

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.

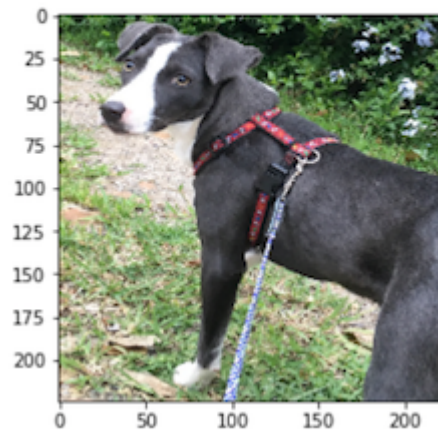
Note: Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. Markdown cells can be edited by double-clicking the cell to enter edit mode.

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!).

```
hello, dog!  
your predicted breed is ...  
American Staffordshire terrier
```



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) (<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) (<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/) (<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("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) (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_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 [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)

- 100% of human images with a detected face
- 14.0% of dog images with a detected face

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. ###

## Test the performance of the face_detector algorithm
## on the images in human_files_short and dog_files_short.
true_positive = 0.
for human_img_path in tqdm(human_files_short):
    if face_detector(human_img_path):
        true_positive += 1.
true_positive /= 100.

false_negative = 0.
for dog_img_path in tqdm(dog_files_short):
    if face_detector(dog_img_path):
        false_negative += 1.
false_negative /= 100.

print("%f of human images with a detected face"%true_positive)
print("%f of dog images with a detected face"%false_negative)

```

```

100%|██████████| 100/100 [00:02<00:00, 33.15it/s]
100%|██████████| 100/100 [00:09<00:00, 10.43it/s]

```

```

1.000000 of human images with a detected face
0.140000 of dog images with a detected face

```

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 []:

```

### (Optional)
### TODO: Test performance of another face 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 [5]:

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

In [6]:

```

from PIL import Image
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True
import torchvision.transforms as transforms
from torch.autograd import Variable

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
    """
    VGG16.eval()

    img_to_tensor = transforms.ToTensor()
    img = Image.open(img_path)
    img = img.resize((224, 224))
    tensor = img_to_tensor(img)
    tensor = tensor.resize_(1, 3, 224, 224).cuda()

    prob = VGG16(Variable(tensor))
    prob_npy = prob.data.cpu().numpy()
    index = np.argmax(prob_npy[0])
    return index

```

In [7]:

```

index = VGG16_predict('dogImages/train/001.Affenpinscher/Affenpinscher_00001.jpg')
print(index)

```

252

(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 [8]:

```

### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    index = VGG16_predict(img_path)
    return index >= 151 and index <= 268

```


(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:

- 0% of the images in `human_files_short` have a detected dog?
- 86% of the images in `dog_files_short` have a detected dog?

In [9]:

```
from tqdm import tqdm

## Test the performance of the face_detector algorithm
## on the images in human_files_short and dog_files_short.
false_negative = 0.
for human_img_path in tqdm(human_files_short):
    if dog_detector(human_img_path):
        false_negative += 1.
false_negative /= 100.

true_positive = 0.
for dog_img_path in tqdm(dog_files_short):
    if dog_detector(dog_img_path):
        true_positive += 1.
true_positive /= 100.

print("%f of human images with a detected dog"%false_negative)
print("%f of dog images with a detected dog"%true_positive)
```

```
100%|████████████████████| 100/100 [00:01<00:00, 95.21it/s]
100%|████████████████████| 100/100 [00:02<00:00, 39.52it/s]
```

```
0.000000 of human images with a detected dog
0.860000 of dog images with a detected dog
```

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#resnet-50) (<http://pytorch.org/docs/master/torchvision/models.html#resnet-50>), 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`.

