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!).

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:

Note: if you are using the Udacity workspace, you *DO NOT* need to re-download these - they can be found in the `/data` folder as noted in the cell below.

- 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 `/dog_images` .
- 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]:
```
import numpy as np
from glob import glob

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

# 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.
```

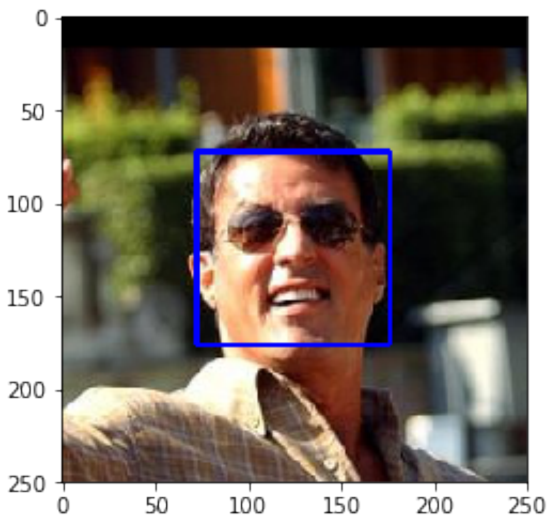
Step 1: Detect Humans

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)](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)](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_frontal
face_alt.xml')
#print (type(human_files))
# load color (BGR) image
#img = cv2.imread('C:/Users/admin/lfw/lfw/Aaron_Eckhart/Aaron_Eckhart_
0001.jpg')
#img = cv2.imread(human_files[0])
#imname = human_files[0] + '\Aaron_Eckhart_0001.jpg'
#imname = human_files[0] + '\Aaron_Eckhart_0001.jpg'

img = cv2.imread(human_files[459])
# 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.
human_faces =0
for image in human_files_short:
    if face_detector(image):
        human_faces +=1



print ("percentage of the first 100 images in human_files have a detec
ted human face")
print (human_faces)
human_in_dog_face = 0
for image in dog_files_short:
    if face_detector(image):
        human_in_dog_face +=1

#return human_faces
print ("percentage of the first 100 images in dog_files have a detecte
d human face")
print (human_in_dog_face)
```

```
percentage of the first 100 images in human_files have a detected hu
man face
98
percentage of the first 100 images in dog_files have a detected huma
n face
17
```

In [ ]:

In [5]:
```
#print (type(human_files_short))
```

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 [6]:
```
### (Optional)
### TODO: Test performance of anotherface detection algorithm.
### Feel free to use as many code cells as needed.
```

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 [7]:
```
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:
    #print ("Cuda available")
    VGG16 = VGG16.cuda()
```

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth
" to /root/.torch/models/vgg16-397923af.pth
100%|██████████| 553433881/553433881 [00:07<00:00, 72067566.52it/s]
```

In [8]:
```
print (type(VGG16))
print (use_cuda)
#print (human_files_short.device)
```

```
<class 'torchvision.models.vgg.VGG'>
True
```

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)](http://pytorch.org/docs/stable/torchvision/models.html).

In [9]:
```
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 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
    '''
    transform to size 224x224, normalize
    '''
    # Open image
    dog_img = Image.open(img_path)
    # Tranforms
    transform = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
```

```
        transforms.ToTensor(),
        transforms.Normalize(

            mean=[0.485, 0.456, 0.406],
            std=[0.229, 0.224, 0.225]

    )])

    img_transformed = transform(dog_img)

    #Define a batch to be passed through the network
    batch_t = torch.unsqueeze(img_transformed, 0)
    # move the data to cuda if available

    if torch.cuda.is_available():
        #print ("cuda available")
        batch_t = batch_t.cuda()


    VGG16.eval()
    output = VGG16(batch_t)
    #print (output.shape)
    max_val, index = torch.max(output,1)
    #index where the maximum score in output vector out occurs
    return index



#img = VGG16_predict(dog_files[200])
#print (img)
```

(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]:
```
img = VGG16_predict(dog_files[200])
print (img)
```

```
tensor([ 169], device='cuda:0')
```

In [11]:
```
### returns "True" if a dog is detected in the image stored at img_pat
h
def dog_detector(img_path):
    ## TODO: Complete the function.
    predicted_index = VGG16_predict(img_path)
    if predicted_index >= 151 and  predicted_index <= 268:
        return True
    return 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.
#print (type(human_files_short))
human_faces =0
for image in human_files_short:
    if dog_detector(image):
        human_faces +=1


print ("percentage of the images in human_files_short have a detected dog")
print (human_faces)
dog_faces = 0
for image in dog_files_short:
    if dog_detector(image):
        dog_faces +=1

#return human_faces
print ("percentage of the images in dog_files_short have a detected dog")
print (dog_faces)
```

```
percentage of the images in human_files_short have a detected dog
0
percentage of the images in dog_files_short have a detected dog
100
```

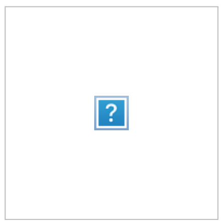
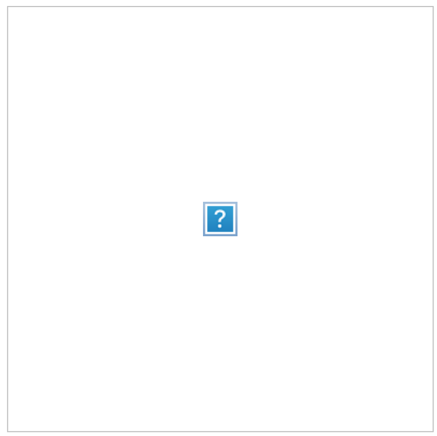
We suggest VGG-16 as a potential network to detect dog images in your algorithm, but you are free to explore other pre-trained networks (such as [Inception-v3 (http://pytorch.org/docs/master/torchvision/models.html#inception-v3)](http://pytorch.org/docs/master/torchvision/models.html#inception-v3), [ResNet-50 (http://pytorch.org/docs/master/torchvision/models.html#id3)](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 [13]:
```
### (Optional)
### TODO: Report the performance of another pre-trained network.
### Feel free to use as many code cells as needed.
```

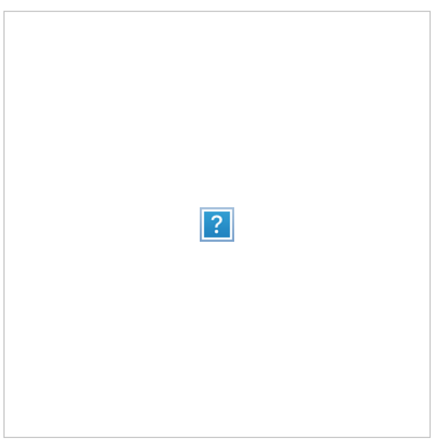
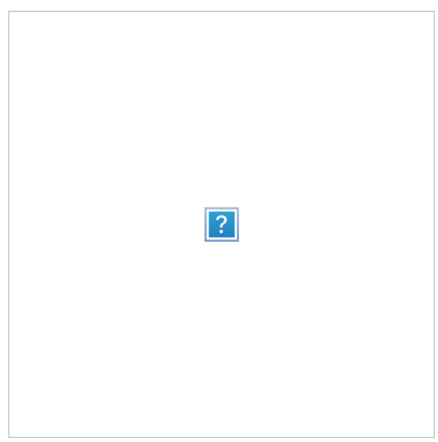
Step 3: Create a CNN to Classify Dog Breeds (from Scratch)

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 `dog_images/train`, `dog_images/valid`, and `dog_images/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 [14]:
```
import os
from torchvision import datasets

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

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

train_dir = '/data/dog_images/train/'
test_dir = '/data/dog_images/test/'
```

```
valid_dir = '/data/dog_images/valid/'



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

data_transform_train = transforms.Compose([transforms.RandomResizedCro
p(224),
                                    transforms.RandomHorizontalFlip()
, # randomly flip and rotate
                                    transforms.RandomRotation(10),
                                    transforms.ToTensor(),
                                    transforms.Normalize((0.485, 0.45
6, 0.406), (0.229, 0.224, 0.225))])
data_transform_others = transforms.Compose([transforms.RandomResizedCr
op(224),
                                    transforms.ToTensor(),
                                    transforms.Normalize((0.485, 0.45
6, 0.406), (0.229, 0.224, 0.225))])

# Data set directories
train_data = datasets.ImageFolder(train_dir, transform=data_transform_
train)
test_data = datasets.ImageFolder(test_dir, transform=data_transform_ot
hers)
valid_data = datasets.ImageFolder(valid_dir, transform=data_transform_
others)
# print out some data stats
print('Num training images: ', len(train_data))
print('Num test images: ', len(test_data))
#print (type(train_data))

batch_size = 20
num_workers=0

# prepare data loaders
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batc
h_size,
                                          num_workers=num_workers, sh
uffle=True)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_
size,
                                         num_workers=num_workers, shu
ffle=True)
valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=batc
h_size,
                                         num_workers=num_workers, shu
ffle=True)
```

```
loaders_scratch = {
    'train': train_loader,
    'valid': valid_loader,
    'test': test_loader
}

# get the classes or dog breeds
classes = list(np.array(glob("/data/dog_images/train/*")))
#print (classes)
```

```
Num training images:  6680
Num test images:  836
```

