Neural Style Transfer (Gatys)

Implementing neural style transfer based on the classic "Gatys" method (Leon A. Gatys et al., *A Neural Algorithm of Artistic Style*). The idea is to take:

1. A **content image** (whose overall structure you want to preserve).
2. A **style image** (whose texture/colors/patterns you want to emulate).
3. A **generated image** (the output you iteratively optimize).

Using a pretrained CNN (e.g., VGG19), you compute two types of losses:

- **Content Loss**: Ensures the generated image's high-level features match those of the content image.
- **Style Loss**: Ensures the generated image's textures and patterns match those of the style image.

By minimizing the weighted sum of these losses with respect to the pixels of the generated image, you produce an output that preserves the main subject of the content image but is rendered with the textures/colors of the style image.

---

1. Overall Steps

1. **Load Pretrained Network**: Commonly, VGG19 (trained on ImageNet) is used because it captures robust feature representations at multiple layers.
2. **Pick Content/Style Layers**:
3. For **content**, typically use a deeper layer (e.g. `conv4_2` in VGG) to capture structural information.
4. For **style**, use multiple shallower and deeper layers (e.g., `conv1_1`, `conv2_1`, `conv3_1`, `conv4_1`, `conv5_1`) to capture textures across scales.
5. **Compute Losses**:
6. **Content Loss**: Compare the feature maps of the generated image vs. the content image in your chosen content layer(s).
7. **Style Loss**: Compare the Gram matrices (feature correlations) of the generated image vs. the style image in your chosen style layers.
8. **Optimize**:

9. Your “parameter” is the **pixel values of the generated image** (not the weights of the network).
10. Use gradient-based optimization (e.g., LBFGS or Adam) to minimize:
    [\mathcal{L} = \alpha \times \text{ContentLoss} + \beta \times \text{StyleLoss}]
11. After sufficient iterations, the generated image should visually blend the content’s structure with the style’s textures.

---

2. Example Implementation in PyTorch

Below is a condensed example. It follows the logic of the official PyTorch tutorial “Neural Transfer Using PyTorch,” but trimmed for brevity. You can adapt it for your own use.

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.models as models
import torchvision.transforms as transforms
from PIL import Image
import copy

# --------------------
# 1. Load & Preprocess Images
# --------------------
def image_loader(image_path, max_size=512, device='cpu'):
    # Desired size to keep memory in check
    transform = transforms.Compose([
        transforms.Resize((max_size, max_size)),  # or scale by min dimen-
sion
        transforms.ToTensor()
    ])
    image = Image.open(image_path)
    # Convert to [C,H,W] tensor
    image = transform(image).unsqueeze(0)
    return image.to(device, torch.float)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
content_img = image_loader("path/to/content.jpg", device=device)
style_img   = image_loader("path/to/style.jpg", device=device)

# The generated image starts as a copy of the content image (or noise).
generated_img = content_img.clone().requires_grad_(True)

# --------------------
# 2. Load VGG Network
# --------------------
# We'll use VGG19 pre-trained on ImageNet
vgg = models.vgg19(pretrained=True).features.to(device).eval()

# We’ll define layers of interest. For instance:
# content_layers = ['conv4_2']
# style_layers   = ['conv1_1','conv2_1','conv3_1','conv4_1','conv5_1']

# --------------------
# 3. Create a Model that Returns Intermediate Activations
# --------------------
# We'll capture layers as the image passes through VGG
# so we can compute style/content losses at the correct points.
class VGGExtractor(nn.Module):
    def __init__(self, style_layers, content_layers):
        super(VGGExtractor, self).__init__()
        self.style_layers = style_layers
        self.content_layers = content_layers

                self.model  =  nn.ModuleList()    #  We'll  add  sub-layers
(conv/ReLU/pool)
        self.style_outputs = {}
        self.content_outputs = {}

        conv_num = 0
        # VGG has a sequence of blocks, let's register them:
        for layer in vgg.children():
            self.model.append(layer)

        # We'll store the outputs from specific layers by name
        self.layer_names = []
        conv_count = 0
        # We'll flatten the structure into a single sequence
        # and note the 'layer name' so we can match style/content layers
        # This is not the only way to do it, just an example approach.
        # e.g. 'conv1_1', 'conv1_2', 'relu1_1', etc.
        self.names_map = {}
        current_block = 1
        current_conv = 1
        for i, layer in enumerate(self.model):
            if isinstance(layer, nn.Conv2d):
                name = f"conv{current_block}_{current_conv}"
                self.names_map[i] = name
                current_conv += 1
            elif isinstance(layer, nn.ReLU):
                name = f"relu{current_block}_{current_conv-1}"
                self.names_map[i] = name
                # Convert in-place ReLU to out-of-place for correct gradi-
ent
                self.model[i] = nn.ReLU(inplace=False)
            elif isinstance(layer, nn.MaxPool2d):
                name = f"pool{current_block}"
                self.names_map[i] = name
                current_block += 1
                current_conv = 1

    def forward(self, x):
        style_outputs = {}
        content_outputs = {}

        for i, layer in enumerate(self.model):
            x = layer(x)
            layer_name = self.names_map.get(i, None)
            if layer_name in self.style_layers:
                style_outputs[layer_name] = x
            if layer_name in self.content_layers:
                content_outputs[layer_name] = x
        return style_outputs, content_outputs

# Instantiate our feature extractor
content_layers = ['conv4_2']
style_layers   = ['conv1_1','conv2_1','conv3_1','conv4_1','conv5_1']
extractor = VGGExtractor(style_layers, content_layers).to(device)
```

```
# --------------------
# 4. Define Style/Content Loss Computations
# --------------------
def gram_matrix(tensor):
    # tensor: B x C x H x W
    b, c, h, w = tensor.size()
    features = tensor.view(b*c, h*w)
    # Gram matrix: matrix multiplication of feature vectors
    G = torch.mm(features, features.t())
    return G.div(b*c*h*w)

# Get target features once (no grad) from style/content images
with torch.no_grad():
    style_feats, content_feats = extractor(style_img)
    style_grams = {ly: gram_matrix(style_feats[ly]) for ly in style_feats}
    content_feats, _ = extractor(content_img)

# We will use a dictionary to store the "ideal" feature maps
target_content_feats = {ly: content_feats[ly] for ly in content_feats}

# Style/content weights
style_weight = 1e6
content_weight = 1

# --------------------
# 5. Optimize the Generated Image
# --------------------
optimizer = optim.LBFGS([generated_img])

num_steps = 300
step = [0]  # mutable integer so LBFGS sees it in scope

def closure():
    with torch.no_grad():
        generated_img.clamp_(0, 1)  # keep image in [0,1]

    optimizer.zero_grad()

    style_out, content_out = extractor(generated_img)
    # Content loss
    content_loss = 0
    for ly in content_out:
        content_loss += torch.nn.functional.mse_loss(
            content_out[ly], target_content_feats[ly]
        )

    # Style loss
    style_loss = 0
    for ly in style_out:
        gen_gram = gram_matrix(style_out[ly])
        style_gram = style_grams[ly]
        style_loss += torch.nn.functional.mse_loss(gen_gram, style_gram)

    total_loss = content_weight*content_loss + style_weight*style_loss
    total_loss.backward()

    if step[0] % 50 == 0:
        print(f"Iteration {step[0]}/{num_steps}, "
              f"Total loss: {total_loss.item():.4f}")
    step[0] += 1
    return total_loss

for i in range(num_steps):
    optimizer.step(closure)

# Clamp final image
with torch.no_grad():
    generated_img.clamp_(0,1)

# --------------------
# 6. Save or Display Results
# --------------------
def save_image(tensor, path):
    # convert tensor [1,C,H,W] -> PIL Image
    image = tensor.cpu().clone().squeeze(0)
    image = transforms.ToPILImage()(image)
    image.save(path)

save_image(generated_img, "stylized_output.jpg")
```

Explanation:

1. **Image Loading**: We scale images to a manageable size (e.g. 512×512) and load them as tensors on the same device as VGG.
2. **Feature Extraction**: We pass images through VGG19's layers. We save the activations at specific layers for content and style comparisons.
3. **Content Loss**: MSE between feature maps of the generated image and the content image at the chosen layer(s).
4. **Style Loss**: MSE between the Gram matrices of the generated image and the style image across the chosen style layers. The Gram matrix captures correlations between feature channels.
5. **Optimization**: Rather than updating network weights, we treat the generated image as the parameter to be optimized. We typically use LBFGS or Adam for a few hundred iterations.
6. **Save Result**: The final image is a combination of the content structure and style patterns.

---

3. Key Tips & Variations

1. **Layer Selection**:
2. Generally, deeper layers capture *content* better (e.g. `conv4_2` or `relu4_2`).
3. Multiple shallow & deep layers are used for *style* to capture fine and coarse textural details.
4. **Weights (α and β)**:
5. If the stylized result is too close to the content image, increase the style weight.
6. If it's overly stylized (content lost), decrease the style weight.
7. Balancing these hyperparameters is part art, part experimentation.
8. **Preprocessing**:
9. VGG is typically trained on ImageNet means/standard deviations, so normalizing your inputs using those statistics (mean = [0.485, 0.456, 0.406], std = [0.229, 0.224, 0.225]) can help.
10. Make sure to invert it if you want to visualize or save the final image properly.
11. **Initialization**:
12. Starting the generated image as a copy of the content image often converges faster.
13. Alternatively, random noise can be used if you want a more dramatic transformation (but it may take more iterations).
14. **Speed-Ups**:

15. Running on a GPU is almost mandatory for higher resolutions.
16. Use smaller image sizes for faster iteration.
17. There are also “fast style transfer” models that train a feed-forward network to approximate this process.
18. **Advanced Approaches**:
19. **Fast Style Transfer** (Johnson et al., 2016) or **Arbitrary Style Transfer** (Huang et al., 2017) train specialized networks for near-real-time style transfer.
20. **Multiple Styles**: You can combine multiple style images, or even do style interpolation.

---

4. Conclusion

Neural style transfer is an **optimization-based** technique that transforms a content image to reflect the texture/color “style” of a reference image. By leveraging powerful pretrained CNNs (like VGG19) to compute content and style losses, you can iteratively refine the generated image to strike a balance between preserving content and emulating style. Once you understand the basics (content/style layers, Gram matrices, and iterative image optimization), you can adapt the technique for a wide variety of creative effects.