AC 209B: Homework 6

Neural Style Transfer

Harvard University
Spring 2018
Instructors: Pavlos Protopapas and Mark Glickman

INSTRUCTIONS

- To submit your assignment follow the instructions given in canvas.
- Make sure the homework runs correctly before you submit.

Your partner's name (if you submit separately):

Introduction

We will implement the neural style transfer technique presented in "Image Style Transfer Using Convolutional Neural Networks" (Gatys et al., CVPR 2015) (http://www.cv-foundation.org/openaccess/content_cvpr_2016/papers/Gatys_Image_Style_Transfer_CVPR_2016_paper.pdf). As we have seen in the a-section, this technique combines the content and style of two given images, and generates a third one, reflecting the elements of the original images.

The purpose of this homework will be to understand and program the loss functions that mimics the contents of an image, the style, and combines them in a general loss function together with some regularizer.

We recommend you to read the lecture notes in order to implement the following exercises.

Setup

Run the following code to load the necessary libraries.

In [1]:
```
import time
import numpy as np

import tensorflow as tf
from keras import backend as K
from keras.applications import vgg16, vgg19
from keras.preprocessing.image import load_img

from scipy.misc import imsave
from scipy.optimize import fmin_l_bfgs_b

# preprocessing
from utils import preprocess_image, deprocess_image

%load_ext autoreload
%autoreload 2
```

```
/anaconda3/lib/python3.6/site-packages/h5py/__init__.py:36: FutureWarning: Conversion of the second argument of issubdtype from `flo
at` to `np.floating` is deprecated. In future, it will be treated as `np.float64 == np.dtype(float).type`.
  from ._conv import register_converters as _register_converters
Using TensorFlow backend.
```

Part A: Content loss (1 pt)

We can generate an image that combines the content of one image with the style of another with a loss function that incorporates this information. This is achieved with two terms, one that mimics the specific ativations of a certain layer for the content image, and a second term that mimics the style. The variable to optimize in the loss function will be a generated image that aims to minimize the proposed cost. Note that to optimize this function, we will perform gradient descent **on the pixel values**, rather than on the neural network weights.

We will load a trained neural network called VGG-16 proposed in 1 (https://arxiv.org/pdf/1409.1556.pdf), who secured the first and second place in the localisation and classification tracks of ImageNet Challenge in 2014, respectively. This network has been trained to discriminate over 1000 classes over more than a million images. We will use the activation values obtained for an image of interest to represent the content and styles. In order to do so, we will feed-forward the image of interest and observe it's activation values at the indicated layer.

The content loss function measures how much the feature map of the generated image differs from the feature map of the source image. We will only consider a single layer to represent the contents of an image. The authors of this technique indicated they obtained better results when doing so. We denote the feature maps for layer l with $a^{[l]} \in \mathbb{R}^{n_H^{[l]} \times n_W^{[l]} \times n_C^{[l]}}$. Parameter $n_C^{[l]}$ is the number of filters/channels in layer l, $n_H^{[l]}$ and $n_W^{[l]}$ are the height and width.

The content loss is then given by:

$$J_C^{[l]} = \left\| a^{[l](G)} - a^{[l](C)} \right\|_F^2,$$

where $a^{[l](G)}$ refers to the layer's activation values of the generated image, and $a^{[l](C)}$ to those of the content image.

Implement funtion `feature_reconstruction_loss` that computes the loss of two feature inputs. You will need to use keras backend functions (https://keras.io/backend/#backend-functions) to complete the exercise.

In [2]:
```
def feature_reconstruction_loss(base, output):
    """
    Compute the content loss for style transfer.

    Inputs:
    - output: features of the generated image, Tensor with shape [height, width, channels]
    - base: features of the content image, Tensor with shape [height, width, channels]

    Returns:
    - scalar content loss
    """
    return K.sum(K.square(output - base))
```

Test your implementation:

In [3]:
```
np.random.seed(1)
base = np.random.randn(10,10,3)
output = np.random.randn(10,10,3)
a = K.constant(base)
b = K.constant(output)
test = feature_reconstruction_loss(a, b)
print('Result:          ', K.eval(test))
print('Expected result: ', 605.62195)
```

```
Result:           605.6219
Expected result:  605.62195
```

Part B: Style loss: computing the Gram matrix (2 pts)

The style measures the similarity among filters in a set of layers. In order to compute that similarity, we will compute the Gram matrix of the activation values for the style layers, i.e., $a^{[l]}$ for some set $\mathcal{L}$. The Gram matrix is related to the empirical covariance matrix, and therefore, reflects the statistics of the activation values.

