Neural-Style-Transfer

Report By - Garv

En. No. – 22112039

❖ ***Introduction :-***

- Neural style transfer (NST) is a technique in the field of artificial intelligence and computer vision that allows for the artistic transformation of images. It merges the content of one image (often a photograph) with the style of another image (typically an artwork) to create a new image that combines the content details of the original image with the visual style of the reference artwork.\

❖ ***Objective :-***

- The fundamental aim of this project is to develop an advanced neural style transfer model capable of achieving a harmonious synthesis between the content of a given photograph and the stylistic characteristics derived from a chosen artwork. This synthesis entails preserving the underlying structure, spatial relationships, and semantic content of the original photograph

while imbuing it with the expressive brushstrokes, color palette, and visual motifs that define the selected artistic style.

➢ ***Key Objectives*** :-

1.) **Content Preservation:** The model should accurately retain the essential elements and details of the input photograph, ensuring that the semantic content and spatial arrangement remain intact after style transfer.

2.) **Stylistic Transformation:** Leveraging convolutional neural networks (CNNs) and optimization algorithms, the model must extract the intricate stylistic features from the reference artwork and effectively apply them to the content image. This involves capturing the texture, brushstrokes, color distribution, and overall artistic essence characteristic of the chosen style.

3.) **Perceptual Loss Function:** To achieve realistic and aesthetically pleasing results, the model will utilize perceptual loss functions that measure the perceptual difference between the generated image and the target style, rather than relying solely on pixel-wise differences. This approach ensures that the

transferred style is perceptually faithful to the original artwork.

4.) **Balanced Integration:** The ultimate goal is to strike a delicate balance between preserving the content of the photograph and integrating the desired artistic style. This balance is crucial in ensuring that the generated images are not only visually appealing but also artistically coherent, maintaining fidelity to both the content and style domains.

5.) **User Accessibility:** Beyond technical excellence, the model should be designed with user accessibility in mind, enabling individuals with varying levels of technical expertise to seamlessly apply neural style transfer to their own photographs. This democratization of artistic expression empowers users to explore and personalize artistic styles in their visual creations.

❖ ***Methodology :-***

- Neural Style Transfer: Creating Art with Deep Learning using tf.keras and eager execution.

  - The principle of neural style transfer is to define two distance functions, one that describes how different the content of two images are, Lcontent, and one that describes the difference between the two images in terms of their style, Lstyle. Then, given three images, a desired style image, a desired content image, and the input image (initialized with the content image), we try to transform the input image to minimize the content distance with the content image and its style distance with the style image.

    In summary, we'll take the base input image, a content image that we want to match, and the style image that we want to match. We'll transform the base input image by minimizing the content and style distances (losses) with backpropagation, creating an image that matches the content of the content image and the style of the style image.

❖ ***We will follow the general steps to perform style transfer:***

- ➢ Visualize data
- ➢ Basic Preprocessing/preparing our data
- ➢ Set up loss functions
- ➢ Create model
- ➢ Optimize for loss function

❖ ***Implementation -:***

We'll begin by enabling eager execution. Eager execution allows us to work through this technique in the clearest and most readable way.

```
tf.enable_eager_execution()

print("Eager execution: {}".format(tf.executing_eagerly()))



Here are the content and style images we will use:

plt.figure(figsize=(10,10))



content = load_img(content_path).astype('uint8')

style = load_img(style_path)
```

```
plt.subplot(1, 2, 1)

imshow(content, 'Content Image')



plt.subplot(1, 2, 2)

imshow(style, 'Style Image')

plt.show()
```

❖ ***Define content and style representations :-***

- In order to get both the content and style representations of our image, we will look at some intermediate layers within our model. Intermediate layers represent feature maps that become increasingly higher ordered as you go deeper. In this case, we are using the network architecture VGG19, a pretrained image classification network. These intermediate layers are necessary to define the representation of content and style from our images. For an input image, we will try to match the corresponding style and content target representations at these intermediate layers.

➢Why intermediate layers ?

- You may be wondering why these intermediate outputs within our pretrained image classification network allow us to define style and content representations. At a high level, this phenomenon can be explained by the fact that in order for a network to perform image classification (which our network has been trained to do), it must understand the image. This involves taking the raw image as input pixels and building an internal representation through transformations that turn the raw image pixels into a complex understanding of the features present within the image.

- Specifically we'll pull out these intermediate layers from our network.

```
# Content layer where will pull our feature maps

content_layers = ['block5_conv2']



# Style layer we are interested in

style_layers = ['block1_conv1',

                'block2_conv1',

                'block3_conv1',

                'block4_conv1',
```

```
   'block5_conv1'

               ]



num_content_layers = len(content_layers)

num_style_layers = len(style_layers)
```

❖ ***Model :-***

In this case, we load VGG19, and feed in our input tensor to the model. This will allow us to extract the feature maps (and subsequently the content and style representations) of the content, style, and generated images.

We use VGG19, as suggested in the paper. In addition, since VGG19 is a relatively simple model (compared with ResNet, Inception, etc) the feature maps actually work better for style transfer.

```
def get_model():

  """ Creates our model with access to intermediate layers.



  This function will load the VGG19 model and access the
intermediate layers.

  These layers will then be used to create a new model that will
take input image

  and return the outputs from these intermediate layers from the VGG
model.
```

```
  Returns:

    returns a keras model that takes image inputs and outputs the
style and

      content intermediate layers.

  """


  # Load our model. We load pretrained VGG, trained on imagenet data
(weights='imagenet')

  vgg = tf.keras.applications.vgg19.VGG19(include_top=False,
weights='imagenet')

  vgg.trainable = False

  # Get output layers corresponding to style and content layers

  style_outputs = [vgg.get_layer(name).output for name in
style_layers]

content_outputs = [vgg.get_layer(name).output for name in
content_layers]

  model_outputs = style_outputs + content_outputs

  # Build model

  return models.Model(vgg.input, model_outputs)
```

❖ ***Define and create our loss functions (content and style distances) :-***

Content Loss:

Our content loss definition is actually quite simple. We'll pass the network both the desired content image and our base input image. This will return the intermediate layer outputs (from the layers defined above) from our model. Then we simply take the euclidean distance between the two intermediate representations of those images. More formally, content loss is a function that describes the distance of content from our input image x and our content image, p .

$$L^l_{content}(p, x) = \sum_{i,j}(F^l_{ij}(x) - P^l_{ij}(p))^2$$

We perform backpropagation in the usual way such that we minimize this content loss. We thus change the initial image until it generates a similar response in a certain layer (defined in content_layer) as the original content image.

```
def get_content_loss(base_content, target):

  return tf.reduce_mean(tf.square(base_content - target))
```

Style Loss:

Computing style loss is a bit more involved, but follows the same principle, this time feeding our network the base input image and the style image. However, instead of comparing the raw intermediate outputs of the base input image and the style image, we instead compare the Gram matrices of the two outputs.

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} (G_{ij}^l - A_{ij}^l)^2$$

$$L_{content}^l(p, x) = \sum_{i,j} (F_{ij}^l(x) - P_{ij}^l(p))^2$$

This is implemented simply:

```
def gram_matrix(input_tensor):

  # We make the image channels first

  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)



def get_style_loss(base_style, gram_target):

  """Expects two images of dimension h, w, c"""

  # height, width, num filters of each layer

  height, width, channels = base_style.get_shape().as_list()

  gram_style = gram_matrix(base_style)



  return tf.reduce_mean(tf.square(gram_style - gram_target))
```

❖ ***Run Gradient Descent -:***

If you aren't familiar with gradient descent/backpropagation or need a refresher, you should definitely check out this resource. In this case, we use the Adam optimizer in order to minimize our loss. We iteratively update our output image such that it minimizes our loss: we don't update the weights associated with our network, but instead we train our input image to minimize loss.

We'll define a little helper function that will load our content and style image, feed them forward through our network, which will then output the content and style feature representations from our model.

```
def get_feature_representations(model, content_path, style_path):
```

```
  """Helper function to compute our content and style feature
representations.



  This function will simply load and preprocess both the content and
style

  images from their path. Then it will feed them through the network
to obtain

  the outputs of the intermediate layers.



  Arguments:

    model: The model that we are using.

    content_path: The path to the content image.

    style_path: The path to the style image



  Returns:

    returns the style features and the content features.

  """

  # Load our images in

  content_image = load_and_process_img(content_path)

  style_image = load_and_process_img(style_path)



  # batch compute content and style features

  stack_images = np.concatenate([style_image, content_image],
axis=0)
```

```
  model_outputs = model(stack_images)



  # Get the style and content feature representations from our model

  style_features = [style_layer[0] for style_layer in
model_outputs[:num_style_layers]]

  content_features = [content_layer[1] for content_layer in
model_outputs[num_style_layers:]]

  return style_features, content_features
```

Here we use tf.GradientTape to compute the gradient. It allows us to take advantage of the automatic differentiation available by tracing operations for computing the gradient later. It records the operations during the forward pass and then is able to compute the gradient of our loss function with respect to our input image for the backwards pass.

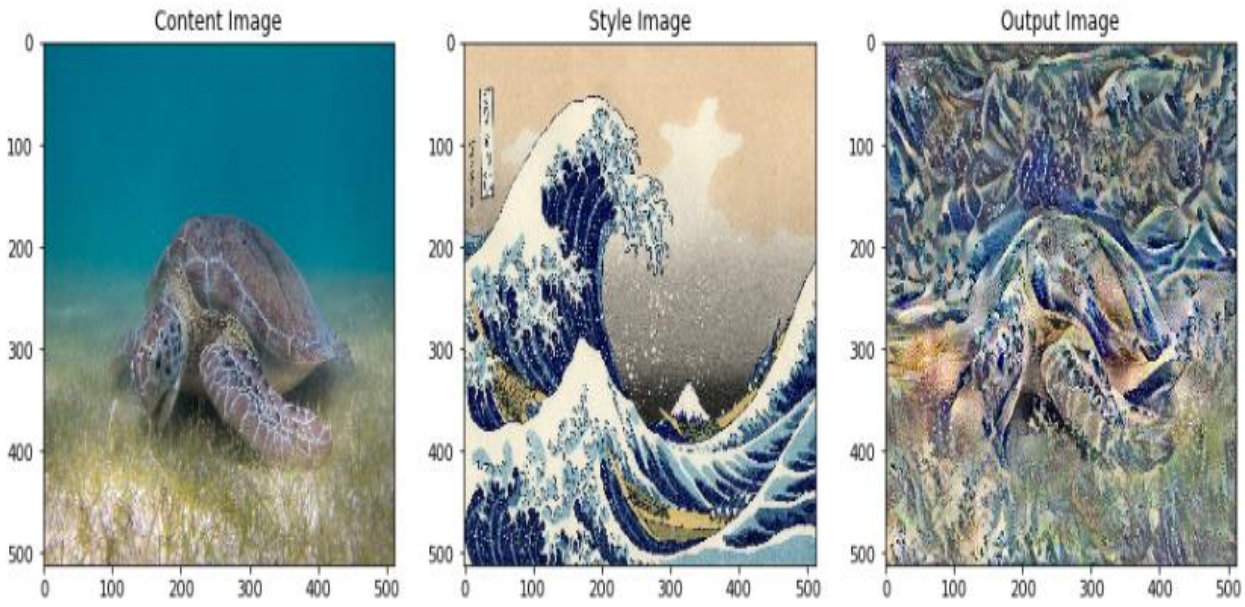
❖ *And at last apply and run the style transfer process*

```
def run_style_transfer(content_path,

                       style_path,

                       num_iterations=1000,

                       content_weight=1e3,

                       style_weight = 1e-2):
```

❖ ***Now to run it on an image :-***

```
❖ best, best_loss = run_style_transfer(content_path,
❖                                      style_path,
❖                                      verbose=True,
❖                                      show_intermediates=True)
```

❖ ***Results :-***



❖ ***Analyzing the results :-***

NST aims to preserve the content of the input image while applying the style of a reference artwork. Key points of analysis include:

- **Fidelity to Content**: Evaluate how accurately the content of the original image is preserved in the generated result.
- **Adherence to Style**: Assess how faithfully the generated image reflects the textures, colors, and overall aesthetic of the chosen artistic style.
- **Insights Gained**: Understand how neural networks interpret and apply artistic styles, refine feature extraction techniques, and consider subjective aspects of aesthetic judgment.
- **Significance**: NST democratizes artistic expression, advances AI in image processing, and fosters interdisciplinary dialogue between technology and the arts.

❖ ***Conclusion :-***

- Neural style transfer (NST) effectively blends AI and artistic expression, preserving image content while applying intricate artistic styles. Future improvements should focus on enhancing style representation, semantic understanding, real-time applications, user customization, and exploring cross-domain transfers to advance creativity and practical applications.

❖ ***References :-***

https://arxiv.org/abs/1508.06576