# dog_app

November 21, 2018

## 1 Convolutional Neural Networks

### 1.1 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.

## Step 0: Import Datasets

Make sure that you've downloaded the required human and dog datasets: * Download the dog dataset. Unzip the folder and place it in this project's home directory, at the location `/dog_images`.

- Download the human dataset. 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 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 [5]: 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

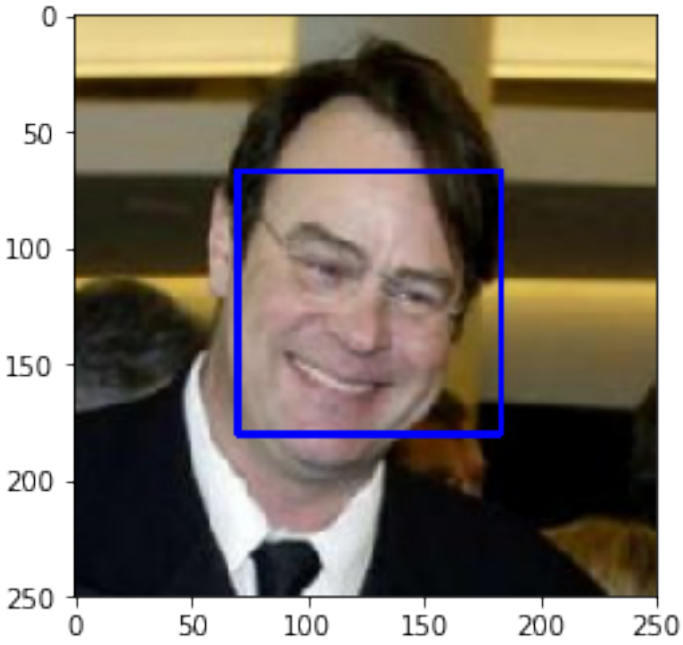
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

OpenCV provides many pre-trained face detectors, stored as XML files on github. 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 [6]: 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.

### 1.1.1 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 [7]: # 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
```

### 1.1.2 (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:**

- Percentage of humans detected: 98.0%
- Percentage of dogs detected: 17.0%

In [8]:
```
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.
humans_detected = sum([1 for file in tqdm(human_files_short) if face_detector(file)])
dogs_detected = sum([1 for file in tqdm(dog_files_short) if face_detector(file)])

print(f"Percentage of humans detected: {humans_detected/100*100}%")
print(f"Percentage of dogs detected: {dogs_detected/100*100}%")
```

```
  0%|          | 0/100 [00:00<?, ?it/s]
  4%|         | 4/100 [00:00<00:02, 35.73it/s]
  8%|         | 8/100 [00:00<00:02, 34.93it/s]
 12%|        | 12/100 [00:00<00:02, 35.32it/s]
 16%|        | 16/100 [00:00<00:02, 35.00it/s]
 20%|        | 20/100 [00:00<00:02, 34.52it/s]
 25%|       | 25/100 [00:00<00:02, 35.80it/s]
 30%|       | 30/100 [00:00<00:01, 37.89it/s]
 35%|      | 35/100 [00:00<00:01, 40.04it/s]
 40%|      | 40/100 [00:01<00:01, 39.66it/s]
 44%|     | 44/100 [00:01<00:01, 38.42it/s]
 48%|     | 48/100 [00:01<00:01, 37.83it/s]
 52%|    | 52/100 [00:01<00:01, 37.01it/s]
 56%|    | 56/100 [00:01<00:01, 37.18it/s]
```

```
 60%|    | 60/100 [00:01<00:01, 37.36it/s]
 65%|   | 65/100 [00:01<00:00, 39.50it/s]
 70%|   | 70/100 [00:01<00:00, 40.93it/s]
 75%|  | 75/100 [00:01<00:00, 43.06it/s]
 80%|  | 80/100 [00:02<00:00, 42.35it/s]
 85%| | 85/100 [00:02<00:00, 41.14it/s]
 90%| | 90/100 [00:02<00:00, 39.01it/s]
 94%|| 94/100 [00:02<00:00, 36.95it/s]
 98%|| 98/100 [00:02<00:00, 36.05it/s]
100%|| 100/100 [00:02<00:00, 38.16it/s]
  0%|          | 0/100 [00:00<?, ?it/s]
  1%|          | 1/100 [00:00<00:28,  3.52it/s]
  3%|         | 3/100 [00:00<00:20,  4.66it/s]
  4%|         | 4/100 [00:02<01:25,  1.12it/s]
  6%|         | 6/100 [00:03<01:01,  1.53it/s]
  7%|         | 7/100 [00:03<00:47,  1.97it/s]
  8%|         | 8/100 [00:03<00:37,  2.45it/s]
  9%|         | 9/100 [00:03<00:38,  2.36it/s]
 10%|         | 10/100 [00:03<00:30,  2.99it/s]
 11%|         | 11/100 [00:04<00:25,  3.54it/s]
 12%|        | 12/100 [00:04<00:21,  4.02it/s]
 13%|        | 13/100 [00:04<00:18,  4.83it/s]
 14%|        | 14/100 [00:04<00:17,  4.91it/s]
 15%|        | 15/100 [00:04<00:14,  5.78it/s]
 16%|        | 16/100 [00:04<00:12,  6.53it/s]
 17%|        | 17/100 [00:04<00:12,  6.51it/s]
 18%|        | 18/100 [00:05<00:13,  6.15it/s]
 19%|        | 19/100 [00:05<00:12,  6.52it/s]
 21%|        | 21/100 [00:05<00:11,  6.96it/s]
 22%|       | 22/100 [00:05<00:14,  5.22it/s]
 23%|       | 23/100 [00:06<00:14,  5.41it/s]
 24%|       | 24/100 [00:06<00:13,  5.65it/s]
 25%|       | 25/100 [00:06<00:12,  5.91it/s]
 26%|       | 26/100 [00:06<00:12,  5.98it/s]
 27%|       | 27/100 [00:07<00:33,  2.19it/s]
 29%|       | 29/100 [00:07<00:24,  2.85it/s]
 30%|       | 30/100 [00:07<00:19,  3.55it/s]
 31%|       | 31/100 [00:08<00:17,  4.05it/s]
 32%|      | 32/100 [00:08<00:14,  4.55it/s]
 33%|      | 33/100 [00:08<00:14,  4.62it/s]
 34%|      | 34/100 [00:08<00:12,  5.21it/s]
 35%|      | 35/100 [00:09<00:29,  2.18it/s]
 36%|      | 36/100 [00:09<00:23,  2.76it/s]
 37%|      | 37/100 [00:11<00:38,  1.62it/s]
 38%|      | 38/100 [00:11<00:28,  2.15it/s]
 39%|      | 39/100 [00:11<00:23,  2.61it/s]
 40%|      | 40/100 [00:11<00:19,  3.11it/s]
 41%|      | 41/100 [00:12<00:25,  2.31it/s]
```

```
 42%|     | 42/100 [00:12<00:20,  2.79it/s]
 43%|     | 43/100 [00:12<00:18,  3.03it/s]
 44%|     | 44/100 [00:12<00:15,  3.50it/s]
 45%|     | 45/100 [00:12<00:13,  4.13it/s]
 46%|     | 46/100 [00:13<00:16,  3.30it/s]
 47%|     | 47/100 [00:13<00:13,  3.92it/s]
 49%|     | 49/100 [00:13<00:10,  4.75it/s]
 50%|     | 50/100 [00:13<00:09,  5.09it/s]
 52%|    | 52/100 [00:14<00:08,  5.83it/s]
 53%|    | 53/100 [00:14<00:08,  5.82it/s]
 54%|    | 54/100 [00:14<00:08,  5.36it/s]
 55%|    | 55/100 [00:14<00:07,  5.96it/s]
 57%|    | 57/100 [00:15<00:07,  5.90it/s]
 59%|    | 59/100 [00:15<00:06,  6.69it/s]
 60%|    | 60/100 [00:17<00:25,  1.56it/s]
 61%|    | 61/100 [00:17<00:19,  2.00it/s]
 62%|   | 62/100 [00:17<00:18,  2.11it/s]
 64%|   | 64/100 [00:17<00:12,  2.85it/s]
 65%|   | 65/100 [00:17<00:09,  3.59it/s]
 67%|   | 67/100 [00:18<00:07,  4.60it/s]
 68%|   | 68/100 [00:18<00:06,  5.18it/s]
 69%|   | 69/100 [00:18<00:05,  5.60it/s]
 71%|   | 71/100 [00:18<00:04,  6.78it/s]
 72%|  | 72/100 [00:22<00:39,  1.39s/it]
 73%|  | 73/100 [00:22<00:27,  1.01s/it]
 74%|  | 74/100 [00:22<00:19,  1.35it/s]
 75%|  | 75/100 [00:23<00:15,  1.57it/s]
 76%|  | 76/100 [00:23<00:11,  2.03it/s]
 77%|  | 77/100 [00:23<00:09,  2.51it/s]
 78%|  | 78/100 [00:24<00:08,  2.60it/s]
 80%|  | 80/100 [00:26<00:12,  1.66it/s]
 81%|  | 81/100 [00:26<00:08,  2.14it/s]
 83%| | 83/100 [00:26<00:06,  2.72it/s]
 84%| | 84/100 [00:26<00:04,  3.27it/s]
 85%| | 85/100 [00:27<00:07,  2.14it/s]
 86%| | 86/100 [00:27<00:05,  2.69it/s]
 87%| | 87/100 [00:28<00:04,  3.22it/s]
 88%| | 88/100 [00:28<00:03,  3.11it/s]
 89%| | 89/100 [00:28<00:02,  3.76it/s]
 90%| | 90/100 [00:29<00:04,  2.23it/s]
 92%|| 92/100 [00:29<00:02,  2.98it/s]
 93%|| 93/100 [00:29<00:02,  3.48it/s]
 94%|| 94/100 [00:29<00:01,  4.27it/s]
 96%|| 96/100 [00:30<00:00,  4.96it/s]
 97%|| 97/100 [00:30<00:00,  5.81it/s]
 99%|| 99/100 [00:30<00:00,  7.03it/s]
100%|| 100/100 [00:30<00:00,  3.29it/s]
```

```
Percentage of humans detected: 98.0%
Percentage of dogs detected: 17.0%
```

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 [9]: ### (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 to detect dogs in images.

### 1.1.3 Obtain Pre-trained VGG-16 Model

The code cell below downloads the VGG-16 model, along with weights that have been trained on ImageNet, 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.

```
In [10]: import torch
         import torchvision.models as models

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

         print(VGG16)
```

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg

  0%|          | 0/553433881 [00:00<?, ?it/s]
  1%|          | 3768320/553433881 [00:00<00:14, 37667250.50it/s]
  1%|         | 7053312/553433881 [00:00<00:15, 36077296.67it/s]
  2%|         | 11558912/553433881 [00:00<00:14, 38370508.34it/s]
```

```
 3%|          | 15646720/553433881 [00:00<00:13, 39089267.56it/s]
 4%|          | 20373504/553433881 [00:00<00:12, 41135861.95it/s]
 4%|          | 23912448/553433881 [00:00<00:14, 35608900.03it/s]
 5%|          | 28237824/553433881 [00:00<00:14, 37363200.70it/s]
 6%|          | 32415744/553433881 [00:00<00:13, 38585913.50it/s]
 7%|          | 36757504/553433881 [00:00<00:13, 39699499.66it/s]
 7%|          | 40665088/553433881 [00:01<00:14, 35061868.98it/s]
 8%|          | 44294144/553433881 [00:01<00:14, 34992727.59it/s]
 9%|          | 48701440/553433881 [00:01<00:13, 37291741.18it/s]
10%|          | 52862976/553433881 [00:01<00:13, 38239553.14it/s]
10%|          | 56745984/553433881 [00:01<00:14, 33822171.75it/s]
11%|          | 60260352/553433881 [00:01<00:16, 30477486.21it/s]
12%|         | 63815680/553433881 [00:01<00:15, 31840655.20it/s]
12%|         | 67911680/553433881 [00:01<00:14, 34114824.16it/s]
13%|         | 71770112/553433881 [00:01<00:13, 35219317.49it/s]
14%|         | 75399168/553433881 [00:02<00:13, 34650974.37it/s]
14%|         | 78938112/553433881 [00:02<00:14, 31709792.45it/s]
15%|         | 83410944/553433881 [00:02<00:13, 34442959.47it/s]
16%|         | 87605248/553433881 [00:02<00:12, 36344072.65it/s]
17%|         | 91840512/553433881 [00:02<00:12, 37890068.26it/s]
17%|         | 95739904/553433881 [00:02<00:12, 36956159.41it/s]
18%|         | 100786176/553433881 [00:02<00:11, 40130895.08it/s]
19%|         | 106020864/553433881 [00:02<00:10, 42976260.97it/s]
20%|         | 110469120/553433881 [00:02<00:10, 41015543.14it/s]
21%|         | 115261440/553433881 [00:03<00:10, 42547074.67it/s]
22%|        | 120242176/553433881 [00:03<00:09, 44436234.49it/s]
23%|        | 125288448/553433881 [00:03<00:09, 45944246.58it/s]
23%|        | 129966080/553433881 [00:03<00:10, 41700054.61it/s]
24%|        | 134266880/553433881 [00:03<00:10, 41153907.76it/s]
25%|        | 139034624/553433881 [00:03<00:09, 42909184.46it/s]
26%|        | 143949824/553433881 [00:03<00:09, 44600683.90it/s]
27%|        | 148488192/553433881 [00:03<00:09, 42723038.89it/s]
28%|        | 153239552/553433881 [00:03<00:09, 44051387.20it/s]
29%|        | 158646272/553433881 [00:04<00:08, 46530771.37it/s]
30%|        | 164806656/553433881 [00:04<00:07, 49259884.84it/s]
31%|        | 169836544/553433881 [00:04<00:09, 41022727.87it/s]
31%|       | 174243840/553433881 [00:04<00:09, 41621394.28it/s]
32%|       | 179798016/553433881 [00:04<00:08, 44999631.35it/s]
33%|       | 184532992/553433881 [00:04<00:08, 45030803.46it/s]
34%|       | 190021632/553433881 [00:04<00:07, 47589885.10it/s]
35%|       | 195321856/553433881 [00:04<00:07, 47708853.21it/s]
36%|       | 200204288/553433881 [00:04<00:08, 43530497.80it/s]
38%|       | 208969728/553433881 [00:05<00:06, 51272091.10it/s]
39%|       | 214753280/553433881 [00:05<00:07, 47002199.00it/s]
40%|       | 221093888/553433881 [00:05<00:06, 50864953.34it/s]
41%|       | 226639872/553433881 [00:05<00:07, 42525046.09it/s]
42%|      | 231776256/553433881 [00:05<00:07, 44224708.41it/s]
43%|      | 239484928/553433881 [00:05<00:06, 50707097.69it/s]
```

```
44%|     | 245178368/553433881 [00:05<00:06, 51279684.92it/s]
45%|     | 250863616/553433881 [00:05<00:05, 52828607.56it/s]
46%|     | 256466944/553433881 [00:06<00:05, 52450775.42it/s]
48%|     | 264904704/553433881 [00:06<00:04, 58750226.20it/s]
49%|     | 271187968/553433881 [00:06<00:05, 54347566.00it/s]
50%|     | 276971520/553433881 [00:06<00:05, 52711304.79it/s]
51%|    | 284844032/553433881 [00:06<00:04, 58501593.18it/s]
53%|    | 291078144/553433881 [00:06<00:04, 55467077.18it/s]
54%|    | 297099264/553433881 [00:06<00:04, 56725177.51it/s]
55%|    | 302989312/553433881 [00:06<00:04, 56235260.01it/s]
56%|    | 309583872/553433881 [00:06<00:04, 58826466.23it/s]
57%|    | 315604992/553433881 [00:07<00:04, 55138226.61it/s]
58%|    | 321265664/553433881 [00:07<00:04, 53230430.95it/s]
59%|    | 328163328/553433881 [00:07<00:03, 56922508.53it/s]
60%|    | 334012416/553433881 [00:07<00:04, 50609206.02it/s]
61%|   | 339976192/553433881 [00:07<00:04, 53006174.42it/s]
63%|   | 349036544/553433881 [00:07<00:03, 60539164.90it/s]
64%|   | 355606528/553433881 [00:07<00:03, 60351455.60it/s]
66%|   | 364814336/553433881 [00:07<00:02, 67305496.96it/s]
67%|   | 372056064/553433881 [00:07<00:02, 65005654.54it/s]
68%|   | 378929152/553433881 [00:08<00:02, 61181137.13it/s]
70%|   | 388055040/553433881 [00:08<00:02, 67888527.48it/s]
71%|  | 395288576/553433881 [00:08<00:02, 66004854.32it/s]
73%|  | 402808832/553433881 [00:08<00:02, 66797130.48it/s]
74%|  | 409722880/553433881 [00:08<00:02, 67151679.22it/s]
76%|  | 418889728/553433881 [00:08<00:01, 72903403.05it/s]
77%|  | 426442752/553433881 [00:08<00:02, 55876938.52it/s]
79%|  | 435052544/553433881 [00:08<00:01, 62453364.50it/s]
80%|  | 444432384/553433881 [00:08<00:01, 69404867.83it/s]
82%| | 452173824/553433881 [00:09<00:01, 70792018.66it/s]
83%| | 459825152/553433881 [00:09<00:01, 67661592.09it/s]
84%| | 467017728/553433881 [00:09<00:01, 65182203.97it/s]
86%| | 473858048/553433881 [00:09<00:01, 63484174.01it/s]
87%| | 480444416/553433881 [00:09<00:01, 58284603.39it/s]
88%| | 486522880/553433881 [00:09<00:01, 52804740.47it/s]
89%| | 492077056/553433881 [00:09<00:01, 51095801.24it/s]
90%| | 497393664/553433881 [00:09<00:01, 44972101.23it/s]
91%| | 502169600/553433881 [00:10<00:01, 36985645.91it/s]
91%|| 506298368/553433881 [00:10<00:01, 30847554.58it/s]
92%|| 509853696/553433881 [00:10<00:01, 26533329.45it/s]
93%|| 513032192/553433881 [00:10<00:01, 27578073.39it/s]
93%|| 516112384/553433881 [00:10<00:01, 26120739.64it/s]
94%|| 519995392/553433881 [00:10<00:01, 28961369.99it/s]
95%|| 525500416/553433881 [00:10<00:00, 33757150.29it/s]
96%|| 529367040/553433881 [00:11<00:00, 34875904.37it/s]
97%|| 535511040/553433881 [00:11<00:00, 39459792.33it/s]
98%|| 543375360/553433881 [00:11<00:00, 46392111.71it/s]
99%|| 548814848/553433881 [00:11<00:00, 38351524.62it/s]
```

```
100%|| 553433881/553433881 [00:11<00:00, 47646809.53it/s]
```

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace)
    (2): Dropout(p=0.5)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace)
    (5): Dropout(p=0.5)
    (6): Linear(in_features=4096, out_features=1000, bias=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.

### 1.1.4 (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.

```
In [11]: from PIL import Image
         import torchvision.transforms as transforms

         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
             img = Image.open(img_path)
             normalize = transforms.Normalize(
                 mean=[0.485, 0.456, 0.406],
                 std=[0.229, 0.224, 0.225]
             )
             preprocess = transforms.Compose([
                 transforms.Resize(256),
                 transforms.CenterCrop(224),
                 transforms.ToTensor(),
                 normalize
             ])
             img_tensor = preprocess(img)
             img_tensor.unsqueeze_(0)

             if use_cuda:
                 img_tensor = img_tensor.cuda()

             ## Return the *index* of the predicted class for that image
             output = VGG16(img_tensor)
```

```
    if use_cuda:
        output = output.cpu()

    prediction = output.data.numpy().argmax()
    return prediction # predicted class index
```

### 1.1.5 (IMPLEMENTATION) Write a Dog Detector

While looking at the dictionary, 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 [12]: ### returns "True" if a dog is detected in the image stored at img_path
         def dog_detector(img_path):
             ## TODO: Complete the function.
             index = VGG16_predict(img_path)
             return index >= 151 and index <= 268
```

### 1.1.6 (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:**

- Percentage of dogs detected in human_files_short: 0.0%
- Percentage of dogs detected in dog_files_short: 100.0%

```
In [13]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.
         humans_detected = sum([1 for file in human_files_short if dog_detector(file)])
         dogs_detected = sum([1 for file in dog_files_short if dog_detector(file)])

         print(f"Percentage of dogs detected in human_files_short: {humans_detected/100*100}%")
         print(f"Percentage of dogs detected in dog_files_short: {dogs_detected/100*100}%")
```

```
Percentage of dogs detected in human_files_short: 0.0%
Percentage of dogs detected in dog_files_short: 100.0%
```

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

---

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

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

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

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

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

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

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

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

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!

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

Use the code cell below to write three separate data loaders 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 to be a useful resource. If you are interested in augmenting your training and/or validation data, check out the wide variety of transforms!

In [15]:
```
import os
from torchvision import datasets

# This fixes the following error:
# OSError: image file is truncated (150 bytes not processed)
# Taken from this StackOverflow issue: https://stackoverflow.com/a/23575424/1741325
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

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

num_workers = 0
batch_size = 20

normalize = transforms.Normalize(
    mean=[0.485, 0.456, 0.406],
    std=[0.229, 0.224, 0.225]
)
preprocess = transforms.Compose([
    transforms.RandomResizedCrop(224),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(10),
    transforms.ToTensor(),
    normalize
])
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    normalize
])

train_data = datasets.ImageFolder('/data/dog_images/train', transform=preprocess)
valid_data = datasets.ImageFolder('/data/dog_images/valid', transform=transform)
test_data = datasets.ImageFolder('/data/dog_images/test', transform=transform)

train_loader = torch.utils.data.DataLoader(
    train_data,
    batch_size=batch_size,
    shuffle=True,
    num_workers=num_workers
)
valid_loader = torch.utils.data.DataLoader(
    valid_data,
    batch_size=batch_size,
    shuffle=True,
    num_workers=num_workers
)
```

```
test_loader = torch.utils.data.DataLoader(
    test_data,
    batch_size=batch_size,
    shuffle=True,
    num_workers=num_workers
)
loaders_scratch = {
    'train': train_loader,
    'valid': valid_loader,
    'test': test_loader
}
```

**Question 3:** Describe your chosen procedure for preprocessing the data. - How does your code resize the images (by cropping, stretching, etc)? What size did you pick for the input tensor, and why? - Did you decide to augment the dataset? If so, how (through translations, flips, rotations, etc)? If not, why not?

**Answer**: - For the training data I, randomly resize and crop the images to 224x224 px so that some of the images will have spatial dimensions similar to the valid and test data. I decided to augment the training data set by randomly flipping the images horizontally as I don't want to overfit my predictions to the test data. I also apply random rotations of 10 degrees for the same reason. I then convert those images to tensors and normalize their values. - For the valid and test data, I resize all images to 224x224 px in order to have a fixed size input. I then convert those images to tensors and normalize their values. - I picked 224x224 px for the height and width, so that the spatial dimensions are a square as well as a number divisible by 2.

### 1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [16]: 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__()
                 dog_breeds = 133
                 ## Define layers of a CNN
                 # nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0)
                 # Convolutional Layer 1 (sees 224x224 image tensor)
                 self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
                 self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
                 self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
                 self.conv4 = nn.Conv2d(64, 128, 3, padding=1)
                 # Max Pooling layer (downsize by half)
                 self.pool = nn.MaxPool2d(2, 2)
                 # Linear layer (128 * 14 * 14 -> 500)
```

```
        self.fc1 = nn.Linear(128 * 14 * 14, 500)
        # Linear layer (500 -> 133)
        self.fc2 = nn.Linear(500, dog_breeds)
        # Prevent overfitting
        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)))
        x = self.pool(F.relu(self.conv4(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 = self.fc2(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:**

Initialization: - Conv1: load 224x224x3 tensor and output a stack of 16 feature maps - Conv2: increase the depth of conv1's output to 16 - Conv3: double the depth of conv2's output to 32 - Conv4: double the depth of conv3's output to 64 - Downsample any x-y size by 2 with the pooling layer - For the first fully connected layer, go from `(224 * (224 / 2 / 2 / 2 / 2)**2)` to 500 features - For the second fully connected layer, go from 500 to 133 features (the number of dog breeds to identify) - Add a dropout layer with a probability of 0.25 to prevent overfitting

Forward pass: - Apply the ReLu activation function to our conv1, and downsample using MaxPool2d - Apply the ReLu activation function to our conv2, and downsample using MaxPool2d - Apply the ReLu activation function to our conv3, and downsample using MaxPool2d - Apply the ReLu activation function to our conv4, and downsample using MaxPool2d - Flatten the image and add a dropout layer - Add 1st hidden layer with relu activation function - Add dropout layer - Return the 2nd hidden layer with relu activation function

### 1.1.9 (IMPLEMENTATION) Specify Loss Function and Optimizer

Use the next code cell to specify a loss function and optimizer. Save the chosen loss function as `criterion_scratch`, and the optimizer as `optimizer_scratch` below.

In [17]:
```
import torch.optim as optim

### TODO: select loss function
# CrossEntropyLoss is useful for classification tasks
criterion_scratch = nn.CrossEntropyLoss()

### TODO: select optimizer
optimizer_scratch = optim.Adam(model_scratch.parameters())
```

### 1.1.10 (IMPLEMENTATION) Train and Validate the Model

Train and validate your model in the code cell below. Save the final model parameters at filepath `'model_scratch.pt'`.

In [18]:
```
def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
    """returns trained model"""
    # initialize tracker for minimum validation loss
    valid_loss_min = np.Inf

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

        ##################
        # train the model #
        ##################
        for batch_idx, (data, target) in enumerate(loaders['train']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            # clear the gradients of all optimized variables
            optimizer.zero_grad()
            # forward pass: compute predicted outputs by passing inputs to the model
            output = model(data)
            # calculate the batch loss
            loss = criterion(output, target)
            # backward pass: compute gradient of the loss
            loss.backward()
            # perform a single optimization step
            optimizer.step()
            # update training loss
            train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss)
```

```
            ######################
            # validate the model #
            ######################
            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
                # forward pass: compute predicted outputs by passing inputs to the model
                output = model(data)
                # calculate the batch loss
                loss = criterion(output, target)
                # update average validation loss
                valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_loss)

            train_loss = train_loss/len(train_loader.dataset)
            valid_loss = valid_loss/len(valid_loader.dataset)

            # 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:
                print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.fo
                        valid_loss_min,
                        valid_loss))
                torch.save(model.state_dict(), save_path)
                valid_loss_min = valid_loss

        # return trained model
        return model


    # train the model
    model_scratch = train(
        30,
        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'))
```

```
Epoch: 1         Training Loss: 0.000728         Validation Loss: 0.005745
Validation loss decreased (inf --> 0.005745).  Saving model ...
Epoch: 2         Training Loss: 0.000708         Validation Loss: 0.005587
Validation loss decreased (0.005745 --> 0.005587).  Saving model ...
Epoch: 3         Training Loss: 0.000698         Validation Loss: 0.005524
Validation loss decreased (0.005587 --> 0.005524).  Saving model ...
Epoch: 4         Training Loss: 0.000692         Validation Loss: 0.005477
Validation loss decreased (0.005524 --> 0.005477).  Saving model ...
Epoch: 5         Training Loss: 0.000687         Validation Loss: 0.005417
Validation loss decreased (0.005477 --> 0.005417).  Saving model ...
Epoch: 6         Training Loss: 0.000679         Validation Loss: 0.005422
Epoch: 7         Training Loss: 0.000671         Validation Loss: 0.005350
Validation loss decreased (0.005417 --> 0.005350).  Saving model ...
Epoch: 8         Training Loss: 0.000665         Validation Loss: 0.005264
Validation loss decreased (0.005350 --> 0.005264).  Saving model ...
Epoch: 9         Training Loss: 0.000656         Validation Loss: 0.005196
Validation loss decreased (0.005264 --> 0.005196).  Saving model ...
Epoch: 10         Training Loss: 0.000651         Validation Loss: 0.005141
Validation loss decreased (0.005196 --> 0.005141).  Saving model ...
Epoch: 11         Training Loss: 0.000643         Validation Loss: 0.005214
Epoch: 12         Training Loss: 0.000637         Validation Loss: 0.005128
Validation loss decreased (0.005141 --> 0.005128).  Saving model ...
Epoch: 13         Training Loss: 0.000631         Validation Loss: 0.005041
Validation loss decreased (0.005128 --> 0.005041).  Saving model ...
Epoch: 14         Training Loss: 0.000625         Validation Loss: 0.005011
Validation loss decreased (0.005041 --> 0.005011).  Saving model ...
Epoch: 15         Training Loss: 0.000621         Validation Loss: 0.005063
Epoch: 16         Training Loss: 0.000613         Validation Loss: 0.005038
Epoch: 17         Training Loss: 0.000611         Validation Loss: 0.005085
Epoch: 18         Training Loss: 0.000604         Validation Loss: 0.004985
Validation loss decreased (0.005011 --> 0.004985).  Saving model ...
Epoch: 19         Training Loss: 0.000597         Validation Loss: 0.004914
Validation loss decreased (0.004985 --> 0.004914).  Saving model ...
Epoch: 20         Training Loss: 0.000591         Validation Loss: 0.004925
Epoch: 21         Training Loss: 0.000588         Validation Loss: 0.004889
Validation loss decreased (0.004914 --> 0.004889).  Saving model ...
Epoch: 22         Training Loss: 0.000583         Validation Loss: 0.004891
Epoch: 23         Training Loss: 0.000577         Validation Loss: 0.005004
Epoch: 24         Training Loss: 0.000573         Validation Loss: 0.004812
Validation loss decreased (0.004889 --> 0.004812).  Saving model ...
Epoch: 25         Training Loss: 0.000571         Validation Loss: 0.004807
Validation loss decreased (0.004812 --> 0.004807).  Saving model ...
Epoch: 26         Training Loss: 0.000569         Validation Loss: 0.004816
Epoch: 27         Training Loss: 0.000562         Validation Loss: 0.004778
Validation loss decreased (0.004807 --> 0.004778).  Saving model ...
```

```
Epoch: 28         Training Loss: 0.000559         Validation Loss: 0.004899
Epoch: 29         Training Loss: 0.000554         Validation Loss: 0.004712
Validation loss decreased (0.004778 --> 0.004712).  Saving model ...
Epoch: 30         Training Loss: 0.000554         Validation Loss: 0.004646
Validation loss decreased (0.004712 --> 0.004646).  Saving model ...
```

### 1.1.11 (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 [19]:
```
def test(loaders, model, criterion, use_cuda):

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

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

```
Test Loss: 3.801193


Test Accuracy: 12% (107/836)
```

---

## 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.

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

Use the code cell below to write three separate data loaders 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 [20]: ## TODO: Specify data loaders
         train_data = datasets.ImageFolder('/data/dog_images/train', transform=preprocess)
         valid_data = datasets.ImageFolder('/data/dog_images/valid', transform=transform)
         test_data = datasets.ImageFolder('/data/dog_images/test', transform=transform)

         train_loader = torch.utils.data.DataLoader(
             train_data,
             batch_size=batch_size,
             shuffle=True,
             num_workers=num_workers
         )
         valid_loader = torch.utils.data.DataLoader(
             valid_data,
             batch_size=batch_size,
             shuffle=True,
             num_workers=num_workers
         )
         test_loader = torch.utils.data.DataLoader(
             test_data,
             batch_size=batch_size,
             shuffle=True,
             num_workers=num_workers
         )
         loaders_transfer = {
             'train': train_loader,
             'valid': valid_loader,
             'test': test_loader
         }
```

### 1.1.13 (IMPLEMENTATION) Model Architecture

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

```
In [21]: import torchvision.models as models
         import torch.nn as nn

         ## TODO: Specify model architecture
         dog_breeds = 133
         model_transfer = models.resnet50(pretrained=True)

         #print(model_transfer)

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

         # print(model_transfer.fc)

         n_inputs = model_transfer.fc.in_features
         last_layer = nn.Linear(n_inputs, dog_breeds, bias=True)
         model_transfer.fc = last_layer

         for param in model_transfer.fc.parameters():
             param.requires_grad = True

         # print(model_transfer)
         # print(model_transfer.fc.out_features)

         if use_cuda:
             model_transfer = model_transfer.cuda()

Downloading: "https://download.pytorch.org/models/resnet50-19c8e357.pth" to /root/.torch/models/

  0%|          | 0/102502400 [00:00<?, ?it/s]
  4%|         | 4440064/102502400 [00:00<00:02, 43604035.18it/s]
  9%|         | 9027584/102502400 [00:00<00:02, 44126680.23it/s]
 14%|        | 14245888/102502400 [00:00<00:01, 46162098.91it/s]
 18%|        | 18456576/102502400 [00:00<00:01, 44738349.58it/s]
 23%|       | 23314432/102502400 [00:00<00:01, 45817737.38it/s]
 28%|       | 28459008/102502400 [00:00<00:01, 47368075.53it/s]
 34%|      | 34373632/102502400 [00:00<00:01, 47532546.88it/s]
 38%|      | 39354368/102502400 [00:00<00:01, 48182712.45it/s]
 43%|     | 43941888/102502400 [00:00<00:01, 47395852.89it/s]
 47%|     | 48521216/102502400 [00:01<00:01, 45450982.95it/s]
 52%|    | 52977664/102502400 [00:01<00:01, 42644867.36it/s]
 57%|    | 58638336/102502400 [00:01<00:00, 45947674.85it/s]
 62%|   | 63291392/102502400 [00:01<00:00, 43421058.93it/s]
 68%|   | 70189056/102502400 [00:01<00:00, 48844366.97it/s]
 74%|  | 75808768/102502400 [00:01<00:00, 50828481.63it/s]
 80%|  | 81920000/102502400 [00:01<00:00, 53505802.91it/s]
 85%| | 87547904/102502400 [00:01<00:00, 54292884.33it/s]
 92%|| 94150656/102502400 [00:01<00:00, 56818459.11it/s]
```

```
 98%|| 99958784/102502400 [00:02<00:00, 52158875.38it/s]
100%|| 102502400/102502400 [00:02<00:00, 49414535.69it/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 use a pretrained resnet-50 model as a base. I then freeze all parameters by setting `requires_grad = False`. I then take in the fully connected layer's in features and replace it with a linear layer that has 133 output features (the number of dog breeds we're trying to classify). I then substitute this for the base model's final layer and set `requires_grad = True` so that it can train this last layer appropriately.

I chose ResNet-50 to try and eliminate degrading accuracy, after having tried ResNet-18 which was not giving optimal results. According to the paper by Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, when trained on the ImageNet data set, this architecture can handle nets with a depth with more layers - far more than VGG nets - but still maintain a lower complexity.

### 1.1.14 (IMPLEMENTATION) Specify Loss Function and Optimizer

Use the next code cell to specify a loss function and optimizer. Save the chosen loss function as `criterion_transfer`, and the optimizer as `optimizer_transfer` below.

```
In [22]: criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.Adam(model_transfer.fc.parameters())
```

### 1.1.15 (IMPLEMENTATION) Train and Validate the Model

Train and validate your model in the code cell below. Save the final model parameters at filepath `'model_transfer.pt'`.

```
In [23]: # train the model
         n_epochs = 20
         model_transfer = train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer,

         # 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: 0.000401         Validation Loss: 0.001288
Validation loss decreased (inf --> 0.001288).  Saving model ...
Epoch: 2         Training Loss: 0.000214         Validation Loss: 0.001075
Validation loss decreased (0.001288 --> 0.001075).  Saving model ...
Epoch: 3         Training Loss: 0.000192         Validation Loss: 0.001025
Validation loss decreased (0.001075 --> 0.001025).  Saving model ...
Epoch: 4         Training Loss: 0.000179         Validation Loss: 0.000912
Validation loss decreased (0.001025 --> 0.000912).  Saving model ...
Epoch: 5         Training Loss: 0.000167         Validation Loss: 0.000943
Epoch: 6         Training Loss: 0.000165         Validation Loss: 0.000908
Validation loss decreased (0.000912 --> 0.000908).  Saving model ...
Epoch: 7         Training Loss: 0.000165         Validation Loss: 0.000929
Epoch: 8         Training Loss: 0.000157         Validation Loss: 0.000883
```

```
Validation loss decreased (0.000908 --> 0.000883).  Saving model ...
Epoch: 9         Training Loss: 0.000155         Validation Loss: 0.000902
Epoch: 10         Training Loss: 0.000160         Validation Loss: 0.000803
Validation loss decreased (0.000883 --> 0.000803).  Saving model ...
Epoch: 11         Training Loss: 0.000155         Validation Loss: 0.000977
Epoch: 12         Training Loss: 0.000156         Validation Loss: 0.001006
Epoch: 13         Training Loss: 0.000149         Validation Loss: 0.000974
Epoch: 14         Training Loss: 0.000149         Validation Loss: 0.001011
Epoch: 15         Training Loss: 0.000149         Validation Loss: 0.000958
Epoch: 16         Training Loss: 0.000147         Validation Loss: 0.000960
Epoch: 17         Training Loss: 0.000143         Validation Loss: 0.000867
Epoch: 18         Training Loss: 0.000151         Validation Loss: 0.000981
Epoch: 19         Training Loss: 0.000147         Validation Loss: 0.000933
Epoch: 20         Training Loss: 0.000140         Validation Loss: 0.000972
```

#### 1.1.16 (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%.

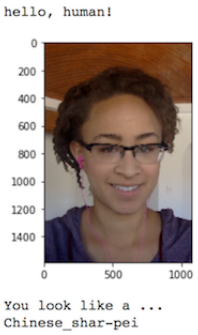
```
In [24]: test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.756263


Test Accuracy: 78% (654/836)
```

#### 1.1.17 (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 [25]: ### TODO: Write a function that takes a path to an image as input
         ### and returns the dog breed that is predicted by the model.

         # list of class names by index, i.e. a name can be accessed like class_names[0]
         class_names = [breed[4:].replace("_", " ") for breed in train_data.classes]

         def predict_breed_transfer(img_path):
             # load the image and return the predicted breed
             img = Image.open(img_path)
             preprocess = transforms.Compose([
                 transforms.Resize((224,224)),
                 transforms.ToTensor(),
                 transforms.Normalize(
                     (0.485, 0.456, 0.406),
                     (0.229, 0.224, 0.225)
```




Sample Human Output

```
    )
])
img_tensor = preprocess(img)
img_tensor.unsqueeze_(0)

if use_cuda:
    img_tensor = img_tensor.cuda()

model_transfer.eval()
output = model_transfer(img_tensor)
_, prediction = torch.max(output.data, 1)
breed = class_names[prediction-1]

return breed
```

---

## 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!

### 1.1.18 (IMPLEMENTATION) Write your Algorithm

In [26]:
```
### 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
    dog_detected = dog_detector(img_path)
```

```
    human_detected = face_detector(img_path)
    if dog_detected or human_detected:
        breed = predict_breed_transfer(img_path)
        if dog_detected:
            return f"Predicted dog breed: {breed}!"
        else:
            return f"Face detected - looks like a: {breed}"
    else:
        return 'Neither a dog nor a human was detected in the image.'
```

---

## Step 6: Test Your Algorithm

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

### 1.1.19 (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:**

Using ResNet-18, the output was worse than expected. I then tried ResNet-50. This improved the transfer learning model by a few percentage points. However, the output was worse than expected. I uploaded an image of myself which it identified as a Doberman pinscher :( I also uploaded a picture of when my dog was a puppy (olden Retriever) but he was identified as a Glen of imaal terrier.

Some possible points of improvement include:

- Apply different transformations and enhancements to the training data.
- Add more convolutions & pooling layers to my CNN.
- Try a different transfer model.

In [27]:
```
## 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
img_idx = 0
test_images = np.array(glob('app_test/*'))
# test_image_paths = [img for img in test_images]

for file in test_images:
    img_idx += 1
    img = np.asarray(Image.open(file))

    ax = plt.subplot(6, 3, img_idx)
```

```
            ax.axis('off')
            plt.imshow(img)

            print(run_app(file))
```

```
Predicted dog breed: Manchester terrier!
Predicted dog breed: Glen of imaal terrier!
Face detected - looks like a: Doberman pinscher
Predicted dog breed: Ibizan hound!
Face detected - looks like a: Doberman pinscher
Face detected - looks like a: Doberman pinscher
Face detected - looks like a: Cane corso
Predicted dog breed: Glen of imaal terrier!
```

In [ ]: