Artistic Vision: Neural Style Transfer for Image Synthesis

Vatsal Kumar (231126)
Vibhor Srivastava (231144)
Mohnish Bhandarkar (230662)
D.Sahasra (240313)
Sarthak Sehgal (240941)

Abstract

Artistic Style Transfer using Neural Networks (NST) has emerged as a powerful technique at the intersection of computer vision and art, capable of rendering a content image in the artistic style of a source artwork. This technique, first introduced by Gatys et al., leverages the deep feature representations learned by Convolutional Neural Networks (CNNs) to separate and recombine image content and style. However, these foundational NST methods are often limited in their flexibility. They typically perform a static, 1-to-1 transfer that offers the user no control over the stylistic elements. A significant drawback is that this process holistically transfers the style image's color palette, often overwriting the content image's original colors and leading to unnatural results.

This paper proposes and implements a novel, controllable system that extends the foundational NST algorithm to address these key limitations. Our system introduces two distinct novelties to transform the process from a static filter into a flexible artistic tool. The first innovation is controllable style interpolation, which allows the system to take two distinct style sources as input. By modifying the style loss to be a weighted average of the losses from both sources, a user can fluidly blend textures from different artworks. The second novelty is a perceptual color preservation loss. This is achieved by introducing a new loss term, calculated in the YUV color space, which explicitly penalizes deviations in the chrominance (U and V) channels from the original content image. This effectively decouples the transfer of texture from the transfer of color.

Our experimental results demonstrate the success of this dual-component system. We show a smooth and continuous transition between styles as the interpolation weight is varied. Furthermore, we demonstrate that our color preservation loss successfully retains the content image's original color palette, even when applying aggressive textural styles. This combined approach provides a high degree of artistic control, enabling the creation of more desirable, customized, and aesthetically pleasing blended-style images.

1 Introduction

The ability to separate and recombine the content and style of digital images has been a long-standing goal in computer vision and graphics. The seminal work by Gatys et al. (2015) demonstrated that Convolutional Neural Networks (CNNs), pre-trained on object recognition, are uniquely suited for this task. By using the deep feature maps of a network like VGG-19, their method could extract the ”content” of one image and the ”style” (texture, color, and patterns) of another and combine them into a new, synthesized image.

While powerful, this foundational technique is a static, 1-to-1 filter. It offers no control to the user, and the style image’s color palette often completely overwhelms the content image, leading to unnatural results.

This project addresses these limitations by proposing two novel extensions to create a more flexible and controllable artistic tool.

- **Style Interpolation.** Our first novelty, Style Interpolation, directly addresses the creative limitations of standard, single-style transfer. Instead of being restricted to a static, 1-to-1 transfer from a single artistic source, our method takes two distinct style images as input: in our case, the abstract and chaotic **”Composition VII” by Kandinsky** and the iconic, swirling **”Starry Night” by Van Gogh.**

  We then introduce a user-controllable parameter, the interpolation weight (ω), which operates in the range [0.0, 1.0]. This weight determines the precise balance of influence between the two styles. This is not a simple visual cross-fade; the interpolation happens at the feature level. We modify the core style loss function to be a weighted sum of the individual style losses from each source, as shown in Equation 3.

  By adjusting , a user can precisely navigate the entire ”stylistic spectrum” between the two artworks. For example, setting ω =0.0 applies 100% of the Kandinsky style, ω =1.0 applies 100% of the Van Gogh style, and ω =0.7 creates a novel hybrid image that derives 30% of its texture from Kandinsky and 70% from Van Gogh. This transforms the style transfer process from a static filter into a dynamic tool for creating new, blended aesthetics.

- **Color Preservation.** Our second, more advanced, novelty solves the critical problem of color contamination. A standard style transfer will force the style’s entire color palette onto the content image, often creating an unnatural result (e.g., the green foliage and blue sky of your landscape photo being replaced by the dark blues and yellows of ”Starry Night”).

  To prevent this, we introduce a new, perceptual loss function that **decouples texture from color**. We achieve this by calculating an additional loss in the **YUV color space**, which separates luminance (Y, or brightness) from chrominance (U and V, or color). Our loss function specifically penalizes any deviation in the U and V color channels between the generated image and the original content image. This allows our system to transfer the complex textures (e.g., the chaotic brush strokes of Kandinsky or the iconic swirls of Van Gogh) while rigorously preserving the original color palette of the content photograph (e.g., the specific greens of the trees, the browns of the building,

Our work combines these two features into a single, unified system, providing a high degree of artistic control over the final generated image.

2 Proposed Method

Our method is built upon the foundational VGG-19-based style transfer algorithm. The core idea is to define a total loss function, L_{total}, which we minimize to generate the output image G (which starts as the content image C). Our total loss is a weighted sum of three distinct components: content, style, and color.

$$L_{\text{total}} = \alpha L_{\text{content}} + \beta L_{\text{style\_blend}} + \gamma L_{\text{color}} \tag{1}$$

Where α, β, and γ are weighting factors for content, style, and color respectively.

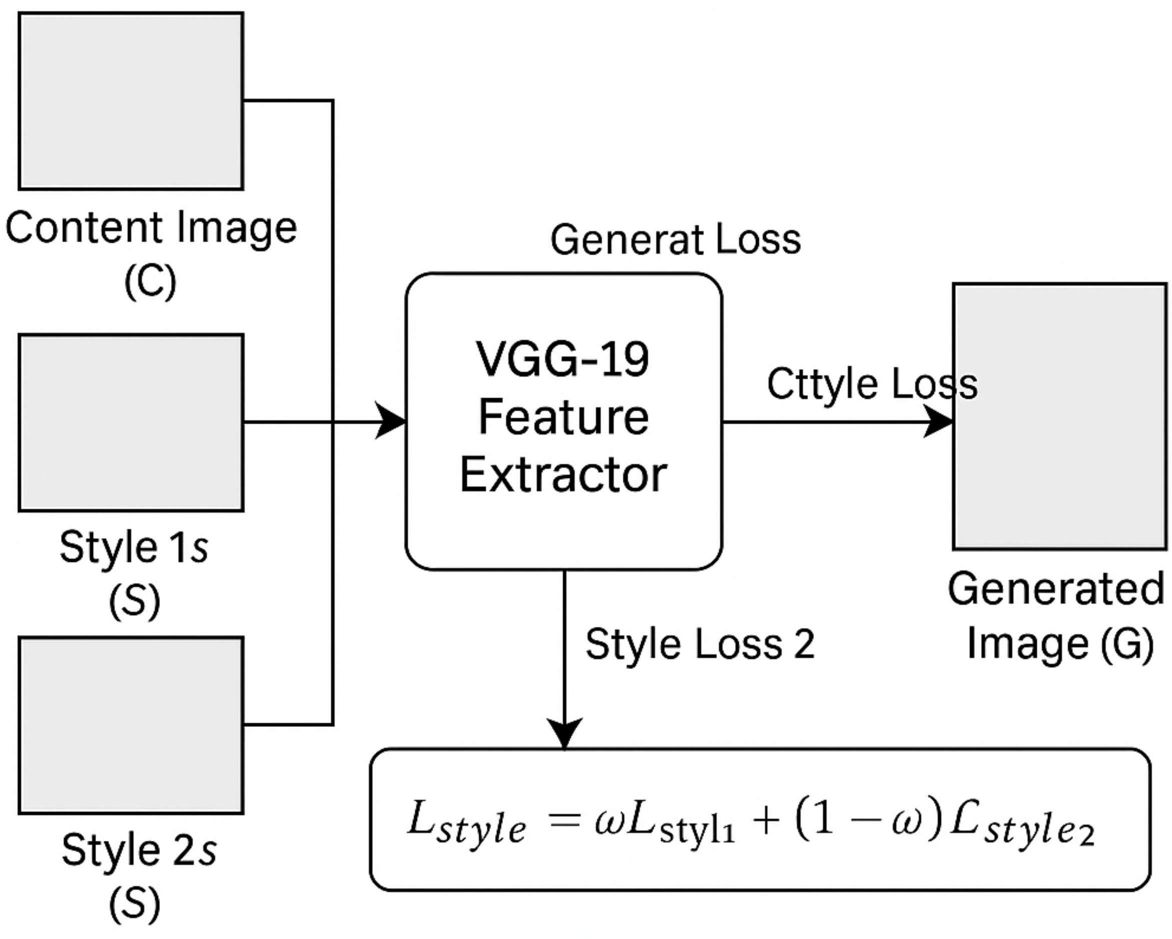

Figure 1: Overview of our proposed method

Figure 1: Overview of our proposed method. One Content (C) and two Style (S , S) images are fed into the VGG-19 extractor. The losses for the Generated Image (G) are computed, and the two style losses are blended with weight ω.

2.1 Content Loss

The content loss, L_{content}, ensures the generated image G retains the spatial structure of the content image C. It is the mean squared error between their feature representations from a higher-level layer (e.g., 'block5_conv2') of the VGG-19 network.

$$L_{\text{content}} = \frac{1}{2} \sum_{l} (F^l(G) - F^l(C))^2 \tag{2}$$

where $F^l(X)$ is the feature map of image X at layer l.

2.2 Blended Style Loss

Style is represented by the Gram matrix ($Gram(X)$), which captures the correlations between feature maps. The standard style loss compares the Gram matrices of G and a single style image S.

