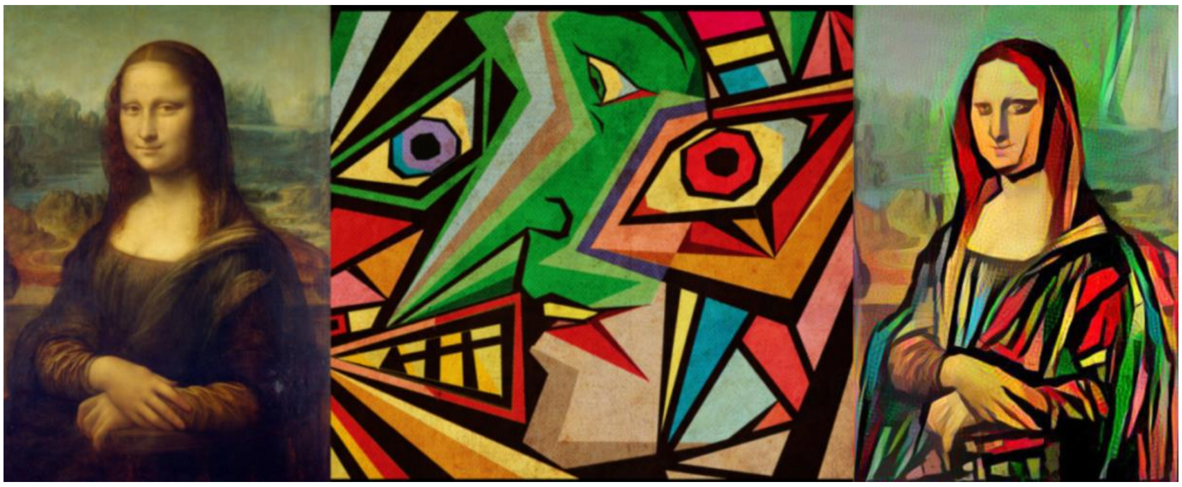
ARIES Open Project 2024

Neural Style Transfer

Neural Style Transfer (NST) is a sophisticated deep learning technique that allows the blending of content and style from different images to create unique artworks. The key idea is to use a neural network to reinterpret a content image (like a photograph) in the style of another image (like a painting). This project focused on building an accessible and interactive web application using Streamlit, which would allow users to apply NST without needing to understand the underlying technical complexities.

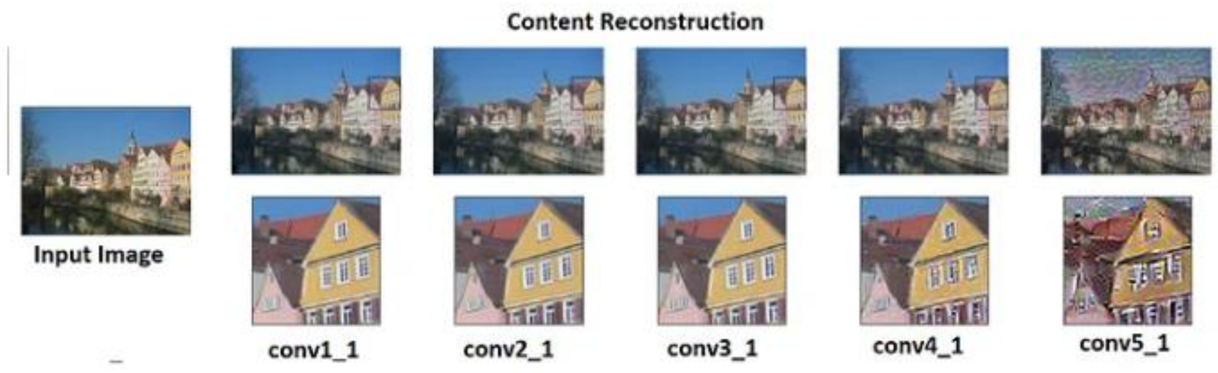
Streamlit was chosen because it is a Python library that simplifies the creation of custom web applications for data science and machine learning. It enables the rapid deployment of interactive applications, which is ideal for showcasing and utilizing complex algorithms like NST in a user-friendly manner.



Content and Style Representation

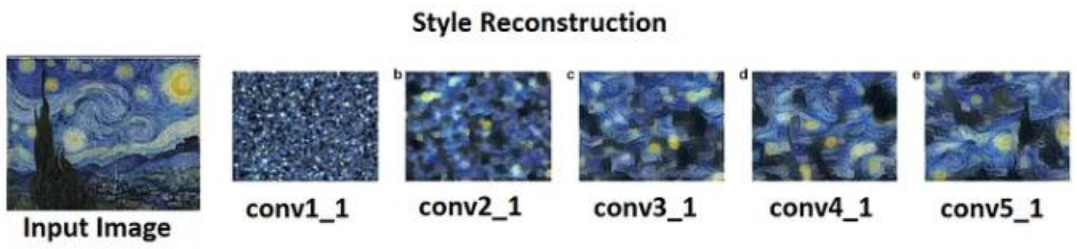
Content Representation

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 than 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. In contrast, the higher layer's reconstruction will capture the high-level content and hence we refer to the feature responses from the higher layer as the **content representation.**



Style Representation

To extract the representation of the style content, we build a feature space on the top of the filter responses in each network layer. 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. This creates images that match a given image's style on an increasing scale while discarding information of the global arrangement. This multi-scale representation is called style representation.

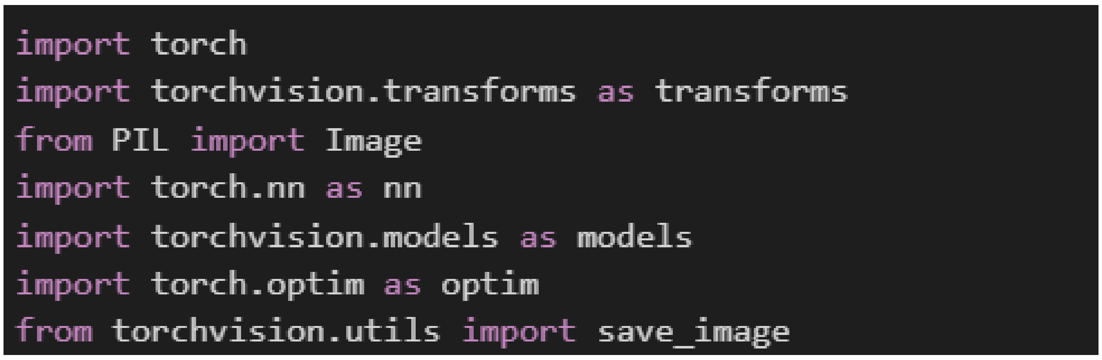


The Style and content images are merged into a third target image. Basically, we try to bring target image to content image as close as possible while maintaining the style of the style image.

We utilize a pre-trained VGG19 convolutional neural network to create artistic images by reconstructing both content and style. The network extracts structural information from a content image (e.g., a photograph) and texture/style information from a style image (e.g., a painting). By combining these features, we generate a new image that can prioritize either content or style. Emphasizing style results in an image that closely matches the artwork's appearance, giving a texturized version but with minimal recognizable content from the original photograph. Conversely, emphasizing content allows the photograph's details to be more identifiable, though the painting style is less prominent. Using gradient descent, we iteratively adjust the generated image to align with the feature responses of both the original content and style images, balancing the two aspects to achieve the desired artistic effect.

Implementation

1. Importing Necessary Libraries



```python
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
```

2. Calculation of necessary Losses:

Content Loss:

The content image and the input base image are passed to our model and the intermediate layers' outputs (listed above as 'conv1_1', 'conv2_1', 'conv3_1','conv4_1' and 'conv5_1') are extracted using the above-defined class. Then we calculate the

Euclidean distance between the intermediate representation of the content image and the input base image. Hence the content loss for a layer l is defined by:

$$\mathcal{L}_{content}(\vec{p}, \vec{x}, l) = \frac{1}{2} \sum_{i.i} \left(F_{ij}^{l} - P_{ij}^{l}\right)^2$$

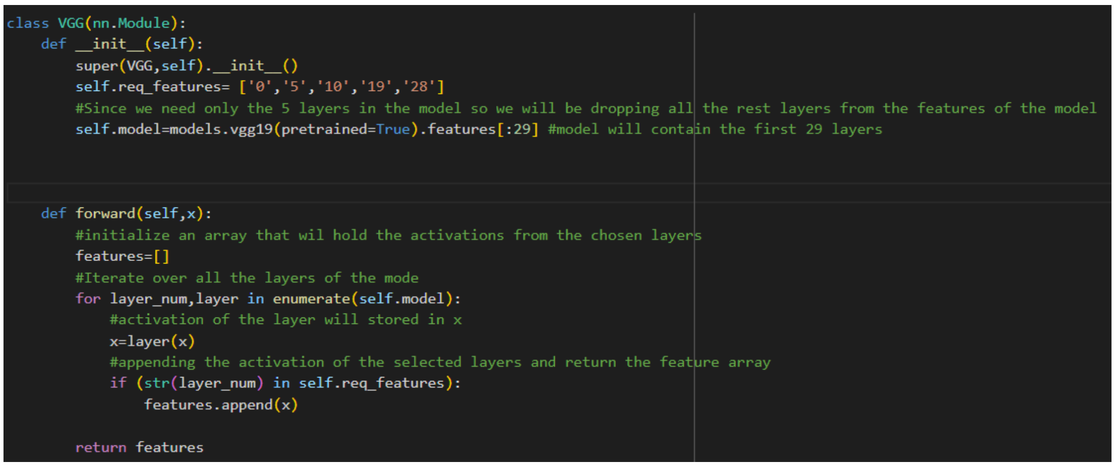
Style Loss:

We build a feature space over each layer of the network representing the correlation between the different filter responses. The Gram matrix calculates these feature correlations.

The gram matrix represents the correlation between each filter in an intermediate representation. Gram matrix is calculated by taking the dot product of the unrolled intermediate representation and its transpose. The gram matrix G dimension is $n^l c$ x $n^l c$, where $n^l c$ is the number of channels in the intermediate representation of layer l.

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

3. Training:


```
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


    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
```

Here, create an object for the class *VGG.* Initializing the object will call the constructor and it will return a model with the first 29 layers and load it to the device. Epoch is 1000, the learning rate is 0.01, alpha(weighting coefficient of content loss) is 1 and beta(weighting coefficient of style loss) is 0.01.

Adam is used as an optimizer. The pixel data of the generated image will be optimized to pass the generated image as the optimizer parameter.

Use a for loop to iterate over the number of epochs. Extract the feature representation of the intermediate layers of the content, style and the generated image using the model. On passing an image to the model, it will return an array of length 5. Each element corresponds to the feature representation of each intermediate layer.

Calculate the total loss by using the above-defined functions. Set the gradients to zero with *optimizer.zero_grads(),* backpropagate the total loss with *total_loss. backward()* and then update the pixel values of the generated image using *optimizer. Step().* We will save the generated image after every 100 epochs and print the total loss**.**

4. Result: