**Practical Assignment**

**Objective: - Neural Style Transfer**

The idea of this project is to make art by using one image and then transferring the style of that image to the target image.

**Dataset Link: -**

Use anyone you like.

**Task: -** Create a Web Application using FASTAPI. Use the end user should be able to upload an 2 images and get the output photo .

**Assignment Submission: -** Only submit the GitHub Link. Create a proper Readme documentation.

# Deep Learning & Art: Neural Style Transfer

In this assignment, you will learn about Neural Style Transfer. This algorithm was created by [Gatys et al. (2015).](https://arxiv.org/abs/1508.06576)

**In this assignment, you will:**

* Implement the neural style transfer algorithm
* Generate novel artistic images using your algorithm

Most of the algorithms you've studied optimize a cost function to get a set of parameter values. In Neural Style Transfer, you'll optimize a cost function to get pixel values!

## Updates

#### If you were working on the notebook before this update...

* The current notebook is version "3a".
* You can find your original work saved in the notebook with the previous version name ("v2")
* To view the file directory, go to the menu "File->Open", and this will open a new tab that shows the file directory.

List of updates

* Use pprint.PrettyPrinter to format printing of the vgg model.
* computing content cost: clarified and reformatted instructions, fixed broken links, added additional hints for unrolling.
* style matrix: clarify two uses of variable "G" by using different notation for gram matrix.
* style cost: use distinct notation for gram matrix, added additional hints.
* Grammar and wording updates for clarity.
* model\_nn: added hints.

**import** os

**import** sys

**import** scipy.io

**import** scipy.misc

**import** matplotlib.pyplot **as** plt

**from** matplotlib.pyplot **import** imshow

**from** PIL **import** Image

**from** nst\_utils **import** **\***

**import** numpy **as** np

**import** tensorflow **as** tf

**import** pprint

**%matplotlib** inline

## 1 - Problem Statement

Neural Style Transfer (NST) is one of the most fun techniques in deep learning. As seen below, it merges two images, namely: a **"content" image (C) and a "style" image (S), to create a "generated" image (G**).

The generated image G combines the "content" of the image C with the "style" of image S.

In this example, you are going to generate an image of the Louvre museum in Paris (content image C), mixed with a painting by Claude Monet, a leader of the impressionist movement (style image S).

## 2 - Transfer Learning

Neural Style Transfer (NST) uses a previously trained convolutional network, and builds on top of that. The idea of using a network trained on a different task and applying it to a new task is called transfer learning.

Following the [original NST paper](https://arxiv.org/abs/1508.06576), we will use the VGG network. Specifically, we'll use VGG-19, a 19-layer version of the VGG network. This model has already been trained on the very large ImageNet database, and thus has learned to recognize a variety of low level features (at the shallower layers) and high level features (at the deeper layers).

Run the following code to load parameters from the VGG model. This may take a few seconds.

pp **=** pprint**.**PrettyPrinter(indent**=**4)

model **=** load\_vgg\_model("pretrained-model/imagenet-vgg-verydeep-19.mat")

pp**.**pprint(model)

#### Assign input image to the model's input layer

To run an image through this network, you just have to feed the image to the model. In TensorFlow, you can do so using the [tf.assign](https://www.tensorflow.org/api_docs/python/tf/assign) function. In particular, you will use the assign function like this:

model["input"]**.**assign(image)

This assigns the image as an input to the model.

#### Activate a layer

After this, if you want to access the activations of a particular layer, say layer 4\_2 when the network is run on this image, you would run a TensorFlow session on the correct tensor conv4\_2, as follows:

sess**.**run(model["conv4\_2"])

## 3 - Neural Style Transfer (NST)

We will build the Neural Style Transfer (NST) algorithm in three steps:

* Build the content cost function 𝐽𝑐𝑜𝑛𝑡𝑒𝑛𝑡(𝐶,𝐺)
* Build the style cost function 𝐽𝑠𝑡𝑦𝑙𝑒(𝑆,𝐺)
* Put it together to get (𝐺)=𝛼𝐽𝑐𝑜𝑛𝑡𝑒𝑛𝑡(𝐶,𝐺)+𝛽𝐽𝑠𝑡𝑦𝑙𝑒(𝑆,𝐺).

### 3.1 - Computing the content cost

In our running example, the content image C will be the picture of the Louvre Museum in Paris. Run the code below to see a picture of the Louvre.

content\_image **=** scipy**.**misc**.**imread("images/louvre.jpg")

imshow(content\_image);

#### Shallower versus deeper layers

* The shallower layers of a ConvNet tend to detect lower-level features such as edges and simple textures.
* The deeper layers tend to detect higher-level features such as more complex textures as well as object classes.

#### Choose a "middle" activation layer [𝑙]

We would like the "generated" image G to have similar content as the input image C. Suppose you have chosen some layer's activations to represent the content of an image.

* In practice, you'll get the most visually pleasing results if you choose a layer in the **middle** of the network--neither too shallow nor too deep.
* (After you have finished this exercise, feel free to come back and experiment with using different layers, to see how the results vary.)

#### Forward propagate image "C"

* Set the image C as the input to the pretrained VGG network, and run forward propagation.
* Let (𝐶) be the hidden layer activations in the layer you had chosen. (In lecture, we had written this as [𝑙](𝐶), but here we'll drop the superscript [𝑙] to simplify the notation.) This will be an 𝑛𝐻×𝑛𝑊×𝑛𝐶 tensor.

#### Forward propagate image "G"

* Repeat this process with the image G: Set G as the input, and run forward progation.
* Let (𝐺) be the corresponding hidden layer activation.

#### Content Cost Function 𝐽𝑐𝑜𝑛𝑡𝑒𝑛(𝐶,𝐺)

We will define the content cost function as:

(1)𝐽𝑐𝑜𝑛𝑡𝑒𝑛(𝐶,𝐺)=14×𝑛𝐻×𝑛𝑊×𝑛𝐶∑all entries(𝑎(𝐶)−𝑎(𝐺))2

* Here, 𝑛𝐻,𝑊 and 𝑛𝐶 are the height, width and number of channels of the hidden layer you have chosen, and appear in a normalization term in the cost.
* For clarity, note that (𝐶) and 𝑎(𝐺) are the 3D volumes corresponding to a hidden layer's activations.
* In order to compute the cost 𝐽𝑐𝑜𝑛𝑡𝑒𝑛(𝐶,𝐺), it might also be convenient to unroll these 3D volumes into a 2D matrix, as shown below.
* Technically this unrolling step isn't needed to compute 𝐽𝑐𝑜𝑛𝑡𝑒𝑛𝑡, but it will be good practice for when you do need to carry out a similar operation later for computing the style cost 𝐽𝑠𝑡𝑦𝑙𝑒.

**Exercise:** Compute the "content cost" using TensorFlow.

**Instructions**: The 3 steps to implement this function are:

1. Retrieve dimensions from a\_G:
   * To retrieve dimensions from a tensor X, use: X.get\_shape().as\_list()
2. Unroll a\_C and a\_G as explained in the picture above
   * You'll likey want to use these functions: [tf.transpose](https://www.tensorflow.org/versions/r1.15/api_docs/python/tf/transpose) and [tf.reshape](https://www.tensorflow.org/versions/r1.15/api_docs/python/tf/reshape).
3. Compute the content cost:
   * You'll likely want to use these functions: [tf.reduce\_sum](https://www.tensorflow.org/api_docs/python/tf/reduce_sum), [tf.square](https://www.tensorflow.org/api_docs/python/tf/square) and [tf.subtract](https://www.tensorflow.org/api_docs/python/tf/subtract).

#### Additional Hints for "Unrolling"

* To unroll the tensor, we want the shape to change from (𝑚,𝐻,𝑛𝑊,𝑛𝐶) to (𝑚,𝑛𝐻×𝑛𝑊,𝑛𝐶).
* tf.reshape(tensor, shape) takes a list of integers that represent the desired output shape.
* For the shape parameter, a -1 tells the function to choose the correct dimension size so that the output tensor still contains all the values of the original tensor.
* So tf.reshape(a\_C, shape=[m, n\_H \* n\_W, n\_C]) gives the same result as tf.reshape(a\_C, shape=[m, -1, n\_C]).
* If you prefer to re-order the dimensions, you can use tf.transpose(tensor, perm), where perm is a list of integers containing the original index of the dimensions.
* For example, tf.transpose(a\_C, perm=[0,3,1,2]) changes the dimensions from (𝑚,𝑛𝐻,𝑛𝑊,𝑛𝐶) to (𝑚,𝑛𝐶,𝑛𝐻,𝑛𝑊).
* There is more than one way to unroll the tensors.
* Notice that it's not necessary to use tf.transpose to 'unroll' the tensors in this case but this is a useful function to practice and understand for other situations that you'll encounter

*# GRADED FUNCTION: compute\_content\_cost*

**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.

"""

*### START CODE HERE ###*

*# 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 **=** **None**

a\_G\_unrolled **=** **None**

*# compute the cost with tensorflow (≈1 line)*

J\_content **=** 1 **/** (4 **\*** n\_H **\*** n\_W **\*** n\_C) **\*** tf**.**reduce\_sum((a\_C **-** a\_G)**\*\***2)

*### END CODE HERE ###*

**return** J\_content

tf**.**reset\_default\_graph()

**with** tf**.**Session() **as** test:

tf**.**set\_random\_seed(1)

a\_C **=** tf**.**random\_normal([1, 4, 4, 3], mean**=**1, stddev**=**4)

a\_G **=** tf**.**random\_normal([1, 4, 4, 3], mean**=**1, stddev**=**4)

J\_content **=** compute\_content\_cost(a\_C, a\_G)

print("J\_content = " **+** str(J\_content**.**eval()))

### 3.2 - Computing the style cost

For our running example, we will use the following style image:

style\_image **=** scipy**.**misc**.**imread("images/monet\_800600.jpg")

imshow(style\_image);

### 3.2.1 - Style matrix

#### Gram matrix

* The style matrix is also called a "Gram matrix."
* In linear algebra, the Gram matrix G of a set of vectors (𝑣1,…,𝑣𝑛) is the matrix of dot products, whose entries are 𝐺𝑖𝑗=𝑣𝑖𝑇𝑣𝑗=𝑛𝑝.𝑑𝑜𝑡(𝑣𝑖,𝑣𝑗).
* In other words, 𝐺𝑖𝑗 compares how similar 𝑣𝑖 is to 𝑣𝑗: If they are highly similar, you would expect them to have a large dot product, and thus for 𝐺𝑖𝑗 to be large.

#### Two meanings of the variable 𝐺

* Note that there is an unfortunate collision in the variable names used here. We are following common terminology used in the literature.
* 𝐺 is used to denote the Style matrix (or Gram matrix)
* 𝐺 also denotes the generated image.
* For this assignment, we will use 𝐺𝑔𝑟𝑎𝑚 to refer to the Gram matrix, and 𝐺 to denote the generated image.

#### Compute 𝐺𝑔𝑟𝑎𝑚

In Neural Style Transfer (NST), you can compute the Style matrix by multiplying the "unrolled" filter matrix with its transpose:

#### (𝑔𝑟𝑎𝑚)𝑖,𝑗: correlation

The result is a matrix of dimension (𝑛𝐶,𝐶) where 𝑛𝐶 is the number of filters (channels). The value (𝑔𝑟𝑎𝑚)𝑖,𝑗 measures how similar the activations of filter 𝑖 are to the activations of filter 𝑗.

#### (𝑔𝑟𝑎𝑚),𝑖,𝑖: prevalence of patterns or textures

* The diagonal elements (𝑔𝑟𝑎𝑚)𝑖𝑖 measure how "active" a filter 𝑖 is.
* For example, suppose filter 𝑖 is detecting vertical textures in the image. Then (𝑔𝑟𝑎𝑚)𝑖𝑖 measures how common vertical textures are in the image as a whole.
* If (𝑔𝑟𝑎𝑚)𝑖𝑖 is large, this means that the image has a lot of vertical texture.

By capturing the prevalence of different types of features ((𝑔𝑟𝑎𝑚)𝑖𝑖), as well as how much different features occur together (𝐺(𝑔𝑟𝑎𝑚)𝑖𝑗), the Style matrix 𝐺𝑔𝑟𝑎𝑚 measures the style of an image.

**Exercise**:

* Using TensorFlow, implement a function that computes the Gram matrix of a matrix A.
* The formula is: The gram matrix of A is 𝐺𝐴=𝐴𝐴𝑇.
* You may use these functions: [matmul](https://www.tensorflow.org/api_docs/python/tf/matmul) and [transpose](https://www.tensorflow.org/api_docs/python/tf/transpose)

*# GRADED FUNCTION: gram\_matrix*

**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)

"""

*### START CODE HERE ### (≈1 line)*

GA **=** tf**.**matmul(A, A, transpose\_b**=True**)

*### END CODE HERE ###*

**return** GA

tf**.**reset\_default\_graph()

**with** tf**.**Session() **as** test:

tf**.**set\_random\_seed(1)

A **=** tf**.**random\_normal([3, 2**\***1], mean**=**1, stddev**=**4)

GA **=** gram\_matrix(A)

print("GA = \n" **+** str(GA**.**eval()))

### 3.2.2 - Style cost

*# GRADED FUNCTION: compute\_layer\_style\_cost*

**def** compute\_layer\_style\_cost(a\_S, a\_G):

"""

Arguments:

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)

"""

*### START CODE HERE ###*

*# 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\_C, n\_H\*n\_W) (≈2 lines)*

a\_S **=** tf**.**transpose(tf**.**reshape(a\_S, shape**=**[**-**1, n\_C]))

a\_G **=** tf**.**transpose(tf**.**reshape(a\_G, shape**=**[**-**1, 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 **/** (2 **\*** n\_C **\*** n\_H **\*** n\_W) **\*\*** 2 **\*** tf**.**reduce\_sum((GS **-** GG) **\*\*** 2)

*### END CODE HERE ###*

**return** J\_style\_layer

tf**.**reset\_default\_graph()

**with** tf**.**Session() **as** test:

tf**.**set\_random\_seed(1)

a\_S **=** tf**.**random\_normal([1, 4, 4, 3], mean**=**1, stddev**=**4)

a\_G **=** tf**.**random\_normal([1, 4, 4, 3], mean**=**1, stddev**=**4)

J\_style\_layer **=** compute\_layer\_style\_cost(a\_S, a\_G)

print("J\_style\_layer = " **+** str(J\_style\_layer**.**eval()))

### 3.2.3 Style Weights

STYLE\_LAYERS **=** [

('conv1\_1', 0.2),

('conv2\_1', 0.2),

('conv3\_1', 0.2),

('conv4\_1', 0.2),

('conv5\_1', 0.2)]

### Exercise: compute style cost

**def** compute\_style\_cost(model, STYLE\_LAYERS):

"""

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:

J\_style -- tensor representing a scalar value, style cost defined above by equation (2)

"""

*# initialize the overall style cost*

J\_style **=** 0

**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

### 3.3 - Defining the total cost to optimize

*# GRADED FUNCTION: total\_cost*

**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.

"""

*### START CODE HERE ### (≈1 line)*

J **=** alpha **\*** J\_content **+** beta **\*** J\_style

*### END CODE HERE ###*

**return** J

tf**.**reset\_default\_graph()

**with** tf**.**Session() **as** test:

np**.**random**.**seed(3)

J\_content **=** np**.**random**.**randn()

J\_style **=** np**.**random**.**randn()

J **=** total\_cost(J\_content, J\_style)

print("J = " **+** str(J))

## 4 - Solving the optimization problem

#### Start the interactive session.

*# Reset the graph*

tf**.**reset\_default\_graph()

*# Start interactive session*

sess **=** tf**.**InteractiveSession()

#### Content image

content\_image **=** scipy**.**misc**.**imread("images/louvre\_small.jpg")

content\_image **=** reshape\_and\_normalize\_image(content\_image)

#### Style image

style\_image **=** scipy**.**misc**.**imread("images/monet.jpg")

style\_image **=** reshape\_and\_normalize\_image(style\_image)

generated\_image **=** generate\_noise\_image(content\_image)

imshow(generated\_image[0]);

#### Load pre-trained VGG19 model

model **=** load\_vgg\_model("pretrained-model/imagenet-vgg-verydeep-19.mat")

#### Content Cost

*# Assign the content image to be the input of the VGG model.*

sess**.**run(model['input']**.**assign(content\_image))

*# Select the output tensor of layer conv4\_2*

out **=** model['conv4\_2']

*# Set a\_C to be the hidden layer activation from the layer we have selected*

a\_C **=** sess**.**run(out)

*# Set a\_G to be the hidden layer activation from same layer. Here, a\_G references model['conv4\_2']*

*# 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 the content cost*

J\_content **=** compute\_content\_cost(a\_C, a\_G)

#### Style cost

*# Assign the input of the model to be the "style" image*

sess**.**run(model['input']**.**assign(style\_image))

*# Compute the style cost*

J\_style **=** compute\_style\_cost(model, STYLE\_LAYERS)

### Exercise: total cost

*### START CODE HERE ### (1 line)*

J **=** total\_cost(J\_content, J\_style, alpha**=**10, beta**=**40)

*### END CODE HERE ###*

### Optimizer

*# define optimizer (1 line)*

optimizer **=** tf**.**train**.**AdamOptimizer(2.0)

*# define train\_step (1 line)*

train\_step **=** optimizer**.**minimize(J)

### Exercise: implement the model

**def** model\_nn(sess, input\_image, num\_iterations **=** 200):

*# Initialize global variables (you need to run the session on the initializer)*

*### START CODE HERE ### (1 line)*

sess**.**run(tf**.**global\_variables\_initializer())

*### END CODE HERE ###*

*# Run the noisy input image (initial generated image) through the model. Use assign().*

*### START CODE HERE ### (1 line)*

sess**.**run(model['input']**.**assign(input\_image))

*### END CODE HERE ###*

**for** i **in** range(num\_iterations):

*# Run the session on the train\_step to minimize the total cost*

*### START CODE HERE ### (1 line)*

sess**.**run(train\_step)

*### END CODE HERE ###*

*# Compute the generated image by running the session on the current model['input']*

*### START CODE HERE ### (1 line)*

generated\_image **=** sess**.**run(model['input'])

*### END CODE HERE ###*

*# Print every 20 iteration.*

**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))

*# save current generated image in the "/output" directory*

save\_image("output/" **+** str(i) **+** ".png", generated\_image)

*# save last generated image*

save\_image('output/generated\_image.jpg', generated\_image)

**return** generated\_image

model\_nn(sess, generated\_image, 20)