

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 (α), 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 α , a user can precisely navigate the entire "stylistic spectrum" between the two artworks. For example, setting $\alpha=0.0$ applies 100% of the Kandinsky style, $\alpha=1.0$ applies 100% of the Van Gogh style, and $\alpha=0.7$ 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.

$$L_{\text{total}} = \alpha L_{\text{content}} + \beta L_{\text{style_blend}} + \gamma L_{\text{color}} \quad (1)$$

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

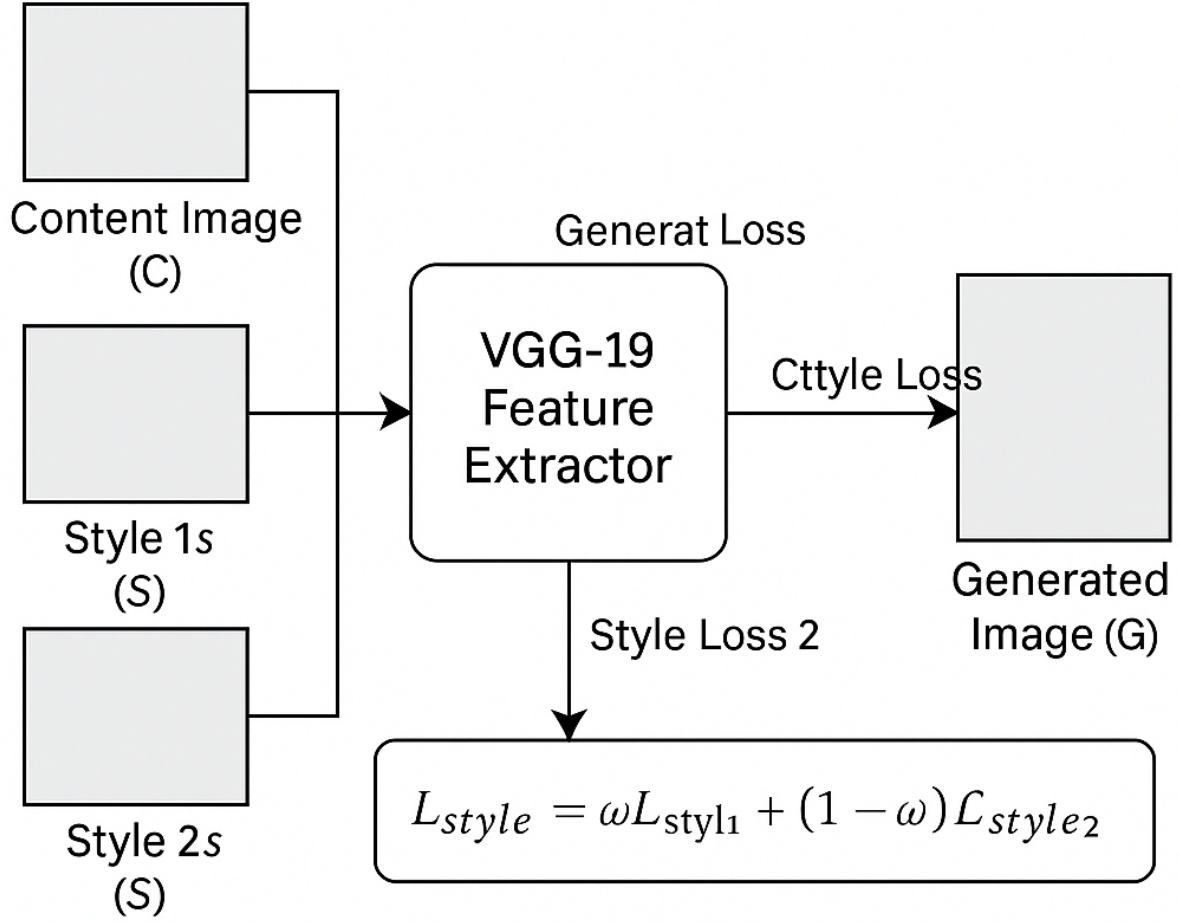


Figure 1: Overview of our proposed method

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.

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

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 $\omega \in [0, 1]$.

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

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.

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

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 $\gamma = 0$ (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 $\omega = 0.0$, 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 $\omega = 1.0$, the system generates a pure Van Gogh style. The image is dominated by the iconic, cohesive, and swirling brushstrokes characteristic of "Starry Night."

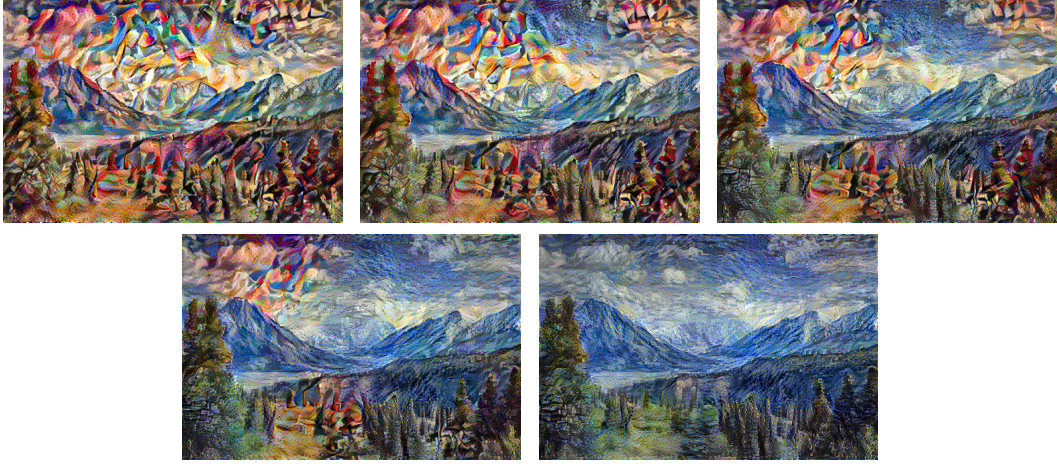


Figure 2: Results of varying the interpolation weight ω . (Top row: $\omega = 0.0, 0.25, 0.5$; bottom row: $\omega = 0.75, 1.0$)

- The intermediate images at $\omega = 0.25, 0.5$, and 0.75 are the most compelling. At $\omega = 0.5$, 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 ω 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 $\omega = 0.5$ (a 50/50 mix of Kandinsky and Van Gogh) and compared the results of a standard transfer ($\gamma = 0$) against our color-preserved transfer ($\gamma = 10^6$). 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.

```

1 # Cell 1: Imports
2 import tensorflow as tf
3 import IPython.display as display
4 import matplotlib.pyplot as plt
5 import matplotlib.pyplot as mpl
6 import numpy as np
7 import PIL.Image
8 import time
9 import functools
10
11 # Set default display options for matplotlib
12 mpl.rcParams['figure.figsize'] = (12, 12)
13 mpl.rcParams['axes.grid'] = False
14
15 print("Libraries imported successfully.")
16
17 # Cell 2: Image Helper Functions
18
19 def load_img(path_to_img):
20     """Loads and processes an image for the model."""
21     max_dim = 512
22     img = tf.io.read_file(path_to_img)
23     img = tf.image.decode_image(img, channels=3)
24     img = tf.image.convert_image_dtype(img, tf.float32)
25
26     shape = tf.cast(tf.shape(img)[: -1], tf.float32)
27     long_dim = max(shape)
28     scale = max_dim / long_dim
29
30     new_shape = tf.cast(shape * scale, tf.int32)
31
32     img = tf.image.resize(img, new_shape)
33     img = img[tf.newaxis, :]
34     return img
35
36 def imshow(image, title=None):
37     """Displays an image tensor."""
38     if len(image.shape) > 3:
39         image = tf.squeeze(image, axis=0)
40
41     plt.imshow(image)
42     if title:
43         plt.title(title)
44
45 print("Image helper functions defined.")
46
47 # Cell 3: Load and Display Images
48 # This cell assumes 'content.jpg', 'style1.jpg',
49 # and 'style2.jpg' have been uploaded to Colab.
50
51 content_path = 'content.jpg'
52 style_path_1 = 'style1.jpg'
53 style_path_2 = 'style2.jpg'
54
55 content_image = load_img(content_path)
56 style_image_1 = load_img(style_path_1)
57 style_image_2 = load_img(style_path_2)
58
59 plt.figure(figsize=(15, 5))
60 plt.subplot(1, 3, 1)
61 imshow(content_image, 'Content Image')
62 plt.subplot(1, 3, 2)
63 imshow(style_image_1, 'Style Image 1')

```

```

64 plt.subplot(1, 3, 3)
65 imshow(style_image_2, 'Style Image 2')
66
67 print("Images loaded and displayed successfully.")
68
69
70 # Cell 4: Define the Core Style and Content Model
71
72 # --- 1. Define Layer Names ---
73 content_layers = ['block5_conv2']
74 style_layers = ['block1_conv1',
75                'block2_conv1',
76                'block3_conv1',
77                'block4_conv1',
78                'block5_conv1']
79 num_content_layers = len(content_layers)
80 num_style_layers = len(style_layers)
81
82 # --- 2. Helper function to create the VGG model ---
83 def vgg_layers(layer_names):
84     """ Creates a VGG model that returns a list of intermediate output values. """
85     vgg = tf.keras.applications.VGG19(include_top=False, weights='imagenet')
86     vgg.trainable = False
87
88     outputs = [vgg.get_layer(name).output for name in layer_names]
89     model = tf.keras.Model([vgg.input], outputs)
90     return model
91
92 # --- 3. Gram Matrix function ---
93 def gram_matrix(input_tensor):
94     """Calculates the Gram matrix (the key to capturing style)."""
95     result = tf.linalg.einsum('bijc,bijd->bcd', input_tensor, input_tensor)
96     input_shape = tf.shape(input_tensor)
97     num_locations = tf.cast(input_shape[1]*input_shape[2], tf.float32)
98     return result/(num_locations)
99
100 # --- 4. The main StyleContentModel class ---
101 class StyleContentModel(tf.keras.models.Model):
102     def __init__(self, style_layers, content_layers):
103         super(StyleContentModel, self).__init__()
104         self.vgg = vgg_layers(style_layers + content_layers)
105         self.style_layers = style_layers
106         self.content_layers = content_layers
107         self.num_style_layers = len(style_layers)
108         self.vgg.trainable = False
109
110     def call(self, inputs):
111         "Expects float input in [0,1]"
112         inputs = inputs*255.0
113         preprocessed_input = tf.keras.applications.vgg19.preprocess_input(inputs)
114         outputs = self.vgg(preprocessed_input)
115
116         style_outputs, content_outputs = (outputs[:self.num_style_layers],
117                                         outputs[self.num_style_layers:])
118
119         style_outputs = [gram_matrix(style_output)
120                         for style_output in style_outputs]
121
122         content_dict = {content_name: value
123                        for content_name, value
124                        in zip(self.content_layers, content_outputs)}
125
126         style_dict = {style_name: value

```



```

127         for style_name, value
128             in zip(self.style_layers, style_outputs)}
129
130     return {'content': content_dict, 'style': style_dict}
131
132 # --- 5. Create an instance of our model ---
133 extractor = StyleContentModel(style_layers, content_layers)
134
135 print("StyleContentModel and extractor defined successfully.")
136
137
138 # Cell 5: Define Loss Functions (Novelty 1 & 2)
139
140 # --- Novelty 2: Color Preservation Loss ---
141 def calculate_color_loss(generated_image, content_image):
142     """Calculates the loss in color (chrominance)."""
143     generated_yuv = tf.image.rgb_to_yuv(generated_image)
144     content_yuv = tf.image.rgb_to_yuv(content_image)
145     color_loss = tf.reduce_mean((generated_yuv[..., 1:3] - content_yuv[..., 1:3])
146                                 **2)
147     return color_loss
148
149 # --- Novelty 1: Blended Style & Content Loss ---
150 def calculate_style_content_loss(outputs, content_targets, style_targets_1,
151                                 style_targets_2, interpolation_weight):
152     """Calculates the blended style loss and the content loss."""
153
154     style_outputs = outputs['style']
155     content_outputs = outputs['content']
156
157     # --- Content Loss ---
158     content_loss = tf.add_n([tf.reduce_mean((content_outputs[name]-content_targets[
159     name])**2)
160                             for name in content_outputs.keys()])
161     content_loss *= content_weight / num_content_layers
162
163     # --- Blended Style Loss (Novelty 1) ---
164     style_loss_1 = tf.add_n([tf.reduce_mean((style_outputs[name]-style_targets_1[
165     name])**2)
166                             for name in style_outputs.keys()])
167     style_loss_2 = tf.add_n([tf.reduce_mean((style_outputs[name]-style_targets_2[
168     name])**2)
169                             for name in style_outputs.keys()])
170     style_loss = (1.0 - interpolation_weight) * style_loss_1 + interpolation_weight
171     * style_loss_2
172     style_loss *= style_weight / num_style_layers
173
174     return content_loss, style_loss
175
176 print("Loss functions defined.")
177
178
179 # Cell 6: Set Weights and Define the Main Training Step
180
181 # --- 1. Set Weights for all 3 Losses ---
182 # These values are changed for each experiment run
183 content_weight = 1e4
184 style_weight = 1e-2
185 interpolation_weight = 0.5 # Example value
186 color_weight = 1e6 # Example value
187
188 # --- 2. Pre-calculate Target Features ---
189 content_targets = extractor(content_image)['content']

```

```

184 style_targets_1 = extractor(style_image_1)['style']
185 style_targets_2 = extractor(style_image_2)['style']
186
187 # --- 3. Set up the Optimizer ---
188 opt = tf.keras.optimizers.Adam(learning_rate=0.02, beta_1=0.99, epsilon=1e-1)
189
190 # --- 4. Define the Main Training Step ---
191 @tf.function()
192 def train_step(image):
193     """Performs one step of optimization on the image."""
194     with tf.GradientTape() as tape:
195         outputs = extractor(image)
196         content_loss, style_loss = calculate_style_content_loss(outputs, A
197                                                                 content_targets,
198                                                                 style_targets_1,
199                                                                 style_targets_2,
200                                                                 interpolation_weight)
201         color_loss = calculate_color_loss(image, content_image)
202         total_loss = content_loss + style_loss + (color_weight * color_loss)
203
204     grad = tape.gradient(total_loss, image)
205     opt.apply_gradients([(grad, image)])
206     image.assign(tf.clip_by_value(image, clip_value_min=0.0, clip_value_max=1.0))
207
208 print("Weights and train_step() function defined successfully.")
209
210
211 # Cell 7: Run the Final Optimization Loop
212
213 # --- 1. Define final helper functions ---
214 def tensor_to_image(tensor):
215     """Converts a tensor back to a displayable PIL image."""
216     tensor = tensor*255
217     tensor = np.array(tensor, dtype=np.uint8)
218     if np.ndim(tensor)>3:
219         assert tensor.shape[0] == 1
220         tensor = tensor[0]
221     return PIL.Image.fromarray(tensor)
222
223 # --- 2. Initialize the image to be generated ---
224 image = tf.Variable(content_image)
225
226 # --- 3. Run the optimization loop ---
227 start = time.time()
228 epochs = 10
229 steps_per_epoch = 100
230 step = 0
231
232 for n in range(epochs):
233     print(f"Epoch {n+1}/{epochs}")
234     for m in range(steps_per_epoch):
235         step += 1
236         train_step(image)
237         print(".", end='', flush=True)
238
239     # Display the image at the end of each epoch
240     img = tensor_to_image(image)
241     display.display(img)
242
243     # Save the final image of each epoch
244     img.save(f"epoch_{n+1}.png")
245
246 end = time.time()

```

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