Artistic Neural Style Transfer using Convolution Neural Network

By Tejas Mahale

EE554/CSE586 Computer Vision Spring 2018

Abstract:

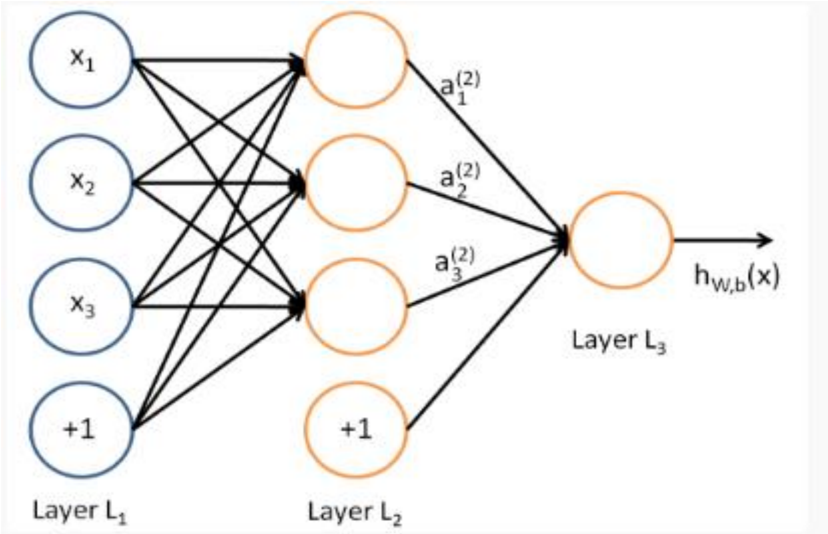
Neural nets are set of algorithms that are designed to recognise the pattern. They recognise sensor's data obtained by machine perception, labelling or clustering raw input data. Convolution neural network are similar to ordinary neural network[1]. In convolution neural net, we try to extract features using filters before passing them to fully connected neural layers[1]. Neural style transfer is one of interesting application of current research of ConvNet. Neural style transfer allows to generate new image form content image which has some features of style image[2]. Different layers of convolution neural net layers extracts different feature from images. This property can be effectively utilised to generate combined image attributes from content image and style image by applying appropriate ration of extraction. In this project, rather than training images from scratch, I am going to use pre-trained model which has already extracted features from ImageNet dataset to obtain style transfer image.

Table of content:

Content	Page No
1. Neural network	2
2. VGG	6
3. Neural Algorithm for Artistic style	7
4. Setup & Parameter tuning	10
5. Result	12
6. Conclusion	18
7. References	18
8. Project Members	18
9. IPython notebook code with outputs	19

1 Neural Network:

For convolution neural net, we need to properly understand deep neural network architecture. In deep neural net, multiple neural networks with connected neurons attached together. Initial layer is input layer which is fed to first hidden neuron layer as per fig 1.1.1



This is one hidden layer neural network. Layer L_1 as input layer, layer L_2 is hidden layer giving output to output layer layer L_1. Every layer has assigned weights and dot product of inputs and weights given to activation function which servers as input to next hidden layer. We will denote activation of unit i in layer l as $a_i^{(l)}$.

The computation of neural network for layer 2 is given as:

$$a_1^{(2)} = f(\, W_{11}^{(1)}\, x_1 + \; W_{12}^{(1)}\, x_2 + \; W_{13}^{(1)}\, x_3 + \; b_1^{(1)})$$

$$a_2^{(2)} = f(\, W_{21}^{(1)}\, x_1 + \; W_{22}^{(1)}\, x_2 + \; W_{23}^{(1)}\, x_3 + \; b_2^{(1)})$$

$$a_3^{(2)} = f(\, W_{31}^{(1)}\, x_1 + \; W_{32}^{(1)}\, x_2 + \; W_{33}^{(1)}\, x_3 + \; b_3^{(1)})$$

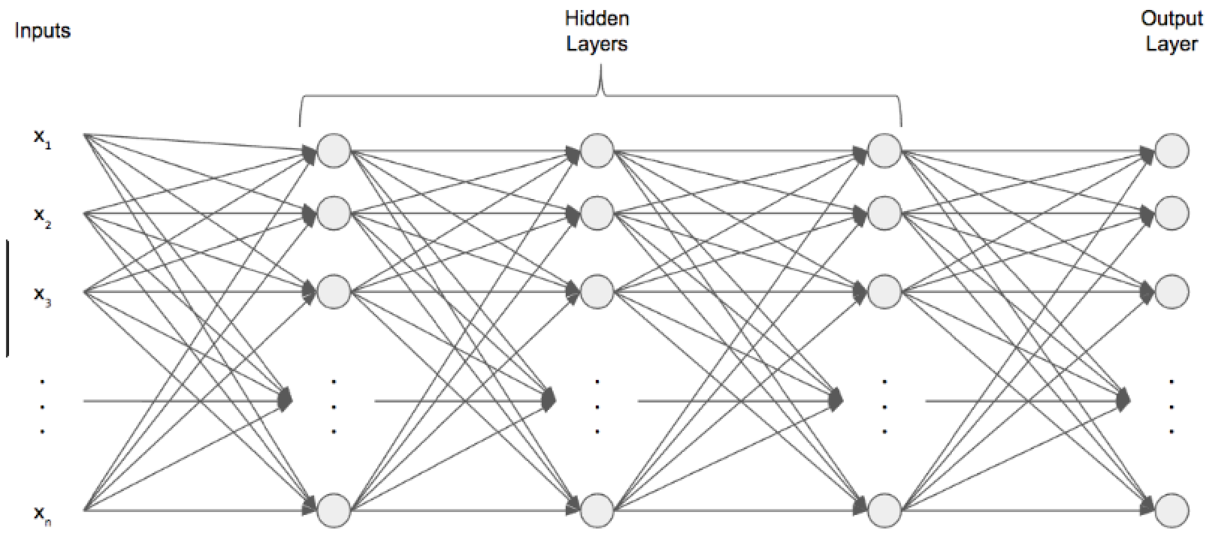
$$h_{w,b}(x) = a_1^{(3)} = \; f(\, W_{11}^{(2)}\, x_1 + \; W_{12}^{(2)}\, x_2 + \; W_{13}^{(2)}\, x_3 + \; b_1^{(2)})$$

here $f()$ is called as activation function. In general for forward propogation,

$$z^{(l+1)} = \; W^{(l)}a^{(l)} + b^{(l)}$$

$$a^{(l+1)} = f(z^{(l+1)})$$

l is current layer. We can generalise this result to multi-layer neural network.



Back propagation:

For m example we will try to find cost function which is difference between actual output label and internally generated label by neural network[4].

$$J(W,b) = \left[\frac{1}{m}\sum_{i=1}^{m}\left(\frac{1}{2}\left\|h_{W,b}\left(x^{(i)}\right) - y^{(i)}\right\|^2\right)\right] + \frac{\lambda}{2}\sum_{l=1}^{n_i-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}(W_{ji}^{(l)})^2$$

We need to find values of parameters W, b such that above cost function has minimum value.

here we can use derivatives to achieve global minima

$$\frac{\partial(J(W,b))}{\partial W_{ij}^{(l)}} = \left[\frac{1}{m}\sum_{i=1}^{m}\frac{\partial J\left(W,b;x^{(i)},y^{(i)}\right)}{\partial W_{ij}^{(l)}}\right] + \lambda W_{ij}^{(l)}$$

$$\frac{\partial(J(W,b))}{\partial b_{i}^{(l)}} = \left[\frac{1}{m}\sum_{i=1}^{m}\frac{\partial J\left(W,b;x^{(i)},y^{(i)}\right)}{\partial b_{ij}^{(l)}}\right]$$

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha\frac{\partial J(W,b)}{\partial W_{ij}^{(l)}}$$

$$b_{i}^{(l)} = b_{i}^{(l)} - \alpha\frac{\partial J(W,b)}{\partial b_{i}^{(l)}}$$

here α is learning rate

1.1 Convolution Neural Network:

Natural images have the property of being ”‘stationary”’, meaning that the statistics of one part of the image are the same as any other part. This suggests that the features that we learn at one part of the image can also be applied to other parts of the image, and we can use the same features at all locations[4].

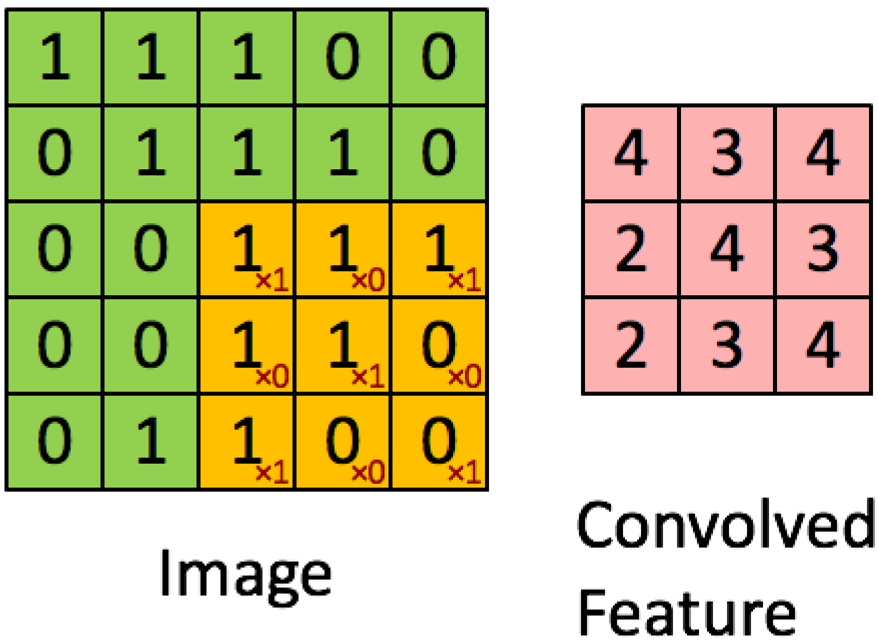


fig 1.1.3

In convolution layer we take image convolve with filter which represents particular feature of image. CovNet is made up layers. Each layer transforms input 3D volume to output 3D volume with some differentiable function with or without patameters[3].

Suppose image has size n x n x 3 convolve with 20 filters of size f x f x 3 we get output as n-f +1 x n-f+1 x 3. But this is general case we never use such dimension to compute output. Because when we use filter with convolution, it won't work for outer layer of image which reduced dimension of image ultimately it reduces information which can't be ignored. We need to understand that 3rd dimension of output is number of filter used rather than related to 3rd dimension of input. Here we introduce concept of padding and stride selection. Lets discuss more on this in ConvNet Layers section

1.2 ConvNet Layers:

Convolution layers:

As discussed earlier, in this layer we implement information extraction using convolution. We can use randomly generated weight filters or custom filters like edge detection filter.

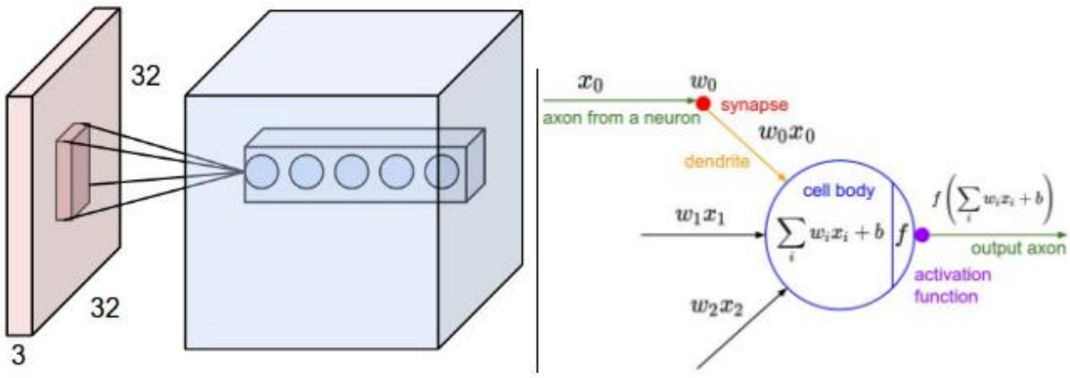


fig 1.3.1

Padding:

To take care of outer layer while filtering, we use padding. We pads image on both dimensions so filter can work on boundaries of image. We usually use zero padding.

Stride:

Stride number shows number of pixels that our filter will move while filtering. When the stride is 1 then we move the filters one pixel at a time. stride value can be 2 or more but higher value of stride will reduce efficiency of filter.

So number of neurons fits in layer is given by formula:

$$(n - f + 2P)/S + 1$$

n x n → Size of filter

f x f → filter size

P → Padding

S → Stride

2. VGG

This model is trained on a subset of the ImageNet database, which is used in the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC). VGG-19 is trained on more than a million images and can classify images into 1000 object categories.

Layers:

1	'input'	Image Input	224x224x3 images with 'zerocenter' normalization
2	'conv1_1'	Convolution	64 3x3x3 convolutions with stride [1 1] and padding [1 1]
3	'relu1_1'	ReLU	ReLU
4	'conv1_2'	Convolution	64 3x3x64 convolutions with stride [1 1] and padding [1 1]
5	'relu1_2'	ReLU	ReLU
6	'pool1'	Max Pooling	2x2 max pooling with stride [2 2] and padding [0 0]
7	'conv2_1'	Convolution	128 3x3x64 convolutions with stride [1 1] and padding [1 1]
8	'relu2_1'	ReLU	ReLU
9	'conv2_2'	Convolution	128 3x3x128 convolutions with stride [1 1] and padding [1 1]
10	'relu2_2'	ReLU	ReLU
11	'pool2'	Max Pooling	2x2 max pooling with stride [2 2] and padding [0 0]
12	'conv3_1'	Convolution	256 3x3x128 convolutions with stride [1 1] and padding [1 1]
13	'relu3_1'	ReLU	ReLU
14	'conv3_2'	Convolution	256 3x3x256 convolutions with stride [1 1] and padding [1 1]
15	'relu3_2'	ReLU	ReLU
16	'conv3_3'	Convolution	256 3x3x256 convolutions with stride [1 1] and padding [1 1]
17	'relu3_3'	ReLU	ReLU
18	'conv3_4'	Convolution	256 3x3x256 convolutions with stride [1 1] and padding [1 1]
19	'relu3_4'	ReLU	ReLU
20	'pool3'	Max Pooling	2x2 max pooling with stride [2 2] and padding [0 0]
21	'conv4_1'	Convolution	512 3x3x256 convolutions with stride [1 1] and padding [1 1]
22	'relu4_1'	ReLU	ReLU
23	'conv4_2'	Convolution	512 3x3x512 convolutions with stride [1 1] and padding [1 1]
24	'relu4_2'	ReLU	ReLU
25	'conv4_3'	Convolution	512 3x3x512 convolutions with stride [1 1] and padding [1 1]
26	'relu4_3'	ReLU	ReLU
27	'conv4_4'	Convolution	512 3x3x512 convolutions with stride [1 1] and padding [1 1]
28	'relu4_4'	ReLU	ReLU
29	'pool4'	Max Pooling	2x2 max pooling with stride [2 2] and padding [0 0]
30	'conv5_1'	Convolution	512 3x3x512 convolutions with stride [1 1] and padding [1 1]
31	'relu5_1'	ReLU	ReLU
32	'conv5_2'	Convolution	512 3x3x512 convolutions with stride [1 1] and padding [1 1]
33	'relu5_2'	ReLU	ReLU
34	'conv5_3'	Convolution	512 3x3x512 convolutions with stride [1 1] and padding [1 1]
35	'relu5_3'	ReLU	ReLU
36	'conv5_4'	Convolution	512 3x3x512 convolutions with stride [1 1] and padding [1 1]
37	'relu5_4'	ReLU	ReLU
38	'pool5'	Max Pooling	2x2 max pooling with stride [2 2] and padding [0 0]
39	'fc6'	Fully Connected	4096 fully connected layer
40	'relu6'	ReLU	ReLU
41	'drop6'	Dropout	50% dropout
42	'fc7'	Fully Connected	4096 fully connected layer
43	'relu7'	ReLU	ReLU
44	'drop7'	Dropout	50% dropout
45	'fc8'	Fully Connected	1000 fully connected layer
46	'prob'	Softmax	softmax
47	'output'	Classification Output	crossentropyex with 'tench', 'goldfish', and 998 other classes

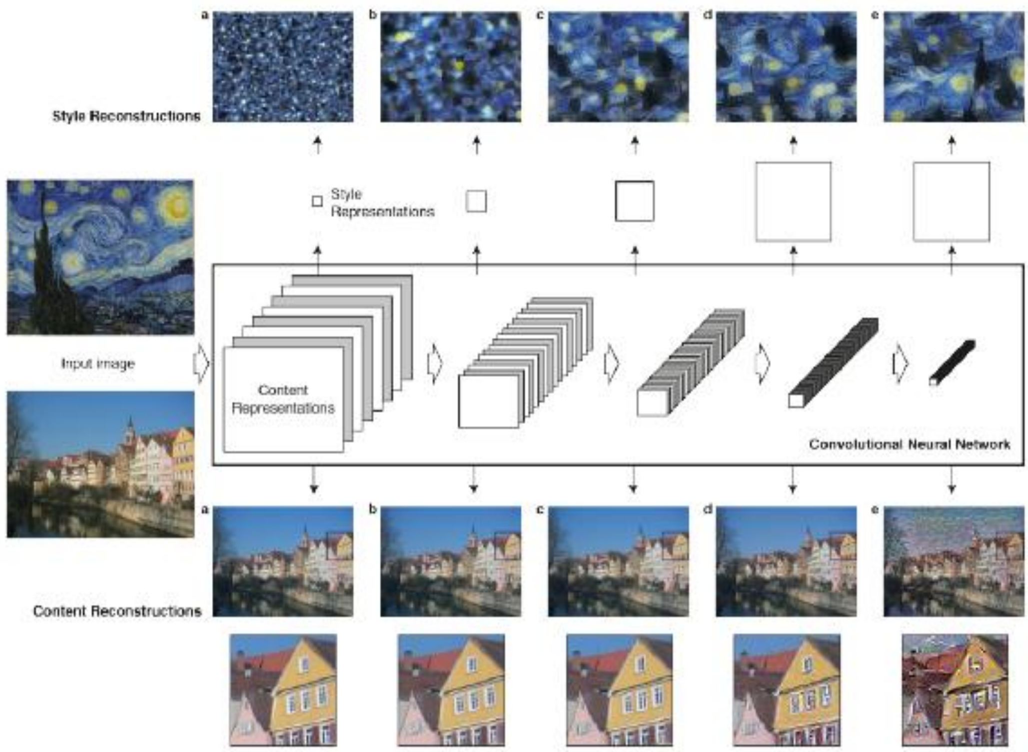
For this project I am not using last 3 fully connected layers with classification layer for 1000 classes.

3 Neural Algorithm for Artistic Style:

3.1 Introduction

When CNN are trained for object recognition, feature can be extracted which makes object information more explicit along processing hierarchy(layers). So along layers, input image converts into representation of feature that is relevant to actual content of image. Interestingly, it is possible to visualise each layer image by reconstructing image from feature map in that layer. In ConvNet, more the number of layer, higher the degree of feature extraction. So high level content of corresponding object can be extracted using increasing number of layers. This is basic idea of neural algorithm for artistic style[4].

In this we merge content image with style image to get generated image. Generated image contains feature from content image on top of style feature of style image. Neural style transfer uses previously trained network and transfers feature by mounting more layer on pre-trained network. In this project, I am using VGG 19 network which is already trained on ImageNet dataset. First we extract features from content image and style image on pre-trained network and we try to optimise the combine cost of both images.



3.2 Cost computation:

First we train the input content image on pre-trained VGG, run the forward propogation and find activation value $a^{(C)[l]}$ for layer l. Similarly we train some randomly values noissy image(which later becomes generated image) for same VGG network to get activation $a^{(G)[l]}$ for layer l.

Then content cost is given by:

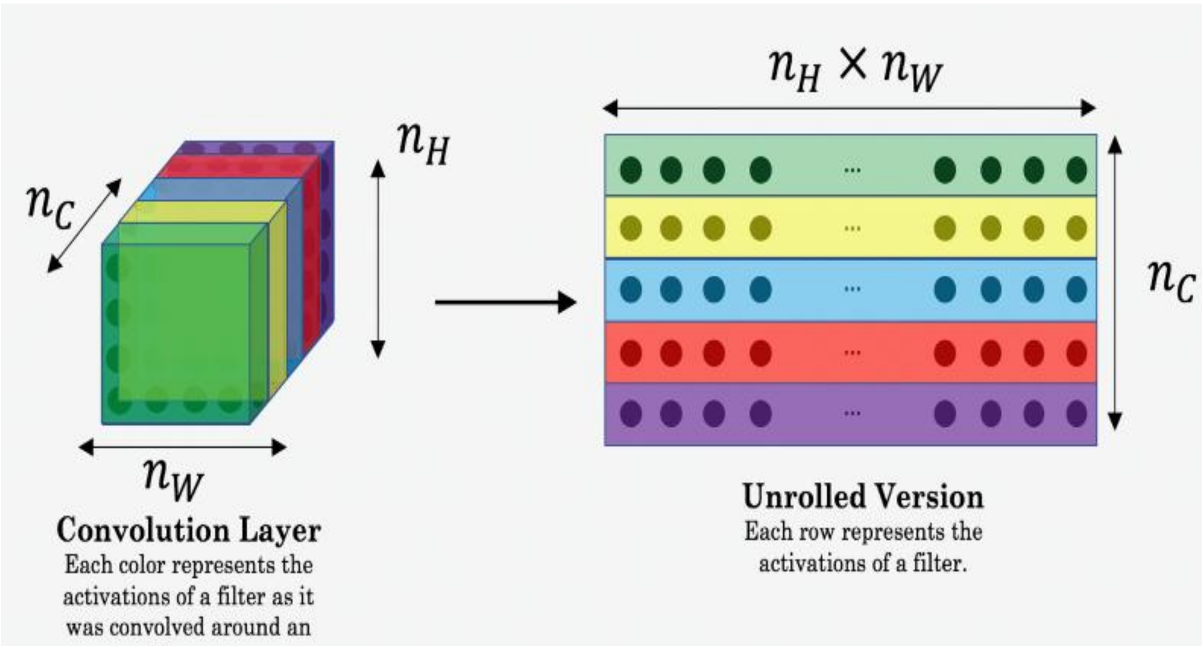
$$J_{content}\ (C, G) = \frac{1}{4 * n_H * n_W * n_C} \sum_{all\ entries} (a^{(C)[l]} - a^{(G)[l]})^2$$

$n_H * n_w$ = Height * Width of content image

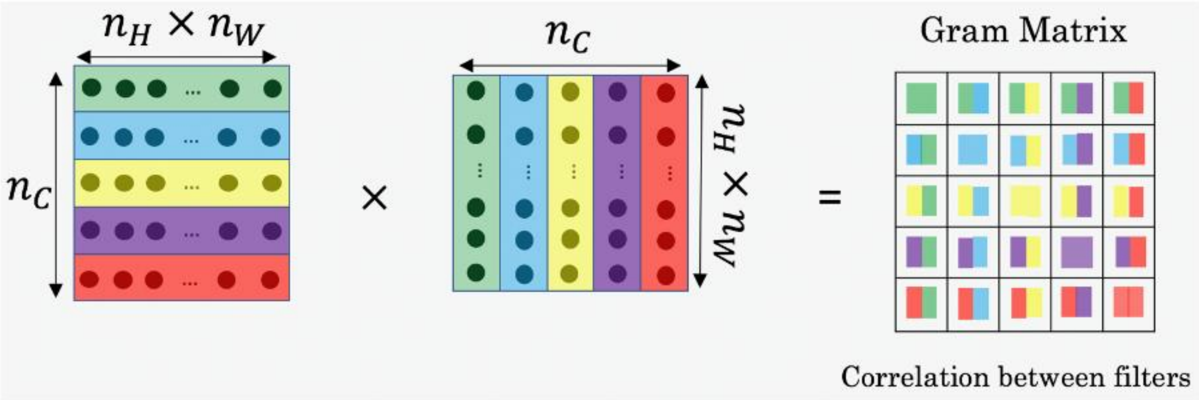
n_C = Number of Channels in filter

$a^{(C)[l]}$= Activation of content image

$a^{(G)[l]}$= Activation of generated image(in this case noisy image)



The style of an image can be represented using the Gram matrix of a hidden layer's activations. These representation is combined with number of layer for good result[4].



Here G_{ij} measures similarity between activation of filter i and filter j.

$$J_{style}{}^{[l]}(S,G) = \frac{1}{4 * n_c^2 * (n_H * n_w)^2} \sum_{i=1}^{n_c} \sum_{j=1}^{n_c} (G_{ij}{}^{S} - G_{ij}{}^{G})^2$$

$G_{ij}{}^{S}$= Gram matrix of style image

$G_{ij}{}^{G}$= Gram matrix of generated image

$J_{style}{}^{[l]}(S,G)$= Style cost of single layer l

Then total style cost over total layer l is given as:

$$\boldsymbol{J_{style}(S,G) = \sum_l \lambda^l * J_{style}{}^{[l]}(S,G)}$$

λ^l = style layer coefficient

Now with content cost and style cost, we can find total cost which is linear combination of both the cost.

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

Here α & β represents respective proportion of content cost and style cost which should be minimized.

Now we have defined cost which was calculated in forward propagation from pre-trained network. This cost can be optimized using backward propagation to get artistic neural style.

4. Setup and Parameter tuning:

4.1 Framework :

No.	Tool
1.	Python 3.6.5
2.	TensorFlow 1.7
3.	Tesla K80 GPU (12Gbyte video memory)
4.	Google Colaboratory(IPython notebook)

4.2 Transfer learning:

Advantage of pre-trained network is that we don't have to train model on whole dataset from scratch and still possible to achieve good results. In this project I am using VGG model (VGG-19) which has been trained on large ImageNet dataset. So that pre-trained model is sufficient and explicit dataset is not needed. We only needed one image as content image and other image as style image. Here we don't require 1000 class classifier and fully connected layers. Hence classifier output layer as well as last 3 fully connected layers are removed. In this application, I am using 'conv4_3' layer form VGG to get visualization.

Link: http://www.vlfeat.org/matconvnet/pretrained/

Model : imagenet-vgg-verydeep-19

4.3 Normalize and Reshape steps:

1. First we need to generate noisy image which will act as generated image. We try to minimize cost of this noisy image with respect to content and style image to get final generated image. Noise image is generated using numpy np.random.uniform function between poins -20 to 20 pixel values. Final input image(generated image from noise) is given as
   input_image = noise_image * noise_ratio + cont_img * (1 - noise_ratio)

2. For given pre-trained VGG model, mean values after normalisation are given as 123.68, 116.779, 103.939 in height, width, channels respectively. These values are used to normalise image shape.
3. Input of pre-trained VGG model expects image resolution as 400x300x3 Hence using CV2 library in python, input is resized.

4.4 Model:

As explained earlier, I am using VGG model. This is exact same model which has been given by VGG paper[3]. TensorFlow code of VGG model is pretty standard which can be used directly. Idea of this is taken from following link:
https://github.com/chiphuyen/stanford-tensorflow-tutorials/blob/master/2017/assignments/style_transfer_starter/vgg_model.py

4.5 Parameters/Hyper-parameters:

Hyper-parameters	Value	Reason
1. Style layer coefficient	'conv1_1', 0.1 'conv2_1', 0.3 'conv3_1', 0.3 'conv4_1', 0.3 'conv5_1', 0.1	Every layer may have same values. It's about weighting on certain layers.
2. Content & Style optimization proportion (α, β)	α= 2 or 1 for content image β = 3 or 5 for style image	Style coefficient should be higher for higher optimization of style
3. Noise_ratio	0.6	Initially less part of content should be in noisy image
4. Learning rate	0.5	Tried combinations to get good image styler image
5. Iteration	2000	Minimized cost improved sense of image but took more time
6. Visualization layer	'conv4_3'	Deep layer has deep features

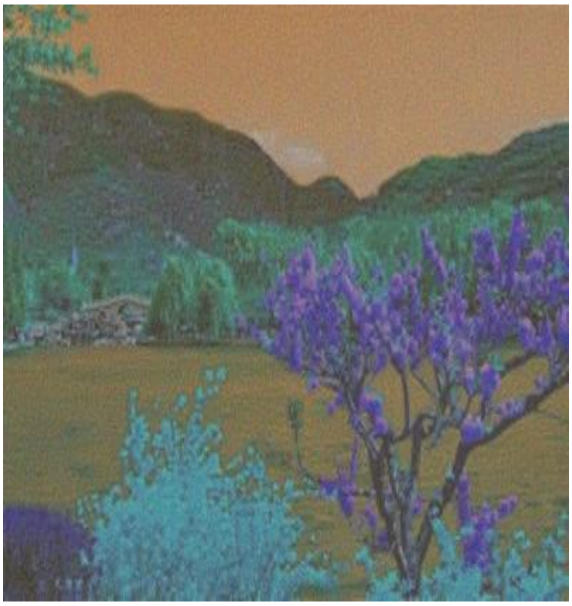
5. Results

Result 1:

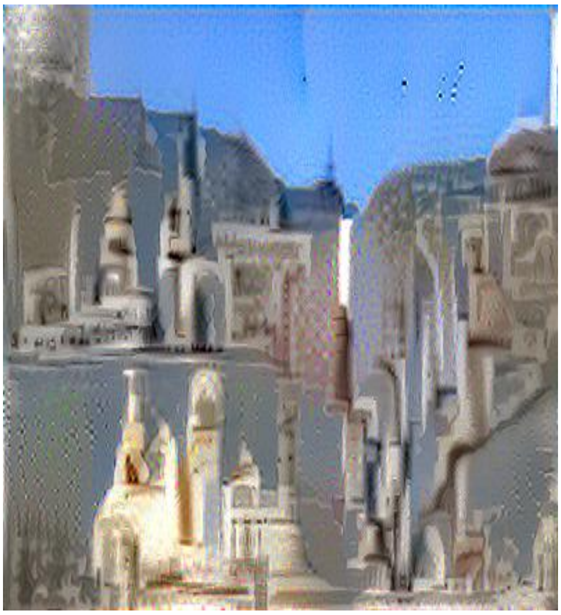
Content Image

Style Image

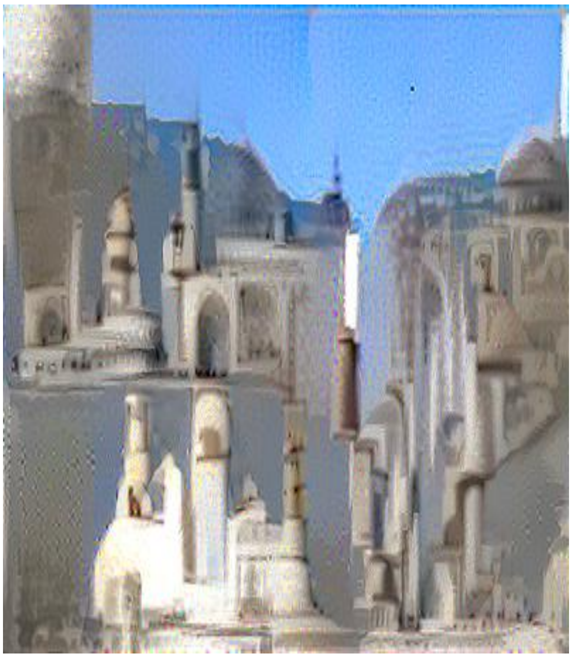
Generated Images:



(0 iteration)



(500 iteration)



(1000 iterations)

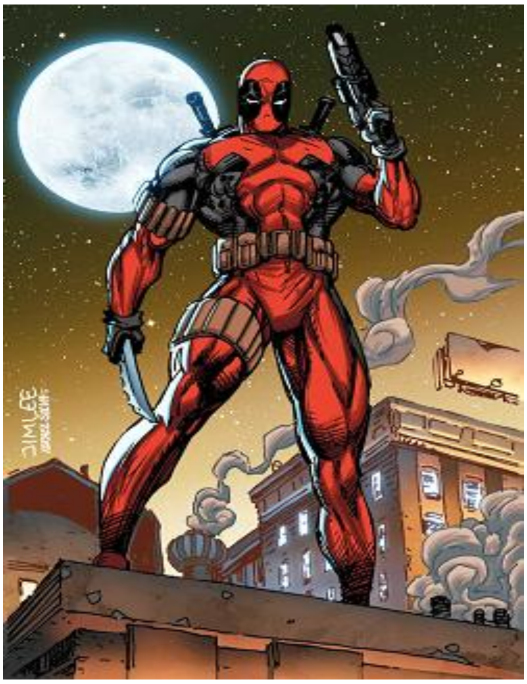
(1500 iterations)

(2000 iterations)

Interestingly final image is similar to architecture of middle east monuments from basic idea of Taj Mahal came. Overall it is giving feel of Arab city. Here choice of style image was not proper. But my code work optimized images clearly as you can see that edges of style image are filled with features of content image.

Result 1:

Content Image



Style Image

+

Generated images:

(0 iterations)

(500 iterations)

(1000 iterations)

(1500 iterations)

(2000 iterations)

Finally super hero fans would be happy to see DC comic hero Green Lantern and Marvel Deadpool combined have decent look!!

In this case, from 1000 iterations to 2000 iteration, no significant difference in image is seen. But numerical results shows that with iterations, loss decreases.

Result:

```
Iteration number: 0 :
Total cost after these iterations = 17802367000.0
```

```
content cost after these iterations= 12447.214
style cost after these iterations= 593403900.0
Iteration number: 100 :
Total cost after these iterations = 299030400.0
content cost after these iterations= 24503.096
style cost after these iterations= 9951345.0
Iteration number: 200 :
Total cost after these iterations = 120466810.0
content cost after these iterations= 25994.969
style cost after these iterations= 3998230.5
Iteration number: 300 :
Total cost after these iterations = 62105668.0
content cost after these iterations= 26905.791
style cost after these iterations= 2052251.8
Iteration number: 400 :
Total cost after these iterations = 38295970.0
content cost after these iterations= 27529.902
style cost after these iterations= 1258178.9
Iteration number: 500 :
Total cost after these iterations = 27516700.0
content cost after these iterations= 28043.836
style cost after these iterations= 898527.5
Iteration number: 600 :
Total cost after these iterations = 21908864.0
content cost after these iterations= 28474.363
style cost after these iterations= 711312.56
Iteration number: 700 :
Total cost after these iterations = 18445444.0
content cost after these iterations= 28826.568
style cost after these iterations= 595630.4
Iteration number: 800 :
Total cost after these iterations = 16016844.0
content cost after these iterations= 29095.898
style cost after these iterations= 514497.53
Iteration number: 900 :
Total cost after these iterations = 14191179.0
content cost after these iterations= 29320.258
style cost after these iterations= 453492.47
Iteration number: 1000 :
Total cost after these iterations = 12732073.0
content cost after these iterations= 29519.482
style cost after these iterations= 404722.78
Iteration number: 1100 :
Total cost after these iterations = 11524371.0
content cost after these iterations= 29708.98
style cost after these iterations= 364339.7
Iteration number: 1200 :
Total cost after these iterations = 10507995.0
content cost after these iterations= 29898.307
style cost after these iterations= 330334.3
Iteration number: 1300 :
Total cost after these iterations = 9647478.0
content cost after these iterations= 30069.217
style cost after these iterations= 301536.47
Iteration number: 1400 :
```

```
Total cost after these iterations = 8913836.0
content cost after these iterations= 30225.547
style cost after these iterations= 276977.5
Iteration number: 1500 :
Total cost after these iterations = 8280933.5
content cost after these iterations= 30369.916
style cost after these iterations= 255784.5
Iteration number: 1600 :
Total cost after these iterations = 7728743.0
content cost after these iterations= 30514.178
style cost after these iterations= 237281.98
Iteration number: 1700 :
Total cost after these iterations = 7245909.5
content cost after these iterations= 30650.295
style cost after these iterations= 221096.78
Iteration number: 1800 :
Total cost after these iterations = 6822920.5
content cost after these iterations= 30788.88
style cost after these iterations= 206904.77
Iteration number: 1900 :
Total cost after these iterations = 6457366.0
content cost after these iterations= 30910.451
style cost after these iterations= 194638.56
```

6. Conclusion:

1. With help of neural style transfer, content image and style image generates new artistic image.
2. Hidden layer activations individually play important role in deep learning applications.
3. Accurate value of hyper parameter is depend on specific application. Sometimes just reducing cost by increasing number of iterations makes no sense. This is neural style transfer is perfect application of this as image at 1000 iterations is same as image at 2000 iterations.
4. Using pre-trained networks with transfer learning can save lot of time as well as computation cost.

7. References:

[1] Wikipedia

[2] Cs 321 Stanford course notes

[3] Karen Simonyan & Andrew Zisserman, "very deep convolutional networks for large-scale image recognition".

[4] Leon A. Gatys, Alexander S. Ecker, Matthias Bethge"A Neural Algorithm of Artistic Style".

8. Contribution and Project group members:

[1] Tejas Mahale (only)

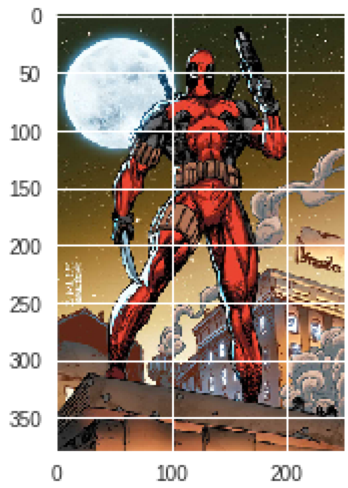
```
import os
import sys
import scipy.io
import scipy.misc
import matplotlib.pyplot as plt
from matplotlib.pyplot import imshow
from PIL import Image
#from nst_utils import *
import numpy as np
import tensorflow as tf

%matplotlib inline
```

```
content_directory = "drive/Computer Vision_Neural Styler/Deadpool.jpg"
style_directory = "drive/Computer Vision_Neural Styler/Green.jpg"
output_image_dir = "drive/Computer Vision_Neural Styler/Output_images_6/"
pretrained_model_dir = "drive/Computer Vision_Neural Styler" + "/" + "imagenet-vgg-verydeep-
```

```
cont_img = scipy.misc.imread(content_directory)
imshow(cont_img)
```

```
<matplotlib.image.AxesImage at 0x7fb036fc03c8>
```



```
print('Content Image shape:', cont_img.shape)
```

```
Content Image shape: (379, 250, 4)
```

```
def cost_of_content(C_dim, G_dim):
  m, n_H, n_W, n_C = G_dim.get_shape().as_list()
  C_dim_changed= tf.transpose(tf.reshape(C_dim, [-1]))
  G_dim_changed = tf.transpose(tf.reshape(G_dim, [-1]))

  content_cost = tf.reduce_sum((C_dim_changed - G_dim_changed)**2) / (4 * n_H * n_W * n_C)
  return content_cost
```

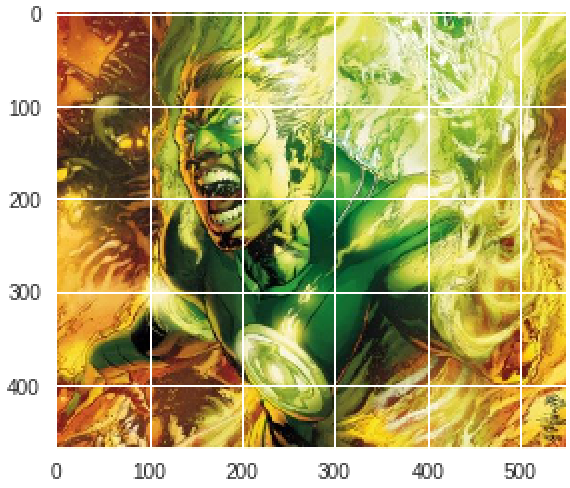
```
tf.reset_default_graph()

with tf.Session() as test:
    tf.set_random_seed(1)
    C_dim_changed = tf.random_normal([1, 4, 4, 3], mean=1, stddev=4)
    G_dim_changed = tf.random_normal([1, 4, 4, 3], mean=1, stddev=4)
    content_cost = cost_of_content(C_dim_changed, G_dim_changed)
    print("Cost of Content is " + str(content_cost.eval()))
```

```
Cost of Content is 6.7655935
```

```
style_image = scipy.misc.imread(style_directory)
imshow(style_image)
```

```
<matplotlib.image.AxesImage at 0x7fb018d94c88>
```



```
print('Style image shape:', style_image.shape)
```

```
Style image shape: (469, 550, 3)
```

```
def cost_of_style_layer(S_dim, G_dim):
  m, n_H, n_W, n_C = G_dim.get_shape().as_list()
  S_dim = tf.reshape(S_dim, [n_H*n_W, n_C])
  G_dim = tf.reshape(G_dim, [n_H*n_W, n_C])

  GS = tf.matmul(tf.transpose(S_dim), S_dim)
  GG = tf.matmul(tf.transpose(G_dim), G_dim)

  Style_cost = tf.reduce_sum((GS - GG)**2) / (4 * n_C**2 * (n_W * n_H)**2)

  return Style_cost
```

```
tf.reset_default_graph()

with tf.Session() as test:
    tf.set_random_seed(1)
    S_dim = tf.random_normal([1, 4, 4, 3], mean=1, stddev=4)
    G_dim = tf.random_normal([1, 4, 4, 3], mean=1, stddev=4)
    Style_cost = cost_of_style_layer(S_dim, G_dim)

    print("Cost of Style is " + str(Style_cost.eval()))
```

```
Cost of Style is 9.190278
```

```
STYLE_LAYERS = [
    ('conv1_1', 0.2),
    ('conv2_1', 0.2),
    ('conv3_1', 0.2),
    ('conv4_1', 0.2),
    ('conv5_1', 0.2)]
```

```
def cost_of_style(model, STYLE_LAYERS):
  J=0
  for layer, value in STYLE_LAYERS:
    ip_layer = model[layer]
    S_dim = sess.run(ip_layer)
    G_dim = ip_layer
    cost_of_style_lyr = cost_of_style_layer(S_dim, G_dim)
```

```
        J= J + value * cost_of_style_lyr

    return J
```

```
alpha = 8
beta = 40
tf.reset_default_graph()

with tf.Session() as test:
    np.random.seed(3)
    content_cost = np.random.randn()
    J = np.random.randn()
    J_tot = alpha * content_cost + beta * J
    print("Total content + style cost is  = " + str(J_tot))
```

```
Total content + style cost is  = 31.769421807922125
```

```
tf.reset_default_graph()

# Start interactive session
sess = tf.InteractiveSession()
```

```
import cv2
def resize(img_dir):

  img = cv2.imread(img_dir)
  img = cv2.resize(img,dsize=(400,300))
  out = np.array(img)
  return out
```

```
cont_img = resize(content_directory)
style_img = resize(style_directory)
print(cont_img.shape)
print(style_img.shape)
```

```
(300, 400, 3)
(300, 400, 3)
```

```
def reshape_and_normalize_image(image):
    # Reshape image to mach expected input of VGG19
    image = np.reshape(image, ((1,) + image.shape))
    MEANS = np.array([123.68, 116.779, 103.939]).reshape((1,1,1,3))
    # Substract the mean to match the expected input of VGG19
    image = image - MEANS

    return image
```

```
cont_img=reshape_and_normalize_image(cont_img)
```

```
print(cont_img.shape)
```

```
(1, 300, 400, 3)
```

```
style_image = reshape_and_normalize_image(style_img)
print(style_image.shape)
```

```
(1, 300, 400, 3)
```

```
def generate_noise_image(cont_img, noise_ratio = 0.6):
```

```
    noise_image = np.random.uniform(-20, 20, (1, 300, 400, 3)).astype('float32')
    input_image = noise_image * noise_ratio + cont_img * (1 - noise_ratio)

    return input_image


noisy_content__image = generate_noise_image(cont_img)



def Load_vgg_model(path):
    vgg = scipy.io.loadmat(path)

    vgg_layers = vgg['layers']

    def _weights(layer, expected_layer_name):

        weight_bias = vgg_layers[0][layer][0][0][2]
        W = weight_bias[0][0]
        b = weight_bias[0][1]
        layer_name = vgg_layers[0][layer][0][0][0][0]
        return W, b

    def _relu(conv2d_layer):
        return tf.nn.relu(conv2d_layer)

    def _conv2d(prev_layer, layer, layer_name):
        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 _relu(_conv2d(prev_layer, layer, layer_name))

    def _avgpool(prev_layer):
        return tf.nn.avg_pool(prev_layer, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding=


    graph = {}
    graph['input']   = tf.Variable(np.zeros((1, 300, 400, 3)), 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')
    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


model = Load_vgg_model(pretrained_model_dir)


sess.run(model['input'].assign(cont_img))
output = model['conv4_3']
C_dim = sess.run(output)
```

```
G_dim = output
Final_content_cost = cost_of_content(C_dim, G_dim)


sess.run(model['input'].assign(style_image))
Final_cost_style= cost_of_style(model, STYLE_LAYERS)


alpha = 20
beta = 30
Final_total_cost = alpha * Final_content_cost + beta * Final_cost_style


optimizer = tf.train.AdamOptimizer(0.5)


train_step = optimizer.minimize(Final_total_cost)


def save_image(path, image):

    # Un-normalize the image so that it looks good
    image = image + np.array([123.68, 116.779, 103.939]).reshape((1,1,1,3))

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


def Art_neural_model(sess, ip):
  iter = 2000
  sess.run(tf.global_variables_initializer())
  sess.run(model['input'].assign(ip))

  for k in range(iter):
    dk = sess.run(train_step)
    generated_image = sess.run(model['input'])

    if k%100 == 0:

      a,b,c = sess.run([Final_total_cost, Final_content_cost, Final_cost_style])
      print("Iteration number: " + str(k) + " :")
      print("Total cost after these iterations = " + str(a))
      print("content cost after these iterations= " + str(b))
      print("style cost after these iterations= " + str(c))
      save_image(output_image_dir + str(k) + ".png", generated_image)


  save_image(output_image_dir+'/'+'Final_generated_image.jpg', generated_image)
  return generated_image


Art_neural_model(sess,noisy_content__image)
```

```
style cost after these iterations= 766546.6
Iteration number: 1800 :
Total cost after these iterations = 21401854.0
content cost after these iterations= 11450.767
style cost after these iterations= 705761.25
Iteration number: 1900 :
Total cost after these iterations = 19864732.0
content cost after these iterations= 11513.806
style cost after these iterations= 654481.9
array([[[[ -30.467806 ,  -39.001022 ,  -10.462537 ],
         [ -53.742077 ,  -55.460297 , -118.07169  ],
         [ -90.01758  ,  -42.3311   ,  -31.774616 ],
         ...,
         [ -63.505833 ,  -59.667336 ,  -31.35586  ],
         [ -83.71975  ,  -51.89839  , -185.52179  ],
         [ -68.34331  ,  -68.71208  ,  -92.811005 ]],

        [[ -70.72339  ,  -62.7815   ,  -83.58585  ],
         [ -75.5782   ,  -45.24441  ,  -53.21999  ],
         [ -82.33038  ,  -37.60445  ,   39.002598 ],
         ...,
         [ -87.96739  ,  -46.369858 ,  -18.89513  ],
         [ -86.305305 ,  -50.377953 ,  -89.49467  ],
         [-105.40438  ,  -44.002853 ,  -94.78533  ]],

        [[ -68.94241  ,  -59.86015  ,   44.333164 ],
         [ -68.85777  ,  -45.622635 ,   16.840427 ],
         [ -75.56593  ,   -1.3836285,   56.810608 ],
         ...,
         [ -85.18602  ,  -65.026764 ,  -16.054668 ],
         [-106.80345  ,  -55.664387 ,  -47.323006 ],
         [ -68.49692  ,  -53.150497 ,  -31.829712 ]],

        ...,

        [[ -39.98747  ,  -94.69842  ,  -20.304136 ],
         [ -96.47846  ,  -98.8691   ,  -69.49347  ],
         [ -90.070984 , -123.469315 ,  -15.774889 ],
         ...,
         [ -79.005264 ,  103.23915  ,  109.096214 ],
         [ -90.41783  ,  -48.35042  ,   -6.36548  ],
         [ -92.77473  ,  -58.463158 ,  -12.205064 ]],

        [[ -10.6789665,  -85.92047  ,  -87.565735 ],


         ...,
         [ -64.16634  ,   -5.140916 ,   15.596747 ],
         [ -83.360596 ,  -72.34371  ,  -48.192642 ],
         [-128.79027  ,  -94.1179   ,  -45.28116  ]],

        [[  30.35327  ,  -58.04929  ,   -1.799109 ],
         [  40.20036  ,  -80.669    ,  -71.013664 ],
         [  -8.286025 , -154.48383  ,  -57.40033  ],
         ...,
         [ -31.047516 ,    9.031505 ,   44.11404  ],
         [ -66.467735 ,  -61.22342  ,    8.97068  ],
         [  -9.610145 ,  -10.393085 ,   27.895872 ]]]], dtype=float32)
```