别匹配相应图片的内容和风格，然后在图片本身上做梯度下降。

我们用来做特征提取器的深度网络是SqueezeNet, 一个在ImageNet上面训练的小模型。你可以用任何一个网络，但是我们选择用SqueezeNet，因为它又小又有效。

这里有一个示例图片，你将会在这个作业的最后面的时候生成它。

Style Source

Content Source

Output Image

Setup

```
%load_ext autoreload
%autoreload 2
from scipy.misc import imread, imresize
import numpy as np

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

# Helper functions to deal with image preprocessing
from cs231n.image_utils import load_image, preprocess_image, deprocess_image
import os
os.environ["CUDA_VISIBLE_DEVICES"] = "2"    # 设置使用的GPU

%matplotlib inline

def get_session():
    """Create a session that dynamically allocates memory."""
    # See: https://www.tensorflow.org/tutorials/using_gpu#allowing_gpu_memory_growth
    config = tf.ConfigProto()
    config.gpu_options.allow_growth = True
    session = tf.Session(config=config)
    return session
```

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

# 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
    vnum = int(scipy.__version__.split('.')[1])
    assert vnum >= 16, "You must install SciPy >= 0.16.0 to complete this notebook."

check_scipy()
```

加载预训练的SqueezeNet模型，这个模型是从PyTorch上转换过来的，详见`cs231n/classifiers/squeezenet.py`参考模型架构。

要想使用SqueezeNet, 你首先需要下载模型参数。切换到`cs231n/datasets`目录下，运行`get_squeezenet_tf.sh`。注意，如果你之前跑过`get_assignment3_data.sh`，那么SqueezeNet应该已经下载好了。

```
from cs231n.classifiers.squeezenet import SqueezeNet
import tensorflow as tf

tf.reset_default_graph() # remove all existing variables in the graph
sess = get_session() # start a new Session

# Load pretrained SqueezeNet model
SAVE_PATH = 'cs231n/datasets/squeezenet.ckpt'
#if not os.path.exists(SAVE_PATH):
#    raise ValueError("You need to download SqueezeNet!")
model = SqueezeNet(save_path=SAVE_PATH, sess=sess)

# Load data for testing
content_img_test = preprocess_image(load_image('styles/tubingen.jpg', size=192))[None]
style_img_test = preprocess_image(load_image('styles/starry_night.jpg', size=192))[None]
answers = np.load('style-transfer-checks-tf.npz')
```

```
INFO:tensorflow:Restoring parameters from cs231n/datasets/squeezenet.ckpt
```

计算损失函数

我们将计算我们损失函数的三个组成部分。损失函数是三个部分的加权和: 内容损失 + 风格损失 + 整体多样性损失。 你会在下面的函数中计算这些加权的部分。

内容损失

我们希望生成一张图片，这张图片可以反映一张图片的内容和另一个张图片的风格。为此，我们要把这二者都加入到我们的损失函数中去。我们希望可以惩罚对于内容图片的内容的偏移，以及对风格图片的风格偏移。我们可以用这样的混合的损失函数来进行梯度下降，注意不是在模型的参数上，而是在我们的原始图片的像素值上进行梯度下降。

我们首先来写一下内容的损失函数。 内容损失衡量的是生成的图片的特征图和源图片的特征图的差异程度。 我们只关心网络的某一个层的内容表示（比如，层 ℓ ）,特征图是$A^\ell \in \mathbb{R}^{1\times C_\ell \times H_\ell \times W_\ell}$. C_ℓ 是ℓ这层的通道(filters/channels)数量, H_ℓ and W_ℓ 是高和宽. 我们将会在reshape之后的特征图上来计算(把二维图像拉伸成一维)。假设$F^\ell \in \mathbb{R}^{N_\ell \times M_\ell}$ 是当前图片的特征图 以及 $P^\ell \in \mathbb{R}^{N_\ell \times M_\ell}$ 是内容的源图片的特征图 其中 $M_\ell = H_\ell \times W_\ell$ 是每个特征图中的元素个数. 每一行的 F^ℓ or P^ℓ 表示的是一个特定的filter在图片的所有位置上卷积之后的向量化的激活值，最后, 假设 w_c 是损失函数中的内容损失的权重。

那么内容损失值也就是:

$L_c = w_c \times \sum_{i,j}(F_{ij}^\ell - P_{ij}^\ell)^2$

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

    Inputs:
    - content_weight: scalar constant we multiply the content_loss by.
    - content_current: features of the current image, Tensor with shape [1, height, width,
channels]
    - content_target: features of the content image, Tensor with shape [1, height, width,
channels]

    Returns:
    - scalar content loss
    """
    # tf.squared_difference(x,y,name=None) 返回的是(x-y)(x-y)
    return content_weight * tf.reduce_sum(tf.squared_difference(content_current,
content_original))
```

测试你的内容损失，你应该可以看到错误小于0.001。

```
def content_loss_test(correct):
    content_layer = 3
    content_weight = 6e-2
    c_feats = sess.run(model.extract_features()[content_layer], {model.image: content_img_test})
    bad_img = tf.zeros(content_img_test.shape)
    feats = model.extract_features(bad_img)[content_layer]
    student_output = sess.run(content_loss(content_weight, c_feats, feats))
    error = rel_error(correct, student_output)
    print('Maximum error is {:.3f}'.format(error))

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

```
Maximum error is 0.000
```

风格损失

现在我们可以来解决风格损失了。 对于一个给定的层ℓ， 风格损失定义如下:

首先计算Gram矩阵G，代表的是每个filter之间的相关性，其中F和上面的定义一样。Gram矩阵是协方差矩阵的一个近似，我们希望我们生成的图片的激活值的统计信息和我们的风格图片的激活值的统计信息相匹配，使(近似)协方差相匹配就是其中的一种方法。你有很多的方法可以来实现这一点，但是Gram矩阵比较好的一点是实践中计算简单并且效果比较好。

给定一个特征图F^ℓ ，尺寸是 $(1, C_\ell, M_\ell)$, Gram矩阵是$(1, C_\ell, C_\ell)$ 它的元素计算如下:

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

假设 G^ℓ 是当前图片的特征图的Gram矩阵, A^ℓ 是源风格图片的特征图Gram矩阵，以及w_ℓ 是权重值, 那么对于层ℓ的风格损失就是两个Gram矩阵之间的加权欧几里德距离:

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

实践中，我们会计算好几个层 $\mathcal{L}$ 而不仅仅是一个层 ℓ; 那么总的风格损失就是每个层的风格损失之和:

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

我们先开始计算gram矩阵:

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

    Inputs:
    - features: Tensor of shape (1, H, W, C) giving features for
      a single image.
    - normalize: optional, whether to normalize the Gram matrix
        If True, divide the Gram matrix by the number of neurons (H * W * C)

    Returns:
    - gram: Tensor of shape (C, C) giving the (optionally normalized)
      Gram matrices for the input image.
    """

    features = tf.transpose(features, [0,3,1,2]) # ( 1 , C, H, W )
    shape = tf.shape(features)
    features = tf.reshape(features, (shape[0],shape[1],-1)) # 合并H和W (1,C, H*W)
    transpose_features = tf.transpose(features,[0,2,1]) # (1, H*W, C)

    result = tf.matmul(features,transpose_features)
    if normalize:
        result = tf.div(result,tf.cast(shape[0]*shape[1]*shape[2]*shape[3],tf.float32))
    return result
```

测试Gram矩阵代码，你应该可以看到错误小于0.001。

```
def gram_matrix_test(correct):
    gram = gram_matrix(model.extract_features()[5])
    student_output = sess.run(gram, {model.image: style_img_test})
    error = rel_error(correct, student_output)
    print('Maximum error is {:.3f}'.format(error))

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

```
Maximum error is 0.000
```

接下来，实现风格损失:

```
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 Tensor 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 Tensor contataining the scalar style loss.
    """
    # Hint: you can do this with one for loop over the style layers, and should
    # not be very much code (~5 lines). You will need to use your gram_matrix function.
    style_losses = 0
    for i in range(len(style_layers)):
        cur_index = style_layers[i]
        cur_feat = feats[cur_index]
        cur_weight = style_weights[i]
        cur_style_target = style_targets[i] # 注意它已经是一个Gram矩阵了
        gramMatrix = gram_matrix(cur_feat) #计算当前层的特征图的gram矩阵
        style_losses += cur_weight * tf.reduce_sum(tf.squared_difference(gramMatrix,
cur_style_target))
    return style_losses
```

测试你的风格损失的实现。误差应当小于0.001。

```
def style_loss_test(correct):
    style_layers = [1, 4, 6, 7]
```

```
    style_weights = [300000, 1000, 15, 3]

    feats = model.extract_features()
    style_target_vars = []
    for idx in style_layers:
        style_target_vars.append(gram_matrix(feats[idx]))
    style_targets = sess.run(style_target_vars,
                             {model.image: style_img_test})

    s_loss = style_loss(feats, style_layers, style_targets, style_weights)
    student_output = sess.run(s_loss, {model.image: content_img_test})
    error = rel_error(correct, student_output)
    print('Error is {:.3f}'.format(error))

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

```
Error is 0.000
```

整体方差的正则化（TV loss）

事实证明在图片中加入平滑也是很有帮助的。我们可以在损失函数中加入另一项来惩罚wiggles(类似于波形的形状),或者说是像素值的整体变化(total variation)。

你可以计算水平或者垂直方向的所有的像素对之间的值的差的平方和作为整体的像素值的变化(total variation)。这里我们为每个RGB通道都计算一个总体的像素值变化的正则值，然后求和，并且用权重w_t来加权:

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

在下面，完成TV损失项(total variation regularization)的定义， 为了得到满分，你不可以用循环。

```
def tv_loss(img, tv_weight):
    """
    Compute total variation loss.

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

    Returns:
    - loss: Tensor holding a scalar giving the total variation loss
      for img weighted by tv_weight.
    """
    # Your implementation should be vectorized and not require any loops!
    shape = tf.shape(img)
    img_row_before = tf.slice(img, [0,0,0,0],[-1,shape[1]-1, -1, -1])
    img_row_after = tf.slice(img,[0,1,0,0],[-1,shape[1]-1, -1,-1])
    img_col_before = tf.slice(img,[0,0,0,0],[-1,-1,shape[2]-1,-1])
    img_col_after = tf.slice(img,[0,0,1,0],[-1,-1,shape[2]-1,-1])
    #print(img_row_before.get_shape())

    result = tv_weight *(tf.reduce_sum(tf.squared_difference(img_row_before, img_row_after)) +
```

```
tf.reduce_sum(tf.squared_difference(img_col_before, img_col_after)))
    return result
```

测试你的TV loss 实现。误差应当小于0.001。

```
def tv_loss_test(correct):
    tv_weight = 2e-2
    t_loss = tv_loss(model.image, tv_weight)
    student_output = sess.run(t_loss, {model.image: content_img_test})
    print(student_output)
    print(correct)
    error = rel_error(correct, student_output)
    print('Error is {:.3f}'.format(error))

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

```
303.47
303.223052979
Error is 0.000
```

风格迁移

把这些都组合在一起来生成一些好看的图片! 下面的`style_transfer`方程 把你前面写的所有的损失都合在了一起，然后优化一张图片来最小化整体的损失值。

```
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 from the content image
    content_img = preprocess_image(load_image(content_image, size=image_size))
    feats = model.extract_features(model.image)
    content_target = sess.run(feats[content_layer],

                              {model.image: content_img[None]})
```

```
# Extract features from the style image
style_img = preprocess_image(load_image(style_image, size=style_size))
style_feat_vars = [feats[idx] for idx in style_layers]
style_target_vars = []
# Compute list of TensorFlow Gram matrices
for style_feat_var in style_feat_vars:
    style_target_vars.append(gram_matrix(style_feat_var))
# Compute list of NumPy Gram matrices by evaluating the TensorFlow graph on the style image
style_targets = sess.run(style_target_vars, {model.image: style_img[None]})

# Initialize generated image to content image

if init_random:
    img_var = tf.Variable(tf.random_uniform(content_img[None].shape, 0, 1), name="image")
else:
    img_var = tf.Variable(content_img[None], name="image")

# Extract features on generated image
feats = model.extract_features(img_var)
# Compute loss
c_loss = content_loss(content_weight, feats[content_layer], content_target)
s_loss = style_loss(feats, style_layers, style_targets, style_weights)
t_loss = tv_loss(img_var, tv_weight)
loss = c_loss + s_loss + t_loss

# Set up optimization hyperparameters
initial_lr = 3.0
decayed_lr = 0.1
decay_lr_at = 180
max_iter = 200

# Create and initialize the Adam optimizer
lr_var = tf.Variable(initial_lr, name="lr")
# Create train_op that updates the generated image when run
with tf.variable_scope("optimizer") as opt_scope:
    train_op = tf.train.AdamOptimizer(lr_var).minimize(loss, var_list=[img_var])
# Initialize the generated image and optimization variables
opt_vars = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES, scope=opt_scope.name)
sess.run(tf.variables_initializer([lr_var, img_var] + opt_vars))
# Create an op that will clamp the image values when run
clamp_image_op = tf.assign(img_var, tf.clip_by_value(img_var, -1.5, 1.5))

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_image(content_img))
axarr[1].imshow(deprocess_image(style_img))
plt.show()
plt.figure()
```

```
    # Hardcoded handcrafted
    for t in range(max_iter):
        # Take an optimization step to update img_var
        sess.run(train_op)
        if t < decay_lr_at:
            sess.run(clamp_image_op)
        if t == decay_lr_at:
            sess.run(tf.assign(lr_var, decayed_lr))
        if t % 100 == 0:
            print('Iteration {}'.format(t))
            img = sess.run(img_var)
            plt.imshow(deprocess_image(img[0], rescale=True))
            plt.axis('off')
            plt.show()
    print('Iteration {}'.format(t))
    img = sess.run(img_var)
    plt.imshow(deprocess_image(img[0], rescale=True))
    plt.axis('off')
    plt.show()
```

生成一些有趣的图片!

试着在下面三组不同的参数上运行`style_transfer`函数。确保三组都运行。随意加上你自己的参数，但是一定要确保在你提交的Notebook中包含了在第三组参数(星空)上运行的结果。

- `content_image` 内容图片
- `style_image` 风格图片
- `image_size` 内容图片最小的纬度大小(用于内容损失和生成的图片)
- `style_size` 风格图片最小的纬度大小
- `content_layer` 指定了对于内容损失用哪个层
- `content_weight` 内容损失在全局损失函数上的权重，增加这个参数的值会让图片看起来更加接近原图
- `style_layers` 指定了风格损失需要用哪些层
- `style_weights` 为style_layers中的每个层指定了一系列的权重，我们通常会在越前面的风格层上用更大的权重因为它们描述了更加局部/小的特征，对于图片的纹理比后面更大的接收阈上的特征更加重要。总之，增加这些权重会让结果图片看起来更加的不像原始图片，而更加向风格图片靠近。
- `tv_weight` 指定了图片总体的变化正则在总体的损失函数上的权重。增加这个值会让结果图片看起来更加的光滑，但是代价是会在风格和内容上更加失真。

下面三部分代码(不要改变超参数)，随意复制黏贴各种参数来运行，并看看结果图片。

```
# 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
```

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

style_transfer(**params2)
```

Content Source Img.

Style Source Img.

```
Iteration 0
```

```
Iteration 100
```

```
Iteration 199
```

```
# 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
```

特征反推

你写的代码还可以做些其他有趣的事情。 为了理解卷积网络学习出来的特征的类型， 最近的一篇论文[1]尝试从图片的特征表示来还原一张图片。 我们用预训练的网络上的图片梯度很容易就可以实现这个想法了， 这其实也就是我们前面在做的(但是上面是两个不同的特征表示)。

现在，如果你把风格权重设置为0，并且初始化开始的图片为随机噪音而不是内容源图片，你就可以从内容原图片的特征表示上来重建这张图片。你初始化是用的随机噪音，但是最后你得到的应该是一张看起来很像你原先图片的图片。

(类似的， 你可以做"纹理合成"如果你把内容的权重设置成0，然后初始化图片为随机噪音，但是我们这里并不要求)

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

```
# 从内容上重建图片
# 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
```

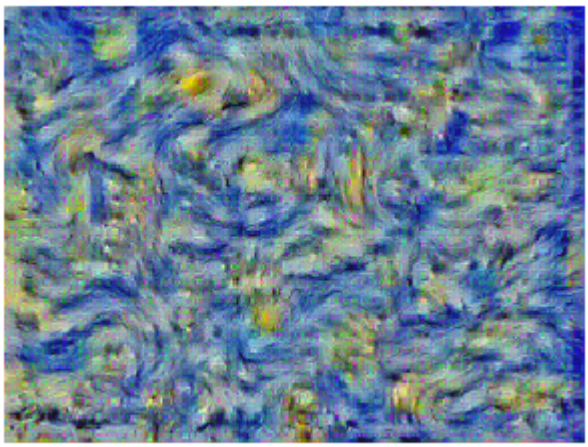
```
Iteration 199
```

```
# 从风格上显示图片纹理
# 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' : 0,
    'style_layers' : [1, 4, 6, 7],
    'style_weights' : [300000, 1000, 15, 3], # 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
```