《计算机图形学实验》综合实验报告

题目	基于神经网络的图像风格迁移

学 号	20191060003
姓 名	刘文长
指导教师	钱文华
日 期	2022.6.8

摘要

风格迁移，其实就是提供一幅画(Reference style image)，将任意一张照片转化成这个风格，并尽量保留原照的内容(Content)。之前比较火的修图软件 Prisma 就提供了这个功能。即风格迁移，将一张图的风格迁移到另一张图片上，也可以理解为生成问题，根据两种图片，生成第三种（风格）图片然而，原始的风格迁移的速度是非常慢的。在 GPU 上，生成一张图片都需要 10 分钟左右，而如果只使用 CPU 而不使用 GPU 运行程序，甚至需要几个小时。这个时间还会随着图片尺寸的增大而迅速增大。这其中的原因在于，在原始的风格迁移过程中，把生成图片的过程当做一个“训练”的过程。每生成一张图片，都相当于要训练一次模型，这中间可能会迭代几百几千次。但从头训练一个模型要比执行一个已经训练好的模型要费时太多。而这也正是原始的风格迁移速度缓慢的原因。所以采用预训练的 VGG19 模型来进行图像风格迁移实验。

关键词：神经网络；风格迁移；预训练

目录

1. 实验背景

图像风格迁移指的是通过某种方法，把图像从原风格转换到另外一个风格，同时保证图像内容没有变化。

假设没有现在的方法，历史上的方法会从两种思路下手：

将两幅图像叠加；这是最简洁且看上去效果还有点相似的方法，但是它有个很大的问题是，图像风格改变的同时原图像的内容也改变了，并不符合图像风格迁移的定义；

先分析给定风格的图像，统计其像素分布的数学性质，建立数学模型，再分析要做迁移的图像，使其贴合特定的模型；这看上去可行，但是每换一种风格，就要换一种模型，并不具有应用意义；

2. 实验内容

利用深度学习算法，实现图像识别、图像分类、图像分割、图像生成、风格迁移、图像分辨率增强等，要求根据公有数据集或者自建数据集，完成深度学习的网络搭建、网络训练、网络测试等，获得实验结果。

3. 开发工具，程序设计及实现目的

开发工具：python,pycharm,anaconda,tensorflow

首先设置参数并加载需要风格迁移的图片：

```
# 参数
alpha = 10
beta = 40
iterations = 200

content_image = imageio.imread("images/content.jpg")
content_image = reshape_and_normalize_image(content_image)
style_image = imageio.imread("images/style.jpg")
style_image = reshape_and_normalize_image(style_image)
generated_image = generate_noise_image(content_image)
```

然后重置图表，初始化噪声图像，并加载 VGG19 预训练模型：

```
# 重置图表
tf.reset_default_graph()

# 开始互动会话
sess = tf.InteractiveSession()

# 通过向 content_image 添加随机噪声来初始化噪声图像
generated_image = generate_noise_image(content_image)

# 加载 VGG19 模型
model = load_vgg_model("pretrained-model/imagenet-vgg-verydeep-19.mat")
```

将内容图像输入到 VGG 模型，并计算内容成本：

```
# 将内容图像分配为 VGG 模型的输入。
sess.run(model['input'].assign(content_image))

# 选择层conv4_2的输出张量
out = model['conv4_2']

# 将 a_C 设置为我们选择的层的隐藏层激活
a_C = sess.run(out)

# 将 a_G 设置为来自同一层的隐藏层激活。在这里，a_G 引用了模型['conv4_2']
# 并且尚未评估。稍后在代码中，将图像 G 指定为模型输入，这样
# 当运行会话时，这将是从适当的层提取的激活，以 G 作为输入。
a_G = out

# 计算内容成本
J_content = compute_content_cost(a_C, a_G)
```

将风格图像输入，并计算样式成本：

```
# 将模型的输入分配为“风格”图像
sess.run(model['input'].assign(style_image))

# 计算样式成本
J_style = compute_style_cost(sess, model)

J = total_cost(J_content, J_style, alpha, beta)  # 10,40
```

定义优化器和训练步长

```
# define optimizer (1 line)
optimizer = tf.train.AdamOptimizer(2.0)

# define train_step (1 line)
train_step = optimizer.minimize(J)
```

初始化全局变量：

```
# Initialize global variables (you need to run the session on the initializer)
sess.run(tf.global_variables_initializer())

# Run the noisy input image (initial generated image) through the model. Use assign().
sess.run(model["input"].assign(generated_image))
```

迭代计算生成图像：

```
for i in range(num_iterations):

    # 在 train_step 上运行会话以最小化总成本
    sess.run(train_step)

    # 通过在当前模型上运行会话来计算生成的图像['input']
    generated_image = sess.run(model["input"])

    # 每 20 次迭代打印一次。
    if i % 20 == 0:
        Jt, Jc, Js = sess.run([J, J_content, J_style])
        print("Iteration " + str(i) + " :")
        print("total cost = " + str(Jt))
        print("content cost = " + str(Jc))
        print("style cost = " + str(Js))

        # 将当前生成的图像保存在“/output”目录中
        save_image("images/" + str(i) + ".png", generated_image)
```

最后保存生成图像：

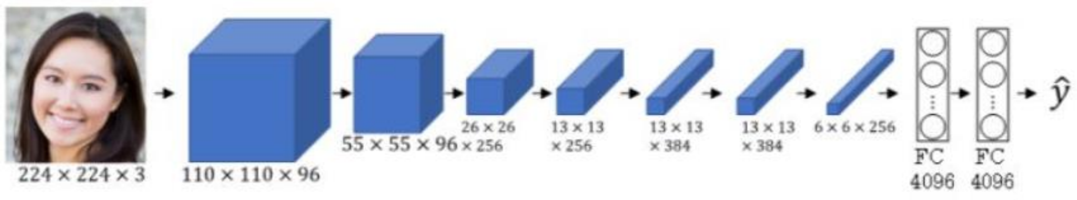
```
    # 保存最后生成的图像

    save_image("images/output.png", generated_image)

    return generated_image
```

4. 关键算法理论

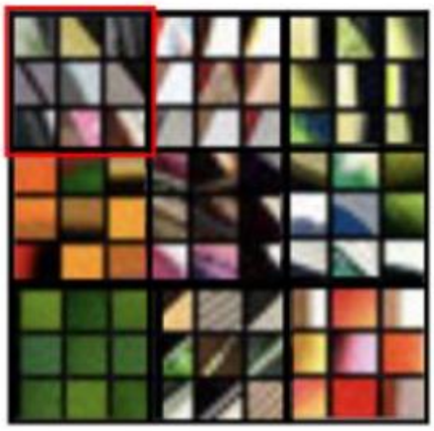
神经风格迁移是 CNN 模型一个非常有趣的应用。它可以实现将一张图片的风格“迁移”到另外一张图片中，生成具有其特色的图片。

Content
Style
Generated image
Content
Style
Generated image

典型的 CNN 网络如下所示：



首先来看第一层隐藏层，遍历所有训练样本，找出让该层激活函数输出最大的 9 块图像区域；然后再找出该层的其它单元（不同的滤波器通道）激活函数输出最大的 9 块图像区域；最后共找 9 次，得到 9 x 9 的图像如下所示，其中每个 3 x 3 区域表示一个运算单元。



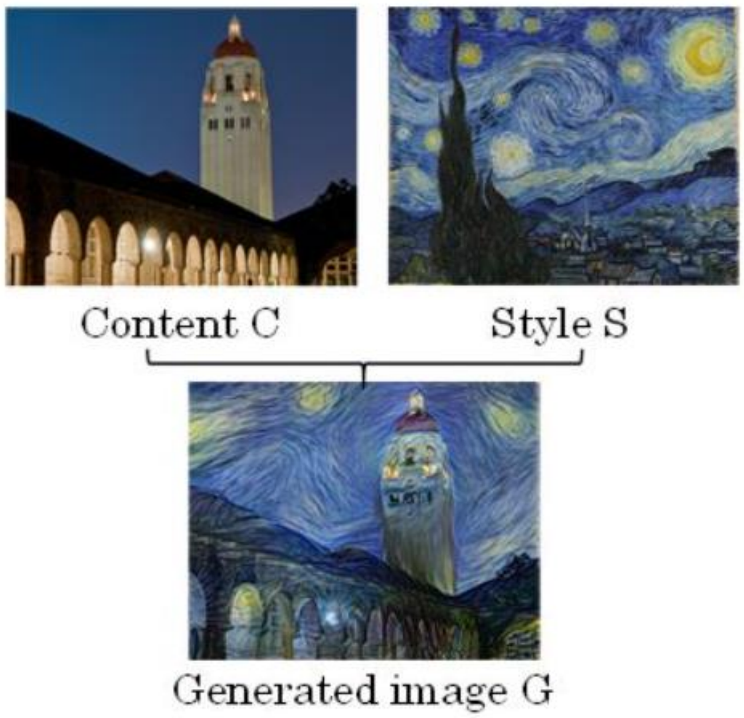
可以看出，第一层隐藏层一般检测的是原始图像的边缘和颜色阴影等简单信息。

继续看 CNN 的更深隐藏层，随着层数的增加，捕捉的区域更大，特征更加复杂，从边缘到纹理再到具体物体。

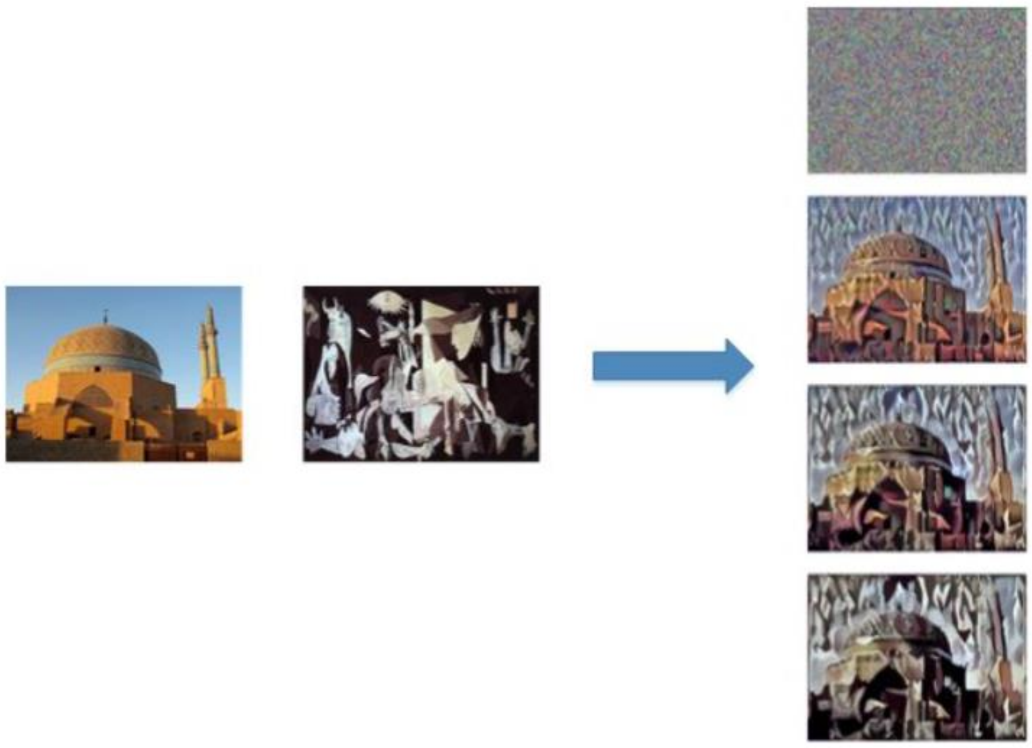
Layer 1
Layer 2
Layer 3
Layer 4
Layer 5

神经风格迁移生成图片 G 的 cost function 由两部分组成：C 与 G 的相似程度和 S 与 G 的相似程度。

$$J(G) = \alpha \cdot J_{content}(C, G) + \beta \cdot J_{style}(S, G)$$



神经风格迁移的基本算法流程是：首先令 G 为随机像素点，然后使用梯度下降算法，不断修正 G 的所有像素点，使得$J(G)$不断减小，从而使 G 逐渐有 C 的内容和 G 的风格，如下图所示。

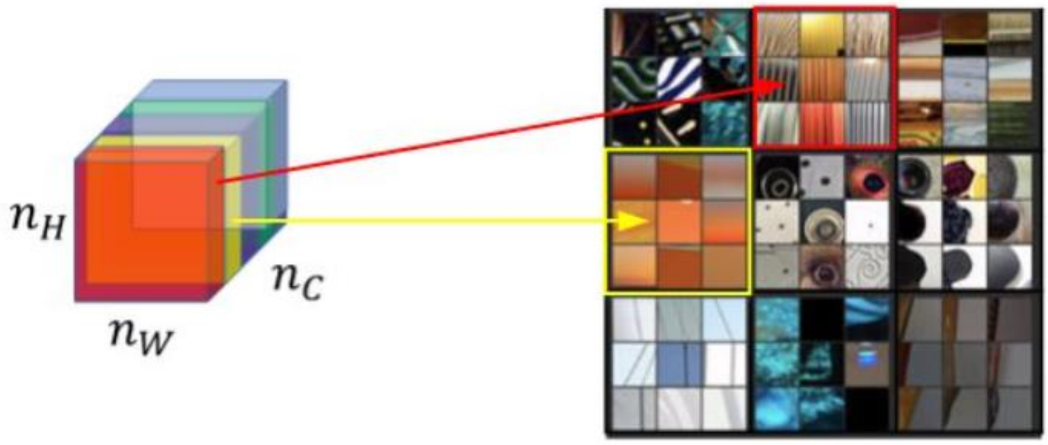


然后比较 C 和 G 在 l 层的激活函数输出

$$J_{content}(C,G)=\frac{1}{2}||a^{[l](C)}-a^{[l](G)}||^2$$

例如我们选取第 l 层隐藏层，其各通道使用不同颜色标注，如下图所示。因为每个通道提取图片的特征不同，比如 1 通道（红色）提取的是图片的垂直纹理特征，2 通道（黄色）提取的是图片的橙色背景特征。那么计算这两个通道的相关性大小，相关性越大，表示原始图片及既包含了垂直纹理也包含了该橙色背景；相关性越

小，表示原始图片并没有同时包含这两个特征。也就是说，计算不同通道的相关性，反映了原始图片特征间的相互关系，从某种程度上刻画了图片的“风格”。



接下来我们就可以定义图片的风格矩阵（style matrix）为：

$$G_{kk'}^{[l]} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_W^{[l]}} a_{ijk}^{[l]} a_{ijk'}^{[l]}$$

为了提取的“风格”更多，也可以使用多层隐藏层，然后相加，表达式为：

$$J_{style}(S,G) = \sum_{l} \lambda^{[l]} \cdot J_{style}^{[l]}(S,G)$$

最终的 cost function 为：

$$J(G) = \alpha \cdot J_{content}(C,G) + \beta \cdot J_{style}(S,G)$$

使用梯度下降算法进行迭代优化。

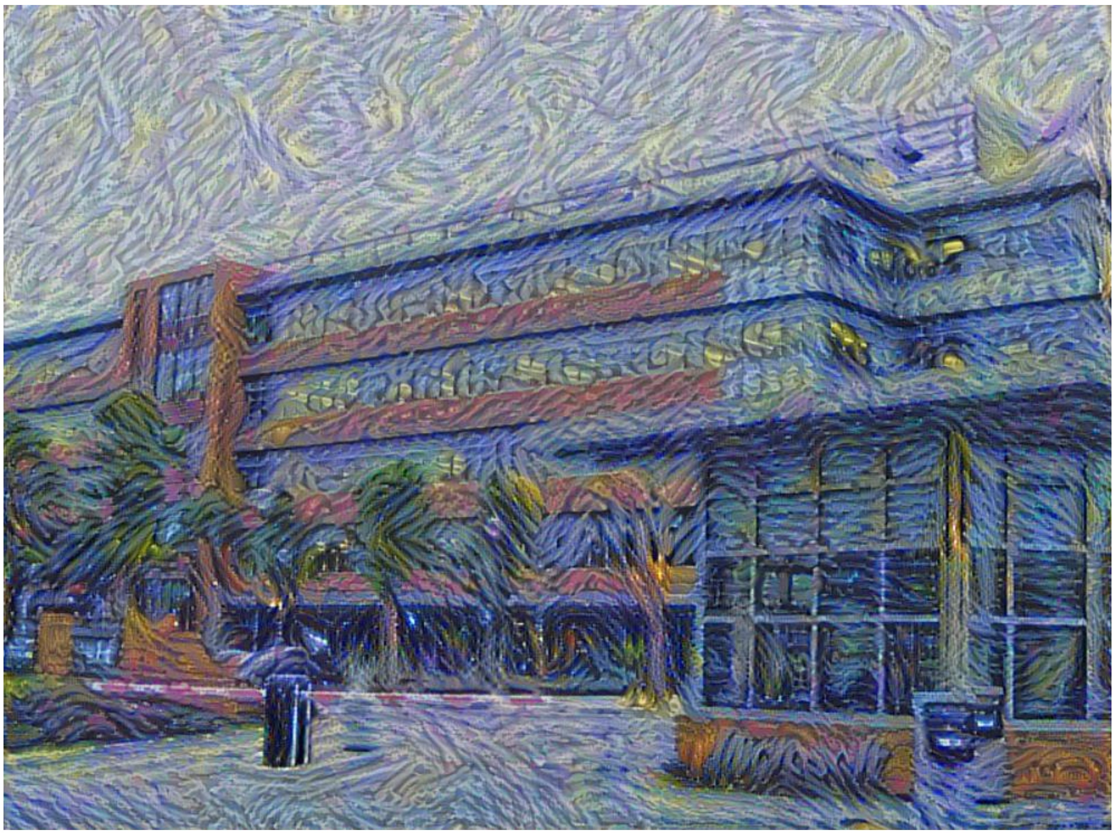
5. 实验心得及小结

实验结果

内容图片:

风格图片:

合成图片：



心得体会：

通过一个学期的学习，了解了什么是计算机图形学、什么是图形API、为什么需要计算机图形学以及计算机图形学在各个领域的应用。计算机图形学是一种使用数学算法将二维或三维图形转化为计算机显示器的栅格形式的科学，研究的是应用计算机产生图像的所有工作，不管图像是静态的还是动态的，可交互的还是固定的，等等。图形API是允许程序员开发包含交互式计算机图形操作的应用而不需要关注图形操作细节或任务系统细节的工具集。计算机图形学有着广泛的应用领域，包括物理、航天、电影、电视、游戏、艺术、广告、通信、天气预报等几乎所有领域都用到了计算机图形学的知识，这些领域通过计算机图形学将

几何模型生成图像，将问题可视化从而为各领域更好的服务。

通过这些图形学基础的学习使我有了理论基础，而且有了浓厚的兴趣，所以我选择期末做深度学习的图像风格迁移，通过完成此次实验使我的图形学知识得到了扩展和巩固。

参考文献：

[1]吴恩达 Convolutional Neural Networks

[2]吴子扬,贺丹,李映琴. 基于 VGG-19 神经网络模型的图像风格迁移[J]. 科技与创新,2021(13):171-173. DOI:10.15913/j.cnki.kjycx.2021.13.073.

[3]李文书,赵朋,尹灵芝,等. 基于高斯采样的区域多元化图像风格迁移方法[J]. 计算机辅助设计与图形学学报,2022,34(5):743-750. DOI:10.3724/SP.J.1089.2022.19027.

[4]王茜. 图像风格迁移技术研究[J]. 吕梁学院学报,2022,12(2):37-39. DOI:10.3969/j.issn.2095-185X.2022.02.011.

[5]陈淮源,张广驰,陈高,等. 基于深度学习的图像风格迁移研究进展[J]. 计算机工程与应用,2021,57(11):37-45. DOI:10.3778/j.issn.1002-8331.2101-0019.

附录：

下载预训练的 VGG19 模型放入 pretrained-model 目录，然后执行：

GitHub 地址

https://github.com/Liu-Vince/Neural-Style-Transfer

main.py

```
import tensorflow as tf
from tensorflow.python.framework import ops
ops.reset_default_graph()
import imageio


from utils import compute_content_cost, compute_style_cost, total_cost, generate_noise_image, save_image, \
    reshape_and_normalize_image, load_vgg_model


def transfer(num_iterations=200):
    # 重置图表
    tf.reset_default_graph()

    # 开始互动会话
    sess = tf.InteractiveSession()

    # 通过向 content_image 添加随机噪声来初始化噪声图像
    generated_image = generate_noise_image(content_image)

    # 加载 VGG19 模型
    model = load_vgg_model("pretrained-model/imagenet-vgg-verydeep-19.mat")

    # 将内容图像分配为 VGG 模型的输入。
    sess.run(model['input'].assign(content_image))

    # 选择层 conv4_2 的输出张量
    out = model['conv4_2']

