TECHNOLOGICAL INSTITUTE OF THE PHILIPPINES
938 Aurora Blvd., Cubao, Quezon City

COLLEGE OF ENGINEERING AND ARCHITECTURE
ELECTRONICS ENGINEERING DEPARTMENT

1ST SEMESTER SY 2022 - 2023

Prediction and Machine Learning

COE 005
ECE41S11

Homework 2
Neural Style Transfer
Submitted to:
Engr. Christian Lian Paulo Rioflorido

Submitted on:
10/19/2022

Submitted by:
Joseph Lance C. Hilario

Discussion:

Style transfer is one of the many techniques that we can use in Python that can combine two images together. There is a content image and a style image and the outcome of this program would be the style of art in the style image layover on the content image. This uses convolutional Neural Networks to achieve this.

Style transfer has two types of losses, content Loss and style loss. To calculate for these losses, we must first initialize a third image which is the output image. This image is what the program will produce given the style and content images. The content loss is calculated by selecting one of the many layers of the model then it is feedforwarded to the content image thru a pre-trained model. This will give s a feature map that we will be used to calculate the loss. The next would be the style loss. To calculate for the style loss there should be a style representation of the image. In this model, the style representation defined is the Gram matrix of the feature maps. It measures the statistic features and correlation between the feature maps. Finally, we will have the total loss. It is the weighted summation of the style loss and the content loss. We also define the style and content weight to multiply them to the content loss. Once we multiply them and add them the summation is the total loss.

For this we will use Pytorch to recreate the Image style Transfer Using Convolutional Neural Networks by Gates in pytorch, using the features found in the 19-layer VGG Network. This comprises a series of convolutional and pooling layers, and some fully-connected layers.

First, we import the necessary libraries and resources.

```
In [72]: from PIL import Image
         import matplotlib.pyplot as plt
         import numpy as np

         import torch
         import torch.optim as optim
         from torchvision import transforms, models
```

Next, we will load the model. The model to be used is the VGG19, this has two parts. The first consists of a convolutional layer and the second composes of two fully connected layers that acts as the classifier. Our goal is to get the feature maps produced in the convolutional layers so that we can load the first part. This model is loaded from torchvision pre-trained model. Also, we need to freeze the parameters of the convolutional layers so that it will not change. We only

need to change the output image. And lastly, we will move the model to the GPU if its available.

```
In [73]: # loading the first part
         vgg = models.vgg19(pretrained=True).features

         # freezing the parameters
         for param in vgg.parameters():
             param.requires_grad_(False)

         # move the model to GPU, if available
         device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

         vgg.to(device)
```

```
Downloading: "https://download.pytorch.org/models/vgg19-dcbb9e9d.pth" to C:\Users\Lance/.cache\torch\hub\checkpoints\vgg19-dcbb9e9d.pth
100%|██████████| 548M/548M [04:32<00:00, 2.11MB/s]

Out[73]: 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)
         )
```

Next, we will load the content image and the style image. This will load the image from the image path. Then use image.open from the pillow library. And convert it into RGB. We will then put a threshold for its size using max_style to help with processing because bigger pictures tend to be processed slow. Next, we will preprocess the image using transform. This will do three operations. It will resize the image. Then convert it into a tensor. And normalize the produced tensor. Lastly. We will just discard the alpha channel and add a one dimension for the batch number.

In [74]:
```
def load_image(img_path, max_size=400, shape=None):

    image = Image.open(img_path).convert('RGB')

    # large images will slow down processing
    if max(image.size) > max_size:
        size = max_size
    else:
        size = max(image.size)

    if shape is not None:
        size = shape

    in_transform = transforms.Compose([
                        transforms.Resize(size),
                        transforms.ToTensor(),
                        transforms.Normalize((0.485, 0.456, 0.406),
                                             (0.229, 0.224, 0.225))])

    # discard the transparent, alpha channel (that's the :3) and add the batch dimension
    image = in_transform(image)[:3,:,:].unsqueeze(0)

    return image
```

After preprocessing, we will now call the content image and style image.

In [75]:
```
# Here you must write the path of your content image
content_image_path = 'tip1.jpg'

content = load_image(content_image_path).to(device)

# Here you must write the path of your style image
style_image_path = 'amor2.jpg'

# Resize style to match content.
style = load_image(style_image_path, shape=content.shape[-2:]).to(device)
```

Now we get the Features. To get the features we will use the ```get_feature``` function to return the features maps produced by the layers in the dictionary. These features are for calculating the losses. This will be indexed using numbers. We will use the layers shown below in the code to feedforward the image through. Then store the feature map of each layers in a new dictionary features.

In [76]:
```
def get_features(image, model, layers=None):

    # The layers that are mentioned in Gatys et al (2016) paper
    if layers is None:
        layers = {'0': 'conv1_1',
                  '5': 'conv2_1',
                 '10': 'conv3_1',
                 '19': 'conv4_1',
                 '21': 'conv4_2',
                 '28' : 'conv5_1'}

    features = {}
    x = image
    # model._modules is a dictionary holding each module in the model
    for name, layer in model._modules.items():
        x = layer(x)
        if name in layers:
            features[layers[name]] = x

    return features
```

Next is the Gram matrix. We need to get the depth, height, and width of the tensor via tensor.shape and reshape that tensor so that its spatial dimensions are flattened and use the

view function to reshape the tensor. Lastly, we will calculate the Gram matrix by multiplying the reshaped tensor by the transpose using Torch.mm

In [77]:
```
def gram_matrix(tensor):

    ## reshape it, so we are multiplying the features for each channel
    tensor = tensor.view(tensor.shape[1], tensor.shape[2]*tensor.shape[3])

    ## calculate the gram matrix
    gram = torch.mm(tensor,tensor.t())

    return gram
```

Next, we will be getting the features for both the style image and content image and calculate its gram matrix for each layer for our style representation. Also to create a copy from our content image and assign it to our output image.

In [78]:
```
# get content and style features
content_features = get_features(content, vgg)
style_features = get_features(style, vgg)

# calculate the gram matrices for each layer of our style representation
style_grams = {layer: gram_matrix(style_features[layer]) for layer in style_features}

# Initalizing our target image
target = content.clone().requires_grad_(True).to(device)
```

Next we will un-normalize the image and convert it from tensor t oa numpy image for display.

In [82]:
```
def im_convert(tensor):

    image = tensor.to("cpu").clone().detach()
    image = image.numpy().squeeze()
    image = image.transpose(1,2,0)
    image = image * np.array((0.229, 0.224, 0.225)) + np.array((0.485, 0.456, 0.406))
    image = image.clip(0, 1)

    return image
```

Next we will look at the two kinds of weights in the progarm. The style representation weights will put weights for the layers that can be used to calculate the gram matrix and the style and content loss weights as explained in the above discussion. This ratio will affect how the program will style the final image.

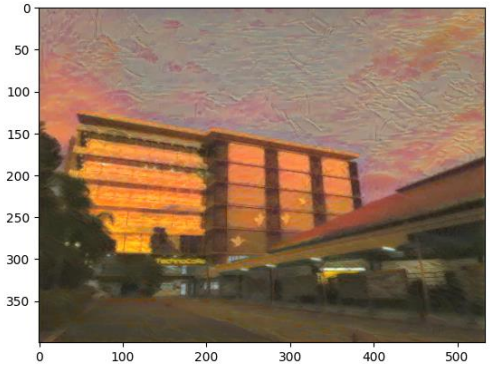
In [83]:
```
# weights for each style layer
# weighting earlier layers more will result in *larger* style artifacts
# notice we are excluding `conv4_2` our content representation
style_weights = {'conv1_1': 1.,
                 'conv2_1': 0.8,
                 'conv3_1': 0.5,
                 'conv4_1': 0.3,
                 'conv5_1': 0.1}

content_weight = 1  # alpha
style_weight = 1e6  # beta
```

Now its type time to update the output image using the content features and style features to generate that we need for the output. We need to determine the hyperparameters and we chose Adam as our optimizer. Then we need to determine the number of steps are needed to update the output image. Then we get the features. And calculate the loss between the output and content features. We also calculate the style loss that we iterate through each style layer and get the coresponding target featuers from ```target features```. After calculating the total

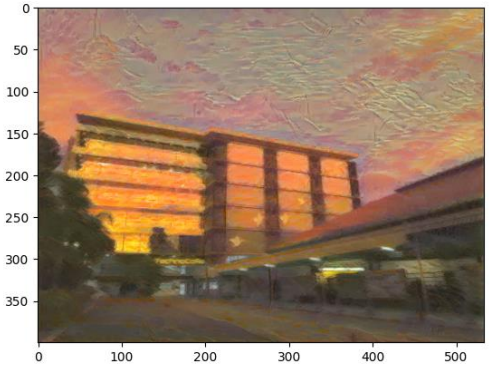
loss wich is the weighted sum of the content loss and style loss. We will update the target image usign the back-propagation loss ```total_loss.backward() and do one optimizer step. Then display the loss and intermediate images every show_every steps.

In [84]:
```
# for displaying the target image, intermittently
show_every = 400

# iteration hyperparameters
optimizer = optim.Adam([target], lr=0.003)
steps = 2000  # decide how many iterations to update your image (5000)

for ii in range(1, steps+1):

    # get the features from your target image
    target_features = get_features(target, vgg)

    # calculate the content loss
    content_loss = torch.mean((target_features['conv4_2'] - content_features['conv4_2'])**2)

    # initialize the style loss to 0
    style_loss = 0
    # iterate through each style layer and add to the style loss
    for layer in style_weights:

        # get the "target" style representation for the layer
        target_feature = target_features[layer]
        _, d, h, w = target_feature.shape

        # Calculate the target gram matrix
        target_gram = gram_matrix(target_feature)

        # get the "style" style representation
        style_gram = style_grams[layer]

        # Calculate the style loss for one layer, weighted appropriately
        layer_style_loss = style_weights[layer]* torch.mean((target_gram - style_gram)**2)

        # add to the style loss
        style_loss += layer_style_loss / (d * h * w)


    # calculate the *total* loss
    total_loss = content_weight*content_loss + style_weight*style_loss

    # update your target image
    optimizer.zero_grad()
    total_loss.backward()
    optimizer.step()

    # display intermediate images and print the loss
    if  ii % show_every == 0:
        print('Total loss: ', total_loss.item())
        plt.imshow(im_convert(target))
        plt.show()
```

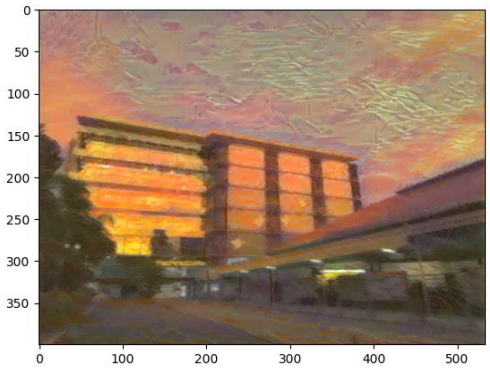
Total loss: 546381.3125



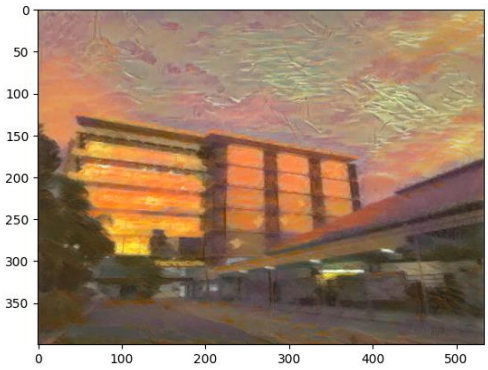
Total loss: 183283.09375



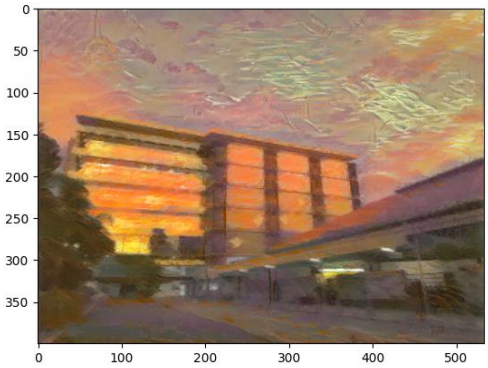
```
Total loss:  113555.5078125
```



```
Total loss:  79379.640625
```



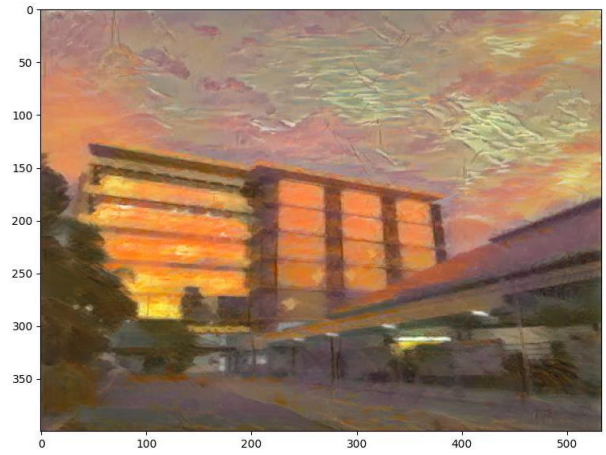
```
Total loss:  58938.8671875
```



Then we just display the final result.

```
In [109]: # display content and final, target image
          fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 10))
          ax1.imshow(im_convert(content))
          ax2.imshow(im_convert(target))
          plt.savefig('tipaomrsolo.jpg')
```

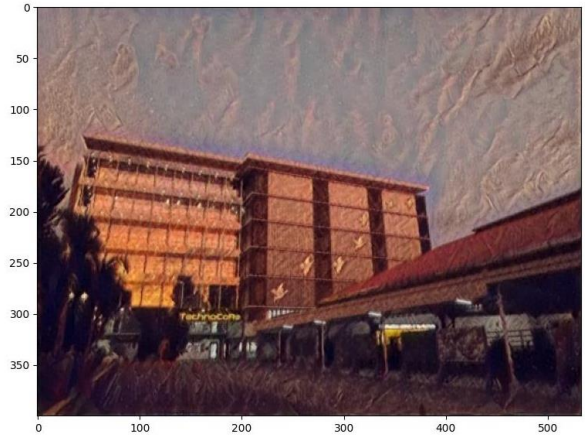
Content image



Output image with Amorsolo style

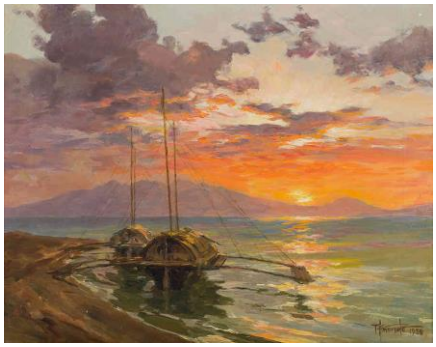
We will just repeat the same program with the other style and show the output.

Content image



Output image with Juan luna style

Spolarium by Juan Luna as reference



Sunrise by Fernando Amorsolo as reference

References:

M. Almaki, “Neural style transfer using CNN,” *OpenGenus IQ: Computing Expertise & Legacy*, 20-Mar-2019. [Online]. Available: https://iq.opengenus.org/neural-style-transfer-cnn/. [Accessed: 19-Oct-2022].

E. Mirzazada, “Neural style transfer with deep VGG model,” *Medium*, 13-May-2020. [Online]. Available: https://medium.com/@mirzezadeh.elvin/neural-style-transfer-with-deep-vgg-model-26b11ea06b7e. [Accessed: 19-Oct-2022].