Given a feature map $a^{[l]}$ of shape $(n_H^{[l]}, n_W^{[l]}, n_C^{[l]})$, the Gram matrix has shape $(n_C^{[l]}, n_C^{[l]})$ and its elements are given by:

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

The output is a 2-D matrix which approximately measures the cross-correlation among different filters for a given layer. This in essence constitutes the style of a layer.

Implement a function that computes the Gram matrix of a given keras tensor. To receive full credit, do not use any loops. This can be accomplished efficiently if x is reshaped as a tensor of shape $(n_C^{[l]} \times n_H^{[l]} n_W^{[l]})$. You will need to use keras backend functions (https://keras.io/backend/#backend-functions) to complete the exercise.

In [4]:
```
def gram_matrix(x):
    """
    Computes the outer-product of the input tensor x.

    Input:
    - x: input tensor of shape (H, W, C)

    Returns:
    - tensor of shape (C, C) corresponding to the Gram matrix of
    the input image.
    """
    # reshape to (C=2, H=0, W=1) and flatten
    features = K.batch_flatten(K.permute_dimensions(x, (2, 0, 1)))
    return K.dot(features, K.transpose(features))
```

Test your implmentation:

In [5]:
```
np.random.seed(1)
x_np = np.random.randn(10,10,3)
x = K.constant(x_np)
test = gram_matrix(x)
print('Result:\n', K.eval(test))
print('Expected:\n', np.array([[99.75723, -9.96186, -1.4740534], [-9.96186, 86.854324, -4.141108 ], [-1.4740534, -4.141108, 82.301(
```

```
Result:
 [[99.757225  -9.961859  -1.4740529]
 [-9.961859  86.854355  -4.1411047]
 [-1.4740529 -4.1411047 82.30109  ]]
Expected:
 [[99.75723   -9.96186   -1.4740534]
 [-9.96186   86.854324  -4.141108 ]
 [-1.4740534 -4.141108  82.30106  ]]
```

Part C: Style loss: layer's loss (2 pts)

Now we can tackle the style loss. For a given layer l, the style loss is defined as follows:

$$J_S^{[l]} = \frac{1}{4(n_W^{[l]} n_H^{[l]})^2} \left\| G^{[l](S)} - G^{[l](G)} \right\|_F^2.$$

In practice we compute the style loss at a set of layers $\mathcal{L}$ rather than just a single layer l; then the total style loss is the sum of style losses at each layer:

$$J_S = \sum_{l \in \mathcal{L}} \lambda_l J_S^{[l]}$$

where λ_l corresponds to a weighting parameter. You do not need to implement the weighted sum, this is given to you and coded in the main function of this implmentation.

Implement `style_reconstruction_loss` that computes the loss for a given layer l. To receive full credit, do not use any loops. You will need to use keras backend functions (https://keras.io/backend/#backend-functions) to complete the exercise.

In [6]:
```
def style_reconstruction_loss(base, output):
    """
    Computes the style reconstruction loss. It encourages the output img
    to have same stylistic features as style image.

    Inputs:
    - base: features at given layer of the style image.
    - output: features of the same length as base of the generated image.

    Returns:
    - style_loss: scalar style loss
    """
    H, W, C = [int(x) for x in base.shape]
    gram_base = gram_matrix(base)
    gram_output = gram_matrix(output)
    factor = 1/ (4* (H*W)**2)
    loss = factor * K.sum(K.square(gram_output - gram_base))
    return loss
```

Test your implementation:

In [7]:
```
np.random.seed(1)
x = np.random.randn(10,10,3)
y = np.random.randn(10,10,3)
a = K.constant(x)
b = K.constant(y)
test = style_reconstruction_loss(a, b)
print('Result:  ', K.eval(test))
print('Expected:', 0.09799164)
```

```
Result:   0.09799156
Expected: 0.09799164
```

Part D: Total-variation regularization (2 pts)

We will also encourage smoothness in the image using a total-variation regularizer. This penalty term will reduce variation among the neighboring pixel values.