    # 将 a_C 设置为我们选择的层的隐藏层激活
    a_C = sess.run(out)

    # 将 a_G 设置为来自同一层的隐藏层激活。在这里，a_G 引用了模型['conv4_2']
    # 并且尚未评估。稍后在代码中，将图像 G 指定为模型输入，这样
    # 当运行会话时，这将是从适当的层提取的激活，以 G 作为输入。
    a_G = out

    # 计算内容成本
    J_content = compute_content_cost(a_C, a_G)

    # 将模型的输入分配为“风格”图像
    sess.run(model['input'].assign(style_image))

    # 计算样式成本
    J_style = compute_style_cost(sess, model)

    J = total_cost(J_content, J_style, alpha, beta)  # 10,40

    # define optimizer (1 line)
    optimizer = tf.train.AdamOptimizer(2.0)

    # define train_step (1 line)
    train_step = optimizer.minimize(J)

    # Initialize global variables (you need to run the session on the initializer)
    sess.run(tf.global_variables_initializer())

    # Run the noisy input image (initial generated image) through the model. Use assign().
    sess.run(model["input"].assign(generated_image))

    for i in range(num_iterations):

        # 在 train_step 上运行会话以最小化总成本
        sess.run(train_step)

        # 通过在当前模型上运行会话来计算生成的图像['input']
        generated_image = sess.run(model["input"])

        # 每 20 次迭代打印一次。
        if i % 20 == 0:
            Jt, Jc, Js = sess.run([J, J_content, J_style])
            print("Iteration " + str(i) + " :")
            print("total cost = " + str(Jt))
            print("content cost = " + str(Jc))
```

```python
            print("style cost = " + str(Js))

            # 将当前生成的图像保存在“/output”目录中
            save_image("images/" + str(i) + ".png", generated_image)

    # 保存最后生成的图像

    save_image("images/output.png", generated_image)

    return generated_image


if __name__ == '__main__':
    # 参数
    alpha = 10
    beta = 40
    iterations = 200

    content_image = imageio.imread("images/content.jpg")
    content_image = reshape_and_normalize_image(content_image)
    style_image = imageio.imread("images/style.jpg")
    style_image = reshape_and_normalize_image(style_image)
    generated_image = generate_noise_image(content_image)

    transfer()
```

utils.py

```python
import matplotlib.pyplot as plt
import numpy as np
import scipy.io
import scipy.misc
import tensorflow as tf

IMAGE_HEIGHT = 600
IMAGE_WIDTH = 800
COLOR_CHANNELS = 3


def compute_content_cost(a_C, a_G):
    """
    Computes the content cost
    Arguments:
    a_C -- tensor of dimension (1, n_H, n_W, n_C), hidden layer activations representing content of the image C
    a_G -- tensor of dimension (1, n_H, n_W, n_C), hidden layer activations representing content of the image G
    Returns:
    J_content -- scalar that you compute using equation 1 above.
    """

    # Retrieve dimensions from a_G (≈1 line)
    m, n_H, n_W, n_C = a_G.get_shape().as_list()

    # Reshape a_C and a_G (≈2 lines)
    a_C_unrolled = tf.transpose(tf.reshape(a_C, [n_H * n_W, n_C]))
    a_G_unrolled = tf.transpose(tf.reshape(a_G, [n_H * n_W, n_C]))

    # compute the cost with tensorflow (≈1 line)
    J_content = (1 / (4 * n_H * n_W * n_C)) * tf.reduce_sum(tf.square(tf.subtract(a_C_unrolled, a_G_unrolled)))

    return J_content


def gram_matrix(A):
    """
    Argument:
    A -- matrix of shape (n_C, n_H*n_W)
    Returns:
    GA -- Gram matrix of A, of shape (n_C, n_C)
    """
    GA = tf.matmul(A, tf.transpose(A))

    return GA


def compute_layer_style_cost(a_S, a_G):
    """
    Arguments:
```

```python
    a_S -- tensor of dimension (1, n_H, n_W, n_C), hidden layer activations representing style of the image S
    a_G -- tensor of dimension (1, n_H, n_W, n_C), hidden layer activations representing style of the image G
    Returns:
    J_style_layer -- tensor representing a scalar value, style cost defined above by equation (2)
    """

    # Retrieve dimensions from a_G (≈1 line)
    m, n_H, n_W, n_C = a_G.get_shape().as_list()

    # Reshape the images to have them of shape (n_H*n_W, n_C) (≈2 lines)
    a_S = tf.transpose(tf.reshape(a_S, [n_H * n_W, n_C]))
    a_G = tf.transpose(tf.reshape(a_G, [n_H * n_W, n_C]))

    # Computing gram_matrices for both images S and G (≈2 lines)
    GS = gram_matrix(a_S)
    GG = gram_matrix(a_G)

    # Computing the loss (≈1 line)
    J_style_layer = (1 / (4 * n_C * n_C * (n_H * n_W) * (n_H * n_W))) * (tf.reduce_sum(tf.square(tf.subtract(GS, GG))))

    return J_style_layer


def compute_style_cost(sess, model):
    """
    Computes the overall style cost from several chosen layers
    Arguments:
    model -- our tensorflow model
    STYLE_LAYERS -- A python list containing:
                        - the names of the layers we would like to extract style from
                        - a coefficient for each of them
    Returns:
```

```python
    J_style -- tensor representing a scalar value, style cost defined above by equation (2)
    """

    # initialize the overall style cost
    J_style = 0
    STYLE_LAYERS = [
        ('conv1_1', 0.2),
        ('conv2_1', 0.2),
        ('conv3_1', 0.2),
        ('conv4_1', 0.2),
        ('conv5_1', 0.2)]

    for layer_name, coeff in STYLE_LAYERS:
        # Select the output tensor of the currently selected layer
        out = model[layer_name]

        # Set a_S to be the hidden layer activation from the layer we have selected, by running the session on out
        a_S = sess.run(out)

        # Set a_G to be the hidden layer activation from same layer. Here, a_G references model[layer_name]
        # and isn't evaluated yet. Later in the code, we'll assign the image G as the model input, so that
        # when we run the session, this will be the activations drawn from the appropriate layer, with G as input.
        a_G = out

        # Compute style_cost for the current layer
        J_style_layer = compute_layer_style_cost(a_S, a_G)

        # Add coeff * J_style_layer of this layer to overall style cost
        J_style += coeff * J_style_layer

    return J_style


def total_cost(J_content, J_style, alpha=10, beta=40):
    """
    Computes the total cost function
    Arguments:
    J_content -- content cost coded above
    J_style -- style cost coded above
    alpha -- hyperparameter weighting the importance of the content cost
    beta -- hyperparameter weighting the importance of the style cost
    Returns:
    J -- total cost as defined by the formula above.
    """

    J = alpha * J_content + beta * J_style

    return J


def show_images(images, row=1, col=3):
    # Split the figure in rows and columns to put images
    figure, axes = plt.subplots(row, col)

    for i, ax in enumerate(axes.flat):
        plt.subplot(ax)
        plt.imshow(images[i])
        ax.set_xticks([])
        ax.set_yticks([])

    plt.show()


def generate_noise_image(content_image, noise_ratio=0.6):
    """
    Generates a noisy image by adding random noise to the content_image
    """

    # Generate a random noise_image
    noise_image = np.random.uniform(-20, 20,
                                    (1, IMAGE_HEIGHT, IMAGE_WIDTH, COLOR_CHANNELS)).astype(
        'float32')

    # Set the input_image to be a weighted average of the content_image and a noise_image
    input_image = noise_image * noise_ratio + content_image * (1 - noise_ratio)

    return input_image


def reshape_and_normalize_image(image):
    """
    Reshape and normalize the input image (content or style)
    """

    # Reshape image to mach expected input of VGG16
    image = np.reshape(image, ((1,) + image.shape))

    # Substract the mean to match the expected input of VGG16
    image = image - [[[[123.68, 116.779, 103.939]]]]

    return image


def save_image(path, image):
    # Un-normalize the image so that it looks good
    image = image + [[[[123.68, 116.779, 103.939]]]]

    # Clip and Save the image
    image = np.clip(image[0], 0, 255).astype('uint8')
    scipy.misc.imsave(path, image)


def load_vgg_model(path):
    """
    Returns a model for the purpose of 'painting' the picture.
    Takes only the convolution layer weights and wrap using the TensorFlow
    Conv2d, Relu and AveragePooling layer. VGG actually uses maxpool but
    the paper indicates that using AveragePooling yields better results.
    The last few fully connected layers are not used.
```

```
      Here is the detailed configuration of the VGG model:
          0 is conv1_1 (3, 3, 3, 64)
          1 is relu
          2 is conv1_2 (3, 3, 64, 64)
          3 is relu
          4 is maxpool
          5 is conv2_1 (3, 3, 64, 128)
          6 is relu
          7 is conv2_2 (3, 3, 128, 128)
          8 is relu
          9 is maxpool
          10 is conv3_1 (3, 3, 128, 256)
          11 is relu
          12 is conv3_2 (3, 3, 256, 256)
          13 is relu
          14 is conv3_3 (3, 3, 256, 256)
          15 is relu
          16 is conv3_4 (3, 3, 256, 256)
          17 is relu
          18 is maxpool
          19 is conv4_1 (3, 3, 256, 512)
          20 is relu
          21 is conv4_2 (3, 3, 512, 512)
          22 is relu
          23 is conv4_3 (3, 3, 512, 512)
          24 is relu
          25 is conv4_4 (3, 3, 512, 512)
          26 is relu
          27 is maxpool
          28 is conv5_1 (3, 3, 512, 512)
          29 is relu
          30 is conv5_2 (3, 3, 512, 512)
          31 is relu
          32 is conv5_3 (3, 3, 512, 512)
          33 is relu
          34 is conv5_4 (3, 3, 512, 512)
          35 is relu
          36 is maxpool
          37 is fullyconnected (7, 7, 512, 4096)
          38 is relu
          39 is fullyconnected (1, 1, 4096, 4096)
          40 is relu
          41 is fullyconnected (1, 1, 4096, 1000)
          42 is softmax
```

```python
    """

    vgg = scipy.io.loadmat(path)

    vgg_layers = vgg['layers']

    def _weights(layer, expected_layer_name):
        """
        Return the weights and bias from the VGG model for a given layer.
        """
        wb = vgg_layers[0][layer][0][0][2]
        W = wb[0][0]
        b = wb[0][1]
        layer_name = vgg_layers[0][layer][0][0][0][0]
        assert layer_name == expected_layer_name
        return W, b

        return W, b

    def _relu(conv2d_layer):
        """
        Return the RELU function wrapped over a TensorFlow layer. Expects a
        Conv2d layer input.
        """
        return tf.nn.relu(conv2d_layer)

    def _conv2d(prev_layer, layer, layer_name):
        """
        Return the Conv2D layer using the weights, biases from the VGG
        model at 'layer'.
        """
        W, b = _weights(layer, layer_name)
        W = tf.constant(W)
        b = tf.constant(np.reshape(b, (b.size)))
        return tf.nn.conv2d(prev_layer, filter=W, strides=[1, 1, 1, 1], padding='SAME') + b

    def _conv2d_relu(prev_layer, layer, layer_name):
        """
        Return the Conv2D + RELU layer using the weights, biases from the VGG
        model at 'layer'.
        """
        return _relu(_conv2d(prev_layer, layer, layer_name))

    def _avgpool(prev_layer):
        """
        Return the AveragePooling layer.
        """
        return tf.nn.avg_pool(prev_layer, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

    # Constructs the graph model.
    graph = {}
    graph['input'] = tf.Variable(np.zeros((1, IMAGE_HEIGHT, IMAGE_WIDTH, COLOR_CHANNELS)),
                             dtype='float32')
    graph['conv1_1'] = _conv2d_relu(graph['input'], 0, 'conv1_1')
    graph['conv1_2'] = _conv2d_relu(graph['conv1_1'], 2, 'conv1_2')
    graph['avgpool1'] = _avgpool(graph['conv1_2'])
    graph['conv2_1'] = _conv2d_relu(graph['avgpool1'], 5, 'conv2_1')
    graph['conv2_2'] = _conv2d_relu(graph['conv2_1'], 7, 'conv2_2')
    graph['avgpool2'] = _avgpool(graph['conv2_2'])
    graph['conv3_1'] = _conv2d_relu(graph['avgpool2'], 10, 'conv3_1')
    graph['conv3_2'] = _conv2d_relu(graph['conv3_1'], 12, 'conv3_2')
    graph['conv3_3'] = _conv2d_relu(graph['conv3_2'], 14, 'conv3_3')
    graph['conv3_4'] = _conv2d_relu(graph['conv3_3'], 16, 'conv3_4')
    graph['avgpool3'] = _avgpool(graph['conv3_4'])
    graph['conv4_1'] = _conv2d_relu(graph['avgpool3'], 19, 'conv4_1')
    graph['conv4_2'] = _conv2d_relu(graph['conv4_1'], 21, 'conv4_2')
    graph['conv4_3'] = _conv2d_relu(graph['conv4_2'], 23, 'conv4_3')
    graph['conv4_4'] = _conv2d_relu(graph['conv4_3'], 25, 'conv4_4')
```

```python
        graph['avgpool4'] = _avgpool(graph['conv4_4'])
        graph['conv5_1'] = _conv2d_relu(graph['avgpool4'], 28, 'conv5_1')
        graph['conv5_2'] = _conv2d_relu(graph['conv5_1'], 30, 'conv5_2')
        graph['conv5_3'] = _conv2d_relu(graph['conv5_2'], 32, 'conv5_3')
        graph['conv5_4'] = _conv2d_relu(graph['conv5_3'], 34, 'conv5_4')
        graph['avgpool5'] = _avgpool(graph['conv5_4'])

        return graph
```