Neural network programming: style transfer

Erik Spence

SciNet HPC Consortium

22 May 2018

Today's code and slides

You can get the slides for today's class at the SciNet Education web page.

`https://support.scinet.utoronto.ca/education`

Click on the link for the class, and look under "File Storage".

Style transfer

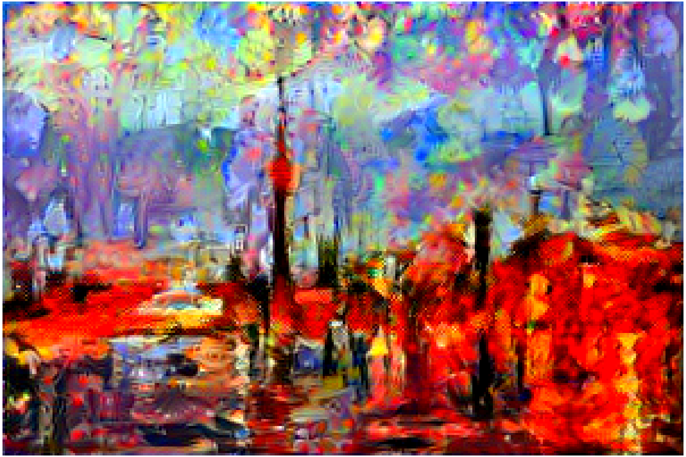
As a change of pace, today we're going to play with an interesting application of neural networks, rather than build and train one from scratch.

This application leverages Convolutional Neural Network's strengths in visual pattern recognition and object localization.

Interestingly, CNNs can also distinguish between "content" of an image, and "style". This can be used to impose the style of one image onto another. This is known as "style transfer".

Style transfer, an example

Style transfer, an example, continued



Style transfer, another example

Style transfer, another example, continued

Style transfer, continued

So how does it work? The technique proceeds as follows.

- Start with two images, the 'content' image (the photo), and the 'style' image (the painting).
- We create a loss function, which measures how close the 'content' of the created image is to the 'content' image.
- We create a second loss function, which measures how close the 'style' of the created image is to the 'style' image.
- We combine these two loss functions into a single loss function.
- We use scipy, rather than Keras, to minimize the combined loss function, in the process creating our new image.
- Note that the minimization is done, not by adjusting the weights and biases of a neural network, but rather by choosing the output which minimizes the loss function.

VGG19

Rather than build our own neural network this class, we will use one which has already been trained: VGG19.

- This network was developed by the Visual Geometry Group (VGG), at Oxford University.
- The VGG group won second prize in the 2014 ILSVRC (ImageNet) competition in the "localization and classification" category.
- The network we will use is deep, consisting of 19 trainable layers.
- The network is available, pre-trained on ImageNet data, as part of Keras.
- Most of the trainable layers are convolutional layers. These are the ones we're interested in.

We can use the output of the intermediate layers of the neural network to assess both 'content' and 'style' of the new image, and contrast them to content and style images.

VGG19, continued

The VGG19 network is too big to draw. It's organized into blocks. Here's an outline.

- Block 1:
  - Conv2D(64, (3, 3), padding = 'same')
  - Conv2D(64, (3, 3), padding = 'same')
  - MaxPooling2D((2, 2), strides = (2, 2))
- Block 2:
  - Conv2D(128, (3, 3), padding = 'same')
  - Conv2D(128, (3, 3), padding = 'same')
  - MaxPooling2D((2, 2), strides = (2, 2))
- Block 3:
  - Conv2D(256, (3, 3), padding = 'same')
  - Conv2D(256, (3, 3), padding = 'same')
  - Conv2D(256, (3, 3), padding = 'same')
  - Conv2D(256, (3, 3), padding = 'same')
  - MaxPooling2D((2, 2), strides = (2, 2))
- Block 4 & 5:
  - Conv2D(512, (3, 3), padding = 'same')
  - Conv2D(512, (3, 3), padding = 'same')
  - Conv2D(512, (3, 3), padding = 'same')
  - Conv2D(512, (3, 3), padding = 'same')
  - MaxPooling2D((2, 2), strides = (2, 2))
- Dense(4096)
- Dense(4096)
- Dense(numclasses)

There are 19 trainable layers.

Content loss function

We can use the intermediate layers of the VGG19 network to measure 'content'.

- As you saw on the previous slide, the network is organized into blocks of hidden layers, with max-pooling layers between each block.
- The convolutional layers learn features of the training data.
- Let the output of an intermediate layer when the content image is used as input to the network be $\hat{\boldsymbol{F}}$, and when the generated image is used as input be $\boldsymbol{F}$.

We can then define a content loss function as

$$\boldsymbol{L}_{\textbf{content}} = \sum_{\boldsymbol{i}} \left(\hat{\boldsymbol{F}}_{\boldsymbol{i}} - \boldsymbol{F}_{\boldsymbol{i}} \right)^{\mathbf{2}}$$

Where the sum over $\boldsymbol{i}$ is over all feature maps of the given layer.

Style loss function

We can use the intermediate layers of the VGG19 network to also measure 'style'.

- How do we measure 'style'? We use a technique which was originally developed in 'texture analysis' (a subfield of computer image analysis) to calculate the 'texture' of an image.
- This attempts to quantify the perceived 'texture' of the image (smoothness, bumpiness, roughness, silkiness).
- The correlations between the output of the feature maps in a given layer are used to capture the desired information.
- These are computed using a Gram matrix.

$$G^l_{ij} = \sum_k F^l_{ik} F^l_{jk}$$

where G^l_{ij} is the inner product between feature map i and j in layer l.

Style loss function, continued

We can use the Gram matrices of the output of the feature maps as a measure of the 'style'.

- Rather than use the output of a single layer, we will use the output of several layers.
- We use the Gram matrices of both the style image and the new image to create a loss function.
- Weights can be added to the loss function, to control the importance of one layer over another.

$$L_{\text{style}} = \sum_l \frac{w_l}{4 N_l^2 M_l^2} \sum_{ij} \left(G_{ij}^l - \tilde{G}_{ij}^l \right)^2$$

where G_{ij}^l and $\tilde{G}_{ij}^l$ are the Gram matrices for the new and style images, N_l and M_l are the number and size of the feature maps in layer l, and w_l is the weight given to the feature maps in layer l.

Total loss function

We can now construct the total loss function we will try to minimize.

$$L_{\text{total}} = \alpha L_{\text{content}} + \beta L_{\text{style}}$$

- The values of α and β can be varied to get different effects.
- Generally speaking, the value of α is at least 2 orders of magnitude larger than β, otherwise you end up with all style and no content.

We are now ready to proceed.

The algorithm

Ok, so what's the algorithm?

- Load the content and style images. Create a Keras placeholder for the generated image.
- Load the VGG19 model, with the ImageNet training weights.
- Using one of the later VGG19 layers, construct the content loss function.
- Using the layers succeeding each max-pooling layer, construct the style loss function.
- Create the total loss function. Calculate the gradient of the total loss.
- Iterate, using scipy.optimize's fmin_l_bfgs_b function, to optimize the generated image. This uses the L-BFGS-B algorithm.

Let's do it!

Style transfer, the code

```python
# my_style_transfer.py
import scipy.misc as scm
import scipy.optimize as sco
import numpy as np

import keras.preprocessing.image as kpi
import keras.applications.vgg19 as kav19
import keras.backend as K

def gram_matrix(x):
  new_x = K.permute_dimensions(x, (2, 0, 1))
  features = K.batch_flatten(new_x)
  gram = K.dot(features, K.transpose(features))
  return gram
```

```python
# my_style_transfer.py, continued

def style_loss(style_img, new_img):

  nrows, ncols, nchannels = style_img.shape

  S = gram_matrix(style_img)
  C = gram_matrix(new_img)

  s_loss = K.sum(K.square(S - C)) /
    (2. * nchannels * nrows * ncols)**2

  return s_loss
```

Style transfer, the code, continued

```python
# my_style_transfer.py, continued

def content_loss(content_img, new_img):
  return K.sum(K.square(content_img - new_img))

def get_loss_and_grads(x):
  # x comes in flattened, so reshape.
  x = x.reshape((1, img_nrows, img_ncols, 3))

  # This function is defined below.
  outs = f_outputs([x])

  loss = outs[0]
  if len(outs[1:]) == 1:
    grads = outs[1].flatten()
  else:
    grads = np.array(outs[1:]).flatten()
  return loss, grads
```

```python
# my_style_transfer.py, continued
class Evaluator(object):
  def __init__(self):
    self.loss_value = None
    self.grad_values = None

  def loss(self, x):
    loss_value, grad_values = \
      get_loss_and_grads(x)

    self.loss_value = loss_value
    self.grad_values = grad_values
    return self.loss_value

  def grads(self, x):
    return self.grad_values
```

Style transfer, the code, continued more

```python
# my_style_transfer.py, continued

content_img = K.variable(preprocess_image(
  content_image_path, img_nrows, img_ncols))
style_img = K.variable(preprocess_image(
  style_image_path, img_nrows, img_ncols))
new_img = K.placeholder(
  (1, img_nrows, img_ncols, 3))

input_tensor = K.concatenate([content_img,
  style_img, new_img], axis = 0)

model = kav19.VGG19(input_tensor = input_tensor,
  weights = "imagenet", include_top = False)

outputs_dict = {layer.name: layer.output
  for layer in model.layers}
```

```python
# my_style_transfer.py, continued

layer_fms = outputs_dict["block5_conv2"]
content_fm = layer_fms[0, :, :, :]
new_fm = layer_fms[2, :, :, :]
loss = content_weight *
  content_loss(content_fm, new_fm)

feature_layers = ["block1_conv1", "block2_conv1",
  "block3_conv1", "block4_conv1", "block5_conv1"]

for layer in feature_layers:
  layer_fms = outputs_dict[layer]
  style_fm = layer_fms[1, :, :, :]
  new_fm = layer_fms[2, :, :, :]
  loss += style_weight *
    style_loss(style_fm, new_fm)
```

Style transfer, the code, continued even more

```python
# my_style_transfer.py, continued

# Use the backend to calculate the gradients.
grads = K.gradient(loss, new_img)

outputs = [loss]
outputs.append(grads)

# Create a Keras function to calculate the
# loss and gradient.
f_outputs = K.function([new_img], outputs)

# The class that calculates the loss and
# gradient in a single pass.
evaluator = Evaluator()
```

```python
# my_style_transfer.py, continued

# We start with the content image.
x = preprocess_image(content_image_path,
  img_nrows, img_ncols)

# Now loop and do small improvements, saving
# images as we go.
for i in range(num_iter):
  x, min_val, info = \
    sco.fmin_l_bfgs_b(evaluator.loss, x.flatten(),
    fprime = evaluator.grads, maxfun = 20)
  print "Current loss:", min_val

  img = deprocess_image(x.copy(),
    img_nrows, img_ncols)
  scm.imsave("image_%d.png" % i, img)
```

Notes about the code

The code invokes some Theano/Tensorflow magic.

- We cast all variables as Keras backend variables and placeholders. This allows us to use the built-in functionality of the backend to do magical operations.
- In particular, once the loss is set up, we can calculate its derivative with a call to the backend's 'gradient' function.
- We can also use a Keras function to calculate the loss, once the graph has been constructed by the backend. This simplifies matters significanly, if mysteriously.
- If you don't quite understand why this all works, you should review the Tensorflow class material.

The code looks alot like Tensorflow code, because we're using the backend functionality of Keras, rather than the higher-level functionality.

Running the code

```
ejspence@mycomp ~> python my_style_transfer.py toronto-skyline.jpg image1.jpg output
Using Theano backend.
Model loaded.
Start of iteration 0
Current loss value:  6418551894.97
Image saved as output_at_iteration_0.png
Iteration 0 completed in 2s
Start of iteration 1
Current loss value:  3014825460.88
Image saved as output_at_iteration_1.png
Iteration 1 completed in 2s
⋮
Iteration 198 completed in 2s
Start of iteration 199
Current loss value:  135095763.633
Image saved as output_at_iteration_199.png
ejspence@mycomp ~>
```

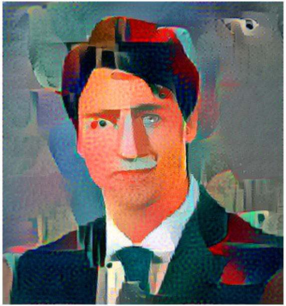
Animating the result

Varying the content weight

As you would expect, for a style weight of 1.0, varying the content weight changes the final image.

0.01

0.025



0.1

0.5

Photo style transfer

It’s all well and good to take the style of a painting and apply it to a photo. But what about photo-to-photo transfers?

- This is known as photo style transfer. Also called ’neural Photoshop’.
- It allows you to transfer things like lighting style, time of day, weather, etc., from one photo to another.
- Photo-to-photo style transfer, using the algorithm presented previously, doesn’t work. You end up with painting-like distortions to the output photo.
- To be considered successful, the output must maintain the structure of the content of the input photograph.
- The output must also maintain "symantic accuracy": the sky should still look like the sky, buildings like buildings.

Photo style transfer, continued

How is this accomplished?

- The authors of this work add a "photorealism" regularization term to the cost function, constraining the output image to be represented by locally affine colour transformations.
- The symantic accuracy problem is addressed by creating labelled image segmentation masks (water, building, sky, etc.) for the input and reference images, and adding the masks to the input image as additional channels.
- The calculation of the Gram matrices is modified to account for the extra mask channels.

The results are impressive. The paper is linked at the end of the slides.

Photographic style transfer, results

input

previous technique

reference

current technique

Linky goodness

Style transfer:

- https://arxiv.org/pdf/1508.06576.pdf (the original paper)
- https://medium.com/mlreview/making-ai-art-with-style-transfer-using-keras-8bb5fa44b216
- https://github.com/titu1994/Neural-Style-Transfer
- https://chrisrodley.com/2017/06/19/dinosaur-flowers

Photo style transfer:

- https://arxiv.org/pdf/1703.07511.pdf