# Convolutional Neural Networks

## Project: Write an Algorithm for a Dog Identification App

In this notebook, some template code has already been provided for you, and you will need to implement additional functionality to successfully complete this project. You will not need to modify the included code beyond what is requested. Sections that begin with **'(IMPLEMENTATION)'** in the header indicate that the following block of code will require additional functionality which you must provide. Instructions will be provided for each section, and the specifics of the implementation are marked in the code block with a 'TODO' statement. Please be sure to read the instructions carefully!

> **Note**: Once you have completed all of the code implementations, you need to finalize your work by exporting the Jupyter Notebook as an HTML document. Before exporting the notebook to html, all of the code cells need to have been run so that reviewers can see the final implementation and output. You can then export the notebook by using the menu above and navigating to **File -> Download as -> HTML (.html)**. Include the finished document along with this notebook as your submission.

In addition to implementing code, there will be questions that you must answer which relate to the project and your implementation. Each section where you will answer a question is preceded by a **'Question X'** header. Carefully read each question and provide thorough answers in the following text boxes that begin with **'Answer:'**. Your project submission will be evaluated based on your answers to each of the questions and the implementation you provide.

> **Note:** Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. Markdown cells can be edited by double-clicking the cell to enter edit mode.

The rubric contains *optional* "Stand Out Suggestions" for enhancing the project beyond the minimum requirements. If you decide to pursue the "Stand Out Suggestions", you should include the code in this Jupyter notebook.

### Why We're Here

In this notebook, you will make the first steps towards developing an algorithm that could be used as part of a mobile or web app. At the end of this project, your code will accept any user-supplied image as input. If a dog is detected in the image, it will provide an estimate of the dog's breed. If a human is detected, it will provide an estimate of the dog breed that is most resembling. The image below displays potential sample output of your finished project (... but we expect that each student's algorithm will behave differently!).

Sample Dog Output

In this real-world setting, you will need to piece together a series of models to perform different tasks; for instance, the algorithm that detects humans in an image will be different from the CNN that infers dog breed. There are many points of possible failure, and no perfect algorithm exists. Your imperfect solution will nonetheless create a fun user experience!

## The Road Ahead

We break the notebook into separate steps. Feel free to use the links below to navigate the notebook.

---

## Step 0: Import Datasets

Make sure that you've downloaded the required human and dog datasets:

- Download the dog dataset (https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/dogImages.zip). Unzip the folder and place it in this project's home directory, at the location `/dogImages` .
- Download the human dataset (https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/lfw.zip). Unzip the folder and place it in the home directory, at location `/lfw` .

*Note: If you are using a Windows machine, you are encouraged to use 7zip (http://www.7-zip.org/) to extract the folder.*

In the code cell below, we save the file paths for both the human (LFW) dataset and dog dataset in the numpy arrays `human_files` and `dog_files` .

In [1]:
```python
import numpy as np
from glob import glob

# load filenames for human and dog images
human_files = np.array(glob("lfw/*/*"))
dog_files = np.array(glob("dogImages/*/*/*"))

# print number of images in each dataset
print('There are %d total human images.' % len(human_files))
print('There are %d total dog images.' % len(dog_files))
```

```
There are 13233 total human images.
There are 8351 total dog images.
```

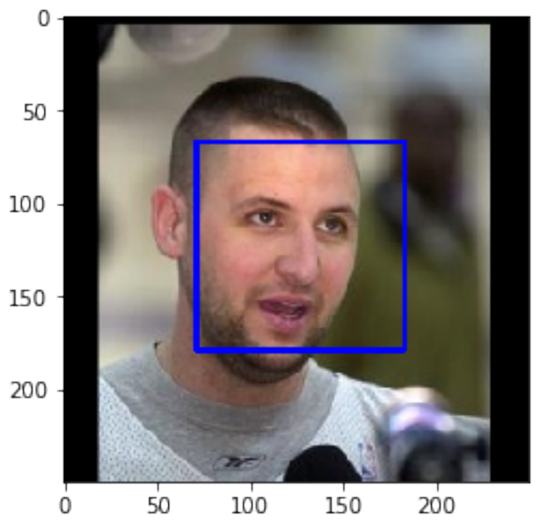
## Step 1: Detect Humans

In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers (http://docs.opencv.org/trunk/d7/d8b/tutorial_py_face_detection.html) to detect human faces in images.

OpenCV provides many pre-trained face detectors, stored as XML files on github (https://github.com/opencv/opencv/tree/master/data/haarcascades). We have downloaded one of these detectors and stored it in the `haarcascades` directory. In the next code cell, we demonstrate how to use this detector to find human faces in a sample image.

In [2]:
```
import cv2
import matplotlib.pyplot as plt
%matplotlib inline

# extract pre-trained face detector
face_cascade = cv2.CascadeClassifier('haarcascades/haarcascade_fron
talface_alt.xml')

# load color (BGR) image
img = cv2.imread(human_files[0])
# convert BGR image to grayscale
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# find faces in image
faces = face_cascade.detectMultiScale(gray)

# print number of faces detected in the image
print('Number of faces detected:', len(faces))

# get bounding box for each detected face
for (x,y,w,h) in faces:
    # add bounding box to color image
    cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)

# convert BGR image to RGB for plotting
cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

# display the image, along with bounding box
plt.imshow(cv_rgb)
plt.show()
```

```
Number of faces detected: 1
```



Before using any of the face detectors, it is standard procedure to convert the images to grayscale. The `detectMultiScale` function executes the classifier stored in `face_cascade` and takes the grayscale image as a parameter.

In the above code, `faces` is a numpy array of detected faces, where each row corresponds to a detected face. Each detected face is a 1D array with four entries that specifies the bounding box of the detected face. The first two entries in the array (extracted in the above code as `x` and `y`) specify the horizontal and vertical positions of the top left corner of the bounding box. The last two entries in the array (extracted here as `w` and `h`) specify the width and height of the box.

## Write a Human Face Detector

We can use this procedure to write a function that returns `True` if a human face is detected in an image and `False` otherwise. This function, aptly named `face_detector`, takes a string-valued file path to an image as input and appears in the code block below.

In [3]:
```
# returns "True" if face is detected in image stored at img_path
def face_detector(img_path):
    img = cv2.imread(img_path)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    faces = face_cascade.detectMultiScale(gray)
    return len(faces) > 0
```

## (IMPLEMENTATION) Assess the Human Face Detector

**Question 1:** Use the code cell below to test the performance of the `face_detector` function.

- What percentage of the first 100 images in `human_files` have a detected human face?
- What percentage of the first 100 images in `dog_files` have a detected human face?

Ideally, we would like 100% of human images with a detected face and 0% of dog images with a detected face. You will see that our algorithm falls short of this goal, but still gives acceptable performance. We extract the file paths for the first 100 images from each of the datasets and store them in the numpy arrays `human_files_short` and `dog_files_short`.

**Answer:** (You can print out your results and/or write your percentages in this cell)

In [4]:
```
from tqdm import tqdm

human_files_short = human_files[:100]
dog_files_short = dog_files[:100]

#-#-# Do NOT modify the code above this line. #-#-#

## TODO: Test the performance of the face_detector algorithm
## on the images in human_files_short and dog_files_short.

def test_face_detection(files):
    detection_cnt = 0;
    total_cnt = len(files)
    for file in files:
        detection_cnt += face_detector(file)
    return detection_cnt, total_cnt

print("detect face in human_files: {} / {}".format(test_face_detect
ion(human_files_short)[0], test_face_detection(human_files_short)[1
]))
print("detect face in dog_files: {} / {}".format(test_face_detectio
n(dog_files_short)[0], test_face_detection(dog_files_short)[1]))
```

```
detect face in human_files: 100 / 100
detect face in dog_files: 9 / 100
```

We suggest the face detector from OpenCV as a potential way to detect human images in your algorithm, but you are free to explore other approaches, especially approaches that make use of deep learning :). Please use the code cell below to design and test your own face detection algorithm. If you decide to pursue this *optional* task, report performance on `human_files_short` and `dog_files_short` .

In [5]:
```
### (Optional)
### TODO: Test performance of another face detection algorithm.
### Feel free to use as many code cells as needed.
# Install a conda package in the current Jupyter kernel
```

---

## Step 2: Detect Dogs

In this section, we use a [pre-trained model (http://pytorch.org/docs/master/torchvision/models.html)](http://pytorch.org/docs/master/torchvision/models.html) to detect dogs in images.

### Obtain Pre-trained VGG-16 Model

The code cell below downloads the VGG-16 model, along with weights that have been trained on [ImageNet (http://www.image-net.org/)](http://www.image-net.org/), a very large, very popular dataset used for image classification and other vision tasks. ImageNet contains over 10 million URLs, each linking to an image containing an object from one of [1000 categories (https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a)](https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a).

In [6]:
```python
import torch
import torchvision.models as models

# define VGG16 model
VGG16 = models.vgg16(pretrained=True)

# check if CUDA is available
use_cuda = torch.cuda.is_available()

# move model to GPU if CUDA is available
if use_cuda:
    VGG16 = VGG16.cuda()
```

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.p
th" to /home/ec2-user/.cache/torch/checkpoints/vgg16-397923af.pth
```

Given an image, this pre-trained VGG-16 model returns a prediction (derived from the 1000 possible categories in ImageNet) for the object that is contained in the image.

## (IMPLEMENTATION) Making Predictions with a Pre-trained Model

In the next code cell, you will write a function that accepts a path to an image (such as `'dogImages/train/001.Affenpinscher/Affenpinscher_00001.jpg'`) as input and returns the index corresponding to the ImageNet class that is predicted by the pre-trained VGG-16 model. The output should always be an integer between 0 and 999, inclusive.

Before writing the function, make sure that you take the time to learn how to appropriately pre-process tensors for pre-trained models in the PyTorch documentation (http://pytorch.org/docs/stable/torchvision/models.html).

In [7]:
```
from PIL import Image
import torchvision.transforms as transforms

# Set PIL to be tolerant of image files that are truncated.
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

def load_image(img_path):
    image = Image.open(img_path).convert('RGB')
    # resize to (244, 244) because VGG16 accept this shape
    in_transform = transforms.Compose([
                        transforms.Resize(size=(244, 244)),
                        transforms.ToTensor()]) # normalizaiton par
ameters from pytorch doc.

    # discard the transparent, alpha channel (that's the :3) and ad
d the batch dimension
    image = in_transform(image)[:3,:,:].unsqueeze(0)
    return image
```

In [8]:
```
def VGG16_predict(img_path):
    '''
    Use pre-trained VGG-16 model to obtain index corresponding to
    predicted ImageNet class for image at specified path

    Args:
        img_path: path to an image

    Returns:
        Index corresponding to VGG-16 model's prediction
    '''

    ## TODO: Complete the function.
    ## Load and pre-process an image from the given img_path
    ## Return the *index* of the predicted class for that image

    img = load_image(img_path)
    if use_cuda:
        img = img.cuda()
    ret = VGG16(img)
    return torch.max(ret,1)[1].item() # predicted class index
```

In [9]:
```
# predict dog using ImageNet class
VGG16_predict(dog_files_short[0])
```

Out[9]:
```
258
```

## (IMPLEMENTATION) Write a Dog Detector

While looking at the [dictionary (https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a)](https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a), you will notice that the categories corresponding to dogs appear in an uninterrupted sequence and correspond to dictionary keys 151-268, inclusive, to include all categories from `'Chihuahua'` to `'Mexican hairless'`. Thus, in order to check to see if an image is predicted to contain a dog by the pre-trained VGG-16 model, we need only check if the pre-trained model predicts an index between 151 and 268 (inclusive).

Use these ideas to complete the `dog_detector` function below, which returns `True` if a dog is detected in an image (and `False` if not).

In [10]:
```
### returns "True" if a dog is detected in the image stored at img_
path
def dog_detector(img_path):
    ## TODO: Complete the function.

    idx = VGG16_predict(img_path)
    return idx >= 151 and idx <= 268
```

```
In [11]: print(dog_detector(dog_files_short[0]))
         print(dog_detector(human_files_short[0]))
```

```
False
False
```

## (IMPLEMENTATION) Assess the Dog Detector

**Question 2:** Use the code cell below to test the performance of your `dog_detector` function.

- What percentage of the images in `human_files_short` have a detected dog?
- What percentage of the images in `dog_files_short` have a detected dog?

**Answer:**

```
In [12]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.

         def dog_detector_test(files):
             detection_cnt = 0;
             total_cnt = len(files)
             for file in files:
                 detection_cnt += dog_detector(file)
             return detection_cnt, total_cnt
```

```
In [13]: print("detect a dog in human_files: {} / {}".format(dog_detector_te
         st(human_files_short)[0], dog_detector_test(human_files_short)[1]))
         print("detect a dog in dog_files: {} / {}".format(dog_detector_test
         (dog_files_short)[0], dog_detector_test(dog_files_short)[1]))
```

```
detect a dog in human_files: 0 / 100
detect a dog in dog_files: 91 / 100
```

We suggest VGG-16 as a potential network to detect dog images in your algorithm, but you are free to explore other pre-trained networks (such as Inception-v3 (http://pytorch.org/docs/master/torchvision/models.html#inception-v3), ResNet-50 (http://pytorch.org/docs/master/torchvision/models.html#id3), etc). Please use the code cell below to test other pre-trained PyTorch models. If you decide to pursue this *optional* task, report performance on `human_files_short` and `dog_files_short`.

```
In [14]: ### (Optional)
         ### TODO: Report the performance of another pre-trained network.
         ### Feel free to use as many code cells as needed.
```

## Step 3: Create a CNN to Classify Dog Breeds (from Scratch)

Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of at least 10%. In Step 4 of this notebook, you will have the opportunity to use transfer learning to create a CNN that attains greatly improved accuracy.

We mention that the task of assigning breed to dogs from images is considered exceptionally challenging. To see why, consider that *even a human* would have trouble distinguishing between a Brittany and a Welsh Springer Spaniel.

| Brittany | Welsh Springer Spaniel |
|---|---|

It is not difficult to find other dog breed pairs with minimal inter-class variation (for instance, Curly-Coated Retrievers and American Water Spaniels).

| Curly-Coated Retriever | American Water Spaniel |
|---|---|

Likewise, recall that labradors come in yellow, chocolate, and black. Your vision-based algorithm will have to conquer this high intra-class variation to determine how to classify all of these different shades as the same breed.

| Yellow Labrador | Chocolate Labrador | Black Labrador |
|---|---|---|

We also mention that random chance presents an exceptionally low bar: setting aside the fact that the classes are slightly imabalanced, a random guess will provide a correct answer roughly 1 in 133 times, which corresponds to an accuracy of less than 1%.

Remember that the practice is far ahead of the theory in deep learning. Experiment with many different architectures, and trust your intuition. And, of course, have fun!

### (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

Use the code cell below to write three separate data loaders (http://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader) for the training, validation, and test datasets of dog images (located at `dogImages/train` , `dogImages/valid` , and `dogImages/test` , respectively). You may find this documentation on custom datasets (http://pytorch.org/docs/stable/torchvision/datasets.html) to be a useful resource. If you are interested in augmenting your training and/or validation data, check out the wide variety of transforms (http://pytorch.org/docs/stable/torchvision/transforms.html?highlight=transform)!

In [15]:
```
import os
from torchvision import datasets
import torchvision.transforms as transforms
import torch
import numpy as np
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

### TODO: Write data loaders for training, validation, and test set
s
## Specify appropriate transforms, and batch_sizes

batch_size = 20
num_workers = 0

data_dir = 'dogImages/'
train_dir = os.path.join(data_dir, 'train/')
valid_dir = os.path.join(data_dir, 'valid/')
test_dir = os.path.join(data_dir, 'test/')
```

In [16]:
```
standard_normalization = transforms.Normalize(mean=[0.485, 0.456, 0
.406],
                                              std=[0.229, 0.224, 0.
225])

data_transforms = {'train': transforms.Compose([transforms.RandomRe
sizedCrop(224),
                                     transforms.RandomHorizontalFli
p(),
                                     transforms.ToTensor(),
                                     standard_normalization]),
                   'val': transforms.Compose([transforms.Resize(256
),
                                     transforms.CenterCrop(224),
                                     transforms.ToTensor(),
                                     standard_normalization]),
                   'test': transforms.Compose([transforms.Resize(si
ze=(224,224)),
                                     transforms.ToTensor(),
                                     standard_normalization])
                  }
```

In [17]:
```
train_data = datasets.ImageFolder(train_dir, transform=data_transfo
rms['train'])
valid_data = datasets.ImageFolder(valid_dir, transform=data_transfo
rms['val'])
test_data = datasets.ImageFolder(test_dir, transform=data_transform
s['test'])
```

In [18]:
```
train_loader = torch.utils.data.DataLoader(train_data,
                                           batch_size=batch_size,
                                           num_workers=num_workers,
                                           shuffle=True)
valid_loader = torch.utils.data.DataLoader(valid_data,
                                           batch_size=batch_size,
                                           num_workers=num_workers,
                                           shuffle=False)
test_loader = torch.utils.data.DataLoader(test_data,
                                           batch_size=batch_size,
                                           num_workers=num_workers,
                                           shuffle=False)
loaders_scratch = {
    'train': train_loader,
    'valid': valid_loader,
    'test': test_loader
}
```

**Question 3:** Describe your chosen procedure for preprocessing the data.

- How does your code resize the images (by cropping, stretching, etc)? What size did you pick for the input tensor, and why?
- Did you decide to augment the dataset? If so, how (through translations, flips, rotations, etc)? If not, why not?

**Answer**: I've applied RandomResizedCrop & RandomHorizontalFlip to just train_data. This will do both image augmentations and resizing jobs. Image augmentation will give randomness to the dataset so, it prevents overfitting and I can expect better performance of model when it's predicting toward test_data. On the other hand, I've done Resize of (256) and then, center crop to make 224 X 224. Since valid_data will be used for validation check, I will not do image augmentations. For the test_data, I've applied only image resizing.

## (IMPLEMENTATION) Model Architecture

Create a CNN to classify dog breed. Use the template in the code cell below.

In [19]:
```
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

dog_names = [item[20:-1] for item in sorted(glob("dogImages/train/*
/"))]

num_classes = len(dog_names) # total classes of dog breeds

print('There are %d total dog categories.' % len(dog_names))
```

```
There are 133 total dog categories.
```

In [20]:
```
import torch.nn as nn
import torch.nn.functional as F

# define the CNN architecture
class Net(nn.Module):
    ### TODO: choose an architecture, and complete the class
    def __init__(self):
        super(Net, self).__init__()
        ## Define layers of a CNN
        self.conv1 = nn.Conv2d(3, 32, 3, stride=2, padding=1)
        self.conv2 = nn.Conv2d(32, 64, 3, stride=2, padding=1)
        self.conv3 = nn.Conv2d(64, 128, 3, padding=1)

        # pool
        self.pool = nn.MaxPool2d(2, 2)

        # fully-connected
        self.fc1 = nn.Linear(7*7*128, 500)
        self.fc2 = nn.Linear(500, num_classes)

        # drop-out
        self.dropout = nn.Dropout(0.3)

    def forward(self, x):
        ## Define forward behavior
        x = F.relu(self.conv1(x))
        x = self.pool(x)
        x = F.relu(self.conv2(x))
        x = self.pool(x)
        x = F.relu(self.conv3(x))
        x = self.pool(x)

        # flatten
        x = x.view(-1, 7*7*128)

        x = self.dropout(x)
        x = F.relu(self.fc1(x))

        x = self.dropout(x)
        x = self.fc2(x)
        return x

#-#-# You do NOT have to modify the code below this line. #-#-#

# instantiate the CNN
model_scratch = Net()

# move tensors to GPU if CUDA is available
if use_cuda:
    model_scratch.cuda()
```

**Question 4:** Outline the steps you took to get to your final CNN architecture and your reasoning at each step.

**Answer:**

(conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))

(conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))

(conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))

(pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)

(fc1): Linear(in_features=6272, out_features=500, bias=True)

(fc2): Linear(in_features=500, out_features=133, bias=True)

(dropout): Dropout(p=0.3, inplace=False)

First 2 conv layers I've applied kernel_size of 3 with stride 2, this will lead to downsize of input image by 2. after 2 conv layers, maxpooling with stride 2 is placed and this will lead to downsize of input image by 2. The 3rd conv layers is consist of kernel_size of 3 with stride 1, and this will not reduce input image. after final maxpooling with stride 2, the total output image size is downsized by factor of 32 and the depth will be 128. I've applied dropout of 0.3 in order to prevent overfitting. Fully-connected layer is placed and then, 2nd fully-connected layer is intended to produce final output_size which predicts classes of breeds.

## (IMPLEMENTATION) Specify Loss Function and Optimizer

Use the next code cell to specify a [loss function (http://pytorch.org/docs/stable/nn.html#loss-functions)](http://pytorch.org/docs/stable/nn.html#loss-functions) and [optimizer (http://pytorch.org/docs/stable/optim.html)](http://pytorch.org/docs/stable/optim.html). Save the chosen loss function as `criterion_scratch` , and the optimizer as `optimizer_scratch` below.

In [21]:
```
import torch.optim as optim

### TODO: select loss function
criterion_scratch = nn.CrossEntropyLoss()

### TODO: select optimizer
optimizer_scratch = optim.SGD(model_scratch.parameters(), lr = 0.05
)
```

## (IMPLEMENTATION) Train and Validate the Model

Train and validate your model in the code cell below. [Save the final model parameters (http://pytorch.org/docs/master/notes/serialization.html)](http://pytorch.org/docs/master/notes/serialization.html) at filepath `'model_scratch.pt'` .

In [43]:
```
# the following import is required for training to be robust to tru
ncated images
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

def train(n_epochs, loaders, model, optimizer, criterion, use_cuda,
save_path, last_validation_loss=None):
    """returns trained model"""
    # initialize tracker for minimum validation loss
    if last_validation_loss is not None:
        valid_loss_min = last_validation_loss
    else:
        valid_loss_min = np.Inf

    for epoch in range(1, n_epochs+1):
        # initialize variables to monitor training and validation l
oss
        train_loss = 0.0
        valid_loss = 0.0

        ###################
        # train the model #
        ###################
        model.train()
        for batch_idx, (data, target) in enumerate(loaders['train']
):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## find the loss and update the model parameters accord
ingly
            ## record the average training loss, using something li
ke
            ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (
loss.data - train_loss))

            # initialize weights to zero
            optimizer.zero_grad()

            output = model(data)

            # calculate loss
            loss = criterion(output, target)

            # back prop
            loss.backward()

            # grad
            optimizer.step()

            train_loss = train_loss + ((1 / (batch_idx + 1)) * (los
s.data - train_loss))

            if batch_idx % 100 == 0:
                print('Epoch %d, Batch %d loss: %.6f' %
                  (epoch, batch_idx + 1, train_loss))

        ######################
        # validate the model #
        ######################
        model.eval()
        for batch_idx, (data, target) in enumerate(loaders['valid']
):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## update the average validation loss
            output = model(data)
            loss = criterion(output, target)
            valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (los
s.data - valid_loss))

        # print training/validation statistics
        print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss:
{:.6f}'.format(
            epoch,
            train_loss,
            valid_loss
            ))

        ## TODO: save the model if validation loss has decreased
        if valid_loss < valid_loss_min:
            torch.save(model.state_dict(), save_path)
            print('Validation loss decreased ({:.6f} --> {:.6f}).
Saving model ...'.format(
            valid_loss_min,
            valid_loss))
            valid_loss_min = valid_loss

    # return trained model
    return model


# train the model
model_scratch = train(100, loaders_scratch, model_scratch, optimize
r_scratch,
                      criterion_scratch, use_cuda, 'model_scratch.p
t')
```

```python
# load the model that got the best validation accuracy
model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

```
Epoch 1, Batch 1 loss: 2.918628
Epoch 1, Batch 101 loss: 3.221683
Epoch 1, Batch 201 loss: 3.289750
Epoch 1, Batch 301 loss: 3.306213
Epoch: 1        Training Loss: 3.307929        Validation Loss: 3
.199630
Validation loss decreased (inf --> 3.199630).  Saving model ...
Epoch 2, Batch 1 loss: 3.092693
Epoch 2, Batch 101 loss: 3.285815
Epoch 2, Batch 201 loss: 3.290284
Epoch 2, Batch 301 loss: 3.318204
Epoch: 2        Training Loss: 3.311082        Validation Loss: 3
.241782
Epoch 3, Batch 1 loss: 3.259407
Epoch 3, Batch 101 loss: 3.175607
Epoch 3, Batch 201 loss: 3.227169
Epoch 3, Batch 301 loss: 3.246825
Epoch: 3        Training Loss: 3.251623        Validation Loss: 3
.124890
Validation loss decreased (3.199630 --> 3.124890).  Saving model .
..
Epoch 4, Batch 1 loss: 3.207994
Epoch 4, Batch 101 loss: 3.183179
Epoch 4, Batch 201 loss: 3.188883
Epoch 4, Batch 301 loss: 3.227762
Epoch: 4        Training Loss: 3.235133        Validation Loss: 3
.225542
Epoch 5, Batch 1 loss: 4.183380
Epoch 5, Batch 101 loss: 3.156750
Epoch 5, Batch 201 loss: 3.193622
Epoch 5, Batch 301 loss: 3.212517
Epoch: 5        Training Loss: 3.224883        Validation Loss: 3
.257174
Epoch 6, Batch 1 loss: 3.690592
Epoch 6, Batch 101 loss: 3.137492
Epoch 6, Batch 201 loss: 3.172515
Epoch 6, Batch 301 loss: 3.165958
Epoch: 6        Training Loss: 3.170963        Validation Loss: 3
.254240
Epoch 7, Batch 1 loss: 3.311813
Epoch 7, Batch 101 loss: 3.122486
Epoch 7, Batch 201 loss: 3.141460
Epoch 7, Batch 301 loss: 3.159331
Epoch: 7        Training Loss: 3.153516        Validation Loss: 3
.135317
Epoch 8, Batch 1 loss: 2.404134
Epoch 8, Batch 101 loss: 3.178955
Epoch 8, Batch 201 loss: 3.167447
Epoch 8, Batch 301 loss: 3.167445
Epoch: 8        Training Loss: 3.176495        Validation Loss: 3
.021331
```

```
Validation loss decreased (3.124890 --> 3.021331).  Saving model .
..
Epoch 9, Batch 1 loss: 2.563126
Epoch 9, Batch 101 loss: 3.093402
Epoch 9, Batch 201 loss: 3.101562
Epoch 9, Batch 301 loss: 3.125563
Epoch: 9 	Training Loss: 3.128212 	Validation Loss: 3
.065044
Epoch 10, Batch 1 loss: 2.718918
Epoch 10, Batch 101 loss: 3.032216
Epoch 10, Batch 201 loss: 3.046798
Epoch 10, Batch 301 loss: 3.063488
Epoch: 10 	Training Loss: 3.081234 	Validation Loss: 3
.065432
Epoch 11, Batch 1 loss: 2.532519
Epoch 11, Batch 101 loss: 3.053833
Epoch 11, Batch 201 loss: 3.032621
Epoch 11, Batch 301 loss: 3.063879
Epoch: 11 	Training Loss: 3.068458 	Validation Loss: 3
.042473
Epoch 12, Batch 1 loss: 2.451418
Epoch 12, Batch 101 loss: 3.084696
Epoch 12, Batch 201 loss: 3.063165
Epoch 12, Batch 301 loss: 3.081162
Epoch: 12 	Training Loss: 3.083979 	Validation Loss: 3
.074014
Epoch 13, Batch 1 loss: 2.747142
Epoch 13, Batch 101 loss: 2.986626
Epoch 13, Batch 201 loss: 3.017360
Epoch 13, Batch 301 loss: 3.015573
Epoch: 13 	Training Loss: 3.028197 	Validation Loss: 3
.273300
Epoch 14, Batch 1 loss: 2.777329
Epoch 14, Batch 101 loss: 2.999544
Epoch 14, Batch 201 loss: 2.992452
Epoch 14, Batch 301 loss: 3.017071
Epoch: 14 	Training Loss: 3.023845 	Validation Loss: 3
.064282
Epoch 15, Batch 1 loss: 3.004068
Epoch 15, Batch 101 loss: 3.033193
Epoch 15, Batch 201 loss: 3.037702
Epoch 15, Batch 301 loss: 3.035058
Epoch: 15 	Training Loss: 3.042484 	Validation Loss: 3
.259527
Epoch 16, Batch 1 loss: 1.963495
Epoch 16, Batch 101 loss: 2.949900
Epoch 16, Batch 201 loss: 2.965564
Epoch 16, Batch 301 loss: 2.981721
Epoch: 16 	Training Loss: 2.991417 	Validation Loss: 3
.086831
Epoch 17, Batch 1 loss: 2.323443
Epoch 17, Batch 101 loss: 2.916742
Epoch 17, Batch 201 loss: 2.967009
Epoch 17, Batch 301 loss: 2.973333
```

```
Epoch: 17       Training Loss: 2.987592         Validation Loss: 2
.936970
Validation loss decreased (3.021331 --> 2.936970).  Saving model .
..
Epoch 18, Batch 1 loss: 2.761749
Epoch 18, Batch 101 loss: 2.928266
Epoch 18, Batch 201 loss: 2.932333
Epoch 18, Batch 301 loss: 2.961367
Epoch: 18       Training Loss: 2.971156         Validation Loss: 3
.030679
Epoch 19, Batch 1 loss: 2.784611
Epoch 19, Batch 101 loss: 2.933200
Epoch 19, Batch 201 loss: 2.920540
Epoch 19, Batch 301 loss: 2.957278
Epoch: 19       Training Loss: 2.964658         Validation Loss: 3
.086737
Epoch 20, Batch 1 loss: 3.094895
Epoch 20, Batch 101 loss: 2.888259
Epoch 20, Batch 201 loss: 2.906236
Epoch 20, Batch 301 loss: 2.946159
Epoch: 20       Training Loss: 2.947231         Validation Loss: 3
.008924
Epoch 21, Batch 1 loss: 2.771761
Epoch 21, Batch 101 loss: 2.881482
Epoch 21, Batch 201 loss: 2.885442
Epoch 21, Batch 301 loss: 2.921321
Epoch: 21       Training Loss: 2.912856         Validation Loss: 3
.221095
Epoch 22, Batch 1 loss: 2.950565
Epoch 22, Batch 101 loss: 2.832697
Epoch 22, Batch 201 loss: 2.867759
Epoch 22, Batch 301 loss: 2.901806
Epoch: 22       Training Loss: 2.913217         Validation Loss: 3
.207551
Epoch 23, Batch 1 loss: 2.634189
Epoch 23, Batch 101 loss: 2.867587
Epoch 23, Batch 201 loss: 2.892083
Epoch 23, Batch 301 loss: 2.900461
Epoch: 23       Training Loss: 2.893768         Validation Loss: 3
.172623
Epoch 24, Batch 1 loss: 2.983750
Epoch 24, Batch 101 loss: 2.846616
Epoch 24, Batch 201 loss: 2.883505
Epoch 24, Batch 301 loss: 2.903487
Epoch: 24       Training Loss: 2.896640         Validation Loss: 3
.129655
Epoch 25, Batch 1 loss: 2.731677
Epoch 25, Batch 101 loss: 2.809287
Epoch 25, Batch 201 loss: 2.829960
Epoch 25, Batch 301 loss: 2.831648
Epoch: 25       Training Loss: 2.842415         Validation Loss: 2
.968261
Epoch 26, Batch 1 loss: 2.379530
Epoch 26, Batch 101 loss: 2.792344
Epoch 26, Batch 201 loss: 2.790948
Epoch 26, Batch 301 loss: 2.797786
Epoch: 26       Training Loss: 2.809493         Validation Loss: 3
.153161
Epoch 27, Batch 1 loss: 3.139229
Epoch 27, Batch 101 loss: 2.828769
Epoch 27, Batch 201 loss: 2.814367
Epoch 27, Batch 301 loss: 2.821673
Epoch: 27       Training Loss: 2.840612         Validation Loss: 3
.078581
Epoch 28, Batch 1 loss: 2.257049
Epoch 28, Batch 101 loss: 2.805391
Epoch 28, Batch 201 loss: 2.826566
Epoch 28, Batch 301 loss: 2.842007
Epoch: 28       Training Loss: 2.857179         Validation Loss: 3
.047788
Epoch 29, Batch 1 loss: 2.958695
Epoch 29, Batch 101 loss: 2.718238
Epoch 29, Batch 201 loss: 2.767614
Epoch 29, Batch 301 loss: 2.788312
Epoch: 29       Training Loss: 2.794181         Validation Loss: 3
.084526
Epoch 30, Batch 1 loss: 2.421609
Epoch 30, Batch 101 loss: 2.761370
Epoch 30, Batch 201 loss: 2.788253
Epoch 30, Batch 301 loss: 2.798777
Epoch: 30       Training Loss: 2.804379         Validation Loss: 3
.083914
Epoch 31, Batch 1 loss: 2.859287
Epoch 31, Batch 101 loss: 2.711850
Epoch 31, Batch 201 loss: 2.740164
Epoch 31, Batch 301 loss: 2.816247
Epoch: 31       Training Loss: 2.852515         Validation Loss: 3
.116521
Epoch 32, Batch 1 loss: 3.455798
Epoch 32, Batch 101 loss: 2.847440
Epoch 32, Batch 201 loss: 2.833255
Epoch 32, Batch 301 loss: 2.856749
Epoch: 32       Training Loss: 2.864081         Validation Loss: 3
.105147
Epoch 33, Batch 1 loss: 1.917945
Epoch 33, Batch 101 loss: 2.731455
Epoch 33, Batch 201 loss: 2.757036
Epoch 33, Batch 301 loss: 2.760255
Epoch: 33       Training Loss: 2.758474         Validation Loss: 3
.114823
Epoch 34, Batch 1 loss: 2.504156
Epoch 34, Batch 101 loss: 2.733533
Epoch 34, Batch 201 loss: 2.711686
Epoch 34, Batch 301 loss: 2.745030
Epoch: 34       Training Loss: 2.754000         Validation Loss: 3
.020136
Epoch 35, Batch 1 loss: 2.128798
Epoch 35, Batch 101 loss: 2.733469
Epoch 35, Batch 201 loss: 2.725822
Epoch 35, Batch 301 loss: 2.732501
Epoch: 35       Training Loss: 2.726997         Validation Loss: 3
.039073
Epoch 36, Batch 1 loss: 2.492565
Epoch 36, Batch 101 loss: 2.660691
Epoch 36, Batch 201 loss: 2.689794
Epoch 36, Batch 301 loss: 2.713023
Epoch: 36       Training Loss: 2.724287         Validation Loss: 3
.076482
Epoch 37, Batch 1 loss: 3.070942
Epoch 37, Batch 101 loss: 2.700633
Epoch 37, Batch 201 loss: 2.695314
Epoch 37, Batch 301 loss: 2.714947
Epoch: 37       Training Loss: 2.714000         Validation Loss: 3
.117216
Epoch 38, Batch 1 loss: 3.145581
Epoch 38, Batch 101 loss: 2.610207
Epoch 38, Batch 201 loss: 2.696827
Epoch 38, Batch 301 loss: 2.704425
Epoch: 38       Training Loss: 2.703193         Validation Loss: 3
.061894
Epoch 39, Batch 1 loss: 2.794775
Epoch 39, Batch 101 loss: 2.625060
Epoch 39, Batch 201 loss: 2.659373
Epoch 39, Batch 301 loss: 2.711572
Epoch: 39       Training Loss: 2.726904         Validation Loss: 3
.117506
Epoch 40, Batch 1 loss: 3.118272
Epoch 40, Batch 101 loss: 2.685221
Epoch 40, Batch 201 loss: 2.698396
Epoch 40, Batch 301 loss: 2.698029
Epoch: 40       Training Loss: 2.707303         Validation Loss: 2
.971020
Epoch 41, Batch 1 loss: 2.480512
Epoch 41, Batch 101 loss: 2.637603
Epoch 41, Batch 201 loss: 2.621913
Epoch 41, Batch 301 loss: 2.640407
Epoch: 41       Training Loss: 2.662461         Validation Loss: 3
.008535
Epoch 42, Batch 1 loss: 2.697654
Epoch 42, Batch 101 loss: 2.599003
Epoch 42, Batch 201 loss: 2.639944
Epoch 42, Batch 301 loss: 2.663624
Epoch: 42       Training Loss: 2.668747         Validation Loss: 3
.059430
Epoch 43, Batch 1 loss: 2.091295
Epoch 43, Batch 101 loss: 2.527978
Epoch 43, Batch 201 loss: 2.591383
Epoch 43, Batch 301 loss: 2.615644
Epoch: 43       Training Loss: 2.631690         Validation Loss: 3
.087224
Epoch 44, Batch 1 loss: 2.476293
Epoch 44, Batch 101 loss: 2.586285
Epoch 44, Batch 201 loss: 2.602380
Epoch 44, Batch 301 loss: 2.603488
Epoch: 44       Training Loss: 2.611991         Validation Loss: 3
.109641
Epoch 45, Batch 1 loss: 2.147672
Epoch 45, Batch 101 loss: 2.570391
Epoch 45, Batch 201 loss: 2.618181
Epoch 45, Batch 301 loss: 2.637173
Epoch: 45       Training Loss: 2.640724         Validation Loss: 3
.057404
Epoch 46, Batch 1 loss: 2.589255
Epoch 46, Batch 101 loss: 2.577376
Epoch 46, Batch 201 loss: 2.601587
Epoch 46, Batch 301 loss: 2.616400
Epoch: 46       Training Loss: 2.632764         Validation Loss: 2
.938483
Epoch 47, Batch 1 loss: 2.832523
Epoch 47, Batch 101 loss: 2.674163
Epoch 47, Batch 201 loss: 2.633794
Epoch 47, Batch 301 loss: 2.640394
Epoch: 47       Training Loss: 2.640421         Validation Loss: 2
.989507
Epoch 48, Batch 1 loss: 1.563405
Epoch 48, Batch 101 loss: 2.615142
Epoch 48, Batch 201 loss: 2.628876
Epoch 48, Batch 301 loss: 2.632274
Epoch: 48       Training Loss: 2.650002         Validation Loss: 3
.027513
Epoch 49, Batch 1 loss: 2.443305
Epoch 49, Batch 101 loss: 2.600885
Epoch 49, Batch 201 loss: 2.559845
Epoch 49, Batch 301 loss: 2.593221
Epoch: 49       Training Loss: 2.597759         Validation Loss: 3
.071438
Epoch 50, Batch 1 loss: 2.261232
Epoch 50, Batch 101 loss: 2.525638
Epoch 50, Batch 201 loss: 2.552005
Epoch 50, Batch 301 loss: 2.579549
Epoch: 50       Training Loss: 2.575054         Validation Loss: 3
.109624
Epoch 51, Batch 1 loss: 2.331091
Epoch 51, Batch 101 loss: 2.548120
Epoch 51, Batch 201 loss: 2.556005
Epoch 51, Batch 301 loss: 2.553039
Epoch: 51       Training Loss: 2.551229         Validation Loss: 3
.057757
Epoch 52, Batch 1 loss: 3.219290
Epoch 52, Batch 101 loss: 2.498138
Epoch 52, Batch 201 loss: 2.565975
Epoch 52, Batch 301 loss: 2.577033
Epoch: 52       Training Loss: 2.590844         Validation Loss: 3
.035457
Epoch 53, Batch 1 loss: 1.778721
Epoch 53, Batch 101 loss: 2.585257
Epoch 53, Batch 201 loss: 2.630905
Epoch 53, Batch 301 loss: 2.635998
Epoch: 53       Training Loss: 2.647718         Validation Loss: 3
.010261
Epoch 54, Batch 1 loss: 3.034108
Epoch 54, Batch 101 loss: 2.556215
Epoch 54, Batch 201 loss: 2.543283
Epoch 54, Batch 301 loss: 2.569809
Epoch: 54       Training Loss: 2.571547         Validation Loss: 3
.122781
Epoch 55, Batch 1 loss: 2.145656
Epoch 55, Batch 101 loss: 2.512729
Epoch 55, Batch 201 loss: 2.536060
Epoch 55, Batch 301 loss: 2.555376
Epoch: 55       Training Loss: 2.557665         Validation Loss: 3
.061721
Epoch 56, Batch 1 loss: 2.896239
Epoch 56, Batch 101 loss: 2.503813
Epoch 56, Batch 201 loss: 2.527331
Epoch 56, Batch 301 loss: 2.575082
Epoch: 56       Training Loss: 2.584782         Validation Loss: 3
.096818
Epoch 57, Batch 1 loss: 2.757879
Epoch 57, Batch 101 loss: 2.504660
Epoch 57, Batch 201 loss: 2.534247
Epoch 57, Batch 301 loss: 2.540496
Epoch: 57       Training Loss: 2.535345         Validation Loss: 3
.219729
Epoch 58, Batch 1 loss: 2.678518
Epoch 58, Batch 101 loss: 2.543174
Epoch 58, Batch 201 loss: 2.556470
Epoch 58, Batch 301 loss: 2.592449
Epoch: 58       Training Loss: 2.587482         Validation Loss: 3
.201838
Epoch 59, Batch 1 loss: 2.001789
Epoch 59, Batch 101 loss: 2.496538
Epoch 59, Batch 201 loss: 2.526327
Epoch 59, Batch 301 loss: 2.547430
Epoch: 59       Training Loss: 2.563679         Validation Loss: 3
.096515
Epoch 60, Batch 1 loss: 2.376029
Epoch 60, Batch 101 loss: 2.570046
Epoch 60, Batch 201 loss: 2.561596
Epoch 60, Batch 301 loss: 2.581009
Epoch: 60       Training Loss: 2.581386         Validation Loss: 3
.129267
Epoch 61, Batch 1 loss: 2.487393
Epoch 61, Batch 101 loss: 2.474390
Epoch 61, Batch 201 loss: 2.509543
Epoch 61, Batch 301 loss: 2.531362
Epoch: 61       Training Loss: 2.532427         Validation Loss: 3
.128937
Epoch 62, Batch 1 loss: 2.401991
Epoch 62, Batch 101 loss: 2.358077
Epoch 62, Batch 201 loss: 2.468173
Epoch 62, Batch 301 loss: 2.497471
Epoch: 62       Training Loss: 2.508074         Validation Loss: 3
.016359
Epoch 63, Batch 1 loss: 1.845563
Epoch 63, Batch 101 loss: 2.470813
Epoch 63, Batch 201 loss: 2.500232
Epoch 63, Batch 301 loss: 2.549671
Epoch: 63       Training Loss: 2.558184         Validation Loss: 3
.092916
Epoch 64, Batch 1 loss: 2.590133
Epoch 64, Batch 101 loss: 2.531426
Epoch 64, Batch 201 loss: 2.554724
Epoch 64, Batch 301 loss: 2.535498
Epoch: 64       Training Loss: 2.553844         Validation Loss: 3
.094963
Epoch 65, Batch 1 loss: 3.198025
Epoch 65, Batch 101 loss: 2.458868
Epoch 65, Batch 201 loss: 2.492734
Epoch 65, Batch 301 loss: 2.517398
Epoch: 65       Training Loss: 2.533792         Validation Loss: 3
.096785
Epoch 66, Batch 1 loss: 3.003726
Epoch 66, Batch 101 loss: 2.541976
Epoch 66, Batch 201 loss: 2.544878
Epoch 66, Batch 301 loss: 2.556627
Epoch: 66       Training Loss: 2.559719         Validation Loss: 3
.239180
Epoch 67, Batch 1 loss: 2.646923
Epoch 67, Batch 101 loss: 2.390568
Epoch 67, Batch 201 loss: 2.460775
Epoch 67, Batch 301 loss: 2.469724
Epoch: 67       Training Loss: 2.476529         Validation Loss: 3
.205570
Epoch 68, Batch 1 loss: 2.682815
Epoch 68, Batch 101 loss: 2.467050
Epoch 68, Batch 201 loss: 2.493742
Epoch 68, Batch 301 loss: 2.504480
Epoch: 68       Training Loss: 2.516115         Validation Loss: 3
.089036
Epoch 69, Batch 1 loss: 2.815473
Epoch 69, Batch 101 loss: 2.468833
Epoch 69, Batch 201 loss: 2.491901
Epoch 69, Batch 301 loss: 2.515463
Epoch: 69       Training Loss: 2.512825         Validation Loss: 3
.099096
Epoch 70, Batch 1 loss: 2.276610
Epoch 70, Batch 101 loss: 2.418893
Epoch 70, Batch 201 loss: 2.448633
Epoch 70, Batch 301 loss: 2.476793
Epoch: 70       Training Loss: 2.494068         Validation Loss: 3
.059645
Epoch 71, Batch 1 loss: 2.144571
Epoch 71, Batch 101 loss: 2.468919
Epoch 71, Batch 201 loss: 2.471390
Epoch 71, Batch 301 loss: 2.513283
Epoch: 71       Training Loss: 2.519278         Validation Loss: 3
.490804
Epoch 72, Batch 1 loss: 2.591059
Epoch 72, Batch 101 loss: 2.368181
Epoch 72, Batch 201 loss: 2.442715
Epoch 72, Batch 301 loss: 2.465788
Epoch: 72       Training Loss: 2.459814         Validation Loss: 3
.064104
Epoch 73, Batch 1 loss: 2.052522
Epoch 73, Batch 101 loss: 2.346975
Epoch 73, Batch 201 loss: 2.390579
Epoch 73, Batch 301 loss: 2.422739
Epoch: 73       Training Loss: 2.432180         Validation Loss: 3
.079348
Epoch 74, Batch 1 loss: 1.926699
Epoch 74, Batch 101 loss: 2.407346
Epoch 74, Batch 201 loss: 2.437426
Epoch 74, Batch 301 loss: 2.457254
Epoch: 74       Training Loss: 2.453607         Validation Loss: 3
.177098
Epoch 75, Batch 1 loss: 2.189857
Epoch 75, Batch 101 loss: 2.393192
Epoch 75, Batch 201 loss: 2.389501
Epoch 75, Batch 301 loss: 2.465132
Epoch: 75       Training Loss: 2.467666         Validation Loss: 3
.101878
Epoch 76, Batch 1 loss: 2.023543
Epoch 76, Batch 101 loss: 2.340800
Epoch 76, Batch 201 loss: 2.416945
Epoch 76, Batch 301 loss: 2.441984
Epoch: 76       Training Loss: 2.446344         Validation Loss: 3
.180658
Epoch 77, Batch 1 loss: 2.076280
Epoch 77, Batch 101 loss: 2.399086
Epoch 77, Batch 201 loss: 2.413696
Epoch 77, Batch 301 loss: 2.462631
Epoch: 77       Training Loss: 2.459469         Validation Loss: 3
.110030
Epoch 78, Batch 1 loss: 2.078255
Epoch 78, Batch 101 loss: 2.445366
Epoch 78, Batch 201 loss: 2.448024
Epoch 78, Batch 301 loss: 2.500926
Epoch: 78       Training Loss: 2.484104         Validation Loss: 3
.126999
Epoch 79, Batch 1 loss: 2.867308
Epoch 79, Batch 101 loss: 2.373958
Epoch 79, Batch 201 loss: 2.414383
Epoch 79, Batch 301 loss: 2.433578
Epoch: 79       Training Loss: 2.426444         Validation Loss: 3
.123993
Epoch 80, Batch 1 loss: 1.580312
Epoch 80, Batch 101 loss: 2.461602
Epoch 80, Batch 201 loss: 2.469933
Epoch 80, Batch 301 loss: 2.505858
Epoch: 80       Training Loss: 2.506072         Validation Loss: 3
.173347
Epoch 81, Batch 1 loss: 3.183259
Epoch 81, Batch 101 loss: 2.427786
Epoch 81, Batch 201 loss: 2.437697
Epoch 81, Batch 301 loss: 2.458012
Epoch: 81       Training Loss: 2.460238         Validation Loss: 3
.187314
Epoch 82, Batch 1 loss: 2.605491
Epoch 82, Batch 101 loss: 2.398955
Epoch 82, Batch 201 loss: 2.404615
Epoch 82, Batch 301 loss: 2.453917
Epoch: 82       Training Loss: 2.463429         Validation Loss: 3
.118062
Epoch 83, Batch 1 loss: 3.030079
Epoch 83, Batch 101 loss: 2.310160
Epoch 83, Batch 201 loss: 2.366955
Epoch 83, Batch 301 loss: 2.401325
Epoch: 83       Training Loss: 2.417225         Validation Loss: 3
.260258
Epoch 84, Batch 1 loss: 2.126477
Epoch 84, Batch 101 loss: 2.417737
Epoch 84, Batch 201 loss: 2.447594
Epoch 84, Batch 301 loss: 2.494314
Epoch: 84       Training Loss: 2.486869         Validation Loss: 3
.098482
Epoch 85, Batch 1 loss: 2.965950
Epoch 85, Batch 101 loss: 2.443319
Epoch 85, Batch 201 loss: 2.453618
Epoch 85, Batch 301 loss: 2.456028
Epoch: 85       Training Loss: 2.445006         Validation Loss: 3
.173223
Epoch 86, Batch 1 loss: 1.952532
Epoch 86, Batch 101 loss: 2.393686
Epoch 86, Batch 201 loss: 2.433365
Epoch 86, Batch 301 loss: 2.418580
Epoch: 86       Training Loss: 2.425376         Validation Loss: 3
.129917
Epoch 87, Batch 1 loss: 2.230582
Epoch 87, Batch 101 loss: 2.413737
Epoch 87, Batch 201 loss: 2.435199
Epoch 87, Batch 301 loss: 2.417565
Epoch: 87       Training Loss: 2.421133         Validation Loss: 3
.261008
Epoch 88, Batch 1 loss: 2.042959
Epoch 88, Batch 101 loss: 2.392586
Epoch 88, Batch 201 loss: 2.407826
Epoch 88, Batch 301 loss: 2.446165
Epoch: 88       Training Loss: 2.438335         Validation Loss: 3
.262260
Epoch 89, Batch 1 loss: 2.411618
Epoch 89, Batch 101 loss: 2.341581
Epoch 89, Batch 201 loss: 2.383325
Epoch 89, Batch 301 loss: 2.400562
Epoch: 89       Training Loss: 2.404849         Validation Loss: 3
.096530
Epoch 90, Batch 1 loss: 2.120583
Epoch 90, Batch 101 loss: 2.280442
Epoch 90, Batch 201 loss: 2.329566
Epoch 90, Batch 301 loss: 2.362228
Epoch: 90       Training Loss: 2.362058         Validation Loss: 3
.293759
Epoch 91, Batch 1 loss: 2.223611
Epoch 91, Batch 101 loss: 2.308557
Epoch 91, Batch 201 loss: 2.362944
Epoch 91, Batch 301 loss: 2.386079
Epoch: 91       Training Loss: 2.387696         Validation Loss: 3
.002893
Epoch 92, Batch 1 loss: 2.477330
Epoch 92, Batch 101 loss: 2.408762
Epoch 92, Batch 201 loss: 2.446839
Epoch 92, Batch 301 loss: 2.421407
Epoch: 92       Training Loss: 2.431778         Validation Loss: 3
.054229
Epoch 93, Batch 1 loss: 3.509327
Epoch 93, Batch 101 loss: 2.329432
Epoch 93, Batch 201 loss: 2.393855
Epoch 93, Batch 301 loss: 2.414273
Epoch: 93       Training Loss: 2.429205         Validation Loss: 3
.235039
Epoch 94, Batch 1 loss: 2.380925
Epoch 94, Batch 101 loss: 2.438695
Epoch 94, Batch 201 loss: 2.429906
Epoch 94, Batch 301 loss: 2.426659
Epoch: 94       Training Loss: 2.428618         Validation Loss: 3
.150995
Epoch 95, Batch 1 loss: 2.549477
Epoch 95, Batch 101 loss: 2.324190
Epoch 95, Batch 201 loss: 2.363676
Epoch 95, Batch 301 loss: 2.343451
Epoch: 95       Training Loss: 2.346784         Validation Loss: 3
.360411
Epoch 96, Batch 1 loss: 2.293864
Epoch 96, Batch 101 loss: 2.356846
Epoch 96, Batch 201 loss: 2.400235
Epoch 96, Batch 301 loss: 2.390958
Epoch: 96       Training Loss: 2.386241         Validation Loss: 3
.153911
Epoch 97, Batch 1 loss: 1.714128
Epoch 97, Batch 101 loss: 2.363100
Epoch 97, Batch 201 loss: 2.369848
Epoch 97, Batch 301 loss: 2.398454
Epoch: 97       Training Loss: 2.391994         Validation Loss: 3
.079845
Epoch 98, Batch 1 loss: 2.729854
Epoch 98, Batch 101 loss: 2.345734
Epoch 98, Batch 201 loss: 2.335580
Epoch 98, Batch 301 loss: 2.371019
Epoch: 98       Training Loss: 2.386664         Validation Loss: 3
.161851
Epoch 99, Batch 1 loss: 2.188588
Epoch 99, Batch 101 loss: 2.284200
Epoch 99, Batch 201 loss: 2.351975
Epoch 99, Batch 301 loss: 2.356364
Epoch: 99       Training Loss: 2.364943         Validation Loss: 3
.134000
Epoch 100, Batch 1 loss: 2.194869
Epoch 100, Batch 101 loss: 2.283167
Epoch 100, Batch 201 loss: 2.356908
Epoch 100, Batch 301 loss: 2.390168
Epoch: 100      Training Loss: 2.393477         Validation Loss: 3
.305022
```

```
Out[43]: <All keys matched successfully>
```

### (IMPLEMENTATION) Test the Model

Try out your model on the test dataset of dog images. Use the code cell below to calculate and print the test loss and accuracy. Ensure that your test accuracy is greater than 10%.

```
In [44]: def test(loaders, model, criterion, use_cuda):

             # monitor test loss and accuracy
             test_loss = 0.
             correct = 0.
             total = 0.

             model.eval()
             for batch_idx, (data, target) in enumerate(loaders['test']):
                 # move to GPU
                 if use_cuda:
                     data, target = data.cuda(), target.cuda()
                 # forward pass: compute predicted outputs by passing inputs
         to the model
                 output = model(data)
                 # calculate the loss
                 loss = criterion(output, target)
                 # update average test loss
                 test_loss = test_loss + ((1 / (batch_idx + 1)) * (loss.data
         - test_loss))
                 # convert output probabilities to predicted class
                 pred = output.data.max(1, keepdim=True)[1]
                 # compare predictions to true label
                 correct += np.sum(np.squeeze(pred.eq(target.data.view_as(pr
         ed))).cpu().numpy())
                 total += data.size(0)

             print('Test Loss: {:.6f}\n'.format(test_loss))

             print('\nTest Accuracy: %2d%% (%2d/%2d)' % (
                 100. * correct / total, correct, total))

         # call test function
         test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.199255


Test Accuracy: 22% (186/836)
```

---

## Step 4: Create a CNN to Classify Dog Breeds (using Transfer Learning)

You will now use transfer learning to create a CNN that can identify dog breed from images. Your CNN must attain at least 60% accuracy on the test set.

### (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

Use the code cell below to write three separate data loaders (http://pytorch.org/docs/master/data.html#torch.utils.data.DataLoader) for the training, validation, and test datasets of dog images (located at `dogImages/train` , `dogImages/valid` , and `dogImages/test` , respectively).

If you like, **you are welcome to use the same data loaders from the previous step**, when you created a CNN from scratch.

In [45]:
```
## TODO: Specify data loaders
loaders_transfer = loaders_scratch.copy()
```

### (IMPLEMENTATION) Model Architecture

Use transfer learning to create a CNN to classify dog breed. Use the code cell below, and save your initialized model as the variable `model_transfer` .

In [46]:
```
import torchvision.models as models
import torch.nn as nn

## TODO: Specify model architecture
model_transfer = models.resnet50(pretrained=True)

for param in model_transfer.parameters():
    param.requires_grad = False

model_transfer.fc = nn.Linear(2048, 133, bias=True)

fc_parameters = model_transfer.fc.parameters()

for param in fc_parameters:
    param.requires_grad = True

if use_cuda:
    model_transfer = model_transfer.cuda()
```

**Question 5:** Outline the steps you took to get to your final CNN architecture and your reasoning at each step. Describe why you think the architecture is suitable for the current problem.

**Answer:** I picked ResNet as a transfer model because it performed outstanding on Image Classification. I looked into the structure and functions of ResNet. The core idea of ResNet is introducing a so-called "identity shortcut connection" that skips one or more layers. I guess this prevents overfitting when it's training.

## (IMPLEMENTATION) Specify Loss Function and Optimizer

Use the next code cell to specify a loss function (http://pytorch.org/docs/master/nn.html#loss-functions) and optimizer (http://pytorch.org/docs/master/optim.html). Save the chosen loss function as `criterion_transfer`, and the optimizer as `optimizer_transfer` below.

In [47]:
```
criterion_transfer = nn.CrossEntropyLoss()
optimizer_transfer = optim.SGD(model_transfer.fc.parameters(), lr=0
.001)
```

## (IMPLEMENTATION) Train and Validate the Model

Train and validate your model in the code cell below. Save the final model parameters (http://pytorch.org/docs/master/notes/serialization.html) at filepath `'model_transfer.pt'`.

In [58]:
```
def train(n_epochs, loaders, model, optimizer, criterion, use_cuda,
save_path):
    """returns trained model"""
    # initialize tracker for minimum validation loss
    valid_loss_min = np.Inf

    for epoch in range(1, n_epochs+1):
        # initialize variables to monitor training and validation l
oss
        train_loss = 0.0
        valid_loss = 0.0

        ###################
        # train the model #
        ###################
        model.train()
        for batch_idx, (data, target) in enumerate(loaders['train']
):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            # initialize weights to zero
            optimizer.zero_grad()

            output = model(data)

            # calculate loss
            loss = criterion(output, target)

            # back prop
            loss.backward()

            # grad
            optimizer.step()

            train_loss = train_loss + ((1 / (batch_idx + 1)) * (los
s.data - train_loss))

            if batch_idx % 100 == 0:
                print('Epoch %d, Batch %d loss: %.6f' %
                  (epoch, batch_idx + 1, train_loss))

        ######################
        # validate the model #
        ######################
        model.eval()
        for batch_idx, (data, target) in enumerate(loaders['valid']
):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## update the average validation loss
            output = model(data)
            loss = criterion(output, target)
            valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (los
s.data - valid_loss))


        # print training/validation statistics
        print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss:
{:.6f}'.format(
            epoch,
            train_loss,
            valid_loss
            ))

        ## TODO: save the model if validation loss has decreased
        if valid_loss < valid_loss_min:
            torch.save(model.state_dict(), save_path)
            print('Validation loss decreased ({:.6f} --> {:.6f}).
Saving model ...'.format(
            valid_loss_min,
            valid_loss))
            valid_loss_min = valid_loss
```

```
    # return trained model
    return model
```

In [59]:
```
# train the model
train(20, loaders_transfer, model_transfer, optimizer_transfer,
      criterion_transfer, use_cuda, 'model_transfer.pt')
```

```
Epoch 1, Batch 1 loss: 2.258956
Epoch 1, Batch 101 loss: 2.593708
Epoch 1, Batch 201 loss: 2.608153
Epoch 1, Batch 301 loss: 2.588351
Epoch: 1        Training Loss: 2.592503         Validation Loss: 2
.894483
Validation loss decreased (inf --> 2.894483).  Saving model ...
Epoch 2, Batch 1 loss: 2.203717
Epoch 2, Batch 101 loss: 2.586466
Epoch 2, Batch 201 loss: 2.631727
Epoch 2, Batch 301 loss: 2.615519
Epoch: 2        Training Loss: 2.610683         Validation Loss: 2
.894483
Epoch 3, Batch 1 loss: 2.913703
Epoch 3, Batch 101 loss: 2.607636
Epoch 3, Batch 201 loss: 2.586197
Epoch 3, Batch 301 loss: 2.581142
Epoch: 3        Training Loss: 2.581229         Validation Loss: 2
.894483
Epoch 4, Batch 1 loss: 3.067653
Epoch 4, Batch 101 loss: 2.584091
Epoch 4, Batch 201 loss: 2.594370
Epoch 4, Batch 301 loss: 2.581534
Epoch: 4        Training Loss: 2.573230         Validation Loss: 2
.894483
Epoch 5, Batch 1 loss: 2.870629
Epoch 5, Batch 101 loss: 2.647899
Epoch 5, Batch 201 loss: 2.595079
Epoch 5, Batch 301 loss: 2.584820
Epoch: 5        Training Loss: 2.575158         Validation Loss: 2
.894483
Epoch 6, Batch 1 loss: 2.528484
Epoch 6, Batch 101 loss: 2.575885
Epoch 6, Batch 201 loss: 2.567128
Epoch 6, Batch 301 loss: 2.560346
Epoch: 6        Training Loss: 2.560216         Validation Loss: 2
.894483
Epoch 7, Batch 1 loss: 2.490053
Epoch 7, Batch 101 loss: 2.570355
Epoch 7, Batch 201 loss: 2.581635
Epoch 7, Batch 301 loss: 2.570566
Epoch: 7        Training Loss: 2.568640         Validation Loss: 2
.894483
Epoch 8, Batch 1 loss: 2.553142
Epoch 8, Batch 101 loss: 2.558901
Epoch 8, Batch 201 loss: 2.552012
Epoch 8, Batch 301 loss: 2.539370
Epoch: 8        Training Loss: 2.539805         Validation Loss: 2
.894483
Epoch 9, Batch 1 loss: 2.191657
Epoch 9, Batch 101 loss: 2.624343
Epoch 9, Batch 201 loss: 2.591577
Epoch 9, Batch 301 loss: 2.595613
Epoch: 9        Training Loss: 2.591149         Validation Loss: 2
.894483
Epoch 10, Batch 1 loss: 1.898232
Epoch 10, Batch 101 loss: 2.623165
Epoch 10, Batch 201 loss: 2.584813
Epoch 10, Batch 301 loss: 2.583132
Epoch: 10       Training Loss: 2.582235         Validation Loss: 2
.894483
Epoch 11, Batch 1 loss: 2.923880
Epoch 11, Batch 101 loss: 2.559695
Epoch 11, Batch 201 loss: 2.566133
Epoch 11, Batch 301 loss: 2.572084
Epoch: 11       Training Loss: 2.575270         Validation Loss: 2
.894483
Epoch 12, Batch 1 loss: 2.178284
Epoch 12, Batch 101 loss: 2.574616
Epoch 12, Batch 201 loss: 2.582757
Epoch 12, Batch 301 loss: 2.594018
Epoch: 12       Training Loss: 2.594768         Validation Loss: 2
.894483
Epoch 13, Batch 1 loss: 2.679317
Epoch 13, Batch 101 loss: 2.582586
Epoch 13, Batch 201 loss: 2.605113
Epoch 13, Batch 301 loss: 2.600833
Epoch: 13       Training Loss: 2.596388         Validation Loss: 2
.894483
Epoch 14, Batch 1 loss: 2.362601
Epoch 14, Batch 101 loss: 2.566887
Epoch 14, Batch 201 loss: 2.559105
Epoch 14, Batch 301 loss: 2.570556
Epoch: 14       Training Loss: 2.568786         Validation Loss: 2
.894483
Epoch 15, Batch 1 loss: 3.644215
Epoch 15, Batch 101 loss: 2.542604
Epoch 15, Batch 201 loss: 2.573271
Epoch 15, Batch 301 loss: 2.575025
Epoch: 15       Training Loss: 2.572036         Validation Loss: 2
.894483
Epoch 16, Batch 1 loss: 2.072406
Epoch 16, Batch 101 loss: 2.610687
Epoch 16, Batch 201 loss: 2.581779
Epoch 16, Batch 301 loss: 2.586235
Epoch: 16       Training Loss: 2.587819         Validation Loss: 2
.894483
Epoch 17, Batch 1 loss: 2.931309
Epoch 17, Batch 101 loss: 2.581118
Epoch 17, Batch 201 loss: 2.569013
Epoch 17, Batch 301 loss: 2.557482
Epoch: 17       Training Loss: 2.550767         Validation Loss: 2
.894483
Epoch 18, Batch 1 loss: 2.764253
Epoch 18, Batch 101 loss: 2.541946
Epoch 18, Batch 201 loss: 2.543193
Epoch 18, Batch 301 loss: 2.566988
Epoch: 18       Training Loss: 2.570236         Validation Loss: 2
.894483
Epoch 19, Batch 1 loss: 2.629054
Epoch 19, Batch 101 loss: 2.606398
Epoch 19, Batch 201 loss: 2.579588
Epoch 19, Batch 301 loss: 2.582251
Epoch: 19       Training Loss: 2.573852         Validation Loss: 2
.894483
Epoch 20, Batch 1 loss: 2.620398
Epoch 20, Batch 101 loss: 2.590361
Epoch 20, Batch 201 loss: 2.593161
Epoch 20, Batch 301 loss: 2.569988
Epoch: 20       Training Loss: 2.573960         Validation Loss: 2
.894483
```

Out[59]:

```
Net(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2), paddin
g=(1, 1))
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), paddi
ng=(1, 1))
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1
, ceil_mode=False)
  (fc1): Linear(in_features=6272, out_features=500, bias=True)
  (fc2): Linear(in_features=500, out_features=133, bias=True)
  (dropout): Dropout(p=0.3, inplace=False)
)
```

In [60]:

```
# load the model that got the best validation accuracy
model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

Out[60]:

```
<All keys matched successfully>
```

## (IMPLEMENTATION) Test the Model

Try out your model on the test dataset of dog images. Use the code cell below to calculate and print the test loss and accuracy. Ensure that your test accuracy is greater than 20%.

In [61]:
```
test(loaders_transfer, model_transfer, criterion_transfer, use_cuda
)
```

```
Test Loss: 3.222744


Test Accuracy: 24% (203/836)
```

## (IMPLEMENTATION) Predict Dog Breed with the Model

Write a function that takes an image path as input and returns the dog breed ( `Affenpinscher` , `Afghan hound` , etc) that is predicted by your model.

In [62]:
```
### TODO: Write a function that takes a path to an image as input
### and returns the dog breed that is predicted by the model.

# list of class names by index, i.e. a name can be accessed like cl
ass_names[0]
class_names = [item[4:].replace("_", " ") for item in loaders_trans
fer['train'].dataset.classes]
```

In [63]:
```
from PIL import Image
import torchvision.transforms as transforms

def load_input_image(img_path):
    image = Image.open(img_path).convert('RGB')
    prediction_transform = transforms.Compose([transforms.Resize(si
ze=(224, 224)),
                                     transforms.ToTensor(),
                                     standard_normalization])

    # discard the transparent, alpha channel (that's the :3) and ad
d the batch dimension
    image = prediction_transform(image)[:3,:,:].unsqueeze(0)
    return image
```

In [64]:
```
def predict_breed_transfer(model, class_names, img_path):
    # load the image and return the predicted breed
    img = load_input_image(img_path)
    model = model.cpu()
    model.eval()
    idx = torch.argmax(model(img))
    return class_names[idx]
```

In [65]:
```
for img_file in os.listdir('./images'):
    img_path = os.path.join('./images', img_file)
    predition = predict_breed_transfer(model_transfer, class_names,
img_path)
    print("image_file_name: {0}, \t predition breed: {1}".format(im
g_path, predition))
```

```
image_file_name: ./images/Labrador_retriever_06449.jpg,          p
redition breed: Labrador retriever
image_file_name: ./images/sample_cnn.png,        predition breed:
Lowchen
image_file_name: ./images/sample_human_output.png,       predition
breed: Anatolian shepherd dog
image_file_name: ./images/American_water_spaniel_00648.jpg,      p
redition breed: Boykin spaniel
image_file_name: ./images/Brittany_02625.jpg,    predition breed:
Brittany
image_file_name: ./images/Labrador_retriever_06455.jpg,          p
redition breed: Labrador retriever
image_file_name: ./images/Labrador_retriever_06457.jpg,          p
redition breed: Labrador retriever
image_file_name: ./images/sample_dog_output.png,         predition
breed: Boston terrier
image_file_name: ./images/Welsh_springer_spaniel_08203.jpg,      p
redition breed: Welsh springer spaniel
image_file_name: ./images/Curly-coated_retriever_03896.jpg,      p
redition breed: Curly-coated retriever
```

## Step 5: Write your Algorithm

Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then,

- if a **dog** is detected in the image, return the predicted breed.
- if a **human** is detected in the image, return the resembling dog breed.
- if **neither** is detected in the image, provide output that indicates an error.

You are welcome to write your own functions for detecting humans and dogs in images, but feel free to use the `face_detector` and `dog_detector` functions developed above. You are **required** to use your CNN from Step 4 to predict dog breed.

Some sample output for our algorithm is provided below, but feel free to design your own user experience!

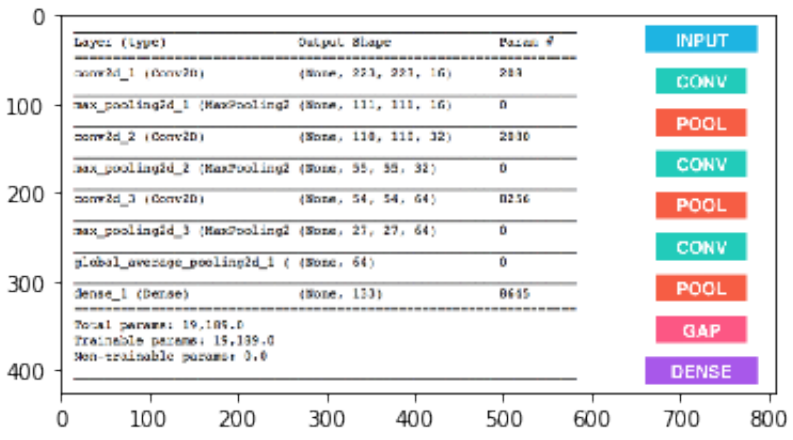
Sample Human Output

### (IMPLEMENTATION) Write your Algorithm

In [66]:
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.

def run_app(img_path):
    ## handle cases for a human face, dog, and neither
    img = Image.open(img_path)
    plt.imshow(img)
    plt.show()
    if dog_detector(img_path) is True:
        prediction = predict_breed_transfer(model_transfer, class_n
ames, img_path)
        print("Dogs Detected!\nIt looks like a {0}".format(predicti
on))
    elif face_detector(img_path) > 0:
        prediction = predict_breed_transfer(model_transfer, class_n
ames, img_path)
        print("Hello, human!\nIf you were a dog..You may look like
a {0}".format(prediction))
    else:
        print("Error! Can't detect anything..")
```

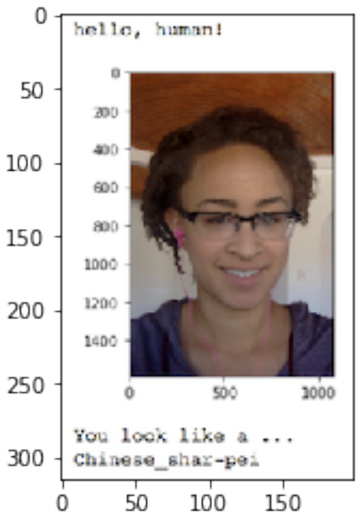
In [67]:
```
for img_file in os.listdir('./images'):
    img_path = os.path.join('./images', img_file)
    run_app(img_path)
```

```
Dogs Detected!
It looks like a Labrador retriever
```



```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 223, 223, 16)      208
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 111, 111, 16)      0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 110, 110, 32)      2080
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 55, 55, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 54, 54, 64)        8256
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 27, 27, 64)        0
_________________________________________________________________
global_average_pooling2d_1 ( (None, 64)                0
_________________________________________________________________
dense_1 (Dense)              (None, 133)               8645
=================================================================
Total params: 19,189.0
Trainable params: 19,189.0
Non-trainable params: 0.0
_________________________________________________________________
```

```
Error! Can't detect anything..
```

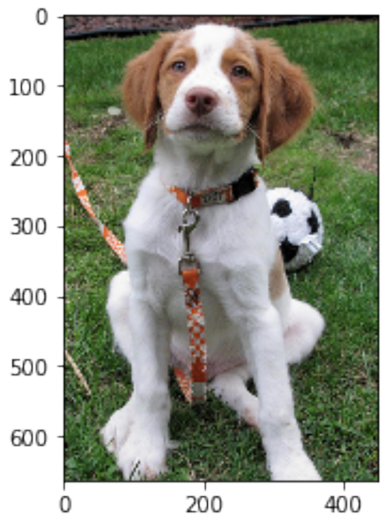


hello, human!

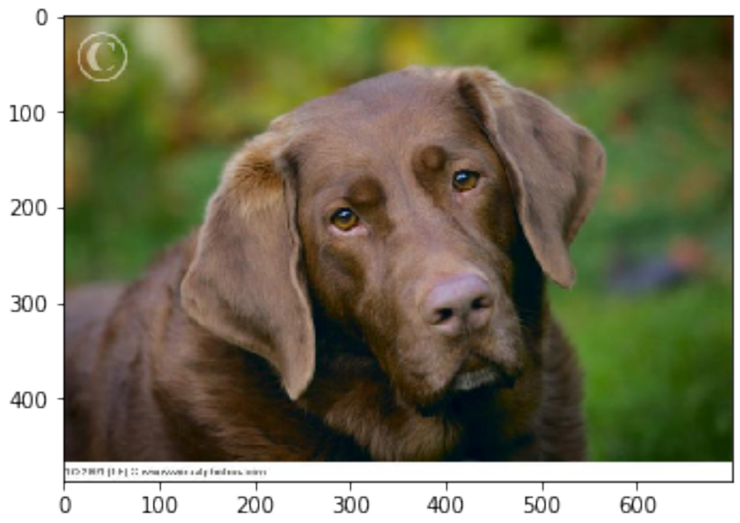
You look like a ...
Chinese_shar-pei

```
Hello, human!
If you were a dog..You may look like a Anatolian shepherd dog
```

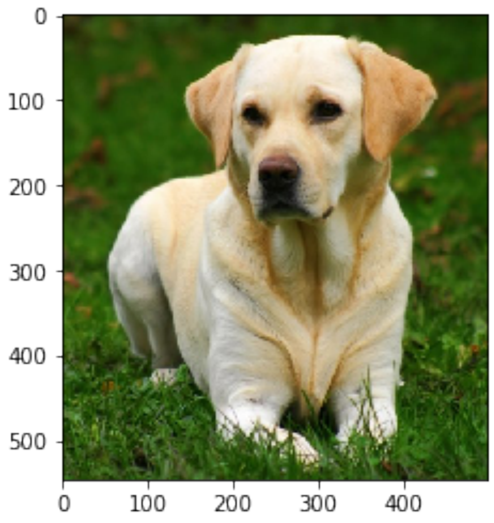
```
Dogs Detected!
It looks like a Boykin spaniel
```



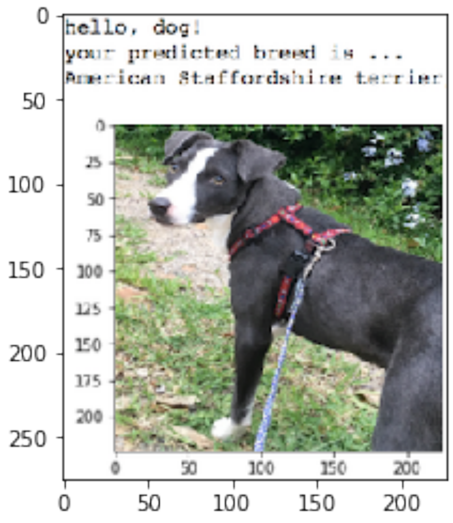
```
Dogs Detected!
It looks like a Brittany
```



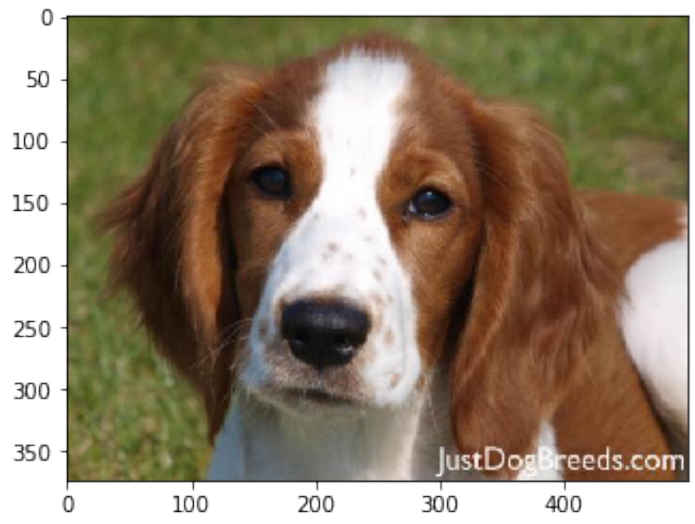
```
Dogs Detected!
It looks like a Labrador retriever
```



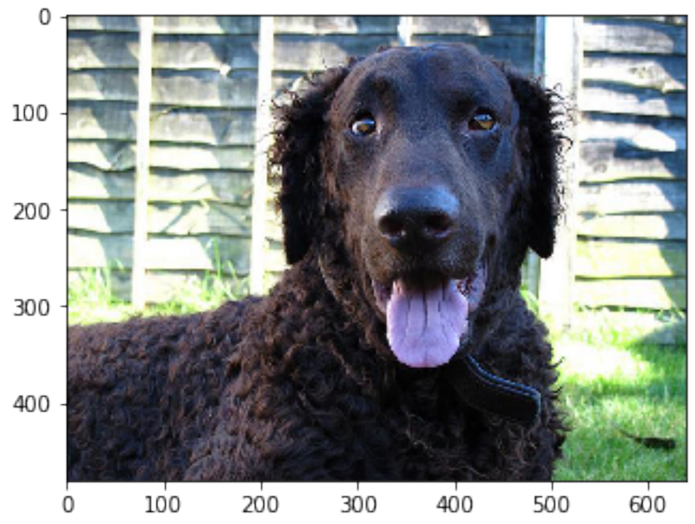
```
Dogs Detected!
It looks like a Labrador retriever
```



```
Dogs Detected!
It looks like a Boston terrier
```



```
Dogs Detected!
It looks like a Welsh springer spaniel
```



```
Dogs Detected!
It looks like a Curly-coated retriever
```

## Step 6: Test Your Algorithm

In this section, you will take your new algorithm for a spin! What kind of dog does the algorithm think that *you* look like? If you have a dog, does it predict your dog's breed accurately? If you have a cat, does it mistakenly think that your cat is a dog?

### (IMPLEMENTATION) Test Your Algorithm on Sample Images!

Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

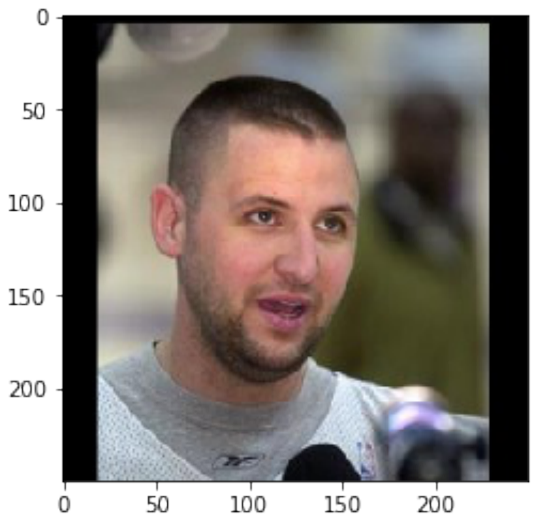
**Question 6:** Is the output better than you expected :) ? Or worse :( ? Provide at least three possible points of improvement for your algorithm.

**Answer:** (Three possible points for improvement) More image datasets of dogs will improve training models. Also, more image augmentations trials (flipping vertically, move left or right, etc.) will improve performance on test data.

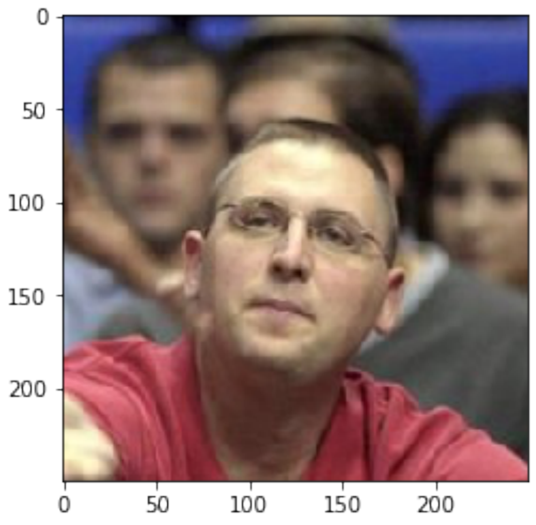
Hyper-parameter tunings: weight initializings, learning rates, drop-outs, batch_sizes, and optimizers will be helpful to improve performances.

Ensembles of models

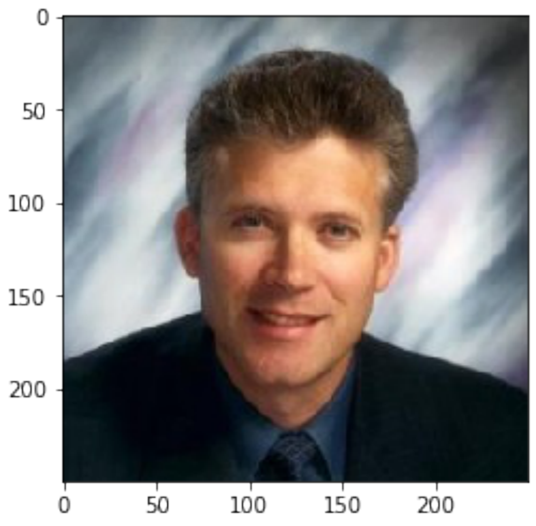
In [68]:
```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.

my_human_files = ['./my_images/human_1.jpg', './my_images/human_2.jpg', './my_images/human_3.jpg' ]
my_dog_files = ['./my_images/dog_shiba.jpeg', './my_images/dog_yorkshire.jpg', './my_images/dog_retreiver.jpg']

## suggested code, below
for file in np.hstack((human_files[:3], dog_files[:3])):
    run_app(file)
```



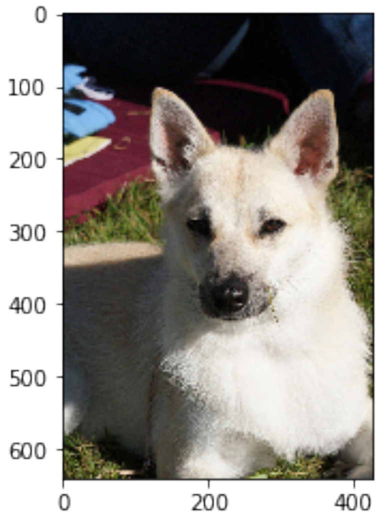
```
Hello, human!
If you were a dog..You may look like a Bulldog
```



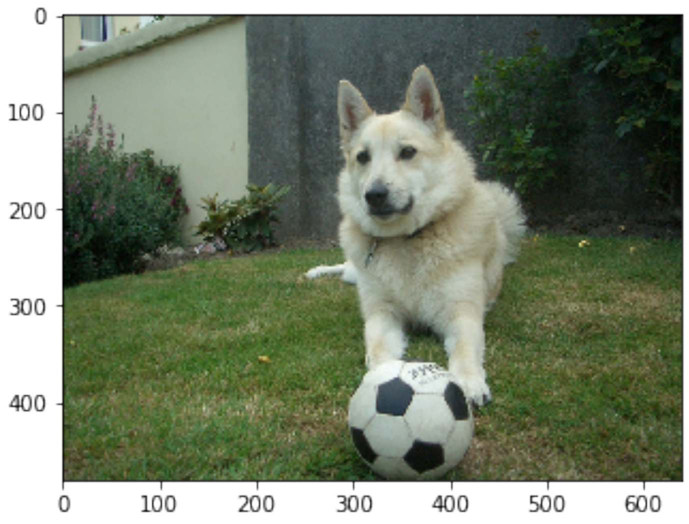
```
Hello, human!
If you were a dog..You may look like a Havanese
```



```
Hello, human!
If you were a dog..You may look like a American staffordshire terr
ier
```



```
Dogs Detected!
It looks like a American eskimo dog
```



```
Dogs Detected!
It looks like a Kuvasz
```

```
Error! Can't detect anything..
```

In [ ]:

```

```

In [ ]:

```

```