Question 2

In this question, we will compare two different methods of Neural Style Transfer (NST). The first method is the so-called 'Gram Matrix' style transfer method, and the second one is the Sliced Wasserstein Distance style transfer method. We will perform several experiments using different content and style images, and include additional numerical metrics to compare image quality. We will also propose different visualization to understand both methods.

The Google Colab notebook is: here.

1 - What is NST?

First of all, we should understand what is Neural Style Transfer.

Neural Style Transfer is a technique in Deep Learning that generates a new image by combining the content of one image with the style of another. It allows artists to create new images that retain the main structures of the content image while adopting the aesthetic style (such as textures, colors, and patterns) of the style image. For instance, the image below has been created using the content image of Mona Lisa, with the style of *The Starry Night* from van Gogh, using NST.

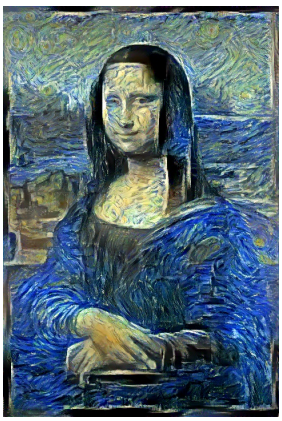


Figure 1: **Mona Lisa with the style of *The Starry Night* from van Gogh.** Source: Wikipedia

This technique leverages Convolutional Neural Networks (CNNs) by using their internal feature maps representation to combine the content and the style images. Feature maps correspond to the outputs of the various layers within a CNN, representing the responses of the filters applied across the input image or the preceding layer's output. Each filter in a CNN is designed to detect specific types of features at various levels of abstraction, such as edges, textures, patterns, or more complex shapes and objects in deeper layers.

NST has been introduced in 2015 by Gatys, Leon A., Alexander S. Ecker, and Matthias Bethge in the their famous paper '*A neural algorithm of artistic style*'. In this paper, they introduce a way to separate and recombine the content and style of images by utilizing the features maps extracted by a pre-trained CNN model (e.g. VGG-19), as well as Gram matrices.

More recently, in 2021, *Eric Heitz et al.* in their paper: *A Sliced Wasserstein Loss for Neural Texture Synthesis* tried to improve this Gram-matrix based technique by introducing a new metric: the Sliced Wasserstein Distance, that is especially well suited for texture synthesis in generated images.

2 - The Gram Matrix style transfer method

In this section, I will explain in details the Gram Matrix style transfer method, but instead of using long (and boring) sentences, I have prepared several visualisations and plots that I think will be useful to understand all the concepts involved here. Indeed, when I was looking for explanations and visualizations about the Gram Matrix NST method, I was a bit frustrated because I couldn't find a visualisation that clearly explained the technique. I have therefore decided to create my own visualization to explain graphically what is the technique, and how and why does it work. [1]

Let's begin with a general overview of the Gram Matrix NST technique:

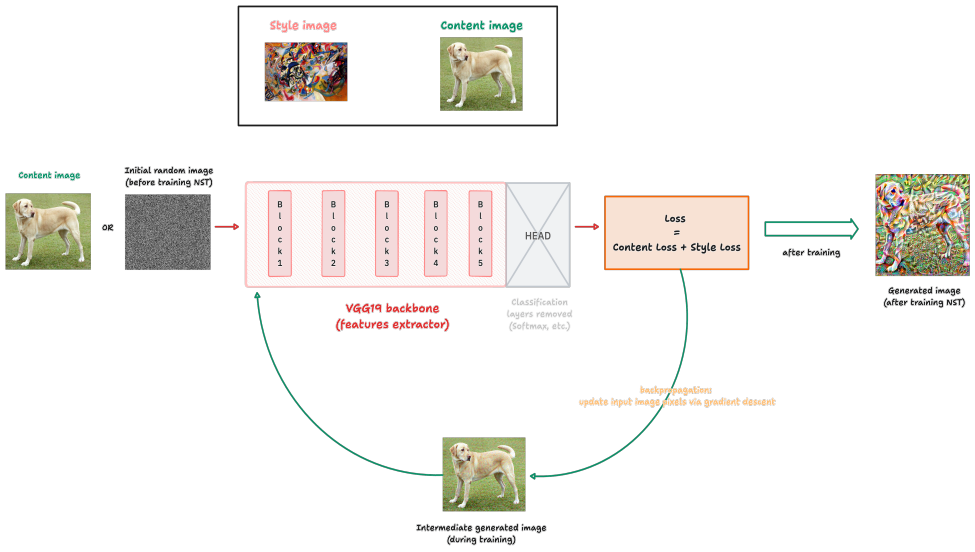


Figure 2: **Overview of NST with Gram Matrix method**

[1] NB: All these visualisations will be saved on a public repository on my personal GitHub, which I hope will help other people to understand these style transfer methods.

Everything starts with two images: the content image and the style image. In this example, we use the content image of Yellow Labrador and a style image from Vassily Kandinsky (on the top of Figure 2). Then, we use a pre-trained CNN (here a VGG-19 features extractor), and we remove the 'head', i.e. all the layers performing the classification task (i.e. Softmax layers, etc.). The input of the NST model is either the content image, or an initial random image.[2] The NST model is then trained via backpropagation to update the input image pixels via gradient descent in order to minimize the total loss (composed of the 'content loss' and the 'style loss'). Therefore, at the end of the training, we obtain the stylized generated image, having the content of the content image (here the Yellow Labrador) and the style of the style image (here the style of Vassily Kandinsky)! (cf. right hand side on Figure 2). However, it seems that we are missing something here... How are the 'content loss' and the 'style loss' calculated?

To really understand the Gram Matrix style transfer method, we must understand how the loss of the network is calculated. For that, we have to address two problems:

1. How can we generate an image that contains a content as in the content image? We will leverage the CNN and especially their feature maps that capture spatial information of an image without directly containing the style information. For instance, the tensorflow tutorial notebook uses the $block5\_conv2$ of the VGG-19 model as a content layer, as deeper layers tend to contain high-level features about the input image. Thus, we can simply compute the MSE loss between the input image's features and content image's features from this specific layer: this is our **content loss** (shown in green and light gray in Figure 10).

2. But we still have another issue. How can we generate an image which contains the style as in the style image? To extract the style of an image, we will use the Gram matrix. The Gram matrix captures the internal correlations among the feature maps at various layers of a CNN. By computing the Gram matrix for each selected layer in the network, we can quantify the patterns and textures that are characteristic of the image's style. From that, we can compute the MSE loss between the Gram matrix of the input image and the style image: this is our **style loss** (shown in purple and light gray in Figure 10).

Finally, these two metrics are combined together in a single loss metric (the Total loss), with corresponding weights depending if we want to add more importance on the style, or on the content, according to the artist's choice.

[2] Using the content image as the initial image for the model speeds up the training, since the model has to focus more on decreasing the style loss.

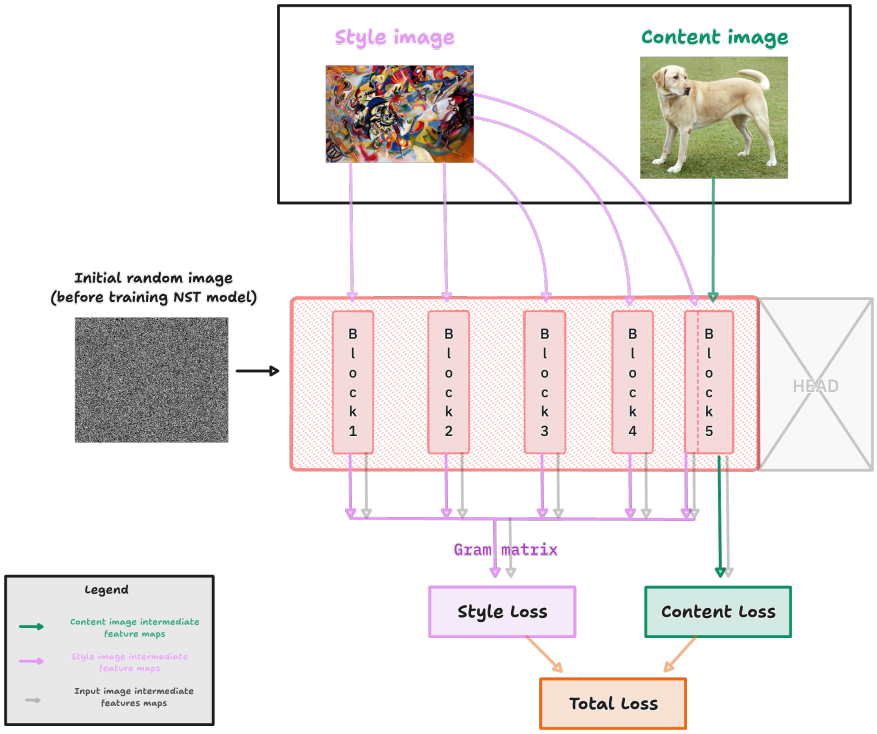


Figure 3: **Focus on the loss computation for NST using Gram Matrix method.**
Source: Personal Production

Analysis of results

To ensure that the results are comparable, we introduce two numerical metrics in order to compare image quality: SSIM (Structural Similarity Index Measure) and PSNR (Peak Signal-to-Noise Ratio). These two metrics allow us to evaluate the effectiveness of the style transfer process by quantifying the visual similarity and fidelity of the newly generated images compared to the original content image. On the one hand, SSIM provides a measure of similarity in terms of structure, brightness, and contrast, telling how well the visual quality of the content image is preserved. PSNR, on the other hand, quantifies the ratio of the maximum possible signal power to the power of corrupting noise, measuring the noise level introduced by the style transfer procedure.

1. Yellow Labrador with Vassily Kandinsky style

Our first experiment uses the Yellow Labrador content image and the style from Kandinsky to generate a new stylized image, in the same way than the

Tensorflow tutorial notebook.

Figure 4: **Content and Style images used for the first experiment**

For this experiment, we used the following weights in the loss function:

$$style_{weight} = 1e^{-2}$$

$$content_{weight} = 1e^{3}$$

This means that we want the content of the image to have more importance on the style. After training, we obtain the following result:

Figure 5: **Result obtained using the Yellow Labrador as content image**

Visually, the result is interesting, as Kandinsky's style can be seen in the original image. However, we must use numerical metrics to assess the quality of the style transfer. The table 4 displays the SSIM and PSNR metrics for this experiment:

Metric	Value
SSIM	0.275
PSNR	13.348 dB

Table 1: Labrador image quality metrics for NST using Gram Matrix Method

2. Eiffel tower with van Ghogh style

As a french exchange student from Paris, I feel quite obliged to try a style transfer experiment with an image of the Eiffel tower. That's why I used the following images for this second experiment:

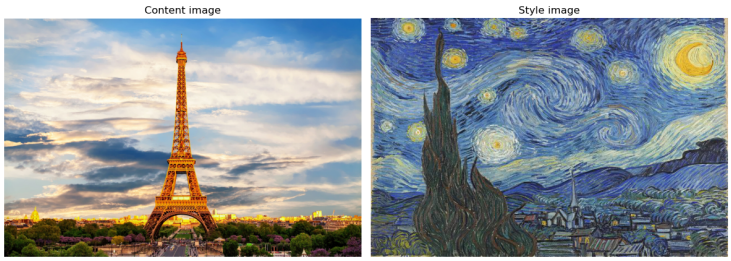


Figure 6: **Content and Style images used for the second experiment**

I obtained the following result which seems reaaly good.

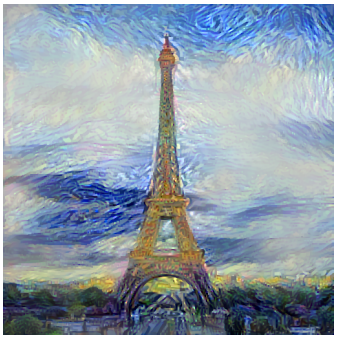


Figure 7: **Result obtained using the Eiffel Tower content image**

Here are the corresponding image quality metrics:

The numerical values for the image quality metrics are much more higher for the second experiment! This can be visually observable since the Eiffel Tower generated image seems like a real painting! Indeed, I noticed (based on the several experiments conducted in the context of this assignment) that neural style transfer works very well when the content and the style images look quite the same.

Metric	Value
SSIM	0.419
PSNR	15.092 dB

Table 2: Eiffel Tower image quality metrics for NST using Gram Matrix Method

Precisely in the second experiment, the image of the Eiffel Tower is in fact very similar to the style image: both are images seen from above, with a view of the city and a large part of the image dedicated to the sky. I believe this can explain why the NST worked much better for the second experiment.

However, one might keep in mind that the performance of a Neural Style transfer experiment is subjective and depends on the artist's intentions. In fact, some artists will want the generated image to be faithful to the original content image, while others will prefer the generated image to be more abstract by containing more style. Therefore, there is no 'absolute' response here.

3 - The Sliced Wasserstein Distance style transfer method

The Sliced Wasserstein Distance (SWD) loss has been introduced as an extension of the Gram Matrix loss for NST and texture synthesis. This technique relies on the Wasserstein distance, which intuitively represents the minimum cost of transforming one distribution into another. The Sliced Wasserstein Distance corresponds to a computationally efficient alternative by projecting high-dimensional distributions onto a set of random one-dimensional lines (slices) and then computing the Wasserstein distance in this one-dimensional space, where it is much simpler and faster to calculate.

As for the Gram Matrix method, the SWD method uses the features maps from a pre-trained CNN to extract information from both the style and content image with respect to the generated image. Indeed, for each layer's feature maps, the Sliced Wasserstein Loss is computed by projecting the high-dimensional feature vectors onto random one-dimensional lines. This projection is done by multiplying the feature vectors by random unit vectors. After projection, we sort the resulting one-dimensional representations and compute the Wasserstein distance between the corresponding sorted lists from the style and generated images. The figure below shows the principle of the Sliced Wasserstein Loss:

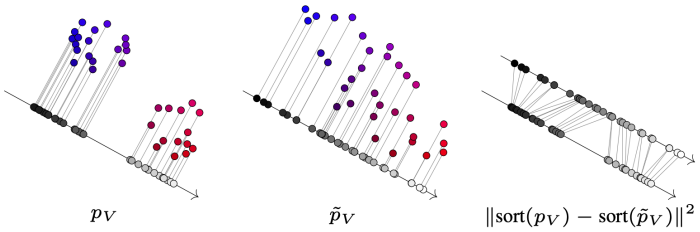


Figure 8: **Principle of the Sliced Wasserstein Loss.**
Source: *Eric Heitz et al. - A Sliced Wasserstein Loss for Neural Texture Synthesis*

As we can see, the high-dimensional features (blue and red dots) are projected onto random directions, and then sorted to compute the difference between the two distributions. This process is then repeated for many random slices, and the resulting distances are averaged to obtain the Sliced Wasserstein Distance.

Analysis of results

1. Yellow Labrador with Vassily Kandinsky style

After performing the Sliced Wassertein Method on the Yellow Labrador content image with Vassily Kandinsky style, we obtain the following results:

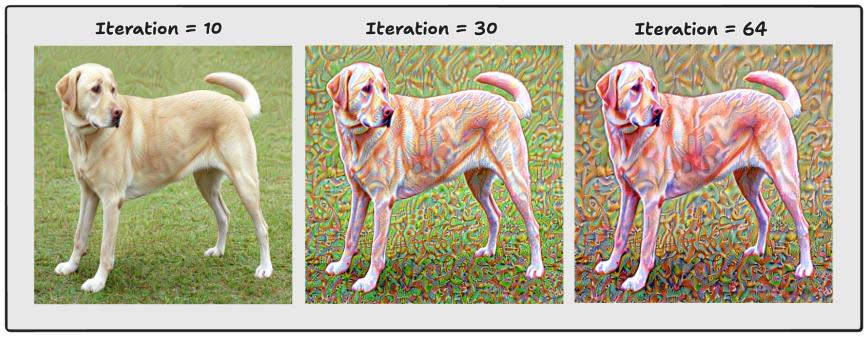


Figure 9: **Output of the style transfer for the Yellow Labrador content image, at iteration 10, 30 and 64 (end of training).**
Source: Personal Production

The table below shows the image quality metrics obtained with the SWD method for this experiment:

Metric	Value
SSIM	0.522
PSNR	16.136 dB

Table 3: Labrador image quality metrics for NST using SWD Method

As we can see, the image quality metrics have much higher score than for the Gram Matrix method. It means that the SWD style transfer method performs better on this specific images (content and style), but one have to remember that the interpretation of these metrics depends on the choice and the artist's wish.

2. Eiffel tower with van Ghogh style

The results obtained are shown below:

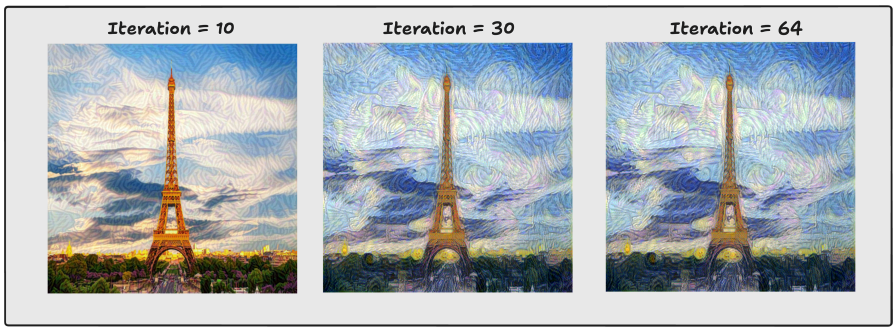


Figure 10: **Output of the style transfer for the Eiffel Tower content image, at iteration 10, 30 and 64 (end of training).**
Source: Personal production

The image metrics for this experiment are shown below:

Metric	Value
SSIM	0.597
PSNR	16.321 dB

Table 4: Eiffel Tower image quality metrics for NST using SWD Method

In this case also, the SWD method improved the image quality metrics!

Conclusion

In conclusion, this work has allowed us to understand and compare two transfer style methods. Both methods perform well in transferring the style

of one image to the content of another image. However, we noticed that the SWD method performs better in terms of runtime, image quality performance and visual rendering. This method seems also to represent texture of the style image much better than the Gram Matrix method.

The results of this work can be extended to other experiments (i.e. using other content and style images) and improved by increasing computing power. In fact, I ran out of GPU-credit on Colab and was thus constraint to run other experiment locally, on a cpu.

SYDE572_A5_style_transfer_SIFAOUI

March 30, 2024

1 SYDE 572 Winter 2024 Assignment 5: Sofiane SIFAOUI (Student ID: 21114530)

1.1 Neural Style Transfer: Comparison of two methods | Gram Matrix and Sliced Wasserstein Distance

In this Assignment, we will compare two different methods of neural style transfer. In this "tutorial" notebook, we will show how to use these two different methods for style transfer and also compare and analyze the results obtained. We will also use additional numerical metrics to compare image quality, between the (initial) content image and the resulting style image. Inspired by https://www.tensorflow.org/tutorials/generative/style_transfer and Anis Ayari GitHub

Instructions: If you want to experiment with your own images, you will have to create a public GitHub repo, as done in it's notebook. Moreover, if you want to train the SWD style transfer loss, you will have to load the vgg19.pth file from the official implementation on GitHub (A-Sliced-Wasserstein-Loss-for-Neural-Texture-Synthesis)

2 Imports and utils functions

[1]:
```
import os
os.environ['TFHUB_MODEL_LOAD_FORMAT'] = 'COMPRESSED'
!pip install -q git+https://github.com/tensorflow/docs

# Utils
import time
import IPython.display as display
import numpy as np
import glob
import time
import functools

# Viz
import matplotlib.pyplot as plt
import matplotlib as mpl
mpl.rcParams['figure.figsize'] = (12, 12)
mpl.rcParams['axes.grid'] = False
from matplotlib import gridspec
```

```
# Image processing
import imageio
import PIL.Image

# Models
import tensorflow as tf
import tensorflow_hub as hub # if needed can use Fast Style Transfer (https://
 ↪www.tensorflow.org/hub/tutorials/tf2_arbitrary_image_stylization)
```

```
  Preparing metadata (setup.py) … done
  Building wheel for tensorflow-docs (setup.py) … done
```

[2]:
```
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 PIL.Image.fromarray(tensor)

def crop_center(image):
  """Returns a cropped square image."""
  shape = image.shape
  new_shape = min(shape[1], shape[2])
  offset_y = max(shape[1] - shape[2], 0) // 2
  offset_x = max(shape[2] - shape[1], 0) // 2
  image = tf.image.crop_to_bounding_box(
      image, offset_y, offset_x, new_shape, new_shape)
  return image


@functools.lru_cache(maxsize=None)
def load_image(image_url,image_name, image_size=(256, 256),␣
 ↪preserve_aspect_ratio=True):
  """Loads and preprocesses images."""
  # Cache image file locally.
  print(os.path.basename(image_url))
  image_path = tf.keras.utils.get_file(os.path.basename(image_url), image_url)
  # Load and convert to float32 numpy array, add batch dimension, and normalize␣
 ↪to range [0, 1].
  img = plt.imread(image_path)[:,:,:3].astype(np.float32)[np.newaxis, ...]
  print(img.shape)
  if img.max() > 1.0:
    img = img / 255.
  if len(img.shape) == 3:
    img = tf.stack([img, img, img], axis=-1)
```

```
  img = crop_center(img)
  img = tf.image.resize(img, image_size, preserve_aspect_ratio=True)
  return img


def show_n(images, titles=('',), download=False):
  n = len(images)
  image_sizes = [image.shape[1] for image in images]
  w = (image_sizes[0] * 6) // 320
  plt.figure(figsize=(w  * n, w))
  gs = gridspec.GridSpec(1, n, width_ratios=image_sizes)
  for i in range(n):
    plt.subplot(gs[i])
    plt.imshow(images[i][0], aspect='equal')
    plt.axis('off')
    plt.title(titles[i] if len(titles) > i else '')
  fig = plt.gcf()
  fig.savefig('all_images_generated_fast.png')
  plt.show()
  if download == True:
    try:
      from google.colab import files
    except ImportError:
      pass
    else:
      files.download('all_images_generated_fast.png')
```

3 Choice of content and style images

[3]:
```
# @title Load content images  { display-mode: "form" ,  run: "auto" }

github_content_image_folder_url = 'https://raw.githubusercontent.com/
 ↪Sofiane-Sif/StyleTransfer/main/images/content/'  # @param {type:"string"}

content_image_name= 'YellowLabradorLooking_new.jpg'  #@param ["eiffel.
 ↪jpg","iron_man.jpeg","mona_lisa.jpeg", "paris.jpeg", "pikachu.jpeg",␣
 ↪"quartier-japonais.jpg", "san_francisco_bridge.jpeg",␣
 ↪"YellowLabradorLooking_new.jpg"]
output_image_size1 = 512  # @param {type:"integer"}
output_image_size2 = 512  # @param {type:"integer"}

# The content image size can be arbitrary.
content_img_size = (output_image_size1, output_image_size2)

content_image_url = os.path.join(github_content_image_folder_url,␣
 ↪content_image_name)
```

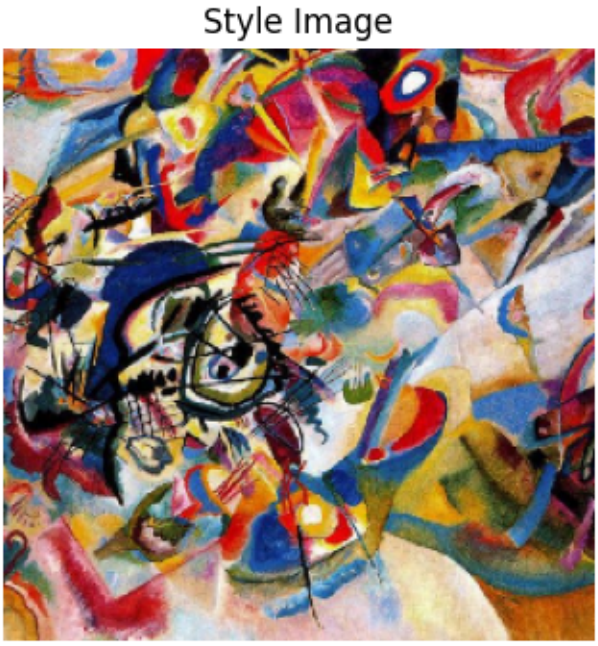
```
content_image = load_image(content_image_url, content_img_size)
show_n([content_image], ['Content Image'])
```

```
YellowLabradorLooking_new.jpg
Downloading data from https://raw.githubusercontent.com/Sofiane-
Sif/StyleTransfer/main/images/content/YellowLabradorLooking_new.jpg
83281/83281 [==============================] - 0s 0us/step
(1, 577, 700, 3)
```

[4]:
```
# @title Load style image  { display-mode: "form" ,  run: "auto" }

github_image_folder_url = 'https://raw.githubusercontent.com/Sofiane-Sif/
 ↪StyleTransfer/main/images/style/'  # @param {type:"string"}

style_image_name = 'Vassily.jpeg'  #@param ["guernica.jpg", "hiroglyphs.jpeg",␣
 ↪"monet_water_lilies.jpg", "mosaique_grecque.jpeg", "persistance_dali.jpeg",␣
 ↪"picasso_portrait.jpeg", "picasso.jpeg", "van_gogh_nuit.jpeg", "Vassily.
 ↪jpeg"]
output_image_size = 512  # @param {type:"integer"}

# The style prediction model was trained with image size 256 and it's the
# recommended image size for the style image (though, other sizes work as
# well but will lead to different results).
style_img_size = (output_image_size, output_image_size)  # Recommended to keep␣
 ↪it at 256.
```

```
style_image_url = os.path.join(github_image_folder_url, style_image_name)

style_image = load_image(style_image_url, style_img_size)
show_n([style_image], ['Style Image'])
```

```
Vassily.jpeg
Downloading data from https://raw.githubusercontent.com/Sofiane-
Sif/StyleTransfer/main/images/style/Vassily.jpeg
195196/195196 [==============================] - 0s 0us/step
(1, 657, 1000, 3)
```



4 Visualize model's architecture

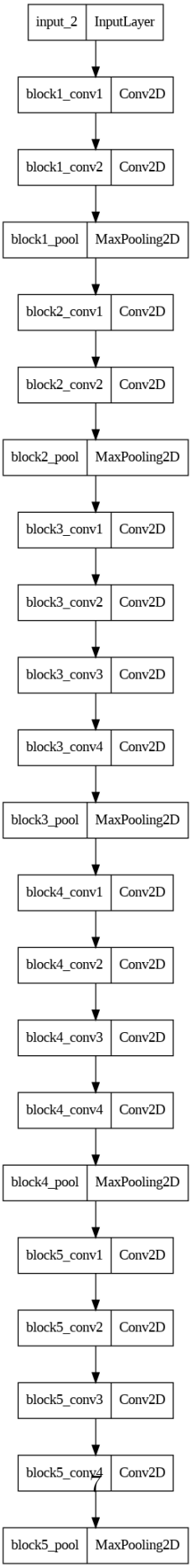
We will use the VGG19 architecture, but we will remove its head (i.e. the classification layers such as Softmax, etc.). Let's take a look at its architecture.

[5]:
```
# Print layers names for the VGG19 architecture (trained on ImageNet) only for␣
↪the backbone:

print("VGG19 layers names: \n")
for layer in tf.keras.applications.VGG19(include_top=False, weights='imagenet').
↪layers:
    print(layer.name)

print("\n")
```

```
# Plot the model and visualize its corresponding layers:
tf.keras.utils.plot_model(
    tf.keras.applications.VGG19(include_top=False, weights='imagenet'))
```

```
VGG19 layers names:

Downloading data from https://storage.googleapis.com/tensorflow/keras-
applications/vgg19/vgg19_weights_tf_dim_ordering_tf_kernels_notop.h5
80134624/80134624 [==============================] - 1s 0us/step
input_1
block1_conv1
block1_conv2
block1_pool
block2_conv1
block2_conv2
block2_pool
block3_conv1
block3_conv2
block3_conv3
block3_conv4
block3_pool
block4_conv1
block4_conv2
block4_conv3
block4_conv4
block4_pool
block5_conv1
block5_conv2
block5_conv3
block5_conv4
block5_pool
```

[5]:


input_2 InputLayer
block1_conv1 Conv2D
block1_conv2 Conv2D
block1_pool MaxPooling2D
block2_conv1 Conv2D
block2_conv2 Conv2D
block2_pool MaxPooling2D
block3_conv1 Conv2D
block3_conv2 Conv2D
block3_conv3 Conv2D
block3_conv4 Conv2D
block3_pool MaxPooling2D
block4_conv1 Conv2D
block4_conv2 Conv2D
block4_conv3 Conv2D
block4_conv4 Conv2D
block4_pool MaxPooling2D
block5_conv1 Conv2D
block5_conv2 Conv2D
block5_conv3 Conv2D
block5_conv4 Conv2D
block5_pool MaxPooling2D

#Define Model

4.0.1 Choose intermediate layers from the network to represent the style and content of the image:

[6]:
```
# We use the same content and style layers than Tensorflow tutorial notebook
# but one can experimenent with other layers, e.g. content_layers =␣
 ↪'block5_conv3'

style_layers = ['block1_conv1',
                'block2_conv1',
                'block3_conv1',
                'block4_conv1',
                'block5_conv1']

content_layers = ['block5_conv2']

num_content_layers = len(content_layers)
num_style_layers = len(style_layers)
```

[7]:
```
def vgg_layers(layer_names):
  """ Creates a VGG model that returns a list of intermediate output values."""
  # Load our model. Load pretrained VGG, trained on ImageNet data
  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
```

4.0.2 Gram Matrix implementation (cf. write for explanations and visualization) and loss definition

[8]:
```
def gram_matrix(input_tensor):
  channels = int(input_tensor.shape[-1])
  a = tf.reshape(input_tensor, [-1, channels])
  n = tf.shape(a)[0]
  gram = tf.matmul(a, a, transpose_a=True)
  return gram / tf.cast(n, tf.float32)
```

[9]:
```
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 for style_name, value in
                     zip(self.style_layers, style_outputs)}


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

#Training and generation

[10]:
```
extractor = StyleContentModel(style_layers, content_layers)

style_targets = extractor(style_image)['style']
content_targets = extractor(content_image)['content']
```

[11]:
```
!rm stylized-image-*.png

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

# Combination between content and style. Do we want something more similar to␣
 ↪the
# content image, or very similar to the style but therefore less similar to␣
 ↪content?
style_weight=1e-2
content_weight=1e3

def style_content_loss(outputs):
```

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

    style_loss = tf.add_n([tf.reduce_mean(tf.
 ↪square((style_outputs[name]-style_targets[name])))
                           for name in style_outputs.keys()])
    style_loss *= style_weight / num_style_layers

    content_loss = tf.add_n([tf.reduce_mean(tf.
 ↪square((content_outputs[name]-content_targets[name])))
                             for name in content_outputs.keys()])
    content_loss *= content_weight / num_content_layers

    loss = style_loss + content_loss
    return loss

opt = tf.optimizers.Adam(learning_rate=0.01, beta_1=0.99, epsilon=1e-1)

total_variation_weight=50
@tf.function()
def train_step(image):
  with tf.GradientTape() as tape:
    outputs = extractor(image)
    loss = style_content_loss(outputs)
    loss += total_variation_weight*tf.image.total_variation(image)

  grad = tape.gradient(loss, image)
  opt.apply_gradients([(grad, image)])
  image.assign(clip_0_1(image))

image = tf.Variable(content_image)
#image = tf.Variable(np.random.rand(1,256,256,3))

epochs = 10
steps_per_epoch = 100
max_step = epochs * steps_per_epoch
step_to_save = max_step*0.1

step = 0
for n in range(epochs):
  for m in range(steps_per_epoch):
    step += 1
    train_step(image)
    print(".", end='')
    if step%step_to_save ==0:
        print(f'STEP: {step} SAVING..')
        file_name = f'stylized-image-{step}.png'
```

```
        tensor_to_image(image).save(file_name)
  display.clear_output(wait=True)
  display.display(tensor_to_image(image))
  print("Train step: {}".format(step))
```

```
Train step: 1000
```

4.1 Sliced Wasserstein Loss NST method

[3]:
```
##############################################################################
# Implementation of
# A Sliced Wasserstein Loss for Neural Texture Synthesis
# Heitz et al., CVPR 2021
##############################################################################

import os
import numpy as np
import torch
import imageio
from skimage.transform import resize
import matplotlib.pyplot as plt

SCALING_FACTOR = 1

device = 'cuda' if torch.cuda.is_available() else "cpu"
```

```
print(device)

def saveImage(filename, image):
    imageTMP = np.clip(image * 255.0, 0, 255).astype('uint8')
    imageio.imwrite(filename, imageTMP)

def loadImage(filename, resize_to=None):
    image = imageio.v2.imread(filename).astype("float32")[:, :, 0:3] / 255.0

    # If `resize_to` dimensions are provided, resize the image
    if resize_to is not None:
        image = resize(image, resize_to, anti_aliasing=True)
    image = image[np.newaxis, ...]
    return image

def display_image(img: torch.tensor):
    img = np.array(img.cpu().detach().squeeze(0))
    img = img.transpose(2,1,0)
    plt.imshow(img)
    plt.axis("off")
    plt.show()
    plt.close()
```

cuda

[4]:
```
# @title Load style image  { display-mode: "form" ,  run: "auto" }

github_image_folder_url = 'https://raw.githubusercontent.com/Sofiane-Sif/
 ↪StyleTransfer/main/images/style/'  # @param {type:"string"}

style_image_name = 'Vassily.jpeg'  #@param ["guernica.jpg", "hiroglyphs.jpeg",␣
 ↪"monet_water_lilies.jpg", "mosaique_grecque.jpeg", "persistance_dali.jpeg",␣
 ↪"picasso_portrait.jpeg", "picasso.jpeg", "van_gogh_nuit.jpeg", "Vassily.
 ↪jpeg"]
output_image_size = 512  # @param {type:"integer"}

style_img_size = (output_image_size, output_image_size)  # Recommended to keep␣
 ↪it at 256.

style_image_url = os.path.join(github_image_folder_url, style_image_name)

style_image_example = loadImage(style_image_url, style_img_size)
style_image_example = np.swapaxes(style_image_example, 1, 3)
style_image_example = torch.from_numpy(style_image_example)
style_image_example = style_image_example.to(torch.device(device))
style_image_example = style_image_example[:, :, 0:(style_image_example.shape[2]/
 ↪/8)*8, 0:(style_image_example.shape[3]//8)*8]
```

```
display_image(style_image_example)
```

[5]:
```
# @title Load content image  { display-mode: "form" ,  run: "auto" }

github_content_image_folder_url = 'https://raw.githubusercontent.com/
 ↪Sofiane-Sif/StyleTransfer/main/images/content/'   # @param {type:"string"}

content_image_name= 'YellowLabradorLooking_new.jpg'  #@param ["eiffel.
 ↪jpg","iron_man.jpeg","mona_lisa.jpeg", "paris.jpeg", "pikachu.jpeg",␣
 ↪"quartier-japonais.jpg", "san_francisco_bridge.jpeg",␣
 ↪"YellowLabradorLooking_new.jpg"]
output_image_size1 = 512  # @param {type:"integer"}
output_image_size2 = 512  # @param {type:"integer"}

# The content image size can be arbitrary.
content_img_size = (output_image_size1, output_image_size2)

content_image_url = os.path.join(github_content_image_folder_url,␣
 ↪content_image_name)
content_image_swd = loadImage(content_image_url, content_img_size)

image_optimized = np.swapaxes(content_image_swd, 1, 3)
```

```
image_optimized = torch.from_numpy(image_optimized)
image_optimized = image_optimized.to(torch.device(device))
image_optimized = image_optimized[:, :, 0:(image_optimized.shape[2]//8)*8, 0:
 ↪(image_optimized.shape[3]//8)*8]

image_optimized = torch.nn.parameter.Parameter(image_optimized)


display_image(image_optimized)
```

4.2 Define model architecture and intermediate feature maps used

[6]:
```
class VGG19(torch.nn.Module):

    def __init__(self):
        super(VGG19, self).__init__()

        self.block1_conv1 = torch.nn.Conv2d(3, 64, (3,3), padding=(1,1),␣
 ↪padding_mode='reflect')
        self.block1_conv2 = torch.nn.Conv2d(64, 64, (3,3), padding=(1,1),␣
 ↪padding_mode='reflect')
```

```
        self.block2_conv1 = torch.nn.Conv2d(64, 128, (3,3), padding=(1,1),␣
↪padding_mode='reflect')
        self.block2_conv2 = torch.nn.Conv2d(128, 128, (3,3), padding=(1,1),␣
↪padding_mode='reflect')

        self.block3_conv1 = torch.nn.Conv2d(128, 256, (3,3), padding=(1,1),␣
↪padding_mode='reflect')
        self.block3_conv2 = torch.nn.Conv2d(256, 256, (3,3), padding=(1,1),␣
↪padding_mode='reflect')
        self.block3_conv3 = torch.nn.Conv2d(256, 256, (3,3), padding=(1,1),␣
↪padding_mode='reflect')
        self.block3_conv4 = torch.nn.Conv2d(256, 256, (3,3), padding=(1,1),␣
↪padding_mode='reflect')

        self.block4_conv1 = torch.nn.Conv2d(256, 512, (3,3), padding=(1,1),␣
↪padding_mode='reflect')
        self.block4_conv2 = torch.nn.Conv2d(512, 512, (3,3), padding=(1,1),␣
↪padding_mode='reflect')
        self.block4_conv3 = torch.nn.Conv2d(512, 512, (3,3), padding=(1,1),␣
↪padding_mode='reflect')
        self.block4_conv4 = torch.nn.Conv2d(512, 512, (3,3), padding=(1,1),␣
↪padding_mode='reflect')

        self.relu = torch.nn.ReLU(inplace=True)
        self.downsampling = torch.nn.AvgPool2d((2,2))

    def forward(self, image):

        # RGB to BGR
        image = image[:, [2,1,0], :, :]

        # [0, 1] --> [0, 255]
        image = 255 * image

        # remove average color
        image[:,0,:,:] -= 103.939
        image[:,1,:,:] -= 116.779
        image[:,2,:,:] -= 123.68

        # block1
        block1_conv1 = self.relu(self.block1_conv1(image))
        block1_conv2 = self.relu(self.block1_conv2(block1_conv1))
        block1_pool = self.downsampling(block1_conv2)

        # block2
        block2_conv1 = self.relu(self.block2_conv1(block1_pool))
```

```
        block2_conv2 = self.relu(self.block2_conv2(block2_conv1))
        block2_pool = self.downsampling(block2_conv2)

        # block3
        block3_conv1 = self.relu(self.block3_conv1(block2_pool))
        block3_conv2 = self.relu(self.block3_conv2(block3_conv1))
        block3_conv3 = self.relu(self.block3_conv3(block3_conv2))
        block3_conv4 = self.relu(self.block3_conv4(block3_conv3))
        block3_pool = self.downsampling(block3_conv4)

        # block4
        block4_conv1 = self.relu(self.block4_conv1(block3_pool))
        block4_conv2 = self.relu(self.block4_conv2(block4_conv1))
        block4_conv3 = self.relu(self.block4_conv3(block4_conv2))
        block4_conv4 = self.relu(self.block4_conv4(block4_conv3))

        return [block1_conv1, block1_conv2, block2_conv1, block2_conv2,␣
↪block3_conv1, block3_conv2, block3_conv3, block3_conv4, block4_conv1,␣
↪block4_conv2, block4_conv3, block4_conv4]

vgg = VGG19().to(torch.device(device))
vgg.load_state_dict(torch.load("vgg19.pth", map_location=torch.device(device)))
```

[6]:
```
<All keys matched successfully>
```

[21]:
```
optimizer = torch.optim.Adam([image_optimized], lr=0.01)

def slicing_loss(image_generated, style_image_example):

    # generate VGG19 activations
    list_activations_generated = vgg(image_generated)
    list_activations_example   = vgg(style_image_example)

    # iterate over layers
    loss = 0
    for l in range(len(list_activations_example)):
        # get dimensions
        b = list_activations_example[l].shape[0]
        dim = list_activations_example[l].shape[1]
        n = list_activations_example[l].shape[2]*list_activations_example[l].
↪shape[3]
        # linearize layer activations and duplicate example activations␣
↪according to scaling factor
        activations_example = list_activations_example[l].contiguous().view(b,␣
↪dim, n).repeat(1, 1, SCALING_FACTOR*SCALING_FACTOR)
        activations_generated = list_activations_generated[l].contiguous().
↪view(b, dim, n*SCALING_FACTOR*SCALING_FACTOR)
```

```
        # sample random directions
        Ndirection = dim
        directions = torch.randn(Ndirection, dim).to(torch.device(device))
        directions = directions / torch.sqrt(torch.sum(directions**2, dim=1,␣
 ↪keepdim=True))
        # project activations over random directions
        projected_activations_example = torch.einsum('bdn,md->bmn',␣
 ↪activations_example, directions)
        projected_activations_generated = torch.einsum('bdn,md->bmn',␣
 ↪activations_generated, directions)
        # sort the projections
        sorted_activations_example = torch.sort(projected_activations_example,␣
 ↪dim=2)[0]
        sorted_activations_generated = torch.
 ↪sort(projected_activations_generated, dim=2)[0]
        # L2 over sorted lists
        loss += torch.mean(␣
 ↪(sorted_activations_example-sorted_activations_generated)**2 )
    return loss

# LBFGS closure function
def closure():
    optimizer.zero_grad()
    loss = slicing_loss(image_optimized, style_image_example)
    print(f"Loss: {loss.item()}")
    loss.backward()
    return loss
```

[22]:
```
# optimization loop
for iteration in range(128):
    tmp = image_optimized.detach().cpu().clone().numpy()
    tmp = np.swapaxes(tmp, 1, 3)
    print(iteration)
    optimizer.step(closure)

display_image(image_optimized)
```

```
0
tensor(0.2421, device='cuda:0', grad_fn=<AddBackward0>)
1
tensor(0.2196, device='cuda:0', grad_fn=<AddBackward0>)
2
tensor(0.2097, device='cuda:0', grad_fn=<AddBackward0>)
3
tensor(0.2131, device='cuda:0', grad_fn=<AddBackward0>)
4
tensor(0.2242, device='cuda:0', grad_fn=<AddBackward0>)
```

```
5
tensor(0.2098, device='cuda:0', grad_fn=<AddBackward0>)
6
tensor(0.2117, device='cuda:0', grad_fn=<AddBackward0>)
7
tensor(0.2139, device='cuda:0', grad_fn=<AddBackward0>)
8
tensor(0.2069, device='cuda:0', grad_fn=<AddBackward0>)
9
tensor(0.2142, device='cuda:0', grad_fn=<AddBackward0>)
10
tensor(0.2116, device='cuda:0', grad_fn=<AddBackward0>)
11
tensor(0.2031, device='cuda:0', grad_fn=<AddBackward0>)
12
tensor(0.1879, device='cuda:0', grad_fn=<AddBackward0>)
13
tensor(0.2037, device='cuda:0', grad_fn=<AddBackward0>)
14
tensor(0.1775, device='cuda:0', grad_fn=<AddBackward0>)
15
tensor(0.1985, device='cuda:0', grad_fn=<AddBackward0>)
16
tensor(0.1916, device='cuda:0', grad_fn=<AddBackward0>)
17
tensor(0.1994, device='cuda:0', grad_fn=<AddBackward0>)
18
tensor(0.1814, device='cuda:0', grad_fn=<AddBackward0>)
19
tensor(0.1642, device='cuda:0', grad_fn=<AddBackward0>)
20
tensor(0.1999, device='cuda:0', grad_fn=<AddBackward0>)
21
tensor(0.1768, device='cuda:0', grad_fn=<AddBackward0>)
22
tensor(0.1931, device='cuda:0', grad_fn=<AddBackward0>)
23
tensor(0.1792, device='cuda:0', grad_fn=<AddBackward0>)
24
tensor(0.1946, device='cuda:0', grad_fn=<AddBackward0>)
25
tensor(0.1719, device='cuda:0', grad_fn=<AddBackward0>)
26
tensor(0.1709, device='cuda:0', grad_fn=<AddBackward0>)
27
tensor(0.1871, device='cuda:0', grad_fn=<AddBackward0>)
28
tensor(0.1618, device='cuda:0', grad_fn=<AddBackward0>)
```

```
29
tensor(0.1635, device='cuda:0', grad_fn=<AddBackward0>)
30
tensor(0.1613, device='cuda:0', grad_fn=<AddBackward0>)
31
tensor(0.1681, device='cuda:0', grad_fn=<AddBackward0>)
32
tensor(0.1691, device='cuda:0', grad_fn=<AddBackward0>)
33
tensor(0.1700, device='cuda:0', grad_fn=<AddBackward0>)
34
tensor(0.1698, device='cuda:0', grad_fn=<AddBackward0>)
35
tensor(0.1817, device='cuda:0', grad_fn=<AddBackward0>)
36
tensor(0.1646, device='cuda:0', grad_fn=<AddBackward0>)
37
tensor(0.1808, device='cuda:0', grad_fn=<AddBackward0>)
38
tensor(0.1494, device='cuda:0', grad_fn=<AddBackward0>)
39
tensor(0.1655, device='cuda:0', grad_fn=<AddBackward0>)
40
tensor(0.1575, device='cuda:0', grad_fn=<AddBackward0>)
41
tensor(0.1783, device='cuda:0', grad_fn=<AddBackward0>)
42
tensor(0.1642, device='cuda:0', grad_fn=<AddBackward0>)
43
tensor(0.1564, device='cuda:0', grad_fn=<AddBackward0>)
44
tensor(0.1617, device='cuda:0', grad_fn=<AddBackward0>)
45
tensor(0.1500, device='cuda:0', grad_fn=<AddBackward0>)
46
tensor(0.1491, device='cuda:0', grad_fn=<AddBackward0>)
47
tensor(0.1622, device='cuda:0', grad_fn=<AddBackward0>)
48
tensor(0.1565, device='cuda:0', grad_fn=<AddBackward0>)
49
tensor(0.1517, device='cuda:0', grad_fn=<AddBackward0>)
50
tensor(0.1506, device='cuda:0', grad_fn=<AddBackward0>)
51
tensor(0.1493, device='cuda:0', grad_fn=<AddBackward0>)
52
tensor(0.1585, device='cuda:0', grad_fn=<AddBackward0>)
```

```
53
tensor(0.1641, device='cuda:0', grad_fn=<AddBackward0>)
54
tensor(0.1392, device='cuda:0', grad_fn=<AddBackward0>)
55
tensor(0.1502, device='cuda:0', grad_fn=<AddBackward0>)
56
tensor(0.1496, device='cuda:0', grad_fn=<AddBackward0>)
57
tensor(0.1391, device='cuda:0', grad_fn=<AddBackward0>)
58
tensor(0.1546, device='cuda:0', grad_fn=<AddBackward0>)
59
tensor(0.1615, device='cuda:0', grad_fn=<AddBackward0>)
60
tensor(0.1439, device='cuda:0', grad_fn=<AddBackward0>)
61
tensor(0.1460, device='cuda:0', grad_fn=<AddBackward0>)
62
tensor(0.1439, device='cuda:0', grad_fn=<AddBackward0>)
63
tensor(0.1345, device='cuda:0', grad_fn=<AddBackward0>)
64
tensor(0.1342, device='cuda:0', grad_fn=<AddBackward0>)
65
tensor(0.1315, device='cuda:0', grad_fn=<AddBackward0>)
66
tensor(0.1344, device='cuda:0', grad_fn=<AddBackward0>)
67
tensor(0.1435, device='cuda:0', grad_fn=<AddBackward0>)
68
tensor(0.1327, device='cuda:0', grad_fn=<AddBackward0>)
69
tensor(0.1188, device='cuda:0', grad_fn=<AddBackward0>)
70
tensor(0.1250, device='cuda:0', grad_fn=<AddBackward0>)
71
tensor(0.1256, device='cuda:0', grad_fn=<AddBackward0>)
72
tensor(0.1360, device='cuda:0', grad_fn=<AddBackward0>)
73
tensor(0.1339, device='cuda:0', grad_fn=<AddBackward0>)
74
tensor(0.1387, device='cuda:0', grad_fn=<AddBackward0>)
75
tensor(0.1356, device='cuda:0', grad_fn=<AddBackward0>)
76
tensor(0.1280, device='cuda:0', grad_fn=<AddBackward0>)
```

```
77
tensor(0.1304, device='cuda:0', grad_fn=<AddBackward0>)
78
tensor(0.1216, device='cuda:0', grad_fn=<AddBackward0>)
79
tensor(0.1321, device='cuda:0', grad_fn=<AddBackward0>)
80
tensor(0.1306, device='cuda:0', grad_fn=<AddBackward0>)
81
tensor(0.1332, device='cuda:0', grad_fn=<AddBackward0>)
82
tensor(0.1401, device='cuda:0', grad_fn=<AddBackward0>)
83
tensor(0.1295, device='cuda:0', grad_fn=<AddBackward0>)
84
tensor(0.1287, device='cuda:0', grad_fn=<AddBackward0>)
85
tensor(0.1315, device='cuda:0', grad_fn=<AddBackward0>)
86
tensor(0.1272, device='cuda:0', grad_fn=<AddBackward0>)
87
tensor(0.1165, device='cuda:0', grad_fn=<AddBackward0>)
88
tensor(0.1290, device='cuda:0', grad_fn=<AddBackward0>)
89
tensor(0.1259, device='cuda:0', grad_fn=<AddBackward0>)
90
tensor(0.1292, device='cuda:0', grad_fn=<AddBackward0>)
91
tensor(0.1253, device='cuda:0', grad_fn=<AddBackward0>)
92
tensor(0.1238, device='cuda:0', grad_fn=<AddBackward0>)
93
tensor(0.1203, device='cuda:0', grad_fn=<AddBackward0>)
94
tensor(0.1177, device='cuda:0', grad_fn=<AddBackward0>)
95
tensor(0.1198, device='cuda:0', grad_fn=<AddBackward0>)
96
tensor(0.1176, device='cuda:0', grad_fn=<AddBackward0>)
97
tensor(0.1216, device='cuda:0', grad_fn=<AddBackward0>)
98
tensor(0.1183, device='cuda:0', grad_fn=<AddBackward0>)
99
tensor(0.1213, device='cuda:0', grad_fn=<AddBackward0>)
100
tensor(0.1224, device='cuda:0', grad_fn=<AddBackward0>)
```

```
101
tensor(0.1135, device='cuda:0', grad_fn=<AddBackward0>)
102
tensor(0.1118, device='cuda:0', grad_fn=<AddBackward0>)
103
tensor(0.1116, device='cuda:0', grad_fn=<AddBackward0>)
104
tensor(0.1185, device='cuda:0', grad_fn=<AddBackward0>)
105
tensor(0.1217, device='cuda:0', grad_fn=<AddBackward0>)
106
tensor(0.1105, device='cuda:0', grad_fn=<AddBackward0>)
107
tensor(0.1103, device='cuda:0', grad_fn=<AddBackward0>)
108
tensor(0.1135, device='cuda:0', grad_fn=<AddBackward0>)
109
tensor(0.1043, device='cuda:0', grad_fn=<AddBackward0>)
110
tensor(0.1120, device='cuda:0', grad_fn=<AddBackward0>)
111
tensor(0.1009, device='cuda:0', grad_fn=<AddBackward0>)
112
tensor(0.1041, device='cuda:0', grad_fn=<AddBackward0>)
113
tensor(0.1111, device='cuda:0', grad_fn=<AddBackward0>)
114
tensor(0.1168, device='cuda:0', grad_fn=<AddBackward0>)
115
tensor(0.1054, device='cuda:0', grad_fn=<AddBackward0>)
116
tensor(0.1023, device='cuda:0', grad_fn=<AddBackward0>)
117
tensor(0.1037, device='cuda:0', grad_fn=<AddBackward0>)
118
tensor(0.1083, device='cuda:0', grad_fn=<AddBackward0>)
119
tensor(0.1099, device='cuda:0', grad_fn=<AddBackward0>)
120
tensor(0.1018, device='cuda:0', grad_fn=<AddBackward0>)
121
tensor(0.1065, device='cuda:0', grad_fn=<AddBackward0>)
122
tensor(0.1100, device='cuda:0', grad_fn=<AddBackward0>)
123
tensor(0.1173, device='cuda:0', grad_fn=<AddBackward0>)
124
tensor(0.1074, device='cuda:0', grad_fn=<AddBackward0>)
```

```
125
tensor(0.0993, device='cuda:0', grad_fn=<AddBackward0>)
126
tensor(0.1095, device='cuda:0', grad_fn=<AddBackward0>)
127
tensor(0.1001, device='cuda:0', grad_fn=<AddBackward0>)
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with
RGB data ([0..1] for floats or [0..255] for integers).
```

5 Comparing image quality using SSIM and PSNR

5.1 For the Gram Matrix method

[12]:
```
from skimage.metrics import structural_similarity as ssim
from skimage.metrics import peak_signal_noise_ratio as psnr

generated_image_arr = np.array(image).squeeze()
content_image_arr = np.array(content_image).squeeze()

ssim_index = ssim(
    generated_image_arr,
    content_image_arr,
    channel_axis=2, # dim 2 represents channel
```

```
)

psnr_value = psnr(content_image_arr, generated_image_arr)


print(f"GRAM MATRIX - SSIM: {ssim_index}")
print(f"GRAM MATRIX - PSNR: {psnr_value} dB")
```

```
GRAM MATRIX - SSIM: 0.2752472758293152
GRAM MATRIX - PSNR: 13.348374005383825 dB
```

5.2 For the SWD method

[24]:
```
from skimage.metrics import structural_similarity as ssim
from skimage.metrics import peak_signal_noise_ratio as psnr

generated_image_arr = np.array(image_optimized.cpu().detach()).squeeze(0)
content_image_arr = np.array(content_image_swd).squeeze(0)
content_image_arr = np.swapaxes(content_image_arr, 0, 2)

ssim_index = ssim(
    generated_image_arr,
    content_image_arr,
    channel_axis=0, # dim 0 represents channel
)

psnr_value = psnr(content_image_arr, generated_image_arr)


print(f"SWD - SSIM: {ssim_index}")
print(f"SWD - PSNR: {psnr_value} dB")
```

```
SWD - SSIM: 0.5216733813285828
SWD - PSNR: 16.135686988242647 dB
```

[ ]:
```
# @title Save and download generated image...  { display-mode: "form" }

print('SAVING AND DOWNLOADING GENERATED IMAGE...')
file_name = f'stylized-image-{step}.png'
tensor_to_image(image).save(file_name)

try:
  from google.colab import files
except ImportError:
   pass
else:
  files.download(file_name)
```

```
print('SAVING AND DOWNLOADING GENERATED IMAGE DONE')
```

```
SAVING AND DOWNLOADING GENERATED IMAGE…
<IPython.core.display.Javascript object>
<IPython.core.display.Javascript object>
SAVING AND DOWNLOADING GENERATED IMAGE DONE
```