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
- Step 1: Detect Humans
- Step 2: Detect Dogs
- Step 3: Create a CNN to Classify Dog Breeds (from Scratch)
- Step 4: Create a CNN to Classify Dog Breeds (using Transfer Learning)
- Step 5: Write your Algorithm
- Step 6: Test Your Algorithm

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 [4]:
```
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.
```

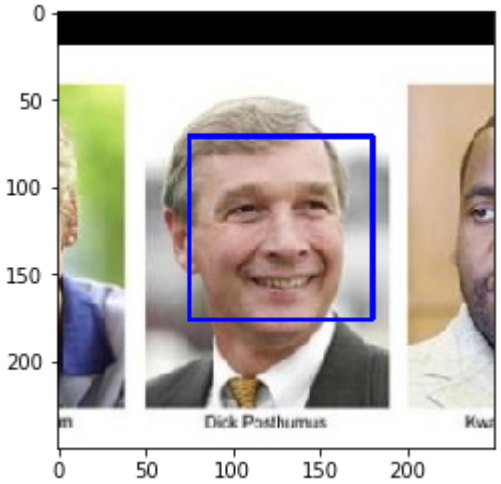
Step 1: Detect Humans

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 [5]:
```
import cv2
import matplotlib.pyplot as plt
%matplotlib inline

# extract pre-trained face detector
face_cascade = cv2.CascadeClassifier('haarcascades/haarcascade_frontalface_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 [6]:
```
# 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 [7]:
```
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 face_detection_classifier(imgs):
    detected = 0
    total = len(imgs)
    for i in imgs:
        detected += face_detector(i)
    percent = str(detected/total)
    return detected ,total, percent
```

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.
#test using dog/human_files_short

print("detected human faces in human_files_short: {}/{} which is {}".format(fa
ce_detection_classifier(human_files_short)[0],face_detection_classifier(human_
files_short)[1],face_detection_classifier(human_files_short)[2]))
print("detected human faces in dog_files_short: {}/{} which is {}".format(face
_detection_classifier(dog_files_short)[0],face_detection_classifier(dog_files_
short)[1],face_detection_classifier(dog_files_short)[2]))
```

```
detected human faces in human_files_short: 98/100 which is 0.98
detected human faces in dog_files_short: 13/100 which is 0.13
```

Step 2: Detect Dogs

In this section, we use a pre-trained model (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/), 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).

In [8]:
```
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()
```

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 [9]:
```
from PIL import Image
import torchvision.transforms as t

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


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
    #pre-process
    transform_img = t.Compose([
        t.Resize(size=(244,244)), #this is the size that vgg accepts
        t.ToTensor(),
        t.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
    ])
    img = Image.open(img_path)
    img = transform_img(img)
    #unsqueeze
    img = img.unsqueeze(0)
    if use_cuda:
        img = img.cuda()
    prediction = VGG16(img)
    #get the right index using torch.max
    prediction = torch.max(prediction,1)[1].item()

    return prediction # predicted class index
```

In [10]:
```
#testing VGG_predict
VGG16_predict(human_files_short[1])
```

Out[10]: 743

(IMPLEMENTATION) Write a Dog Detector

While looking at the dictionary (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 [11]:
```
### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    ## TODO: Complete the function.
    value = VGG16_predict(img_path)
    if value >= 151 and value <= 268:
        result = True
    else:
        result = False
    return result # true/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 [10]:
```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
def dog_detection_test(imgs):
    total = len(imgs)
    detected = 0
    for i in imgs:
        detected += dog_detector(i)
    percent = str(detected/total)
    return detected,total,percent
```

In [11]:
```
print("detected dogs in human_files_short: {} / {} which is {}".format(dog_det
ection_test(human_files_short)[0],dog_detection_test(human_files_short)[1],dog
_detection_test(human_files_short)[2]))
print("detected dogs in dog_files_short: {} / {} which is {}".format(dog_detec
tion_test(dog_files_short)[0], dog_detection_test(dog_files_short)[1],dog_dete
ction_test(dog_files_short)[2]))
```

```
detected dogs in human_files_short: 0 / 100 which is 0.0
detected dogs in dog_files_short: 100 / 100 which is 1.0
```

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 [12]:
```
### (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)](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)](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)](http://pytorch.org/docs/stable/torchvision/transforms.html?highlight=transform)!

In [33]:
```
import os
from torchvision import datasets
import torchvision.transforms as t
### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes
batch_size = 15
num_workers= 0
train_dir = 'dogImages/train/'
val_dir = 'dogImages/valid/'
test_dir = 'dogImages/test/'


#normalize values for pre-processing
normalization = t.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.2
25])

#transformation for data
#apply random/transformations to train for variability
data_transforms = {'train': t.Compose([t.Resize(256),
                                       t.RandomResizedCrop(224),
                                       t.RandomHorizontalFlip(),
                                       t.ToTensor(),
                                       normalization]),
                   'val': t.Compose([t.Resize(256),
                                     t.CenterCrop(224),
                                     t.ToTensor(),
                                     normalization]),
                   'test': t.Compose([t.Resize(256),
                                      t.CenterCrop(224),
                                      t.ToTensor(),
                                      normalization])
                  }

#load dataset
train_data =  datasets.ImageFolder(train_dir, transform=data_transforms['trai
n'])
val_data = datasets.ImageFolder(val_dir, transform=data_transforms['val'])
test_data = datasets.ImageFolder(test_dir, transform = data_transforms['test'
])

train_loader= torch.utils.data.DataLoader(train_data,
                                          batch_size=batch_size,
                                          num_workers=num_workers,
                                          shuffle=True)
val_loader = torch.utils.data.DataLoader(val_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)


#scratch loaders
loaders_scratch = {
    'train' : train_loader,
```

```
    'val' : val_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 resized/augmented the images for train by using torchvision's RandomResizedCrop and RandomHorizontalFlip which allows for both img augmentation/resizing as well as an element of randomness. I picked resize size 256 and center cropped it to be 224 and normalized it to fulfill the req's for processing. For validation/test data I didn't use augmentation.

(IMPLEMENTATION) Model Architecture

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

In [34]:
```
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
        #convolutional layers
        #3-256
        self.conv1 = nn.Conv2d(3,16,3, stride = 2)
        self.conv2 = nn.Conv2d(16,32,3, stride = 2)
        self.conv3 = nn.Conv2d(32,64,3, stride = 2)
        self.conv4 = nn.Conv2d(64,128,3, stride = 2)
        self.conv5 = nn.Conv2d(128,256,3)

        #poolsection
        self.pooling = nn.MaxPool2d(2,2)
        #fully-connected section
        self.fc1 = nn.Linear(256*11*11, 500)
        self.fc2 = nn.Linear(500, 133)  #133 being the number of dog breeds
        #drop-out
        self.dropout = nn.Dropout(0.50)

    def forward(self, x):
        ## Define forward behavior
        x = F.relu(self.conv1(x))
        self.pooling(x)
        x = F.relu(self.conv2(x))
        self.pooling(x)
        x = F.relu(self.conv3(x))
        self.pooling(x)
        x = F.relu(self.conv4(x))
        self.pooling(x)
        x = F.relu(self.conv5(x))
        self.pooling(x)
        #print(x.shape)
        x = x.view(-1, 256*11*11)

        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()
```

```
model_scratch
```

Out[34]:
```
Net(
  (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(2, 2))
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(2, 2))
  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2))
  (conv4): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2))
  (conv5): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1))
  (pooling): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_m
ode=False)
  (fc1): Linear(in_features=30976, out_features=500, bias=True)
  (fc2): Linear(in_features=500, out_features=133, bias=True)
  (dropout): Dropout(p=0.5)
)
```

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

Answer:

- Steps:
  - conv1: relu
  - pool
  - conv2: relu
  - pool
  - conv3: relu
  - pool
  - conv4: relu
  - pool
  - conv5: relu
  - pool
  - dropout
  - fc1

  So i first apply 5 convolutional layers with kernel_size (3,3) and stride (1,1) with relu activation while pooling after each one until the final pooling with stride(2,2). Then I flatten the image. After, I apply a dropout rate of 0.5 to prevent overfitting. Then the fully connected layers are established which would produce a final output size which I would use to predict the breed.

(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) and optimizer (http://pytorch.org/docs/stable/optim.html). Save the chosen loss function as `criterion_scratch`, and the optimizer as `optimizer_scratch` below.

In [35]:
```
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) at filepath `'model_scratch.pt'`.

In [32]:
```
# the following import is required for training to be robust to truncated imag
es
import numpy as np
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

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 loss
        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 accordingly
            ## record the average training loss, using something like
            ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data -
train_loss))
            optimizer.zero_grad()

            output = model(data)

            #loss
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()

            #train_loss
            train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - tr
ain_loss))

        ######################
        # validate the model #
        ######################
        model.eval()
        for batch_idx, (data, target) in enumerate(loaders['val']):
            # 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)) * (loss.data - va
lid_loss))
```

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

        ## TODO: save the model if validation loss has decreased
        if valid_loss < valid_loss_min:
            print("Validation Loss Decreased:{:.6f} --> {:.6f} ".format(valid_
loss_min, valid_loss))
            valid_loss_min = valid_loss
    # return trained model
    return model


# train the model
model_scratch = train(10, loaders_scratch, model_scratch, optimizer_scratch, c
riterion_scratch, use_cuda, 'model_scratch.pt')

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

```
Epoch: 1        Training Loss: 4.878642         Validation Loss: 4.842424
Validation Loss Decreased:inf --> 4.842424
Epoch: 2        Training Loss: 4.838826         Validation Loss: 4.768451
Validation Loss Decreased:4.842424 --> 4.768451
Epoch: 3        Training Loss: 4.775637         Validation Loss: 4.823860
Epoch: 4        Training Loss: 4.736715         Validation Loss: 4.665526
Validation Loss Decreased:4.768451 --> 4.665526
Epoch: 5        Training Loss: 4.665613         Validation Loss: 4.504564
Validation Loss Decreased:4.665526 --> 4.504564
Epoch: 6        Training Loss: 4.603973         Validation Loss: 4.487420
Validation Loss Decreased:4.504564 --> 4.487420
Epoch: 7        Training Loss: 4.560076         Validation Loss: 4.396660
Validation Loss Decreased:4.487420 --> 4.396660
Epoch: 8        Training Loss: 4.520141         Validation Loss: 4.381711
Validation Loss Decreased:4.396660 --> 4.381711
Epoch: 9        Training Loss: 4.490795         Validation Loss: 4.314936
Validation Loss Decreased:4.381711 --> 4.314936
Epoch: 10       Training Loss: 4.442195         Validation Loss: 4.369620

---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-32-43ed915ad774> in <module>
     69
     70 # load the model that got the best validation accuracy
---> 71 model_scratch.load_state_dict(torch.load('model_scratch.pt'))

~/miniconda3/envs/deep-learning/lib/python3.7/site-packages/torch/serializati
on.py in load(f, map_location, pickle_module, **pickle_load_args)
    380             (sys.version_info[0] == 2 and isinstance(f, unicode)):
    381         new_fd = True
--> 382         f = open(f, 'rb')
    383     elif (sys.version_info[0] == 3 and isinstance(f, pathlib.Path)):
    384         new_fd = True

FileNotFoundError: [Errno 2] No such file or directory: 'model_scratch.pt'
```

(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 [36]:
```
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 mod
el
        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_los
s))
        # 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(pred))).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: 4.890923


Test Accuracy:  0% ( 6/836)
```

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)](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 [19]:
```
## TODO: Specify data loaders
#copy previous step's 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 [21]:
```
import torchvision.models as models
import torch.nn as nn

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

In [22]:
```
for param in model_transfer.parameters():
    param.requires_grad = False
```

In [23]:
```
model_transfer.fc = nn.Linear(128*4*4, 133, bias=True)

#fc transfer
fc_parameters = model_transfer.fc.parameters()
for param in fc_parameters:
    param.required_grad = True
model_transfer
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 decided to transfer into ResNet Architecture due to my experience in seeing it's success before. I think it's suitable for the current problem because of it's ability to avoid overfitting which might be a major issue in the current problem

(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 [24]:
```
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 [33]:
```
# train the model
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 loss
        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 accordingly
            ## record the average training loss, using something like
            ## 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)) * (loss.data - tr
ain_loss))


        ######################
        # validate the model #
        ######################
        model.eval()
        for batch_idx, (data, target) in enumerate(loaders['val']):
            # 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)) * (loss.data - va
lid_loss))


        # print training/validation statistics
        print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.fo
rmat(
            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}).'.format(
            valid_loss_min,
            valid_loss))
            valid_loss_min = valid_loss

    return model



model_transfer = train(10, loaders_transfer, model_transfer, optimizer_transfe
r, criterion_transfer, use_cuda, 'model_transfer.pt')
```

```
Epoch: 1        Training Loss: 4.802999         Validation Loss: 4.573606
Validation loss decreased (inf --> 4.573606).
Epoch: 2        Training Loss: 4.545424         Validation Loss: 4.268877
Validation loss decreased (4.573606 --> 4.268877).
Epoch: 3        Training Loss: 4.316434         Validation Loss: 3.986859
Validation loss decreased (4.268877 --> 3.986859).
Epoch: 4        Training Loss: 4.105281         Validation Loss: 3.726073
Validation loss decreased (3.986859 --> 3.726073).
Epoch: 5        Training Loss: 3.903137         Validation Loss: 3.442779
Validation loss decreased (3.726073 --> 3.442779).
Epoch: 6        Training Loss: 3.720345         Validation Loss: 3.206178
Validation loss decreased (3.442779 --> 3.206178).
Epoch: 7        Training Loss: 3.536139         Validation Loss: 3.023011
Validation loss decreased (3.206178 --> 3.023011).
Epoch: 8        Training Loss: 3.375559         Validation Loss: 2.764353
Validation loss decreased (3.023011 --> 2.764353).
Epoch: 9        Training Loss: 3.234337         Validation Loss: 2.618176
Validation loss decreased (2.764353 --> 2.618176).
Epoch: 10       Training Loss: 3.080359         Validation Loss: 2.431511
Validation loss decreased (2.618176 --> 2.431511).
```

In [25]:
```
model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

Out[25]:
```
IncompatibleKeys(missing_keys=[], unexpected_keys=[])
```

(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 60%.

In [26]:
```
test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 2.396152


Test Accuracy: 67% (561/836)
```

(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 [60]:
```
### 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 class_names
[0]
class_names = [item[4:].replace("_", " ") for item in loaders_transfer['train'
].dataset.classes]
model_transfer.load_state_dict(torch.load('model_transfer.pt'))
def predict_breed_transfer(model, img_path):
    img = Image.open(img_path).convert('RGB')
    prediction_transform = t.Compose([t.Resize(size=(224,224)),
                                      t.ToTensor(),
                                      t.Normalize(mean=[0.485, 0.456, 0.406], std=[0.22
9, 0.224, 0.225])
    ])
    #discard channels/unsqueeze
    img = prediction_transform(img)[:3,:,:].unsqueeze(0)
    model = model.cpu()
    model.eval()
    index = torch.argmax(model(img))
    return class_names[index]
```

In [61]:
```
import os
```

In [63]:
```
#returns prediction name
for img in os.listdir('./images'):
    img_path = os.path.join('./images',img)
    prediction = predict_breed_transfer(model_transfer, img_path)
    print("Prediction is: {}".format(prediction))
    print("File name is: {}".format(img_path) )
```

```
Prediction is: Irish water spaniel
File name is: ./images/American_water_spaniel_00648.jpg
Prediction is: Dogue de bordeaux
File name is: ./images/sample_human_output.png
Prediction is: Basset hound
File name is: ./images/Welsh_springer_spaniel_08203.jpg
Prediction is: Affenpinscher
File name is: ./images/sample_cnn.png
Prediction is: Golden retriever
File name is: ./images/Labrador_retriever_06457.jpg
Prediction is: Curly-coated retriever
File name is: ./images/Curly-coated_retriever_03896.jpg
Prediction is: Brittany
File name is: ./images/Brittany_02625.jpg
Prediction is: Flat-coated retriever
File name is: ./images/Labrador_retriever_06449.jpg
Prediction is: Chesapeake bay retriever
File name is: ./images/Labrador_retriever_06455.jpg
Prediction is: Italian greyhound
File name is: ./images/sample_dog_output.png
```

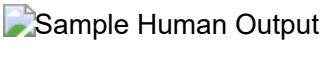
Step 5: Write your Algorithm

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.
from PIL import Image
def run_app(img_path):
    ## handle cases for a human face, dog, and neither
    img = Image.open(img_path)
    plt.imshow(img)
    plt.show()

    #conditions
    if dog_detector(img_path) is True:
        prediction = predict_breed_transfer(model_transfer, img_path)
        print("A dog has been detected which is most likely to be of this bree
d: {0} ".format(prediction))
    elif face_detector(img_path) > 0:
        prediction = predict_breed_transfer(model_transfer, img_path)
        print("Detected a human who looks like the breed: {0}".format(predicti
on))
    else:
        print("Unknown Image")
```

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!

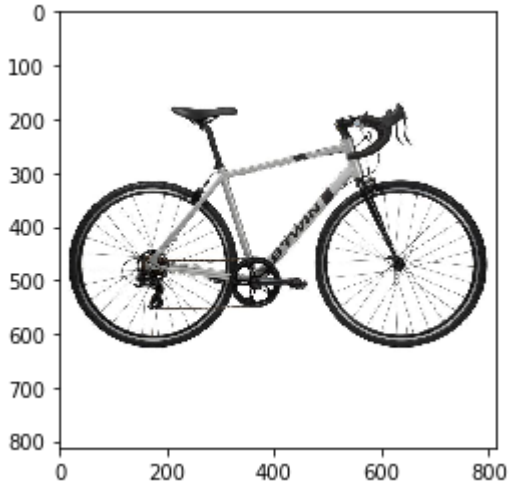
Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

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

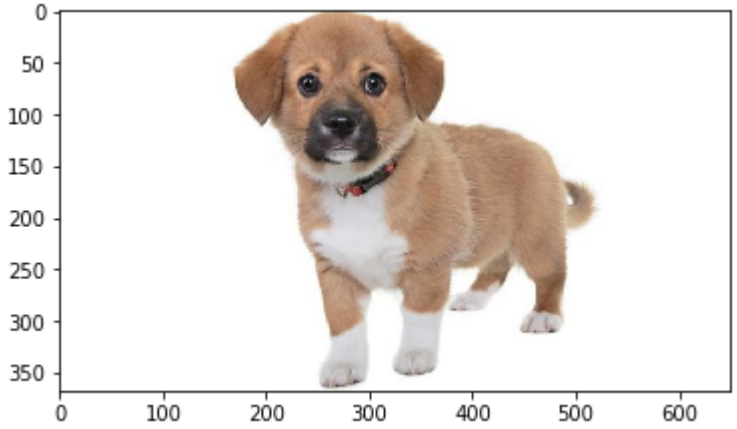
Answer: (Three possible points for improvement) The output is around what I expected. It detected the dogs/humans correctly but the breeds are not correct. Also, considering that I used a mixed breed photo as an example photo it was hard for it to detect a specific breed. One possible improvement would be to expand the datasets of dogs with their labels to provide more data which would increase performance. A second possible point would be to adjust pre-processing, drop-out rates, learning rate, number of epochs, and optimizers to get a better weight after training. Finally, a huge improvement would be to include a bunch of mixed breed datasets in order to distiguish mix-breeds while highlighting the important features of specific breeds that transfer during mix which should weigh more heavily for each specific breed.

In [71]:
```
## 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.

## suggested code, below
for image in os.listdir('./testimages'):
    test_path = os.path.join('./testimages', image)
    print("Test Image filename: {}" .format(test_path))
    run_app(test_path)
```

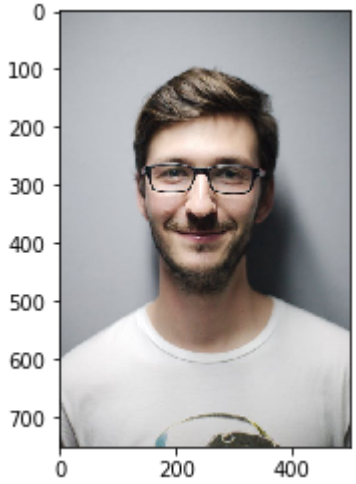
```
Test Image filename: ./testimages/bicycle.jpg
```



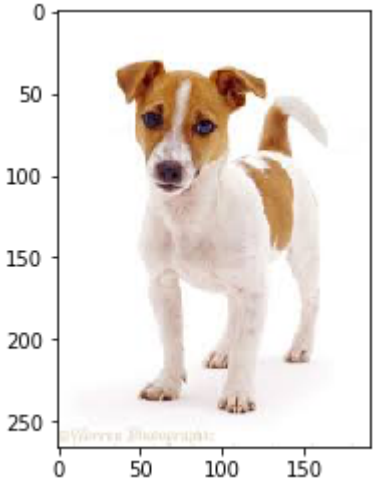
```
Unknown Image
Test Image filename: ./testimages/beaglier_mixed.jpg
```



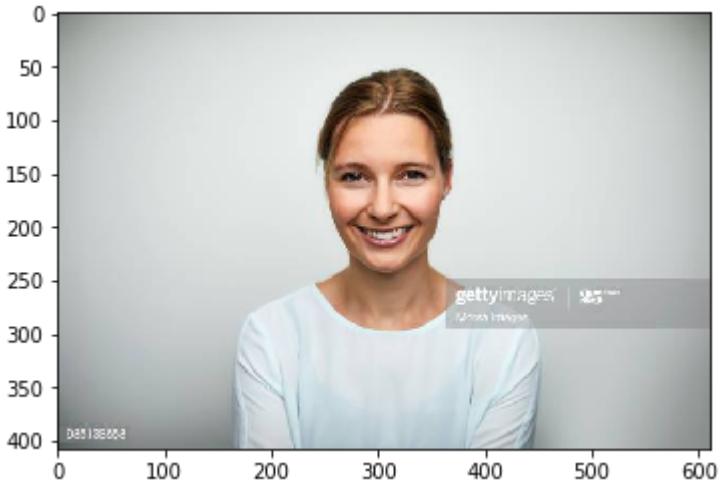
```
A dog has been detected which is most likely to be of this breed: Pembroke we
lsh corgi
Test Image filename: ./testimages/human_man.jpg
```



```
Detected a human who looks like the breed: Japanese chin
Test Image filename: ./testimages/jack_russel.jpg
```



A dog has been detected which is most likely to be of this breed: American st affordshire terrier
Test Image filename: ./testimages/human_female.jpg



Detected a human who looks like the breed: Dogue de bordeaux
Test Image filename: ./testimages/spaghetti.jpg

Unknown Image

In [ ]:

In [ ]: