Neural Art Generation through Artificial Intelligence

By Pulkit Mehta

- This Guide shows step by step implementation of Neural Art Generation through Deep Learning.

You can find the main repository at [https://github.com/pulkitmehta/NeuralArt (https://github.com/pulkitmehta/NeuralArt)](https://github.com/pulkitmehta/NeuralArt)

Lets's Import Necessary Libraries

In [1]:
```
from matplotlib.pyplot import imshow
import matplotlib.pyplot as plt
import cv2
import numpy as np
import tensorflow as tf
import scipy.io
```

Model

```
I will be using imagenet-vgg-verydeep-19.mat downloaded from matcovnet. As this model is highly trained over
large imagenet dataset
Note: Github does not allow upload of files more than 100MB so make sure you download the model(imagenet-vgg
-verydeep-19.mat)
from the link mentioned below and put the file in model subdirectory.
```

[http://www.vlfeat.org/matconvnet/pretrained/ (http://www.vlfeat.org/matconvnet/pretrained/)](http://www.vlfeat.org/matconvnet/pretrained/)

```
This will be a MATLAB file containing weights and bias of respective layers.
scipy gives functionality to open matlab files in python.
Since we will be using TensorFlow so we will have to define TF model architecture ourselves.
```

In [2]:
```
vgg=scipy.io.loadmat("model/imagenet-vgg-verydeep-19.mat")
```

It is a Python Dictionary. Let's look at its keys.

In [3]:
```
vgg.keys()
```

Out[3]:
```
dict_keys(['__header__', '__version__', '__globals__', 'layers', 'meta'])
```

Key 'meta' contains target info and 'layers' contains actual weights and bias which are of our main concern

In [4]:
```python
print(vgg['__header__'])
print(vgg['__version__'])
print(vgg['__globals__'])
```

```
b'MATLAB 5.0 MAT-file, Platform: GLNXA64, Created on: Fri Sep 30 08:35:35 2016'
1.0
[]
```

Let's check shapes of Weights and Bias

In [5]:
```python
vgg['layers'][0][0][0][0][2][0][0].shape,vgg['layers'][0][0][0][0][2][0][1].shape
```

Out[5]:
```
((3, 3, 3, 64), (64, 1))
```

```
So dimentions of weights = 3x3x3x64
and dimentions of bias = 64x1
```

Now let us print all the layers' names

In [6]:
```python
print("LNo. | LName")
for i in range(43):
    print(i,"  |",vgg['layers'][0][i][0][0][0][0])
```

```
LNo. | LName
0   | conv1_1
1   | relu1_1
2   | conv1_2
3   | relu1_2
4   | pool1
5   | conv2_1
6   | relu2_1
7   | conv2_2
8   | relu2_2
9   | pool2
10   | conv3_1
11   | relu3_1
12   | conv3_2
13   | relu3_2
14   | conv3_3
15   | relu3_3
16   | conv3_4
17   | relu3_4
18   | pool3
19   | conv4_1
20   | relu4_1
21   | conv4_2
22   | relu4_2
23   | conv4_3
24   | relu4_3
25   | conv4_4
26   | relu4_4
27   | pool4
28   | conv5_1
29   | relu5_1
30   | conv5_2
31   | relu5_2
32   | conv5_3
33   | relu5_3
34   | conv5_4
35   | relu5_4
36   | pool5
37   | fc6
38   | relu6
39   | fc7
40   | relu7
41   | fc8
42   | prob
```

Let us start building model architecture

Here is a function for the same

```
Sub Functions:

weights_ function will extract weights from model's layer given a layer number.
add_relu function will apply ReLu activation function on previous layer's outputs.
add_conv_layer function will add a Convolution Layer on previous layer's outputs given layer number.
apply_relu_on_conv funtion will stack above both funtions as a single layer for better interpretation.
avg_pool function will apply average pooling on previous layer's output.


Next: Now we will make TensorFlow graph architecture in the form of dictionary
where keys will represent layer names.
Note: I have used same layer name conventions as if vgg

There is an input layer to which we will assign our image.

say convA_B represents Convolution Layer of Bth layer of Ath layer stack.

I did not include last Fully connected or softmax layers as they are of no use for us.
```

In [7]:

```python
def matlab_to_tf_model(model, input_dims=(300,400,3)):
    H=input_dims[0]
    W=input_dims[1]
    C=input_dims[2]
    layers=model['layers']

    '''
    weights_ function will extract weights from model's layer given a layer number.
    add_relu function will apply ReLu activation function on previous layer's outputs.
    add_conv_layer function will add a Convolution Layer on previous layer's outputs given layer number.
    apply_relu_on_conv funtion will stack above both funtions as a single layer for better interpretatio
n.
    avg_pool function will apply average pooling on previous layer's output.
    '''


    def weights_(layer_no, exp_layer_name):
        weights=layers[0][layer_no][0][0][2][0][0]
        bias=layers[0][layer_no][0][0][2][0][1]
        layer_name = layers[0][layer_no][0][0][0][0]
        return weights, bias

    def add_relu(convlayer):
        return tf.nn.relu(convlayer)


    def add_conv_layer(layer_p, layer_no, layer_name):
        W , B= weights_(layer_no, layer_name)
        W= tf.constant(W)
        B= tf.constant(np.reshape(B,(B.size)))
        return tf.nn.conv2d(layer_p, filter=W, strides=[1,1,1,1],
                            padding='SAME') + B


    def apply_relu_on_conv(layer_p, layer_no, layer_name):
        return add_relu(add_conv_layer(layer_p, layer_no, layer_name))


    def avg_pool(layer_p):
        return tf.nn.avg_pool(layer_p, ksize=[1,2,2,1],
                              strides=[1,2,2,1], padding='SAME')



    '''
    Next: Now we will make TensorFlow graph architecture in the form of dictionary
    where keys will represent layer names.
    Note: I have used same layer name conventions as if vgg

    There is an input layer to which we will assign our image.

    say convA_B represents Convolution Layer of Bth layer of Ath layer stack.
```

```
    I did not include last Fully connected or softmax layers as they are of no use for us.
    '''

    g=dict()

    g['input'] = tf.Variable(np.zeros(shape=(1,H,W,C)), dtype='float32')
    g['conv1_1'] = apply_relu_on_conv(g['input'],0,'conv1_1')
    g['conv1_2'] = apply_relu_on_conv(g['conv1_1'],2,'conv1_2')
    g['avgpool1'] = avg_pool(g['conv1_2'])

    g['conv2_1']  = apply_relu_on_conv(g['avgpool1'], 5, 'conv2_1')
    g['conv2_2']  = apply_relu_on_conv(g['conv2_1'], 7, 'conv2_2')
    g['avgpool2'] = avg_pool(g['conv2_2'])

    g['conv3_1']  = apply_relu_on_conv(g['avgpool2'], 10, 'conv3_1')
    g['conv3_2']  = apply_relu_on_conv(g['conv3_1'], 12, 'conv3_2')
    g['conv3_3']  = apply_relu_on_conv(g['conv3_2'], 14, 'conv3_3')
    g['conv3_4']  = apply_relu_on_conv(g['conv3_3'], 16, 'conv3_4')
    g['avgpool3'] = avg_pool(g['conv3_4'])

    g['conv4_1']  = apply_relu_on_conv(g['avgpool3'], 19, 'conv4_1')
    g['conv4_2']  = apply_relu_on_conv(g['conv4_1'], 21, 'conv4_2')
    g['conv4_3']  = apply_relu_on_conv(g['conv4_2'], 23, 'conv4_3')
    g['conv4_4']  = apply_relu_on_conv(g['conv4_3'], 25, 'conv4_4')
    g['avgpool4'] = avg_pool(g['conv4_4'])

    g['conv5_1']  = apply_relu_on_conv(g['avgpool4'], 28, 'conv5_1')
    g['conv5_2']  = apply_relu_on_conv(g['conv5_1'], 30, 'conv5_2')
    g['conv5_3']  = apply_relu_on_conv(g['conv5_2'], 32, 'conv5_3')
    g['conv5_4']  = apply_relu_on_conv(g['conv5_3'], 34, 'conv5_4')
    g['avgpool5'] = avg_pool(g['conv5_4'])

    return g
```

In [8]:
```
model=matlab_to_tf_model(vgg)
```

```
WARNING:tensorflow:From D:\Anaconda\lib\site-packages\tensorflow\python\framework\op_def_library.py:263
: colocate_with (from tensorflow.python.framework.ops) is deprecated and will be removed in a future ve
rsion.
Instructions for updating:
Colocations handled automatically by placer.
```

So here we have our model with sequence of tensors with respective shapes.

In [9]:
```
for key in model:
    print(key,'|',model[key])
```

```
input | <tf.Variable 'Variable:0' shape=(1, 300, 400, 3) dtype=float32_ref>
conv1_1 | Tensor("Relu:0", shape=(1, 300, 400, 64), dtype=float32)
conv1_2 | Tensor("Relu_1:0", shape=(1, 300, 400, 64), dtype=float32)
avgpool1 | Tensor("AvgPool:0", shape=(1, 150, 200, 64), dtype=float32)
conv2_1 | Tensor("Relu_2:0", shape=(1, 150, 200, 128), dtype=float32)
conv2_2 | Tensor("Relu_3:0", shape=(1, 150, 200, 128), dtype=float32)
avgpool2 | Tensor("AvgPool_1:0", shape=(1, 75, 100, 128), dtype=float32)
conv3_1 | Tensor("Relu_4:0", shape=(1, 75, 100, 256), dtype=float32)
conv3_2 | Tensor("Relu_5:0", shape=(1, 75, 100, 256), dtype=float32)
conv3_3 | Tensor("Relu_6:0", shape=(1, 75, 100, 256), dtype=float32)
conv3_4 | Tensor("Relu_7:0", shape=(1, 75, 100, 256), dtype=float32)
avgpool3 | Tensor("AvgPool_2:0", shape=(1, 38, 50, 256), dtype=float32)
conv4_1 | Tensor("Relu_8:0", shape=(1, 38, 50, 512), dtype=float32)
conv4_2 | Tensor("Relu_9:0", shape=(1, 38, 50, 512), dtype=float32)
conv4_3 | Tensor("Relu_10:0", shape=(1, 38, 50, 512), dtype=float32)
conv4_4 | Tensor("Relu_11:0", shape=(1, 38, 50, 512), dtype=float32)
avgpool4 | Tensor("AvgPool_3:0", shape=(1, 19, 25, 512), dtype=float32)
conv5_1 | Tensor("Relu_12:0", shape=(1, 19, 25, 512), dtype=float32)
conv5_2 | Tensor("Relu_13:0", shape=(1, 19, 25, 512), dtype=float32)
conv5_3 | Tensor("Relu_14:0", shape=(1, 19, 25, 512), dtype=float32)
conv5_4 | Tensor("Relu_15:0", shape=(1, 19, 25, 512), dtype=float32)
avgpool5 | Tensor("AvgPool_4:0", shape=(1, 10, 13, 512), dtype=float32)
```

Images

We will need to have 2 Images:

- CONTENT Image: This image will be the one which we want to style. We will call it C
- STYLING Image: This image's style will be applied. We will call it S

Then we will generate noise image (G) which will be coorelated to C to drive the optimisation towards C Our objective will be to transform G near to C.

```
Necessary Image Processing for vgg model
```

```
Notice that we will use matplotlib to display images in notebook which uses RGB scheme but Most Image Viewer
s use BGR scheme
Hence cv2.cvtColor(img, cv2.COLOR_BGR2RGB) or cv2.cvtColor(img, cv2.COLOR_RGB2BGR) would be frequently used
to reverse channels.
```

In [3]:

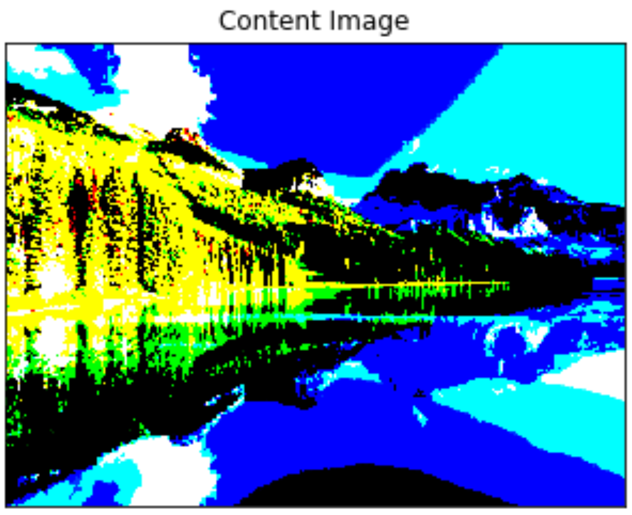
```
def load_preprocess(path):
    img=cv2.imread(path)
    origdims=img.shape[:2]
    img=cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    img=cv2.resize(img,(400,300))
    img=img.reshape(1,300,400,3)
    img=img-(np.array([123.68, 116.779, 103.939]).reshape((1,1,1,3)))
    return img,origdims
def postprocess(img):
    img=img+(np.array([123.68, 116.779, 103.939]).reshape((1,1,1,3)))
    img = np.clip(img[0], 0, 255).astype('uint8')


    return img
def generate_noisy(C, ratio=0.6):
    noise_overlay=np.random.uniform(-20, 20,
                                    (1,300,400,3)).astype('float32')
    img= (noise_overlay*ratio)+(C*(1-ratio))
    return img
```

In [4]:

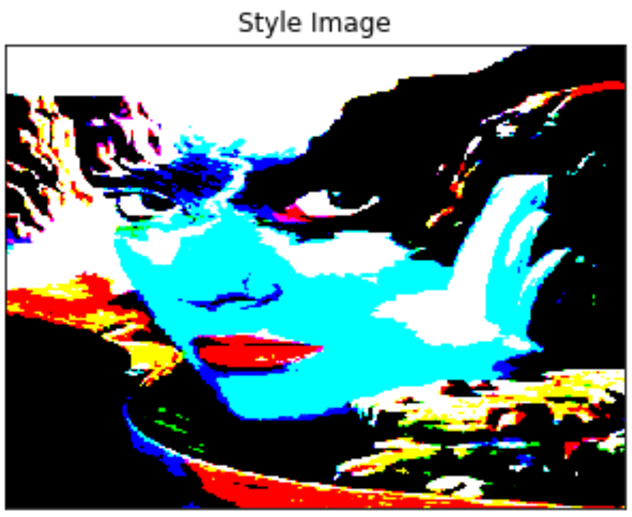
```
C, origdims =load_preprocess("./images/C.png")
S, tardims =load_preprocess("./images/S.jpg")
G=generate_noisy(C,0.7)
```

In [5]:
```
plt.title("Content Image")
imshow(C[0])
plt.xticks(())
plt.yticks(())
plt.show()
plt.title("Style Image")
imshow(S[0])
plt.xticks(())
plt.yticks(())
plt.show()
plt.title("Noisy Image")
imshow(G[0])
plt.show()
```

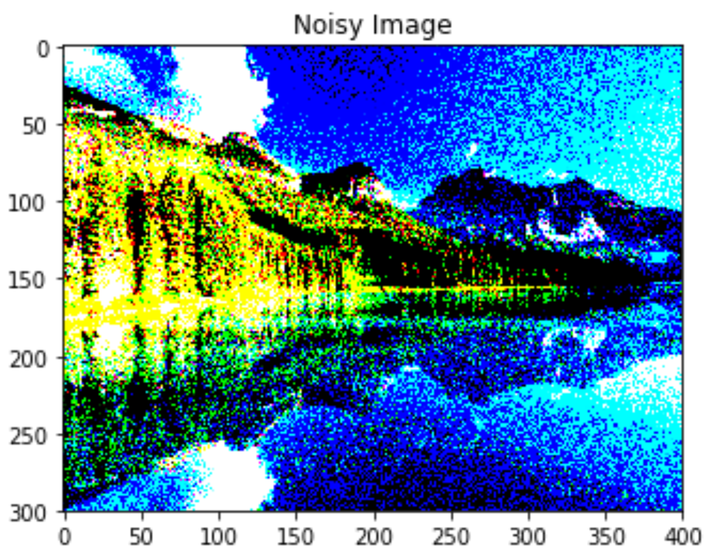
```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for inte
gers).
```




```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for inte
gers).
```




```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for inte
gers).
```



Algorithm

- We will Choose any hidden layer as our output one. It is advisable to choose somewhere in middle of depth of network.

Loss Functions

Here we need to commpute separate Losses for layer activations of C and S where G stands as our target activations.

- $Loss_{content}(C, G)$
- $Loss_{style}(S, G)$
- $Loss(G) = \alpha Loss_{content}(C, G) + \beta Loss_{style}(S, G)$.

Content Loss:

The content Loss is defined as :

$$Loss_{content}(C, G) = \frac{1}{4 \times H \times W \times C} \sum_{\text{all entries}} (a^{(C)} - a^{(G)})^2$$

- Here, H, W and C are the height, width and number of channels of the hidden layer we have chosen, and appear in a normalization term in the cost.
- Note that $a^{(C)}$ and $a^{(G)}$ are corresponding layer's activations.

In [28]:
```
def C_cost(activations_C, activations_G):

    n,H,W,C= activations_G.get_shape().as_list()
    cost= (1/(4*H*W*C))*tf.reduce_sum((activations_C-activations_G)**2)

    return cost
```

- Before going to Style Loss, Let us find a Style matrix which contains coorelations of all the filters of chosen layers.

Defined by:

$$\mathbf{SM} = \mathbf{A}_{2d}\mathbf{A}_{2d}^T$$

- where A is activations of respective layer.

In [29]:
```
def style_matrix(M):

    SM=tf.matmul(M,M, transpose_b=True)

    return SM
```

Style Loss over particular Layer is defined by:

$$Loss_{style}^{[layer]}(S, G) = \frac{1}{4 \times C^2 \times (H \times W)^2} \sum_{i=1}^{C} \sum_{j=1}^{C} (G^{(S)}_{(SM)i,j} - G^{(G)}_{(SM)i,j})^2$$

In [30]:
```
def S_cost(activations_S, activations_G):

    m,H,W,C= activations_G.get_shape().as_list()
    activations_S=tf.transpose(tf.reshape(activations_S, shape=[-1,C]))
    activations_G=tf.transpose(tf.reshape(activations_G, shape=[-1,C]))

    SM_S=style_matrix(activations_S)
    SM_G=style_matrix(activations_G)

    cost= (1/(2*C*H*W)**2)*tf.reduce_sum((SM_G-SM_S)**2)

    return cost
```

- We will find the Loss over all conv layers and give a weightage to each layer as a coefficient as shown:

In [31]:
```
style_weights=[
    ('conv1_1', 0.2),
    ('conv2_1', 0.2),
    ('conv3_1', 0.2),
    ('conv4_1', 0.2),
    ('conv5_1', 0.2)
]
```

- Now total Loss over layers

```
In [35]: def total_style_cost(model, style_weights):
    cost=0

    for layer_name, weightage in style_weights:

        activations_S= sess.run(model[layer_name])
        activations_G= model[layer_name]
        layer_cost= S_cost(activations_S,activations_G)

        cost=cost+weightage*layer_cost

    return cost
```

Total Loss would be:

$Loss(G) = \alpha Loss_{content}(C, G) + \beta Loss_{style}(S, G).$

- So we will minimize it with our optimizer

```
In [33]: def total_cost(cost_C, cost_S, alpha=10, beta=40):

    cost= (alpha*cost_C)+(beta*cost_S)

    return cost
```

Load Images

```
In [109]: C, origdims =load_preprocess("./images/C.png")
S, tardims =load_preprocess("./images/S.jpg")
G=generate_noisy(C,0.7)
```

Optimizer

- Reset the Graph and start TensorFlow Session

```
In [110]: sess.close()
tf.reset_default_graph()
sess= tf.InteractiveSession()
```

- Load the model in Graph

```
In [111]: model=matlab_to_tf_model(vgg)
```

- Select the Output Layer, I will be choosing conv4_2
- Compute the activations by running session over them
- Compute the losses

```
In [112]: sess.run(model['input'].assign(C))
output= model['conv4_2']

activations_C = sess.run(output)
activations_G = output

Cost_C= C_cost(activations_C,activations_G)


sess.run(model['input'].assign(S))

Cost_S= total_style_cost(model, style_weights)

COST = total_cost(Cost_C,Cost_S, alpha=10,beta=40)
```

- Make an Optimizer object, I will be using Adam with Learning Rate of 2.0
- At each Iteration it will minimize Total Loss

In [113]:
```
optimizer= tf.train.AdamOptimizer(2.0)

step= optimizer.minimize(COST)
```

NeuralArt Model

- Now let us define our art Model which is self interpretable
- After 100 Interactions it will show the generated Image and save it, you can change accordingly.
- The generated Images will be saved in Output Directory.
- Note the Generation Process may take longer on non GPU enabled devices

In [114]:
```
def artModel(sess, input_img, num_iter= 200):
    sess.run(tf.global_variables_initializer())
    sess.run(model['input'].assign(input_img))

    for i in range(num_iter):
        sess.run(step)
        gen_img= sess.run(model['input'])

        if i%100==0:
            print(i, sess.run(COST))
            plt.figure()
            img=cv2.resize(postprocess(gen_img[0]), (300,400))
            plt.imshow(img)
            plt.xticks(())
            plt.yticks(())
            plt.show()
            img=cv2.resize(img, (origdims[1],origdims[0]))
            img=cv2.cvtColor(img, cv2.COLOR_RGB2BGR)
            cv2.imwrite("./output/output"+str(i)+".jpg",img)
    img=postprocess(gen_img[0])
    plt.figure()
    img=cv2.resize(img, (origdims[1],origdims[0]))
    plt.title("Output:")
    plt.imshow(img)
    plt.xticks(())
    plt.yticks(())
    img=cv2.cvtColor(img, cv2.COLOR_RGB2BGR)


    cv2.imwrite("./output/output.jpg",img)
    plt.show()
```

In [115]:
```
artModel(sess, G, 800)
```

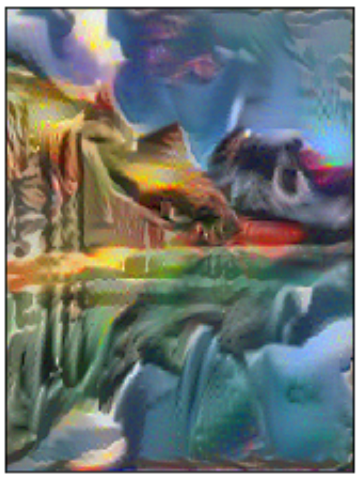
```
0 5621002000.0
```

```
100 148367120.0
```

200 78252710.0

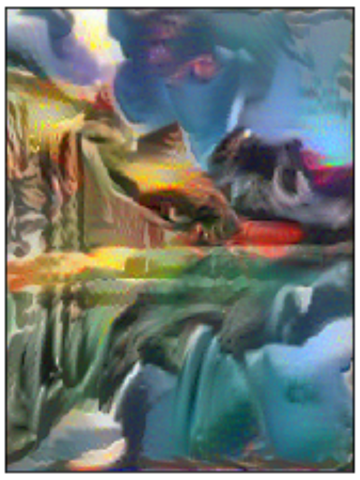
300 51728384.0

400 37663360.0

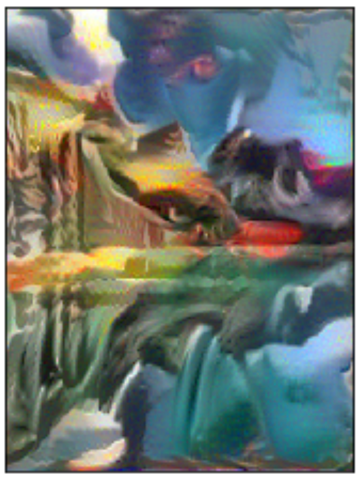


500 28781086.0

600 23160844.0



```
700 19235670.0
```



Output:

Thus multiple variations of results can be obtained by adjusting various hyperparameters

```
End
```