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.

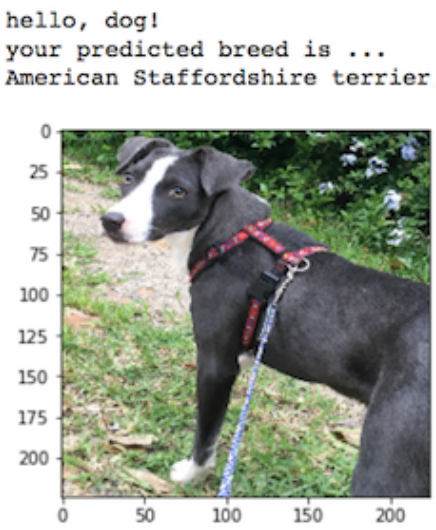
In addition to implementing code, there will be questions that you must answer which relate to the project and your implementation. Each section where you will answer a question is preceded by a **'Question X'** header. Carefully read each question and provide thorough answers in the following text boxes 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 [4]:
```
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.
```

In [22]:
```
!ls /data/dog_images/
```

```
test  train  valid
```

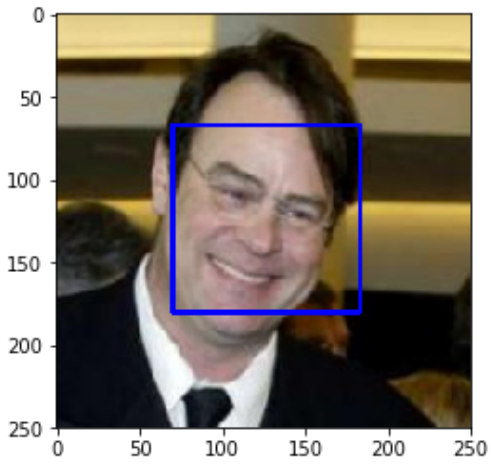
Step 1: Detect Humans

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 [10]:
```
# 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: Human Face: 98.0% Dog Faces: 17.0%

In [11]:
```
from tqdm import tqdm

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

#-#-# Do NOT modify the code above this line. #-#-#
count_humans = 0
count_dogs   = 0

for i in human_files_short:
    if face_detector(i):
        count_humans += 1

for i in dog_files_short:
    if face_detector(i):
        count_dogs += 1


print("Total percentage humans detected: {}%".format(count_humans))
print("Total percentage dogs detected: {}%".format(count_dogs))
```

```
Total percentage humans detected: 98%
Total percentage dogs detected: 17%
```

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 [12]:
```
### (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) 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 [13]:
```
import torch
import torchvision.models as models

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

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

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

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg16-397923af.pth
100%|██████████| 553433881/553433881 [00:05<00:00, 95749415.87it/s]
```

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 [14]:
```
from PIL import Image
import torchvision.transforms as transforms
from torch import autograd

from PIL import ImageFile

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
    '''
    transform = transforms.Compose([transforms.Resize((224, 224)), transforms.ToTensor(), transforms.Normalize(mean = [0.485, 0.4
    img = Image.open(img_path)
    img = transform(img)
    img = img.unsqueeze(0)
    img = autograd.Variable(img)
    if use_cuda:
        img = img.cuda()

    prediction = VGG16(img)
    _, index = torch.max(prediction, 1)

    return index.item()
```

(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 [15]:
```
### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    ## TODO: Complete the function.
    pred = (VGG16_predict(img_path) >= 151) and (VGG16_predict(img_path) <= 268)
    return pred
```

(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: Total percentage dogs detected: 100%

Total percentage humans detected: 0%

In [16]:
```
### Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
from tqdm import tqdm
detect_dogs = 0
detect_humans = 0

for i in tqdm(range(len(human_files_short))):
    detect_humans += int(dog_detector(human_files_short[i]))


for i in tqdm(range(len(dog_files_short))):
    detect_dogs += int(dog_detector(dog_files_short[i]))

print("Total percentage dogs detected: {}%".format(detect_dogs))
print("Total percentage humans detected: {}%".format(detect_humans))
```

```
100%|██████████| 100/100 [00:06<00:00, 15.08it/s]
100%|██████████| 100/100 [00:08<00:00, 11.30it/s]

Total percentage dogs detected: 100%
Total percentage humans detected: 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 [ ]:
```
### (Optional)
### TODO: Report the performance of another pre-trained network.
### Feel free to use as many code cells as needed.
```

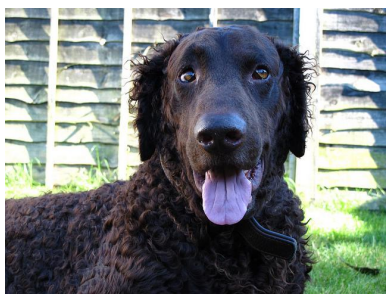
Step 3: Create a CNN to Classify Dog Breeds (from Scratch)

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
	

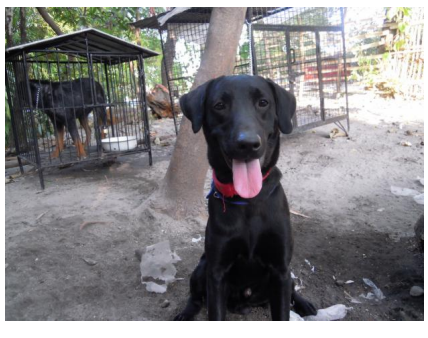
Likewise, recall that labradors come in yellow, chocolate, and black. Your vision-based algorithm will have to conquer this high intra-class variation to determine how to classify all of these different shades as the same breed.

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 [17]:
```
import os
from torchvision import datasets

### Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes
transformed_train = transforms.Compose([transforms.Resize(256), transforms.RandomResizedCrop(224), transforms.RandomHorizontalFli
transformed_test = transforms.Compose([transforms.Resize(256), transforms.CenterCrop(224), transforms.ToTensor(), transforms.Norm
transformed_valid = transforms.Compose([transforms.Resize(256), transforms.CenterCrop(224), transforms.ToTensor(), transforms.Nor
```

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 the image into 256px and crops the image to 224 * 224. Then, I horizontal flip the images for augmentation.

(IMPLEMENTATION) Model Architecture

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

In [23]:
```
import torch.nn as nn
import torch.nn.functional as F

# define the CNN architecture
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.layer1 = nn.Conv2d(3, 16, 3, padding = 1)
        self.layer2 = nn.Conv2d(16, 32, 3, padding = 1)
        self.layer3 = nn.Conv2d(32, 64, 3, padding = 1)
        self.maxpool = nn.MaxPool2d(2, 2)
        self.linear1 = nn.Linear(64 * 28 * 28, 500)
        self.linear2 = nn.Linear(500, 133)
        self.dropout = nn.Dropout(0.2)
        self.normalized_batches = nn.BatchNorm1d(num_features = 500)


    def forward(self, x):
        x = self.maxpool(F.relu(self.layer1(x)))
        x = self.dropout(x)
        x = self.maxpool(F.relu(self.layer2(x)))
        x = self.dropout(x)
        x = self.maxpool(F.relu(self.layer3(x)))
        x = self.dropout(x)
        x = x.view(x.size(0), -1)
        x = F.relu(self.normalized_batches(self.linear1(x)))
        x = self.dropout(x)
        x = self.linear2(x)

        return x

#-#-# You so 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: The input image is 2 channel 224x224 size image. My model is consist of few convulation block followd by maxpool, finally I added Liner layer. I also add some dropout to reduce overfitting.

(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 [24]:
```
import torch.optim as optim

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

### TODO: select optimizer
learning_rate = 0.001
optimizer_scratch = torch.optim.Adam(model_scratch.parameters(), lr=learning_rate)
```

(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 [25]:
```
def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
    validation_loss = np.Inf
    total_epochs = n_epochs + 1

    for epoch in range(1, total_epochs):
        train_loss = 0.0
        valid_loss = 0.0

        model.train()

        for data, target in loaders["train"]:
            if use_cuda:
                data, target = data.cuda(), target.cuda()

            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()
            train_loss += loss.item() * data.size(0)

        model.eval()

        for data, target in loaders["valid"]:
            if use_cuda:
                data, target = data.cuda(), target.cuda()

            output = model(data)
            loss = criterion(output, target)
            valid_loss += loss.item() * data.size(0)

        train_loss = train_loss / len(loaders["train"].dataset)
        valid_loss = valid_loss / len(loaders["valid"].dataset)

        print("Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}!".format(epoch, train_loss, valid_loss))

        if valid_loss <= validation_loss:
            print("Validation loss decreased ({:.6f} --> {:.6f}).  Saving model!".format(validation_loss, valid_loss))
            torch.save(model.state_dict(), save_path)
            validation_loss = valid_loss
        else:
            print("Validation loss increased!")

    return model
```

In [26]:
```
!ls /data/dog_images/
```

```
test  train  valid
```

In [27]:
```
train_data = datasets.ImageFolder(os.path.join('/data/dog_images/', 'train/'), transform = transformed_train)
valid_data = datasets.ImageFolder(os.path.join('/data/dog_images/', 'valid/'), transform = transformed_valid)
test_data = datasets.ImageFolder(os.path.join('/data/dog_images/', 'test/'), transform = transformed_test)
train_loader = torch.utils.data.DataLoader(train_data, batch_size = 20, shuffle = True, num_workers = 0)
valid_loader = torch.utils.data.DataLoader(train_data, batch_size = 20, shuffle = True, num_workers = 0)
test_loader = torch.utils.data.DataLoader(train_data, batch_size = 20, shuffle = True, num_workers = 0)
loaders_scratch = {"train": train_loader, "valid": valid_loader, "test": test_loader}
# Workaround to OSError:
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True
```

In [31]:
```
%%time
model_scratch = train(10, loaders_scratch, model_scratch, optimizer_scratch, criterion_scratch, use_cuda, "model_scratch.pt")
model_scratch.load_state_dict(torch.load("model_scratch.pt"))
```

```
Epoch: 1        Training Loss: 4.546867         Validation Loss: 4.426538!
Validation loss decreased (inf --> 4.426538).  Saving model!
Epoch: 2        Training Loss: 4.449915         Validation Loss: 4.356498!
Validation loss decreased (4.426538 --> 4.356498).  Saving model!
Epoch: 3        Training Loss: 4.387846         Validation Loss: 4.279342!
Validation loss decreased (4.356498 --> 4.279342).  Saving model!
Epoch: 4        Training Loss: 4.317317         Validation Loss: 4.255357!
Validation loss decreased (4.279342 --> 4.255357).  Saving model!
Epoch: 5        Training Loss: 4.258778         Validation Loss: 4.132090!
Validation loss decreased (4.255357 --> 4.132090).  Saving model!
Epoch: 6        Training Loss: 4.186869         Validation Loss: 4.141408!
Validation loss increased!
Epoch: 7        Training Loss: 4.122899         Validation Loss: 4.054836!
Validation loss decreased (4.132090 --> 4.054836).  Saving model!
Epoch: 8        Training Loss: 4.082578         Validation Loss: 3.961900!
Validation loss decreased (4.054836 --> 3.961900).  Saving model!
Epoch: 9        Training Loss: 4.032396         Validation Loss: 3.992254!
Validation loss increased!
Epoch: 10       Training Loss: 3.971280         Validation Loss: 3.850921!
```

In [32]:
```
256 * 6 * 6
64 * 28 * 28
```

Out[32]:
```
50176
```

(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 [33]:
```
def test(loaders, model, criterion, use_cuda):
    test_loss = 0.
    correct = 0.
    total = 0.

    model.eval()
    for batch_idx, (data, target) in enumerate(loaders['test']):
        if use_cuda:
            data, target = data.cuda(), target.cuda()
        output = model(data)
        loss = criterion(output, target)
        test_loss = test_loss + ((1 / (batch_idx + 1)) * (loss.data - test_loss))
        pred = output.data.max(1, keepdim=True)[1]
        correct += np.sum(np.squeeze(pred.eq(target.data.view_as(pred))).cpu().numpy())
        total += data.size(0)

    test_loss = test_loss/len(loaders['test'].dataset)

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

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

test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 0.000576


Test Accuracy: 12.020958% (803/6680)
```

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 [34]:
```
## TODO: Specify data loaders
print(loaders_scratch)
```

```
{'train': <torch.utils.data.dataloader.DataLoader object at 0x7f017086bb00>, 'valid': <torch.utils.data.dataloader.DataLoader ob
ject at 0x7f0171d10d68>, 'test': <torch.utils.data.dataloader.DataLoader object at 0x7f017086ba20>}
```

(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 [35]:
```
import torchvision.models as models
import torch.nn as nn

## TODO: Specify model architecture
model_transfer = models.resnet18(pretrained = True)

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

```
Downloading: "https://download.pytorch.org/models/resnet18-5c106cde.pth" to /root/.torch/models/resnet18-5c106cde.pth
100%|██████████| 46827520/46827520 [00:00<00:00, 88815155.44it/s]
```

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 used pretrain resnet for this traing. The pretrained models contains trained weights for the network. Hence if a network pretrained for some classification task is used, the number of steps for the output to converge reduces. It is because generally for the classification task, the features extracted will be similar. Instead of initializing the model with random weights, initializing it with the pretrained weights reduces the training time and hence is more efficient.

(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 [36]:
```
criterion_transfer = nn.CrossEntropyLoss()
optimizer_transfer = optim.Adam(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)](http://pytorch.org/docs/master/notes/serialization.html) at filepath `'model_transfer.pt'`.

In [ ]:
```
%%time
# train the model
model_transfer = train(10, loaders_scratch, model_transfer, optimizer_transfer, criterion_transfer, use_cuda, 'model_transfer.pt'

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

```
Epoch: 1        Training Loss: 1.801166         Validation Loss: 1.489327!
Validation loss decreased (inf --> 1.489327).  Saving model!
Epoch: 2        Training Loss: 1.566370         Validation Loss: 1.353572!
Validation loss decreased (1.489327 --> 1.353572).  Saving model!
Epoch: 3        Training Loss: 1.510503         Validation Loss: 1.222027!
Validation loss decreased (1.353572 --> 1.222027).  Saving model!
Epoch: 4        Training Loss: 1.417090         Validation Loss: 1.115728!
Validation loss decreased (1.222027 --> 1.115728).  Saving model!
Epoch: 5        Training Loss: 1.316892         Validation Loss: 1.092205!
Validation loss decreased (1.115728 --> 1.092205).  Saving model!
Epoch: 6        Training Loss: 1.294172         Validation Loss: 1.097452!
Validation loss increased!
Epoch: 7        Training Loss: 1.267938         Validation Loss: 1.018915!
Validation loss decreased (1.092205 --> 1.018915).  Saving model!
Epoch: 8        Training Loss: 1.222505         Validation Loss: 1.027391!
Validation loss increased!
Epoch: 9        Training Loss: 1.212114         Validation Loss: 0.962779!
Validation loss decreased (1.018915 --> 0.962779).  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 60%.

In [43]:
```
%%time
test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.000148


Test Accuracy: 73.128743% (4885/6680)
CPU times: user 1min 59s, sys: 9.63 s, total: 2min 8s
Wall time: 1min 54s
```

(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 [45]:
```
### 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 train_data.classes]

def predict_breed_transfer(img_path):
    image = Image.open(img_path)
    image = image.convert('RGB')
    transform_image = transforms.Compose([transforms.Resize(256), transforms.CenterCrop(224), transforms.ToTensor(), transforms.N
    image = transform_image(image)[:3,:,:].unsqueeze(0)

    if use_cuda:
        pred_transfer = model_transfer(image.cuda())

    else:
        pred_transfer = model_transfer(image)

    _, prediction = torch.max(pred_transfer, 1)

    return class_names[np.squeeze(prediction.cpu().numpy())]
```

Step 5: Write your Algorithm

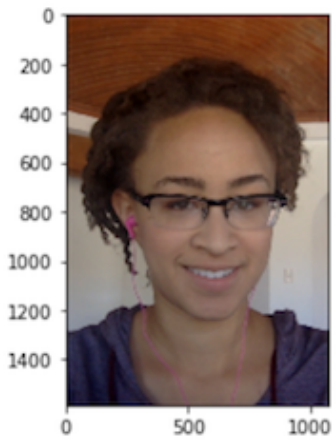
Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then,

- 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!

```
hello, human!
```




```
You look like a ...
Chinese_shar-pei
```

(IMPLEMENTATION) Write your Algorithm

In [46]:
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.

def run_app(img_path):
    opened_image = Image.open(img_path)
    if face_detector(img_path):
        print("Hey there, human! Please note that is designed for doggies! :-)")
        plt.title("You're a {}!".format(predict_breed_transfer(img_path)))
        plt.imshow(opened_image)
        plt.show()
        print("Hey there, {}".format(predict_breed_transfer(img_path)))

    elif dog_detector((img_path)):
        print("Aww... such a cute doggie! :D")
        plt.title("You're a {}!".format(predict_breed_transfer(img_path)))
        plt.imshow(opened_image)
        plt.show()

        print("Hey there, {}!".format(predict_breed_transfer(img_path)))

    else:
        print("Ahhh! Your image shown below is NOT a dog or a human!")
        plt.imshow(opened_image)
        plt.show()
```

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.

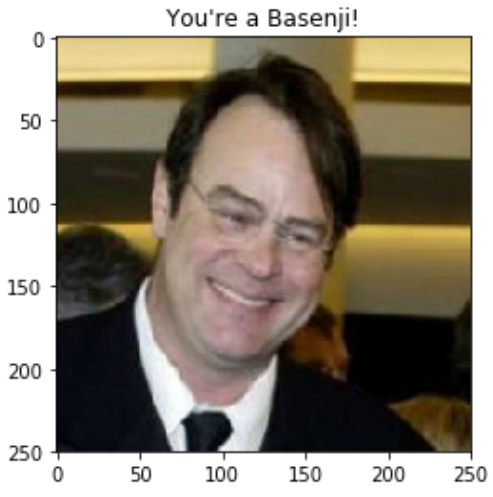
Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer: The output seems ok. 1 error at human.

1. Increse the amount of data
2. Use high droupout to reduce overfitting.
3. use data augmentation.
4. Add more epochs
5. add adoptive learning rate and batch normalization

In [47]:
```
## 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 file in np.hstack((human_files[:3], dog_files[:3])):
    run_app(file)
```

```
Hey there, human! Please note that is designed for doggies! :-)
```




```
Hey there, Basenji
```

In [ ]:
```
1
```

In [ ]:
```

```