Our novel *blended style loss*, $L_{\text{style\_blend}}$, compares G to **two** style images, S_1 and S_2, controlled by an interpolation weight $\omega \in [0, 1]$.

$$L_{\text{style\_blend}} = (1 - \omega) L_{\text{style}}(G, S_1) + \omega L_{\text{style}}(G, S_2) \tag{3}$$

where $L_{\text{style}}(G, S) = \sum_l (Gram^l(G) - Gram^l(S))^2$.

2.3 Color Preservation Loss

To decouple texture from color, we introduce L_{color}. This loss is calculated in the YUV color space, which separates luminance (Y) from chrominance (U and V). We penalize the difference between the U and V channels of the generated image G and the content image C, forcing G to retain C's original colors.

$$L_{\text{color}} = \frac{1}{2} \left(||U(G) - U(C)||_F^2 + ||V(G) - V(C)||_F^2 \right) \tag{4}$$

This is implemented as `tf.reduce_mean((generated_yuv[..., 1:3] - content_yuv[..., 1:3])**2)`.

3 Experiments and Analysis

We implemented our system in Python using TensorFlow and Keras. The pre-trained VGG-19 model was used as our feature extractor, and the Adam optimizer was used for loss minimization. Our experiments used a photograph of a dog as the content image (C), "Composition VII" by Kandinsky as Style 1 (S_1), and "Starry Night" by Van Gogh as Style 2 (S_2).

3.1 Experiment 1: Style Interpolation

To demonstrate our first novelty, we designed an experiment to analyze the system's ability to create novel hybrid styles. We set the color preservation weight γ =0 (disabling it), thereby allowing the full color palettes of the style images to be transferred. We then varied the interpolation weight ω in discrete steps from 0.0 (pure Style 1) to 1.0 (pure Style 2). The results of this progression are shown in Figure 2.

The observations clearly demonstrate the success of our method. The output image transitions smoothly across the stylistic spectrum, not as a simple "cross-fade," but as a true blend of deep textural features.

- At ω =0.0, the system correctly reproduces a pure Kandinsky style. The content image's structure is recognizable, but it is rendered with the sharp, chaotic, and geometrically distinct elements from "Composition VII."
- At ω =1.0, the system generates a pure Van Gogh style. The image is dominated by the iconic, cohesive, and swirling brushstrokes characteristic of "Starry Night."

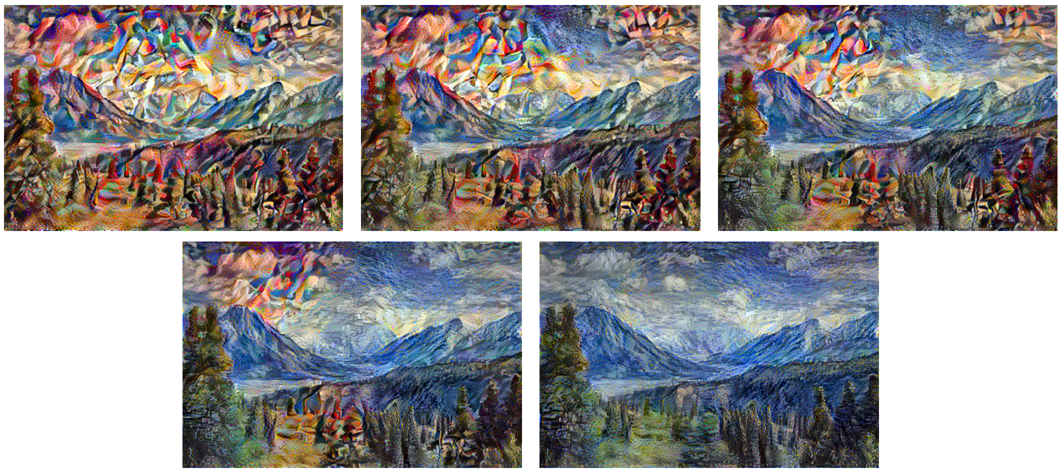

Figure 2: Results of varying the interpolation weight ω. (Top row: $\omega = 0.0, 0.25, 0.5$; bottom row: $\omega = 0.75, 1.0$)

- The intermediate images at ω =0.25, 0.5, and 0.75 are the most compelling. At ω =0.5, a clear hybrid texture is visible. We can observe the large-scale "swirling" motion from Van Gogh forming the overall flow of the image, while the "chaotic" and "geometric" elements from Kandinsky appear within those swirls. This fusion demonstrates that the interpolation is not merely an overlay but a complex merger of the learned feature statistics from both style sources.

This experiment confirms that the parameter acts as a sophisticated and effective control for blending deep textural representations, proving the success of our interpolation method..

3.2 Experiment 2: Color Preservation

To demonstrate our second novelty, we conducted an experiment to isolate the effect of our color preservation loss. We fixed the style blend at ω =0.5 (a 50/50 mix of Kandinsky and Van Gogh) and compared the results of a standard transfer (γ =0) against our color-preserved transfer (γ =10 6). The parameters are detailed in Table 1, and the visual results are shown in Figure 3.

Table 1: Parameters for Color Preservation Experiment

Experiment	α (Content)	β (Style)	ω (Blend)	γ (Color)
Standard Blend	10^4	10^{-2}	0.5	0
Color Preserved	10^4	10^{-2}	0.5	10^6

The results are unambiguous and clearly demonstrate the success of this novelty:

Figure 3(a) (Standard Blend): Without color preservation, the original color palette of the content image is completely lost. The green foliage, the brown and white tones of the building, and the blue sky are all overwritten by the dominant blues, yellows, and chaotic colors from the style images. The resulting image, while texturally interesting, has no color fidelity to the original scene.

Figure 3(b) (Color Preserved): With color preservation enabled, the system's ability to decouple texture from color is evident. The complex, blended textures—the swirls from "Starry Night" and the geometric shapes from "Composition VII"—are successfully applied to the image.

