**Sudeshna Dasgupta**

**18BCB0016**

**EXC1021**

**Prof. Govinda K.**

**Title: Neural Artistic Style Transfer Exploration**

**Aim:**

After coming across the popular paper by Gatys et al., depicting seminal work on style transfer, and various research articles based on improving the aesthetics of style transfer, to improving speed, to stylizing videos, and analysing hyper-parameter tuning, I was deeply interested by this new technological development. I wondered if it would be possible to express our own lived experiences through the unique visual style the medium provides and use this technique as an effective way to convey emotions and other abstract information. In this project I’m trying to create remarkable style transfer effects and apply a new style to an image while still preserving its original content. **[1]**

**Abstract:**

**Neural Style Transfer** (NST) is a class of software algorithms that manipulates digital videos and images, to adopt the appearance or visual style of another video or image. Neural Style Transfer algorithms use deep neural networks to perform image transformation.

**NST** is an optimization based computer vision technique that allows us to manipulate and thus recompose an image’s content in another image’s style. It takes 2 images—a *content image* and a *style image* for *reference* and blends both of them together to get an output image, thereby making it appear as if it’s been painted in the style of the style image, in spite of retaining the core elements of the content image.

Example: A portrait photograph captured by a street photographer could supply the image content, and van Gogh’s masterpiece, ‘The Starry Night’ could be the reference style image. The resultant output image would be a self-portrait that looks like an original van Gogh creation. **[2]**

Neural Artistic Style Transfer model consists of two sub-models:

1. **Style Transform Model** is a neural network that creates a stylised image after taking an image and then applying a style bottleneck vector to the content image.
2. **Style Prediction Model** is a neural network that takes a style image as input to a 100-dimension style bottleneck vector. **[2]**

**How does neural artistic style transfer work?**

1. We take an input image and a style image and then resize them to equal dimensions.
2. Then we load a pre-trained Convolutional Neural Network (CNN) such as VGG19.
3. We can distinguish layers of the CNN that are the responsible for the content (image-specific features) and layers which are responsible for style (basic shapes, colours etc.), we can separate the different layers of the CNN to independently work on the style and content.
4. Next we have to work on the **optimization problem** using which we try to minimize:

* **Content loss** (distance between the output and input images - we strive to preserve the content)
* **Style loss** (distance between the output and style images - we strive to apply a new style)
* **Total variation loss** (regularisation - spatial smoothness to de-noise the output image)

5. Set the gradients and perform optimization.

I have made use of the L-BFGS optimization algorithm (Limited-memory Broyden–Fletcher–Goldfarb–Shanno algorithm), which is generally used in machine learning for parameter estimation.

*It is possible to separate the content representations and style representation in a CNN, learnt during a computer vision task (e.g. image manipulation or recognition task).* NST uses a pre-trained convolution neural network (CNN) to transfer styles from a given image to another. This is done by defining a loss function that tries to minimise the differences between a style image and a content image, by blending them both seamlessly together to create a generated image. **[1]**

NST defines the following inputs:

* A generated image — the image that contains the final output which is **the only trainable variable**
* A style image — the image from which we want to transfer the style that is to be incorporated in the content image
* A content image — the image on which we want to perform the style transfer **[2]**

**Introduction:**

Two networks namely, a pre-trained feature extractor and a transfer network are required to train a style transfer model (where to avoid the usage of paired training data, the pre-trained feature extractor is used). The tendency for the individual layers of CNN trained for image classification can be used to specialize in understanding the specific features of an image. **[1]**

The layers can learn to extract the content and texture of an image. Style transfer runs the two images through a pre-trained NN where it compares the similarity of the pre-trained network’s output at multiple different layers. Images producing similar output at one of the pre-trained model’s layers most likely has similar content while having a similar style with matching output at another layer. **[1]**

The transfer network has an encoder-decoder architecture which helps us to create the stylized image since the pre-trained model only which lets us compare the content and style of both the images. **[1]**

At the beginning of the training, one or more style images are made with a pre-trained feature keyboard, and the results of various style layers are retained for later comparison. Content images are then included in the program. Each content image goes through a pre-trained feature extractor, where different layers of content are extracted. The image content then passes through the transfer network, which renders the styled image. The styled image is also processed with a feature button, and the effects on both content and style layers are saved. **[2]**

A custom loss function with our goals for both the content and style, defines the quality of the stylized image. The output elements of the stylized image are compared to the original image. The output style features are compared to the style features present in the reference style images. After each step, only the transmission network is updated. The weights of the pre-trained feature trailer remain constant throughout. By measuring the different terms of the lost work, we can train the models to produce images by producing a simple or complex style. **[2]**

**Content and Style representations**

In order to find both the content and style of our image, we will look at some of the inner layers within our model. The middle layers represent the feature maps that start to get more organized as you go deeper. In this project, the construction of the VGG19 network, which is a pre-trained image editing network, is being used. These middle layers are needed to define content and style representation from our images. We will try to match the corresponding style styles and content to these intermediate layers with the installation image.

We will download our pre-trained photo editing network. After that, we hold the layers of interest. We then describe the Model by setting the model input into the image and the effects of the style and content output. In other words, we have created a model that will take the image of the insert and extract the content and style of the middle layers. **[3]**

**Intermediate layers**

These are central effects within our pre-trained image editing network that enables us to define the style as well as content representation. In order for a network to perform image classification, it must understand the image. This includes taking a green image as pixels to insert and create an internal representation with the conversion that converts the green pixels into a complex understanding of the elements present in the image. This is also partly why CNNs can generalize well: they are able to capture abnormalities and explain features present in classes that are agnostic in background noise and other issues. Therefore, somewhere between where the green image is inserted and the separation label is removed, the model acts as a complex feature output; which is why by finding the middle layers, we are able to define the content and style of input images. **[3]**

**Model: VGG network features and implementation**

VGG19 network was developed during The ImageNet Large Scale Visual Recognition Challenge (ILSVRC)-2014, for image classification, with an error rate of 7.3%. VGG-19 is a 19-layer deep convolutional neural network. We can download a pre-trained network version of over a million images from the ImageNet database. A pre-trained network can classify images into 1000 objects, such as a keyboard, mouse, pencil, and many more animals. As a result, the network has read rich presentations of various images. The network has 224-by-224 image input size. We can use the split to separate new images using the VGG-19 network. **[3]**

As suggested in the paper by Gatys et al., I’ve used VGG19 network for this project. VGG19 lets us extract the feature maps and the style and content representations of the style, content, and the generated images. VGG19 is a relatively simple model of CNN as compared to Inception and ResNet. VGG19’s feature maps work better for neural artistic style transfer. VGG19 has 5 stacks of convolutional layers- conv1 to conv5, each possessing 2 to 4 layers called conv1\_2…conv5\_4, which are in turn followed by fully connected layers that are actually performing the classification. The convolutional layers perform feature extraction by finding various geometrical shapes and patterns of progressing complexity while the fully connected layers act as a perceptron and classifies objects based on the shapes that are present in the image. Hence, a pre-trained VGG network with detached completely convolutional layers will act as a pure pattern and feature extractor. **[1]**

***VGG network features and implementation***

**Content Loss**

To use the VGG19 network for content loss or perceptual loss, in applications such as single-resolution single-release devices from various components of VGG19. The deeper the layer, more the network focuses on generalized information and patterns.

We will transfer the network to both the desired content image and our basic input image. This will return the effect of the middle layer (from the layers described above) from our model. After that we just take the Euclidean distance between the two symbols between these images.

Loss of content is a function that describes the range of content from our installation image and our content image. We do a back distribution in the normal way to reduce this content loss. We thus modify the original image until it produces the same response in a certain layer as the original image content.

With the loss of content, we can create an almost complete picture with the first layers of convolution (layers 1-3), the quality begins to gradually decrease to layers 4 and 5, as the network begins to capture only more detail. **[4]**

**Style Loss:**

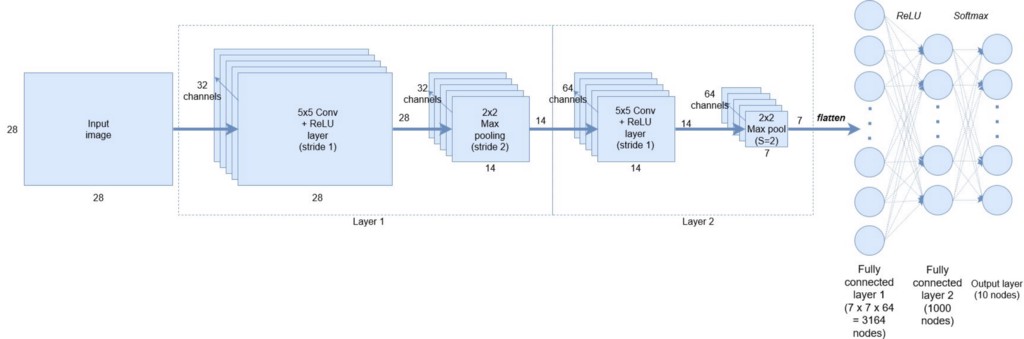
We follow the same principle while computing the style loss but it gets a bit more complex this time for, we feed out network a base input and style image altogether compare the Gram Matrices of the two outputs instead of comparing the Style or Gram Matrices of 2 outputs.

Gradient descent is performed on content image to change it into an image that’s going to match with the style of the original image and generate a style for our base input image by minimizing mean squared distance between feature correlation map of input image and style image.

We can use the VGG network-based Gram Matrix to extract texture info from the image where we’ll need to perform an inner dot product on it to get a style of the image. We can vectorize the VGG later content followed by a multiplication which can be used to get the style loss directly for training.

Style loss based on initial layers keeps spatial awareness but starts hallucinating heavily with deeper layers where layers 3 till 5 are typically used for style transfer applications. **[4]**

***What Convolutional Neural Network Captures?***



At the 1st Layer, using 32 filters, the convoluted neural network (CNN) can capture simple patterns, for example, a horizontal line or a straight line, which might not make any sense to us but is of great importance to the CNN.

Slowly upon moving down to the 2nd Layer, having 64 filters, CNN starts capturing more and more complex features that it couldn’t by using its previous layer. It might be the face of a cat or dog or human or an inanimate object. Feature Representation is the process of capturing of different simple and complex features.

The important thing to note here is that CNNs in spite of not knowing what a particular image is can learn to encode what that image is representing. Convolutional Neural Networks (CNNs)’s encoding nature helps us in Neural Artistic Style Transfer. **[6]**

**How to capture Style and Content of images using Convolutional Neural Networks (CNNs)?**

VGG19 network, a 19 layers deep CNN, is widely used for Neural Artistic Style transfer and has been trained on a million images (approximately) from ImageNet database, due to which it has the ability to detect high-level features in an image. The ‘encoding nature’ of CNNs is the key in Neural Artistic Style Transfer.

Our first step will be the initialization of a noisy image, which is later going to be our generated or output image (G). Second step is the calculation of the similarity of this image to the style and the content image at any particular given layer of our VGG19 network.