The following expression constitues the regularization penalty over all pairs that are next to each other horizontally or vertically. The expression is independent among different RGB channels.

$$J_{tv} = \sum_{c=1}^{3} \sum_{i=1}^{n_H^{[l]}-1} \sum_{j=1}^{n_W^{[l]}-1} \left((x_{i,j+1,c} - x_{i,j,c})^2 + (x_{i+1,j,c} - x_{i,j,c})^2 \right)$$

In the next cell, fill in the definition for the TV loss term. To receive full credit, your implementation should not have any loops.

Remark: in this exercice x has dimension $(1, n_H^{[l]}, n_W^{[l]}, n_C^{[l]})$, which is different from the 3D-tensors we used before.

In [8]:
```
def total_variation_loss(x):
    """
    Total variational loss. Encourages spatial smoothness
    in the output image.

    Inputs:
    - x: image with pixels, has shape 1 x H x W x C.

    Returns:
    - total variation loss, a scalar number.
    """
    I, H, W, C = [int(j) for j in x.shape]
    a = K.square(x[:, :H-1, :W-1, :] - x[:, 1:, :W-1, :])
    b = K.square(x[:, :H-1, :W-1, :] - x[:, :H-1, 1:, :])

    return K.sum(a+b)
```

Test your implementation:

In [9]:
```
np.random.seed(1)
x_np = np.random.randn(1,10,10,3)
x = K.constant(x_np)
test = total_variation_loss(x)
print('Result:  ', K.eval(test))
print('Expected:', 937.0538)
```

```
Result:   937.0538
Expected: 937.0538
```

Part E: Style transfer (2 pts)

We now put it all together and generate some images! The `style_transfer` function below combines all the losses you coded up above and optimizes for an image that minimizes the total loss. Read the code and comments to understand the procedure.

In [10]:

```
def style_transfer(base_img_path, style_img_path, output_img_path, convnet='vgg16',
        content_weight=3e-2, style_weights=(20000, 500, 12, 1, 1), tv_weight=5e-2, content_layer='block4_conv2',
        style_layers=['block1_conv1', 'block2_conv1', 'block3_conv1', 'block4_conv1', 'block5_conv1'], iterations=50):

    print('\nInitializing Neural Style model...')

    # Determine the image sizes. Fix the output size from the content image.
    print('\n\tResizing images...')
    width, height = load_img(base_img_path).size
    new_dims = (height, width)

    # Preprocess content and style images. Resizes the style image if needed.
    content_img = K.variable(preprocess_image(base_img_path, new_dims))
    style_img = K.variable(preprocess_image(style_img_path, new_dims))

    # Create an output placeholder with desired shape.
    # It will correspond to the generated image after minimizing the loss function.
    output_img = K.placeholder((1, height, width, 3))

    # Sanity check on dimensions
    print("\tSize of content image is: {}".format(K.int_shape(content_img)))
    print("\tSize of style image is: {}".format(K.int_shape(style_img)))
    print("\tSize of output image is: {}".format(K.int_shape(output_img)))

    # Combine the 3 images into a single Keras tensor, for ease of manipulation
    # The first dimension of a tensor identifies the example/input.
    input_img = K.concatenate([content_img, style_img, output_img], axis=0)

    # Initialize the vgg16 model
    print('\tLoading {} model'.format(convnet.upper()))

    if convnet == 'vgg16':
        model = vgg16.VGG16(input_tensor=input_img, weights='imagenet', include_top=False)
    else:
        model = vgg19.VGG19(input_tensor=input_img, weights='imagenet', include_top=False)

    print('\tComputing losses...')
    # Get the symbolic outputs of each "key" layer (they have unique names).
    # The dictionary outputs an evaluation when the model is fed an input.
    outputs_dict = dict([(layer.name, layer.output) for layer in model.layers])

    # Extract features from the content layer
    content_features = outputs_dict[content_layer]

    # Extract the activations of the base image and the output image
    base_image_features = content_features[0, :, :, :]  # 0 corresponds to base
    combination_features = content_features[2, :, :, :] # 2 coresponds to output

    # Calculate the feature reconstruction loss
    content_loss = content_weight * feature_reconstruction_loss(base_image_features, combination_features)

    # For each style layer compute style loss
    # The total style loss is the weighted sum of those losses
    temp_style_loss = K.variable(0.0)       # we update this variable in the loop
    weight = 1.0 / float(len(style_layers))

    for i, layer in enumerate(style_layers):
        # extract features of given layer
        style_features = outputs_dict[layer]
        # from those features, extract style and output activations
        style_image_features = style_features[1, :, :, :]   # 1 corresponds to style image
        output_style_features = style_features[2, :, :, :]  # 2 coresponds to generated image
        temp_style_loss += style_weights[i] * weight * \
                    style_reconstruction_loss(style_image_features, output_style_features)
    style_loss = temp_style_loss

    # Compute total variational loss.
    tv_loss = tv_weight * total_variation_loss(output_img)

    # Composite loss
    total_loss = content_loss + style_loss + tv_loss

    # Compute gradients of output img with respect to total_loss
    print('\tComputing gradients...')
    grads = K.gradients(total_loss, output_img)

    outputs = [total_loss] + grads
    loss_and_grads = K.function([output_img], outputs)

    # Initialize the generated image from random noise
    x = np.random.uniform(0, 255, (1, height, width, 3)) - 128.

    # Loss function that takes a vectorized input image, for the solver
    def loss(x):
        x = x.reshape((1, height, width, 3))   # reshape
        return loss_and_grads([x])[0]

    # Gradient function that takes a vectorized input image, for the solver
    def grads(x):
        x = x.reshape((1, height, width, 3))   # reshape
        return loss_and_grads([x])[1].flatten().astype('float64')

    # Fit over the total iterations
    for i in range(iterations):
        print('\n\tIteration: {}'.format(i+1))
```

```
        toc = time.time()
        x, min_val, info = fmin_l_bfgs_b(loss, x.flatten(), fprime=grads, maxfun=20)

        # save current generated image
        img = deprocess_image(x.copy(), height, width)
        fname = output_img_path + '_at_iteration_%d.png' % (i+1)
        imsave(fname, img)

        tic = time.time()

        print('\t\tImage saved as', fname)
        print('\t\tLoss: {:.2e}, Time: {} seconds'.format(float(min_val), float(tic-toc)))
```

In [ ]:
```
m1 = vgg16.VGG16(weights='imagenet', include_top=False)
m1_layers = [layer.name for layer in m1.layers]
m1_layers
```

In [ ]:
```
m2 = vgg19.VGG19(weights='imagenet', include_top=False)
m2_layers = [layer.name for layer in m2.layers]
m2_layers
```

Generate pictures

Try `style_transfer` on the three different parameter sets below. Feel free to add your own, and make sure to include the results of the style transfer in your submitted notebook. You may adjust any parameter you feel can improve your result.

- The `base_img_path` is the filename of content image.
- The `style_img_path` is the filename of style image.
- The `output_img_path` is the filename of generated image.
- The `convnet` is for the neural network weights, VGG-16 or VGG-19.
- The `content_layer` specifies which layer to use for content loss.
- The `content_weight` weights the content loss in the overall composite loss function. Increasing the value of this parameter will make the final image look more realistic (closer to the original content).
- `style_layers` specifies a list of which layers to use for the style loss.
- `style_weights` specifies a list of weights to use for each layer in style_layers (each of which will contribute a term to the overall style loss). We generally use higher weights for the earlier style layers because they describe more local/smaller scale features, which are more important to texture than features over larger receptive fields. In general, increasing these weights will make the resulting image look less like the original content and more distorted towards the appearance of the style image.
- `tv_weight` specifies the weighting of total variation regularization in the overall loss function. Increasing this value makes the resulting image look smoother and less jagged, at the cost of lower fidelity to style and content.

Submit the best created images for each three content-style pairs.

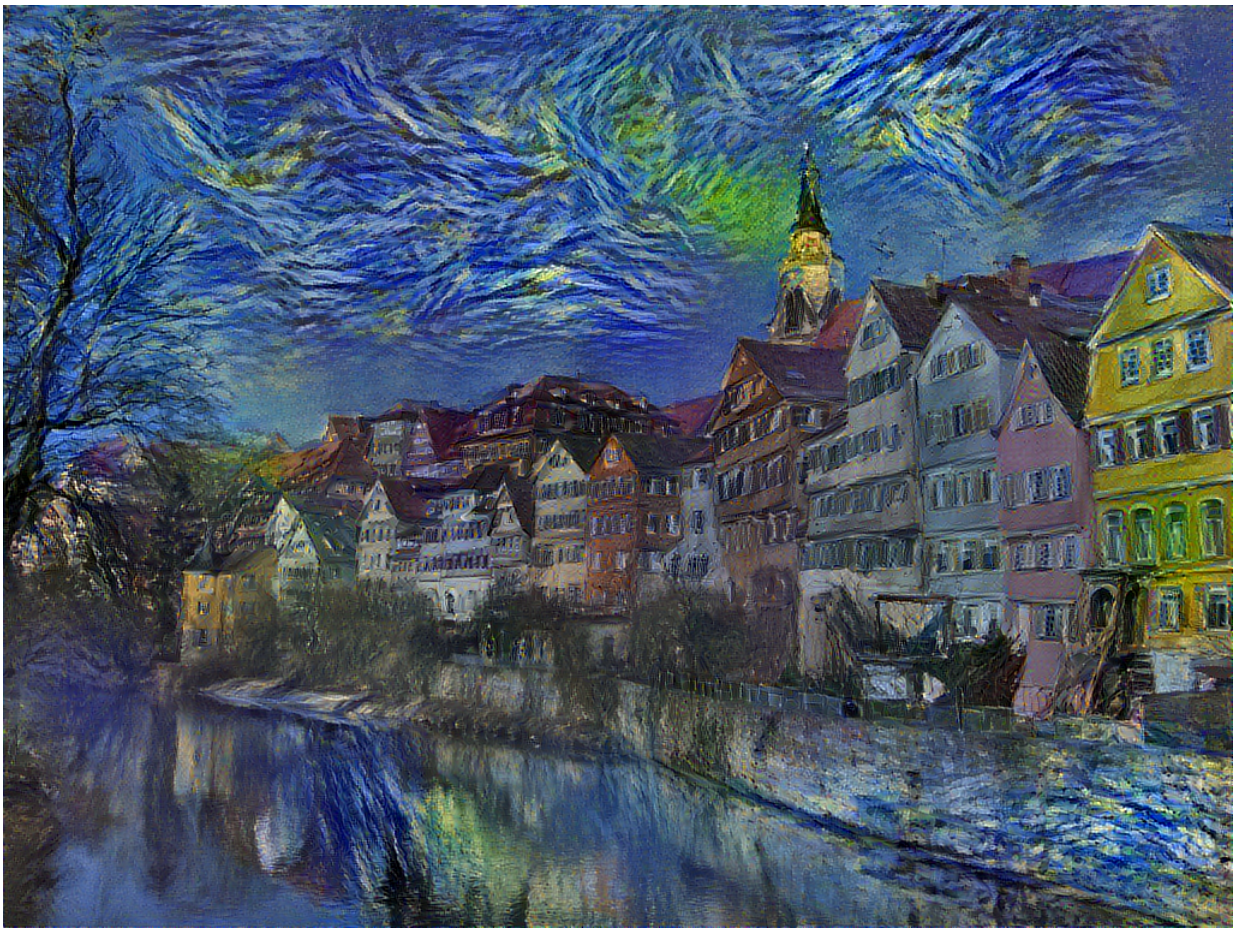
Great wave of Kanagawa + Chicago

In [ ]:
```
params = {
'base_img_path' : 'images/inputs/chicago.jpg',
'style_img_path' : 'images/inputs/great_wave_of_kanagawa.jpg',
'output_img_path' : 'images/results/wave_chicago',
'convnet' : 'vgg16',
'content_weight' : 500,
'style_weights' : (20, 20, 30, 10, 10),
'tv_weight' : 200,
'content_layer' : 'block4_conv2',
'style_layers' : ['block1_conv1',
                  'block2_conv1',
                  'block3_conv1',
                  'block4_conv1',
                  'block5_conv1'],
'iterations' : 50
}

style_transfer(**params)
```

Starry night + Tübingen

In [ ]:

```
params = {
'base_img_path' : 'images/inputs/tubingen.jpg',
'style_img_path' : 'images/inputs/starry_night.jpg',
'output_img_path' : 'images/results/starry_tubingen',
'convnet' : 'vgg16',
'content_weight' : 100,
'style_weights' : (1000, 100, 12, 1, 1),
'tv_weight' : 200,
'content_layer' : 'block4_conv2',
'style_layers' : ['block1_conv1',
                  'block2_conv1',
                  'block3_conv1',
                  'block4_conv1',
                  'block5_conv1'],
'iterations' : 50
}

style_transfer(**params)
```



Portrait of a man (El Greco) + Pavlos

In [ ]:

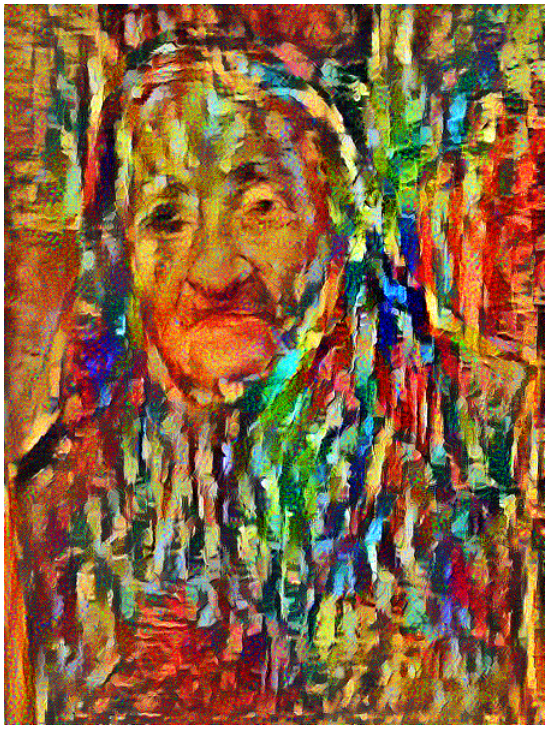
```
params = {
'base_img_path' : 'images/inputs/pavlos_protopapas.jpg',
'style_img_path' : 'images/inputs/portrait_of_a_man.jpg',
'output_img_path' : 'images/results/portrait_of_pavlos',
'convnet' : 'vgg16',
'content_weight' : 1000,
'style_weights' : (500, 100, 12, 1, 1),
'tv_weight' : 200,
'content_layer' : 'block4_conv2',
'style_layers' : ['block1_conv1',
                  'block2_conv1',
                  'block3_conv1',
                  'block4_conv1',
                  'block5_conv1'],
'iterations' : 50
}

style_transfer(**params)
```

Part F: Pavlos Grandmother (1 pt)

Use the image of Pavlos' grandmother `images/inputs/pavlos_grandmother.jpg` and an artistic style of your choice.

In [ ]:

```
params = {
'base_img_path' : 'images/inputs/resized_pavlos_grandma.jpg',
'style_img_path' : 'images/inputs/woman-with-hat-matisse.jpg',
'output_img_path' : 'images/results/pavlos_grandma_woman_with_hat',
'convnet' : 'vgg19',
'content_weight' : 1000,
'style_weights' : (500, 100, 12, 1, 1),
'tv_weight' : 200,
'content_layer' : 'block4_conv2',
'style_layers' : ['block1_conv2',
                  'block2_conv2',
                  'block3_conv4',
                  'block4_conv4',
                  'block5_conv4'],
'iterations' : 50
}

style_transfer(**params)
```



Acknowledgments

- The implementation uses code from Francois Chollet's neural style transfer.
- The implementation uses code from Kevin Zakka's neural style transfer, under MIT license.
- The hierarchy borrows from Giuseppe Bonaccorso's gist, under MIT license.
- Some of the documentation guidelines and function documentation have been borrowed from Stanford's cs231n course.

In [ ]:

```

```