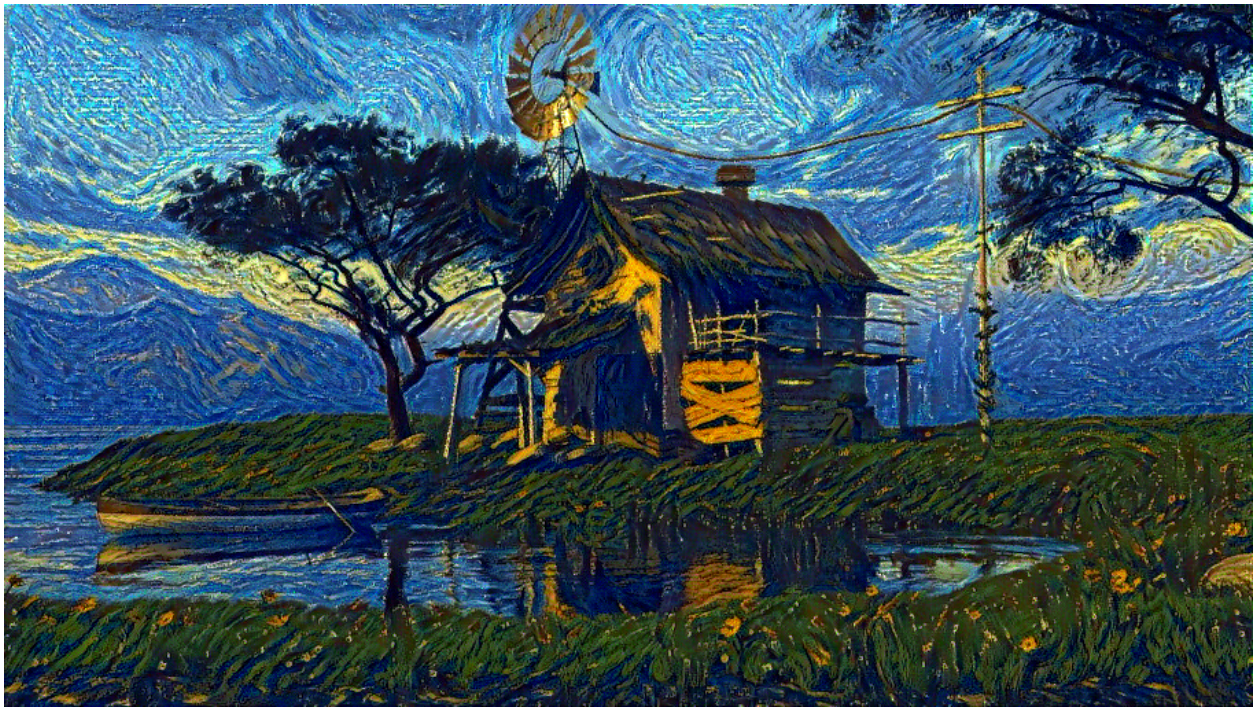
Neural Style Transfer of an Image using CNN

RnD task by Shaurya Choudhary.

NST(Neural Style Transfer) was introduced in 2015 by Gatys et al., demonstrating how convolutional neural networks could blend the content of one image with the artistic style of another.

The system uses neural representations to separate and recombine the content and style of arbitrary images, providing a neural algorithm for the creation of artistic images.

Style type + content image → stylized image



CNN - Convolutional Neural Networks consist of layers of small computational units that process visual information hierarchically in a feed-forward manner. Each layer of units can be understood as a collection of image filters, each of which extracts a certain feature from the input image. Thus, the output of a given layer consists of so-called feature maps.

In this project, we will be using a pre-trained model of VGG-19 and mainly the PyTorch library.

Methodology-

a. Prepare Input

- Style image: (e.g., Van Gogh painting)
- Content image: Here I am using a Blender-rendered 3D scene of a riverside hut.

b. Preprocessing

```
image_size1_width = 911
image_size2_height = 512


loader = transforms.Compose(
    [
        transforms.Resize((image_size2_height, image_size1_width)),
        transforms.ToTensor(),
        # transforms.Normalize(mean=[], std[])
    ]
)

def load_image(image_name):
    image = Image.open(image_name)
    image = loader(image).unsqueeze(0)
    return image.to(device)

original_img = load_image("content.jpg")
style_img = load_image("style.jpg")
```

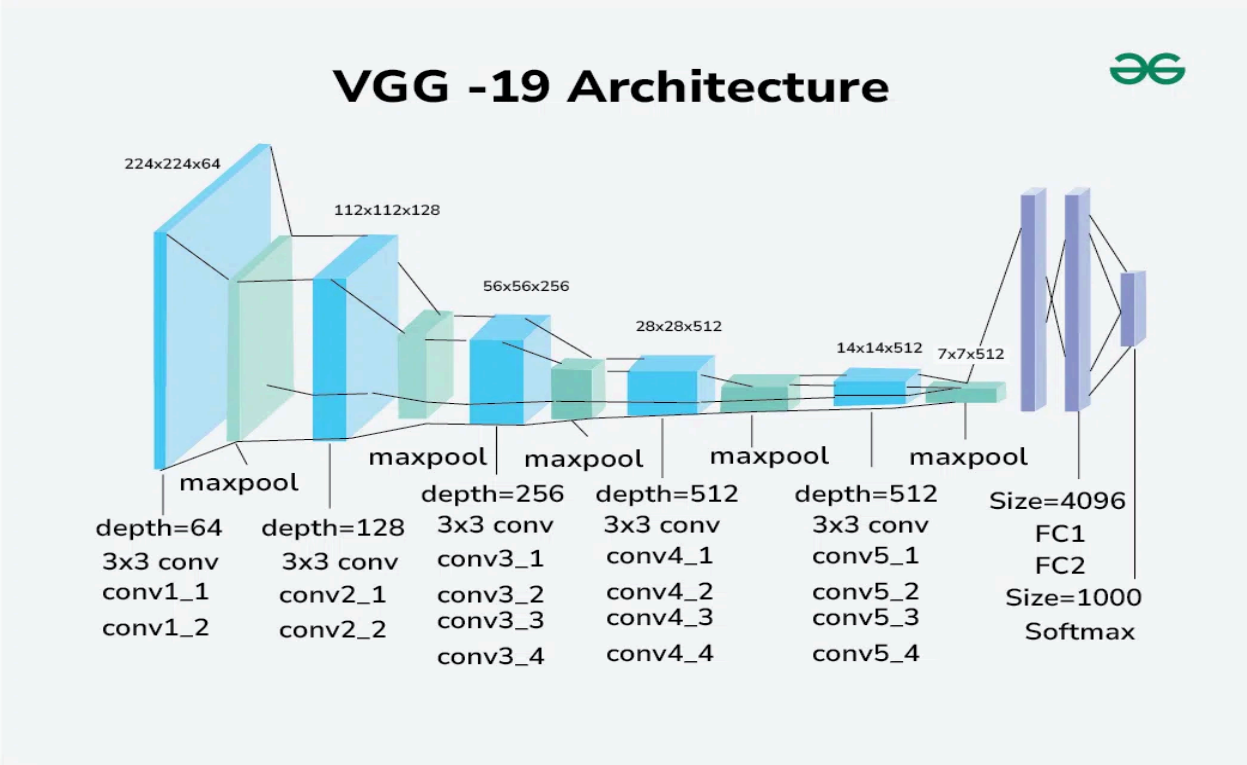
- Here we are first fixing the pixel size of the images that we will be loading in the process.

- Then, we define an image loading function for images, in which we first resize the images to the above-set size and then transform them to Tensor.

- Now we call this function for both style and content images with the desired image name inside the loader function.

c. Model

- Using pretrained VGG19 from PyTorch



```
Sequential(
  (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): ReLU(inplace=True)
  (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (3): ReLU(inplace=True)
  (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (6): ReLU(inplace=True)
  (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (8): ReLU(inplace=True)
  (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (11): ReLU(inplace=True)
  (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (13): ReLU(inplace=True)
  (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (15): ReLU(inplace=True)
  (16): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (17): ReLU(inplace=True)
  (18): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (19): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (20): ReLU(inplace=True)
  (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (22): ReLU(inplace=True)
  (23): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (24): ReLU(inplace=True)
  (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (26): ReLU(inplace=True)
  (27): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (29): ReLU(inplace=True)
  (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (31): ReLU(inplace=True)
  (32): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (33): ReLU(inplace=True)
  (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (35): ReLU(inplace=True)
  (36): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
)
```

VGG-19 Layers

- As we can see VGG-19 model has many different layers of which we will be using some to extract feature maps of our content and style images by giving them as input to the model.

```
class VGG(nn.Module):
    def __init__(self):
        super(VGG, self).__init__()

        self.chosen_features = ['0','5','10','19','28']
        self.model = models.vgg19(pretrained=True).features[:29]

    def forward(self, x):
        features = []

        for layer_num, layer in enumerate(self.model):
            x = layer(x)

            if str(layer_num) in self.chosen_features:
                features.append(x)
        return features
```

- First, we set up the model in which we have chosen a specific layer from which we will extract our features in the future, especially the first ones of every Conv layer(eg, Conv1_1, Conv2_1, so on).
- And at the end we are returning a list containing of features of those specific feature maps.
- Also, here we are only using our VGG-19 model till its 29th line(i.e.till the conv5_1 layer), because after that the model is of no use.

```
model = VGG().to(device).eval()
#generated = torch.randn(original_img.shape, device=device, requires_grad=True)
generated = original_img.clone().requires_grad_(True)

total_steps = 6000
learning_rate = 0.03
alpha = 1
beta = 1000

optimizer = optim.Adam([generated], lr=learning_rate)
```

- We have set generated image initially same as content image for better result.

- At the end for our model we have to fix some hyperparameters that will affect the result like ***total no. of steps***, ***learning rate***, ***Alpha***(it affects how much of content image features is needed in the result) and ***Beta***(it affects the amount of style is visible in result).
- Also, we have to choose an optimizer for our model(like Adam or LBFGS).

d. Training

- Now we will go one by one to total steps and in each step extract feature maps of style img, content img and generated img from the chosen layers and using them our aim is to calculate Style loss and Content loss from which at end we can get Total loss.

```
for step in range(total_steps):
    generated_features = model(generated)
    original_img_features = model(original_img)
    style_features = model(style_img)

    style_loss = original_loss = 0

    for gen_feature, orig_feature, style_feature in zip(
        generated_features, original_img_features, style_features
    ):
        batch_size, channel, height, width = gen_feature.shape
        original_loss += torch.mean((gen_feature - orig_feature) ** 2)

        #computing gram matrix
        G = gen_feature.view(channel, height*width).mm(
            gen_feature.view(channel, height*width).t()
        )

        A = style_feature.view(channel, height*width).mm(
            style_feature.view(channel, height*width).t()
        )

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

    total_loss = alpha * original_loss + beta * style_loss
    optimizer.zero_grad()
    total_loss.backward()
    optimizer.step()
```

- *Content_Loss*:- If p and x be the original image and the image that is generated and P(l) and F(l) are their respective feature representation in layer l. We then define the squared-error loss between the two feature representations

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

The derivative of this loss with respect to the activations in layer l equals

$$\frac{\partial \mathcal{L}_{content}}{\partial F_{ij}^l} = \begin{cases} \left(F^l - P^l\right)_{ij} & \text{if } F_{ij}^l > 0 \\ 0 & \text{if } F_{ij}^l < 0 . \end{cases}$$

from which the gradient with respect to the image x can be computed using standard error back-propagation. Thus, we can change the initially random image x until it generates the same response in a certain layer of the CNN as the original image p.

- *Style_Loss*:- A style representation that computes the correlations between the different filter responses, where the expectation is taken over the spatial extent of the input image. These feature correlations are given by the Gram matrix G.
  Where G(l) ij is the inner product between the vectorised feature map i and j in layer l:

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

Minimising the mean-squared distance between the entries of the Gram matrix from the original image and the Gram matrix of the image to be generated.

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

and the total loss is

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

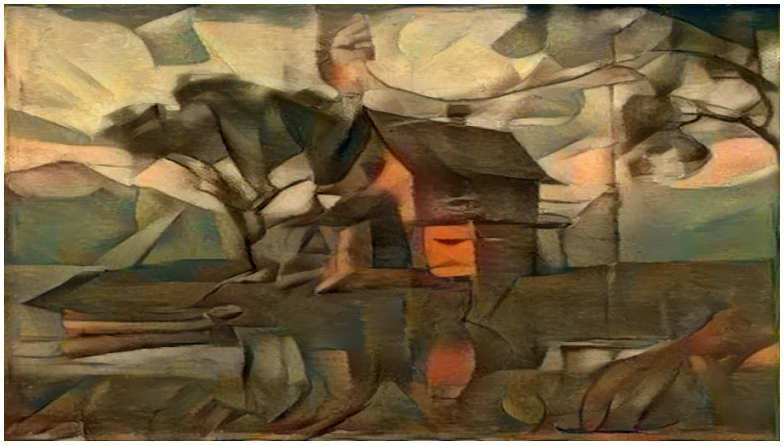
- *Total_Loss:-*

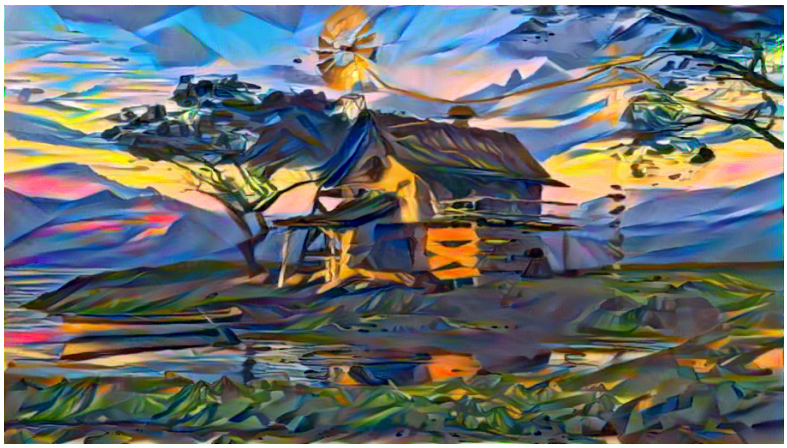
$$\mathcal{L}_{total}(\vec{p}, \vec{a}, \vec{x}) = \alpha \mathcal{L}_{content}(\vec{p}, \vec{x}) + \beta \mathcal{L}_{style}(\vec{a}, \vec{x})$$

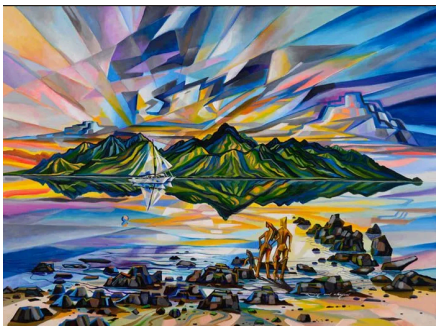
- After calculating total loss we use backward gradient descent to correct the weights and apply on the generated image by changing its pixel according to the loss grad.

e. Results

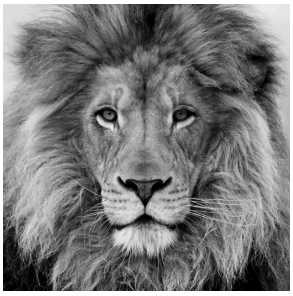
First, I used my Blender render as the content image for different styles.

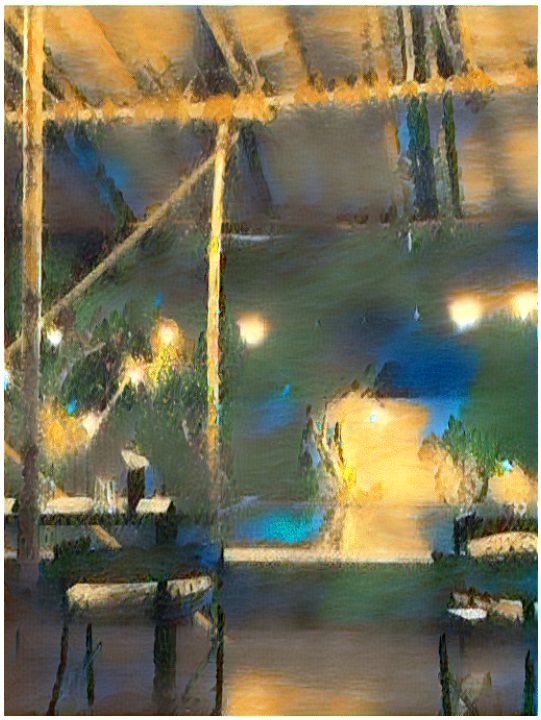






Some other results:-





f. Problems

- The image generation is a small process.
- The hyperparameters dosent has a fix value and gives different output depending on them.

G. References

- Basic PyTorch theory
  Building a CNN using PyTorch | Video 11 | CampusX
- Official paper on NST, including everything and is a great read to get the idea (A Neural Algorithm of Artistic Style arXiv:1508.06576v2 [cs.CV] 2 Sep 2015)
- Video reference of the theory part
  Basic Theory | Neural Style Transfer #2
- VGG19 model
  VGG19 architecture & implementation | Image Classification | Deep l…
- Code reference video Pytorch Neural Style Transfer Tutorial

Further ideas:-

Currently working on Semantic segmentation and instance segmentation using OpenCV.