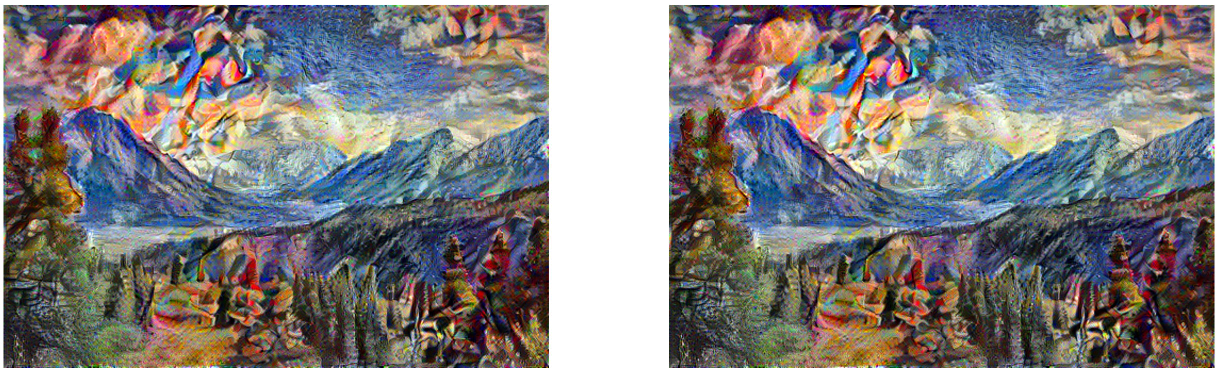

Figure 3: Effect of our Color Preservation Loss. (a) Left: Standard Blend ($\gamma = 0$). (b) Right: With Color Preservation ($\gamma = 10^6$).

However, the original color palette is rigorously maintained. The foliage remains green, the building retains its brown and white tones, and the sky stays blue.

This experiment confirms that our perceptual color loss in the YUV space is highly effective, allowing for the transfer of artistic texture without sacrificing the color realism of the content source.

4 Links and Resources

- **Our GitHub Repository:** Artistic Vision: Neural Style Transfer for Image Synthesis
- **Foundational Paper (Gatys et al.):** A Neural Algorithm of Artistic Style
- **TensorFlow Tutorial:** Neural style transfer

5 Conclusion

In this work, we presented a controllable artistic style transfer framework that enhances the traditional Neural Style Transfer (NST) algorithm by introducing two key extensions: *style interpolation* and *color preservation*. Our system allows users to dynamically blend multiple artistic styles using an interpolation weight ω, offering a continuum of stylistic outputs rather than a fixed one-to-one transfer. Furthermore, the integration of a color preservation loss in the YUV color space ensures that the generated images retain the natural color palette of the original content while adopting the textures and brushstrokes of the style images.

Experimental results demonstrated that the proposed model provides fine-grained control over stylistic effects and produces visually appealing results without distorting content structure or color. This controllability makes the approach more practical for creative applications such as digital art generation, personalized filters, and interactive design tools. Future extensions could explore real-time implementations, multimodal inputs, or user-guided semantic style transfer to further broaden its artistic potential.

A Full Implementation Code

This is the complete Python code used for this project, run in a Google Colab notebook.

```python
# Cell 1: Imports
import tensorflow as tf
import IPython.display as display
import matplotlib.pyplot as plt
import matplotlib.pyplot as mpl
import numpy as np
import PIL.Image
import time
import functools

# Set default display options for matplotlib
mpl.rcParams['figure.figsize'] = (12, 12)
mpl.rcParams['axes.grid'] = False

print("Libraries imported successfully.")

# Cell 2: Image Helper Functions

def load_img(path_to_img):
  """Loads and processes an image for the model."""
  max_dim = 512
  img = tf.io.read_file(path_to_img)
  img = tf.image.decode_image(img, channels=3)
  img = tf.image.convert_image_dtype(img, tf.float32)

  shape = tf.cast(tf.shape(img)[:-1], tf.float32)
  long_dim = max(shape)
  scale = max_dim / long_dim

  new_shape = tf.cast(shape * scale, tf.int32)

  img = tf.image.resize(img, new_shape)
  img = img[tf.newaxis, :]
  return img

def imshow(image, title=None):
  """Displays an image tensor."""
  if len(image.shape) > 3:
    image = tf.squeeze(image, axis=0)

  plt.imshow(image)
  if title:
    plt.title(title)

print("Image helper functions defined.")

# Cell 3: Load and Display Images
# This cell assumes 'content.jpg', 'style1.jpg',
# and 'style2.jpg' have been uploaded to Colab.

content_path = 'content.jpg'
style_path_1 = 'style1.jpg'
style_path_2 = 'style2.jpg'

content_image = load_img(content_path)
style_image_1 = load_img(style_path_1)
style_image_2 = load_img(style_path_2)

plt.figure(figsize=(15, 5))
plt.subplot(1, 3, 1)
imshow(content_image, 'Content Image')
plt.subplot(1, 3, 2)
imshow(style_image_1, 'Style Image 1')
```

```python
plt.subplot(1, 3, 3)
imshow(style_image_2, 'Style Image 2')

print("Images loaded and displayed successfully.")


# Cell 4: Define the Core Style and Content Model

# --- 1. Define Layer Names ---
content_layers = ['block5_conv2']
style_layers = ['block1_conv1',
                'block2_conv1',
                'block3_conv1',
                'block4_conv1',
                'block5_conv1']
num_content_layers = len(content_layers)
num_style_layers = len(style_layers)

# --- 2. Helper function to create the VGG model ---
def vgg_layers(layer_names):
  """ Creates a VGG model that returns a list of intermediate output values."""
  vgg = tf.keras.applications.VGG19(include_top=False, weights='imagenet')
  vgg.trainable = False

  outputs = [vgg.get_layer(name).output for name in layer_names]
  model = tf.keras.Model([vgg.input], outputs)
  return model

# --- 3. Gram Matrix function ---
def gram_matrix(input_tensor):
  """Calculates the Gram matrix (the key to capturing style)."""
  result = tf.linalg.einsum('bijc,bijd->bcd', input_tensor, input_tensor)
  input_shape = tf.shape(input_tensor)
  num_locations = tf.cast(input_shape[1]*input_shape[2], tf.float32)
  return result/(num_locations)

# --- 4. The main StyleContentModel class ---
class StyleContentModel(tf.keras.models.Model):
  def __init__(self, style_layers, content_layers):
    super(StyleContentModel, self).__init__()
    self.vgg = vgg_layers(style_layers + content_layers)
    self.style_layers = style_layers
    self.content_layers = content_layers
    self.num_style_layers = len(style_layers)
    self.vgg.trainable = False

  def call(self, inputs):
    "Expects float input in [0,1]"
    inputs = inputs*255.0
    preprocessed_input = tf.keras.applications.vgg19.preprocess_input(inputs)
    outputs = self.vgg(preprocessed_input)

    style_outputs, content_outputs = (outputs[:self.num_style_layers],
                                      outputs[self.num_style_layers:])

    style_outputs = [gram_matrix(style_output)
                     for style_output in style_outputs]

    content_dict = {content_name: value
                    for content_name, value
                    in zip(self.content_layers, content_outputs)}

    style_dict = {style_name: value
```

```python
                  for style_name, value
                  in zip(self.style_layers, style_outputs)}

    return {'content': content_dict, 'style': style_dict}

# --- 5. Create an instance of our model ---
extractor = StyleContentModel(style_layers, content_layers)

print("StyleContentModel and extractor defined successfully.")


# Cell 5: Define Loss Functions (Novelty 1 & 2)

# --- Novelty 2: Color Preservation Loss ---
def calculate_color_loss(generated_image, content_image):
    """Calculates the loss in color (chrominance)."""
    generated_yuv = tf.image.rgb_to_yuv(generated_image)
    content_yuv = tf.image.rgb_to_yuv(content_image)
    color_loss = tf.reduce_mean((generated_yuv[..., 1:3] - content_yuv[..., 1:3])
    **2)
    return color_loss

# --- Novelty 1: Blended Style & Content Loss ---
def calculate_style_content_loss(outputs, content_targets, style_targets_1,
    style_targets_2, interpolation_weight):
    """Calculates the blended style loss and the content loss."""

    style_outputs = outputs['style']
    content_outputs = outputs['content']

    # --- Content Loss ---
    content_loss = tf.add_n([tf.reduce_mean((content_outputs[name]-content_targets[
    name])**2)
                             for name in content_outputs.keys()])
    content_loss *= content_weight / num_content_layers

    # --- Blended Style Loss (Novelty 1) ---
    style_loss_1 = tf.add_n([tf.reduce_mean((style_outputs[name]-style_targets_1[
    name])**2)
                           for name in style_outputs.keys()])
    style_loss_2 = tf.add_n([tf.reduce_mean((style_outputs[name]-style_targets_2[
    name])**2)
                           for name in style_outputs.keys()])
    style_loss = (1.0 - interpolation_weight) * style_loss_1 + interpolation_weight
    * style_loss_2
    style_loss *= style_weight / num_style_layers

    return content_loss, style_loss

print("Loss functions defined.")


# Cell 6: Set Weights and Define the Main Training Step

# --- 1. Set Weights for all 3 Losses ---
# These values are changed for each experiment run
content_weight = 1e4
style_weight = 1e-2
interpolation_weight = 0.5 # Example value
color_weight = 1e6 # Example value

# --- 2. Pre-calculate Target Features ---
content_targets = extractor(content_image)['content']
```

```
style_targets_1 = extractor(style_image_1)['style']
style_targets_2 = extractor(style_image_2)['style']

# --- 3. Set up the Optimizer ---
opt = tf.keras.optimizers.Adam(learning_rate=0.02, beta_1=0.99, epsilon=1e-1)

# --- 4. Define the Main Training Step ---
@tf.function()
def train_step(image):
  """Performs one step of optimization on the image."""
  with tf.GradientTape() as tape:
    outputs = extractor(image)
    content_loss, style_loss = calculate_style_content_loss(outputs, A
                                                            content_targets,
                                                            style_targets_1,
                                                            style_targets_2,
                                                            interpolation_weight)
    color_loss = calculate_color_loss(image, content_image)
    total_loss = content_loss + style_loss + (color_weight * color_loss)

  grad = tape.gradient(total_loss, image)
  opt.apply_gradients([(grad, image)])
  image.assign(tf.clip_by_value(image, clip_value_min=0.0, clip_value_max=1.0))

print("Weights and train_step() function defined successfully.")


# Cell 7: Run the Final Optimization Loop

# --- 1. Define final helper functions ---
def tensor_to_image(tensor):
  """Converts a tensor back to a displayable PIL image."""
  tensor = tensor*255
  tensor = np.array(tensor, dtype=np.uint8)
  if np.ndim(tensor)>3:
    assert tensor.shape[0] == 1
    tensor = tensor[0]
  return PIL.Image.fromarray(tensor)

# --- 2. Initialize the image to be generated ---
image = tf.Variable(content_image)

# --- 3. Run the optimization loop ---
start = time.time()
epochs = 10
steps_per_epoch = 100
step = 0

for n in range(epochs):
  print(f"Epoch {n+1}/{epochs}")
  for m in range(steps_per_epoch):
    step += 1
    train_step(image)
    print(".", end='', flush=True)

  # Display the image at the end of each epoch
  img = tensor_to_image(image)
  display.display(img)

  # Save the final image of each epoch
  img.save(f"epoch_{n+1}.png")

end = time.time()
```

```python
print(f"\nTotal time: {end-start:.1f} seconds")
print("Optimization complete.")
```