TECHNOLOGICAL INSTITUTE OF THE PHILIPPINES
938 Aurora Blvd., Cubao, Quezon City

COLLEGE OF ENGINEERING AND ARCHITECTURE
ELECTRONICS ENGINEERING DEPARTMENT

1ST SEMESTER SY 2022 - 2023

Prediction and Machine Learning

COE 005
ECE41S11

Homework 2
Neural Style Transfer

Submitted to:
Engr. Christian Lian Paulo Riofloriddo

Submitted on:
10/19/2022

Submitted by:
Anthony R. Romano

Simulations and Discussions:

The first part of the code is just mounting the google drive account and loading the necessary libraries and parts that will be needed later such as the TFHub neural transfer model.

```
[54] from google.colab import drive #import google colab library
     drive.mount('/content/drive')  #drive the google drive to get the images

     Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

```
[55] import os               #import os for file calling
     import tensorflow as tf #import tensorflow for training later
     #Load compressed models from tensorflow_hub
     os.environ['TFHUB_MODEL_LOAD_FORMAT'] = 'COMPRESSED'
```

In this part, this is a part I added that I learned from the TensorFlow tutorial course, this is a code to store the image URL with its corresponding name and by using the @param feature, I was able to create a dropdown list to select which style I want to use.

```
#this code is to help the user choose which style they want to use
STYLE_IMAGE_NAME = 'Vincent_Van_Gogh' #@param ['Vincent_Van_Gogh',
#this is where the urls are stored based on  the name of the parame
corresponding_url = {
    'Vincent_Van_Gogh': 'https://upload.wikimedia.org/wikipedia/com
    'Juan_Luna': 'https://upload.wikimedia.org/wikipedia/commons/c/
}
#this this connects the style image name and the url that was used
style_image_path = tf.keras.utils.get_file(STYLE_IMAGE_NAME, corres
```

```
STYLE_IMAGE_NAME: Vincent_Van_Gogh
Vincent_Van_Gogh
Juan_Luna
```

Starting from this point, a large chunk of the code is mostly used for image loading and processing. The first part is importing of the different libraries to be used in the code. Then the next part is a function in which you convert the pixels values into RGB values that range from 0 to 255, that's why 255 was multiplied to the tensor in the second line after defining the function. It then converts it into a numpy array so that the machine will be able to read the colors.

```
[57] import IPython.display as display #library used for interactive computing

     import matplotlib.pyplot as plt   #libraries used for plotting and visualization
     import matplotlib as mpl
     mpl.rcParams['figure.figsize'] = (12, 12)
     mpl.rcParams['axes.grid'] = False

     import numpy as np #library for linear algebra
     import PIL.Image   #library used for image processing
     import time        #library to control time and represent it in code
     import functools   #library used for high order functions
     #defines the image converter and converts into a numpy array
     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)
     #directory of the base image and style image
     base_image_path = "/content/drive/MyDrive/Homework 2/Base Photo/Basephoto1.jpg"
     style_path = style_image_path
```

This part of the code is a continuation of the previous one, this is where we define the function that we will use to load the images from a directory into the code itself. And using the TensorFlow syntax, we use it to read the files from the directories, decode the image, and convert the image data type so that it will have the same data type as the other data, which is in float32.

```
[58] #defines the loading of an image and reading the file in the directory
     def load_img(path_to_img):
       max_dim = 512
       img = tf.io.read_file(path_to_img)
       img = tf.image.decode_image(img, channels=3)
       img = tf.image.convert_image_dtype(img, tf.float32)
     #scales the shape and dimension of the image
       shape = tf.cast(tf.shape(img)[:-1], tf.float32)
       long_dim = max(shape)
       scale = max_dim / long_dim

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

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

This is another continuation of the previous codes which just shows the plot of the image using the matplotlib and TensorFlow libraries.

```
[59] #defines the function to display the images
     def imshow(image, title=None):
       if len(image.shape) > 3:
         image = tf.squeeze(image, axis=0)

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

```
[60] #plots base image and the artist image
     base_image = load_img(base_image_path)
     style_image = load_img(style_path)

     plt.subplot(1, 2, 1)
     imshow(base_image, 'Base Image')

     plt.subplot(1, 2, 2)
     imshow(style_image, 'Artist Image')
```

At this part of the code, we load the imported hud neural transfer model that we used from TFHub. We set the parameters on what image to apply the style to and in this case the base image. More image processing takes place in the next few code blocks in order to make sure that the dimensions match.

```
[61] import tensorflow_hub as hub #import the library to use the tensorflow hub and import the model used from the hub
     hub_model = hub.load('https://tfhub.dev/google/magenta/arbitrary-image-stylization-v1-256/2')
     stylized_image = hub_model(tf.constant(base_image), tf.constant(style_image))[0]
     tensor_to_image(stylized_image)
```

```
     #processes the image and defines the VGG model to be used in testing and training
     x = tf.keras.applications.vgg19.preprocess_input(base_image*255)
     x = tf.image.resize(x, (224, 224))
     vgg = tf.keras.applications.VGG19(include_top=True, weights='imagenet')
     prediction_probabilities = vgg(x)
     prediction_probabilities.shape
```

```
[63] #predicts the top 5 vgg predictions
     predicted_top_5 = tf.keras.applications.vgg19.decode_predictions(prediction_probabilities.numpy())[0]
     [(class_name, prob) for (number, class_name, prob) in predicted_top_5]
```

This is the start of the VGG model, this neural transfer uses VGG19.

```
[64] #the model to be used in training and testing
     vgg = tf.keras.applications.VGG19(include_top=False, weights='imagenet')

     print()
     for layer in vgg.layers:
       print(layer.name) #prints the layers in the vgg
```

This part control which layers the base image and the style image are going to use in the model. This feeds the images to the corresponding layers where they will undergo convolution to generate a new image based on the two images.

```
[65] #determines which layers will include the base image layers and the style image layers
     content_layers = ['block5_conv2']

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

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

The following codes sets the parameters for each layer, and since this uses VGG19, it uses the premade model of the VGG19 and displays the overview of each layer.

```
[66] #defines the vgg layers based on the parameters on the previous code
     def vgg_layers(layer_names):
       """ Creates a VGG model that returns a list of intermediate output values."""
       #Load the vgg model, 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
```

```
[78] #extracts the layers where the style image layers were included based on earlier
     style_extractor = vgg_layers(style_layers)
     style_outputs = style_extractor(style_image*255)

     #Look at the statistics of each layer's output
     for name, output in zip(style_layers, style_outputs):
       print(name)
       print("  shape: ", output.numpy().shape)
       print("  min: ", output.numpy().min())
       print("  max: ", output.numpy().max())
       print("  mean: ", output.numpy().mean())
       print()
```

This part of the code is where we define the gram matrix of the neural transfer. The gram matrix determines the loss of the style image, each layer is calculated and contributes to the style loss based on the distance of the gram matrix.

```
[79] #defines the new matrix coputed using the einsum function in tensorflow
     def gram_matrix(input_tensor):
       result = tf.linalg.einsum('bijc,bijd->bcd', input_tensor, input_tensor)
       input_shape = tf.shape(input_tensor)
       num_locations = tf.cast(input_shape[1]*input_shape[2], tf.float32)
       return result/(num_locations)
```

The function returns the style image and base image tensors with the following parameters. And processed the gram matrix and the style layers to determine the loss of the style layer.

```
[82] #this entire code block returns the matrix ealier, it includes the style image layer matrix and the base image layer matrix
     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}
```

```
[84] #continuation of the StyleContent Model
     extractor = StyleContentModel(style_layers, content_layers)

     results = extractor(tf.constant(base_image))

     print('Styles:')
     for name, output in sorted(results['style'].items()):
       print("  ", name)
       print("    shape: ", output.numpy().shape)
       print("    min: ", output.numpy().min())
       print("    max: ", output.numpy().max())
       print("    mean: ", output.numpy().mean())
       print()

     print("Contents:")
     for name, output in sorted(results['content'].items()):
       print("  ", name)
       print("    shape: ", output.numpy().shape)
       print("    min: ", output.numpy().min())
       print("    max: ", output.numpy().max())
       print("    mean: ", output.numpy().mean())
```

```
[91] #this optimizes the image to prepare it for the style transfer
     style_targets = extractor(style_image)['style']
     content_targets = extractor(base_image)['content']

     image = tf.Variable(base_image)

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

     opt = tf.keras.optimizers.Adam(learning_rate=0.02, beta_1=0.99, epsilon=1e-1)

     style_weight=1e-2
     content_weight=1e4
```

```
[93] #this optimizes the image even more
@tf.function()
def train_step(image):
  with tf.GradientTape() as tape:
    outputs = extractor(image)
    loss = style_content_loss(outputs)

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

```
#to test is everything is working
train_step(image)
train_step(image)
train_step(image)
tensor_to_image(image)
```

This is the initial training, testing and application of the neural transfer in the images. This is where the machine attempts to recreate the base image with the style of choice. However, this still has loss that can still be reduced by changing the parameters.

