Neural Style Transfer

Aryan Mishra

June 2024

Abstract

Neural Style Transfer (NST) is an advanced technique in artificial intelligence that merges the content of one image with the artistic style of another using convolutional neural networks. This report explores the implementation and evaluation of fast Neural Style Transfer using the TransformerNet architecture, highlighting its applications, methodology, results, and potential improvements. Through detailed analysis, the report demonstrates how NST can transform images, maintaining the structural elements of the content image while adopting the stylistic features of the reference image. Key findings include high-quality image generation and flexibility in style application, though challenges such as computational demands and handling complex styles remain.

1 Introduction

Neural Style Transfer (NST) is a fascinating intersection of art and technology. By leveraging the power of convolutional neural networks, NST allows for the creation of images that blend the content of one image with the stylistic elements of another. This capability opens up numerous possibilities in various fields, from artistic creation to design and augmented reality.

In this report, we focus on the implementation of fast Neural Style Transfer (NST) using the innovative TransformerNet architecture. Traditional NST methods rely on an iterative optimization process that can be computationally intensive and time-consuming, making them less practical for real-time applications. TransformerNet, in contrast, is designed to significantly speed up the style transfer process by generating the styled image in a single forward pass through the network. This approach leverages a feedforward neural network, which is trained to apply the artistic style to the content image efficiently. The architecture of TransformerNet consists of several convolutional layers that capture and transform the content and style features, ultimately producing a high-quality styled image in a fraction of the time required by traditional methods. This advancement opens up new possibilities for applications that require rapid image processing, such as video editing, real-time filters, and interactive design tools.

The training process for TransformerNet involves optimizing the network to minimize both content and style losses. Content loss ensures that the output image retains the essential structural elements of the original content image, while style loss captures the distinctive textures and color patterns of the reference style image. This dual-loss approach ensures that the generated image maintains a balance between accurately reflecting the original content and adopting the desired artistic style. The training dataset used for this implementation is the COCO 2017 dataset, specifically utilizing 40,000 test images. This extensive dataset includes a diverse range of images, enhancing the model's robustness and generalization capabilities. By training on such a comprehensive dataset, TransformerNet is well-equipped to effectively apply various artistic styles to a wide array of content images, making it a versatile tool for creative and practical applications in digital art and beyond.

2 Table of Contents

The Pipeline can be divided into following parts:

3 Requirements

pillow
torch
torchvision
tqdm
collections

4 Datasets Used

I have used COCO 2017 dataset for training the TransformerNet. The COCO (Common Objects in Context) dataset is a large-scale object detection, segmentation, and captioning dataset widely used in the computer vision community. By selecting 40,000 test images from the COCO 2017 dataset, we ensured a diverse and comprehensive training set that includes a wide range of content types and visual scenarios. This diversity is crucial for enhancing the robustness and generalization capabilities of the TransformerNet model.

8 Styles images were used to train TransformerNets. Each style need a saparate TransformetNet to train.

5 Data Preprocessing

5.1 Data Preprocessing for COCO 2017 Images

1. Data Loading :

```
def TrainImagesPath(path):
    images_path = []
    for file in os.listdir(path):
        if file == "test2017":
            file_path = os.path.join(path, file)
            for f in os.listdir(file_path):
                image_path = os.path.join(file_path, f)
                images_path.append(image_path)
                if len(images_path) == 40000:
                    break
    return images_path
```

Listing 1: Function to load Images paths

The code focuses on a specific folder named "test2017" within the provided path. This suggests it's only interested in a particular subset of images for training, possibly due to limitations or training efficiency.

2. Applying Transform to Image :

```
train_transform = transforms.Compose([
                                transforms.Resize((256, 256)),
                                transforms.CenterCrop((256, 256)),
                                transforms.ToTensor(),
                                transforms.Lambda(lambda x: x.mul(255))
                                        ])

```

Listing 2: Train Transform

By applying transforms.Resize and transforms.CenterCrop, it enforces a uniform size (256x256 pixels) on all images. This ensures the model receives data with consistent dimensions. The transforms.ToTensor function converts the loaded images (likely in PIL format) into tensors, which is a format commonly used by machine learning frameworks like PyTorch. A lambda function to multiply each pixel value by 255. This normalizes the pixel intensities to a range between 0 and 1, which can be beneficial for certain machine learning algorithms. This is a lambda function that takes a tensor x (representing an image) as input. The mul function (short for multiply) element-wise multiplies each value in the tensor by 255.

5.2 Data Preprocessing for Style Images

1. Data Loading :

```
def LoadStyleImage(path, batch_size, device):
    style = Image.open(path).convert('RGB')
    style = style.resize((256, 256))
    style = style_transform(style)
    style = style.repeat(batch_size, 1, 1, 1)
    style = style.to(device)
    return style

```

Listing 3: Train Transform

It opens the image from the provided path using Image.open and converts it to RGB format (if necessary) using convert('RGB'). This ensures the image has three channels (red, green, blue) expected by most image processing tasks. It resizes the image to a fixed size of (256, 256) pixels using style.resize((256, 256)). This ensures consistency with the model's expectations.It replicates the preprocessed style image batchsize times using style.repeat(batchsize, 1, 1, 1). This step might be necessary if the model processes multiple content images in a batch during training or inference. Finally, it transfers the processed style image tensor to the specified device (device) using .to(device). This is likely the CPU or GPU depending on your hardware configuration and computational needs.

2. Applying Transforms to style :

```
style_transform = transforms.Compose([
                                transforms.ToTensor(),
                                transforms.Lambda(lambda x: x.mul(255))
                                ])
```

Listing 4: Style Transform

Converts the PIL image format into a PyTorch tensor suitable for computations. Normalizes the pixel values between 0 and 255.

6 Models Description

6.1 ConvLayer

```
class ConvLayer(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, stride):
        super(ConvLayer, self).__init__()
        reflection_padding = kernel_size // 2
        self.reflection_pad = nn.ReflectionPad2d(reflection_padding)
        self.conv2d = nn.Conv2d(in_channels, out_channels, kernel_size,
    stride)

    def forward(self, x):
        out = self.reflection_pad(x)
        out = self.conv2d(out)
        return out
```

Listing 5: ConvLayer

This class represents a convolutional layer, a fundamental building block in Convolutional Neural Networks (CNNs). It performs the core operation of a CNN - extracting features from the input data using learnable filters (kernels).

6.2 ResidualBlock

```
class ResidualBlock(nn.Module):
    def __init__(self, channels):
        super(ResidualBlock, self).__init__()
        self.conv1 = ConvLayer(channels, channels, kernel_size=3, stride
    =1)
        self.in1 = nn.InstanceNorm2d(channels, affine=True)
        self.conv2 = ConvLayer(channels, channels, kernel_size=3, stride
    =1)
        self.in2 = nn.InstanceNorm2d(channels, affine=True)
        self.relu = nn.ReLU()

    def forward(self, x):
```

```
        out = self.relu(self.in1(self.conv1(x)))
        out = self.in2(self.conv2(out))
        out = out + x
        return out
```

Listing 6: ResidualBlock

This class defines a residual block, a commonly used architecture in deep residual networks. It introduces a "shortcut connection" that allows the gradient to flow directly through the layers, addressing the vanishing gradient problem that can hinder training deep neural networks.

6.3 UpsampleConvLayer

```
class UpsampleConvLayer(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, stride, upsample=None):
        super(UpsampleConvLayer, self).__init__()
        self.upsample = upsample
        reflection_padding = kernel_size // 2
        self.reflection_pad = nn.ReflectionPad2d(reflection_padding)
        self.conv2d = nn.Conv2d(in_channels, out_channels, kernel_size, stride)

    def forward(self, x):
        if self.upsample:
            x = nn.functional.interpolate(x, mode='nearest', scale_factor=self.upsample)
        out = self.reflection_pad(x)
        out = self.conv2d(out)
        return out
```

Listing 7: UpsampleConvLayer

This class defines a convolutional layer with upsampling capabilities. It's often used in decoder parts of networks where the goal is to increase the spatial resolution of the feature maps.

6.4 TransformerNet

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

        self.conv1 = ConvLayer(3, 32, kernel_size=9, stride=1)
        self.in1 = nn.InstanceNorm2d(32, affine=True)
        self.conv2 = ConvLayer(32, 64, kernel_size=3, stride=2)
        self.in2 = nn.InstanceNorm2d(64, affine=True)
```

```
        self.conv3 = ConvLayer(64, 128, kernel_size=3, stride=2)
        self.in3 = nn.InstanceNorm2d(128, affine=True)

        self.res1 = ResidualBlock(128)
        self.res2 = ResidualBlock(128)
        self.res3 = ResidualBlock(128)
        self.res4 = ResidualBlock(128)
        self.res5 = ResidualBlock(128)

        self.deconv1 = UpsampleConvLayer(128, 64, kernel_size=3, stride=1,
    upsample=2)
        self.in4 = nn.InstanceNorm2d(64, affine=True)
        self.deconv2 = UpsampleConvLayer(64, 32, kernel_size=3, stride=1,
   upsample=2)
        self.in5 = nn.InstanceNorm2d(32, affine=True)
        self.deconv3 = ConvLayer(32, 3, kernel_size=9, stride=1)

        self.relu = nn.ReLU()

    def forward(self, X):
        y = self.relu(self.in1(self.conv1(X)))
        y = self.relu(self.in2(self.conv2(y)))
        y = self.relu(self.in3(self.conv3(y)))
        y = self.res1(y)
        y = self.res2(y)
        y = self.res3(y)
        y = self.res4(y)
        y = self.res5(y)
        y = self.relu(self.in4(self.deconv1(y)))
        y = self.relu(self.in5(self.deconv2(y)))
        y = self.deconv3(y)
        return y
```

Listing 8: TransformerNet

It combines convolutional layers, residual blocks, and upsampling convolutions to achieve the desired style transfer effect.

7 Loss Functions

We define two perceptual loss functions that measure high-level perceptual and semantic differences between images. They make use of a loss network pretrained for image classification, meaning that these perceptual loss functions are themselves deep convolutional neural networks. In all our experiments is the 16-layer VGG network pretrained on the ImageNet dataset.

Combining the content of one image with the style of another by jointly minimizing the

feature reconstruction loss of and a style reconstruction loss also based on features extracted from a pretrained convolutional network.

- Feature Reconstruction Loss : Instead of focusing on making the output image look exactly like the target image, feature reconstruction loss encourages the model to capture similar underlying features.

  Imagine the target image and the output image as two different paintings trying to depict the same scene. Pixel-by-pixel comparison might not be ideal because the artist might use different colors or brushstrokes to achieve the same effect.

  Feature reconstruction loss works like comparing the "artistic style" of the paintings. We use a separate pre-trained network () to analyze both images and extract their key features. These features could be things like edges, shapes, textures, or color palettes.

  The loss function then calculates the difference between the extracted features of the target image and the output image. This difference tells us how well the model captured the essence of the target image, even if the exact pixel values are not identical.

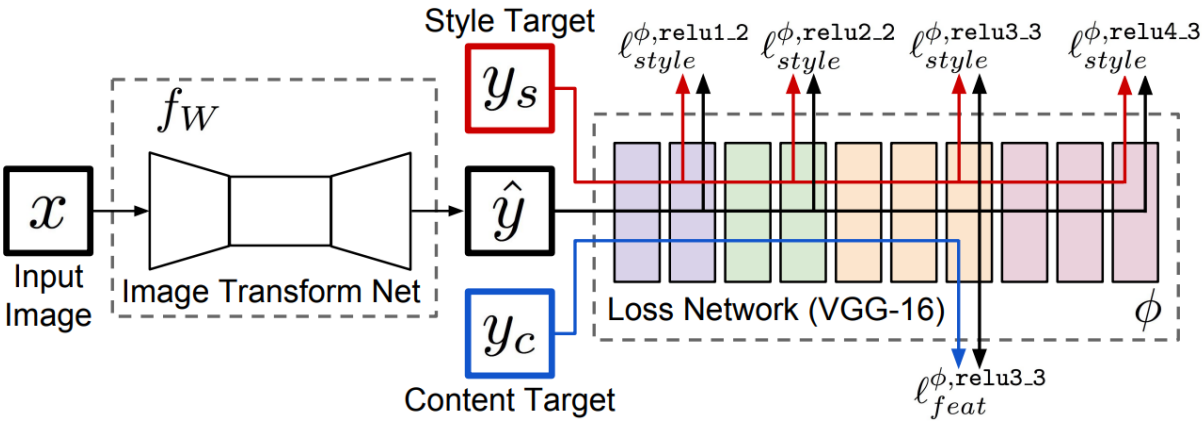

Figure 1: We use a loss network pretrained for image classification to define perceptual loss functions that measure perceptual differences in content and style between images. The loss network remains fixed during the training process.

- Style Reconstruction Loss : Style reconstruction loss works by analyzing the way features activate together in different layers of a pre-trained network (). This network acts like an "art detective," identifying patterns in how colors, textures, and other stylistic elements co-occur within the image.

  1. Gram Matrix: This captures the "co-occurrence" of features in a layer. High values indicate features often activate together, contributing to the overall style.
  2. Style Loss: It compares the Gram matrices of the generated image and the reference image. A large difference means the generated image doesn't capture the stylistic "fingerprint" well.

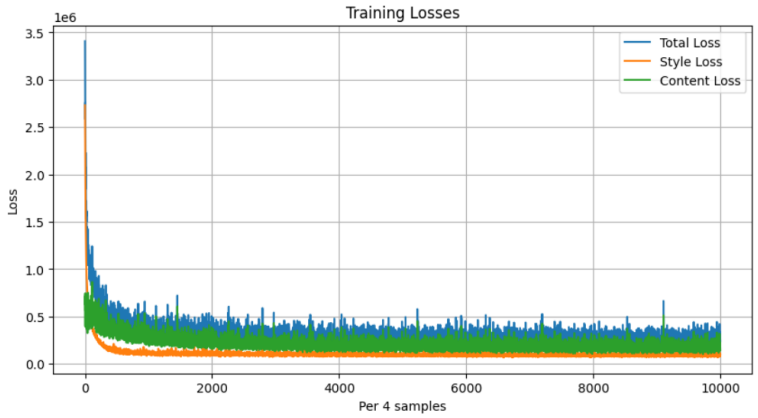

Figure 2: We use a loss network pretrained for image classification to define perceptual loss functions that measure perceptual differences in content and style between images. The loss network remains fixed during the training process.

3. Multiple Layers: Style loss can be applied to different layers in the network. Lower layers capture smaller-scale stylistic elements like brushstrokes, while higher layers capture larger-scale patterns like color harmony.

8 Model Training and Hyperparamets

All the model were trained on GPU P100 that is available on kaggle.

8.1 Training Loop

```
def TrainTransformerNet(style_path, vgg, device, train_dataloader,
    batch_size, epochs=1, lr=1e-3, content_weight = 1e5, style_weight = 1
    e10):

     style_image = LoadStyleImage(style_path, batch_size, device) ##
    Getting style image
     features_style = vgg(normalize_batch(style_image)) ## getting features
     maps of style image
     gram_style = [gram_matrix(y) for y in features_style] ## creating gram
     matrix of style features

     transformer = TransformerNet()
     transformer = transformer.to(device)
     optimizer = torch.optim.Adam(transformer.parameters(), lr=lr)
     criterion = nn.MSELoss()

     loss = []
     style_loss_ = []
     content_loss_ = []
```

```
15
16    for epoch in range(epochs):
17
18        transformer.train()
19        agg_content_loss = 0.
20        agg_style_loss = 0.
21
22        for x, _ in tqdm(train_dataloader):
23            optimizer.zero_grad()
24
25            x = x.to(device)
26            y = transformer(x)
27
28            y = normalize_batch(y)
29            x = normalize_batch(x)
30
31            features_y = vgg(y)
32            features_x = vgg(x)
33
34            content_loss = content_weight * criterion(features_y.relu2_2,
    features_x.relu2_2)
35
36            style_loss = 0.
37            n_batch = len(x)
38            for ft_y, gm_s in zip(features_y, gram_style):
39                gm_y = gram_matrix(ft_y)
40                style_loss += criterion(gm_y, gm_s[:n_batch, :, :])
41            style_loss *= style_weight
42
43            total_loss_b = content_loss + style_loss
44            total_loss_b.backward()
45            optimizer.step()
46
47            agg_content_loss += content_loss.item()
48            agg_style_loss += style_loss.item()
49
50            style_loss_.append(style_loss.item())
51            content_loss_.append(content_loss.item())
52            loss.append(style_loss.item() + content_loss.item())
53
54        avg_content_loss = agg_content_loss / len(train_dataloader)
55        avg_style_loss = agg_style_loss / len(train_dataloader)
56        avg_loss = avg_content_loss + avg_style_loss
57
58        print(f" {epoch +1} / {epochs} Style Loss : {avg_style_loss:.4f},
    Content Loss : {avg_content_loss:.4f}, Total Loss : {avg_loss:.4f}")
59
60
61    return transformer, loss, style_loss_, content_loss_
```

Listing 9: Training Loop

8.2 Hyperparameters

- Batch size : 4
- Content weight : 1e5
- Style weight : 1e10
- Learning rate : 0.001
- epochs : 1

9 Results

Figure 3: Style Image

Figure 4: Content Image

Figure 5: Stylized Image

Figure 6: Style Image

.pngFigure 7: Content Image

Figure 8: Stylized Image

10 Challenges Faced

- Due to limited GPU memory, training was not possible if the size of style image is more than 2MB so we should resize it to (256, 256).
- Each style requires a separate training, so its a time consuming process to get weights for 8 styles.

11 Limitations

- Hardware Constraints: Limited processing power on mobile devices may hinder performance.
- Generalizability Issues: Models might struggle with unseen images or diverse styles.

12 References

1. Perceptual Losses for Real-Time Style Transfer and Super-Resolution
2. A Neural Algorithm of Artistic Style