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 have cropped the images to size of 224 and it is also the size of tensor, for two reasons: a) As taught in udacity classroom, we need the input images of fixed size for a CNN to work. Secondly, as i researched for VGG16, I found that it takes input image of 224. Based on the philosophy of VGG16, I cropped my images to 224 size. I have augmented the training dataset by randomly flipping and rotating the images in the dataset

(IMPLEMENTATION) Model Architecture

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

In [15]:
```
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 layer
        self.conv1 = nn.Conv2d(3, 32, 3, stride = 2, padding=1)
        self.conv2 = nn.Conv2d(32, 64, 3, padding=1)
        self.conv3 = nn.Conv2d(64, 128, 3, padding=1)
```

```
        #self.fc1 = nn.Linear(64*28*28, 512)
        self.fc1 = nn.Linear(128*14*14, 512)
        self.fc2 = nn.Linear(512, 133)
        # max pooling layer
        self.pool = nn.MaxPool2d(2, 2)
        # Dropout
        self.dropout = nn.Dropout(0.25)



    def forward(self, x):
        ## Define forward behavior
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        # flatten image input
        x = x.view(-1, 128 * 14 * 14)
        # add dropout layer
        x = self.dropout(x)
        # add 1st hidden layer, with relu activation function
        x = F.relu(self.fc1(x))
        # add dropout layer
        x = self.dropout(x)
        # add 2nd hidden layer, with relu activation function
        #x = F.softmax(self.fc2(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: My CNN has 3 convulational layers and 2 linear layers. The first layer takes an input image of size 224 (as cropped earlier) and applies kernel_size of 3, stride of 2 and padding of 1 to this image. I have then applied relu activation function followed by maxpooling layer to reduce the size of image by 2. The second convulational layer increases the depth with kernel_size of 3 and number of filters = 64, stride =1 and padding =1 . After applying relu function, I have applied the maxpooling layer that further reduces the image size by 2. The last convulational layer, again has a kernel size of 3 and padding =1, but number of filters = 128, which will be the new depth of the layer. Using the formula : (original size - kernal size + 2*padding)/ stride +1, I found that the original image is reduced to 14*14*128 size. I have flattened this 3d image/ layer into a linear structure of size (128* 14 * 14) and passed it to first fully connected layer. I have added a dropout of 0.25 to prevent overfitting. 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 [16]:
```
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 [17]:
```
# the following import is required for training to be robust to trunca
ted images
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, sa
ve_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 according
ly
            optimizer.zero_grad()
            # forward pass: compute predicted outputs by passing input
s to the model
            output = model(data)
            # calculate the batch loss
            loss = criterion(output, target)
            # backward pass: compute gradient of the loss with respect
to model parameters
            loss.backward()
            # perform a single optimization step (parameter update)
            optimizer.step()

            ## record the average training loss, using something like
            ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (los
s.data - train_loss))
            train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.d
ata - train_loss))

            ## change according to batch size
            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)) * (loss.d
ata - 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}).  Sav
ing model ...'.format(
            valid_loss_min,
            valid_loss))
            valid_loss_min = valid_loss

    # return trained model
    return model
```

In [18]:

```
model_scratch = train(25, loaders_scratch, model_scratch, optimizer_sc
ratch,
                      criterion_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, Batch 1 loss: 4.887694
Epoch 1, Batch 101 loss: 4.886893
Epoch 1, Batch 201 loss: 4.883052
Epoch 1, Batch 301 loss: 4.872168
Epoch: 1 	Training Loss: 4.863156 	Validation Loss: 4.7
91080
Validation loss decreased (inf --> 4.791080).  Saving model ...
Epoch 2, Batch 1 loss: 4.945999
Epoch 2, Batch 101 loss: 4.752713
Epoch 2, Batch 201 loss: 4.714556
Epoch 2, Batch 301 loss: 4.696077
Epoch: 2 	Training Loss: 4.687455 	Validation Loss: 4.6
03589
Validation loss decreased (4.791080 --> 4.603589).  Saving model ...
Epoch 3, Batch 1 loss: 4.543184
Epoch 3, Batch 101 loss: 4.578360
```

```
Epoch 3, Batch 201 loss: 4.577922
Epoch 3, Batch 301 loss: 4.579591
Epoch: 3        Training Loss: 4.576003         Validation Loss: 4.5
46527
Validation loss decreased (4.603589 --> 4.546527).  Saving model ...
Epoch 4, Batch 1 loss: 4.517303
Epoch 4, Batch 101 loss: 4.520955
Epoch 4, Batch 201 loss: 4.518223
Epoch 4, Batch 301 loss: 4.527127
Epoch: 4        Training Loss: 4.523706         Validation Loss: 4.4
86432
Validation loss decreased (4.546527 --> 4.486432).  Saving model ...
Epoch 5, Batch 1 loss: 4.353242
Epoch 5, Batch 101 loss: 4.451839
Epoch 5, Batch 201 loss: 4.448764
Epoch 5, Batch 301 loss: 4.448919
Epoch: 5        Training Loss: 4.446542         Validation Loss: 4.3
80929
Validation loss decreased (4.486432 --> 4.380929).  Saving model ...
Epoch 6, Batch 1 loss: 4.540212
Epoch 6, Batch 101 loss: 4.365202
Epoch 6, Batch 201 loss: 4.374483
Epoch 6, Batch 301 loss: 4.375330
Epoch: 6        Training Loss: 4.381719         Validation Loss: 4.3
35485
Validation loss decreased (4.380929 --> 4.335485).  Saving model ...
Epoch 7, Batch 1 loss: 4.217647
Epoch 7, Batch 101 loss: 4.294018
Epoch 7, Batch 201 loss: 4.316713
Epoch 7, Batch 301 loss: 4.301040
Epoch: 7        Training Loss: 4.302712         Validation Loss: 4.3
96812
Epoch 8, Batch 1 loss: 4.413953
Epoch 8, Batch 101 loss: 4.275630
Epoch 8, Batch 201 loss: 4.240616
Epoch 8, Batch 301 loss: 4.247145
Epoch: 8        Training Loss: 4.244026         Validation Loss: 4.2
59802
Validation loss decreased (4.335485 --> 4.259802).  Saving model ...
Epoch 9, Batch 1 loss: 4.255514
Epoch 9, Batch 101 loss: 4.216896
Epoch 9, Batch 201 loss: 4.183490
Epoch 9, Batch 301 loss: 4.195334
Epoch: 9        Training Loss: 4.195887         Validation Loss: 4.2
08416
Validation loss decreased (4.259802 --> 4.208416).  Saving model ...
Epoch 10, Batch 1 loss: 4.305554
Epoch 10, Batch 101 loss: 4.079574
Epoch 10, Batch 201 loss: 4.095361
Epoch 10, Batch 301 loss: 4.103439
```

```
Epoch: 10       Training Loss: 4.110876         Validation Loss: 4.1
76606
Validation loss decreased (4.208416 --> 4.176606).  Saving model ...
Epoch 11, Batch 1 loss: 4.222272
Epoch 11, Batch 101 loss: 4.062510
Epoch 11, Batch 201 loss: 4.078339
Epoch 11, Batch 301 loss: 4.077611
Epoch: 11       Training Loss: 4.077302         Validation Loss: 4.1
37654
Validation loss decreased (4.176606 --> 4.137654).  Saving model ...
Epoch 12, Batch 1 loss: 4.308134
Epoch 12, Batch 101 loss: 4.011127
Epoch 12, Batch 201 loss: 4.003627
Epoch 12, Batch 301 loss: 4.010712
Epoch: 12       Training Loss: 4.020555         Validation Loss: 4.0
74533
Validation loss decreased (4.137654 --> 4.074533).  Saving model ...
Epoch 13, Batch 1 loss: 4.062015
Epoch 13, Batch 101 loss: 3.954307
Epoch 13, Batch 201 loss: 3.951986
Epoch 13, Batch 301 loss: 3.960644
Epoch: 13       Training Loss: 3.962029         Validation Loss: 4.1
00493
Epoch 14, Batch 1 loss: 3.713747
Epoch 14, Batch 101 loss: 3.918628
Epoch 14, Batch 201 loss: 3.907531
Epoch 14, Batch 301 loss: 3.902966
Epoch: 14       Training Loss: 3.904242         Validation Loss: 4.1
13522
Epoch 15, Batch 1 loss: 3.769019
Epoch 15, Batch 101 loss: 3.819653
Epoch 15, Batch 201 loss: 3.840877
Epoch 15, Batch 301 loss: 3.850784
Epoch: 15       Training Loss: 3.850793         Validation Loss: 3.9
41297
Validation loss decreased (4.074533 --> 3.941297).  Saving model ...
Epoch 16, Batch 1 loss: 3.650696
Epoch 16, Batch 101 loss: 3.832428
Epoch 16, Batch 201 loss: 3.834061
Epoch 16, Batch 301 loss: 3.840398
Epoch: 16       Training Loss: 3.835469         Validation Loss: 3.9
51683
Epoch 17, Batch 1 loss: 3.598594
Epoch 17, Batch 101 loss: 3.756478
Epoch 17, Batch 201 loss: 3.755041
Epoch 17, Batch 301 loss: 3.766078
Epoch: 17       Training Loss: 3.768604         Validation Loss: 3.9
68526
Epoch 18, Batch 1 loss: 3.740308
Epoch 18, Batch 101 loss: 3.686824
```

```
Epoch 18, Batch 201 loss: 3.707451
Epoch 18, Batch 301 loss: 3.705823
Epoch: 18       Training Loss: 3.715443         Validation Loss: 3.8
39210
Validation loss decreased (3.941297 --> 3.839210).  Saving model ...
Epoch 19, Batch 1 loss: 4.207103
Epoch 19, Batch 101 loss: 3.594044
Epoch 19, Batch 201 loss: 3.620978
Epoch 19, Batch 301 loss: 3.647491
Epoch: 19       Training Loss: 3.655019         Validation Loss: 3.9
74550
Epoch 20, Batch 1 loss: 3.110772
Epoch 20, Batch 101 loss: 3.658441
Epoch 20, Batch 201 loss: 3.640069
Epoch 20, Batch 301 loss: 3.625586
Epoch: 20       Training Loss: 3.631907         Validation Loss: 3.9
56982
Epoch 21, Batch 1 loss: 3.632023
Epoch 21, Batch 101 loss: 3.558103
Epoch 21, Batch 201 loss: 3.574190
Epoch 21, Batch 301 loss: 3.580229
Epoch: 21       Training Loss: 3.583688         Validation Loss: 3.9
41900
Epoch 22, Batch 1 loss: 3.480891
Epoch 22, Batch 101 loss: 3.478178
Epoch 22, Batch 201 loss: 3.522028
Epoch 22, Batch 301 loss: 3.528005
Epoch: 22       Training Loss: 3.526533         Validation Loss: 4.0
07714
Epoch 23, Batch 1 loss: 3.509146
Epoch 23, Batch 101 loss: 3.476574
Epoch 23, Batch 201 loss: 3.524328
Epoch 23, Batch 301 loss: 3.528250
Epoch: 23       Training Loss: 3.521207         Validation Loss: 3.8
75647
Epoch 24, Batch 1 loss: 4.068637
Epoch 24, Batch 101 loss: 3.441596
Epoch 24, Batch 201 loss: 3.481608
Epoch 24, Batch 301 loss: 3.475900
Epoch: 24       Training Loss: 3.470488         Validation Loss: 3.8
27964
Validation loss decreased (3.839210 --> 3.827964).  Saving model ...
Epoch 25, Batch 1 loss: 3.797667
Epoch 25, Batch 101 loss: 3.393915
Epoch 25, Batch 201 loss: 3.413336
Epoch 25, Batch 301 loss: 3.424859
Epoch: 25       Training Loss: 3.416065         Validation Loss: 3.8
27668
Validation loss decreased (3.827964 --> 3.827668).  Saving model ...
```

(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 [ ]:

In [19]:

```
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(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
```

In [20]:
```
test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.741652


Test Accuracy: 14% (125/836)
```

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 [21]:
```
## TODO: Specify data loaders
import os
from torchvision import datasets
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

train_dir = '/data/dog_images/train/'
test_dir = '/data/dog_images/test/'
valid_dir = '/data/dog_images/valid/'

# classes are folders in each directory with these names
#classes = ['daisy', 'dandelion', 'roses', 'sunflowers', 'tulips']

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

```
data_transform_train = transforms.Compose([transforms.RandomResizedCro
p(224),
                                     transforms.RandomHorizontalFlip()
, # randomly flip and rotate
                                     transforms.RandomRotation(10),
                                     transforms.ToTensor(),
                                     transforms.Normalize((0.5, 0.5, 0
.5), (0.5, 0.5, 0.5))])
data_transform_other = transforms.Compose([transforms.RandomResizedCro
p(224),
                                     transforms.ToTensor(),
                                     transforms.Normalize((0.5, 0.5, 0
.5), (0.5, 0.5, 0.5))])

# Data set directories
train_data = datasets.ImageFolder(train_dir, transform=data_transform_
train)
test_data = datasets.ImageFolder(test_dir, transform=data_transform_ot
her)
valid_data = datasets.ImageFolder(valid_dir, transform=data_transform_
other)
# print out some data stats
print('Num training images: ', len(train_data))
print('Num test images: ', len(test_data))
#print (type(train_data))

batch_size = 20
num_workers=0

# prepare data loaders
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batc
h_size,
                                           num_workers=num_workers, sh
uffle=True)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_
size,
                                          num_workers=num_workers, shu
ffle=True)
valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=batc
h_size,
                                          num_workers=num_workers, shu
ffle=True)

loaders_transfer={
    'train': train_loader,
    'valid': valid_loader,
    'test': test_loader

}
# get the classes or dog breeds
```

```
classes = list(np.array(glob("/data/dog_images/train/*")))
```

```
Num training images:  6680
Num test images:  836
```

(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 [22]:

```
import torchvision.models as models
import torch.nn as nn

## TODO: Specify model architecture

model_transfer = models.vgg16(pretrained=True)
if use_cuda:
    model_transfer = model_transfer.cuda()
```

In [23]:

```
print(model_transfer.classifier[6].in_features)
print(model_transfer.classifier[6].out_features)
# Freeze training for all "features" layers
for param in model_transfer.features.parameters():
    param.requires_grad = False

#Modify the last layer fully connected layer
n_inputs = model_transfer.classifier[6].in_features


# new layers automatically have requires_grad = True
last_layer = nn.Linear(n_inputs, len(classes))

model_transfer.classifier[6] = last_layer

# if GPU is available, move the model to GPU
#if train_on_gpu:
if use_cuda:
    model_transfer.cuda()

# check to see that last layer produces the expected number of outputs
print(model_transfer.classifier[6].out_features)
```

```
4096
1000
133
```

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: Using the concept of transfer learning, I have used the VGG16 model to detect dogs. Since VGG16 is already trained on ImageNet dataset, I have leveraged its learning here. I have modified the last fully connected layer, it takes in the original number of inputs, however, the number of outputs =133, which is the number of classes.

(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)](http://pytorch.org/docs/master/nn.html#loss-functions) and [optimizer (http://pytorch.org/docs/master/optim.html)](http://pytorch.org/docs/master/optim.html). Save the chosen loss function as `criterion_transfer` , and the optimizer as `optimizer_transfer` below.

In [24]:
```
import torch.optim as optim
criterion_transfer = nn.CrossEntropyLoss()

optimizer_transfer = optim.SGD(model_transfer.classifier.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)](http://pytorch.org/docs/master/notes/serialization.html) at filepath `'model_transfer.pt'` .

In [25]:
```
# train the model
n_epochs = 10
model_transfer = train(n_epochs, loaders_transfer, model_transfer, opt
imizer_transfer, criterion_transfer, use_cuda, 'model_transfer.pt')
```

```
Epoch 1, Batch 1 loss: 5.069785
Epoch 1, Batch 101 loss: 4.834227
Epoch 1, Batch 201 loss: 4.665232
Epoch 1, Batch 301 loss: 4.508173
Epoch: 1        Training Loss: 4.453574         Validation Loss: 3.6
78838
Validation loss decreased (inf --> 3.678838).  Saving model ...
Epoch 2, Batch 1 loss: 3.952502
Epoch 2, Batch 101 loss: 3.698307
Epoch 2, Batch 201 loss: 3.501250
Epoch 2, Batch 301 loss: 3.308172
```

```
Epoch: 2        Training Loss: 3.248672         Validation Loss: 2.3
53349
Validation loss decreased (3.678838 --> 2.353349).  Saving model ...
Epoch 3, Batch 1 loss: 2.569170
Epoch 3, Batch 101 loss: 2.589341
Epoch 3, Batch 201 loss: 2.494047
Epoch 3, Batch 301 loss: 2.411776
Epoch: 3        Training Loss: 2.387765         Validation Loss: 1.7
77198
Validation loss decreased (2.353349 --> 1.777198).  Saving model ...
Epoch 4, Batch 1 loss: 2.177922
Epoch 4, Batch 101 loss: 2.050360
Epoch 4, Batch 201 loss: 2.004336
Epoch 4, Batch 301 loss: 1.979956
Epoch: 4        Training Loss: 1.968962         Validation Loss: 1.4
46091
Validation loss decreased (1.777198 --> 1.446091).  Saving model ...
Epoch 5, Batch 1 loss: 1.371338
Epoch 5, Batch 101 loss: 1.801560
Epoch 5, Batch 201 loss: 1.782211
Epoch 5, Batch 301 loss: 1.778577
Epoch: 5        Training Loss: 1.768278         Validation Loss: 1.3
85418
Validation loss decreased (1.446091 --> 1.385418).  Saving model ...
Epoch 6, Batch 1 loss: 1.934087
Epoch 6, Batch 101 loss: 1.713235
Epoch 6, Batch 201 loss: 1.700749
Epoch 6, Batch 301 loss: 1.658874
Epoch: 6        Training Loss: 1.651584         Validation Loss: 1.2
17292
Validation loss decreased (1.385418 --> 1.217292).  Saving model ...
Epoch 7, Batch 1 loss: 1.409009
Epoch 7, Batch 101 loss: 1.600665
Epoch 7, Batch 201 loss: 1.574593
Epoch 7, Batch 301 loss: 1.556478
Epoch: 7        Training Loss: 1.548958         Validation Loss: 1.1
70674
Validation loss decreased (1.217292 --> 1.170674).  Saving model ...
Epoch 8, Batch 1 loss: 1.954287
Epoch 8, Batch 101 loss: 1.469319
Epoch 8, Batch 201 loss: 1.518271
Epoch 8, Batch 301 loss: 1.495836
Epoch: 8        Training Loss: 1.493075         Validation Loss: 1.2
34170
Epoch 9, Batch 1 loss: 1.086879
Epoch 9, Batch 101 loss: 1.501101
Epoch 9, Batch 201 loss: 1.468078
Epoch 9, Batch 301 loss: 1.461478
Epoch: 9        Training Loss: 1.452642         Validation Loss: 1.1
24516
```

```
Validation loss decreased (1.170674 --> 1.124516).  Saving model ...
Epoch 10, Batch 1 loss: 0.645355
Epoch 10, Batch 101 loss: 1.393519
Epoch 10, Batch 201 loss: 1.407277
Epoch 10, Batch 301 loss: 1.395655
Epoch: 10 	Training Loss: 1.389330 	Validation Loss: 1.1
18025
Validation loss decreased (1.124516 --> 1.118025).  Saving model ...
```

In [26]:

```
# load the model that got the best validation accuracy (uncomment the
line below)
model_transfer.load_state_dict(torch.load('model_transfer.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 60%.

In [27]:

```
#test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 1.144976


Test Accuracy: 69% (577/836)
```

(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 [28]:
```
### TODO: Write a function that takes a path to an image as input
### and returns the dog breed that is predicted by the model.
from PIL import Image
import torchvision.transforms as transforms
# 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]

def predict_breed_transfer(img_path):
    # load the image and return the predicted breed
    # Open image
    dog_img = Image.open(img_path)
    # Tranforms
    transform = transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(

            mean=[0.485, 0.456, 0.406],
            std=[0.229, 0.224, 0.225]

    )])

    img_transformed = transform(dog_img)

    #Define a batch to be passed through the network
    batch_t = torch.unsqueeze(img_transformed, 0)
    if torch.cuda.is_available():
        batch_t = batch_t.cuda()
    model_transfer.eval()
    output = model_transfer(batch_t)
    #print (output.shape)
    #max_val, index = torch.max(output,1)
    idx = torch.argmax(output)
    return class_names[idx]
```

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 `human_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!

(IMPLEMENTATION) Write your Algorithm

In [29]:
```
### 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
    ## 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(img_path)
        print("Dogs Detected!\nIt appears to be a {0}".format(predicti
on))
    elif face_detector(img_path) > 0:
        prediction = predict_breed_transfer(img_path)
        print("Hello, there buddy!\n If you were a dog..You may resemb
le a {0}".format(prediction))
    else:
        print("Error! .. It appears that the dog and the hooman have g
one for a walk..nothing detected here")
```

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?

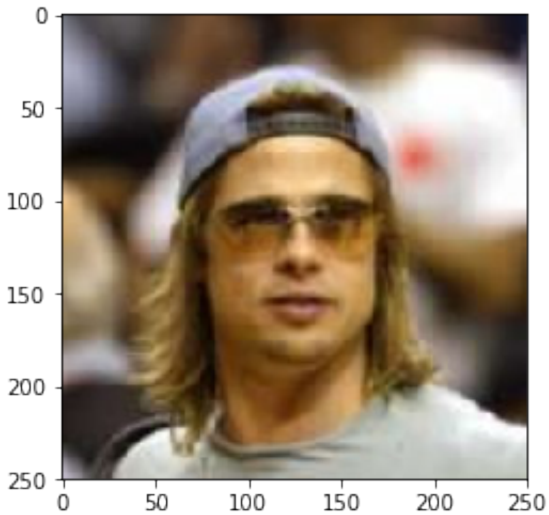
(IMPLEMENTATION) Test Your Algorithm on Sample Images!

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.

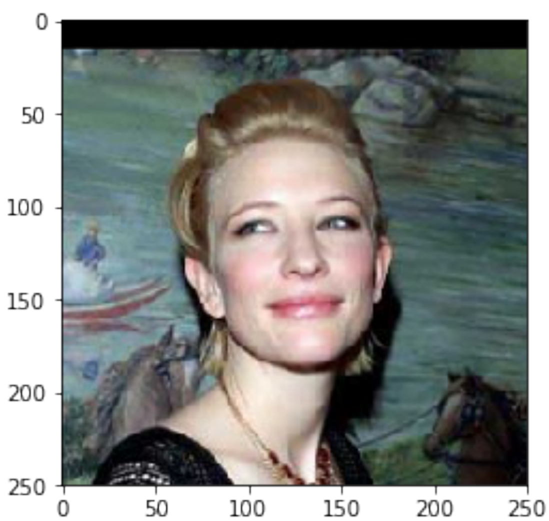
Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer: (Three possible points for improvement) The output is similar to what I had expected. The accuracy could have been improved by: 1) Using transfer learning with a model which is better suited for detecting animal faces rather than VGG16 which detects a lot of other classes as well 2) Use of more images in the dataset or better augmentation of data set 3) Use of a better learnng rate in the algorithm

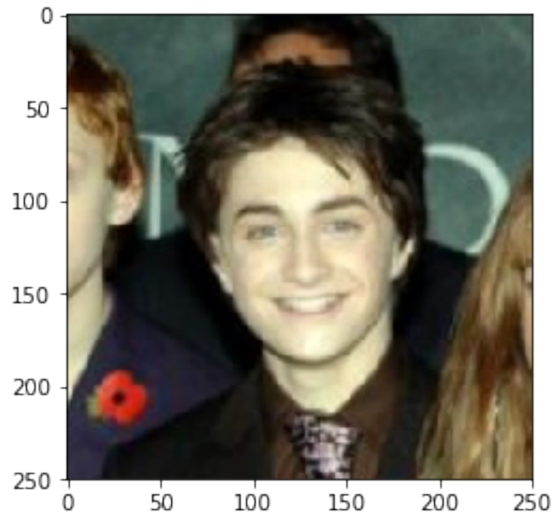
In [30]:
```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.
human_files = ['/data/lfw/Brad_Pitt/Brad_Pitt_0001.jpg', '/data/lfw/Cate_Blanchett/Cate_Blanchett_0001.jpg', '/data/lfw/Daniel_Radcliffe/Daniel_Radcliffe_0003.jpg' ]
dog_files = ['/data/dog_images/test/129.Tibetan_mastiff/Tibetan_mastiff_08138.jpg', '/data/dog_images/test/124.Poodle/Poodle_07910.jpg', '/data/dog_images/test/005.Alaskan_malamute/Alaskan_malamute_00346.jpg']


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




```
Hello, there buddy!
 If you were a dog..You may resemble a Entlebucher mountain dog
```

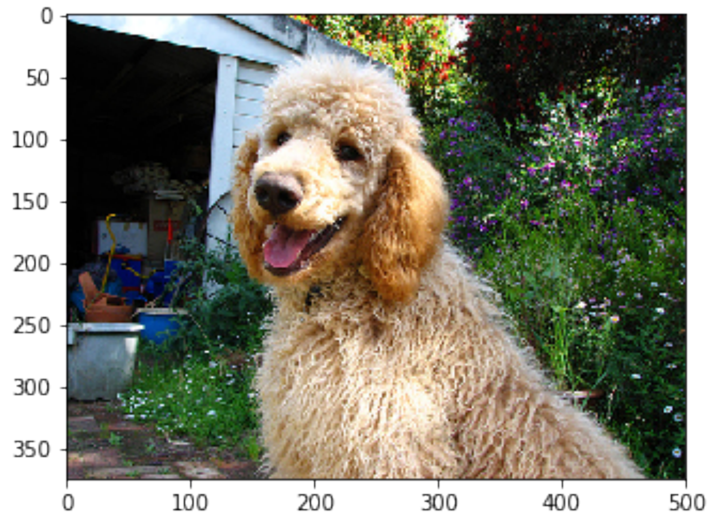



```
Hello, there buddy!
 If you were a dog..You may resemble a Afghan hound
```

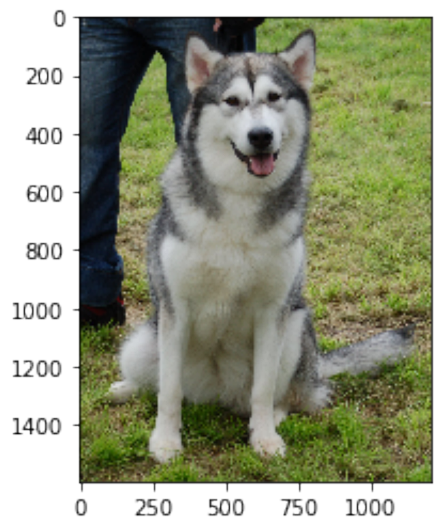



```
Hello, there buddy!
 If you were a dog..You may resemble a Welsh springer spaniel
```


```
Dogs Detected!
It appears to be a Tibetan mastiff
```




```
Dogs Detected!
It appears to be a English cocker spaniel
```




```
Dogs Detected!
It appears to be a Alaskan malamute
```

In [ ]:

In [ ]:

In [ ]:

In [ ]: