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 (https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html)](https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html). 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 (../GCP/getting_started.md)](../GCP/getting_started.md) 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 (demo2_flickr_images.ipynb)](demo2_flickr_images.ipynb). 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.

Loading a Pre-Trained Deep Network

We follow the [VGG16 demo (./demo3_vgg16.ipynb)](./demo3_vgg16.ipynb) to load a pre-trained deep VGG16 network. First, run a command to verify your instance is connected to a GPU.

In [1]:

```
import torch
from torch import nn, optim
from torch.utils.data import Dataset, DataLoader

import torchvision
from torchvision import datasets, transforms
from torchvision import transforms, utils
```

In [2]:

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(torch.cuda.get_device_name(0))
print(device)
```

```
GeForce GTX 1050 Ti with Max-Q Design
cuda:0
```

We also load some standard packages.

In [3]:

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

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` .

In [4]:

```
# TODO:  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 (./vgg16.ipynb)](./vgg16.ipynb) 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.

In [5]:

```
# TODO:  Load the VGG16 network
# input_shape = ...
# base_model = applications.VGG16(weights='imagenet', ...)
input_shape = (nrow, ncol, 3)
base_model = torchvision.models.vgg16(pretrained=True,)
```

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.

In [6]:

```
# Create a new model
# model = Sequential()

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

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.

In [7]:

```
# TODO
for param in base_model.parameters():
    param.requires_grad = 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.

In [8]:

```
# TODO
# model.add(...)
# model.add(...)
# ....
num_ftrs = base_model.classifier[0].in_features

classifier = nn.Sequential(
    nn.Linear(num_ftrs, 256), nn.ReLU(), nn.Dropout(), nn.Linear(256, 1),
    nn.Sigmoid())
base_model.classifier = classifier
model = base_model.to(device)
```

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

In [9]:

```python
# TODO
print(model)
```

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=256, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.5)
    (3): Linear(in_features=256, out_features=1, bias=True)
    (4): Sigmoid()
  )
)
```

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.

In [10]:

```
train_data_dir = './train'
batch_size = 32

transform = transforms.Compose([
    transforms.Resize((nrow, ncol)),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])
])
trainset = datasets.ImageFolder(train_data_dir, transform=transform)
trainloader = DataLoader(
    trainset, batch_size=batch_size, shuffle=True, num_workers=2)
```

Now, create a similar test_generator for the test data.

In [11]:

```
# TODO
# test_generator = ...
test_data_dir = './test'
batch_size = 32

testset = datasets.ImageFolder(test_data_dir, transform=transform)
testloader = DataLoader(
    testset, batch_size=batch_size, shuffle=True, num_workers=2)
```

The following function displays images that will be useful below.

In [12]:

```
# Display the image
def disp_image(inp, title=None):
    """Imshow for Tensor."""
    inp = inp.numpy().transpose((1, 2, 0))
    mean = np.array([0.5, 0.5, 0.5])
    std = np.array([0.5, 0.5, 0.5])
    inp = std * inp + mean
    inp = np.clip(inp, 0, 1)
    plt.imshow(inp)
    if title is not None:
        plt.title(title)
    plt.pause(0.001)
```

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 .

In [13]:

```
# TODO
class_names = ['bicyle', 'car']
inputs, classes = next(iter(trainloader))
# print(inputs.shape)
# Make a grid from batch
# disp_image(inputs)
out = torchvision.utils.make_grid(inputs[:8])
plt.figure(figsize=(12, 5))
plt.axis('off')
disp_image(out, title=[class_names[x] for x in classes[:8]])
```

Train the Model

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

In [14]:

```
# TODO.
# model.compile(...)
criterion = nn.BCELoss()

optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
```

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.

In [15]:

```
# TODO
```

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.

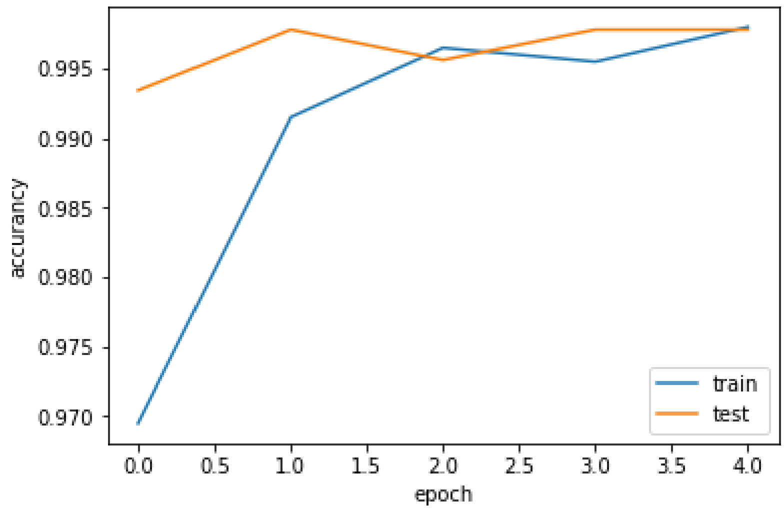
In [16]:

```
nepochs = 5  # Number of epochs

train_acc = []
test_acc = []

for epoch in range(nepochs):

    print('Epoch {}/{}'.format(epoch, nepochs - 1))
    print('-' * 10)
    running_loss = 0
    running_corrects = 0
    validation_loss = 0
    validation_corrects = 0

    model.train()

    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)

        optimizer.zero_grad()

        outputs = model(inputs)

        loss = criterion(outputs.view(-1), labels.float())

        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        running_corrects += torch.sum(outputs.round().long().view(-1) == labels).item()

    # validation
    model.eval()
    with torch.no_grad():
        for j, data in enumerate(testloader, 0):

            inputs, labels = data
            inputs, labels = inputs.to(device), labels.to(device)

            outputs = model(inputs)

            loss = criterion(outputs.view(-1), labels.float())

            # print statistics
            validation_loss += loss.item()
            validation_corrects += torch.sum(outputs.round().long().view(-1) == labels).item()

    epoch_loss = running_loss / len(trainset)
    epoch_acc = running_corrects / len(trainset)

    val_epoch_loss = validation_loss / len(testset)
    val_epoch_acc = validation_corrects / len(testset)

    train_acc.append(epoch_acc)
    test_acc.append(val_epoch_acc)

    print(f"train Loss: {epoch_loss:.4f} Acc: {epoch_acc:4f}\n\
val Loss: {val_epoch_loss:.4f} Acc: {val_epoch_acc:4f}")
```

```
Epoch 0/4
----------
train Loss: 0.0033 Acc: 0.969500
val Loss: 0.0006 Acc: 0.993435
Epoch 1/4
----------
train Loss: 0.0009 Acc: 0.991500
val Loss: 0.0004 Acc: 0.997812
Epoch 2/4
----------
train Loss: 0.0006 Acc: 0.996500
val Loss: 0.0004 Acc: 0.995624
Epoch 3/4
----------
train Loss: 0.0005 Acc: 0.995500
val Loss: 0.0003 Acc: 0.997812
Epoch 4/4
----------
train Loss: 0.0003 Acc: 0.998000
val Loss: 0.0002 Acc: 0.997812
```

In [17]:

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

# TO DO
plt.plot(train_acc)
plt.plot(test_acc)
plt.xlabel("epoch")
plt.ylabel("accurancy")
plt.legend(["train", "test"])
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

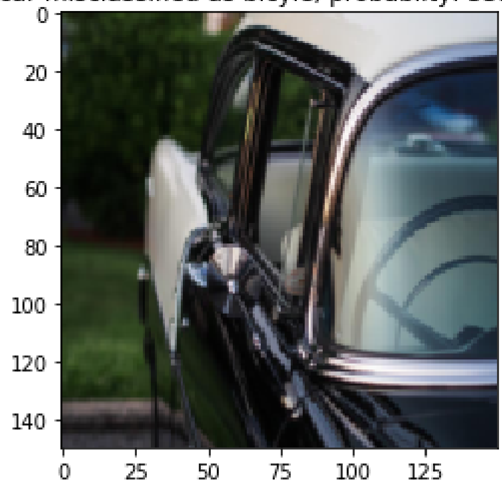
In [18]:

```
# TO DO
model.eval()
error_images = []
error_labels = []
prob = []
with torch.no_grad():
    for data in testloader:
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)

        outputs = model(inputs)
        outputs = outputs.detach().view(-1)

        error_l = outputs.round().long() != labels

        error_images.extend(list(inputs[error_l]))
        error_labels.extend(list(labels[error_l]))
        prob.extend(list(outputs[error_l]))
        if len(error_images) > 4:
            break
```

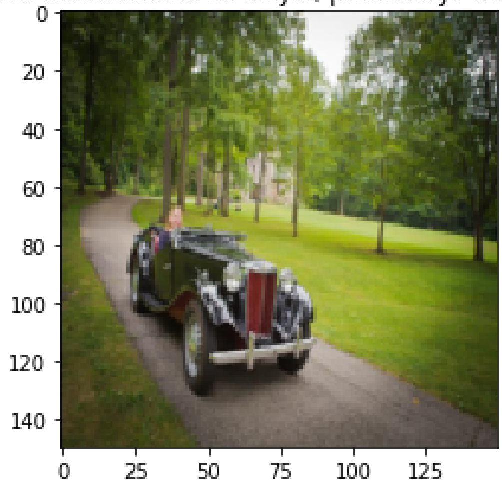
In [20]:

```python
for i, j, p in zip(error_images, error_labels, prob):
    disp_image(
        i.cpu(),
        title=
        f"{class_names[j]} misclassified as {class_names[1-j]}, probablity: {(1 - j.item() + (2*j.
    )
```

car misclassified as bicyle, probablity: 33.84%



car misclassified as bicyle, probablity: 42.21%



In [ ]: