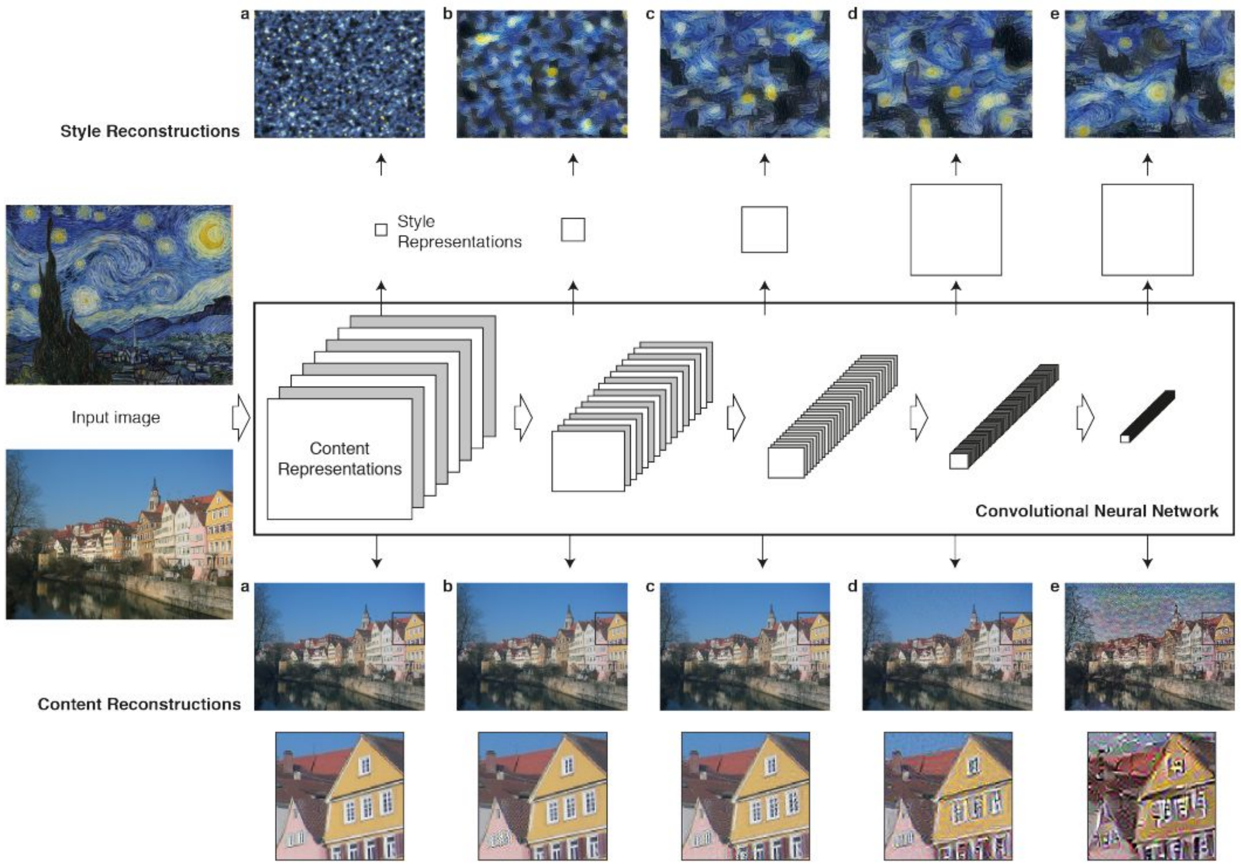
# Neural Style Transfer using Pytorch

Neural style transfer is an artificial system based on Deep Neural Network to generate an artistic image. This approach uses two arbitrary images one is called the content and the other is the style image and it extracts the structural features from the content image whereas the texture features from the style image.

The convolutional neural network develops representations of the image along the processing hierarchy. As we move deeper into the network the representations will care more about the structural features or the actual content as compared to the detailed pixel data. To obtain these representations, we can reconstruct the images using the feature maps of that layer. Reconstruction from the lower layer will reproduce the exact image whereas the reconstruction from the higher layer will capture the high-level content and hence we refer to the feature responses from the higher layer as the **content representation.**

To extract the representation of the style content, we build a feature space on the top of the filter responses in each layer of the network. It consists of the correlations between the different filter responses over the spatial extent of the feature maps. The filter correlation of different layers captures the texture information of the input image.



The above figure shows the model architecture to implement the neural style transfer. Here we use a pre-trained VGG19 network's convolutional neural network. As we can see here we are performing the content and style reconstructions. The image reconstruction from the content image shows that the reconstruction of lower layers exactly reproduces the exact pixels of the input image, whereas the higher layer captures the structural information. Reconstructions from the style features produce texturized versions of the input image that capture its general appearance in terms of color and localized structures.

So by entangling the structural information from the content representation and the texture/style information from the style representation, we generate the artistic image. We can emphasize either on reconstructing the style or the content. A strong emphasis on style will result in images that match the appearance of the artwork, effectively giving a texturized version of it, but hardly show any of the photograph's content. When placing a strong emphasis on content, one can clearly identify the photograph, but the style of the painting is not as well-matched. We perform the gradient descent on the generated image to find another image that matches the feature responses of the original image.

The loss function used for neural style transfer is defined as :

$$\mathcal{L}_{total}(\vec{p}, \vec{a}, \vec{x}) = \alpha \mathcal{L}_{content}(\vec{p}, \vec{x}) + \beta \mathcal{L}_{style}(\vec{a}, \vec{x})$$

Where p is the content image, x is the generated image and a is the style image.

The total loss consists of content loss and style loss with the weighting factors alpha and beta respectively. The coefficient of alpha and beta controls the emphasis of the style or the content reconstruction.

**Content Loss:**

$$\mathcal{L}_{content}(\vec{p}, \vec{x}, l) = \frac{1}{2} \sum_{i,j} \left(F^l_{ij} - P^l_{ij}\right)^2$$

We define the content loss as the squared error loss between the two feature representation. Let p and x be the original image and the image that is generated and Pl and Fl their respective feature representation in layer l.

**Style Loss:**
We build a feature space over each layer of the network that represents the correlation between the different filter responses. These feature correlations are calculated by the Gram matrix.
A gram matrix is an inner product between the vectorized feature map. It is basically an nxn matrix where n is the number of feature responses.
The correlation between two feature responses is calculated by taking the sum of all the elements of the element-wise multiplication of the two feature responses.

$$G_{ij}^{l} = \sum_{k} F_{ik}^{l} F_{jk}^{l}$$

To calculate the style loss, the mean-squared distance is calculated between the entries of the Gram matrix from the original image and the Gram matrix of the image to be generated.

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} \left(G_{ij}^{l} - A_{ij}^{l}\right)^2$$

Where ¼*Nl**2*Ml**2 is the constant.

The total cost is defined as :

$$\mathcal{L}_{style}(\vec{a}, \vec{x}) = \sum_{l=0}^{L} w_l E_l$$

Where wl is the weight factor of the contribution of each layer to the total loss.

**Implementing the Neural Style transfer using PyTorch**

Let's begin with importing the required libraries.

```
import torch
import torchvision.transforms as transforms
from PIL import Image
import torch.nn as nn
import torchvision.models as models
import torch.optim as optim
from torchvision.utils import save_image
import matplotlib.pyplot as plt
```

Using the vgg19 model from the torchvision.models and use the pre-trained weights. The vgg19 model contains three components:

1. Feature: containing all the convolution, Relu, and max pool layers.
2. Avgpool: containing the avgpool layer

3. Classifier: containing the dense layers (Fully Connected part of the model).

Assigning the GPU to the device variable.

```
model=models.vgg19(pretrained=True).features
device=torch.device("cuda")
```

Defining a class to extract the features from the selected layers of the loaded model.

```
class VGG(nn.Module):
    def __init__(self):
        super(VGG,self).__init__()
        self.req_features= ['0','5','10','19','28']
        #Since we need only the 5 layers in the model so we will be
dropping all the rest layers from the features of the model
        self.model=models.vgg19(pretrained=True).features[:29] #model will
contain the first 29 layers


    #x holds the input tensor(image) that will be feeded to each layer
    def forward(self,x):
        #initialize an array that wil hold the activations from the chosen
layers
        features=[]
        #Iterate over all the layers of the mode
        for layer_num,layer in enumerate(self.model):
            #activation of the layer will stored in x
            x=layer(x)
            #appending the activation of the selected layers and return
the feature array
            if (str(layer_num) in self.req_features):
                features.append(x)

        return features
```

Defining the image transformation steps to be performed before feeding them to the model. The preprocessing steps involve resizing the image and then converting it to a tensor.

```
loader=transforms.Compose([transforms.Resize((512,512)),transforms.ToTenso
r()])
```

Using a function to do the preprocessing, so let's define the function for preprocessing the image to make it suitable for the model's input layer. The function takes the path of the image as the parameter, then we load the image using the Image.open() method of the PIL and add an extra dimension for the batch size and add the tensor to the GPU and return it.

```
def image_loader(path):
    image=Image.open(path)
    image=loader(image).unsqueeze(0)
    return image.to(device,torch.float)
```

Load the images using the above-defined methods. Here we will be using the clone of the content image as the initial generated image.

```
#Loading the original and the style image
original_image=image_loader('Content image.jpg')
style_image=image_loader('style image.jpg')


#Creating the generated image from the original image
generated_image=original_image.clone().requires_grad_(True)
```

Initialize the model variable and load the model to the GPU. In the next step, initialize the epochs, learning rate, alpha value, beta value, and the optimizer.

```
#Load the model to the GPU
model=VGG().to(device).eval()
#initialize the paramerters required for fitting the model
epoch=1000
```

```
lr=0.001
alpha=1
beta=0.01
#using adam optimizer and it will update the generated image not the model
parameter
optimizer=optim.Adam([generated_image],lr=lr)
```

Here we are passing the generated image to the optimizer as in this approach we will update the pixel value of the generated image instead of the model parameters.

Let’s start iterating over the range of a number of the epoch using the for loop. The initial step is to extract the features of the of generated, content and the original required for calculating the loss.

```
for e in range (epoch):
    gen_features=model(generated_image)
    orig_feautes=model(original_image)
    style_featues=model(style_image)
```

Now we will use the above-calculated features to calculate the loss and perform gradient descent on the pixel values of the generated image. Initially, we will initialize the content loss and the style loss with 0. Then, we iterate over the various feature representations extracted in the above step and get the shape of the feature from the generated image.

```
    content_loss=style_loss=0


    for gen,cont,style in zip(gen_features,orig_feautes,style_featues):
        #extracting the dimensions from the generated image
        batch_size,channel,height,width=gen.shape
```

Calculate the content loss of each feature representation and add it to the total content loss (content_loss). To calculate the style loss we need to find the gram matrix. Then we will calculate the squared error between the gram matrix of the style image and the gram matrix of the generated image and append the loss of each feature representation to the total style loss.

```
        content_loss+=torch.mean((gen-cont)**2)

        #Calculating the gram matrix for the style and the generated image

G=torch.mm(gen.view(channel,height*width),gen.view(channel,height*width).t
())

A=torch.mm(style.view(channel,height*width),style.view(channel,height*widt
h).t())

               style_loss+=torch.mean((G-A)**2)
```

With the content and the style loss, we will calculate the total loss. Then we will initialize the gradients to zero then backpropagate the loss calculate and update the parameters based on the current gradient.

```
    #calculating the total loss of e th epoch
    total_loss=alpha*content_loss + beta*style_loss
    optimizer.zero_grad()
    total_loss.backward()
    optimizer.step()
```

To keep track of the training we will print the loss after every 100 iterations/epochs and save the generated image.

Start the training and better go for a coffee….