Project Documentation: Neural Style Transfer

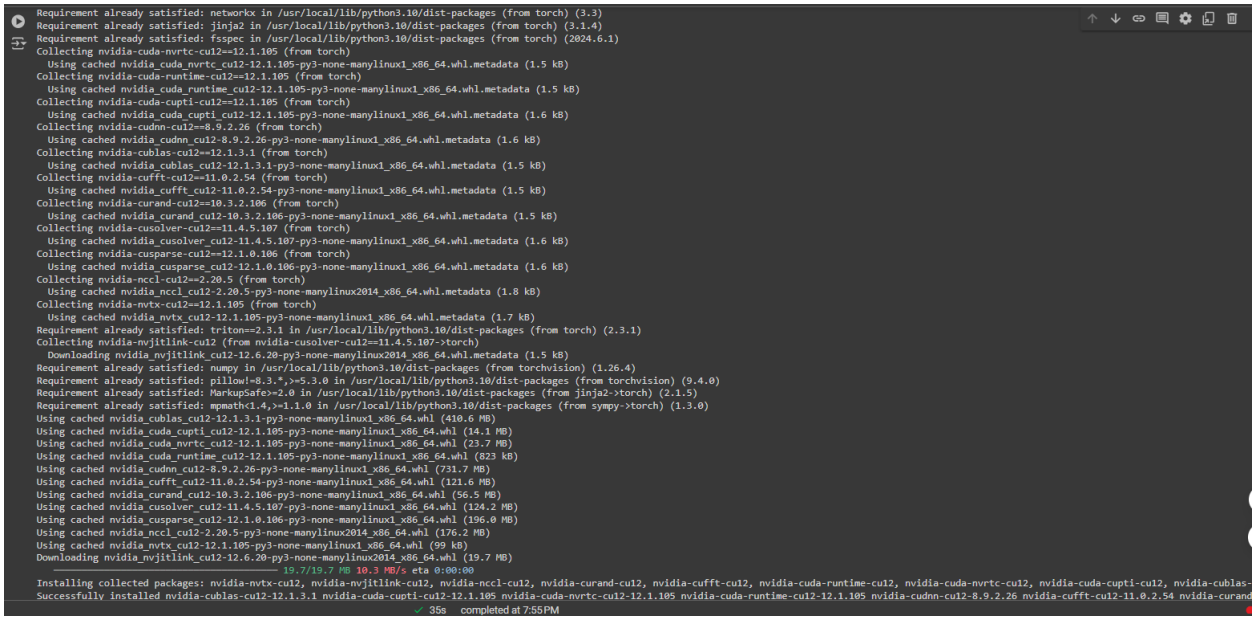
Overview

The goal of this project was to apply Neural Style Transfer (NST) techniques to merge the content of one image with the artistic style of another. This process involves leveraging deep learning models to achieve a visually compelling result that combines the structural details of one image with the stylistic elements of another.

Setup and Installation

1. The command `!pip install torch torchvision` installs essential Python libraries for the project. **`torch`** is the core library of PyTorch, used for creating and training neural networks. **torchvision`** complements PyTorch by providing utilities for image processing, including pre-trained models and transformation tools. These libraries are crucial for implementing Neural Style Transfer and executing deep learning tasks efficiently.

```
# Install required libraries
!pip install torch torchvision
```



2. The code imports necessary libraries and defines helper functions for Neural Style Transfer. **`load_image`** preprocesses and converts an image to a tensor, **`imshow`** displays the image tensor, and **`gram_matrix`** computes the Gram matrix to measure style similarity.

```
[ ]
# Import Libraries and Define Helper Functions
import torch
import torch.optim as optim
from torchvision import models, transforms
from PIL import Image
import matplotlib.pyplot as plt
import numpy as np

# Helper function to load an image and apply necessary transformations
def load_image(image_path, size=512):
    image = Image.open(image_path).convert('RGB')  # Open and convert image to RGB
    preprocess = transforms.Compose([
        transforms.Resize(size),  # Resize image
        transforms.ToTensor(),  # Convert image to tensor
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])  # Normalize image
    ])
    return preprocess(image).unsqueeze(0).to(torch.device('cuda' if torch.cuda.is_available() else 'cpu'))

# Helper function to display an image tensor
def imshow(tensor, title=None):
    image = tensor.cpu().clone().detach().squeeze(0)  # Move tensor to CPU, remove batch dimension
    image = transforms.ToPILImage()(image)  # Convert tensor to PIL image
    plt.imshow(image)  # Display the image
    if title:
        plt.title(title)
    plt.show()

# Helper function to compute the Gram matrix for style loss
def gram_matrix(tensor):
    batch_size, f_map_num, h, w = tensor.size()  # Get dimensions of the tensor
    features = tensor.view(batch_size * f_map_num, h * w)  # Reshape tensor
    gram = torch.mm(features, features.t())  # Compute Gram matrix
    return gram.div(batch_size * f_map_num * h * w)  # Normalize
```

3. This line of code sets the computing device for the neural network operations. It checks if a GPU is available using CUDA; if so, it selects the GPU for faster processing. If not, it defaults to using the CPU. This ensures that the code can run efficiently on available hardware.

```
# Set the device to GPU if available, otherwise use CPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

4. This code imports the `VGG19_Weights` class from `torchvision.models` and uses it to load a pre-trained VGG19 model with updated weights. The model is configured to use weights trained on ImageNet (specified as `IMAGENET1K_V1`) or the latest weights available. The model's feature extractor part is then transferred to the specified computing device (GPU or

CPU) and set to evaluation mode, preparing it for feature extraction in the Neural Style Transfer process.

```
[ ]  from torchvision.models import VGG19_Weights
     # Load a pre-trained VGG19 model with updated weights parameter
     weights = VGG19_Weights.IMAGENET1K_V1  # You can also use VGG19_Weights.DEFAULT for the latest weights
     model = models.vgg19(weights=weights).features.to(device).eval()
```

5. This code loads two images, `unsplash1.jpg` as the content image and `unsplash2.jpg` as the style image, using the `load_image` function. It then transfers both images to the specified computing device (GPU or CPU), ensuring they are ready for processing in the Neural Style Transfer model.

```
✓ 1s  [8]  # Load the content image and style image, ensuring they are on the correct device
           content_img = load_image('unsplash1.jpg').to(device)
           style_img = load_image('unsplash2.jpg').to(device)
```

6. This code defines two custom loss classes for Neural Style Transfer: **`ContentLoss`** and **`StyleLoss`**. The **`ContentLoss`** class calculates the content loss by comparing the features of the input image to a target image using mean squared error (MSE). The **`StyleLoss`** class computes the style loss by comparing the Gram matrices of the input and target images, also using MSE. Both classes use the target values without gradients to ensure that they are not updated during optimization.

```python
import torch.nn as nn
import torch.nn.functional as F

# Define a loss class for content
class ContentLoss(nn.Module):
    def __init__(self, target):
        super(ContentLoss, self).__init__()
        self.target = target.detach()  # Store target features, no gradient needed

    def forward(self, input):
        self.loss = F.mse_loss(input, self.target)  # Compute content loss
        return input

# Define a loss class for style
class StyleLoss(nn.Module):
    def __init__(self, target_feature):
        super(StyleLoss, self).__init__()
        self.target = gram_matrix(target_feature).detach()  # Compute target Gram matrix

    def forward(self, input):
        G = gram_matrix(input)  # Compute Gram matrix for input
        self.loss = F.mse_loss(G, self.target)  # Compute style loss
        return input
```

7. This code defines a function to create a model for Neural Style Transfer that includes content and style loss modules. The function first sets the model to evaluation mode and normalizes the images. It then resizes the style image to match the content image's dimensions. The function builds a new model by iterating through layers of the provided CNN, adding loss modules at specific points. Content loss is added after the fourth convolutional layer, and style loss is added after several convolutional layers. Finally, the model is trimmed to include only the layers up to the last content and style loss modules. This setup allows for efficient computation of content and style losses during the style transfer process.

```python
# Function to create a model with content and style loss modules
def get_style_model_and_losses(cnn, style_img, content_img):
    cnn = cnn.to(device).eval()  # Move model to device and set to evaluation mode
    normalization_mean = torch.tensor([0.485, 0.456, 0.406]).to(device)
    normalization_std = torch.tensor([0.229, 0.224, 0.225]).to(device)

    # Resize style image to match content image dimensions
    style_img = F.interpolate(style_img, size=content_img.shape[-2:], mode='bilinear', align_corners=False)

    normalization = nn.Sequential(
        nn.Conv2d(3, 3, kernel_size=1),
        nn.InstanceNorm2d(3, affine=True)
    ).to(device)

    content_losses = []
    style_losses = []

    model = nn.Sequential(normalization)

    i = 0
    for layer in cnn.children():
        if isinstance(layer, nn.Conv2d):
            i += 1
            name = 'conv_{}'.format(i)
        elif isinstance(layer, nn.ReLU):
            name = 'relu_{}'.format(i)
            layer = nn.ReLU(inplace=False)
        elif isinstance(layer, nn.MaxPool2d):
            name = 'pool_{}'.format(i)
        elif isinstance(layer, nn.BatchNorm2d):
            name = 'bn_{}'.format(i)
        else:
            raise RuntimeError('Unrecognized layer: {}'.format(layer.__class__.__name__))

        model.add_module(name, layer)
```

```python
[13]        # Add content loss
            if name == 'conv_4':
                target = model(content_img).detach()
                content_loss = ContentLoss(target)
                model.add_module("content_loss_{}".format(i), content_loss)
                content_losses.append(content_loss)

            # Add style loss
            if name in ['conv_1', 'conv_2', 'conv_3', 'conv_4', 'conv_5']:
                target_feature = model(style_img).detach() # Pass the resized style image
                style_loss = StyleLoss(target_feature)
                model.add_module("style_loss_{}".format(i), style_loss)
                style_losses.append(style_loss)

        # Trim off the layers after the last content and style losses
        for i in range(len(model) - 1, -1, -1):
            if isinstance(model[i], ContentLoss) or isinstance(model[i], StyleLoss):
                break

        model = model[:i + 1]

        return model, style_losses, content_losses
```

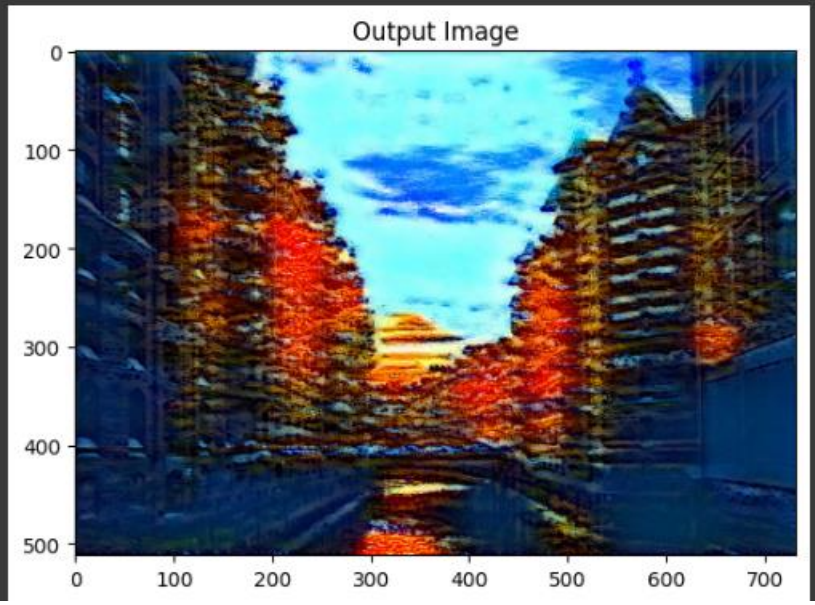
8. This code defines a function to perform style transfer using an optimization approach. It first builds the style transfer model with content and style loss modules. The function then initializes an LBFGS optimizer to update the input image. During optimization, it repeatedly computes the total loss by summing weighted style and content losses, adjusts the input image, and prints progress every 50 steps. The final image is clamped to ensure pixel values remain within a valid range. This iterative process refines the input image to combine the content of one image with the style of another.

```
[14] # Function to run style transfer
     def run_style_transfer(cnn, content_img, style_img, input_img, num_steps=300,
                            style_weight=1000000, content_weight=1):
         print('Building the style transfer model..')
         model, style_losses, content_losses = get_style_model_and_losses(cnn, style_img, content_img)
         optimizer = optim.LBFGS([input_img.requires_grad_()])

         print('Optimizing..')
         run = [0]
         while run[0] <= num_steps:
             def closure():
                 input_img.data.clamp_(0, 1)  # Clamp values to valid range

                 optimizer.zero_grad()
                 model(input_img)
                 style_score = 0
                 content_score = 0

                 for sl in style_losses:
                     style_score += sl.loss
                 for cl in content_losses:
                     content_score += cl.loss

                 loss = style_score * style_weight + content_score * content_weight
                 loss.backward()

                 run[0] += 1
                 if run[0] % 50 == 0:
                     print("run {}:".format(run[0]))
                     print('Style Loss : {:4f} Content Loss: {:4f}'.format(
                         style_score.item(), content_score.item()))
                     print()

                 return style_score + content_score

             optimizer.step(closure)

         input_img.data.clamp_(0, 1)  # Clamp final output to valid range

         return input_img
```

9. This code first clones the content image to use as the initial input for style transfer. It then calls the `run_style_transfer` function to apply the style of the style image to the content image, resulting in the `output_img`. Finally, the `imshow` function is used to display the generated image with the applied style, showing the combined artistic effect of the style transfer process.

```
[15] # Clone content image to use as the initial input
     input_img = content_img.clone().to(device)

     # Perform the style transfer
     output_img = run_style_transfer(model, content_img, style_img, input_img)

     # Display the output image
     imshow(output_img, title='Output Image')
```

```
Building the style transfer model..
Optimizing..
run 50:
Style Loss : 0.000030 Content Loss: 11.676895

run 100:
Style Loss : 0.000015 Content Loss: 10.340895

run 150:
Style Loss : 0.000012 Content Loss: 9.438254

run 200:
Style Loss : 0.000010 Content Loss: 8.826137

run 250:
Style Loss : 0.000008 Content Loss: 8.447785

run 300:
Style Loss : 0.001597 Content Loss: 13.650244
```



Conclusion

The Neural Style Transfer project effectively demonstrates the integration of deep learning techniques to combine the content of one image with the artistic style of another. By leveraging a pre-trained VGG19 model, custom loss functions for content and style, and optimization strategies, the project achieved visually compelling results that blend artistic and structural elements from two distinct images. The use of PyTorch and Google Colab provided a robust environment for implementing and refining the style transfer process. This project not only showcases the practical application of neural networks in image processing but also highlights the capabilities of advanced deep learning techniques in creative and artistic domains.