Neural style transfer

NOTE for DSR users: use the Tensorflow GPU 2.4 SpyDER kernel.

Description

The tutorial uses deep learning to compose one image in the style of another image (ever wondered what would happen if you asked a neural network to mash up two images? Wonder no more). The technique is outlined in A Neural Algorithm of Artistic Style (Gatys et al.).

Here, this tutorial implements the original style transfer algorithm. It optimizes the image content to match the content statistics and the style statistics of the style reference image. These statistics are extracted from the image using a convolutional network.

For example, let's take an image of the Neue Meue Museum in Kreuzberg:

Photo: Michael Jungblut, from Wikimedia Commons by CC Lizenz 3.0 DE-NC-ND

Photo: Kandinsky Circle 1913 Munich, decided to paint the window of his flat big enough, with Kandinsky's help, from Wikimedia Commons

the style of

Photo: Wassily Kandinsky - Composition 7, from Wikimedia Commons

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 image looks like the content image, but "painted" 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 a convolutional network.

For example, let's take an image of the Neue Meue Museum in Kreuzberg:

Import and configure modules

```
import os
import tensorflow as tf
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
tf.keyword_deprecation_warnings(False)

import matplotlib.pyplot as plt
import numpy as np, PIL, IPIL
from IPIL.iptools.imtools import *
from IPIL.iptools.ndimage import gaussian_filter
from IPIL.iptools.ndimage import sobel, generic_gradient_magnitude as ggm

NUMPY_DTYPE = 'float32'
IMAGE_SIZE = 512
CONTENT_WEIGHT = 5e0
STYLE_WEIGHT = 1e2
TV_WEIGHT = 1e-7
LEARNING_RATE = 1e-1
MAXFUN = 40
PRINTED_FREQ = 10
SAVED_FREQ = 10
```

Select your style and content images.

```
# Download the content image.
!curl -o content.jpg https://storage.googleapis.com/tensorflow-models-tutorials/neural-style-transfer/Green_Sea_Turtle_grazing_seagrass.jpg

# Download the style image.
!curl -o style.jpg https://storage.googleapis.com/tensorflow-models-tutorials/neural-style-transfer/Vassily_Kandinsky,_1913_-_Composition_7.jpg
```

Visualize the input

Define a function to load an image and limit its maximum dimension to 512 pixels.

```
def load_image(path, max_dim=None):
  img = PIL.Image.open(path).convert('L', dim=3)
  img = np.array(img, dtype=np.uint8, dtype=NUMPY_DTYPE)
  if max_dim is not None:
    img = resize(img, (max_dim, max_dim), preserve_range=True)
  img = np.expand_dims(img, axis=-1, dtype=NUMPY_DTYPE)
  return img / 255.
```

Create a single function to display an image.

```
def imshow(image, title=None):
  if image.shape[-1] == 3:
    image = image[...,0]
  plt.imshow(image, cmap='gray')
  if title:
    plt.title(title)
  plt.axis('off')

content_path = 'content.jpg'
style_path = 'style.jpg'

content_image = load_image(content_path)
style_image = load_image(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 tutorial demonstrates 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:

```
content_image = load_image(content_path, max_dim=512)
style_image = load_image(style_path, max_dim=512)

hub_module = tf.compat.v2.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]

plt.subplot(1, 3, 1)
imshow(content_image, 'Content Image')

plt.subplot(1, 3, 2)
imshow(style_image, 'Style Image')

plt.subplot(1, 3, 3)
imshow(stylized_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, the first few layer activations represent low-level features (like edges and texture). As one goes further, the last few layers represent higher-level features—object parts like wheels or eyes. In this case, you are using the VGG19 network architecture, a pre-trained image classification network. These intermediate layers are necessary to define the representations of content and style from the images. For an input image, try to find the corresponding style and content target representations at these intermediate layers.

Load a VGG19 and peek at its layers names to check it's well formed.

```
# Load images.
content_path = tf.keras.utils.get_file('YellowLabradorLooking_new.jpg', 'https://storage.googleapis.com/tensorflow-models-tutorials/neural-style-transfer/YellowLabradorLooking_new.jpg')
style_path = tf.keras.utils.get_file('kandinsky5.jpg', 'https://storage.googleapis.com/tensorflow-models-tutorials/neural-style-transfer/kandinsky5.jpg')

content_image = load_image(content_path, (256, 256))
style_image = load_image(style_path, (256, 256))

# Preprocess images.
content_image = content_image[tf.newaxis, ...]
style_image = style_image[tf.newaxis, ...]

# Define and create the model.
extractor = StyleContentModel(style_layers, content_layers)

# Look at the statistics of each layer.
style_outputs = extractor(style_image)*255
print('Styles:')
for name, output in sorted(zip(style_layers, style_outputs.keys())):
  print("  {:10:}".format(name, output.numpy().shape))
print("  mean: {:0.4f}, sd: {:0.4f}, min: {:0.4f}, max: {:0.4f}".format(output.numpy().mean(), output.numpy().std(), output.numpy().min(), output.numpy().max()))

print("Contents:")
content_outputs = extractor(content_image)*255
for name, output in sorted(zip(content_layers, content_outputs.keys())):
  print("  {:10:}".format(name, output.numpy().shape))
  print("  mean: {:0.4f}, sd: {:0.4f}, min: {:0.4f}, max: {:0.4f}".format(output.numpy().mean(), output.numpy().std(), output.numpy().min(), output.numpy().max()))
```

```
Styles:
  block1_conv1 (1, 256, 256, 64)
  mean: 0.1126, sd: 0.1966, min: 0.0000, max: 0.9966
  block2_conv1 (1, 128, 128, 128)
  mean: 0.2000, sd: 0.3000, min: 0.0000, max: 1.0000
  block3_conv1 (1, 64, 64, 256)
  mean: 0.0000, sd: 0.0000, min: 0.0000, max: 0.0000
  block4_conv1 (1, 32, 32, 512)
  mean: 0.0000, sd: 0.0000, min: 0.0000, max: 0.0000
  block5_conv1 (1, 16, 16, 512)
  mean: 0.0000, sd: 0.0000, min: 0.0000, max: 0.0000
Contents:
  block5_conv2 (1, 16, 16, 512)
  mean: 0.0000, sd: 0.0000, min: 0.0000, max: 0.0000
```

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']
```

Intermediate layers for style and content

So why do these intermediate outputs within our pre-trained 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 a representation of the image that is sufficient to extract visual features that could be confused by a human. One example of this is a trained convolutional network. The earlier layers in a convolutional network learn low-level feature detectors, while later layers in the network build on this understanding of the features present within the image.

This is also a convolutional neural network that has been trained to perform image classification. Using intermediate layers of the model allows us to get back complex feature maps corresponding to progressively larger and more sophisticated visual features. Convolutional between shows the non-linearities and pooling layers. Pooling layers are used to downsample the image. Consequently, the deeper convolutional layers represent the spatial extent of the image, but are lacking the fine-grained details.

Build the model

The networks in tf.keras.applications are developed so you can easily extract the intermediate layer values using the Functional API.

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

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

This simple function builds a VGG19 model that returns a list of intermediate layer outputs:

```
def vgg_layers(layer_names):
  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

content_layer = 'block5_conv2'
style_layers = ['block1_conv1',
                'block2_conv1',
                'block3_conv1',
                'block4_conv1',
                'block5_conv1']

# Build the model.
content_extractor = vgg_layers(content_layer)
style_extractor = vgg_layers(style_layers)

# Look at the statistics of each layer.
style_outputs = style_extractor(style_image)*255
print('Styles:')
for name, output in sorted(zip(style_layers, style_outputs.keys())):
  print("  {:10:}".format(name, output.numpy().shape))
  print("  mean: {:0.4f}, sd: {:0.4f}, min: {:0.4f}, max: {:0.4f}".format(output.numpy().mean(), output.numpy().std(), output.numpy().min(), output.numpy().max()))

print("Contents:")
content_outputs = content_extractor(content_image)*255
for name, output in sorted(zip(content_layer, content_outputs.keys())):
  print("  {:10:}".format(name, output.numpy().shape))
  print("  mean: {:0.4f}, sd: {:0.4f}, min: {:0.4f}, max: {:0.4f}".format(output.numpy().mean(), output.numpy().std(), output.numpy().min(), output.numpy().max()))
```

```
Styles:
  block1_conv1 (1, 256, 256, 64)
  mean: 0.1126, sd: 0.1966, min: 0.0000, max: 0.9966
  block2_conv1 (1, 128, 128, 128)
  mean: 0.2000, sd: 0.3000, min: 0.0000, max: 1.0000
  block3_conv1 (1, 64, 64, 256)
  mean: 0.0000, sd: 0.0000, min: 0.0000, max: 0.0000
  block4_conv1 (1, 32, 32, 512)
  mean: 0.0000, sd: 0.0000, min: 0.0000, max: 0.0000
  block5_conv1 (1, 16, 16, 512)
  mean: 0.0000, sd: 0.0000, min: 0.0000, max: 0.0000
Contents:
  block5_conv2 (1, 16, 16, 512)
  mean: 0.0000, sd: 0.0000, min: 0.0000, max: 0.0000
```

Extract style and content

Build a model that returns the style and content tensors.

```
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)

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 an image, this model returns the gram matrix (style) of the style_layers and the 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'].keys()):
  print("  {:10:}".format(name, output.numpy().shape))
  print("  mean: {:0.4f}, sd: {:0.4f}, min: {:0.4f}, max: {:0.4f}".format(output.numpy().mean(), output.numpy().std(), output.numpy().min(), output.numpy().max()))

print("Contents:")
for name, output in sorted(results['content'].keys()):
  print("  {:10:}".format(name, output.numpy().shape))
  print("  mean: {:0.4f}, sd: {:0.4f}, min: {:0.4f}, max: {:0.4f}".format(output.numpy().mean(), output.numpy().std(), output.numpy().min(), output.numpy().max()))
```

```
Styles:
  block1_conv1 (1, 64, 64)
  mean: 0.0018, sd: 0.0010, min: 0.0000, max: 0.0055
  block2_conv1 (1, 128, 128)
  mean: 0.0004, sd: 0.0006, min: 0.0000, max: 0.0023
  block3_conv1 (1, 256, 256)
  mean: 0.0000, sd: 0.0000, min: 0.0000, max: 0.0000
  block4_conv1 (1, 512, 512)
  mean: 0.0000, sd: 0.0000, min: 0.0000, max: 0.0000
  block5_conv1 (1, 512, 512)
  mean: 0.0000, sd: 0.0000, min: 0.0000, max: 0.0000
Contents:
  block5_conv2 (1, 16, 16, 512)
  mean: 0.0000, sd: 0.0000, min: 0.0000, max: 0.0000
```

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 can be calculated for a particular layer as:

$$G^l_{cd} = \frac{\sum_{ij} F^l_{ic}(x)F^l_{jd}(x)}{IJ}$$

This can be implemented in a single line 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 an image, this model returns the gram matrix (style) of the style_layers and the 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'].keys()):
  print("  {:10:}".format(name, output.numpy().shape))
  print("  mean: {:0.4f}, sd: {:0.4f}, min: {:0.4f}, max: {:0.4f}".format(output.numpy().mean(), output.numpy().std(), output.numpy().min(), output.numpy().max()))

print("Contents:")
for name, output in sorted(results['content'].keys()):
  print("  {:10:}".format(name, output.numpy().shape))
  print("  mean: {:0.4f}, sd: {:0.4f}, min: {:0.4f}, max: {:0.4f}".format(output.numpy().mean, output.numpy().std(), output.numpy().min(), output.numpy().max()))
```

```
Styles:
  block1_conv1 (1, 64, 64)
  mean: 0.0018, sd: 0.0010, min: 0.0000, max: 0.0055
  block2_conv1 (1, 128, 128)
  mean: 0.0004, sd: 0.0006, min: 0.0000, max: 0.0023
  block3_conv1 (1, 256, 256)
  mean: 0.0000, sd: 0.0000, min: 0.0000, max: 0.0000
  block4_conv1 (1, 512, 512)
  mean: 0.0000, sd: 0.0000, min: 0.0000, max: 0.0000
  block5_conv1 (1, 512, 512)
  mean: 0.0000, sd: 0.0000, min: 0.0000, max: 0.0000
Contents:
  block5_conv2 (1, 16, 16, 512)
  mean: 0.0000, sd: 0.0000, min: 0.0000, max: 0.0000
```

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 output with respect 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 make this quick, initialize it with the content image (the tf.Variable must be in the same shape as the content image):

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

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 also use Adam:

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

To calculate the loss, 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
```

Use tf.GradientTape to update the image.

```
@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))
```

Now run a few steps to test:

```
train_step(image)
train_step(image)
train_step(image)
plt.imshow(np.squeeze(image.numpy(), axis=0))
```

Loop it for a longer, better result:

```
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("Train step: {}".".format(step), end="\r")
  plt.imshow(np.squeeze(image.numpy(), axis=0))
  plt.title("Train step: {}".".format(step))
  plt.show()

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


Total variation loss

The underlying style transfer implementation results in pixel artifacts of high frequency artifacts. Decrease these using an explicit regularization term on the high-frequency components of the image. In style transfer, this often leads to better looking images:

```
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))

image = tf.Variable(content_image)

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("Train step: {}".".format(step), end="\r")
  plt.imshow(np.squeeze(image.numpy(), axis=0))
  plt.title("Train step: {}".".format(step))
  plt.show()

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


The output shows the high frequency components have decreased.

Also, the high frequency component is basically an edge detector. You can get similar results from the Sobel edge detector, for example:

```
plt.figure(figsize=(14,10))
sobel = np.squeeze(sobel(np.array(image[0], dtype=np.float32)).numpy(), axis=-1)
plt.imshow(np.abs(sobel), cmap='gray')
```

The regularization loss associated with this is the sum of the squares of the values:

In [31]:
```
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))
```

In [32]:
```
total_variation_loss(image).numpy()
```

Out[32]:
```
99456.53
```

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([99456.53], 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: 27.5
```

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.