lab_fine_tune_partial

November 28, 2022

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.

0.1 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.

[ ]:
```python
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

api_key = u'07abaed9e5830f832f294cb75da267f3'
api_secret = u'36353dcc1378cdc5'
flickr = flickrapi.FlickrAPI(api_key, api_secret)
```

[ ]:
```python
import os
def create_dir(dir_name):
    dir_exists = os.path.isdir(dir_name)
    if not dir_exists:
        os.mkdir(dir_name)
        print("Making directory %s" % dir_name)
    else:
        print("Directory %s already exist" % dir_name)
```

[ ]:
```python
import warnings
import os
def create_photos(keyword, nimage, dir_name):
```

```
    photos = flickr.walk(text=keyword, tag_mode='all',␣
↪tags=keyword,extras='url_c',\
                     sort='relevance',per_page=100)
    i = 0
    exist = 0
    nrow = 224
    ncol = 224
    for photo in photos:

        local_name = '{0:s}/{1:s}_{2:04d}.jpg'.format(dir_name,keyword, i)
        if os.path.exists(local_name):
            exist += 1
            i += 1
            if (i >= nimage):
                break
            continue

        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
            local_name = '{0:s}/{1:s}_{2:04d}.jpg'.format(dir_name,keyword, i)
            skimage.io.imsave(local_name, im2)
            print(local_name)
            i = i + 1
        if (i >= nimage):
            break
    print('{} images already exist'.format(exist))
```

[ ]:
```
create_dir('train')
create_dir('test')
```

```
create_dir('train/car')
create_dir('train/bicycle')
create_dir('test/car')
create_dir('test/bicycle')

train_num = 1000
test_num = 300
create_photos('car', train_num, 'train/car')
create_photos('bicycle', train_num, 'train/bicycle')
create_photos('car', test_num, 'test/car')
create_photos('bicycle', test_num, 'test/bicycle')
```

```
Directory train already exist
Directory test already exist
Directory train/car already exist
Directory train/bicycle already exist
Directory test/car already exist
Directory test/bicycle already exist
1000 images already exist
1000 images already exist
300 images already exist
300 images already exist
```

0.2 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.config.list_physical_devices('GPU')
```

[ ]:
```
[PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
```

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 = 150
ncol = 150
```

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

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

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.

[ ]:
```
# Create a new model
model = Sequential()

# TODO 5:  Loop over base_model.layers and add each layer to model

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(256, activation='relu'))
```

```python
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))
```

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

[ ]:
```python
# TODO 8
model.summary()
```

```
Model: "sequential"

_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 block1_conv1 (Conv2D)       (None, 150, 150, 64)      1792

 block1_conv2 (Conv2D)       (None, 150, 150, 64)      36928

 block1_pool (MaxPooling2D)  (None, 75, 75, 64)        0

 block2_conv1 (Conv2D)       (None, 75, 75, 128)       73856

 block2_conv2 (Conv2D)       (None, 75, 75, 128)       147584

 block2_pool (MaxPooling2D)  (None, 37, 37, 128)       0

 block3_conv1 (Conv2D)       (None, 37, 37, 256)       295168

 block3_conv2 (Conv2D)       (None, 37, 37, 256)       590080

 block3_conv3 (Conv2D)       (None, 37, 37, 256)       590080

 block3_pool (MaxPooling2D)  (None, 18, 18, 256)       0

 block4_conv1 (Conv2D)       (None, 18, 18, 512)       1180160

 block4_conv2 (Conv2D)       (None, 18, 18, 512)       2359808

 block4_conv3 (Conv2D)       (None, 18, 18, 512)       2359808

 block4_pool (MaxPooling2D)  (None, 9, 9, 512)         0

 block5_conv1 (Conv2D)       (None, 9, 9, 512)         2359808

 block5_conv2 (Conv2D)       (None, 9, 9, 512)         2359808

 block5_conv3 (Conv2D)       (None, 9, 9, 512)         2359808

 block5_pool (MaxPooling2D)  (None, 4, 4, 512)         0
```

```
 flatten (Flatten)           (None, 8192)              0

 dense (Dense)               (None, 256)               2097408

 dropout (Dropout)           (None, 256)               0

 dense_1 (Dense)             (None, 1)                 257

=================================================================
Total params: 16,812,353
Trainable params: 2,097,665
Non-trainable params: 14,714,688
_________________________________________________________________
```

0.3 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.

[ ]:
```
from google.colab import drive
import os
drive.mount('/content/gdrive/')
os.chdir('/content/gdrive/MyDrive/Colab_Notebooks/unit10_cnn')
```

```
Drive already mounted at /content/gdrive/; to attempt to forcibly remount, call
drive.mount("/content/gdrive/", force_remount=True).
```

[ ]:
```
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'
batch_size = 32
test_datagen = ImageDataGenerator(rescale=1./255,
                                   shear_range=0.2,
                                   zoom_range=0.2,
                                   horizontal_flip=True)
test_generator = test_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 mini-batch 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
plt.figure(figsize=(10,10))
nplot = 8
X_batch, y_batch = train_generator.next()
for i in range(nplot):
    plt.subplot(1,nplot,i+1)
    disp_image(X_batch[i,:,:,:])
    plt.xlabel(y_batch[i])
```

0.4 Train the Model

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

[ ]:
```
# TODO 11
from tensorflow.keras.optimizers import Adam
opt = Adam(lr=1e-3)
model.compile(optimizer=opt,
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

```
/usr/local/lib/python3.7/dist-
packages/keras/optimizers/optimizer_v2/adam.py:110: UserWarning: The `lr`
argument is deprecated, use `learning_rate` instead.
  super(Adam, self).__init__(name, **kwargs)
```

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.

[ ]:
```python
nepochs = 5  # Number of epochs

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

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:10: UserWarning:
`Model.fit_generator` is deprecated and will be removed in a future version.
Please use `Model.fit`, which supports generators.
  # Remove the CWD from sys.path while we load stuff.
```

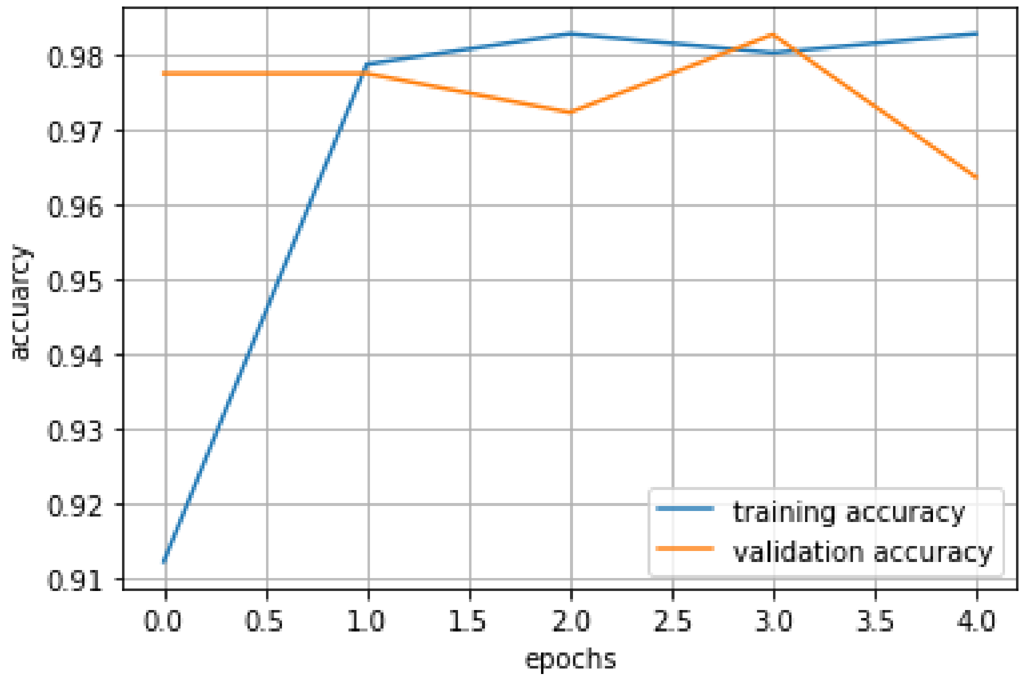
```
Epoch 1/5
62/62 [==============================] - 139s 2s/step - loss: 0.2302 - accuracy:
0.9121 - val_loss: 0.0618 - val_accuracy: 0.9774
Epoch 2/5
62/62 [==============================] - 19s 306ms/step - loss: 0.0643 -
accuracy: 0.9787 - val_loss: 0.0604 - val_accuracy: 0.9774
Epoch 3/5
62/62 [==============================] - 18s 292ms/step - loss: 0.0446 -
accuracy: 0.9827 - val_loss: 0.0644 - val_accuracy: 0.9722
Epoch 4/5
62/62 [==============================] - 18s 295ms/step - loss: 0.0540 -
accuracy: 0.9802 - val_loss: 0.0483 - val_accuracy: 0.9826
Epoch 5/5
62/62 [==============================] - 19s 308ms/step - loss: 0.0395 -
accuracy: 0.9827 - val_loss: 0.1051 - val_accuracy: 0.9635
```

[ ]:
```python
# Plot the training accuracy and validation accuracy curves on the same figure.

# TODO 13
tr_accuracy = hist.history['accuracy']
val_accuracy = hist.history['val_accuracy']

plt.plot(tr_accuracy)
plt.plot(val_accuracy)
plt.grid()
plt.xlabel('epochs')
plt.ylabel('accuarcy')
plt.legend(['training accuracy', 'validation accuracy'])
```

[ ]:
```
<matplotlib.legend.Legend at 0x7f073c86da10>
```



0.5 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_batch, yts_batch = test_generator.next()
ypred_batch = model.predict(Xts_batch)
ypred_batch = np.squeeze(ypred_batch)
yhat_batch = np.round(ypred_batch)
acc = np.mean(yhat_batch == yts_batch)
print(acc)

error_mat = yhat_batch != yts_batch
error_index = np.where(error_mat)[0]

nplot = len(error_index)
plt.figure(figsize=(20,20))
```

```python
for i in range(nplot):
    error = error_index[i]
    plt.subplot(1,nplot,i+1)
    disp_image(Xts_batch[error,:,:,:])
    plt.xlabel(ypred_batch[error])
```

```
1/1 [==============================] - 0s 20ms/step
0.90625
```

[ ]: