***reate an Art with Neural Style Transfer using Deep Learning***

**1. Introduction**

**Neural Style Transfer (NST)** is a deep learning technique where we take two images:

* A **Content Image** (e.g., a photo of the Eiffel Tower)
* A **Style Image** (e.g., an oil painting)

and **blend them** together, so that the output image:

* **Preserves the content** of the original photo
* **Takes on the artistic style** (colors, textures) of the painting

This is achieved by using **Convolutional Neural Networks (CNNs)**, especially **VGG-19**.

**2. Key Concepts**

**2.1 Content Representation**

* Content represents **the objects, layout, and structure** of an image.
* In CNNs, deeper layers capture **high-level content features**.
* In NST, **we extract content features** from specific layers (like conv2\_1) of VGG-19.

**2.2 Style Representation**

* Style represents **textures, colors, and patterns**, *irrespective of object positions*.
* In CNNs, **style is captured** by computing the **correlation** between different feature maps.
* This correlation is calculated using the **Gram Matrix**:

Gram Matrix=Feature Maps×(Feature Maps)T\text{Gram Matrix} = \text{Feature Maps} \times (\text{Feature Maps})^TGram Matrix=Feature Maps×(Feature Maps)T

* **Multiple layers** (like conv1\_1, conv2\_1) are used to capture fine to coarse style features.

**2.3 Loss Function**

The total loss in Neural Style Transfer is:

Total Loss=Content Loss+Style Loss\text{Total Loss} = \text{Content Loss} + \text{Style Loss}Total Loss=Content Loss+Style Loss

* **Content Loss** measures how different the content of the generated image is from the content image.
* **Style Loss** measures how different the style of the generated image is from the style image.

We balance these two using **weights**:

* content\_weight (usually small)
* style\_weight (usually large)

In the code:

* style\_weight = 1e6
* content\_weight = 1

**2.4 Optimization**

* Instead of training a new model, we **treat the input image as a variable**.
* We **optimize the pixels** of the input image directly to minimize the loss.
* An optimizer like **Adam** is used to adjust the pixel values.

**3. Important Components**

| **Component** | **Description** |
| --- | --- |
| **VGG-19** | Pre-trained deep CNN used for feature extraction (not training, only feature extraction). |
| **Content Layer** | Layer to extract content features (e.g., conv2\_1). |
| **Style Layers** | Layers to extract style features (e.g., conv1\_1, conv2\_1). |
| **Gram Matrix** | Captures texture information by correlating feature maps. |
| **Optimizer** | Adam optimizer updates the target image pixels. |
| **PyTorch Framework** | Used for tensor operations, model loading, and backpropagation. |

**4. Flow of the Practical**

1. **Load the content and style images**.
2. **Preprocess**: Resize, normalize images according to VGG-19 input requirements.
3. **Load VGG-19** pretrained model.
4. **Extract content and style features** from selected layers.
5. **Initialize the target image** (copy of content image).
6. **Calculate Content Loss** between target and content features.
7. **Calculate Style Loss** between target's and style's Gram Matrices.
8. **Compute total loss** (weighted sum of content and style losses).
9. **Backpropagate** and **update** the target image pixels to minimize loss.
10. **Display** the final stylized image.

**5. Why Normalize the Images?**

VGG models were trained on images normalized with:

* **Mean** = [0.485, 0.456, 0.406]
* **Std** = [0.229, 0.224, 0.225]

So we must **normalize the input images** similarly, to get correct feature maps.

**6. Why Use Pretrained VGG-19?**

* VGG-19 is **excellent at extracting features** from images.
* It is **already trained** on **ImageNet** (millions of images), so it captures rich features.
* We **freeze** VGG-19’s weights and **only update the target image**.

**7. Advantages of Neural Style Transfer**

* Allows **artistic image creation** without needing to train models from scratch.
* High-quality **photo-to-art transformations**.
* Works even with **small datasets** (only two images needed: one content, one style).

**8. Applications of Neural Style Transfer**

| **Area** | **Examples** |
| --- | --- |
| **Art and Design** | Create paintings, digital artwork automatically. |
| **Entertainment** | Stylizing movie frames or animation scenes. |
| **Marketing** | Artistic advertisements from real photos. |
| **Fashion** | Generate new textile designs using style transfer. |
| **Augmented Reality** | Stylized filters in real-time camera apps. |

**9. Important Terms Glossary**

| **Term** | **Meaning** |
| --- | --- |
| **Content Image** | Photo providing structure and shapes |
| **Style Image** | Artwork providing texture and color |
| **Stylized Image** | Output combining both content and style |
| **Feature Map** | Output from CNN layers |
| **Gram Matrix** | Matrix showing feature correlations (texture info) |
| **Loss Function** | Measures how good the output is compared to targets |

**📋 Summary of Practical:**

* Use VGG-19 to extract features from content and style images.
* Minimize a loss function combining content and style losses.
* Optimize the pixels of the input image itself using backpropagation.
* Result: A new image that looks like the content photo but painted in the style of the artwork!

**Full Code Explanation (Line-by-Line)**

**1. Import Required Libraries**

python

CopyEdit

import torch

import torch.nn as nn

import torch.optim as optim

from torchvision import transforms, models

from PIL import Image

import matplotlib.pyplot as plt

✅ **Explanation:**

* torch: The main PyTorch library for tensor operations.
* torch.nn: For neural network operations (like building models, loss functions).
* torch.optim: Provides optimizers (like Adam) to update tensors.
* torchvision.transforms: Contains utilities for image preprocessing (resize, normalize, etc.).
* torchvision.models: Gives access to pretrained models like VGG-19.
* PIL.Image: For loading and handling images.
* matplotlib.pyplot: For displaying images.

**2. Set Device (CPU)**

python

CopyEdit

device = torch.device("cpu") # Use CPU to avoid GPU crashes

✅ **Explanation:**

* It specifies that the model and tensors will be processed on **CPU**.
* Useful when you don’t have a GPU or want to avoid GPU memory issues.

**3. Function to Load and Preprocess an Image**

python

CopyEdit

def load\_image(path, max\_size=256):

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

transform = transforms.Compose([

transforms.Resize((max\_size, max\_size)),

transforms.ToTensor(),

transforms.Normalize([0.485, 0.456, 0.406],

[0.229, 0.224, 0.225])

])

return transform(image).unsqueeze(0).to(device)

✅ **Explanation:**

* path: Path to the image file.
* Opens the image and converts it to **RGB**.
* **Preprocessing:**
  + Resize the image to **256x256**.
  + Convert image to **Tensor** (PyTorch format).
  + **Normalize** using ImageNet mean and standard deviation (required for VGG-19).
* unsqueeze(0): Adds a fake **batch dimension** (important: VGG expects batches).
* to(device): Moves the tensor to CPU.

**4. Function to Convert Tensor Back to Image**

python

CopyEdit

def im\_convert(tensor):

image = tensor.clone().detach().cpu().squeeze(0)

image = image \* torch.tensor([0.229, 0.224, 0.225]).view(3,1,1) + \

torch.tensor([0.485, 0.456, 0.406]).view(3,1,1)

return transforms.ToPILImage()(image.clamp(0,1))

✅ **Explanation:**

* clone().detach(): Creates a new tensor without gradient tracking.
* cpu(): Moves tensor back to CPU if needed.
* squeeze(0): Removes batch dimension.
* **Denormalization**:
  + Multiply by std, add mean to undo normalization.
* clamp(0,1): Ensures pixel values are between 0 and 1.
* ToPILImage(): Converts tensor back to a PIL image so it can be displayed.

**5. Load Content and Style Images**

python

CopyEdit

content = load\_image("Eiffel Tower.jpg", max\_size=256)

style = load\_image("Oil Painting.jpg", max\_size=256)

✅ **Explanation:**

* Load the **content image** (e.g., photo of Eiffel Tower).
* Load the **style image** (e.g., oil painting).
* Both are resized and preprocessed into tensors.

**6. Load Pretrained VGG-19 Model**

python

CopyEdit

vgg = models.vgg19(pretrained=True).features.to(device).eval()

✅ **Explanation:**

* Load **VGG-19** model pretrained on ImageNet.
* .features means **only convolutional layers** (no classifier).
* .to(device): Moves model to CPU.
* .eval(): Puts model in **evaluation mode** (deactivates dropout, batch norm).

**7. Freeze VGG Parameters**

python

CopyEdit

for p in vgg.parameters():

p.requires\_grad = False

✅ **Explanation:**

* Ensures **VGG weights will not change** during optimization.
* We are only updating the **target image**, not training the model.

**8. Select Layers for Features**

python

CopyEdit

layer\_map = {'0': 'conv1\_1', '5': 'conv2\_1'}

content\_layer = 'conv2\_1'

style\_layers = ['conv1\_1', 'conv2\_1']

✅ **Explanation:**

* VGG-19 has many layers (indexed as strings '0', '5', etc.).
* **layer\_map**: Maps specific VGG layers to human-readable names.
* We extract:
  + **Content** from conv2\_1
  + **Style** from conv1\_1 and conv2\_1

**9. Function to Get Features from VGG**

python

CopyEdit

def get\_features(x):

features = {}

for name, layer in vgg.\_modules.items():

x = layer(x)

if name in layer\_map:

features[layer\_map[name]] = x

return features

✅ **Explanation:**

* Pass input x through VGG layers one-by-one.
* Whenever the layer's index is in layer\_map, **save its output**.
* **Output** is a dictionary containing the selected feature maps.

**10. Function to Compute Gram Matrix**

python

CopyEdit

def gram\_matrix(tensor):

b, c, h, w = tensor.size()

tensor = tensor.view(c, h \* w)

return torch.mm(tensor, tensor.t())

✅ **Explanation:**

* Reshapes the feature tensor to size (channels, height\*width).
* Computes **Gram Matrix**: measures how feature maps correlate (texture representation).

**11. Extract Content and Style Features**

python

CopyEdit

content\_feat = get\_features(content)

style\_feat = get\_features(style)

style\_grams = {l: gram\_matrix(style\_feat[l]) for l in style\_layers}

✅ **Explanation:**

* Extract content features from the content image.
* Extract style features from the style image.
* For each style feature, compute the **Gram Matrix**.

**12. Initialize Target Image**

python

CopyEdit

target = content.clone().requires\_grad\_(True)

✅ **Explanation:**

* Copy the content image as the **starting point** for the target.
* requires\_grad\_(True): We will update this image’s pixel values using backpropagation.

**13. Define Optimizer**

python

CopyEdit

optimizer = optim.Adam([target], lr=0.005)

✅ **Explanation:**

* Use **Adam** optimizer to update the **target image**.
* Learning rate (lr) is set to 0.005.

**14. Set Loss Weights**

python

CopyEdit

style\_weight, content\_weight = 1e6, 1

✅ **Explanation:**

* style\_weight = 1e6: Gives very high importance to style.
* content\_weight = 1: Lower importance to content.

**15. Optimization Loop**

python

CopyEdit

for i in range(50): # fewer epochs

✅ **Explanation:**

* Perform **50 iterations** to optimize the target image.

**Inside the Loop:**

**a. Extract Target Features**

python

CopyEdit

target\_feat = get\_features(target)

* Extract content and style features from the current **target image**.

**b. Compute Content Loss**

python

CopyEdit

content\_loss = torch.mean((target\_feat[content\_layer] - content\_feat[content\_layer])\*\*2)

* Compare target content features with original content features.

**c. Compute Style Loss**

python

CopyEdit

style\_loss = 0

for l in style\_layers:

target\_gram = gram\_matrix(target\_feat[l])

style\_gram = style\_grams[l]

style\_loss += torch.mean((target\_gram - style\_gram)\*\*2)

* For each style layer:
  + Compute Gram Matrix of target features.
  + Compare with Gram Matrix of style features.

**d. Compute Total Loss**

python

CopyEdit

total\_loss = style\_weight \* style\_loss + content\_weight \* content\_loss

* Combine style loss and content loss with appropriate weights.

**e. Backpropagate and Update**

python

CopyEdit

optimizer.zero\_grad()

total\_loss.backward()

optimizer.step()

* Zero the gradients.
* Perform **backpropagation** to compute gradients.
* Update the target image using **Adam optimizer**.

**f. Print Progress**

python

CopyEdit

if (i+1) % 10 == 0:

print(f"Step {i+1}, Loss: {total\_loss.item():.2f}")

* Print the loss every 10 steps.

**16. Display the Stylized Image**

python

CopyEdit

plt.imshow(im\_convert(target))

plt.axis("off")

plt.title("Stylized Image")

plt.show()

✅ **Explanation:**

* Convert the target tensor back into a PIL image.
* Display the final **stylized image** using matplotlib.

**✨ Conclusion:**

* This code successfully takes a **content image** and a **style image**, and blends them together using **Neural Style Transfer**!
* It leverages a **pretrained VGG-19 model** and uses **loss functions** to create a **new artwork**.

**🔥 Revision Pointers (Quick):**

| **Concept** | **Short Note** |
| --- | --- |
| Content Loss | Measures structure similarity |
| Style Loss | Measures texture similarity |
| Gram Matrix | Correlation of feature maps |
| Target Image | Starts as content image, optimized to match style |
| Optimizer | Adam optimizer tunes pixel values |