We calculate the loss of output/generated image (G) with respect to the content(C) and style(S) image because we want our output/generated image (G) to have the content and style of the content image(C) and style image(S) respectively. **[6]**

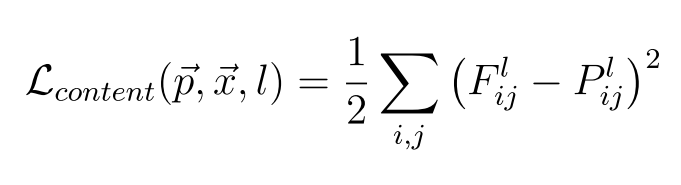
**Proposed Method:**

**Mathematically defining the Style Loss and Content Loss of a noisy image that has been generated randomly**

**Content Loss**

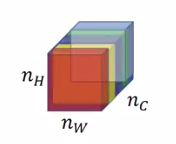
How similar is the noisy image which was randomly generated to the content image? We find this out by calculating content loss using the below proposed method.

Supposedly, a hidden layer (L) that is present in a pre-trained CNN network (namely, the VGG network) is chosen to compute the loss. Assuming, P to be the original image and F to be the generated image and F[I] and P[I] both be the feature representation of those images in. The content loss of the layer L can be defined as given below:



**Content Cost Function:** The **content cost function** ensures that the **content** present in the **content** image can be seen in the generated image. L {**content**} captures the root mean squared error between the activations produced by the generated image and the **content** image.

**How to capture an image’s style**

******

***Different Feature maps in layer L***

Different channels/feature maps/filters at a particular chosen layer I can be seen in the above image. We, then, calculate how correlated these filters are to each other to capture the style of an image

**Example:** Assuming the first two channels in the image shown above to be red and yellow, if red captures some feature supposedly and if both of these 2 channels were correlated, a yellowish effect of the second channel can be observed whenever the red channel detects vertical lines in the image

**Calculating these correlations mathematically**.

We are going to calculate the dot product of the vectors of activations of two filters to calculate a correlation between two filters/channels, the matrix out of which when received is called the Gram Matrix

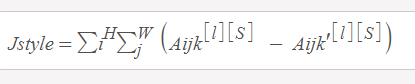
**Style loss is computed using the Gram matrix**: The distribution of features of a set of feature maps in a given layer is captured by the gram matrix where we match the distribution of features of two images by attempting to minimise the style loss between the two images. **[1] [2]**

**Estimating whether the 2 images are correlated or not?**

1. The channels are correlated if the dot product between the activation of two filters is large.
2. The channels are un-correlated if the dot product between the activation of two filters is small.

**Mathematically:**

**Gram Matrix of Style Image(S):** k and k’ are the variables representing the different filters of the layer L. We can assume it to be Gkk’ [l][S].



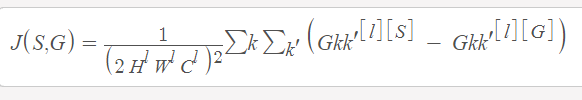
**Gram Matrix for style Image**

**Gram Matrix for Generated Image(G):** The variables k and k’ represent different filters or channels of the layer L. We can assume it to be Gkk’ [l][G].

https://hackernoon.com/hn-images/1*x2OamSGwaMIwYXSEWiDsBw.png

**Gram Matrix for generated Image**

**Style loss:** The cost function between the Generated Image and the Style Image is the square of the difference between the Style Matrix or Gram Matrix of the generated Image and the Style Matrix or Gram Matrix of the style image.

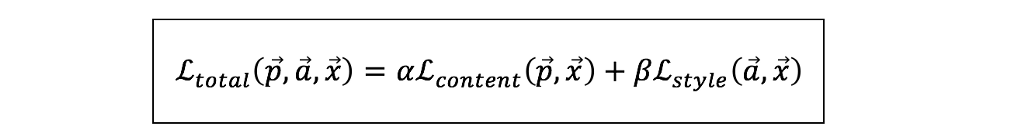


**Style cost Function**

**Style loss Function:** All layers of the CNN are used to extract the style information from VGG network. Style Information can be measured as the amount of correlation between feature maps in a particular layer. The difference of correlation present between the feature maps computed using the generated and style image is termed as loss. We try to compute a style matrix for the generated and style images. The root mean square difference between the two style matrices is termed as style loss.

**Total Loss Function:**

It’s is the sum of the cost of the style image and the content image which can be mathematically expressed as follows:



**Total Loss Function for Neural Style Transfer**

We can weigh content and style cost using alpha and beta respectively from the above equations which define the weightage of both costs in the output image generated. These are user-defined hyper-parameters which can control the amount of content and style to be put in the input image from the style and content images. The loss can be minimized using back-propagation that will optimize the random image into a meaningful art piece, given that we calculate the loss.

The Neural Style algorithms which were developed by Leon A. Gatys, Matthias Bethge, and Alexander S. Ecker have been implemented in this project that allows us to reproduce a selected image with a new artistic style. Three images are taken (input, content, and style) and the input is changed to resemble the content and style of the content and style images are chosen respectively. **[8][9]**

**Underlying Principle**

The underlying principle is simple to understand. Two distances need to be defined namely, content (DC) and style (DS). DC measures the difference in content between the two images while DC measures the difference in their styles. A third image is then taken along with the input and transformed to minimize the content distance with content image and style distance with style image. The necessary packages can then be imported to begin the neural transfer. **[1]**

**Importing Packages and Selecting a Device**

The following packages are required to implement the neural transfer:

* torch, torch.nn, numpy (These are the most essential NN packages with the PyTorch library)
* torch.optim (efficient gradient descents)
* PIL, PIL.Image, matplotlib.pyplot (This package can be used to load and display the images)
* torchvision.transforms (transform PIL images into tensors)
* torchvision.models (train or load pre-trained models)
* copy (This is a system package which is used to deep copy the models)

**Code:**

# importing required packages

# torch and torch.nn are indispensables packages for neural networks with PyTorch

# PyTorch is based on the Torch library and its a machine learning library that is open source

# PyTorch is used for natural language processing and computer vision

# torch.optim (efficient gradient descents)

# PIL, PIL.Image and matplotlib.pyplot - used to display images

# torchvision.transforms (transforms the PIL images into tensors)

# torchvision.models (trains the pre-trained models)

# copy used to deep copy the models

# PIL- free library that adds image processing capabilities to the Python interpreter

# PIL supports a range of image file formats and offers several standard procedures for image processing/manipulation

# NumPy (a Python library) supports large, multi-dimensional arrays, matrices and a large collection of high-level mathematical functions

import numpy as np

from PIL import Image

from \_\_future\_\_ import print\_function

import matplotlib.pyplot as plt

import torch

import torch.nn as nn

import torch.nn.functional as F

import torch.optim as optim

import torchvision.models as models

import torchvision.transforms as transforms

import copy

# choosing which device to run the network on and importing the content and style images

# using torch.cuda.is\_available() to detect if there is a GPU available

# setting the torch.device for use

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

# import the style and content images

# original PIL images contain values btw 0 - 255

# PIL images values are converted btw 0 and 1 when they are transformed into torch tensors

# resizing of both style and content images so that they both have same dimensions

# desired size of the output image

imsize = 512 if torch.cuda.is\_available() else 128

loader = transforms.Compose([

    transforms.Resize(imsize),  # scaling the imported image

    transforms.ToTensor()])  # transforming it into torch tensor

# importing the style and the content images

# neural networks from the torch library are trained with tensor values ranging btw 0-1

# If you try to feed the networks with 0 to 255 tensor images, then the activated feature maps will be unable to sense the intended content and style

# Caffe library's pre-trained networks are trained with 0 to 255 tensor images

def image\_loader(image\_name):

    image = Image.open(image\_name)

    # to fit network's input dimensions, fake batch dimensions are required

    image = loader(image).unsqueeze(0)

    return image.to(device, torch.float)

style\_img = image\_loader("style.jpg")

content\_img = image\_loader("content.jpg")

import cv2

#adding them to a directory with name images in the current working directory

style\_img = image\_loader("style.jpg")

content\_img = image\_loader("content.jpg")

assert style\_img.size() == content\_img.size(), \

    "we need to import style and content images of the same size"

# try:

#   style\_img = cv2.imread("style.jpg")

#   style\_img = cv2.resize(style\_img, dsize=(500, 500))

#   content\_img = cv2.imread("content.jpg")

#   content\_img = cv2.resize("content.jpg", dsize=(500, 500), interpolation=cv2.INTER\_CUBIC)

# except Exception as e:

#   print("error")

#creating a function that will display an image by reconverting its copy to PIL format and then displaying the copy using plt.imshow

unloader = transforms.ToPILImage()  # reconverting into PIL image

plt.ion()

def imshow(tensor, title=None):

    image = tensor.cpu().clone()  # cloning the tensor to avoid making changes on it

    image = image.squeeze(0)      # removing fake batch dimensions

    image = unloader(image)

    plt.imshow(image)

    if title is not None:

        plt.title(title)

    plt.pause(0.001) # pausing a bit for the plots to get updated

#displaying the style and content images to ensure that both of them were imported correctly

plt.figure()

imshow(style\_img, title='Style Image')

plt.figure()

imshow(content\_img, title='Content Image')





***Content Loss***

Content Loss is a function representing a weighted version of the content distance for an individual layer. We are going to add the content loss module directly after the convolution layers that are being used to compute the content distance. In this way every time the convoluted neural network is fed with an input image, the content losses will get computed at the respective desired layers and because of auto grad, all the gradients will get computed. In order to make the content loss layer transparent, we are required to define a forward method that will compute the content loss and then return the layer’s input. The computed loss will be saved as a parameter of the module.

class ContentLoss(nn.Module):

    def \_\_init\_\_(self, target,):

        super(ContentLoss, self).\_\_init\_\_()

        self.target = target.detach()

# we will detach the target content from the tree used to dynamically compute the gradient

# this is a stated value and not a variable because otherwise the forward method of the criterion will throw an error

    def forward(self, input):

        self.loss = F.mse\_loss(input, self.target)

        return input

***Style Loss***

Implementation of the Style Loss module is very similar to that of the Content Loss module. It acts as a transparent layer in a network computing the style loss of that particular layer. We need to compute the gram matrix to calculate the style loss. A gram matrix is a result of multiplying a given matrix by its own transpose.

The gram matrix must be normalized by dividing each of its element by the total number of elements present in the matrix. Normalization of larger values in the Gram matrix is important else they will cause the first layers, that are present before pooling layers, to have a larger impact during the gradient descent. Style features tend to be in the deeper layers of the convoluted neural network so this step of normalization is really important.

def gram\_matrix(input):

    # a=batch size(=1)

    a,b,c,d = input.size()

    # b=number of feature maps

    # (c,d)=dimensions of a f. map (N=c\*d)

    # resise F\_XL into \hat F\_XL

    features=input.view(a\*b,c\*d)

    # compute the gram product

    G = torch.mm(features, features.t())

    # normalizing the values of the gram matrix by dividing it by the no. of element present in every feature maps

    return G.div(a\*b\*c\*d)

#style loss module looks almost exactly like the content loss module

#style distance is also computed using the mean square error between GXL (Graph eXchange Language) and GSL (GNU scientific library - for numerical computing)

class StyleLoss(nn.Module):

    def \_\_init\_\_(self, target\_feature):

        super(StyleLoss, self).\_\_init\_\_()

        self.target = gram\_matrix(target\_feature).detach()

    def forward(self, input):

        G = gram\_matrix(input)

        self.loss = F.mse\_loss(G, self.target)

        return input

**Importing the Model**

We will be importing a pre-trained neural network. I have used a **19 layer VGG network**. PyTorch’s implementation of VGG is a module that divides into 2 child Sequential modules, **Features** that contains convolution and pooling layers, and **Classifier** that contains fully connected layers. We need the output of the individual convolution layers to measure content and style loss so we will be using the features module. We must set the network to evaluate mode using **.eval()** because some layers have different behavior during training than evaluation.

cnn = models.vgg19(pretrained=True).features.to(device).eval()

# VGG networks are trained on images with every channel normalized using mean=[0.485,0.456,0.406] and std=[0.229,0.224,0.225]

# We will be using them to normalize the image before sending it into the VGG network

cnn\_normalization\_mean = torch.tensor([0.485,0.456,0.406]).to(device)

cnn\_normalization\_std = torch.tensor([0.229,0.224,0.225]).to(device)

# creating a module to normalize the input image to put it in a nn.Sequential easily

class Normalization(nn.Module):

    def \_\_init\_\_(self, mean, std):

        super(Normalization, self).\_\_init\_\_()

        # .view the mean and std to make them [Cx1x1] to make them work directly with image Tensor of shape [B x C x H x W]

        # B=batch size, C=no. of channels, H=height, W=width

        self.mean = torch.tensor(mean).view(-1, 1, 1)

        self.std = torch.tensor(std).view(-1, 1, 1)

    def forward(self, img):

        # normalizing img

        return (img - self.mean) / self.std

# an ordered list of child modules is present the Sequential module

# a sequence (Conv2d,ReLU,MaxPool2d,Conv2d,ReLU…) is present in vgg19.features which is aligned in the right order of depth

# style loss and content loss layers are to be added immediately after the convolution layer

# new Sequential module created with style loss and content loss modules inserted correctly at proper depth layers in order to compute style/content losses:

content\_layers\_default = ['conv\_4']

style\_layers\_default = ['conv\_1', 'conv\_2', 'conv\_3', 'conv\_4', 'conv\_5']

def get\_style\_model\_and\_losses(cnn, normalization\_mean, normalization\_std, style\_img, content\_img,

content\_layers = content\_layers\_default, style\_layers = style\_layers\_default):

    cnn = copy.deepcopy(cnn)

    # normalization module

    normalization = Normalization(normalization\_mean, normalization\_std).to(device)

    # having an iterable access to list of content/syle losses

    content\_losses = []

    style\_losses = []

    # assuming that Convoluted Neural Network is a nn.Sequential, we make a new nn.Sequential to add modules which are supposed to be activated sequentially

    model = nn.Sequential(normalization)

    i = 0

    # incrementing each time we encounter a conv

    for layer in cnn.children():

        if isinstance(layer, nn.Conv2d):

            i += 1

            name = 'conv\_{}'.format(i)

        elif isinstance(layer, nn.ReLU):

            name = 'relu\_{}'.format(i)

            layer = nn.ReLU(inplace=False)

        elif isinstance(layer, nn.MaxPool2d):

            name = 'pool\_{}'.format(i)

        elif isinstance(layer, nn.BatchNorm2d):

            name = 'bn\_{}'.format(i)

        else:

            raise RuntimeError('Unrecognized layer: {}'.format(layer.\_\_class\_\_.\_\_name\_\_))

        model.add\_module(name, layer)

        if name in content\_layers:

            # adding content loss:

            target = model(content\_img).detach()

            content\_loss = ContentLoss(target)

            model.add\_module("content\_loss\_{}".format(i), content\_loss)

            content\_losses.append(content\_loss)

        if name in style\_layers:

            # adding style loss:

            target\_feature = model(style\_img).detach()

            style\_loss = StyleLoss(target\_feature)

            model.add\_module("style\_loss\_{}".format(i), style\_loss)

            style\_losses.append(style\_loss)

    # triming off the layers after the last style and content losses

    for i in range(len(model) - 1, -1, -1):

        if isinstance(model[i], ContentLoss) or isinstance(model[i], StyleLoss):

            break

    model = model[:(i + 1)]

    return model, style\_losses, content\_losses

# select the respective input image

# either use white noise or a copy of the content image

input\_img = content\_img.clone()

# to use white noise instead a copy of the content image uncomment the line below:

# input\_img = torch.randn(content\_img.data.size(), device=device)

# adding the original input image to the figure:

plt.figure()

imshow(input\_img, title='Input Image')



**Gradient Descent**

I have used the L-BFGS algorithm to run gradient descent. I want to train the input image in order to minimize the content/style losses, unlike training a network. So I'm going to create a PyTorch L-BFGS optimizer optim.LBFGS and pass the input image to it as the tensor to be optimized.

**Limited-memory BFGS** L-BFGS or LM-BFGS is basically an optimization algorithm that uses a limited amount of computer memory to approximate the Broyden–Fletcher–Goldfarb–Shanno algorithm (BFGS). L-BFGS algorithm is widely known for parameter estimation in the domain of machine learning.

def get\_input\_optimizer(input\_img):

    # this line shows that input is a parameter which requires a gradient

    optimizer = optim.LBFGS([input\_img.requires\_grad\_()])

    return optimizer

# defining a function thats going to perform the neural artistic style transfer

# to compute new losses, the network is fed an updated input for each iteration

# running the backward methods of each loss module and dynamically computing their gradients

# the optimizer requires a closure function, which will re-evaluate the module and then return the loss

# the network might try to optimize the input with values that are exceeding the tensor range- 0 to 1, for the image

# address this by using input values that are btw 0 to 1 every time the network is executed

def run\_style\_transfer(cnn, normalization\_mean, normalization\_std,

                       content\_img, style\_img, input\_img, num\_steps=300,

                       style\_weight=1000000, content\_weight=1):

    """Run the style transfer."""

    print('Building the style transfer model..')

    model, style\_losses, content\_losses = get\_style\_model\_and\_losses(cnn,

        normalization\_mean, normalization\_std, style\_img, content\_img)

    optimizer = get\_input\_optimizer(input\_img)

    print('Optimizing..')

    run = [0]

    while run[0] <= num\_steps:

        def closure():

            # the values of updated input image are to be corrected

            input\_img.data.clamp\_(0, 1)

            optimizer.zero\_grad()

            model(input\_img)

            style\_score = 0

            content\_score = 0

            for sl in style\_losses:

                style\_score += sl.loss

            for cl in content\_losses:

                content\_score += cl.loss

            style\_score \*= style\_weight

            content\_score \*= content\_weight

            loss = style\_score + content\_score

            loss.backward()

            run[0] += 1

            if run[0] % 50 == 0:

                print("run {}:".format(run))

                print('Style Loss : {:4f} Content Loss: {:4f}'.format(

                    style\_score.item(), content\_score.item()))

                print()

            return style\_score + content\_score

        optimizer.step(closure)

    input\_img.data.clamp\_(0, 1)

    return input\_img

# getting the output finally by running the algorithm

output = run\_style\_transfer(cnn, cnn\_normalization\_mean, cnn\_normalization\_std,

                            content\_img, style\_img, input\_img)

plt.figure()

imshow(output, title='Output Image')

plt.ioff()

plt.show()

***OUTPUT:***

Building the style transfer model..

Optimizing..

run [50]:

Style Loss : 46.107723 Content Loss: 21.784138

run [100]:

Style Loss : 16.584763 Content Loss: 19.779057

run [150]:

Style Loss : 10.294066 Content Loss: 17.930664

run [200]:

Style Loss : 7.016173 Content Loss: 16.808113

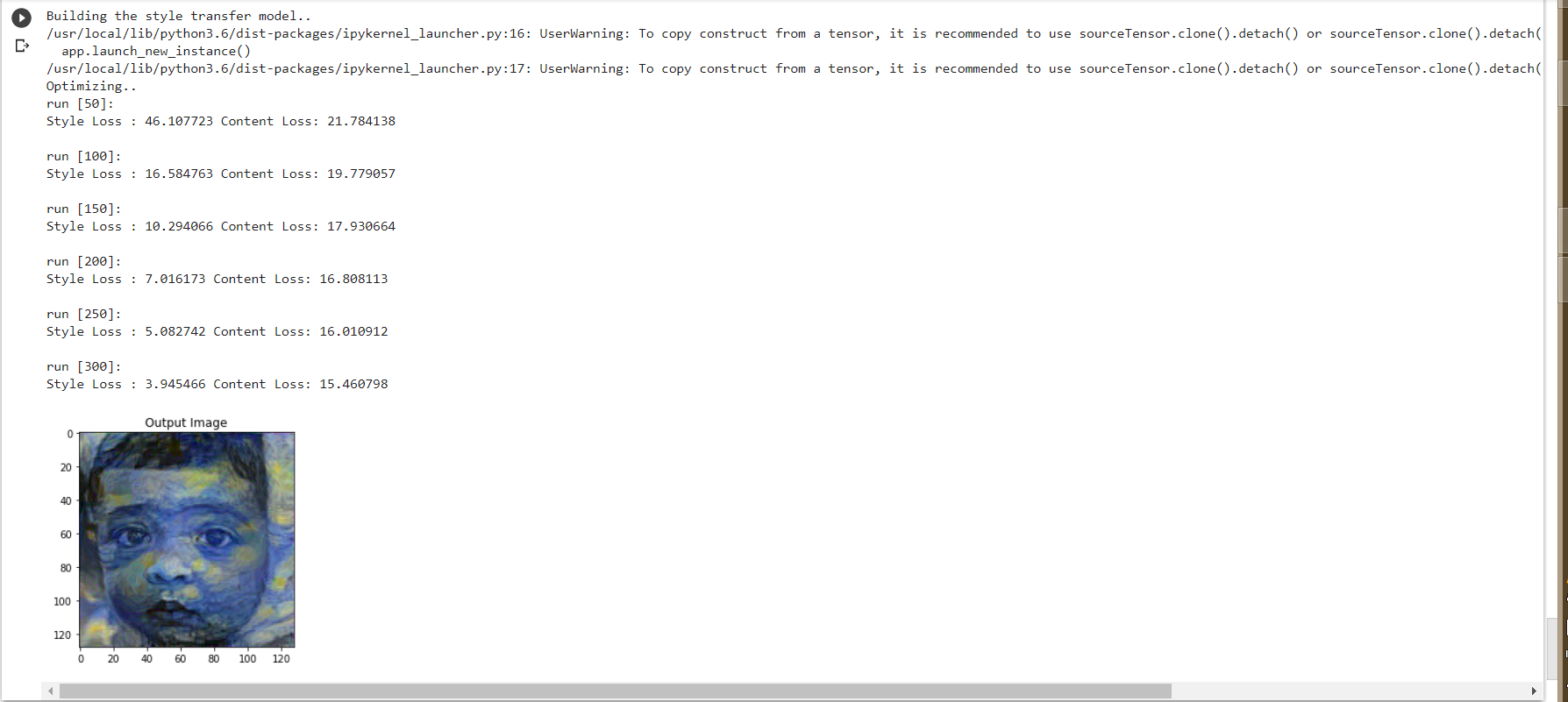
run [250]:

Style Loss : 5.082742 Content Loss: 16.010912

run [300]:

Style Loss : 3.945466 Content Loss: 15.460798





**Results and Discussion:**

**Algorithm & Architecture Discussion:**   
According to the paper published by **Gatys et al** in 2015, it is possible to separate the Style and the Content of an image, thereby, making it possible to combine the style and the content of different images. Gatys made use of a 19 layers deep Convolutional Neural Network (CNN), called VGG-19 (VGG - Visual Geometric Group), with 16 CONV layers and 3 FC layers.  
Stanford University’s Standford Vision Lab pre-trained VGG-19 on the ImageNet dataset. Gatys performed average pooling without the use of any FC layers.  
Pooling reduces the spatial volume of feature vectors, thereby, reducing the number of computations. **[1]**

**Losses in Style Transfer:**

* **Content Loss**

Content Loss function helps the generated images to incorporate the contents of the content image. A deeper layer (*L*)’s output is selected when the content and the generated image are fed to the Convolution Neural Network and the activations of the different layers are computed. That output is then used to find the error between the generated and content image.

The Content Loss function is basically the Euclidean distance between these two intermediate layers of the Convolution Neural Network fed with the generated image (*x*) and the content image (*p*).

The Content Loss function helps us to find out how much different the deeper layers of the generated image and the content image are from one another.

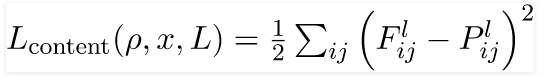
***Calculating the content loss after the selection of a hidden layer (L) in the VGG-19 network:***

**p** represents the original image

**x** represents the generated image

**Pl**and **Fl** depicts the feature representations of the images corresponding to layer L

Content Loss is defined as:



This represents the mean squared error between the activations of the layer *l*of the Convolution Neural Network (CNN), passed with both the generated and the content image.

Instead of multiple layers, we are applying the content loss at only one layer (*L*).

**Need to minimize the content loss**

The Convolutional Neural Networks’ deeper layers, capture features that are even more complex, therefore, we need the generated image (*x*) to have the same content as the content image. After that, we need to minimize the difference between the deeper layers in the feature representation of these two images because the high similarity of features of the two images in the deeper layers of the CNN, implies that both the images have the same content.

* **Style Loss**

The procedure to obtain the style loss is similar to the procedure for the calculation of the content loss with an exception of using the intermediate layer’s raw output. Instead, we will be using the gram matrix of the generated feature maps of the individual layers and then apply the loss to all the layers.

We need to calculate the **Gram Matrix** first for this. The calculation of the dot product of vectorized feature maps, i and j at layer l is involved in the calculation of correlation between different channels or filters. The matrix obtained is called the Style Matrix or the Gram Matrix (G).

Style Loss is defines as the square of the difference between the Gram Matrix of the generated Image with the Gram Matrix of the style image.



Style Matrix or Gram Matrix is simply a matrix who’s (*i,j*)th element has the output of an element-wise multiplication of the*i* th and *j* th feature maps, which is then summed across the height and the width of the image.

**Total style loss has been calculated in two steps:**

1. At any given layer (l), the mean squared error is calculated between the style matrix or the gram matrix of the feature map representation of the generated image (x) and style (p), representing the style loss of that particular layer.

2. The style loss is then applied to all the layers present in the CNN and multiplied with an additional weight i.e. the weight factor of each layer contributing towards the computation of the total loss.

**Need to minimize the Style Loss**

The style loss at any given layer (*l*), is calculated by the difference between the correlation of the Style Matrix or Gram Matrix obtained from the feature maps in that particular layer (*l*) of the style image (*p*) and the generated image (*x*). G (*l*) is the Gram Matrix representing a specific type of correlation between the feature maps of the layer (*l*). The correlation between the feature maps i and j, is represented by G*l* (*i,j*).

Therefore, it is crucial to minimize the distance between the feature representation of the generated image (*x*) and style (p). The generated image would then possess features similar to those of the style image. The generated image is the only trainable image, so the generated image starts incorporating the features of the style image in itself.

Style loss will enable the generated image to possess even the smallest of the features of the style image which are identified in the very initial layers of the network because it is applied at all the feature map representations of all the layers.

* **Total Loss**  
  is defined by the formula given below (where α and β are hyper-parameters that are set as per the requirements).



The generated image X is such that the style loss and content loss is the least. Therefore, X matches both the content of P and the style of A, thus, generating the desired output.

***α****(alpha)*and ***β****(beta)*are hyper-parameters and can be tweaked and set according to the preference or requirement of the user. Upon increasing the value of any of the hyper-parameters, the contribution of the loss associated with it in the generated image increases.

**Optimizing the loss**

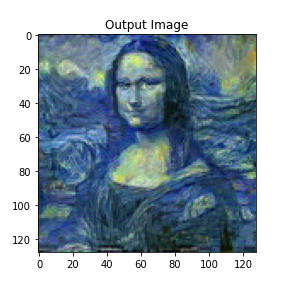
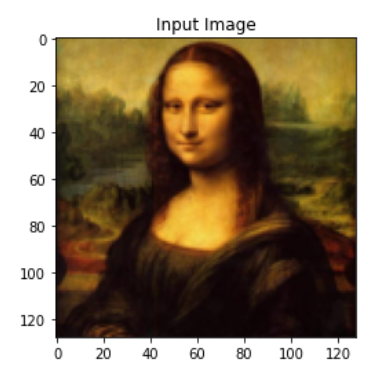
After defining the final loss function, we will finally minimize the loss by using the optimization algorithm. We will be using the L-BFGS optimizer because for the given task at hand (Neural Artistic Style Transfer), the L-BFGS optimizer is faster in learning in comparison to the Adam Optimizer. Using stochastic gradient descent or the Adam Optimizer is not recommended because the data used here is not Stochastic i.e. dividing the dataset into smaller mini-batches won’t work because here we are giving a single static image as the input. **[5][7][8]**

**Style Image**



**Content Image**





**Conclusion:**

Neural Style Transfer allows us to blend two images, Content image and Style Image together to create new piece of art.

* Successfully understood the need of neural style transfer and gave an overview of the architecture of the methodology followed.
* Defined the specifics of the neural style transfer network using TensorFlow.
* Defined several functions that were used to define the variables/inputs, compute the VGG19 output, compute the losses and perform the optimisation.
* Understood the concept of content loss and the style loss in detail, and saw how they defined the final loss together.
* During the course of this project, I learnt how to build several different loss functions and then how to use back propagation to minimize the losses and transform the input images.
* Loaded a pre-trained CNN model – VGG19 and used its feature maps to describe the style and content representation of the images.
* The loss functions defined in the code mainly compute the distance in terms of style and content representations of the images.
* Learnt how to work dynamically with tensors and manipulate them directly, thus making debugging and working with tensors considerably easier by using a natural python control flow.
* Finally ran the custom model after the successful implementation of the above concepts and generated the output artwork by the model.
* Using the command tf.gradient and by applying optimizers update rules, iteratively updated the image.
* The L-BFGS algorithm based optimizer minimized the given losses with respect to the input image. **[1]**

**Future Implementation: Real-world use cases and applications for style transfer:**

**Photo and video editors**

Neural Artistic Style Transfer finds its one of the most in-demand application in various video and photo editing software. From sharing stylized self-portraits to augmenting user-generated movies and music videos, and beyond, the ability to incorporate famous artistic styles to images and video clips holds a whole new level of appeal to the user-base and adds unprecedented power to creativity tools based on NST.

NST’s flexibility and performance regarding current deep learning approaches, artistic style transfer models can be incorporated in edge devices quite easily—for example, mobile phones—allowing for various applications to transform images and video, after processing, in real-time. Thus enabling professional-quality photo and video editing tools to become easier to use and even more widely accessible.

Given below is a list of a few tools that employ neural artistic style transfer as part of their toolkits:

* Instapainting’s AI Painter
* Painter’s Lens
* Arbitrary Style Transfer in the Browser
* Video Star
* Looq

**Commercial art**

Neural Artistic Style Transfer is one of the few Computer Vision techniques that makes AI-powered artwork possible.Whether it’s up-coming artists paving their way to share their aesthetic artwork with the entire world or an artwork sold at a high-end auction, Neural Artistic Style Transfer promises to change what originality means, the ways we think about art and how we represent art in real-time..

Neural Artistic Style Transfer can also be used to create high-quality and reproducible prints for large-scale advertising campaigns, or for office buildings. Neural Artistic Style Transfer could considerably change our perception regarding art’s commercial impacts.

**Neural Artistic Style Transfer on mobile**

The applications of using Neural Artistic Style Transfer aren’t only limited to the ones running in cloud or on servers. We can make Neural Artistic Style Transfer models that are small and fast enough to run directly on various platforms, thus opening up a range of possibilities, including creativity tools, powerful video and image editors, etc.

From user engagement and retention to brand loyalty, and beyond, implementing Neural Artistic Style Transfer on-device has the potential to appeal users in various innovative and lasting ways, while reducing cloud costs and maintaining user data privacy.

**Gaming**

Back in 2019, during a press conference at the Game Developers Conference’19, Google introduced Stadia, its very own cloud-powered video game streaming service. One of Stadia’s main features is its in-game style transfer features that enables automatic re-composition of the virtual world with colour palettes and textures incorporated from a potentially limitless range of artistic styles.

According to Google, efforts like these intends to empower the artist present inside every developer. Therefore, gaming, much like any other applications of neural artistic style transfer, will make artistic creation accessible to those who aren’t traditionally seen as artists.

**Virtual reality**

Virtual Reality, is a lot like gaming. Immersive VRs are a representation of the anchor of the user experience, and also explores the various of possibilities of neural artistic style transfer.

The various applications of neural artistic style transfer in Virtual Reality, still remains largely in the research phase with promising and exciting possibilities. Facebook has been actively testing the potential of neural artistic style transfer to radically alter the ways Virtual Reality developers depict visual stories via their films, applications, games, etc. There are some early demos supporting the augmentation of the immersiveness of worlds created for VR using Neural Artistic Style Transfer. **[6][7][8]**

**References:**

**[[1508.06576] A Neural Algorithm of Artistic Style [1]](https://arxiv.org/abs/1508.06576" \t "_blank)**

**[[1705.04058] Neural Style Transfer: A Review [2]](https://arxiv.org/abs/1705.04058" \t "_blank)**

**[[1703.07511] Deep Photo Style Transfer [3]](https://arxiv.org/abs/1703.07511" \t "_blank)**

**[[1611.07865] Controlling Perceptual Factors in Neural Style Transfer [4]](https://arxiv.org/abs/1611.07865" \t "_blank)**

**[[1607.08022] Instance Normalization: The Missing Ingredient for Fast Stylization [5]](https://arxiv.org/abs/1607.08022" \t "_blank)**

[**[1603.08155] Perceptual Losses for Real-Time Style Transfer and Super-Resolution**](https://arxiv.org/abs/1603.08155) **[6]**

[**[1703.06868] Arbitrary Style Transfer in Real-time with Adaptive Instance Normalization**](https://arxiv.org/abs/1703.06868) **[7]**

[**[1705.06830] Exploring the structure of a real-time, arbitrary neural artistic stylization network**](https://arxiv.org/abs/1705.06830) **[8]**

[**[1610.07629] A Learned Representation For Artistic Style**](https://arxiv.org/abs/1610.07629) **[9]**

[**[1606.05897] Preserving Color in Neural Artistic Style Transfer**](https://arxiv.org/abs/1606.05897) **[10]**