Neural style transfer

NOTE for GIFs lover: uses the TensorFlow GPU 2.4 (`tf2gpu`) kernel.

Description

The basic idea was first by Gatys in computer vision image in the style of another image (even and you could paint the Picasso or Van Gogh). The concept as Gram style transfer and the technique is explained in the paper, in this tutorial we follow a slightly simpler version.

Note: This tutorial demonstrates the original style-transfer algorithm. It optimizes the image content to a particular style. Modern approaches train a model to parameterize the stylized image directly, similar to pix2pix. The approach in both tasks is quite similar.

For a broad application of neural transfer check out this demo or the Deep photo style transfer paper or the paper and the Fast style transfer code or the tfjs-vis style transfer model demo TensorFlow Lite.

What is Neural Style

Neural style transfer is an optimization technique used to take two images—a content image and a style reference image (such as an artwork by a famous painter)—and blend them together so the output looks like the content image, but in the style of the style reference image.

This is implemented by optimizing the output image to match the content statistics of the content image and the style statistics of the style reference image. These statistics are extracted from the images using convolutional networks.

For example, let's take an image of the dog and Wassily Kandinsky's Composition 7:

Picture uploaded by User:Alexandros on Wikipedia licensing for use of Composition 7 in the public domain

Picture uploaded by User:Alexandros on Wikipedia licensing for use of Composition 7 in the public domain

Then the neural style algo (hopefully) returns an image similar to the content image, but in the style of Kandinsky the first

Setup

Pip install

```
!pip install -q tensorflow-hub
!wget -q https://upload.wikimedia.org/wikipedia/commons/0/0d/VanGogh-starry_night_ear.jpg
!wget -q https://upload.wikimedia.org/wikipedia/commons/thumb/6/64/Voyage_Pittoresque_Illustre_Orient_Livre_Deux_Pl_0020_-_Constantinople_Vue_Generale.jpg/1280px-Voyage_Pittoresque_Illustre_Orient_Livre_Deux_Pl_0020_-_Constantinople_Vue_Generale.jpg
!wget -q https://upload.wikimedia.org/wikipedia/commons/thumb/5/57/Hokusai_-_Under_the_Wave_off_Kanagawa.jpg/1280px-Hokusai_-_Under_the_Wave_off_Kanagawa.jpg
!wget -q https://upload.wikimedia.org/wikipedia/commons/thumb/3/4/Kandinsky%2C_Composition_7%2C_1913.jpg/1280px-Kandinsky%2C_Composition_7%2C_1913.jpg
```

Import and configure modules

```
import os
import tensorflow as tf
import tensorflow_hub as hub import tensorflow_text
os.environ["TFHUB_CACHE_DIR"] = "/kaggle/tmp"
```

```
import IPython.display as display
import matplotlib.pyplot as plt
import numpy as np
import PIL.Image, PIL.ImageFont, PIL.ImageDraw
import os, urllib, io, shutil, glob
import tensorflow as tf
import tensorflow_hub as hub
import tensorflow_datasets as tfds
import tensorflow_text
```

```
mpl = mpl.rcParams.update(
  'figure.figsize': (12,12),
  'axes.grid': False,
  'axes.spines.top': False,
  'axes.spines.right': False,
  'axes.spines.bottom': False,
  'axes.spines.left': False,
  'image.interpolation': 'nearest',
  'image.cmap': 'gray',
  'image.aspect': 'equal',
  'image.origin': 'upper',
  'image.resample': False,
  'pdf.compression': 0,
  'pdf.fonttype': 42,
  'ps.fonttype': 42,
  'path.simplify': False,
  'path.simplify_threshold': 1.0,
  'path.sketch': None,
  'path.snap': False,
  'agg.path.chunksize': 10000,
  'hatch.color': 'black',
  'hatch.linewidth': 0.5,
  'ps.papersize': 'letter',
  'ps.distiller.exe': None,
  'ps.useafm': False,
  'pdf.use14corefonts': False,
  'pdf.inheritance': True,
  'svg.image_inheritance_enabled': True
)
```

Select your style and content images.

By default, this notebook grabs a random pair of content and style.

To use your own images, drag and drop the style and content images to this folder, and edit the filenames to be the `content_path` and `style_path` below.

```
# If not using Kaggle:
!ln -s /content/*.jpg /kaggle/tmp/
content_path = "Tower_of_Pisa_Rome.jpg"
style_path = "Hokusai_-_Under_the_Wave_off_Kanagawa.jpg"
# If using Kaggle:
content_path = "/kaggle/input/neural-style-transfer-images/Tower_of_Pisa_Rome.jpg"
style_path = "/kaggle/input/neural-style-transfer-images/Hokusai_-_Under_the_Wave_off_Kanagawa.jpg"
```

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
```

Create a simple function to display an image:

```
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')
```

Fast Style Transfer using TF-Hub

This section introduces the original style-transfer algorithm, which optimizes the image content to a particular style. Before getting into the details, let's see how the TensorFlow Hub model does this.

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

Define content and style representations

Use the intermediate layers of the model to get the content and style representations of the image. Starting from the network's input layer, the first layer activations represent low-level features (like edges and textures). As you step through the network, the final few layers represent higher-level features—object parts like wheels or eyes. In this case, you are using the VGG19 network, a pre-trained image classification network. These intermediate layers are necessary to define the representation of content and style from the images. Given an input image, it's output the corresponding style and content layer representations in intermediate layers.

Load a TFHB and let run it on an image to ensure it's used correctly

```
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_probs = vgg(x)
for p, c, s in zip(*tf.keras.applications.vgg19.decode_predictions(prediction_probs.numpy()[:1], top=3)):
  print(s, '.....', c, '.....', p)
```

```
grassland_crab ..... 0.5 ..... 0.9999
ptarmigan ..... 0.5 ..... 0.9999
ruddy_turnstone ..... 0.5 ..... 0.9999
```

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

```
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)
```

Intermediate layers for style and content

So why do these intermediate layers, trained for a different image classification network allow us to define style and content representations?

At a high level, in order for a network to perform image classification (which this network has been trained to do), it must understand the image. This requires taking the raw image as input and building up internal representations that contains the raw image pixels into a complex understanding of the features present within the image.

It is reasonable to hope that in a deep neural network trained on a task like this, it will be necessary for this to capture the invariances and discriminating features within the image that are needed to define the content of an input image. The connections between where the one image is fed into the model and the output classification label. For deep neural net, it is reasonable to think that it has learned to define the complex representation of the inputted, such that it describes the content and of input image.

Build the model

The networks in VGG19 and MobileNetV2 are designed so you can easily extract the intermediate layer values using the Keras functional API.

For a model using the functional API, specify the inputs and outputs:

`model = tf.keras.Model(inputs, outputs)`

Using this functional model, a RGB image (with shape `[batch, h, w, 3]`) runs through the VGG19 model, and returns a list of intermediate layer outputs.

```
def vgg_layers(layer_names):
  """ Creates a vgg model that returns a list of intermediate output values."""
  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
```

Stack these layers to define a feature extraction 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, "...shape: ", output.shape)
  print("...min: ", output.numpy().min())
  print("...max: ", output.numpy().max())
  print("...mean: ", output.numpy().mean())
```

```
block1_conv1 ...shape:  (1, 256, 256, 64)
...min:  0.0
...max:  1.1178
...mean:  0.0
block2_conv1 ...shape:  (1, 128, 128, 128)
...min:  0.0
...max:  2.6465
...mean:  0.0
block3_conv1 ...shape:  (1, 64, 64, 256)
...min:  0.0
...max:  4.7344
...mean:  0.0
block4_conv1 ...shape:  (1, 32, 32, 512)
...min:  0.0
...max:  8.3047
...mean:  0.0
block5_conv1 ...shape:  (1, 16, 16, 512)
...min:  0.0
...max:  1.3057
...mean:  0.0
```

Calculate style

The content of an image is represented by the values of the intermediate feature maps.

It turns out, the style of an image can be described by the means and correlations across the different feature maps. Calculate a Gram matrix that includes this information by taking the outer product of the feature vector with itself at each location, and averaging that outer product over all locations. This Gram matrix is defined for a particular feature map.

$$G^l_{cd} = \frac{\sum_{ij} F^l_{ic} F^l_{jd}}{\text{shape}(F^l)}$$

The can be implemented concisely using the `tf.linalg.einsum` function:

```
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)
```

Extract style and content

Build a model that returns the style and content tensors.

```
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}
```

When called on a image, this model returns the gram matrix (style) of the `style_layers` and content of the `content_layers`:

```
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("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())
```

```
Styles:
   block1_conv1
     shape:  (1, 64, 64)
     min:  0.0055847
     max:  0.29358
     mean:  0.037189
   block2_conv1
     shape:  (1, 128, 128)
     min:  0.00078125
     max:  2.5161
     mean:  0.18814
   block3_conv1
     shape:  (1, 256, 256)
     min:  0.0
     max:  4.7344
     mean:  0.0
   block4_conv1
     shape:  (1, 512, 512)
     min:  0.0
     max:  8.3047
     mean:  0.0
   block5_conv1
     shape:  (1, 512, 512)
     min:  0.0
     max:  1.3057
     mean:  0.0
Contents:
   block5_conv2
     shape:  (1, 16, 16, 512)
     min:  0.0
     max:  1.3057
     mean:  0.0
```

Run gradient descent

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.

Set your style and content target values:

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

Define a `tf.Variable` to contain the image to optimize. To speed this up, initialize it with the content image (the `tf.Variable` must be in the same shape as the content image).

```
image = tf.Variable(content_image)
```

Since this is a float image, define a function to keep the pixel values between 0 and 1:

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

Create an optimizer. The paper recommends LBFGS, but Adam works okay, too:

```
opt = tf.optimizers.SGD(learning_rate=0.02, momentum=0.99, nesterov=True)
```

To calculate the, use a weighted combination of the two losses to get the total loss:

```
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
```

```
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))
```

Now run a few steps to test:

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

```
WARNING:tensorflow:Gradients do not exist for variables ['mixed_activation/mixed_activation_1:0', 'mixed_activation/mixed_activation_2:0', 'mixed_activation/mixed_activation_3:0', 'mixed_activation/mixed_activation_4:0', 'mixed_activation/mixed_activation_5:0', 'mixed_activation/mixed_activation_6:0', 'mixed_activation/mixed_activation_7:0', 'mixed_activation/mixed_activation_8:0', 'mixed_activation/mixed_activation_9:0', 'mixed_activation/mixed_activation_10:0', 'mixed_activation/mixed_activation_11:0', 'mixed_activation/mixed_activation_12:0', 'mixed_activation/mixed_activation_13:0', 'mixed_activation/mixed_activation_14:0', 'mixed_activation/mixed_activation_15:0', 'mixed_activation/mixed_activation_16:0', 'mixed_activation/mixed_activation_17:0', 'mixed_activation/mixed_activation_18:0', 'mixed_activation/mixed_activation_19:0', 'mixed_activation/mixed_activation_20:0', 'mixed_activation/mixed_activation_21:0', 'mixed_activation/mixed_activation_22:0', 'mixed_activation/mixed_activation_23:0', 'mixed_activation/mixed_activation_24:0', 'mixed_activation/mixed_activation_25:0', 'mixed_activation/mixed_activation_26:0', 'mixed_activation/mixed_activation_27:0', 'mixed_activation/mixed_activation_28:0', 'mixed_activation/mixed_activation_29:0', 'mixed_activation/mixed_activation_30:0', 'mixed_activation/mixed_activation_31:0', 'mixed_activation/mixed_activation_32:0', 'mixed_activation/mixed_activation_33:0', 'mixed_activation/mixed_activation_34:0', 'mixed_activation/mixed_activation_35:0', 'mixed_activation/mixed_activation_36:0', 'mixed_activation/mixed_activation_37:0', 'mixed_activation/mixed_activation_38:0', 'mixed_activation/mixed_activation_39:0', 'mixed_activation/mixed_activation_40:0', 'mixed_activation/mixed_activation_41:0',
```

That demonstrated what it does. But there's no need to implement it yourself, TensorFlow includes a standard implementation:

In [33]:
```
tf.image.total_variation(image).numpy()
```

Out[33]:
```
array([185439.84], dtype=float32)
```

Re-run the optimization

Choose a weight for the `total_variation_loss`:

In [34]:
```
total_variation_weight=30
```

Now include it in the `train_step` function:

In [35]:
```
@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))
```

Reinitialize the optimization variable:

In [36]:
```
image = tf.Variable(content_image)
```

And run the optimization:

In [37]:
```
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))
```

```
Train step: 1000
Total time: 27.3
```

Finally, save the result:

In [38]:
```
file_name = 'stylized-image.png'
tensor_to_image(image).save(file_name)

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

Extensions

- Repeat with your own style and content image.
  - What would be an interesting combination of style and content, and why?
- Extend the training steps? how does this change the result?
- Change the Content and Style weights above. How does this change the results?

References

This tutorial was adapted from a TF example on colab

It demonstrates the original style-transfer algorithm. For a simple application of style transfer check out this tutorial to learn more about how to use the arbitrary image style transfer model from TensorFlow Hub.