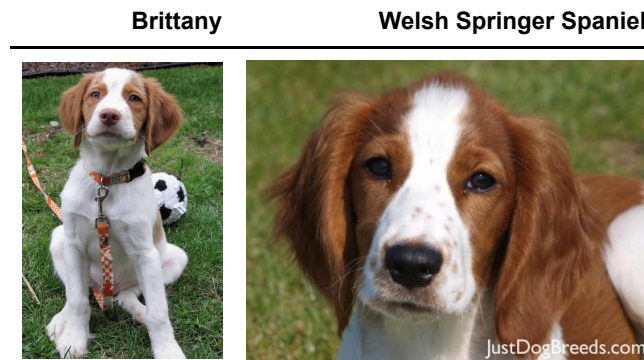
In []:

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

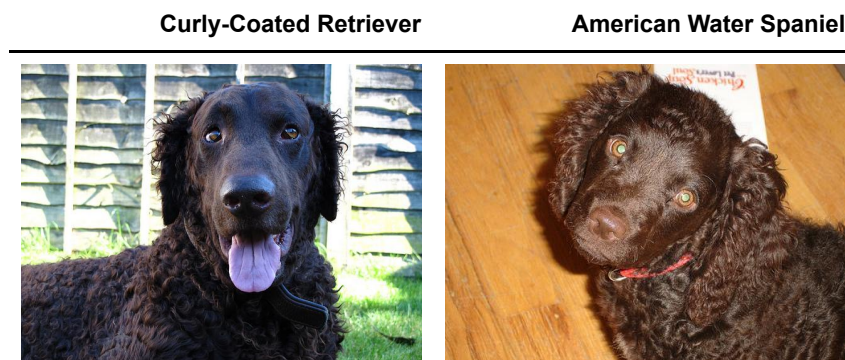
Step 3: Create a CNN to Classify Dog Breeds (from Scratch)

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.



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



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 imbalanced, 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 [10]:

```

import os
from torchvision import datasets

## Data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes
data_transforms = {
    'train': transforms.Compose([
        transforms.Resize(256),
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'valid': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'test': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
}

image_datasets = {x: datasets.ImageFolder(os.path.join('dogImages', x),
                                           data_transforms[x])
                  for x in ['train', 'valid', 'test']}
loaders_scratch = {x: torch.utils.data.DataLoader(image_datasets[x], batch_size=128,
                                                    shuffle=True, num_workers=4)
                  for x in ['train', 'valid', 'test']}

# number of dog breed classes
classes = os.listdir('./dogImages/train/')
nClasses = len(classes)
print('There are %d of dog breed classes'%nClasses)

```

There are 133 of dog breed classes

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: Based on the usual parameters,

- I resize the images into 256x256 (same as imagenet);
- I use a random cropping at training, and a cropping around the center at testing and validation. The cropping size is 224x224, which is popular in some pre-trained models (AlexNet, VGG16, GoogleNet, etc.);
- I normalize the data into zero-mean unit-variance Gaussian distribution, with the mean and variance calculated from imagenet (please refer to <https://github.com/jacobgil/pytorch-grad-cam/issues/6>)

(<https://github.com/jacobgil/pytorch-grad-cam/issues/6>) for detail);

- To augment the dataset, at training phrse, I flip the images horizontally.

remark: number of classed should be used below

(IMPLEMENTATION) Model Architecture

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

In [11]:

```
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.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 32, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=3),

            nn.Conv2d(32, 16, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=3),

            nn.Conv2d(16, 16, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=3),
        )
        self.classifier = nn.Sequential(
            nn.Dropout(),
            nn.Linear(16 * 8 * 8, 133),
        )

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), 16 * 8 * 8)
        x = self.classifier(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: I design the CNN from bottom to up.

1. The first conv-layer should capture some detailed features, so I design the kernel size to be 3x3, with a number of 64.
2. The second and third conv-layers are design to allow a more complex transform of low-level features. After that, I use max-pooling in order to capture the most important features.
3. After that, conv-ReLU-pooling is repeated twice, in order to capture high-level features with lower dimensions.
4. The prediction layer is a single linear layer, I think that's enough, because I have finished most non-linear transformation before.

Some more details:

- activation function: ReLU() is the best choice, because its good property for calculation of gradients;
- max-pooling: it can capture the most important features and reduce the features' dimension;
- dropout: it helps the model to be robust by randomly blinding a part of features when training.

(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 [12]:

```
import torch.optim as optim

criterion_scratch = nn.CrossEntropyLoss()
optimizer_scratch = optim.Adam(model_scratch.parameters())
```

(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 [13]:

```

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.
        valid_loss = 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
            out = model.forward(data)
            loss = criterion(out, target)
            pred = torch.max(out, 1)[1]
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            train_loss += ((1 / (batch_idx + 1)) * (loss.data - 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
            out = model.forward(data)
            loss = criterion(out, target)
            pred = torch.max(out, 1)[1]

            valid_loss += ((1 / (batch_idx + 1)) * (loss.data - valid_loss))

        # print training/validation statistics
        print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}\t'.format(
            epoch,
            train_loss,
            valid_loss))

        # save the model if validation loss has decreased
        if valid_loss < valid_loss_min:
            valid_loss_min = valid_loss
            torch.save(model.state_dict(), save_path)

    # return trained model
    return model

```



```
# train the model
model_scratch = train(100, loaders_scratch, model_scratch, optimizer_scratch,
                      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'))

# empty some cache in GPU to ensure the running of following blocks
torch.cuda.empty_cache()
```

Epoch: 1	Training Loss: 4.867664	Validation Loss: 4.783989
Epoch: 2	Training Loss: 4.731144	Validation Loss: 4.562083
Epoch: 3	Training Loss: 4.572004	Validation Loss: 4.462406
Epoch: 4	Training Loss: 4.498731	Validation Loss: 4.405581
Epoch: 5	Training Loss: 4.449533	Validation Loss: 4.342847
Epoch: 6	Training Loss: 4.421070	Validation Loss: 4.278636
Epoch: 7	Training Loss: 4.380388	Validation Loss: 4.237258
Epoch: 8	Training Loss: 4.377641	Validation Loss: 4.210796
Epoch: 9	Training Loss: 4.344580	Validation Loss: 4.193370
Epoch: 10	Training Loss: 4.317041	Validation Loss: 4.162899
Epoch: 11	Training Loss: 4.285978	Validation Loss: 4.164668
Epoch: 12	Training Loss: 4.285047	Validation Loss: 4.129717
Epoch: 13	Training Loss: 4.234483	Validation Loss: 4.151237
Epoch: 14	Training Loss: 4.232903	Validation Loss: 4.096299
Epoch: 15	Training Loss: 4.206538	Validation Loss: 4.014736
Epoch: 16	Training Loss: 4.175619	Validation Loss: 4.014867
Epoch: 17	Training Loss: 4.144131	Validation Loss: 3.982692
Epoch: 18	Training Loss: 4.115994	Validation Loss: 3.968592
Epoch: 19	Training Loss: 4.107320	Validation Loss: 3.998900
Epoch: 20	Training Loss: 4.093560	Validation Loss: 3.952085
Epoch: 21	Training Loss: 4.053680	Validation Loss: 3.907688
Epoch: 22	Training Loss: 4.013766	Validation Loss: 3.836669
Epoch: 23	Training Loss: 3.990167	Validation Loss: 3.860245
Epoch: 24	Training Loss: 4.030689	Validation Loss: 3.891383
Epoch: 25	Training Loss: 3.974239	Validation Loss: 3.860436
Epoch: 26	Training Loss: 3.940065	Validation Loss: 3.816918
Epoch: 27	Training Loss: 3.925757	Validation Loss: 3.802772
Epoch: 28	Training Loss: 3.879298	Validation Loss: 3.758149
Epoch: 29	Training Loss: 3.880880	Validation Loss: 3.700731
Epoch: 30	Training Loss: 3.850991	Validation Loss: 3.647301
Epoch: 31	Training Loss: 3.868132	Validation Loss: 3.638294
Epoch: 32	Training Loss: 3.808233	Validation Loss: 3.724174
Epoch: 33	Training Loss: 3.816420	Validation Loss: 3.670613
Epoch: 34	Training Loss: 3.811236	Validation Loss: 3.594916
Epoch: 35	Training Loss: 3.788311	Validation Loss: 3.636401
Epoch: 36	Training Loss: 3.752562	Validation Loss: 3.702321
Epoch: 37	Training Loss: 3.749101	Validation Loss: 3.547547
Epoch: 38	Training Loss: 3.745150	Validation Loss: 3.575325
Epoch: 39	Training Loss: 3.710807	Validation Loss: 3.512432
Epoch: 40	Training Loss: 3.726886	Validation Loss: 3.529741
Epoch: 41	Training Loss: 3.714965	Validation Loss: 3.555908
Epoch: 42	Training Loss: 3.656342	Validation Loss: 3.549533
Epoch: 43	Training Loss: 3.665465	Validation Loss: 3.476783
Epoch: 44	Training Loss: 3.662195	Validation Loss: 3.480884
Epoch: 45	Training Loss: 3.669784	Validation Loss: 3.515505
Epoch: 46	Training Loss: 3.654050	Validation Loss: 3.461539
Epoch: 47	Training Loss: 3.635451	Validation Loss: 3.491172
Epoch: 48	Training Loss: 3.627530	Validation Loss: 3.459276
Epoch: 49	Training Loss: 3.612589	Validation Loss: 3.438677

Epoch: 50	Training Loss: 3.606941	Validation Loss: 3.479696
Epoch: 51	Training Loss: 3.618033	Validation Loss: 3.435613
Epoch: 52	Training Loss: 3.578087	Validation Loss: 3.386400
Epoch: 53	Training Loss: 3.580255	Validation Loss: 3.471660
Epoch: 54	Training Loss: 3.590029	Validation Loss: 3.357655
Epoch: 55	Training Loss: 3.578808	Validation Loss: 3.336715
Epoch: 56	Training Loss: 3.566099	Validation Loss: 3.319447
Epoch: 57	Training Loss: 3.536601	Validation Loss: 3.411227
Epoch: 58	Training Loss: 3.531066	Validation Loss: 3.295799
Epoch: 59	Training Loss: 3.513825	Validation Loss: 3.412293
Epoch: 60	Training Loss: 3.521830	Validation Loss: 3.358713
Epoch: 61	Training Loss: 3.553102	Validation Loss: 3.305954
Epoch: 62	Training Loss: 3.502467	Validation Loss: 3.344453
Epoch: 63	Training Loss: 3.504530	Validation Loss: 3.320382
Epoch: 64	Training Loss: 3.471836	Validation Loss: 3.304412
Epoch: 65	Training Loss: 3.487591	Validation Loss: 3.374372
Epoch: 66	Training Loss: 3.485028	Validation Loss: 3.282086
Epoch: 67	Training Loss: 3.478248	Validation Loss: 3.268273
Epoch: 68	Training Loss: 3.453925	Validation Loss: 3.275235
Epoch: 69	Training Loss: 3.464975	Validation Loss: 3.370662
Epoch: 70	Training Loss: 3.462184	Validation Loss: 3.341583
Epoch: 71	Training Loss: 3.434187	Validation Loss: 3.223505
Epoch: 72	Training Loss: 3.474640	Validation Loss: 3.374682
Epoch: 73	Training Loss: 3.425884	Validation Loss: 3.233712
Epoch: 74	Training Loss: 3.404267	Validation Loss: 3.189191
Epoch: 75	Training Loss: 3.411180	Validation Loss: 3.234677
Epoch: 76	Training Loss: 3.386272	Validation Loss: 3.228169
Epoch: 77	Training Loss: 3.432693	Validation Loss: 3.254307
Epoch: 78	Training Loss: 3.389027	Validation Loss: 3.195496
Epoch: 79	Training Loss: 3.377232	Validation Loss: 3.230615
Epoch: 80	Training Loss: 3.400803	Validation Loss: 3.213140
Epoch: 81	Training Loss: 3.430191	Validation Loss: 3.189485
Epoch: 82	Training Loss: 3.340624	Validation Loss: 3.214477
Epoch: 83	Training Loss: 3.355598	Validation Loss: 3.171831
Epoch: 84	Training Loss: 3.373746	Validation Loss: 3.210921
Epoch: 85	Training Loss: 3.358006	Validation Loss: 3.182114
Epoch: 86	Training Loss: 3.369422	Validation Loss: 3.210329
Epoch: 87	Training Loss: 3.320504	Validation Loss: 3.127030
Epoch: 88	Training Loss: 3.344763	Validation Loss: 3.164382
Epoch: 89	Training Loss: 3.322108	Validation Loss: 3.127550
Epoch: 90	Training Loss: 3.347919	Validation Loss: 3.135397
Epoch: 91	Training Loss: 3.309011	Validation Loss: 3.179109
Epoch: 92	Training Loss: 3.353690	Validation Loss: 3.115831
Epoch: 93	Training Loss: 3.332223	Validation Loss: 3.113414
Epoch: 94	Training Loss: 3.332555	Validation Loss: 3.160191
Epoch: 95	Training Loss: 3.297693	Validation Loss: 3.257845
Epoch: 96	Training Loss: 3.315443	Validation Loss: 3.140637
Epoch: 97	Training Loss: 3.303872	Validation Loss: 3.141194
Epoch: 98	Training Loss: 3.295297	Validation Loss: 3.109508
Epoch: 99	Training Loss: 3.276360	Validation Loss: 3.082937
Epoch: 100	Training Loss: 3.321181	Validation Loss: 3.149942

(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 [14]:

```
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
        output = model.forward(data)
        # calculate the loss and average test loss
        loss = criterion(output, target)
        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
test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)

# empty some cache in GPU to ensure the running of following blocks
torch.cuda.empty_cache()
```

Test Loss: 2.934337

Test Accuracy: 28% (239/836)

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 [15]:

```
# just use the same data loaders
loaders_transfer = loaders_scratch
```

(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 [16]:

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

# use VGG16 and freeze all layers (except for the last layer)
model_transfer = models.vgg16(pretrained=True)
for param in model_transfer.features.parameters():
    param.requires_grad = False

for i in range(6):
    for param in model_transfer.classifier[i].parameters():
        param.requires_grad = False

# re-write last layer
model_transfer.classifier[6] = nn.Linear(model_transfer.classifier[6].in_features, 133)

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: Here I use the pretrained VGG16 model, and fix its layers (except for the final layer of classifier) to prevent finetuning. The final layer is replaced to fit the dog breeds classes. This will allow the model to leverage the pretrained model from imagenet.

The VGG16 model also uses a small kernel size, which works well for detailed feature extraction. Note that in my model 'model_scratch', I have also done the same thing. But VGG16 uses more depth to capture complex features (up to a depth of 512, in the 4th and 5th conv-layers of the model), which is beyond the capacity of my GPU resources and needs more data in the dog datasets. So, pre-trained model is needed here.

(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 [17]:

```
criterion_transfer = nn.CrossEntropyLoss()
optimizer_transfer = optim.Adam(model_transfer.classifier[6].parameters())
```

(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 [18]:

```
# train the model
model_transfer = train(20, loaders_transfer, model_transfer, optimizer_transfer, criterion_transfer,

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

# empty some cache in GPU to ensure the running of following blocks
torch.cuda.empty_cache()
```

Epoch: 1	Training Loss: 1.936249	Validation Loss: 0.534076
Epoch: 2	Training Loss: 0.989851	Validation Loss: 0.444353
Epoch: 3	Training Loss: 0.884100	Validation Loss: 0.402288
Epoch: 4	Training Loss: 0.838622	Validation Loss: 0.421448
Epoch: 5	Training Loss: 0.801781	Validation Loss: 0.394000
Epoch: 6	Training Loss: 0.793156	Validation Loss: 0.394425
Epoch: 7	Training Loss: 0.802340	Validation Loss: 0.383207
Epoch: 8	Training Loss: 0.741082	Validation Loss: 0.373505
Epoch: 9	Training Loss: 0.764958	Validation Loss: 0.346904
Epoch: 10	Training Loss: 0.771058	Validation Loss: 0.371586
Epoch: 11	Training Loss: 0.780182	Validation Loss: 0.367939
Epoch: 12	Training Loss: 0.738034	Validation Loss: 0.369762
Epoch: 13	Training Loss: 0.739897	Validation Loss: 0.372397
Epoch: 14	Training Loss: 0.709440	Validation Loss: 0.363485
Epoch: 15	Training Loss: 0.721071	Validation Loss: 0.385969
Epoch: 16	Training Loss: 0.709814	Validation Loss: 0.381767
Epoch: 17	Training Loss: 0.725572	Validation Loss: 0.384239
Epoch: 18	Training Loss: 0.681559	Validation Loss: 0.367523
Epoch: 19	Training Loss: 0.685077	Validation Loss: 0.375234
Epoch: 20	Training Loss: 0.688332	Validation Loss: 0.363750

(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 [19]:

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

# empty some cache in GPU to ensure the running of following blocks
torch.cuda.empty_cache()
```

Test Loss: 0.430544

Test Accuracy: 86% (720/836)

(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 [20]:

```

# 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 image_datasets['train'].classes]

def predict_breed_transfer(img_path):
    transform = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ])

    # load image and transform
    image = transform(Image.open(img_path))
    image = image.unsqueeze(0)

    # prediction
    if use_cuda:
        prediction = model_transfer.forward(Variable(image).cuda()).cpu()
    else:
        prediction = model_transfer.forward(Variable(image))

    # return the class with highest probability
    return class_names[prediction.data.numpy().argmax()]

pred_class = predict_breed_transfer('dogImages/train/001.Affenpinscher/Affenpinscher_00001.jpg')
print(pred_class)

```

Affenpinscher

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 [23]:

```
def run_app(img_path):
    pred_class = predict_breed_transfer(img_path)

    if dog_detector(img_path):
        print("This is a %s dog." % pred_class)
        return

    if face_detector(img_path):
        print("Hello, human! You look like a %s." % pred_class)
        return

    print("Error, Nothing detected!")
```

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 is better than I expected, because I think the problem is difficult.

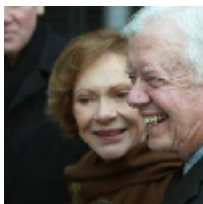
I think the following ways may help improve the algorithm:

1. more data, about 8,000 dog images is not enough for deep network like VGG.
2. training the network with both classification and location (like FasterRCNN), i.e., do not treat human/dog faces detection as a separate module, but train it with classification together. Multi-task learning usually improves the performance of deep networks.
3. pre-processing like salient object proposal, which helps the CNN to focus on a region of interest but neglect the background.

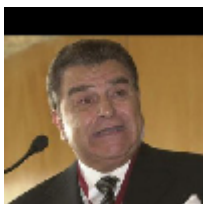
In [24]:

```
from IPython.display import display

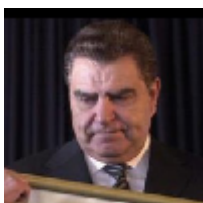
for file in np.hstack((human_files[:3], dog_files[:3])):
    display(Image.open(file).resize((100, 100)))
    run_app(file)
```



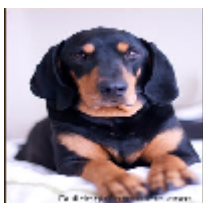
Hello, human! You look like a Bearded collie.



Hello, human! You look like a Poodle.



Hello, human! You look like a Basenji.



This is a Black and tan coonhound dog.



This is a Black and tan coonhound dog.



This is a Black and tan coonhound dog.

