Neural Style Transfer

M.tech Project

submitted by:

Manish Kumar

Roll - 22MA60R03

Under the Guidance of

Dr. Pawan Kumar

in partial fulfillment of the degree

of

MASTER OF TECHNOLOGY

in

COMPUTER SCIENCE AND DATA PROCESSING

at

DEPARTMENT OF MATHEMATICS

INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

Kharagpur-721302, **West Bengal, India**

CERTIFICATE

This is to certify that the project entitled **"Neural Style Transfer"** which is being submitted by **Manish kumar (Roll no.: 22MA60R03**) in partial fulfillment of the award of the degree of Master of Technology in Computer Science and Data Processing is a record of bonafide project work carried out by his in the department of Mathematics under my supervision and guidance.

.....

Prof. Pawan Kumar
Supervisor
Department of Mathematics
IIT Kharagpur
Kharagpur - 721302,India.

Date:

Contents

Neural Style Transfer

Manish kumar

November 2023

1 Introduction

Painting is a highly appreciated form of art. Combining the style of a particular artwork with personal images has led to the emergence of image style transfer as a research trend. The goal of this project is to implement neural style transfer, a deep learning technique that transfers the style of one image onto another while preserving the content. This process, known as style transfer, involves creating artwork by combining the style of one image with the content of another. Gatys et al. (2015) demonstrated a generalized style transfer technique by exploiting feature responses from a pre-trained CNN, opening up the field of neural style transfer. Since Gatys' work, many improvements, alternatives, and extensions to the algorithm have been proposed. In this work, we implement Gatys' algorithm with some minor additions and modifications to produce synthetic artwork.

CNNs filter images using a mathematical pixel operation through the convolutional process. This process reveals various features of the image, such as edges, textures, and colors, providing a meaningful and efficient learning process for the model. The CNN architecture extracts image features hierarchically, learning details of the image.

In our implementation of Gatys' method, we use the weights of VGG-19, a 19-layer deep convolutional neural network pre-trained on the ImageNet dataset.

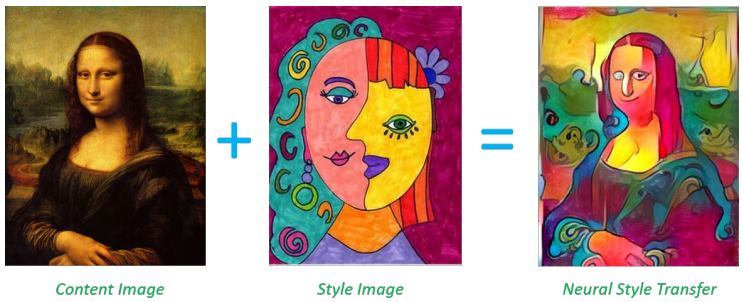

Figure 1:

1.1 What is Neural Style Transfer

Neural style transfer is an optimization technique used to take two images—a content image and a style reference image (such as an artwork by a famous painter)—and blend them together so the output image looks like the content image, but "painted" in the style of the style reference image.

1.2 Transfer Learning

Transfer learning is a machine learning technique that involves reusing a model developed for one task in another task. It is particularly useful in deep learning, allowing the training of deep neural networks with less data. Neural Style Transfer (NST) leverages a pre-trained convolutional network, building on top of it. We will use the VGG-19 network published by the Visual Geometry Group at the University of Oxford in 2014 (https://arxiv.org/abs/1508.06576). This 19-layer version of the VGG network has already been trained on the large ImageNet database, recognizing both low and high-level features.

2 VGG 19 Architecture

AlexNet, introduced in 2012, improved traditional Convolutional Neural Networks (CNNs). VGG, a successor to AlexNet, was created by the Visual Geometry Group at Oxford, incorporating ideas from its predecessors. VGG 19 uses deep Convolutional Neural Layers to enhance accuracy.

VGG 19 Architecture has been trained on the extensive ImageNet database, a large-scale dataset of labeled images widely used for training and evaluating machine learning algorithms, especially in computer vision. ImageNet contains millions of labeled images categorized into thousands of classes, playing a significant role in the progress of deep learning and CNNs.

- A fixed size of (224×224) RGB image served as input, resulting in a matrix shape of (224, 224, 3).
- The only preprocessing involved subtracting the mean of RGB values from each pixel, computed over the entire training set.
- Kernels of (3×3) size were utilized with a stride size of 1 pixel, covering the entire image.
- Spatial padding in VGG19 preserved spatial resolution by adding zeros around the image before applying convolutional filters, ensuring output feature maps matched the input size.
- Max pooling was performed over 2×2 pixel windows with a stride of 2.
- ReLU activation in VGG19 after convolutional and fully connected layers enabled the network to learn complex representations, contributing to its success in image classification tasks on datasets like ImageNet.

- Three fully connected layers were implemented, with the first two of size 4096, followed by a layer with 1000 channels, and the final layer as a Softmax function.

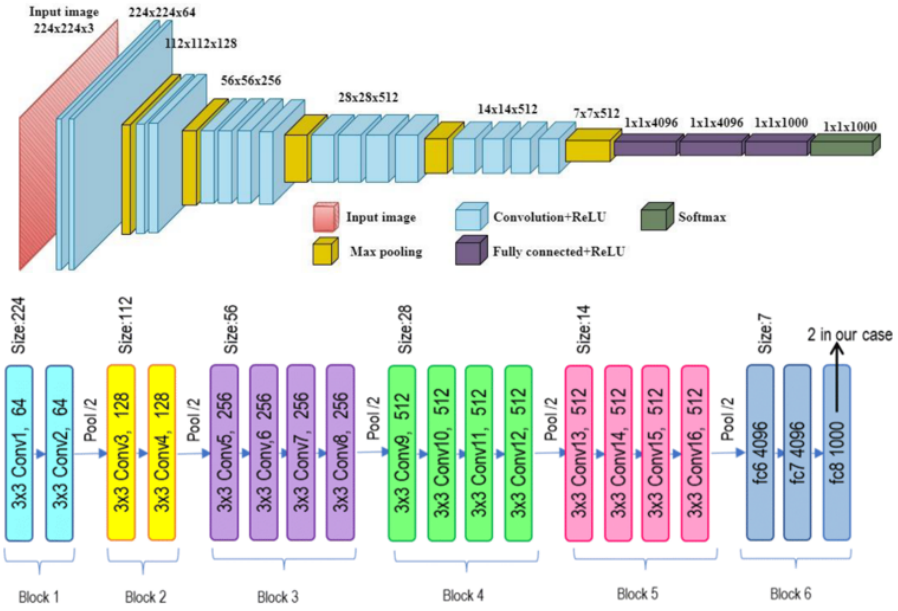

Figure 2:

2.1 ReLU Activation Function

ReLU, which stands for Rectified Linear Unit, is an activation function commonly used in artificial neural networks and deep learning models. It is a simple mathematical function that introduces non-linearity to the network, allowing it to learn complex patterns in the data.

The ReLU function is defined as follows:

$$f(x) = \max(0, x)$$

2.2 Softmax Activation Function

Softmax is an activation function commonly used in the output layer of a neural network for multiclass classification problems. It transforms a vector of real numbers into a probability distribution over multiple classes. The output of the softmax function is a vector where each element represents the probability of the corresponding class.

Mathematically, for a given vector $z = (z_1, z_2, z_3, \ldots, z_k)$, the softmax function is defined as follows:

$$softmax(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$$

In this formula:

- z_i is the i-th element of the input vector.
- The denominator is the sum of the exponential values of all elements in the input vector.

The softmax function transforms raw scores or logits (z) into probabilities that sum to 1. It amplifies the largest element in the vector, making it the most probable class, and suppresses smaller values. This is particularly useful in classification tasks where the goal is to assign an input to one of several possible classes.

3 Content Cost Function

In Neural Style Transfer (NST), the content cost function is crucial for ensuring that the content in the generated image G matches the content of the input image C. This involves understanding the role of different layers in a Convolutional Neural Network (CNN) and selecting an appropriate layer for content representation.

- The shallower layers of a Convolutional Net detect lower-level features such as edges and simple textures.
- The deeper layers tend to detect higher-level features such as more complex textures and object classes.

To achieve a visually pleasing result, it's essential to choose a layer for content representation that is neither too shallow nor too deep. This ensures that the network captures both higher-level and lower-level features.

3.1 Forward Propagation for Content Cost

- Set the image C as the input to the pretrained VGG network and run forward propagation.
- Let a^c be the hidden layer activations in the chosen layer, resulting in an $n_h \times n_w \times n_c$ tensor.
- To forward propagate image G: Repeat the process with image G. Set G as the input and run forward propagation. Let a^g be the corresponding hidden layer activation.

3.2 Content Cost Calculation

The content cost function is defined as follows:

$$J_{Content}(C, G) = \frac{1}{4 \times n_H \times n_w \times n_c} \sum_{allentries} (a^c - a^g)^2$$

Here, n_H, n_w, and n_c are the height, width, and number of channels of the chosen hidden layer. These parameters appear in a normalization term in the cost.

4 Style Cost Function

4.1 Gram Matrix

In linear algebra, the Gram matrix G of a set of vectors $(v_1, v_2, \ldots, v_n)$ is the matrix of dot products, defined as $G_{ij} = v_i^T \cdot v_j$. The entry G_{ij} compares how similar v_i is to v_j.

We will compute the Style matrix by multiplying the "unrolled" filter matrix with its transpose.

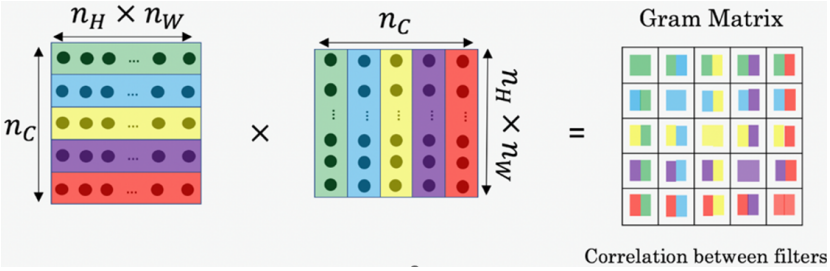

Figure 3:

4.2 Style Cost

In the style cost calculation, we aim to minimize the distance between the Gram matrix of the "style" image S and the Gram matrix of the "generated" image G. For simplicity, we'll consider a single hidden layer $a^{[l]}$.

The corresponding style cost for this layer is defined as:

$$J_{style^{[l]}} = \frac{1}{4 \times n_H^2 \times n_W^2 \times n_C^2} \sum_{i=1}^{n_C} \sum_{j=1}^{n_C} (G^{(S)}_{gram_{ij}} - G^{(G)}_{gram_{ij}})^2$$

Here:

- $G^{(S)}_{gram}$ is the Gram matrix of the "style" image.

- $G_{gram}^{(G)}$ is the Gram matrix of the "generated" image.
- n_H, n_W, and n_C are the height, width, and number of channels of the hidden layer, respectively.

.

5 Total Cost

In Neural Style Transfer (NST), the total cost function is formulated to minimize both the content cost ($J_{Content}(C, G)$) and the style cost ($J_{Style}(S, G)$). The formula for the total cost is expressed as a linear combination of the content and style costs, with hyperparameters α and β determining their relative weights:

$$J(G) = \alpha J_{Content}(C, G) + \beta J_{Style}(S, G)$$

Here:

- $J_{Content}(C, G)$ is the content cost between the generated image G and the content image C.
- $J_{Style}(S, G)$ is the style cost between the generated image G and the style image S.
- α and β are hyperparameters controlling the trade-off between preserving content and capturing style.

Feel free to adjust the values of α and β based on your specific requirements.

5.1 Selected Style Layers

For style representation, specific convolutional layers within each block of the VGG19 model were chosen. The selected layers, along with their corresponding style weights, are as follows:

- **Block 1:**
  - Layer: block1_conv1
  - Style Weight: 1.0
- **Block 2:**
  - Layer: block2_conv1
  - Style Weight: 0.8
- **Block 3:**
  - Layer: block3_conv1
  - Style Weight: 0.7

- **Block 4:**
  - Layer: block4_conv1
  - Style Weight: 0.2
- **Block 5:**
  - Layer: block5_conv1
  - Style Weight: 0.1

These layers were chosen to effectively capture different aspects of artistic style.

5.2 Selected Content Layer

For content preservation, a specific convolutional layer within the VGG19 model was focused on:

- **Block 5:**
  - Layer: block5_conv4

This content layer was chosen to ensure that the generated image retains the content structure of the target image.

5.3 Solving the Optimization Problem

Finally, everything is put together to implement Neural Style Transfer! The program should be able to perform the following steps:

1. Load the content image
2. Load the style image
3. Randomly initialize the image to be generated
4. Load the VGG19 model
5. Compute the content cost
6. Compute the style cost
7. Compute the total cost
8. Define the optimizer and learning rate

Feel free to adapt the LaTeX code based on your specific requirements.

6 Adam Optimizer

The Adam optimizer, short for "Adaptive Moment Estimation," is an iterative optimization algorithm used to minimize the loss function during the training of neural networks. Adam can be looked at as a combination of RMSprop and Stochastic Gradient Descent with momentum. Developed by Diederik P. Kingma and Jimmy Ba in 2014, Adam has become a go-to choice for many machine learning practitioners.

It uses the squared gradients to scale the learning rate and takes advantage of momentum by using the moving average of the gradient instead of the gradient itself. This combines Dynamic Learning Rate and Smoothening to reach the global minima.

Adam maintains two moving averages for each parameter: the first moment (mean) m and the second moment v. These moving averages are initialized to zero vectors.

- Learning rate (α): Adam requires a learning rate, which determines the step size during optimization. It is typically set to a small value.
- Exponential decay rate for the first moment estimate (β_1): A parameter close to 1 that controls the exponential decay of the moving average of gradients. Commonly set to 0.9.
- Exponential decay rate for the second moment estimate (β_2): A parameter close to 1 that controls the exponential decay of the moving average of squared gradients. Commonly set to 0.999.
- $\epsilon(\epsilon)$: A small constant added to the denominator to avoid division by zero. Commonly set to 10^{-8}.

6.1 Steps Involved in the Adam Optimization Algorithm

1. Initialize the first and second moments' moving averages (m and v) to zero.
2. At each iteration t, compute the gradient g_t with respect to the parameters.
3. Update the first moment estimate m_t and the second moment estimate v_t using exponential decay:

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$$

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot (g_t^2)$$

4. Correct the bias in the estimates (bias correction is applied to prevent the estimates from being too biased towards zero, especially in the early stages of training):

$$m_t = \frac{m_t}{1 - \beta_1^t}$$

$$v_t = \frac{v_t}{1 - \beta_2^t}$$

5. Update the parameter:

$$\theta_{t+1} = \theta_t - \frac{\alpha \cdot m_t}{\sqrt{v_t} + \epsilon}$$

6.2 Advantages of Adam

- **Adaptive Learning Rate:** Adam adapts the learning rates for each parameter individually. It uses the first and second moments to adjust the learning rates, which can be particularly useful for training on sparse data.
- **Momentum-Like Behaviour:** The inclusion of momentum-like terms (m_t) allows Adam to accumulate knowledge from past gradients, improving convergence on surfaces with different curvatures.

In the context of neural style transfer, the Adam optimizer is used to iteratively update the pixel values of the generated image to minimize the combined content and style loss.

7 Code:

7.1 Import Libraries

```
import os
import sys
import scipy.io
import scipy.misc
import matplotlib.pyplot as plt
from matplotlib.pyplot import imshow
from PIL import Image
import numpy as np
import tensorflow as tf
from tensorflow.python.framework.ops import EagerTensor
import pprint
from google.colab import drive

drive.mount('/content/drive')
```

7.2 Load VGG19 Model

```
tf.random.set_seed(272)
pp = pprint.PrettyPrinter(indent=4)
img_size = 400
vgg = tf.keras.applications.VGG19(include_top=False,
```

```
                                    input_shape=(img_size, img_size, 3),
                                    weights='imagenet')

vgg.trainable = False

# Print the model summary
vgg.summary()
```

7.3 Load Content Image

```
content_image = Image.open("/content/drive/MyDrive/output/Input/johny.jpeg")
content_image
```

7.4 Define Content Cost Function

```
# GRADED FUNCTION: compute_content_cost

def compute_content_cost(content_output, generated_output):
    """
    Computes the content cost

    Arguments:
    a_C -- tensor of dimension (1, n_H, n_W, n_C), hidden layer activations representing cor
    a_G -- tensor of dimension (1, n_H, n_W, n_C), hidden layer activations representing cor

    Returns:
    J_content -- scalar that you compute using equation 1 above.
    """
    a_C = content_output[-1]
    a_G = generated_output[-1]

    # Retrieve dimensions from a_G
    m, n_H, n_W, n_C = a_G.get_shape().as_list()

    # Reshape a_C and a_G
    a_C_unrolled = tf.reshape(a_C, shape=[m, n_H * n_W, n_C])
    a_G_unrolled = tf.reshape(a_G, shape=[m, n_H * n_W, n_C])

    # Compute the cost with tensorflow
    J_content =  tf.reduce_sum(tf.square(a_C_unrolled - a_G_unrolled))/(4.0 * n_H * n_W * n_

    return J_content
```

7.5 Define Gram Matrix Function

```
# GRADED FUNCTION: gram_matrix
```

```
def gram_matrix(A):
    """
    Argument:
    A -- matrix of shape (n_C, n_H*n_W)

    Returns:
    GA -- Gram matrix of A, of shape (n_C, n_C)
    """
    GA = tf.matmul(A, tf.transpose(A))
    return GA
```

7.6 Define Layer Style Cost Function

```
# GRADED FUNCTION: compute_layer_style_cost

def compute_layer_style_cost(a_S, a_G):
    """
    Arguments:
    a_S -- tensor of dimension (1, n_H, n_W, n_C), hidden layer activations representing sty
    a_G -- tensor of dimension (1, n_H, n_W, n_C), hidden layer activations representing sty

    Returns:
    J_style_layer -- tensor representing a scalar value, style cost defined above by equatio
    """
    m, n_H, n_W, n_C = a_G.get_shape().as_list()

    # Reshape the images to have them of shape (n_C, n_H*n_W)
    a_S = tf.transpose(tf.reshape(a_S, shape=[-1, n_C]))
    a_G = tf.transpose(tf.reshape(a_G, shape=[-1, n_C]))

    # Computing gram_matrices for both images S and G
    GS = gram_matrix(a_S)
    GG = gram_matrix(a_G)

    # Computing the loss
    J_style_layer = tf.reduce_sum(tf.square(GS - GG))/(4.0 *(( n_H * n_W * n_C)**2))

    return J_style_layer
```

7.7 Define Style Cost Function

```
# GRADED FUNCTION: compute_style_cost

def compute_style_cost(style_image_output, generated_image_output, STYLE_LAYERS=STYLE_LAYERS
    """
```

```
    Computes the overall style cost from several chosen layers

    Arguments:
    style_image_output -- our tensorflow model
    generated_image_output --
    STYLE_LAYERS -- A python list containing:
                        - the names of the layers we would like to extract style from
                        - a coefficient for each of them

    Returns:
    J_style -- tensor representing a scalar value, style cost defined above by equation (2)
    """

    J_style = 0
    a_S = style_image_output[1:]
    a_G = generated_image_output[1:]

    for i, weight in zip(range(len(a_S)), STYLE_LAYERS):
        J_style_layer = compute_layer_style_cost(a_S[i], a_G[i])
        J_style += weight[1] * J_style_layer

    return J_style
```

7.8 Define Total Cost Function

```
# GRADED FUNCTION: total_cost

@tf.function()
def total_cost(J_content, J_style, alpha=100, beta=10):
    """
    Computes the total cost function

    Arguments:
    J_content -- content cost coded above
    J_style -- style cost coded above
    alpha -- hyperparameter weighting the importance of the content cost
    beta -- hyperparameter weighting the importance of the style cost

    Returns:
    J -- total cost as defined by the formula above.
    """
    J = alpha * J_content + beta * J_style
    return J
```

7.9 Load Images and Initialize Generated Image

```
content_image = np.array(Image.open("/content/drive/MyDrive/output/Input/johny.jpeg").resize
content_image = tf.constant(np.reshape(content_image, ((1,) + content_image.shape)))

style_image =  np.array(Image.open("/content/drive/MyDrive/output/Input/floral.jpeg").resize
style_image = tf.constant(np.reshape(style_image, ((1,) + style_image.shape)))

generated_image = tf.Variable(tf.image.convert_image_dtype(content_image, tf.float32))
noise = tf.random.uniform(tf.shape(generated_image), 0, 0.8)
generated_image = tf.add(generated_image, noise)
generated_image = tf.clip_by_value(generated_image, clip_value_min=0.0, clip_value_max=1.0)
```

7.10 Get Layer Outputs

```
def get_layer_outputs(vgg, layer_names):
    outputs = [vgg.get_layer(layer[0]).output for layer in layer_names]
    model = tf.keras.Model([vgg.input], outputs)
    return model

content_layer = [('block5_conv4', 1)]
vgg_model_outputs = get_layer_outputs(vgg, STYLE_LAYERS + content_layer)

content_target = vgg_model_outputs(content_image)  # Content encoder
style_targets = vgg_model_outputs(style_image)     # Style encoder

preprocessed_content =  tf.Variable(tf.image.convert_image_dtype(content_image, tf.float32))
a_C = vgg_model_outputs(preprocessed_content)

a_G = vgg_model_outputs(generated_image)

J_content = compute_content_cost(a_C, a_G)
```

7.11 Compute Style Cost

```
preprocessed_style =  tf.Variable(tf.image.convert_image_dtype(style_image, tf.float32))
a_S = vgg_model_outputs(preprocessed_style)

J_style = compute_style_cost(a_S, a_G)

def clip_0_1(image):
    return tf.clip_by_value(image, clip_value_min=0.0, clip_value_max=1.0)

def tensor_to_image(tensor):
    tensor = tensor * 255
    tensor = np.array(tensor, dtype=np.uint8)
```

```
    if np.ndim(tensor) > 3:
        assert tensor.shape[0] == 1
        tensor = tensor[0]
    return Image.fromarray(tensor)
```

7.12 Define Train Step Function

```
# GRADED FUNCTION: train_step

optimizer = tf.keras.optimizers.Adam(0.03)

@tf.function()
def train_step(generated_image, alpha, beta):
    with tf.GradientTape() as tape:
        a_G = vgg_model_outputs(generated_image)
        J_style = compute_style_cost(a_S, a_G)
        J_content = compute_content_cost(a_C, a_G)
        J = total_cost(J_content, J_style, alpha=alpha, beta=beta)

    grad = tape.gradient(J, generated_image)
    optimizer.apply_gradients([(grad, generated_image)])
    generated_image.assign(clip_0_1(generated_image))
    return J

#Training Loop
# Initialize generated image (e.g., with content image or random noise)
generated_image = tf.Variable(tf.image.convert_image_dtype(content_image, tf.float32))

# Lists to store loss values and iteration numbers
loss_values = []
iteration_numbers = []

# Optimization loop
epochs = 1001
for i in range(epochs):
    J = train_step(generated_image, alpha=100, beta=10)

    # Record loss values and iteration number
    loss_values.append(J.numpy())
    iteration_numbers.append(i)

    # Print loss every 500 epochs
    if i % 250 == 0:
        image = tensor_to_image(generated_image)
        imshow(image)
        plt.show()
```

```
        print(f"Epoch {i}, Loss: {J.numpy()}")
        image.save(f"/content/drive/MyDrive/output/image_{i}.jpg")

# Plot the generated image and loss graph after all epochs
image = tensor_to_image(generated_image)

# Create a new figure and axis for each plot
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4))

# Plot the generated image
ax1.imshow(image)
ax1.set_title(f"Final Result")
ax1.axis("off")

# Plot the loss graph
ax2.plot(iteration_numbers, loss_values, label="Total Loss")
ax2.set_title("Total Loss Over Iterations")
ax2.set_xlabel("Iteration")
ax2.set_ylabel("Total Loss")

plt.show()

# Save the final generated image
#image.save("/content/drive/MyDrive/output/final_result.jpg")
```

8 Result

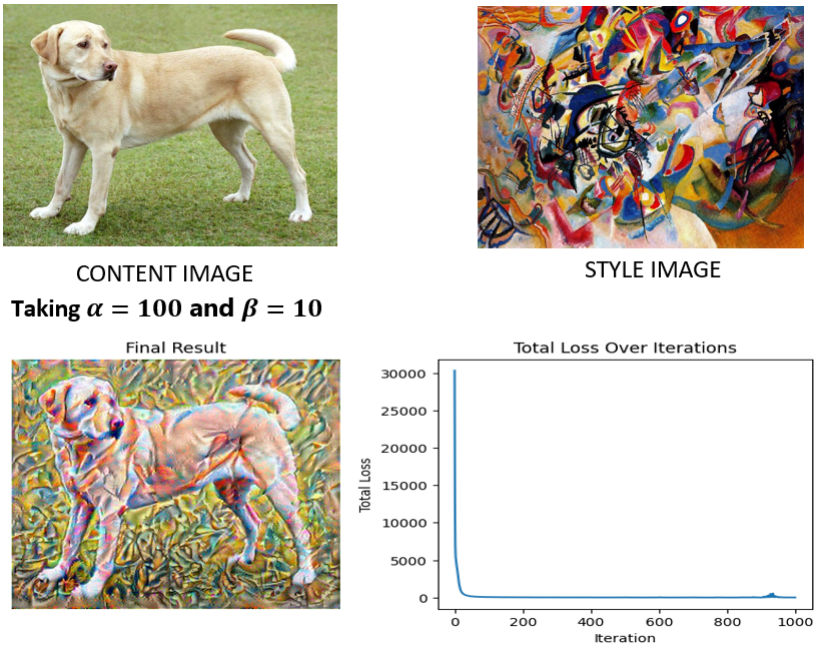

Figure 4:

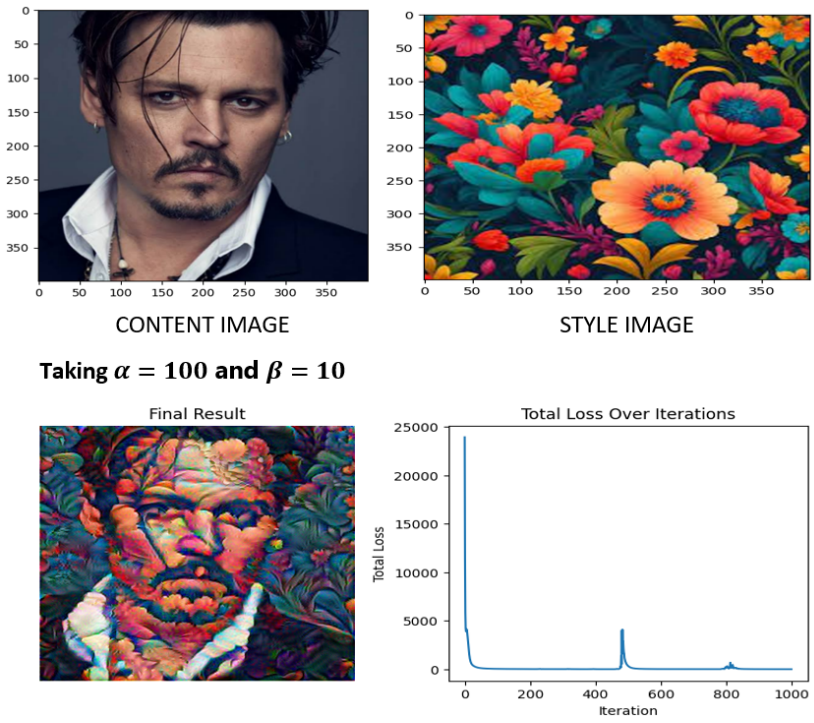

Figure 5:

[1] [2]

References

[1] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge, *A Neural Algorithm of Artistic Style*, arXiv:1508.06576, 2015.

[2] Karen Simonyan and Andrew Zisserman, *Very Deep Convolutional Networks for Large-Scale Image Recognition*, arXiv:1409.1556, 2015.