Neural style transfer

[] by [Dmitry Kabanov](), version: TensorFlow 1(RN), 2.4.1(GoDL), tensor.

Description:

The tutorial uses deep learning to compose one image in the style of another image ever and you could paint like the masters thereafter (must pay $10 for royalties after 1 year). This is known as neural style transfer and the technique is outlined in A Neural Algorithm of Artist Style (Gatys et al.).

Note: This tutorial demonstrates the original style-transfer algorithm. It optimizes the image content to a particular style. Modern approaches train a model to generate the stylized image directly (similar to cycleGAN). This approach is much faster (up to 1000x).

For a quick application of a style transfer model see this Colab or these notes about how to use the pretrained Magenta image stylization model or how to use a style transfer model with 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 image 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 a convolutional network.

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

Yellow Labrador Looking, by Mel Chin(left) Composition 7, by Wassily Kandinsky(right)

Then we can ask the style transfer algorithm to recompose the content of the first dog as painted, with this style:

Setup

Run once:

```
!pip install -q tensorflow-hub
!pip install -q tensorflow-datasets tensorflow
```

```
Requirement already satisfied: tensorflow-hub in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: numpy>=1.19.5 in /usr/local/lib/python3.7/dist-packages (from tensorflow-hub)
Requirement already satisfied: protobuf>=3.19.6 in /usr/local/lib/python3.7/dist-packages (from tensorflow-hub)
ERROR: pip's dependency resolver does not support the current platform.
ERROR: tensorflow-datasets 4.0.1 requires tensorflow-metadata, which requires protobuf, which requires pyt
ERROR: tensorflow-datasets 4.0.1 requires tqdm, which requires pyreadline3, which requires ctypes, which r
ERROR: tensorflow-datasets 4.0.1 requires tqdm, which requires pyreadline3, which requires ctypes, which r
```

Import and configure modules

```
import os
import tensorflow as tf
import tensorflow_hub as hub
import numpy as np
import matplotlib.pyplot as plt
import PIL.Image, PIL.ImageFont, PIL.ImageDraw
import urllib.request
import io
import os.path
import os.path as path_lib
import tarfile
import zipfile
import shutil
import glob
import time
import math
import sys
import tensorflow_datasets as tfds
from tensorflow_datasets.core.utils.gcs_utils import _is_gcs_disabled
from IPO import Image, ImageFont, ImageDraw, ImageOps, ImageColor
from tensorflow_hub.tfv1.module.tf_v1 import Module, Module_v1
from collections import OrderedDict
import numpy as np
import tensorflow as tf
import tensorflow_hub as hub
import os
import os.path as path_lib
import glob
import math
import sys
import time
import urllib.request
import io
import PIL.Image, PIL.ImageFont, PIL.ImageDraw
import matplotlib.pyplot as plt
import tarfile
import zipfile
import shutil
import os.path
import os
import tensorflow as tf
import tensorflow_hub as hub
import numpy as np
import matplotlib.pyplot as plt
import PIL.Image, PIL.ImageFont, PIL.ImageDraw
import urllib.request
import io
import os.path
import os.path as path_lib
import tarfile
import zipfile
import shutil
import glob
import math
import sys
import time
import tensorflow_datasets as tfds
from tensorflow_datasets.core.utils.gcs_utils import _is_gcs_disabled
from IPO import Image, ImageFont, ImageDraw, ImageOps, ImageColor
from tensorflow_hub.tfv1.module.tf_v1 import Module, Module_v1
from collections import OrderedDict
```

Select your style and content images.

Select any two images, one for the content of a photo and one a masterpiece.

To make your images, drag and drop the style and content images to the folder, and edit the filenames to the content_path and to style_path below.

```
# Content image
content_path = tf.keras.utils.get_file('YellowLabradorLooking_new.jpg', 'https://bit.ly/mlcc-ds-yellow-lab-dog')
# Style image
style_path = tf.keras.utils.get_file('kandinsky5.jpg', 'https://bit.ly/mlcc-ds-kandinsky-5')
```

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)
  return img[tf.newaxis, :]
```

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 demonstrates the model image style transfer algorithm, which optimizes the image content to a particular style. Before getting into the details, let's see how TensorFlow Hub does this with a pre-trained model.

```
import tensorflow_hub as hub
hub_module = hub.load('https://tfhub.dev/google/magenta/arbitrary-image-stylization-v1/256/1')
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 texture. As one moves up the network, the last few layers represent higher-level features—object parts like wheels or eyes. In the case of VGG19, we use the following layers to represent the content and style of an image:

Because of the ImageNet pre-training, the early layers of the network tend to learn a general representation of the image. In the middle layers, it is needed to recognizing distinct objects, and in the deeper layers, we start to see the activations representing complex features. Finally, the output layer is dense with information about the presence of various objects in the image. This is why it is sufficient for classification. In the case of style transfer, we need to capture the texture, color, and overall spatial information in the image at each layer. In the case of content representation, we want to add some spatial information but leave the details of the image to be free to move.

Build the model

The networks in VGG19 and InceptionV3 are developed so you can easily extract the intermediate layer values using the Keras functional API.

Define a function to create a model that gives a list of intermediate layer outputs:

```
# Choose intermediate layers from the network to represent the style and content of the image
content_layers = ['block5_pool3']

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 on a large-scale 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 be), it has to understand a representation of the image from its raw pixels. We also have to be able to take this image representation and convert it into an internal representation that conveys the content of the image. Another piece of this puzzle is that we have to be able to understand the spatial and color information for the image. This is also known why convolutional neural networks are able to generalize well: they're able to capture the invariances and defining features within classes (e.g., cats vs. dogs) that are agnostic to background noise and other nuisances. Thus, somewhere between where the raw image is fed into the model and the output classification label, the model serves as a complex feature extractor. By accessing intermediate layers of the model, we're able to describe the content and style of input images.

Build the model

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

And to create the model:

```
style_extractor = vgg_layers(style_layers)
style_outputs = style_extractor(preprocess_image(style_image))

for layer in style_outputs:
  print("...shape: ", layer.shape)
  print("...min: ", tf.reduce_min(layer))
  print("...max: ", tf.reduce_max(layer))
  print("...mean: ", tf.reduce_mean(layer))
```

```
...shape:  (1, 64, 64, 64)
...min:  tf.Tensor(0.0, shape=(), dtype=float32)
...max:  tf.Tensor(1.0, shape=(), dtype=float32)
...mean:  tf.Tensor(0.0, shape=(), dtype=float32)
...shape:  (1, 32, 32, 128)
...min:  tf.Tensor(0.0, shape=(), dtype=float32)
...max:  tf.Tensor(1.0, shape=(), dtype=float32)
...mean:  tf.Tensor(0.0, shape=(), dtype=float32)
...shape:  (1, 16, 16, 256)
...min:  tf.Tensor(0.0, shape=(), dtype=float32)
...max:  tf.Tensor(1.0, shape=(), dtype=float32)
...mean:  tf.Tensor(0.0, shape=(), dtype=float32)
...shape:  (1, 8, 8, 512)
...min:  tf.Tensor(0.0, shape=(), dtype=float32)
...max:  tf.Tensor(1.0, shape=(), dtype=float32)
...mean:  tf.Tensor(0.0, shape=(), dtype=float32)
...shape:  (1, 4, 4, 512)
...min:  tf.Tensor(0.0, shape=(), dtype=float32)
...max:  tf.Tensor(1.0, shape=(), dtype=float32)
...mean:  tf.Tensor(0.0, shape=(), dtype=float32)
```

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:

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

You can implemented concisely using the `tf.linalg.einsum`:

```
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 = preprocess_image(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'].items):
  print("  {:...shape: {:...min: {:...max: {:...mean: {:...".format(name, output.shape,
                                                                   tf.reduce_min(output),
                                                                   tf.reduce_max(output),
                                                                   tf.reduce_mean(output))
print("Contents:")
for name, output in sorted(results['content'].items):
  print("  {:...shape: {:...min: {:...max: {:...mean: {:...".format(name, output.shape,
                                                                   tf.reduce_min(output),
                                                                   tf.reduce_max(output),
                                                                   tf.reduce_mean(output))
```

```
Styles:
  block1_conv1:...shape: (1, 64, 64, 64)...min: tf.Tensor(0.0, shape=(), dtype=float32)...max: tf.Tensor(1.0, shape=(), dtype=float32)...mean: tf.Tensor(0.0, shape=(), dtype=float32)...
  block2_conv1:...shape: (1, 32, 32, 128)...min: tf.Tensor(0.0, shape=(), dtype=float32)...max: tf.Tensor(1.0, shape=(), dtype=float32)...mean: tf.Tensor(0.0, shape=(), dtype=float32)...
  block3_conv1:...shape: (1, 16, 16, 256)...min: tf.Tensor(0.0, shape=(), dtype=float32)...max: tf.Tensor(1.0, shape=(), dtype=float32)...mean: tf.Tensor(0.0, shape=(), dtype=float32)...
  block4_conv1:...shape: (1, 8, 8, 512)...min: tf.Tensor(0.0, shape=(), dtype=float32)...max: tf.Tensor(1.0, shape=(), dtype=float32)...mean: tf.Tensor(0.0, shape=(), dtype=float32)...
  block5_conv1:...shape: (1, 4, 4, 512)...min: tf.Tensor(0.0, shape=(), dtype=float32)...max: tf.Tensor(1.0, shape=(), dtype=float32)...mean: tf.Tensor(0.0, shape=(), dtype=float32)...
Contents:
  block5_pool3:...shape: (1, 16, 16, 512)...min: tf.Tensor(0.0, shape=(), dtype=float32)...max: tf.Tensor(1.0, shape=(), dtype=float32)...mean: tf.Tensor(0.0, shape=(), dtype=float32)...
```

Run gradient descent

With style and content extractors ready, we can now implement the style transfer algorithm. Do this by calculating the mean square error for your image 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 make this quick, initialize it with the content image (the `tf.Variable` must be the same shape as the content image):

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

Since it 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.keras.optimizers.SGD(learning_rate=0.02, momentum=0.99)
```

To calculate the loss 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 * style_loss
    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 * content_loss
    loss = style_loss + content_loss
    return loss
```

Use `tf.GradientTape` to update the image.

```
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 several steps to test everything works.

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

```
Train step: 1
Train step: 2
Train step: 3
Train step: 4
Train step: 5
Train step: 6
Train step: 7
Train step: 8
Train step: 9
Train step: 10
Train step: 11
Train step: 12
Train step: 13
Train step:
```

```
  return tf.reduce_sum(tf.abs(x_deltas)) + tf.reduce_sum(tf.abs(y_deltas))
```

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

Out[32]:
```
147823.25
```

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([147823.25], 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: 27.0
```

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.