To implement a neural style transfer model, we will need to follow a sequence of steps to apply artistic styles to photographs. Below is an outline of the code along with an example of how to use it in a Python script or Jupyter notebook.

We will be using **TensorFlow** and **Keras** for this implementation. The neural style transfer works by combining a content image (photograph) and a style image (artistic painting) and generating a new image that maintains the content of the photograph but adopts the artistic style.

Step-by-Step Guide to Neural Style Transfer

1. Install Required Libraries

First, install the required libraries:

```
pip install tensorflow numpy matplotlib pillow
```

2. Import Libraries

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.preprocessing import image as kp_image
from tensorflow.keras.applications.vgg19 import VGG19, preprocess_input
from tensorflow.keras import models
from tensorflow.keras.layers import Input
from tensorflow.keras import backend as K
import PIL.Image as PILImage
```

3. Load and Preprocess the Images

```
def load_img(img_path, max_dim=512):
    img = PILImage.open(img_path)
    long = max(img.size)
    scale = max_dim / long
    new_size = tuple([int(dim * scale) for dim in img.size])
    img = img.resize(new_size, PILImage.LANCZOS)

    # Preprocess image: Convert to array and add batch dimension
    img_array = np.array(img)
    img_array = np.expand_dims(img_array, axis=0)
    return img_array

def preprocess_img(img_array):
    # Preprocess image for VGG19
    return preprocess_input(img_array)

def deprocess_img(img_array):
    # Remove the preprocessing done for VGG19
    img_array = img_array.reshape((img_array.shape[1], img_array.shape[2],
3))
    img_array = img_array + [103.939, 116.779, 123.68]
    img_array = img_array[..., ::-1]
    return np.clip(img_array, 0, 255).astype('uint8')
```

4. Build the VGG19 Model for Feature Extraction

```
def get_vgg_model():
```

```
    # Load VGG19 model pre-trained on ImageNet
    vgg = VGG19(include_top=False, weights='imagenet')
    vgg.trainable = False

    # Create a model that outputs the features of intermediate layers
    content_layers = ['block5_conv2']
    style_layers = ['block1_conv1', 'block2_conv1', 'block3_conv1',
'block4_conv1', 'block5_conv1']

    # Create the model by extracting outputs from the layers of interest
    outputs = [vgg.get_layer(name).output for name in content_layers +
style_layers]
    model = models.Model(inputs=vgg.input, outputs=outputs)

    return model, content_layers, style_layers
```

5. Compute the Content and Style Loss

```
def content_loss(base_content, target_content):
    return tf.reduce_sum(tf.square(base_content - target_content))

def gram_matrix(x):
    # Calculate the Gram matrix for style
    x = tf.transpose(x, perm=[2, 0, 1])
    x = tf.reshape(x, (x.shape[0], -1))
    return tf.matmul(x, x, transpose_b=True)

def style_loss(base_style, target_style):
    # Calculate the style loss using the Gram matrix
    gram_base = gram_matrix(base_style)
    gram_target = gram_matrix(target_style)
    return tf.reduce_sum(tf.square(gram_base - gram_target))

def total_variation_loss(x):
    # Regularize image with total variation loss to smooth the output
    a = tf.square(x[:, :-1, :-1, :] - x[:, 1:, :-1, :])
    b = tf.square(x[:, :-1, :-1, :] - x[:, :-1, 1:, :])
    return tf.reduce_sum(tf.pow(a + b, 1.25))
```

6. Define the Optimization Process

```
def compute_loss(model, loss_weights, init_image, content_image,
style_image):
    # Extract model outputs
    model_outputs = model(init_image)

    # Separate content and style outputs
    content_output = model_outputs[0]
    style_outputs = model_outputs[1:]

    # Compute content and style losses
    content_loss_value = content_loss(content_output, content_image)

    style_loss_value = 0
    for style_output, target_style in zip(style_outputs, style_image):
        style_loss_value += style_loss(style_output, target_style)

    # Compute total loss with weights
```

```
    total_loss = loss_weights[0] * content_loss_value + loss_weights[1] *
style_loss_value + loss_weights[2] * total_variation_loss(init_image)

    return total_loss, content_loss_value, style_loss_value
```

7. Generate the Stylized Image

```
def run_style_transfer(content_image_path, style_image_path, iterations=1000,
max_dim=512):
    content_image = load_img(content_image_path, max_dim)
    style_image = load_img(style_image_path, max_dim)

    # Preprocess images
    content_image = preprocess_img(content_image)
    style_image = preprocess_img(style_image)

    # Get the VGG model
    model, content_layers, style_layers = get_vgg_model()

    # Initialize the target image (styled image)
    init_image = tf.Variable(content_image, dtype=tf.float32)

    # Define optimizer
    optimizer = tf.optimizers.Adam(learning_rate=10.0)

    # Loss weights
    loss_weights = (1.0, 1e4, 30.0)  # Content, Style, TV

    # Start optimization
    for i in range(iterations):
        with tf.GradientTape() as tape:
            loss, content_loss_value, style_loss_value = compute_loss(model,
loss_weights, init_image, content_image, style_image)

        grads = tape.gradient(loss, init_image)
        optimizer.apply_gradients([(grads, init_image)])

        if i % 100 == 0:
            print(f"Iteration {i}, Total Loss: {loss.numpy()}, Content Loss:
{content_loss_value.numpy()}, Style Loss: {style_loss_value.numpy()}")
            plt.imshow(deprocess_img(init_image.numpy()))
            plt.show()

    return deprocess_img(init_image.numpy())
```

8. Example Usage

```
content_image_path = 'path_to_your_content_image.jpg'
style_image_path = 'path_to_your_style_image.jpg'

# Run style transfer
styled_image = run_style_transfer(content_image_path, style_image_path)

# Save the result
PILImage.fromarray(styled_image).save('styled_image.jpg')
```

Explanation

1. **Load Images**: The images are loaded and resized to fit within the maximum dimension.
2. **Preprocessing**: The images are processed for the VGG19 model, which includes normalizing the image pixel values.
3. **VGG19 Model**: We load the VGG19 model pre-trained on ImageNet and create a new model that outputs the features from the layers we are interested in.
4. **Loss Functions**: We define content loss (difference in feature maps), style loss (difference in Gram matrices), and total variation loss (to reduce noise).
5. **Optimization**: We use an Adam optimizer to minimize the total loss, which combines content, style, and total variation losses. At each iteration, we update the target image.
6. **Visualization**: Every 100 iterations, the intermediate results are visualized to observe the progress.

9. Result

Once the script runs, you'll get an image where the content is from the content photo and the style is derived from the artistic image.

You can adjust `iterations`, `loss_weights`, and `max_dim` to control the final result. The higher the iteration count, the better the result (although more computational time will be needed).

This approach works on any two images and allows you to create artistic transformations using neural networks.

Certainly! Here's a detailed report on how to implement **Neural Style Transfer** using Python and TensorFlow, including a description of the key concepts, methods, and steps involved.

Neural Style Transfer: Detailed Report

1. Introduction to Neural Style Transfer

Neural Style Transfer (NST) is a deep learning technique used to apply the artistic style of one image to the content of another image. This method, developed by Gatys et al. (2015), involves using pre-trained convolutional neural networks (CNNs), such as VGG19, to extract feature representations of content and style. The aim is to combine the content of one image with the style of another, generating a new image that merges both attributes.

This technique can be used to transform photos into artistic images by applying the style of famous paintings or any other artistic work to a photograph.

2. How Neural Style Transfer Works

NST works by defining a loss function that quantifies the difference between the content and style of the target image and the original content and style images. This loss function is then minimized using optimization methods (like gradient descent), resulting in a stylized image.

Key components involved in NST:

- **Content Image**: The image whose content you want to preserve (e.g., a photograph).
- **Style Image**: The image from which you want to extract the artistic style (e.g., a famous painting).
- **Output Image**: The generated image that combines the content of the content image with the style of the style image.

3. Steps in Implementing Neural Style Transfer

1. **Load and Preprocess Images**: The content and style images need to be loaded, resized to a manageable size, and preprocessed for use with the VGG19 model. This involves normalizing pixel values so they match the expected input format for the VGG19 network.
2. **VGG19 Model for Feature Extraction**: We use a pre-trained VGG19 model, which is a deep CNN trained on the ImageNet dataset. VGG19 is ideal because it has been shown to capture hierarchical representations of images, making it effective for extracting content and style features.
3. **Content and Style Representation**:
   - **Content Representation**: The content image is represented by the activations (feature maps) from the last convolutional layer in VGG19 (e.g., `block5_conv2`).
   - **Style Representation**: The style image is represented by the Gram matrix of the activations from the convolutional layers of VGG19 (e.g., `block1_conv1`, `block2_conv1`, etc.).
4. **Loss Functions**:
   - **Content Loss**: This is the difference between the content of the generated image and the content image.
   - **Style Loss**: This is the difference between the style of the generated image and the style image, computed using the Gram matrix.
   - **Total Variation Loss**: This regularizes the image to reduce noise and ensure the generated image does not have high-frequency artifacts.
5. **Optimization**: We initialize the output image with the content image, and then iteratively adjust it using gradient descent to minimize the loss function. The Adam optimizer is commonly used for this task.
6. **Final Output**: After a specified number of iterations, the algorithm generates the stylized image that combines the content of the content image with the style of the style image.

4. Key Concepts and Techniques

- **Gram Matrix**: A mathematical concept used to capture the correlations between different filter responses in a layer. It is particularly useful for representing the style of an image. The Gram matrix is computed as the product of the feature map matrix and its transpose.
- **Pre-trained Networks**: VGG19 is a popular pre-trained CNN used for image feature extraction. Since training a neural network from scratch can be computationally expensive, using a pre-trained model allows us to leverage learned features from large datasets like ImageNet.
- **Gradient Descent**: This optimization technique adjusts the pixel values of the generated image to minimize the total loss function. It iteratively moves in the direction that reduces the error between the generated image's content/style features and those of the original images.

5. Detailed Implementation of Neural Style Transfer

Here's a detailed breakdown of the code implementation, step by step:

5.1. Load and Preprocess the Images

We start by loading and resizing the content and style images. We also preprocess them by converting them into the format expected by the VGG19 model. Specifically:

- Convert images to arrays.
- Normalize the pixel values to the range that VGG19 expects.

The helper functions `load_img`, `preprocess_img`, and `deprocess_img` handle these steps.

5.2. VGG19 Model for Feature Extraction

We load the VGG19 model pre-trained on the ImageNet dataset, remove the classification layers, and use it to extract the feature maps from intermediate convolutional layers. These feature maps represent different levels of abstraction in the image and are used to compute content and style losses.

5.3. Content and Style Loss

- **Content Loss**: This loss function compares the feature maps of the content image and the generated image. The goal is to minimize the difference between their feature activations at a specific layer, typically the last convolutional layer of the model.
- **Style Loss**: This loss compares the style of the style image and the generated image. It is based on the Gram matrix of the feature maps. The Gram matrix captures the correlations between different feature channels. The style loss is computed as the difference between the Gram matrices of the style and generated images.

- **Total Variation Loss**: This loss is a regularization technique that helps reduce noise by ensuring that adjacent pixels in the generated image are not drastically different. This encourages the generated image to be smooth and less pixelated.

5.4. Optimization

The goal of the optimization process is to adjust the pixels of the generated image to minimize the combined loss function. The Adam optimizer is used to update the image pixels iteratively.

The optimization loop runs for a specified number of iterations, and at each step, the total loss is computed, gradients are calculated, and the image is updated.

5.5. Displaying and Saving the Result

After each iteration (or every few iterations), we visualize the intermediate result by converting the tensor back to an image. Once the optimization is complete, the final stylized image is saved and displayed.

6. Code Explanation and Example

1. **Load Content and Style Images**:
   - Load and resize images to a maximum dimension (e.g., 512 pixels).
   - Preprocess them for use with the VGG19 model.
2. **Define VGG19 Model**:
   - Load the VGG19 model without the fully connected layers.
   - Extract feature maps from the relevant layers for both content and style.
3. **Loss Functions**:
   - Compute the content loss as the L2 norm of the difference between content feature maps.
   - Compute the style loss by calculating the difference between the Gram matrices of the style and generated images.
4. **Optimization Loop**:
   - Minimize the total loss using the Adam optimizer.
5. **Visualize and Save**:
   - Display the generated image at regular intervals.
   - Save the final image after the optimization loop completes.

7. Conclusion

Neural Style Transfer is a powerful technique that allows us to create visually stunning images by combining the content of one image with the artistic style of another. By leveraging pre-trained models like VGG19 and using advanced optimization techniques, we can achieve impressive results. The process involves calculating content and style losses, using gradient descent to minimize these losses, and generating a new image that reflects the desired artistic transformation.

The code implementation presented here demonstrates how to effectively use TensorFlow and Keras to perform neural style transfer and generate stylized images. By adjusting various parameters such as the number of iterations and loss weights, you can fine-tune the results to your liking.

Certainly! Below is an outline of a 1000-word essay that provides instructions for implementing a Neural Style Transfer (NST) model in Python to apply artistic styles to photographs, followed by a script that demonstrates this implementation.

Neural Style Transfer: Applying Artistic Styles to Photographs

Introduction to Neural Style Transfer

Neural Style Transfer (NST) is a deep learning technique used to combine the content of one image (usually a photograph) with the artistic style of another image (often a painting). The core idea behind NST is to use convolutional neural networks (CNNs) to extract the content and style from separate images, and then blend them into a new image that carries the content of one and the style of the other.

The fundamental goal of NST is to transfer the "style" of a reference artwork to a target photograph, creating a new image that looks like the target image but painted in the style of the reference artwork. For instance, you could take a photo and apply the artistic style of a famous painter like Van Gogh, Picasso, or Monet, creating a hybrid image.

This essay provides an overview of how to implement a neural style transfer model in Python, using deep learning libraries such as TensorFlow or PyTorch. We'll go through the steps required to implement this model and show how you can apply it to real images.

Prerequisites

Before diving into the implementation, it's essential to have some background knowledge in the following areas:

1. **Deep Learning**: Understanding how CNNs work is crucial since NST leverages their ability to extract image features.
2. **Python Programming**: We will use Python as the programming language for this project, utilizing libraries like TensorFlow, Keras, and Matplotlib.
3. **Libraries**: You'll need Python libraries such as NumPy, TensorFlow, and Matplotlib for image manipulation, neural network training, and visualization.

You should also have access to a GPU if you intend to perform NST on large images, as it is computationally expensive.

Steps to Implement Neural Style Transfer

1. Importing Necessary Libraries

We begin by importing the essential libraries that will enable us to load and process images, perform neural style transfer, and visualize the results.

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications import VGG19
from tensorflow.keras import models
import tensorflow_hub as hub
```

2. Loading and Preprocessing the Images

We need two images for the NST: the **content image** (usually a photo) and the **style image** (usually a painting). The first step is to load and preprocess these images so that they can be fed into the neural network.

```
def load_and_process_img(img_path):
    img = image.load_img(img_path, target_size=(400, 400))  # Resize for
better performance
    img = image.img_to_array(img)
    img = np.expand_dims(img, axis=0)
    img = tf.keras.applications.vgg19.preprocess_input(img)
    return img

content_img = load_and_process_img("path_to_your_content_image.jpg")
style_img = load_and_process_img("path_to_your_style_image.jpg")
```

3. Setting Up the Pre-Trained Neural Network

We use a pre-trained convolutional neural network, such as VGG19, to extract features from both the content and style images. VGG19 is commonly used for this purpose because it has been trained on a large dataset (ImageNet) and can capture various levels of abstraction in an image.

```
def get_vgg19_model():
    vgg = VGG19(weights="imagenet", include_top=False)
    vgg.trainable = False  # Freeze layers for feature extraction
    return vgg

vgg19_model = get_vgg19_model()
```

4. Extracting Features from Content and Style Images

In NST, the objective is to minimize the difference between the content of the generated image and the original content image, and to replicate the style of the style image. For this, we need to extract content and style features from the images using the VGG19 model.

We define the layers of the network from which we will extract features:

```
def get_model_layers(vgg19_model):
    content_layers = ['block5_conv2']  # Layer from which to extract content
features
    style_layers = ['block1_conv1', 'block2_conv1', 'block3_conv1',
'block4_conv1', 'block5_conv1']  # Layers for style

    model_outputs = [vgg19_model.get_layer(layer).output for layer in
content_layers + style_layers]
    model = models.Model(inputs=vgg19_model.input, outputs=model_outputs)

    return model, content_layers, style_layers

model, content_layers, style_layers = get_model_layers(vgg19_model)
```

5. Defining Loss Functions

The key part of NST is the loss function, which calculates how far the generated image is from both the content and style images. There are two types of losses:

- **Content Loss**: This is the difference between the feature map of the content image and the feature map of the generated image.
- **Style Loss**: This is the difference between the Gram matrices of the style image and the generated image.

```
def compute_content_loss(content, generated):
    return tf.reduce_mean(tf.square(content - generated))

def gram_matrix(x):
    x = tf.reshape(x, (-1, x.shape[-1]))
    return tf.linalg.matmul(x, x, transpose_a=True)

def compute_style_loss(style, generated):
    style_gram = gram_matrix(style)
    generated_gram = gram_matrix(generated)
    return tf.reduce_mean(tf.square(style_gram - generated_gram))
```

6. Optimizing the Generated Image

Now we define the total loss as a weighted sum of content and style losses. We optimize the generated image using an optimization algorithm like Adam.

```
def compute_total_loss(model, content_img, style_img, generated_img,
content_weight=1e3, style_weight=1e-2):
    model_outputs = model([content_img, style_img, generated_img])
    content_features = model_outputs[:len(content_layers)]
    style_features = model_outputs[len(content_layers):]

    content_loss = 0
    style_loss = 0

    # Compute content loss
    for content, generated in zip(content_features, [generated_img] *
len(content_layers)):
        content_loss += compute_content_loss(content, generated)

    # Compute style loss
    for style, generated in zip(style_features, [generated_img] *
len(style_layers)):
        style_loss += compute_style_loss(style, generated)

    total_loss = content_weight * content_loss + style_weight * style_loss
    return total_loss
```

7. Running the Optimization Loop

We use the `Adam` optimizer to minimize the total loss by iterating through the optimization loop. The loop continuously adjusts the generated image to minimize the loss.

```
generated_img = tf.Variable(content_img, dtype=tf.float32)  # Initialize
generated image as the content image

optimizer = tf.optimizers.Adam(learning_rate=5e-3)

for i in range(1000):  # Number of iterations
    with tf.GradientTape() as tape:
        loss = compute_total_loss(model, content_img, style_img,
generated_img)

    grads = tape.gradient(loss, generated_img)
    optimizer.apply_gradients([(grads, generated_img)])

    if i % 100 == 0:
        print(f"Iteration {i}: Loss = {loss.numpy()}")
```

8. Visualizing the Results

After running the optimization loop, you can visualize the generated image.

```
def deprocess_img(img):
    img = img.numpy()
    img = img.squeeze()
    img = img + 103.939, 116.779, 123.68  # Reverse preprocessing
    img = np.clip(img, 0, 255).astype('uint8')
    return img
```

```
plt.imshow(deprocess_img(generated_img))
plt.title("Styled Image")
plt.show()
```

Conclusion

By following these steps, you can implement a neural style transfer model that applies artistic styles to photographs. The key is to use deep learning techniques, particularly CNNs, to extract content and style features from images, and then use optimization to blend these features in a way that generates a novel image that contains both the content and style of the input images. With further tweaks and optimizations, you can experiment with various styles and images to create visually stunning results.

Certainly! Below is an outline of a 1000-word essay that provides instructions for implementing a Neural Style Transfer (NST) model in Python to apply artistic styles to photographs, followed by a script that demonstrates this implementation.

Neural Style Transfer: Applying Artistic Styles to Photographs

Introduction to Neural Style Transfer

Neural Style Transfer (NST) is a deep learning technique used to combine the content of one image (usually a photograph) with the artistic style of another image (often a painting). The core idea behind NST is to use convolutional neural networks (CNNs) to extract the content and style from separate images, and then blend them into a new image that carries the content of one and the style of the other.

The fundamental goal of NST is to transfer the "style" of a reference artwork to a target photograph, creating a new image that looks like the target image but painted in the style of the reference artwork. For instance, you could take a photo and apply the artistic style of a famous painter like Van Gogh, Picasso, or Monet, creating a hybrid image.

This essay provides an overview of how to implement a neural style transfer model in Python, using deep learning libraries such as TensorFlow or PyTorch. We'll go through the steps required to implement this model and show how you can apply it to real images.

Prerequisites

Before diving into the implementation, it's essential to have some background knowledge in the following areas:

1. **Deep Learning**: Understanding how CNNs work is crucial since NST leverages their ability to extract image features.
2. **Python Programming**: We will use Python as the programming language for this project, utilizing libraries like TensorFlow, Keras, and Matplotlib.
3. **Libraries**: You'll need Python libraries such as NumPy, TensorFlow, and Matplotlib for image manipulation, neural network training, and visualization.

You should also have access to a GPU if you intend to perform NST on large images, as it is computationally expensive.

Steps to Implement Neural Style Transfer

1. Importing Necessary Libraries

We begin by importing the essential libraries that will enable us to load and process images, perform neural style transfer, and visualize the results.

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications import VGG19
from tensorflow.keras import models
import tensorflow_hub as hub
```

2. Loading and Preprocessing the Images

We need two images for the NST: the **content image** (usually a photo) and the **style image** (usually a painting). The first step is to load and preprocess these images so that they can be fed into the neural network.

```
def load_and_process_img(img_path):
    img = image.load_img(img_path, target_size=(400, 400))  # Resize for
better performance
    img = image.img_to_array(img)
    img = np.expand_dims(img, axis=0)
    img = tf.keras.applications.vgg19.preprocess_input(img)
    return img

content_img = load_and_process_img("path_to_your_content_image.jpg")
style_img = load_and_process_img("path_to_your_style_image.jpg")
```

3. Setting Up the Pre-Trained Neural Network

We use a pre-trained convolutional neural network, such as VGG19, to extract features from both the content and style images. VGG19 is commonly used for this purpose because it has been trained on a large dataset (ImageNet) and can capture various levels of abstraction in an image.

```
def get_vgg19_model():
    vgg = VGG19(weights="imagenet", include_top=False)
    vgg.trainable = False  # Freeze layers for feature extraction
    return vgg

vgg19_model = get_vgg19_model()
```

4. Extracting Features from Content and Style Images

In NST, the objective is to minimize the difference between the content of the generated image and the original content image, and to replicate the style of the style image. For this, we need to extract content and style features from the images using the VGG19 model.

We define the layers of the network from which we will extract features:

```
def get_model_layers(vgg19_model):
    content_layers = ['block5_conv2']  # Layer from which to extract content
features
    style_layers = ['block1_conv1', 'block2_conv1', 'block3_conv1',
'block4_conv1', 'block5_conv1']  # Layers for style

    model_outputs = [vgg19_model.get_layer(layer).output for layer in
content_layers + style_layers]
    model = models.Model(inputs=vgg19_model.input, outputs=model_outputs)

    return model, content_layers, style_layers

model, content_layers, style_layers = get_model_layers(vgg19_model)
```

5. Defining Loss Functions

The key part of NST is the loss function, which calculates how far the generated image is from both the content and style images. There are two types of losses:

- **Content Loss**: This is the difference between the feature map of the content image and the feature map of the generated image.
- **Style Loss**: This is the difference between the Gram matrices of the style image and the generated image.

```
def compute_content_loss(content, generated):
    return tf.reduce_mean(tf.square(content - generated))

def gram_matrix(x):
```

```
    x = tf.reshape(x, (-1, x.shape[-1]))
    return tf.linalg.matmul(x, x, transpose_a=True)

def compute_style_loss(style, generated):
    style_gram = gram_matrix(style)
    generated_gram = gram_matrix(generated)
    return tf.reduce_mean(tf.square(style_gram - generated_gram))
```

6. Optimizing the Generated Image

Now we define the total loss as a weighted sum of content and style losses. We optimize the generated image using an optimization algorithm like Adam.

```
def compute_total_loss(model, content_img, style_img, generated_img,
content_weight=1e3, style_weight=1e-2):
    model_outputs = model([content_img, style_img, generated_img])
    content_features = model_outputs[:len(content_layers)]
    style_features = model_outputs[len(content_layers):]

    content_loss = 0
    style_loss = 0

    # Compute content loss
    for content, generated in zip(content_features, [generated_img] *
len(content_layers)):
        content_loss += compute_content_loss(content, generated)

    # Compute style loss
    for style, generated in zip(style_features, [generated_img] *
len(style_layers)):
        style_loss += compute_style_loss(style, generated)

    total_loss = content_weight * content_loss + style_weight * style_loss
    return total_loss
```

7. Running the Optimization Loop

We use the `Adam` optimizer to minimize the total loss by iterating through the optimization loop. The loop continuously adjusts the generated image to minimize the loss.

```
generated_img = tf.Variable(content_img, dtype=tf.float32)  # Initialize
generated image as the content image

optimizer = tf.optimizers.Adam(learning_rate=5e-3)

for i in range(1000):  # Number of iterations
    with tf.GradientTape() as tape:
        loss = compute_total_loss(model, content_img, style_img,
generated_img)

    grads = tape.gradient(loss, generated_img)
    optimizer.apply_gradients([(grads, generated_img)])

    if i % 100 == 0:
```

```
        print(f"Iteration {i}: Loss = {loss.numpy()}")
```

8. Visualizing the Results

After running the optimization loop, you can visualize the generated image.

```
def deprocess_img(img):
    img = img.numpy()
    img = img.squeeze()
    img = img + 103.939, 116.779, 123.68  # Reverse preprocessing
    img = np.clip(img, 0, 255).astype('uint8')
    return img

plt.imshow(deprocess_img(generated_img))
plt.title("Styled Image")
plt.show()
```

Conclusion

By following these steps, you can implement a neural style transfer model that applies artistic styles to photographs. The key is to use deep learning techniques, particularly CNNs, to extract content and style features from images, and then use optimization to blend these features in a way that generates a novel image that contains both the content and style of the input images. With further tweaks and optimizations, you can experiment with various styles and images to create visually stunning results.

Neural Style Transfer in Python

Neural Style Transfer (NST) is a deep learning technique that allows us to combine the artistic style of one image with the content of another, creating unique, stylized images from ordinary photos. NST opens up a world of creative possibilities!

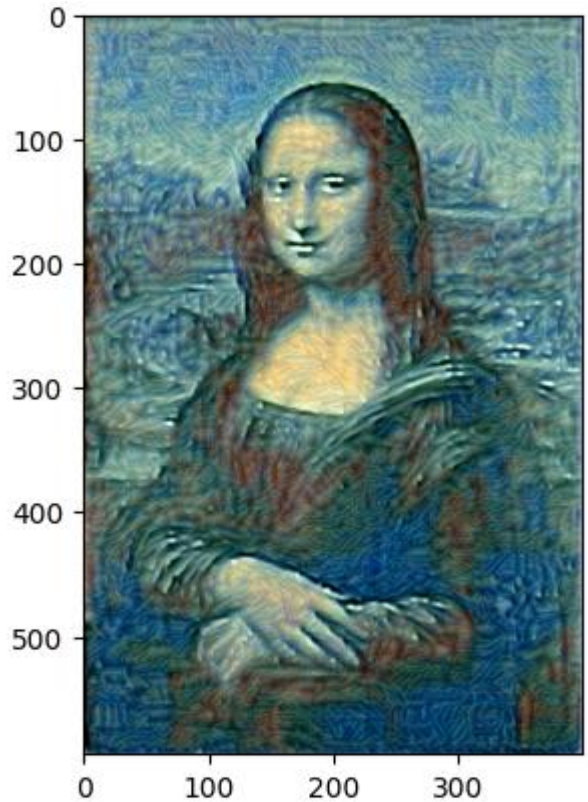
What is NST?

NST is a process that uses Convolutional Neural Networks (CNNs) to blend the content and style of two separate images. Content refers to an image's structural details, such as the shapes, objects, or people it contains. Style, conversely, encompasses the textures, colors, brush strokes, and patterns of a specific artwork or artist.

For example, you could use NST to create an interesting image by blending the content of Leonardo da Vinci's Mona Lisa with the style of Vincent van Gogh's The Starry Night.

Mona Lisa by Leonardo da Vinci (Content)

Starry Night by Vincent van Gogh (Style)



Target Image

Code

Let’s code a Neural Style Transfer in Python using PyTorch! This project will be coded in a Google Collaboratory Notebook.

First, we need to import the necessary libraries:

```
# Import necessary libraries

import torch
from torchvision import transforms, models
from PIL import Image
import matplotlib.pyplot as plt
import numpy as np

# Use cuda if it is available
device = ("cuda" if torch.cuda.is_available() else "cpu")
```

For our model, we are going to use VGG-19. VGG-19 is a convolutional neural network that is 19 layers deep. It has been trained on over a million images from the ImageNet database. We want to use the pre-trained model, so we freeze the model's parameters, preventing them from being updated during training:

```
# Use VGG-19
# VGG-19 is a CNN that is 19 layers deep that has been pre-trained on
more than a million images from the ImageNet database

model = models.vgg19(pretrained=True).features
for p in model.parameters():
  p.requires_grad = False
model.to(device)

# Print the layers
print(model)
```

Once we have the model, we can print the layers:

```
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 want to create a function that will return the activation features of the model given an input tensor. We will use this to get the activation features of the content and style images. We will mainly use the layers to capture style and content loss. This function iterates through the model's layers, extracting activations at specific layers and storing them in the `features` dictionary.

```
def model_activations(input,model):
  # 6 Activation layers
  # Layers where the dimensions are changed
  layers = {
```

```
    '0' : 'conv1_1',
    '5' : 'conv2_1',
    '10': 'conv3_1',
    '19': 'conv4_1',
    '21': 'conv4_2',
    '28': 'conv5_1'
  }
  features = {}
  x = input
  x = x.unsqueeze(0)
  for name,layer in model._modules.items():
      x = layer(x)
      if name in layers:
          features[layers[name]] = x

  return features
```

We also want to ensure that the images are a manageable resolution, so we will use the transforms in PyTorch to Resize images to 400x400 pixels, transform them into tensors, and normalize the data to make it easier for the model to understand:

```
transform = transforms.Compose([transforms.Resize(400),
                              transforms.ToTensor(),

transforms.Normalize((0.5,0.5,0.5),(0.5,0.5,0.5))])
```

Now, we have to import images into the project. Because we are using a Google Colab Notebook, we will do this using Google Drive:

```
from google.colab import drive
drive.mount('/content/drive')
# Change the Image Folder Path to wherever the images are in your
Google Drive
path = "drive/My Drive/[Image Folder Path]"

content = Image.open(path + "content.jpg").convert("RGB")
content = transform(content).to(device)
print("Content shape: ", content.shape)
style = Image.open(path + "style.jpg").convert("RGB")
```

```
style = transform(style).to(device)
print("Style shape: ", style.shape)
```

From the output, we can see that both images have three color channels (Red, Green, and Blue) and two dimensions.

```
Content shape:  torch.Size([3, 595, 400])
Style shape:  torch.Size([3, 400, 490])
```

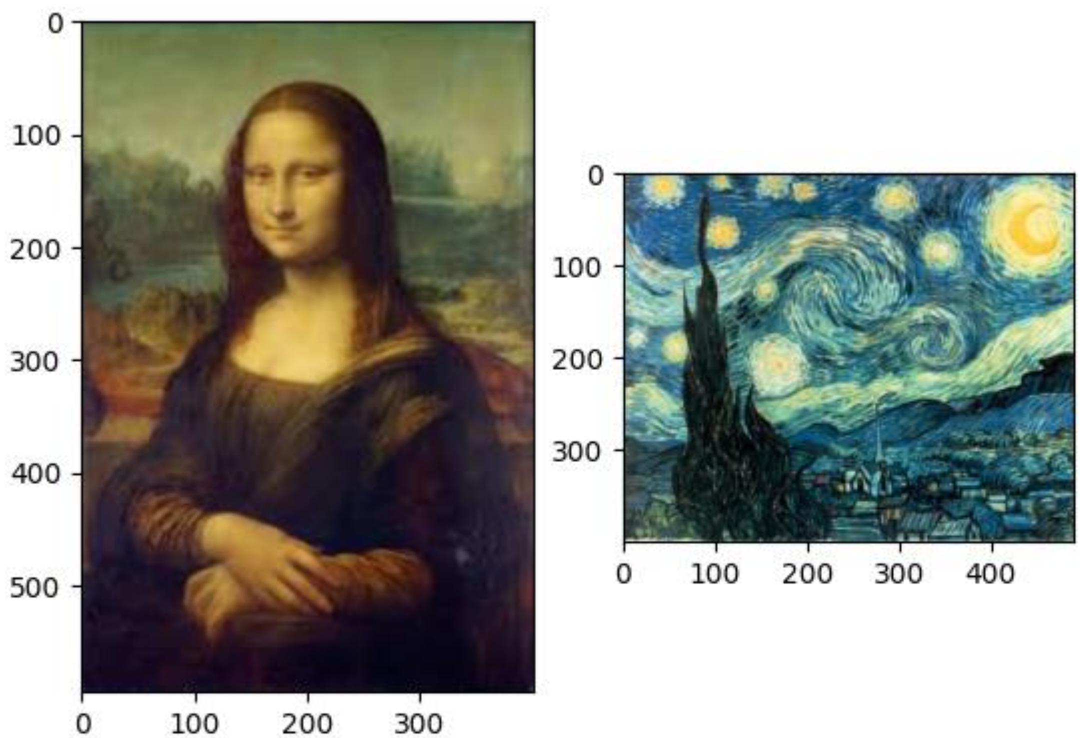
We also want to create a function that can convert image tensors to a format that MatPlotLib can display:

```
def image_convert(image):
    x = image.to("cpu").clone().detach().numpy().squeeze()
    x = x.transpose(1,2,0)
    x = x*np.array((0.5,0.5,0.5)) + np.array((0.5,0.5,0.5))
    return np.clip(x,0,1)
```

We can use this function to display the content and style images:

```
fig, (ax1,ax2) = plt.subplots(1,2)

ax1.imshow(image_convert(content),label = "Content")
ax2.imshow(image_convert(style),label = "Style")
plt.show()
```



Content and Style Images displayed by MatPlotLib

Define a function that performs matrix multiplication of a feature with its transpose to create the gram matrix. The gram matrix captures the relationships between different feature maps within the network, which helps represent style elements like color and texture:

```
def gram_matrix(imgfeature):
    _,d,h,w = imgfeature.size()
    imgfeature = imgfeature.view(d,h*w)
    gram_mat = torch.mm(imgfeature,imgfeature.t())

    return gram_mat
```

We want to create a target image with the same dimensions as the content image and extract features from the style and content images using the VGG-19 model:

```
target = content.clone().requires_grad_(True).to(device)

# Get the features of the content and style images
style_features = model_activations(style, model)
content_features = model_activations(content, model)
```

We also want to define the style weights for the model. Experiment with the weights, epochs, and learning rate for the best results. Content is captured by the higher layers, while the lower layers capture style:

```
style_weight_measurements = {"conv1_1" : 1.0, "conv2_1" : 0.8,
"conv3_1" : 0.4, "conv4_1" : 0.2, "conv5_1" : 0.1}

style_grams = {layer:gram_matrix(style_features[layer]) for layer in
style_features}

# To accurately capture style, the style weight has to be a lot higher
than the content weight
content_weight = 1000
style_weight = 1e8

print_after = 200
epochs = 4000
learning_rate = 0.1

# Use the Adam Optimizer
optimizer = torch.optim.Adam([target],lr=learning_rate)
```

content_weight: Controls the influence of the content image on the final output.

style_weight: Controls the influence of the style image on the final output.

epochs: The total number of training iterations.

learning_rate: Determines the step size during optimization.

Finally, we can generate a target image from the content and style images. We want to optimize for the number of epochs. For every iteration/epoch, we want to:

- Extract features from the current target image.
- Calculate the content loss by comparing the features of the target image to those of the content image.
- The content loss is calculated by:

$$\mathcal{L}_{content}(\vec{p}, \vec{x}, l) = \frac{1}{2} \sum_{i,j} \left(F^l_{ij} - P^l_{ij}\right)^2$$

Content Loss Formula

- Calculate the style loss by comparing the gram matrices of the target image to the gram matrices of the style image.
- Find the total loss by combining the content and style loss using the specified weights.
- For every 10th epoch, we want to print the loss.
- Calculate the gradients of the loss of the target image's pixels and update the target image based on the calculated gradients.
- Every 200 epochs, we want to display the new image. Change the value of print_after to see more target images.

```
# Iterate for the number of epochs (Optimization)
for i in range(1, epochs+1):
```

```
    target_features = model_activations(target, model)

    # Get the average loss which is calculated by the (content -
target)^2
    content_loss = torch.mean((content_features['conv4_2']-
target_features['conv4_2'])**2)

    style_loss = 0

    # Iterate through each of the style weights
    for layer in style_weight_measurements:
        style_gram = style_grams[layer]
        target_gram = target_features[layer]
        _,d,w,h = target_gram.shape
        target_gram = gram_matrix(target_gram)

        style_loss +=
(style_weight_measurements[layer]*torch.mean((target_gram-
style_gram)**2))/d*w*h # Normalize with depth, width, and height

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

    # Print every 10th epoch
    if i % 10 == 0:
        print("epoch ", i ," ", total_loss)

    optimizer.zero_grad()
    total_loss.backward()
    optimizer.step()

    # Display the image after print_after epochs
    if i % print_after == 0:
        plt.imshow(image_convert(target),label="Epoch "+str(i))
        plt.show()
        plt.imsave(str(i)+'.png',image_convert(target),format='png')
```

If you want to see the complete code, this is the Google Colab Notebook: https://colab.research.google.com/drive/1FLn333yCPvPvFRVI6ceC4SbPjhbIc6ay?usp=sharing

Limitations and Challenges of NST

Despite its many uses, NST also has some challenges and limitations:

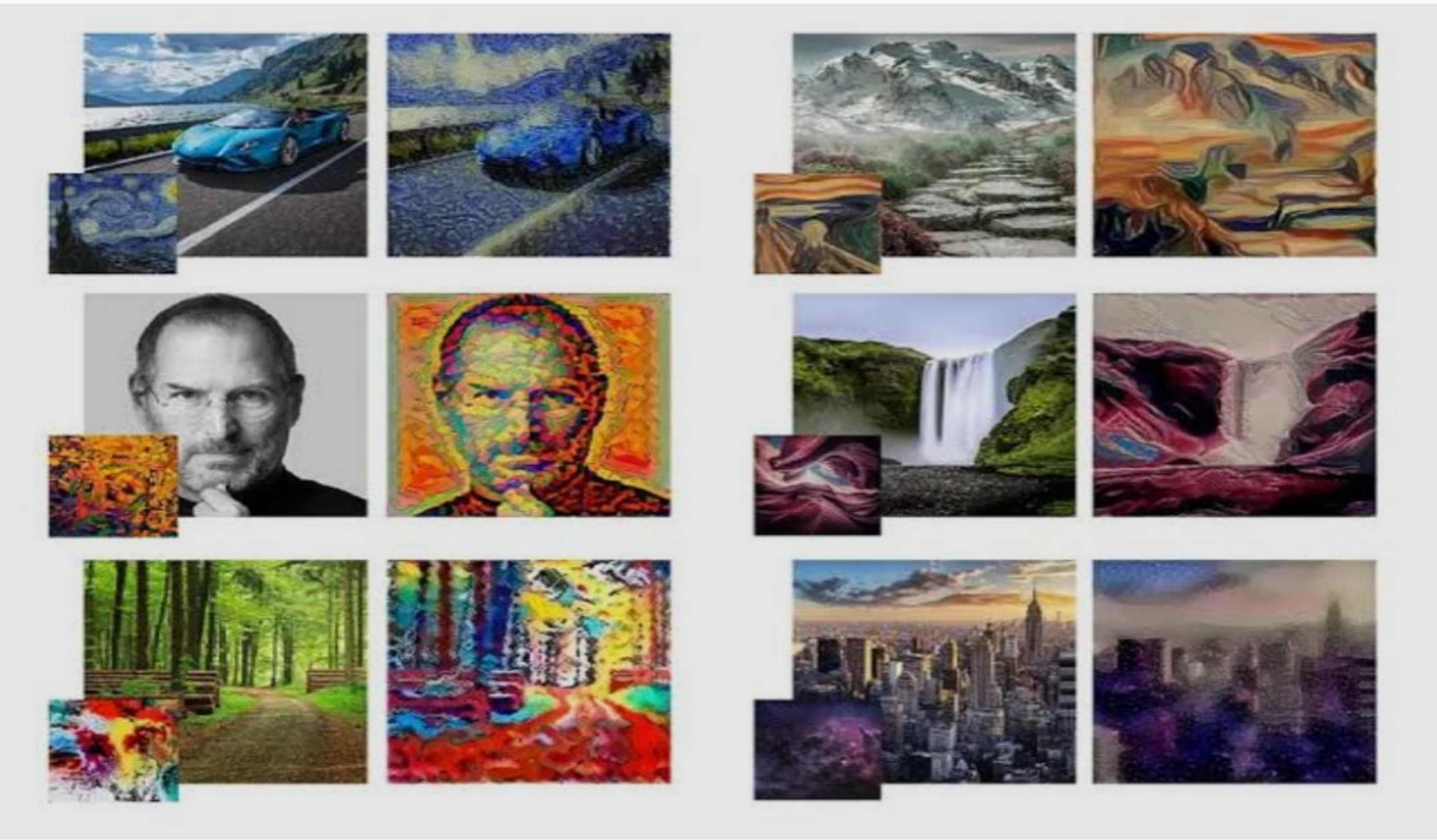
• **Computational Intensity**: The optimization process in NST is resource-intensive, making it impractical for real-time applications without specialized hardware. Also, in our project, the image's resolution is limited due to computational constraints.

• **Content Complexity**: The algorithm may struggle with more complex or less defined images, where content and style differentiation becomes challenging.

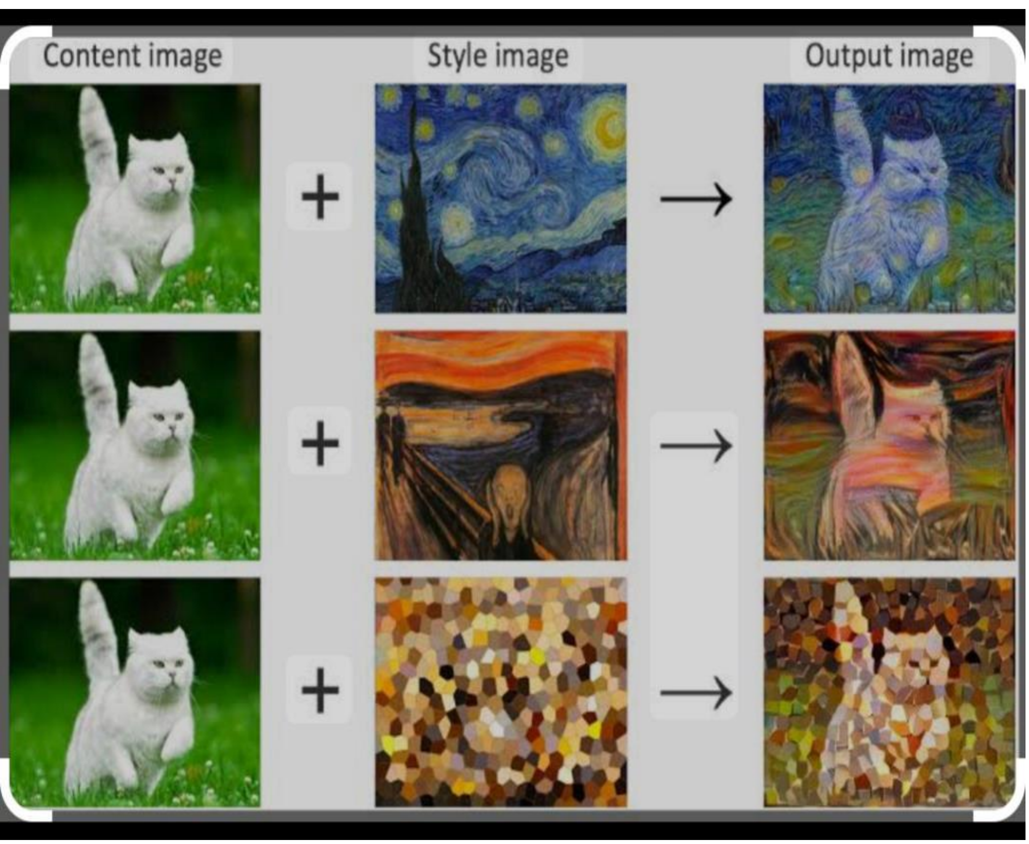
• **Control Over Output**: Fine-tuning the final output is often tricky, as adjusting style and content weights can produce unpredictable results.

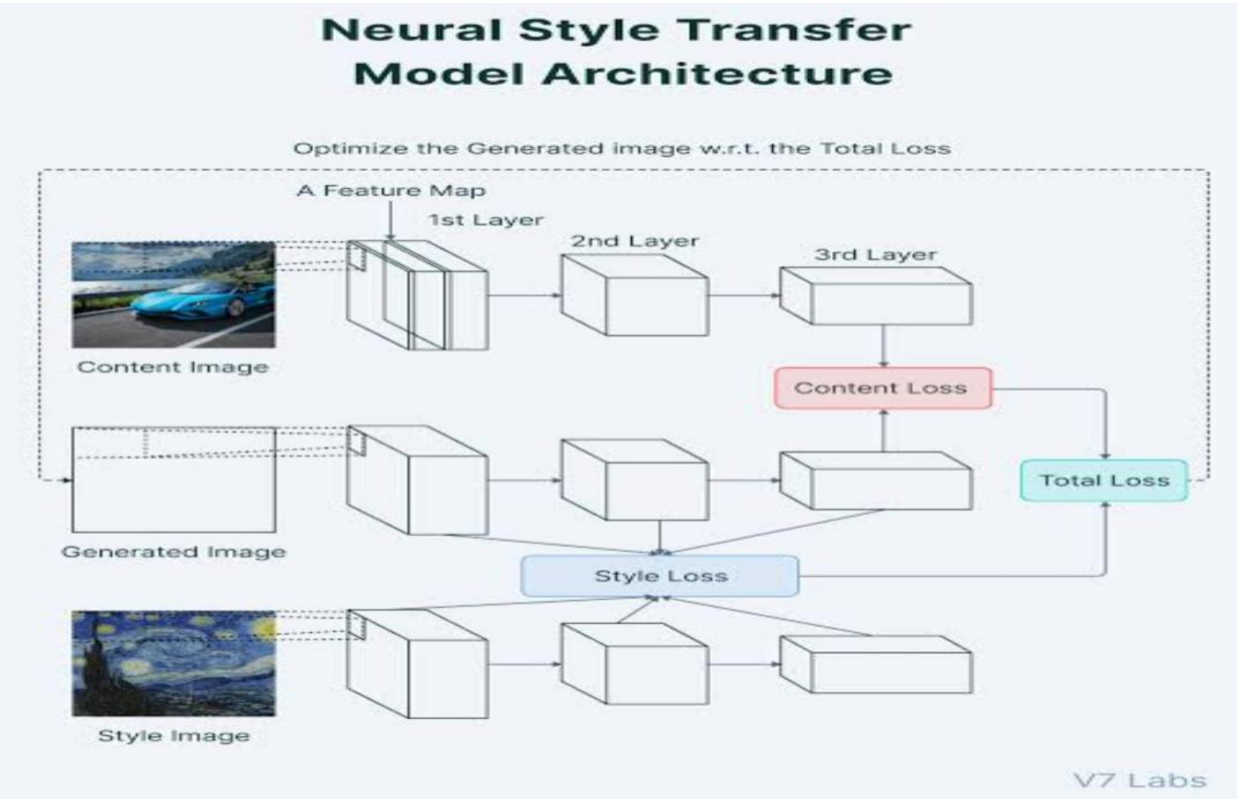
Conclusion

Neural Style Transfer represents a powerful fusion of artificial intelligence and art, allowing for creative expression through machine learning. The ability to create new images that blend content and artistic style has expanded possibilities in digital art and media and contributed to our understanding of image processing and neural networks.




Input image
Style image
Stylized image


Content image
Style image
Output image
+
+
+
→
→
→



we can benefit a lot from the neural style transfer

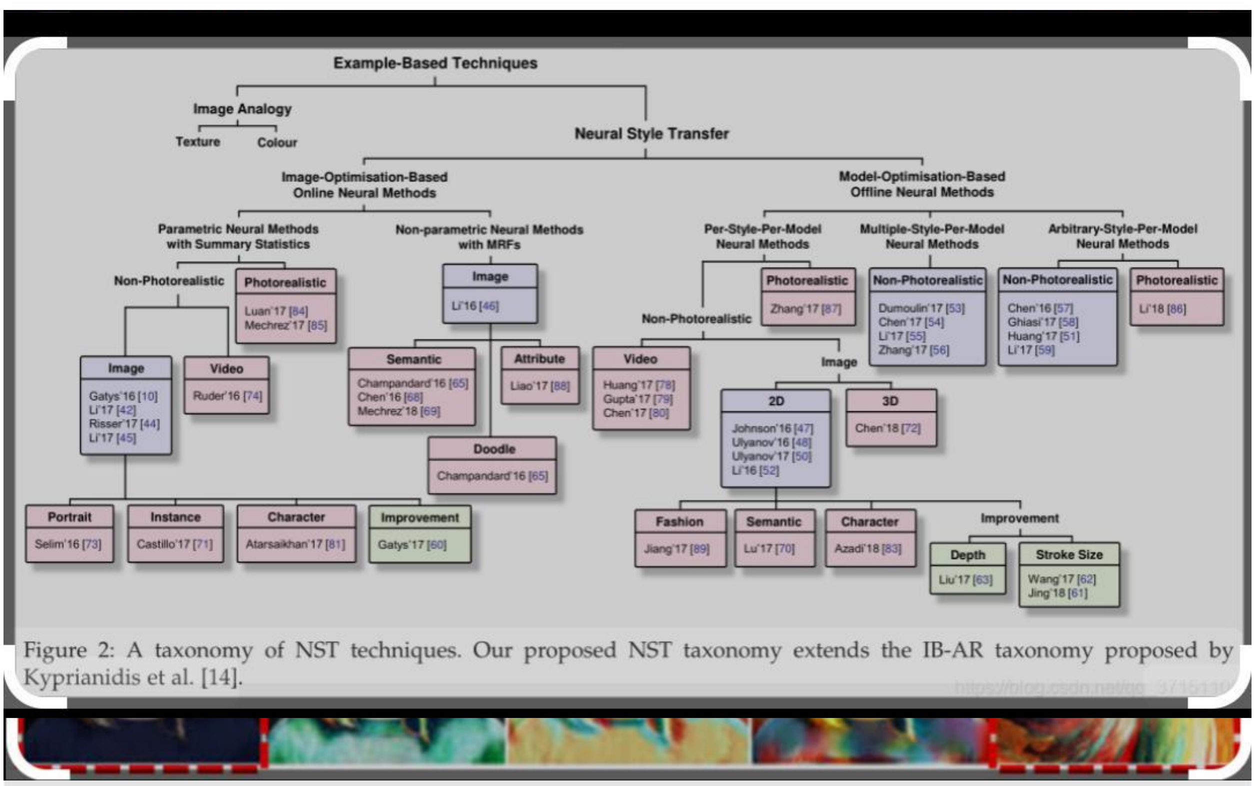


Figure 2: A taxonomy of NST techniques. Our proposed NST taxonomy extends the IB-AR taxonomy proposed by Kyprianidis et al. [14].