conv_filter_visualization

April 16, 2018

1 What CNN See

This notebook is based solely from the content presented in https://blog.keras.io/how-convolutional-neural-networks-see-the-world.html by *Francois Chollet* and https://github.com/keras-team/keras/blob/master/examples/conv_filter_visualization.py.

Some extra content have been added from the original content to aid my understanding of using **Keras**.

This notebook visualises what deep convolutional neural networks see from the images we feed them.

We will start by defining the VGG16 model in Keras by

(1) Import Keras applications using the command `from keras import applications` Keras Applications are deep learning models that are made available alongside pre-trained weights.

(2) Instantiate a VGG16 model (performing this, weights are downloaded automatically to ~/.keras/models/.

```
keras.applications.vgg16.VGG16(include_top=True, weights='imagenet', input_tensor=None, inp
```

- include_top: whether to include the 3 fully-connected layers at the top (end) of the network. _In this code, we set `include_top = False`. This is because adding the FC layers requires us to use a fixed input size for the model (i.e. 224x244, the original ImageNet format).

- weights: it can be `None` for random initilisation or `imagenet` for pre-trained weights on ImageNet

```
In [1]: from keras import applications

        # Build the VGG16 network
        model = applications.VGG16(include_top = False,
                                   weights = 'imagenet')
        model.summary()
```

```
/home/supannee/tensorflow/lib/python3.5/site-packages/h5py/__init__.py:36: FutureWarning: Conver
  from ._conv import register_converters as _register_converters
Using TensorFlow backend.
```

```
-----------------------------------------------------------------
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, None, None, 3)     0
-----------------------------------------------------------------
block1_conv1 (Conv2D)        (None, None, None, 64)    1792
-----------------------------------------------------------------
block1_conv2 (Conv2D)        (None, None, None, 64)    36928
-----------------------------------------------------------------
block1_pool (MaxPooling2D)   (None, None, None, 64)    0
-----------------------------------------------------------------
block2_conv1 (Conv2D)        (None, None, None, 128)   73856
-----------------------------------------------------------------
block2_conv2 (Conv2D)        (None, None, None, 128)   147584
-----------------------------------------------------------------
block2_pool (MaxPooling2D)   (None, None, None, 128)   0
-----------------------------------------------------------------
block3_conv1 (Conv2D)        (None, None, None, 256)   295168
-----------------------------------------------------------------
block3_conv2 (Conv2D)        (None, None, None, 256)   590080
-----------------------------------------------------------------
block3_conv3 (Conv2D)        (None, None, None, 256)   590080
-----------------------------------------------------------------
block3_pool (MaxPooling2D)   (None, None, None, 256)   0
-----------------------------------------------------------------
block4_conv1 (Conv2D)        (None, None, None, 512)   1180160
-----------------------------------------------------------------
block4_conv2 (Conv2D)        (None, None, None, 512)   2359808
-----------------------------------------------------------------
block4_conv3 (Conv2D)        (None, None, None, 512)   2359808
-----------------------------------------------------------------
block4_pool (MaxPooling2D)   (None, None, None, 512)   0
-----------------------------------------------------------------
block5_conv1 (Conv2D)        (None, None, None, 512)   2359808
-----------------------------------------------------------------
block5_conv2 (Conv2D)        (None, None, None, 512)   2359808
-----------------------------------------------------------------
block5_conv3 (Conv2D)        (None, None, None, 512)   2359808
-----------------------------------------------------------------
block5_pool (MaxPooling2D)   (None, None, None, 512)   0
=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
-----------------------------------------------------------------
```

Create a dictionary for (layer.name, layer) using layer.name as key because each layer has a unique name. We could list the layer name by

```
for layer in model.layers:
    print(layer.name)
    #print('input_shape: {}\toutput_shape: {}'.format(layer.input_shape,
                                                     layer.output_shape))
```

This results in a list of all layer in VGG16 network (notice the missing FC layers as the results of setting `include_top = False`).

input_1 block1_conv1 block1_conv2 block1_pool block2_conv1 block2_conv2 block2_pool block3_conv1 block3_conv2 block3_conv3 block3_pool block4_conv1 block4_conv2 block4_conv3 block4_pool block5_conv1 block5_conv2 block5_conv3 block5_pool

```
In [2]: layer_dict = dict([(layer.name, layer) for layer in model.layers])
```

Next, define a **loss function** that will seek to **maximize the activation** of a specific filter (filter_index) in a specific layer (layer_name). We do this via a Keras backend function, which allows our code to run both on top of TensorFlow and Theano.

```
from keras import backend as K
```

`layer_name` can be set using following `layer.name` of any layer with filter. For example,

```
layer_name = 'block5_conv3'
```

`filter_index` can be set to any integer that corresponds to the filter index in that layer. For example, there are 512 filters for `block5_conv3`. Thus, we can set the index to any value between 0 and 511.

```
In [3]: from keras import backend as K

        layer_name = 'block5_conv3'
        filter_index  = 0 # can be any integer from 0 to 511, as there are 512 filters

        # build a loss function that maximises the activation of the nth filter of the chosen la
        # 1. obtain the output of the selected layer 'layer_name'
        layer_output = layer_dict[layer_name].output
        # 2. Get the mean of the tensor at filter_index
        if K.image_data_format() == 'channels_first':
            loss = K.mean(layer_output[:, filter_index, :, :])
        else:
            loss = K.mean(layer_output[:, :, :, filter_index])

        # Define the placeholder for the input images
        input_img = model.input

        # compute the gradient of the input picture wrt this loss
        grads = K.gradients(loss, input_img)[0]
```

```
# normalise the gradient
small_value = 1e-5 # or K.epsilon()
grads /= (K.sqrt(K.mean(K.square(grads))) + small_value)

# This function returns the loss and grads given the input picture
iterate = K.function([input_img], [loss, grads])
```

In the code above, **the gradient of the pixels of the input image is normalised**. This avoids very small and very large gradients which ensure smooth gradient ascent process. Next, we use the Keras function we defined to do **gradient ascent** in the input space, with regard to our filter activation loss.

```
In [8]: import numpy as np
        from scipy.misc import imsave

        # Let's define dimensions of the generated pictures for each filter.
        img_width = 128
        img_height = 128

        # Let's start from a gray image with some noise
        if K.image_data_format() == 'channels_first':
            input_img_data = np.random.random((1, 3, img_width, img_height))
        else:
            input_img_data = np.random.random((1, img_width, img_height, 3))
        input_img_data = (input_img_data - 0.5) * 20 + 128

        imsave('gray_image.png', input_img_data[0,:,:,:])
        print(input_img_data.shape)

        # run gradient ascent for N steps
        N = 20
        step = 1. # step size for gradient ascent
        for i in range(N):
            loss_value, grads_value = iterate([input_img_data])
            input_img_data += grads_value * step

            print('Current loss value:', loss_value)
            if loss_value <= 0.:
                # some filter get stuck to 0, skip them
                break
```

```
(1, 128, 128, 3)
Current loss value: 0.0
```

```
/home/supannee/tensorflow/lib/python3.5/site-packages/ipykernel_launcher.py:15: DeprecationWarni
`imsave` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imwrite`` instead.
```

```
  from ipykernel import kernelapp as app
```

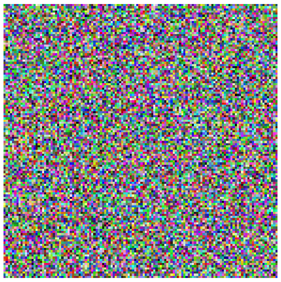
Now we can extract and display the generated input.

```
In [11]: # util function to convert a tensor into a valid image
         def deprocess_image(x):
             # normalise tensor: center at 0., and std at 0.1
             x -= x.mean()
             x /= (x.std() + small_value)
             x *= 0.1

             # clip to [0, 1]
             x += 0.5
             x = np.clip(x, 0, 1)

             # convert to RGB array
             x *= 255
             if K.image_data_format() == 'channels_first': # c,h,w
                 x = x.transpose((1, 2, 0)) # results in h,w,c
             x = np.clip(x, 0, 255).astype('uint8')
             return x
```

Time to run and get the results!

```
In [12]: img = input_img_data[0]
         img = deprocess_image(img)
         imsave('%s_filter_%d.png' % (layer_name, filter_index), img)
```

```
/home/supannee/tensorflow/lib/python3.5/site-packages/ipykernel_launcher.py:3: DeprecationWarnin
`imsave` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imwrite`` instead.
  This is separate from the ipykernel package so we can avoid doing imports until
```

Input image: 

Result: