Style Transfer: Description of GAN

Francia Garcia Romero

A01769680

```
import matplotlib.pyplot as plt
from matplotlib.pyplot import imshow
from PIL import Image
import numpy as np
import tensorflow as tf
import pprint
```

The goal of this notebook is to take an image with a specific style (style_image) and another image (content_image) that you want to modify in the style of style_image. For that we will take the output of the style_image from different middle layers of the vgg19 network and the output of the content_image after the last conv layer of the vgg19 network. After that we will compute two J cost functions, one for the content ouput and one for the style ouput, and add them using two coefficients. Finaly we will iterate and and modifiy the content image (more precisely a copy of it with a little noise) little by little until we obtain a satisfying result.

This notebook is mostly taken from the great deep learning specialization course of Andrew Ng (https://www.coursera.org/specializations/deep-learning).

Here, the instance of the VGG19 model is called. The top layers are excluded, which means that only the bottom layers are being imported to use them for feature extraction. The layers of the model are trained with the ImageNet dataset.

In the last line of the following chunk of code, the layers are set to be non-trainable.

```
# We define the image size as we wish and import the VGG19 network with the imagenet weights. We do not need to train the network for this ap
img_size1, img_size2 = 800,450
vgg = tf.keras.applications.VGG19(include_top=False,
                                  input_shape=(img_size2, img_size1, 3),
                                  weights='imagenet')

vgg.trainable = False
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg19/vgg19_weights_tf_dim_ordering_tf_kernels_notop.
80134624/80134624 [==============================] - 0s 0us/step
```

Using the *summary()* method, we can display all the layers that we are using as part of the VGG19 model instance. We have convolutional and pooling layers.

```
vgg.summary()
```

```
Model: "vgg19"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 450, 800, 3)]     0

 block1_conv1 (Conv2D)       (None, 450, 800, 64)      1792

 block1_conv2 (Conv2D)       (None, 450, 800, 64)      36928

 block1_pool (MaxPooling2D)  (None, 225, 400, 64)      0

 block2_conv1 (Conv2D)       (None, 225, 400, 128)     73856

 block2_conv2 (Conv2D)       (None, 225, 400, 128)     147584

 block2_pool (MaxPooling2D)  (None, 112, 200, 128)     0

 block3_conv1 (Conv2D)       (None, 112, 200, 256)     295168

 block3_conv2 (Conv2D)       (None, 112, 200, 256)     590080

 block3_conv3 (Conv2D)       (None, 112, 200, 256)     590080

 block3_conv4 (Conv2D)       (None, 112, 200, 256)     590080
```

```
 block3_pool (MaxPooling2D)  (None, 56, 100, 256)      0

 block4_conv1 (Conv2D)       (None, 56, 100, 512)      1180160

 block4_conv2 (Conv2D)       (None, 56, 100, 512)      2359808

 block4_conv3 (Conv2D)       (None, 56, 100, 512)      2359808

 block4_conv4 (Conv2D)       (None, 56, 100, 512)      2359808

 block4_pool (MaxPooling2D)  (None, 28, 50, 512)       0

 block5_conv1 (Conv2D)       (None, 28, 50, 512)       2359808

 block5_conv2 (Conv2D)       (None, 28, 50, 512)       2359808

 block5_conv3 (Conv2D)       (None, 28, 50, 512)       2359808

 block5_conv4 (Conv2D)       (None, 28, 50, 512)       2359808

 block5_pool (MaxPooling2D)  (None, 14, 25, 512)       0

=================================================================
Total params: 20024384 (76.39 MB)
Trainable params: 0 (0.00 Byte)
Non-trainable params: 20024384 (76.39 MB)
_________________________________________________________________
```

```
#the list of layers in the network
for layer in vgg.layers:
    print(layer.name)
```

```
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
```

A dictionary with the layers that will be used for transfering style is created. Only the convolutional layers are included in this set.

```
# The list of layers we take for the style_image outputs. Each weight has been set to 0.2 for now.
STYLE_LAYERS = [
    ('block3_conv1', 0.125),
    ('block3_conv2', 0.125),
    ('block3_conv3', 0.125),
    ('block3_conv4', 0.125),
    ('block4_conv1', 0.125),
    ('block4_conv2', 0.125),
    ('block4_conv3', 0.125),
    ('block4_conv4', 0.125),]

content_layer = [('block5_conv4', 1)]

# layer_names = STYLE_LAYERS + content_layer
```

For each layer selected, a VGG model returns a list of the intermediate values

```
# This function gives us the outputs of the selected style and content layers
def get_layer_outputs(vgg, layer_names):
```

```
    """ Creates a vgg model that returns a list of intermediate output values."""
    outputs = [vgg.get_layer(layer[0]).output for layer in layer_names]
    print(outputs) # ------------------------------
    model = tf.keras.Model([vgg.input], outputs)
    return model
```

```
vgg_model_outputs = get_layer_outputs(vgg, STYLE_LAYERS + content_layer)
```

We load the content and style images and then we preprocess them for the vgg network. At the same time we also generate the image that will be modified to match the style of style_image.

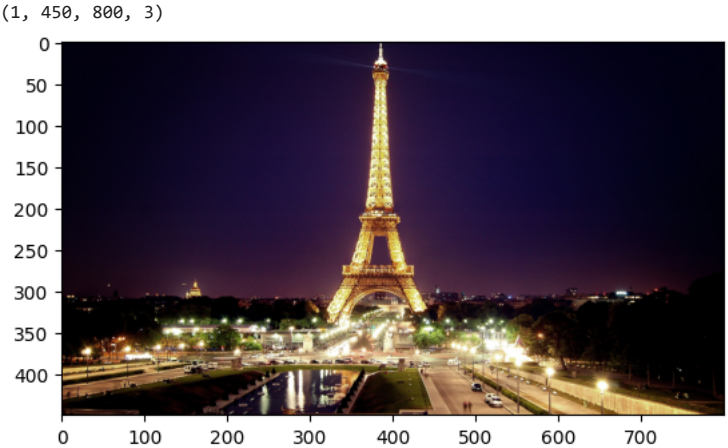
```
content_image_path = "Tour_eiffel_de_nuit.jpg"
content_image = Image.open(content_image_path)
content_image
```

```
style_image_path = "Van_Gogh_-_Starry_Night_-_Google_Art_Project.jpg"
style_image = Image.open(style_image_path)
style_image
```

Both are resized as part of the preprocessing needed to train the model.

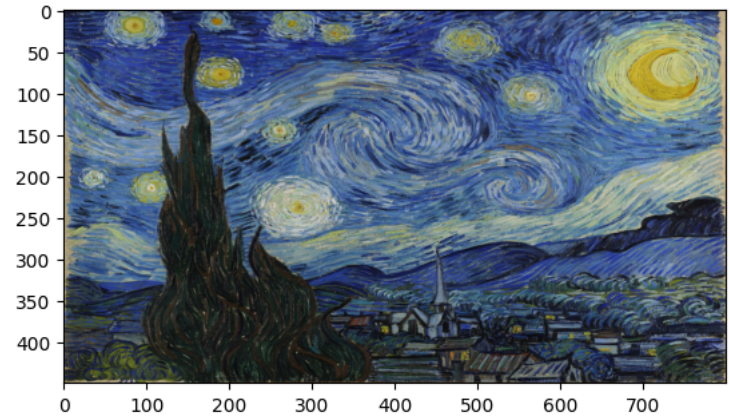
```
content_image = np.array(Image.open(content_image_path).resize((img_size1, img_size2)))
content_image = tf.constant(np.reshape(content_image, ((1,) + content_image.shape)))
print(content_image.shape)
imshow(content_image[0])
plt.show()
```

```
(1, 450, 800, 3)
```



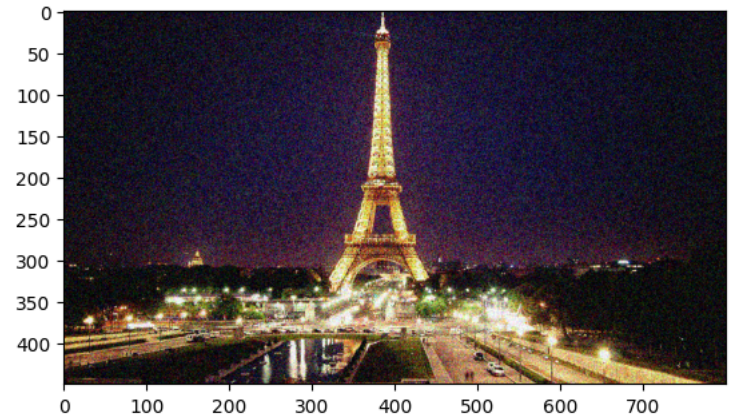
```
style_image = np.array(Image.open(style_image_path).resize((img_size1, img_size2)))
style_image = tf.constant(np.reshape(style_image, ((1,) + style_image.shape)))
print(style_image.shape)
imshow(style_image[0])
plt.show()
```

```
(1, 450, 800, 3)
```



```
#The image is generated by adding a bit of noise to the content_image
generated_image = tf.Variable(tf.image.convert_image_dtype(content_image, tf.float32))
noise = tf.random.uniform(tf.shape(generated_image), -0.25, 0.25)
generated_image = tf.add(generated_image, noise)
generated_image = tf.clip_by_value(generated_image, clip_value_min=0.0, clip_value_max=1.0)

print(generated_image.shape)
imshow(generated_image.numpy()[0])
plt.show()
```

```
(1, 450, 800, 3)
```



Both **preprocessed_content** and **preprocessed_style** are TensorFlow variables that can be updated during training. In this code, it's being used to create TensorFlow variables to hold the preprocessed versions of the content and style images.

The **tf.image.convert_image_dtype** function is being applied to content_image and style_image to convert them to the tf.float32 data type.

The **vgg_model_outputs** is a function that takes an image as input and passes it through a pre-trained VGG model. The VGG model extracts feature representations at different layers, which are useful for style transfer tasks (such as this).

a_C is assigned the feature representations extracted from the preprocessed_content image using the VGG model. These feature representations capture the content of the image at different layers of the network. **a_S** is assigned the feature representations extracted from the preprocessed_style image using the VGG model. These feature representations capture the style of the image at different layers of the network.

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

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

clip_0_1(image)

- Purpose: This function is used to ensure that all pixel values in a given image tensor fall within the range [0, 1]. It clips (truncates) values that are less than 0 to 0 and values greater than 1 to 1.
- Input: image is a TensorFlow tensor representing an image. It can have pixel values outside the desired range.
- Output: The function returns a tensor with all pixel values clipped to the range [0, 1].

tensor_to_image(tensor)

- Purpose: This function converts a tensor representation of an image into a PIL (Python Imaging Library) image. It scales the pixel values from the [0, 1] range to the [0, 255] range to match the expected pixel value range for images in PIL.
- Input: tensor is a TensorFlow tensor representing an image with pixel values in the [0, 1] range.
- Output: The function returns a PIL image with pixel values in the [0, 255] range. It multiplies the input tensor by 255 to scale the values from [0, 1] to [0, 255]. Converts the scaled tensor to a NumPy array with data type uint8 (8-bit unsigned integer), which is the common data type for images. If the tensor has more than 3 dimensions and the first dimension is equal to 1 (commonly used in batch processing), it squeezes the tensor to remove the batch dimension. Finally, it converts the NumPy array into a PIL image using the *Image.fromarray* function.

```
def clip_0_1(image):
    """
    Truncate all the pixels in the tensor to be between 0 and 1

    Arguments:
    image -- Tensor

    Returns:
    Tensor
    """
    return tf.clip_by_value(image, clip_value_min=0.0, clip_value_max=1.0)

def tensor_to_image(tensor):
    """
    Converts the given tensor into a PIL image

    Arguments:
    tensor -- Tensor

    Returns:
    Image: A 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 Image.fromarray(tensor)
```

The **gram_matrix** function computes the Gram matrix of a given matrix A. The Gram matrix encodes information about the style or texture of an image by summarizing how different features within the image are correlated.

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

    Returns:
    GA -- Gram matrix of A, of shape (n_C, n_C)
    """

    GA = tf.linalg.matmul(A,A, transpose_b=True)

    return GA
```

The **compute_content_cost** function calculates how similar the content of a generated image is to the content of a reference image. It does it by comparing the activations of a specific layer in both images and measuring the squared differences. The resulting content cost represents how well the generated image captures the structural content of the reference image. This cost is used to ensure the content remains consistent while changing the style of the image.

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

    Arguments:
    a_C -- tensor of dimension (1, n_H, n_W, n_C), hidden layer activations representing content of the image C
    a_G -- tensor of dimension (1, n_H, n_W, n_C), hidden layer activations representing content of the image G

    Returns:
    J_content -- scalar that you compute using equation 1 above.
    """
    #[-1] because the last layer is the content layer
    a_C = content_output[-1]
    a_G = generated_output[-1]

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


    # Reshape 'a_C' and 'a_G'
    a_C_unrolled = tf.reshape(a_C, shape=[m, -1, n_C])
    a_G_unrolled = tf.reshape(a_G, shape=[m, -1, n_C])

    # compute the cost with tensorflow
    J_content =   tf.reduce_sum(tf.square(tf.subtract(a_C, a_G)))*(1/(4*n_H*n_W*n_C))

    return J_content
```

The **compute_layer_style_cost** function calculates how similar the style of a generated image is to the style of a reference image using a specific layer's activations. It does this by comparing the Gram matrices of the style activations from both images. The style cost measures the difference between these Gram matrices, and it's used in neural style transfer to ensure that the generated image matches the artistic style of the reference image for the chosen layer.

```
def compute_layer_style_cost(a_S, a_G):
    """
    Arguments:
    a_S -- tensor of dimension (1, n_H, n_W, n_C), hidden layer activations representing style of the image S
    a_G -- tensor of dimension (1, n_H, n_W, n_C), hidden layer activations representing style of the image G

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

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

    # Reshape the tensors from (1, n_H, n_W, n_C) to (n_C, n_H * n_W)
    a_S = tf.reshape(tf.transpose(a_S, perm=[0,3,1,2]), shape=[n_C, n_H*n_W])
    a_G = tf.reshape(tf.transpose(a_G, perm=[0,3,1,2]), shape=[n_C, n_H*n_W])

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

    # Computing the loss
    J_style_layer =  tf.reduce_sum(tf.square(GS - GG))*(1/(4*(n_C*n_H*n_W)**2))

    return J_style_layer
```

The **compute_style_cost** function calculates how well the style of a generated image matches the style of a reference image. It does so by considering multiple chosen layers and their associated weights. The overall style cost represents the total style difference between the generated and reference images across these layers. This cost guides the neural style transfer process to generate an image with a similar artistic style to the reference image while preserving its content.

```
def compute_style_cost(style_image_output, generated_image_output, STYLE_LAYERS=STYLE_LAYERS):
    """
    Computes the overall style cost from several chosen layers

    Arguments:
    style_image_output -- our tensorflow model
    generated_image_output --
    STYLE_LAYERS -- A python list containing:
```

```
                        - the names of the layers we would like to extract style from
                        - a coefficient for each of them

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

    # initialize the overall style cost
    J_style = 0

    # Set a_S to be the hidden layer activation from the layer we have selected.
    # [:-1] because last item is the content layer
    a_S = style_image_output[:-1]

    # Set a_G to be the output of the choosen hidden layers.
    a_G = generated_image_output[:-1]
    for i, weight in zip(range(len(a_S)), STYLE_LAYERS):
        # Compute style_cost for the current layer
        J_style_layer = compute_layer_style_cost(a_S[i], a_G[i])

        # Add weight * J_style_layer of this layer to overall style cost
        J_style += weight[1] * J_style_layer

    return J_style
```

The **total_cost function** calculates the overall cost for neural style transfer by combining two components: the content cost and the style cost. It uses hyperparameters alpha and beta to control the importance of each component. Adjusting these hyperparameters allows fine-tuning the trade-off between preserving content and matching style in the final generated image. The function returns the total cost, which guides the optimization process to create an image that balances content and style according to the specified hyperparameters.

```
@tf.function()
def total_cost(J_content, J_style, alpha = 10, beta = 40):
    """
    Computes the total cost function

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

    Returns:
    J -- total cost as defined by the formula above.
    """

    J = alpha*J_content + beta*J_style


    return J
```

optimizer = tf.keras.optimizers.Adam(learning_rate=0.01): This line defines an Adam optimizer with a learning rate of 0.01. The optimizer is used to adjust the generated image during training to minimize the total cost.

@tf.function(): This decorator converts the train_step function into a TensorFlow graph, which can improve performance and enable GPU acceleration.

train_step(generated_image): This function takes a generated image as input and performs the following steps:

Inside the function, the content cost (J_content) and style cost (J_style) are computed by comparing the activations of the generated image (a_G) to the reference content (a_C) and style (a_S) activations. These activations are obtained from a VGG model.

The total cost (J) is computed as a weighted combination of the content and style costs using hyperparameters alpha and beta.

The gradients of the total cost with respect to the generated image are calculated using tf.GradientTape. These gradients indicate how the generated image should be adjusted to minimize the cost.

The optimizer applies the gradients to the generated image to update it. This is done with the line optimizer.apply_gradients([(grad, generated_image)]), where grad is the computed gradient.

The generated image is then clipped to ensure its pixel values are within the valid range of [0, 1] using generated_image.assign(clip_0_1(generated_image)).

Finally, the function returns the values of the total cost (J), style cost (J_style), and content cost (J_content) to monitor the progress of the training.

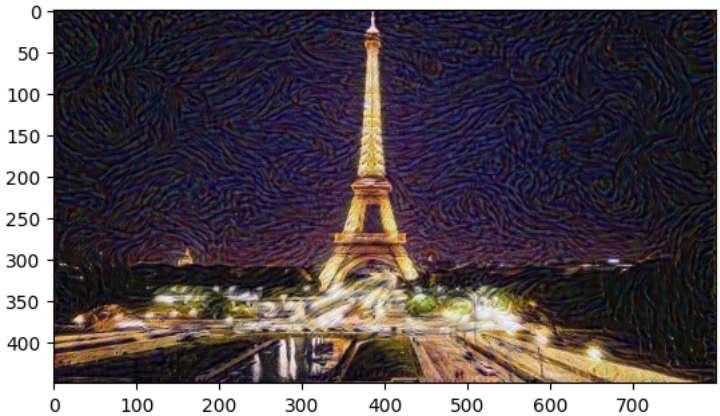
In summary, this code defines a training step that iteratively updates a generated image to minimize the combined content and style costs using the Adam optimizer. The generated image gradually transforms to match the content and style of the reference images.

```
optimizer = tf.keras.optimizers.Adam(learning_rate=0.01)

@tf.function()
def train_step(generated_image):
    with tf.GradientTape() as tape:
        # In this function you must use the precomputed encoded images a_S and a_C

        # Compute a_G as the vgg_model_outputs for the current generated image
        a_G = vgg_model_outputs(generated_image)

        # Compute the style cost
        J_style = compute_style_cost(a_S, a_G)

        # Compute the content cost
        J_content = compute_content_cost(a_C, a_G)
        # Compute the total cost
        J = total_cost(J_content, J_style, alpha = 10, beta = 40)

    grad = tape.gradient(J, generated_image)

    optimizer.apply_gradients([(grad, generated_image)])
    generated_image.assign(clip_0_1(generated_image))

    return J, J_style, J_content


generated_image = tf.Variable(generated_image)
```

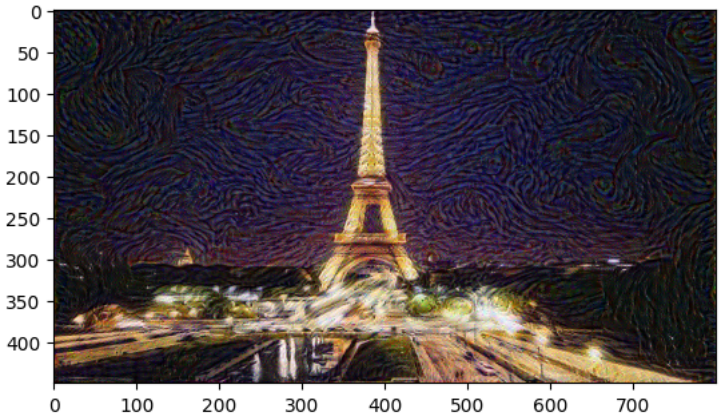
Finalle, we have te the training of the model (using all the functions defined)

```
epochs = 2000
image_list = []
for i in range(epochs):
    J, J_style, J_content =train_step(generated_image)
    if i % 250 == 0:
        print(f"Epoch {i}; J {J}; J_style {J_style}: J_content {J_content}")
    if i % 250 == 0:
        image = tensor_to_image(generated_image)
        imshow(image)
        #image.save(f"/kaggle/working/image_{str(0)*(len(str(epochs)) - len(str(i))) + str(i)}.jpg")
        plt.show()
    if i % 25 == 0:
        image = tensor_to_image(generated_image)
        image_list.append(image)
```

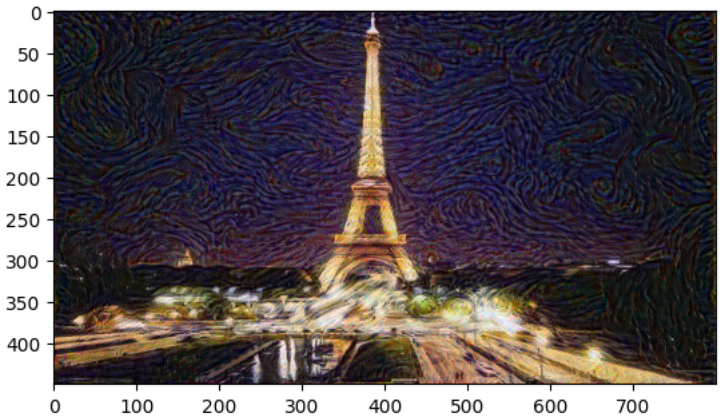
```
Epoch 0; J 16.528614044189453; J_style 0.40917861461639404: J_content 0.01614678092300892
```



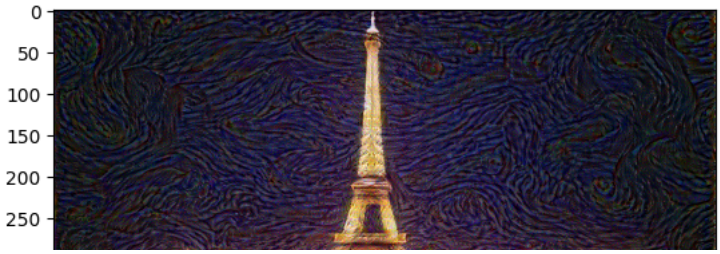
```
Epoch 250; J 8.718348503112793; J_style 0.21379989385604858: J_content 0.016635272651910782
```



```
Epoch 500; J 5.929300308227539; J_style 0.14407019317150116: J_content 0.016649264842271805
```



```
Epoch 750; J 4.5153374671936035; J_style 0.10872547328472137: J_content 0.0166318342089653
```



```
# A little code to make a gif of the generation process
frame_one = image_list[0]
frame_one.save("/kaggle/working/my_gif_eiffel_tower.gif", format="GIF", append_images=image_list,
               save_all=True, duration=100, loop=0)
```

```
# Show the 3 images in a row
fig = plt.figure(figsize=(16, 4))
ax = fig.add_subplot(1, 3, 1)
imshow(content_image[0])
ax.title.set_text('Content image')
ax = fig.add_subplot(1, 3, 2)
imshow(style_image[0])
ax.title.set_text('Style image')
ax = fig.add_subplot(1, 3, 3)
imshow(generated_image[0])
ax.title.set_text('Generated image')
plt.show()
```