# NEURAL STYLE TRANSFER USING VGG19

## INTRODUCTION:

Neural style transfer (NST) aims to merge the content of one image with the artistic style of another. This process involves using a deep neural network to extract and recombine the essential features of both images. The key innovation in the algorithm lies in its ability to independently manipulate and integrate these distinct elements through a sophisticated understanding of neural network layers.

"A Neural Algorithm of Artistic Style," authored by Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge, was published in 2015 and represents a major breakthrough in the field of neural networks applied to artistic creation.

This paper presents a method to create visually appealing images by transferring the style of one image onto the content of another, using convolutional neural networks (CNNs). This technique has not only influenced digital art but also inspired extensive research and development in neural style transfer.

## APPROACH:

The process consists of multiple crucial steps that utilize the capabilities of a VGG-19 model, a pre-trained convolutional neural network. Here is a thorough breakdown of the methodology:

1. Pre-trained CNN (VGG-19): The basis for extracting and displaying content and style is the VGG-19 network, which was pre-trained on the ImageNet dataset. Convolutional and fully linked layers are among the 19 layers that make up VGG-19. These layers gradually abstract the input image into higher-level characteristics.

2. A higher convolutional layer of the VGG-19 network uses feature maps to encode the content of a picture.
   A layer such as conv4_2 is usually selected because it achieves a compromise between abstract feature representation and specific spatial information.
   The arrangement and fundamental structure of the objects in the image are captured by the feature map at this layer.

3. Style Representation:

- The style is captured using the Gram matrix of the feature maps from multiple layers of the CNN.
- The Gram matrix is a measure of the correlation between different feature maps, effectively encoding the texture, color, and patterns characteristic of the artistic style. Mathematically,

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

- This operation captures the relationships between different features, allowing the style of the image to be represented in a holistic manner.

## **4.** Loss Functions:

- Content Loss: The content loss ensures that the generated image maintains the structure and essential elements of the target image. It is calculated as the mean squared error (MSE) between the feature representations of the target image and the generated image at the chosen content layer:

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

- Style Loss: The style loss ensures that the generated image replicates the texture and color patterns of the style image. It is the sum of the MSE between the Gram matrices of the style image and the generated image across several layers:

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

$$\mathcal{L}_{style}(\vec{a}, \vec{x}) = \sum_{l=0}^{L} w_l E_l \tag{5}$$

- Total Variation Loss: This optional term promotes smoothness in the generated image by penalizing abrupt changes in pixel values:

5. Optimization:

The final generated image is obtained by minimizing the total loss, which is a weighted sum of the content, style, and total variation losses:

$$\mathcal{L}_{total}(\vec{p}, \vec{a}, \vec{x}) = \alpha \mathcal{L}_{content}(\vec{p}, \vec{x}) + \beta \mathcal{L}_{style}(\vec{a}, \vec{x}) \tag{7}$$

Where alpha and beta are content weight and style weight respectively.

The optimization process typically uses gradient descent to iteratively update the pixel values of an initial guess (often white noise or a copy of the content image) to minimize the total loss.

The algorithm produces visually striking images that effectively blend the content of a target image with the style of a reference image. For instance, the technique can transform a photograph of a landscape into

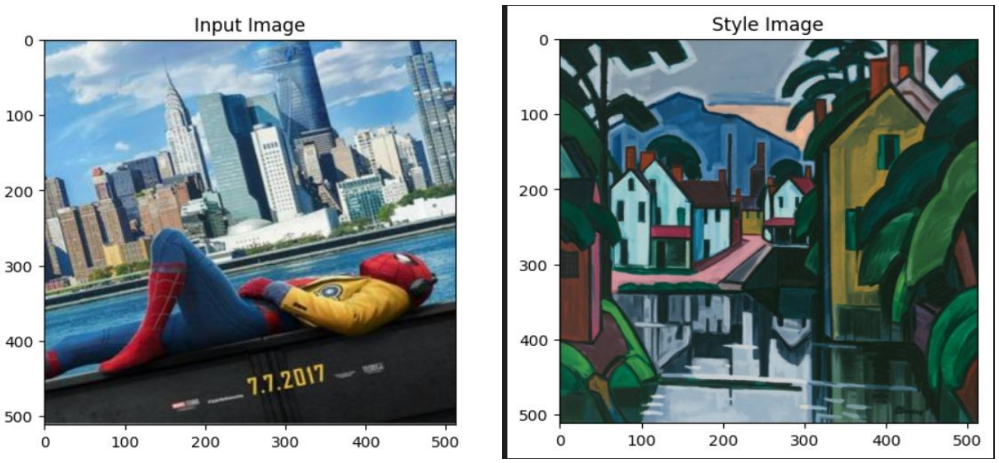
an image that looks like a painting by van Gogh or Picasso, capturing the unique brush strokes, colors, and textures of the artist.

## FAILED APPROACHES:

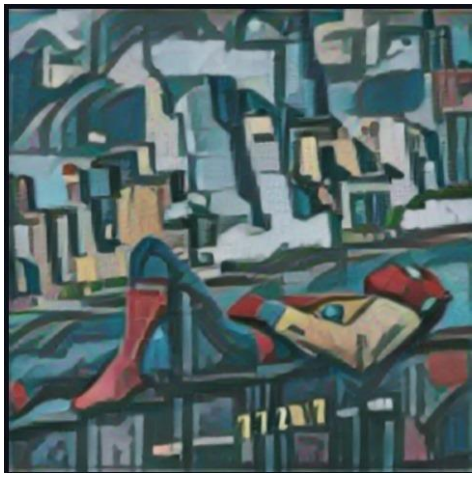
As I was also working on image denoising project simultaneously, so the first approach that I thought would work was treating artistic features in an image as a noise and train upon it, particularly residual learning networks which specifically train upon noise would prove quite effective, that's what I thought. But it didn't and these are the reasons for the failure of this approach:

- Patch-based methods attempted to transfer style by copying and blending patches from the style image onto the content image. I used patching in my denoising project.

- And second reason would be because artistic features aren't really a noise in the first place because they have very distinguishable "patterns" Which cannot be trained upon by denoising models as they have a proper distribution (gaussian, poisons, etc.) and even if artistic features have a distribution, they probably have a very high standard deviation which necessarily renders denoising models as useless.
  Here are some results of failed approach:

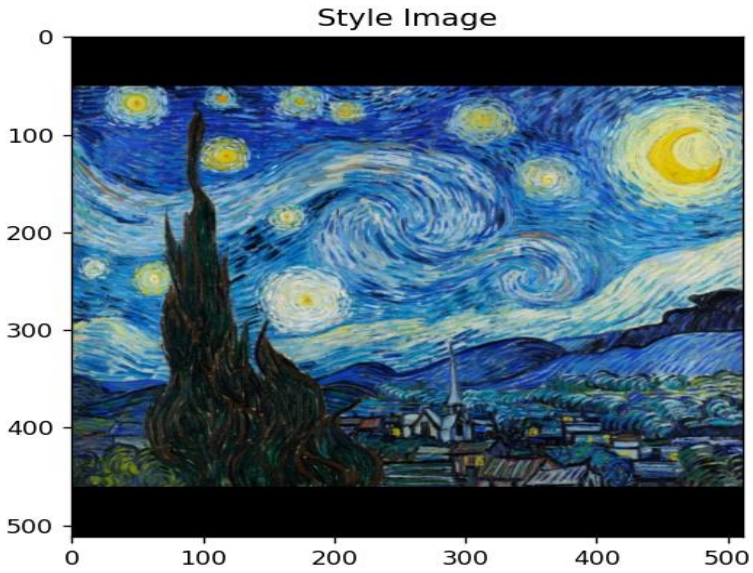



Output was pretty bad,

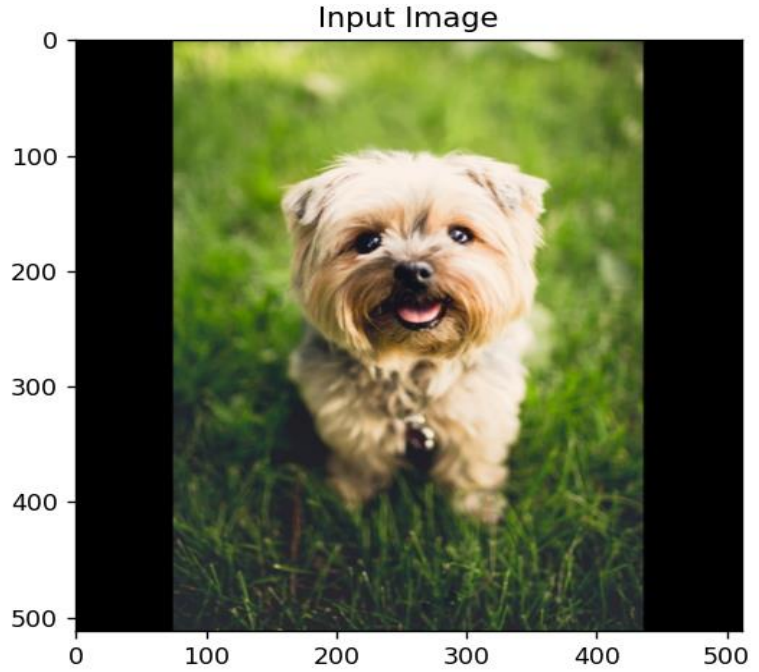


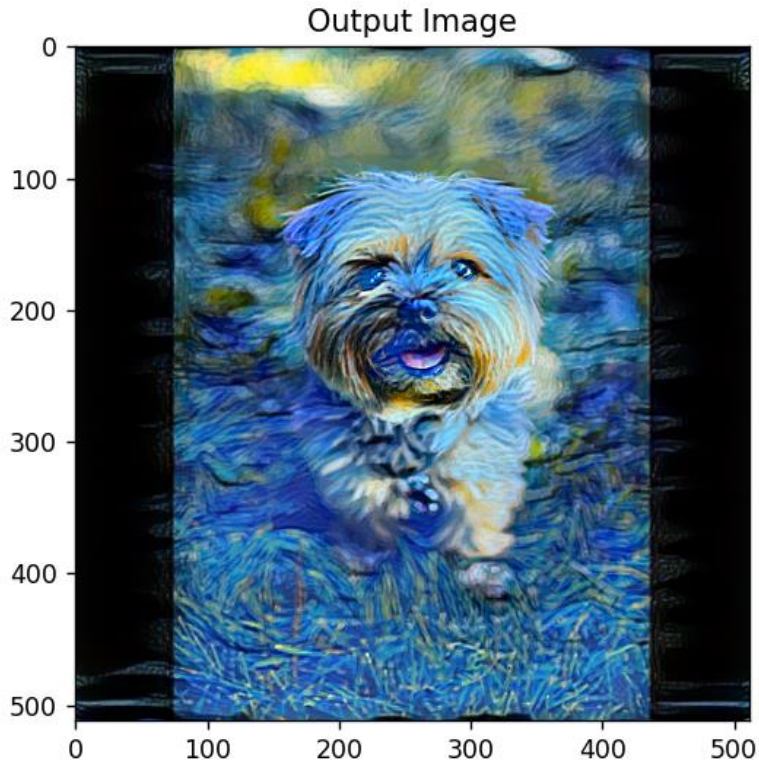
This fails to capture content features and style features.

## RESULTS:




First lets look at visual representation of the results and then we'll look at performance metrics, i.e. content loss and style loss that was calculated using functions. In pytorch implementation by authors, they used LBFGS optimizer but in my own implementation I used adam with learning rate of 0.01, and adam parameters beta = 0.99 and epsilon = 0.1. I thought that lower learning rate would give steady but accurate image. And I also took style weight as 0.01 and content weight as 0.1 because I wanted only a small addition of style to make the artwork look good.




Next I'll be showing a screenshot of content loss and style loss from step 600 to step 1000, which is the final step.

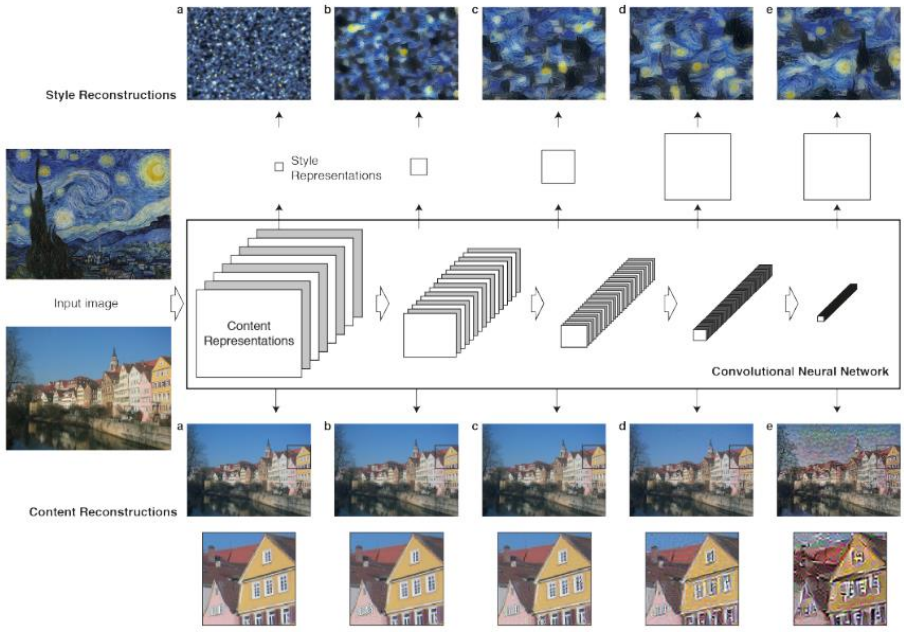
At the end I got content loss as 12.50 and style loss as 0.9375.

```
run [600]:
Style Loss : 1.195673 Content Loss: 13.264277

run [650]:
Style Loss : 1.151106 Content Loss: 13.067209

run [700]:
Style Loss : 1.089733 Content Loss: 12.920897

run [750]:
Style Loss : 1.054048 Content Loss: 12.812530

run [800]:
Style Loss : 1.025453 Content Loss: 12.722008

run [850]:
Style Loss : 0.995683 Content Loss: 12.650919

run [900]:
Style Loss : 0.971355 Content Loss: 12.593427

run [950]:
Style Loss : 0.955819 Content Loss: 12.542355

run [1000]:
Style Loss : 0.937570 Content Loss: 12.506458
```

## DISCUSSION:

As the VGG19 model was primarily used for object detection so the authors used block4_conv2 layer as the input and conv1 layers of different blocks as outputs, which they analysed, last layer gave more sophisticated image with both content features and style features in balance but it didn't look good, and first layer output was too simple to be called a stylised image.

So it was a trade-off between details and style, as shown in the image below



Another insight that I got, when we put some random image as style image which isn't an artwork, we get a very bad and disfigured image as the output and its probably because normal images don't have a pattern that they could train on and often times give garbage and different output every time they're produced.

## REFERENCES:


1. A Neural Algorithm of Artistic Style: https://arxiv.org/pdf/1508.06576

2. pytorch implementation: https://pytorch.org/tutorials/advanced/neural_style_tutorial.html?highlight=style+transfer

3. Arbitrary Image stylization: https://www.kaggle.com/models/google/arbitrary-image-stylization-v1/tensorFlow1/256/2?tfhub-redirect=true

(This was used in deployment as VGG19 was taking too long and streamlit was running out of memory often times, so I chose a lighter model to deploy)

4. VGG19 weights: https://www.kaggle.com/datasets/saksham219/vgg19-weights

(This I downloaded so that I don't have to download VGG19 weights each time I run the code)