Neural Style Tranfer

Introduction

Neural Style Transfer (NST) is a technique which combines two images (content image for the object of the image and the style image from which only the style is extracted) into a third target image. Simply put, take one image and paint it in the style of another image while keeping the initial content.

Description

Dependencies

Following dependencies were used

```
[2]  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 tensorflow as tf
     import matplotlib.pyplot as plt
     import numpy as np
```

Loading the Model

We will use the pretrained VGG19 model here. It has 3 components:

1. features, that hold all the convolutional, max pool ans ReLu layers
2. avgpool, that holds average pool layer
3. classifier, that holds dense layers

We are using only convolutional neural network for simplicity in training. Hence are including only features.

Image Preprocessing

I have used torch.tranform to

1. resize all the images to 512 x 512
2. to convert images into tensor

Then I have loaded the content and style images, and I have used a clone of content image and modified it to obtain the results. For this I have allowed the cloned image to be modified by gradient descent.

```
[14] def image_loader(path):
         image = Image.open(path)
         loader=transforms.Compose([transforms.Resize((512,512)), transforms.ToTensor()])
         image=loader(image).unsqueeze(0)
         return image

     content_path = tf.keras.utils.get_file('YellowLabradorLooking_new.jpg', 'https://storage.googleapis.c
     style_path = tf.keras.utils.get_file('kandinsky5.jpg','https://storage.googleapis.com/download.tensor

     original_image=image_loader(content_path)
     style_image=image_loader(style_path)
     generated_image=original_image.clone().requires_grad_(True)
```

Feature Representation

I've defined a class that provides the feature representations of the intermediate layers. Intermediate layers are utilized because they serve as complex feature extractors, effectively describing the style and content of the input image. In this class, I initialized a model by removing the unused layers (those beyond 'conv5_1') of the VGG19 model. I then extracted the activations or feature representations of the 'conv1_1', 'conv2_1', 'conv3_1', 'conv4_1', and 'conv5_1' layers (with index values [0, 5, 10, 19, 28]). These activations from the five convolutional layers are stored in an array, which the class returns.

```
[15] class VGG(nn.Module):
         def __init__(self):
             super(VGG,self).__init__()
             self.req_features= ['0','5','10','19','28']
             self.model=models.vgg19(pretrained=True).features[:29] #model will contain the first 29 layer

     # X holds the input tensor(image) that will be feeded to each layer
         def forward(self,X):
             features=[]
             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
```

Losses

The complete loss is a combination of the content and style losses: simply put, we try to minimize the jointly losses on the target image:

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

Content Loss

We feed the network both the content image and the target image, which return the feature representation of each image from the intermediate layers. To get the content loss, we calculate the squared error loss between the two feature vectors:

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

In formal terms, the function above calculates the loss using the content image **p**, the target image **x**, and the processed layer l. Here, F and P represent the feature representations of the target image and the content image on layer l, respectively. The gradient is computed using simple error backpropagation, allowing us to adjust the target image until its feature vector matches that of the content image.

Style Loss

Before calculating the style loss, it's essential to understand how to capture the style of an image and measure the correlation between features after each layer. To obtain these correlations, we use Gram matrices. Specifically, the original paper utilizes the Gram matrix G, but any algorithm that disregards feature presence and position can be used. The paper "Demystifying Neural Style Transfer" explores alternative methods that yield results comparable to those achieved with Gram matrices. As noted in the paper, the style loss is computed by determining the Maximum Mean Discrepancy (MMD) between two images. Therefore, as long as the measure implements the MMD algorithm, the loss is computed accurately.

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

The Gram matrix is calculated for each layer. To reconstruct the style, a gradient descent is performed on the target image by computing the mean-squared distance between the Gram matrices. The total loss is obtained by summing the mean-squared distances for each layer, multiplied by the corresponding weighted factor (influence factor) for each layer.

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

```
[16] # Content loss
     def calc_content_loss(gen_feat,orig_feat):
         #Calculating the content loss of each layer by calculating the MSE between the content and generated features ar
         content_l=torch.mean((gen_feat-orig_feat)**2)
         return content_l

     # Style loss
     def calc_style_loss(gen,style):
         #Calculating the gram matrix for the style and the generated image
         batch_size,channel,height,width=gen.shape

         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*width).t())

         #Calcultating the style loss of each layer by calculating the MSE between the gram matrix of the style image anc
         style_l=torch.mean((G-A)**2)
         return style_l

     # Total loss
     def calculate_loss(gen_features, orig_feautes, style_featues):
         style_loss=content_loss=0
         for gen,cont,style in zip(gen_features,orig_feautes,style_featues):
             #extracting the dimensions from the generated image
             content_loss+=calc_content_loss(gen,cont)
             style_loss+=calc_style_loss(gen,style)

         #calculating the total loss of e th epoch
         total_loss=alpha*content_loss + beta*style_loss
         return total_loss
```

Training and Results

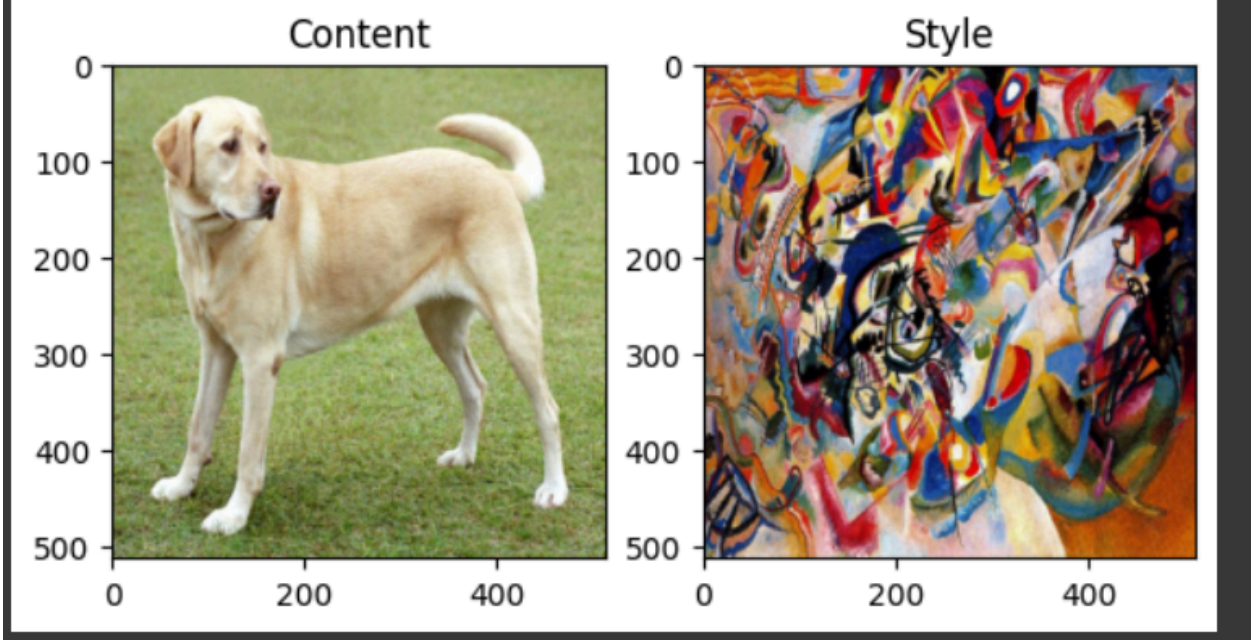
```
[17] # Setting up hyperparameters
     model=VGG().eval()

     epoch=2000
     lr=0.004
     alpha=8
     beta=70

     #Using adam optimizer, it will update the generated image not the model parameter
     optimizer=optim.Adam([generated_image],lr=lr)
```

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

    total_loss=calculate_loss(gen_features, orig_feautes, style_featues)
    (variable) total_loss: Tensor | int
    total_loss.backward()
    optimizer.step()

    if(e/100):
        print(total_loss)

        save_image(generated_image,"gen.png")
```

```
tensor(1.7394e+10, grad_fn=<AddBackward0>)
tensor(1.5979e+10, grad_fn=<AddBackward0>)
tensor(1.4698e+10, grad_fn=<AddBackward0>)
tensor(1.3551e+10, grad_fn=<AddBackward0>)
tensor(1.2535e+10, grad_fn=<AddBackward0>)
tensor(1.1639e+10, grad_fn=<AddBackward0>)
```

Inputs

```
195196/195196 [==============================] - 0s 0us/step
```



Output

Details

Harsh Raj
22119024
Mechanical Engineering
Mobile: 8789071279