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 Rioflorido

Submitted on:
OCTOBER 17, 2022

Submitted by:
RHAN VINCENT M. AUSTRIA

Introduction

Deep learning is a branch of machine learning that teaches a computer to carry out tasks that are similar to those performed by humans, such as speech and image recognition, and prediction making. It strengthens the capacity to categorize, identify, detect, and characterize utilizing data.

Convolutional Neural Networks, also known as CNNs, are a subset of artificial neural networks used in deep learning and are frequently employed for object, and image recognition, and categorization. Thus, Deep Learning uses a CNN to identify items or objects in an image. CNNs are being used extensively in a variety of tasks. Due to their major contribution to these rapidly developing and expanding fields, CNNs are widely used in deep learning.

What is a Convolutional Neural Network? A Convolutional Neural Network or CNN is a Deep Learning method that can take in an input picture, give various elements and objects in the image importance, and be able to distinguish between them. Comparatively speaking, a CNN requires substantially less pre-processing than other classification techniques. CNN have the capacity to learn these filters and properties, whereas in basic techniques filters are hand-engineered.

Convolutional Neural Networks consist of layers of small computational units that process visual information hierarchically in a feed-forward manner. Therefore, along the processing hierarchy of the network, the input image is transformed into representations that increasingly care about the actual content of the image compared to its detailed pixel values. We can directly visualize the information each layer contains about the input image by reconstructing the image only from the feature maps in that layer.

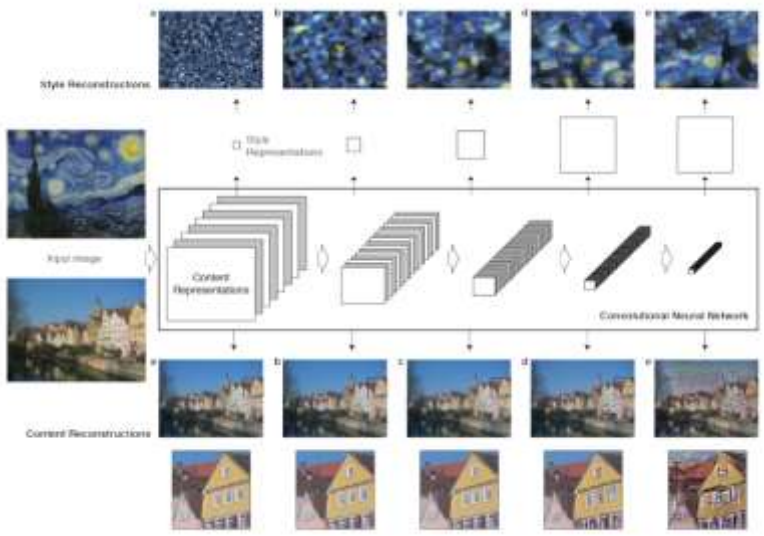

Figure 1.

According to the study of Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. Titled as A Neural Algorithm of Artistic Style, a given input image is represented as a set of filtered images at each processing stage in the CNN. While the number of different filters increases along the processing hierarchy, the size of the filtered images is reduced by some downsampling mechanism (e.g. max-pooling) leading to a decrease in the total number of units per layer of the network.

Coding/Setup

Import and configure modules

```
# import and configure modules

import os
import tensorflow as tf

# load compressed models from tensorflow_hub
os.environ['TFHUB_MODEL_LOAD_FORMAT'] = 'COMPRESSED'
```

Figure 2.

We first import os. Importing os allows us to use our operating system and provides or allows functions to create, move, and/or remove the current directory. TensorFlow is widely and mostly used in machine learning because its uses are in text-based applications, image recognition, voice search, and many more. Os.eniron is where we load the compressed models from tensferflow_hub.

```
import IPython.display as display

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

import numpy as np
import PIL.Image
import time
import functools
```

Figure 3.

IPython provides Interactive shells. It is a browser-based notebook interface with support for code, text, mathematical expressions, inline plots, and other media. Support for interactive data visualization and use of GUI toolkits. Matplotlib is where we create graphs and manipulate elements on a figure etc. It is also one of the most used in machine learning. Numpy is an array that is faster than a python list which is more compact and consumes less memory hence, it is easier or more convenient to use.

```
:
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)
```

Figure 4.

This part is where we define the tensor or we define the dimensions of the array.

```
# choose one image styles and one content

content_path = "Technological_Institute_of_the_Philippines_Quezon_City.jpg"

style_path = "The_Battle_of_Lepanto_of_1571_full_version_by_Juan_Luna.jpg"
style_path2 = "Thirty-six Views of Mount Fuji The Great Wave off Kanagawa Painting by Ukiyo-e.jpg"
```

Figure 5.

In figure 5. With the use of import os earlier, we then proceed to import the chosen base image and content style.

```
# visualize the input
# define a function to load an image and limit its maximum dimension to 512 pixels.


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)

  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
```

```
def imshow(image, title=None):
  if len(image.shape) > 3:
    image = tf.squeeze(image, axis=0)

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

```
content_image = load_img(content_path)
style_image = load_img(style_path)

plt.subplot(1, 2, 1)
imshow(content_image, 'Content Image')

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

Figure 6.

In Figure 6, The function is where we visualize the input while we define the limit and its maximum dimensions and content style.

```
# fast Style transfer using TF-Hub

import tensorflow_hub as hub
hub_model = hub.load('https://tfhub.dev/google/magenta/arbitrary-image-stylization-v1-256/2')
stylized_image = hub_model(tf.constant(content_image), tf.constant(style_image))[0]
tensor_to_image(stylized_image)
```

Figure 7.

In figure 7. The function shows that we import a hub model. We first try to see what a pre-trained algorithm from T-Hub before we execute the algorithm.

```
# define content and style representations

x = tf.keras.applications.vgg19.preprocess_input(content_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
```

```
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]
```

Figure 8.

In figure 8, we define content and style representations.

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

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

Figure 9.

In tensorflow.keras, there is a callable module, we call the VGG19. The VGG19 network architecture, a pretrained image classification network. These intermediate layers are necessary to define the representation of content and style from the images. For an input image, try to match the corresponding style and content target representations at these intermediate 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)
```

Figure 10.

We then choose intermediate layers from the network to represent the style and content of the image.

```
# build the model

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
```

```
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()
```

Figure 11.

In Figure 11, we then build the model. The networks in tf.keras.applications are designed so you can easily extract the intermediate layer values using the Keras functional API.

```
# calculate style

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)
```

Figure 12.

The content of an image is represented by the values of the intermediate feature maps. The Gram matrix includes the outer product of the feature vector with itself at each location, and averaging that outer product over all locations. This Gram matrix can be calculated for a particular layer as tf.linalg.einsum.

```
# extract style and content

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}
```

Figure 13

```
extractor = StyleContentModel(style_layers, content_layers)

results = extractor(tf.constant(content_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())
```

Figure 14.

In figure 13 and 14, we extract style and content. This function returns to the style and content tensors.

```
# run gradient descent

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

```
image = tf.Variable(content_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
```

```
def style_content_loss(outputs):
    style_outputs = outputs['style']
    content_outputs = outputs['content']
    style_loss = tf.add_n([tf.reduce_mean((style_outputs[name]-style_targets[name])**2)
                           for name in style_outputs.keys()])
    style_loss *= style_weight / num_style_layers

    content_loss = tf.add_n([tf.reduce_mean((content_outputs[name]-content_targets[name])**2)
                             for name in content_outputs.keys()])
    content_loss *= content_weight / num_content_layers
    loss = style_loss + content_loss
    return loss
```

Figure 15.

With this style and content extractor, you can now implement the style transfer algorithm. Do this by calculating the mean square error for your image's output relative to each target, then take the weighted sum of these losses. Lastly, we then set the style and content target values.

```
@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))
```

```
train_step(image)
train_step(image)
train_step(image)
tensor_to_image(image)
```

Figure 16.

We used tf.GradientTape to update the image. Then train and run the function. Since it is working we then proceed to perform longer optimization.

```
import time
start = time.time()

epochs = 10
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))
```

Figure 17.

```
# total variation loss

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(content_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")
```

```
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()
```

```
78473.2
```

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

```
array([78473.2], dtype=float32)
```

Figure 18.

Code for Total Variation Loss.

```
# re-run the optimization

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(content_image)
```

Figure 19.

Re-run the Optimization for better result.

Content Images

Base Image:

Photo A

Style Image:

A

The Battle of Lepanto of 1571 by Juan Luna

B

Thirty-Six Views of Mount Fuji The Great Wave off Kanagawa Painting by Ukiyo-e

Results

Result 1

Result 2

Reference(s):

Module 4.3. Convolutional Neural Networks

Code: https://www.tensorflow.org/tutorials/generative/style_transfer

In-Depth Analysis: https://youtu.be/LRzjcwRIoRs

The Study: https://arxiv.org/pdf/1508.06576.pdf