Faiz Ganz (fag277) - Unit 10 Lab

Lab: Transfer Learning with a Pre-Trained Deep Neural Network

As we discussed earlier, state-of-the-art neural networks involve millions of parameters that are prohibitively difficult to train from scratch. In this lab, we will illustrate a powerful technique called *fine-tuning* where we start with a large pre-trained network and then re-train only the final layers to adapt to a new task. The method is also called *transfer learning* and can produce excellent results on very small datasets with very little computational time.

This lab is based partially on this excellent blog. In performing the lab, you will learn to:

- Build a custom image dataset
- Fine tune the final layers of an existing deep neural network for a new classification task.
- Load images with a `DataGenerator`.

The lab has two versions:

- *CPU version*: In this version, you use lower resolution images so that the lab can be performed on your laptop. The resulting accuracy is lower. The code will also take considerable time to execute.
- *GPU version*: This version uses higher resolution images but requires a GPU instance. See the notes on setting up a GPU instance on Google Cloud Platform. The GPU training is much faster (< 1 minute).

MS students must complete the GPU version of this lab.

Create a Dataset

In this example, we will try to develop a classifier that can discriminate between two classes: `cars` and `bicycles`. One could imagine this type of classifier would be useful in vehicle vision systems. The first task is to build a dataset.

TODO: Create training and test datasets with:

- 1000 training images of cars
- 1000 training images of bicylces
- 300 test images of cars
- 300 test images of bicylces
- The images don't need to be the same size. But, you can reduce the resolution if you need to save disk space.

The images should be organized in the following directory structure:

```
./train
    /car
       car_0000.jpg
       car_0001.jpg
       ...
       car_0999.jpg
    /bicycle
       bicycle_0000.jpg
       bicycle_0001.jpg
       ...
       bicycle_0999.jpg
./test
    /car
       car_1001.jpg
       car_1001.jpg
       ...
       car_1299.jpg
    /bicycle
       bicycle_1000.jpg
       bicycle_1001.jpg
       ...
       bicycle_1299.jpg
```

The naming of the files within the directories does not matter. The `ImageDataGenerator` class below will find the filenames. Just make sure there are the correct number of files in each directory.

A nice automated way of building such a dataset if through the FlickrAPI. Remember that if you run the FlickrAPI twice, it may collect the same images. So, you need to run it once and split the images into training and test directories.

Installing the flickrapi in colab:

```
!pip install flickrapi
```

```
Requirement already satisfied: flickrapi in /usr/local/lib/python3.7/dist-packages (2.4.0)
Requirement already satisfied: requests-toolbelt>=0.3.1 in /usr/local/lib/python3.7/dist-packages (from flickrapi) (0.9.1)
Requirement already satisfied: requests-oauthlib>=0.4.0 in /usr/local/lib/python3.7/dist-packages (from flickrapi) (1.3.0)
Requirement already satisfied: requests>=2.2.1 in /usr/local/lib/python3.7/dist-packages (from flickrapi) (2.23.0)
Requirement already satisfied: six>=1.5.2 in /usr/local/lib/python3.7/dist-packages (from flickrapi) (1.15.0)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-packages (from requests>=2.2.1->flickrapi) (2.10)
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.7/dist-packages (from requests>=2.2.1->flickrapi) (3.0.4)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.7/dist-packages (from requests>=2.2.1->flickrapi) (2021.
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /usr/local/lib/python3.7/dist-packages (from requests>=2.2
Requirement already satisfied: oauthlib>=3.0.0 in /usr/local/lib/python3.7/dist-packages (from requests-oauthlib>=0.4.0->flickrapi)
```

```
import flickrapi
import urllib.request
import matplotlib.pyplot as plt
import numpy as np
import skimage.io
import skimage.transform
import requests
from io import BytesIO
%matplotlib inline
```

Make sure to get a api key and secret code and change below:

```
# see the flickr demo on where to get keys.
api_key = u'de6b7baf2588069e362d012b7c1fe818'
api_secret = u'c9b15a34d9032be5'
flickr = flickrapi.FlickrAPI(api_key, api_secret)
```

Download bicycles

```
import warnings
import os
dir_name = 'train'
dir_exists = os.path.isdir(dir_name)
if not dir_exists:
    os.mkdir(dir_name)
    print("Making directory %s" % dir_name)
else:
    print("Will store images in directory %s" % dir_name)

dir_name = 'test'
dir_exists = os.path.isdir(dir_name)
if not dir_exists:
    os.mkdir(dir_name)
    print("Making directory %s" % dir_name)
else:
    print("Will store images in directory %s" % dir_name)

dir_folder = os.path.join(os.getcwd(),'train','bicycle')
dir_exists = os.path.isdir(dir_folder)
if not dir_exists:
    os.mkdir(dir_folder)
    print("Making directory %s" % dir_folder)
else:
    print("Will store images in directory %s" % dir_folder)

dir_folder = os.path.join(os.getcwd(),'train','car')
dir_exists = os.path.isdir(dir_folder)
```

```
if not dir_exists:
    os.mkdir(dir_folder)
    print("Making directory %s" % dir_folder)
else:
    print("Will store images in directory %s" % dir_folder)

dir_folder = os.path.join(os.getcwd(),'test','bicycle')
dir_exists = os.path.isdir(dir_folder)
if not dir_exists:
    os.mkdir(dir_folder)
    print("Making directory %s" % dir_folder)
else:
    print("Will store images in directory %s" % dir_folder)

dir_folder = os.path.join(os.getcwd(),'test','car')
dir_exists = os.path.isdir(dir_folder)
if not dir_exists:
    os.mkdir(dir_folder)
    print("Making directory %s" % dir_folder)
else:
    print("Will store images in directory %s" % dir_folder)
```

```
Will store images in directory train
Will store images in directory test
Will store images in directory /content/train/bicycle
Will store images in directory /content/train/car
Will store images in directory /content/test/bicycle
Will store images in directory /content/test/car
```

```
keyword = 'bicycle'
dir_name = 'bicycle'
photos = flickr.walk(text=keyword, tag_mode='all', tags=keyword,extras='url_c',\
                     sort='relevance',per_page=100)

nimage = 1300
i = 0
nrow = 224
ncol = 224
for photo in photos:
    url=photo.get('url_c')
    if not (url is None):

        # Create a file from the URL
        # This may only work in Python3
        response = requests.get(url)
        file = BytesIO(response.content)

        # Read image from file
        im = skimage.io.imread(file)

        # Resize images
        im1 = skimage.transform.resize(im,(nrow,ncol),mode='constant')

        # Convert to uint8, suppress the warning about the precision loss
        with warnings.catch_warnings():
            warnings.simplefilter("ignore")
            im2 = skimage.img_as_ubyte(im1)

        # Save the image
        if i< 1000:
          local_name = '{0:s}/{1:s}/{2:s}_{3:04d}.jpg'.format('train',dir_name,keyword, i)
        else:
          local_name = '{0:s}/{1:s}/{2:s}_{3:04d}.jpg'.format('test',dir_name,keyword, i)
        skimage.io.imsave(local_name, im2)
        if (i%100) == 1:
          print(local_name)
        i = i + 1
    if (i >= nimage):
        break
```

```
train/bicycle/bicycle_0001.jpg
train/bicycle/bicycle_0101.jpg
train/bicycle/bicycle_0201.jpg
train/bicycle/bicycle_0301.jpg
train/bicycle/bicycle_0401.jpg
train/bicycle/bicycle_0501.jpg
```

```
train/bicycle/bicycle_0601.jpg
train/bicycle/bicycle_0701.jpg
train/bicycle/bicycle_0801.jpg
train/bicycle/bicycle_0901.jpg
test/bicycle/bicycle_1001.jpg
test/bicycle/bicycle_1101.jpg
test/bicycle/bicycle_1201.jpg
```

Download cars

```
keyword = 'car'
dir_name = 'car'
photos = flickr.walk(text=keyword, tag_mode='all', tags=keyword,extras='url_c',\
                     sort='relevance',per_page=100)

nimage = 1300
i = 0
nrow = 224
ncol = 224
for photo in photos:
    url=photo.get('url_c')
    if not (url is None):

        # Create a file from the URL
        # This may only work in Python3
        response = requests.get(url)
        file = BytesIO(response.content)

        # Read image from file
        im = skimage.io.imread(file)

        # Resize images
        im1 = skimage.transform.resize(im,(nrow,ncol),mode='constant')

        # Convert to uint8, suppress the warning about the precision loss
        with warnings.catch_warnings():
            warnings.simplefilter("ignore")
            im2 = skimage.img_as_ubyte(im1)

        # Save the image
        if i< 1000:
          local_name = '{0:s}/{1:s}/{2:s}_{3:04d}.jpg'.format('train',dir_name,keyword, i)
        else:
          local_name = '{0:s}/{1:s}/{2:s}_{3:04d}.jpg'.format('test',dir_name,keyword, i)
        skimage.io.imsave(local_name, im2)
        if (i%100) == 1:
          print(local_name)
        i = i + 1
    if (i >= nimage):
        break
```

```
train/car/car_0001.jpg
train/car/car_0101.jpg
train/car/car_0201.jpg
train/car/car_0301.jpg
train/car/car_0401.jpg
train/car/car_0501.jpg
train/car/car_0601.jpg
train/car/car_0701.jpg
train/car/car_0801.jpg
train/car/car_0901.jpg
test/car/car_1001.jpg
test/car/car_1101.jpg
test/car/car_1201.jpg
```

Loading a Pre-Trained Deep Network

We follow the VGG16 demo to load a pre-trained deep VGG16 network. First, run a command to verify your instance is connected to a GPU.

```
# TODO 1:
import tensorflow as tf

tf.test.is_gpu_available()
```

```
WARNING:tensorflow:From <ipython-input-7-b90ba5067c59>:4: is_gpu_available (from tensorflow.python.framework.test_util) is deprecate
Instructions for updating:
Use `tf.config.list_physical_devices('GPU')` instead.
False
```

Now load the appropriate tensorflow packages.

```
from tensorflow.keras import applications
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras import optimizers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dropout, Flatten, Dense
```

We also load some standard packages.

```
import numpy as np
import matplotlib.pyplot as plt
```

Clear the Keras session.

```
# TODO 2:
import tensorflow.keras.backend as K
K.clear_session()
```

Set the dimensions of the input image. The sizes below would work on a GPU machine. But, if you have a CPU image, you can use a smaller image size, like `64 x 64`.

```
# TODO 3:  Set to smaller values if you are using a CPU.
# Otherwise, do not change this code.
nrow = 64
ncol = 64
```

Now we follow the [VGG16 demo]() and load the deep VGG16 network. Alternatively, you can use any other pre-trained model in keras. When using the `applications.VGG16` method you will need to:

- Set `include_top=False` to not include the top layer
- Set the `image_shape` based on the above dimensions. Remember, `image_shape` should be `height x width x 3` since the images are color.

```
# TODO 4:  Load the VGG16 network
from tensorflow.keras.applications.vgg16 import VGG16

input_shape = (nrow, ncol, 3)
base_model = applications.VGG16(weights='imagenet', include_top=False, input_shape=input_shape)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weights_tf_dim_ordering_tf_kernels_no
58892288/58889256 [==============================] - 0s 0us/step
58900480/58889256 [==============================] - 0s 0us/step
```

To create now new model, we create a Sequential model. Then, loop over the layers in `base_model.layers` and add each layer to the new model.

```
# TODO 5:  Loop over base_model.layers and add each layer to model
from tensorflow.keras.models import Model, Sequential

# Create a new model
model = Sequential()

for layer in base_model.layers:
  model.add(layer)
```

Next, loop through the layers in `model`, and freeze each layer by setting `layer.trainable = False`. This way, you will not have to *re-train* any of the existing layers.

```
# TODO 6
for layer in model.layers:
  layer.trainable = False
```

Now, add the following layers to `model`:

- A `Flatten()` layer which reshapes the outputs to a single channel.
- A fully-connected layer with 256 output units and `relu` activation
- A `Dropout(0.5)` layer.
- A final fully-connected layer. Since this is a binary classification, there should be one output and `sigmoid` activation.

```
# TODO 7
model.add(Flatten())
model.add(Dense(units=256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(units=1, activation='sigmoid'))
```

Print the model summary. This will display the number of trainable parameters vs. the non-trainable parameters.

```
# TODO 8
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 block1_conv1 (Conv2D)       (None, 64, 64, 64)        1792      
                                                                 
 block1_conv2 (Conv2D)       (None, 64, 64, 64)        36928     
                                                                 
 block1_pool (MaxPooling2D)  (None, 32, 32, 64)        0         
                                                                 
 block2_conv1 (Conv2D)       (None, 32, 32, 128)       73856     
                                                                 
 block2_conv2 (Conv2D)       (None, 32, 32, 128)       147584    
                                                                 
 block2_pool (MaxPooling2D)  (None, 16, 16, 128)       0         
                                                                 
 block3_conv1 (Conv2D)       (None, 16, 16, 256)       295168    
                                                                 
 block3_conv2 (Conv2D)       (None, 16, 16, 256)       590080    
                                                                 
 block3_conv3 (Conv2D)       (None, 16, 16, 256)       590080    
                                                                 
 block3_pool (MaxPooling2D)  (None, 8, 8, 256)         0         
                                                                 
 block4_conv1 (Conv2D)       (None, 8, 8, 512)         1180160   
                                                                 
 block4_conv2 (Conv2D)       (None, 8, 8, 512)         2359808   
                                                                 
 block4_conv3 (Conv2D)       (None, 8, 8, 512)         2359808   
                                                                 
 block4_pool (MaxPooling2D)  (None, 4, 4, 512)         0         
                                                                 
 block5_conv1 (Conv2D)       (None, 4, 4, 512)         2359808   
                                                                 
 block5_conv2 (Conv2D)       (None, 4, 4, 512)         2359808   
                                                                 
 block5_conv3 (Conv2D)       (None, 4, 4, 512)         2359808   
                                                                 
 block5_pool (MaxPooling2D)  (None, 2, 2, 512)         0         
                                                                 
 flatten (Flatten)           (None, 2048)              0         
                                                                 
 dense (Dense)               (None, 256)               524544    
                                                                 
 dropout (Dropout)           (None, 256)               0         
                                                                 
 dense_1 (Dense)             (None, 1)                 257       
                                                                 
=================================================================
Total params: 15,239,489
Trainable params: 524,801
Non-trainable params: 14,714,688
_________________________________________________________________
```

Using Generators to Load Data

Up to now, the training data has been represented in a large matrix. This is not possible for image data when the datasets are very large. For these applications, the `keras` package provides a `ImageDataGenerator` class that can fetch images on the fly from a directory of images. Using multi-threading, training can be performed on one mini-batch while the image reader can read files for the next mini-batch. The code below creates an `ImageDataGenerator` for the training data. In addition to the reading the files, the `ImageDataGenerator` creates random deformations of the image to expand the total dataset size. When the training data is limited, using data augmentation is very important.

```
train_data_dir = './train'
batch_size = 32
train_datagen = ImageDataGenerator(rescale=1./255,
                                   shear_range=0.2,
                                   zoom_range=0.2,
                                   horizontal_flip=True)
train_generator = train_datagen.flow_from_directory(
                        train_data_dir,
                        target_size=(nrow,ncol),
                        batch_size=batch_size,
                        class_mode='binary')
```

```
Found 2000 images belonging to 2 classes.
```

Now, create a similar `test_generator` for the test data.

```
# TODO 9
test_data_dir = './test'

test_generator = train_datagen.flow_from_directory(
                        test_data_dir,
                        target_size=(nrow,ncol),
                        batch_size=batch_size,
                        class_mode='binary')
```

```
Found 600 images belonging to 2 classes.
```

The following function displays images that will be useful below.

```
# Display the image
def disp_image(im):
    if (len(im.shape) == 2):
        # Gray scale image
        plt.imshow(im, cmap='gray')
    else:
        # Color image.
        im1 = (im-np.min(im))/(np.max(im)-np.min(im))*255
        im1 = im1.astype(np.uint8)
        plt.imshow(im1)

    # Remove axis ticks
    plt.xticks([])
    plt.yticks([])
```

To see how the `train_generator` works, use the `train_generator.next()` method to get a minibatch of data `X,y`. Display the first 8 images in this mini-batch and label the image with the class label. You should see that bicycles have `y=0` and cars have `y=1`.

```
# TODO 10
X_im, y_im = next(train_generator)

plt.figure(figsize=(20,20))
nplot = 8
for i in range(nplot):
  plt.subplot(1,nplot,i+1)
  plt.xlabel(int(y_im[i]))
  disp_image(X_im[i,:,:,:])
```

Train the Model

Compile the model. Select the correct `loss` function, `optimizer` and `metrics`. Remember that we are performing binary classification.

```
# TODO 11
model.compile(optimizer='Adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

When using an `ImageDataGenerator`, we have to set two parameters manually:

- `steps_per_epoch = training data size // batch_size`
- `validation_steps = test data size // batch_size`

We can obtain the training and test data size from `train_generator.n` and `test_generator.n`, respectively.

```
# TODO 12
steps_per_epoch = train_generator.n // batch_size
validation_steps = test_generator.n // batch_size
```

Now, we run the fit. If you are using a CPU on a regular laptop, each epoch will take about 3-4 minutes, so you should be able to finish 5 epochs or so within 20 minutes. On a reasonable GPU, even with the larger images, it will take about 10 seconds per epoch.

- If you use `(nrow,ncol) = (64,64)` images, you should get around 90% accuracy after 5 epochs.
- If you use `(nrow,ncol) = (150,150)` images, you should get around 96% accuracy after 5 epochs. But, this will need a GPU.

You will get full credit for either version. With more epochs, you may get slightly higher, but you will have to play with the damping.

Remember to record the history of the fit, so that you can plot the training and validation accuracy curve.

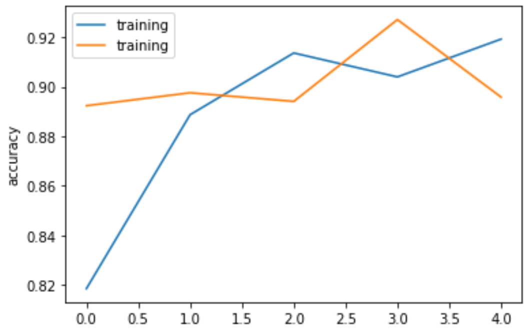
```
nepochs = 5  # Number of epochs

# Call the fit_generator function
hist = model.fit(
    train_generator,
    steps_per_epoch=steps_per_epoch,
    epochs=nepochs,
    validation_data=test_generator,
    validation_steps=validation_steps)
```

```
Epoch 1/5
62/62 [==============================] - 99s 2s/step - loss: 0.3941 - accuracy: 0.8186 - val_loss: 0.2351 - val_accuracy: 0.8924
Epoch 2/5
62/62 [==============================] - 98s 2s/step - loss: 0.2744 - accuracy: 0.8887 - val_loss: 0.2427 - val_accuracy: 0.8976
Epoch 3/5
62/62 [==============================] - 97s 2s/step - loss: 0.2161 - accuracy: 0.9136 - val_loss: 0.2696 - val_accuracy: 0.8941
Epoch 4/5
62/62 [==============================] - 97s 2s/step - loss: 0.2317 - accuracy: 0.9040 - val_loss: 0.2015 - val_accuracy: 0.9271
Epoch 5/5
62/62 [==============================] - 97s 2s/step - loss: 0.2063 - accuracy: 0.9192 - val_loss: 0.2191 - val_accuracy: 0.8958
```

```
# Plot the training accuracy and validation accuracy curves on the same figure.

# TODO 13
plt.plot(hist.history['accuracy'], label='training')
plt.plot(hist.history['val_accuracy'], label='training')
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.legend()
plt.show()
```



Plotting the Error Images

Now try to plot some images that were in error:

- Generate a mini-batch `Xts,yts` from the `test_generator.next()` method
- Get the class probabilities using the `model.predict( )` method and compute predicted labels `yhat`.
- Get the images where `yts[i] ~= yhat[i]`.
- If you did not get any prediction error in one minibatch, run it multiple times.
- After you a get a few error images (say 4-8), plot the error images with the true labels and class probabilities predicted by the classifie

```
# TODO 14
Xts,yts = next(test_generator)

yhat = model.predict(Xts)

loc_err = (np.round(yhat).ravel() == yts)
n_err = (len(loc_err) - np.sum(loc_err))
```

```
plt.figure(figsize=(20,20))
j = 0
for i, err in enumerate(loc_err):
  if err == False:
    plt.subplot(1, n_err, j + 1)
    plt.xlabel('True='+str(int(yts[i]))+', Prob='+str(yhat[i]))
    disp_image(Xts[i,:,:,:])
    j = j + 1
```