```
[96] #testing and training of the model and application of the neural transfer
import time
start = time.time()

epochs = 5
steps_per_epoch = 100

step = 0
for n in range(epochs):
  for m in range(steps_per_epoch):
    step += 1
    train_step(image)
    print(".", end='', flush=True)
  display.clear_output(wait=True)
  display.display(tensor_to_image(image))
  print("Train step: {}".format(step))

end = time.time()
print("Total time: {:.1f}".format(end-start))
```

The following code blocks determine the total loss of the previous training, testing and neural transfer application. And the data from this can be used for the next code block.

```
[97] #determines the loss and plots the horizontal and vertical aspects of the original and transfered image
def high_pass_x_y(image):
  x_var = image[:, :, 1:, :] - image[:, :, :-1, :]
  y_var = image[:, 1:, :, :] - image[:, :-1, :, :]

  return x_var, y_var

x_deltas, y_deltas = high_pass_x_y(base_image)

plt.figure(figsize=(14, 10))
plt.subplot(2, 2, 1)
imshow(clip_0_1(2*y_deltas+0.5), "Horizontal Deltas: Original")

plt.subplot(2, 2, 2)
imshow(clip_0_1(2*x_deltas+0.5), "Vertical Deltas: Original")

x_deltas, y_deltas = high_pass_x_y(image)

plt.subplot(2, 2, 3)
imshow(clip_0_1(2*y_deltas+0.5), "Horizontal Deltas: Styled")

plt.subplot(2, 2, 4)
imshow(clip_0_1(2*x_deltas+0.5), "Vertical Deltas: Styled")

plt.figure(figsize=(14, 10))

sobel = tf.image.sobel_edges(base_image)
plt.subplot(1, 2, 1)
imshow(clip_0_1(sobel[..., 0]/4+0.5), "Horizontal Sobel-edges")
plt.subplot(1, 2, 2)
imshow(clip_0_1(sobel[..., 1]/4+0.5), "Vertical Sobel-edges")
```

```python
#determines the loss of the image and displays the loss
def total_variation_loss(image):
  x_deltas, y_deltas = high_pass_x_y(image)
  return tf.reduce_sum(tf.abs(x_deltas)) + tf.reduce_sum(tf.abs(y_deltas))

total_variation_loss(image).numpy()
tf.image.total_variation(image).numpy()
```

In this code block, the data from the previous codes, the loss, is used here to optimize the parameters to be used in the retraining, retesting and reapplication of the neural transfer. This part is important to make sure that the next application of the neural transfer will produce cleaner and better results.

```python
#optimizes the model even more by applying the new loss values and applying them
total_variation_weight=30

@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))

opt = tf.keras.optimizers.Adam(learning_rate=0.02, beta_1=0.99, epsilon=1e-1)
image = tf.Variable(base_image)
```

The last part of the code is the second testing, training and application of the neural transfer. This second application of the code is to ensure that there will be minimal loss since you determined the loss of the initial application and optimized the parameters used in the code. This makes the neural transfer look even cleaner and clearer.

```python
#testing and training of the model and application of the neural transfer on the new optimized parameters
import time
start = time.time()

epochs = 5
steps_per_epoch = 100

step = 0
for n in range(epochs):
  for m in range(steps_per_epoch):
    step += 1
    train_step(image)
    print(".", end='', flush=True)
  display.clear_output(wait=True)
  display.display(tensor_to_image(image))
  print("Train step: {}".format(step))

end = time.time()
print("Total time: {:.1f}".format(end-start))
```

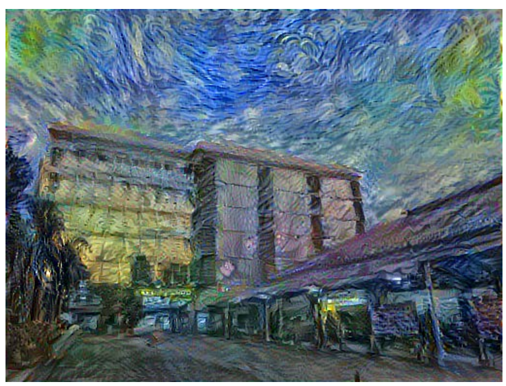
Final Output:

The final output shows the neural style transfer of the base image into the two styles chosen. They do not portray the thoughts and feelings of the original artworks; however, you can see that it created a filter that changed the style of the original image and made it look like they were painted by the famous artists.

Base Image

Vincent Van Gogh –
Starry Night

Juan Luna -
Spoliarium



Base Image – Vincent
Van Gogh Style

Base Image –
Juan Luna Style