dog_app

January 10, 2021

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.

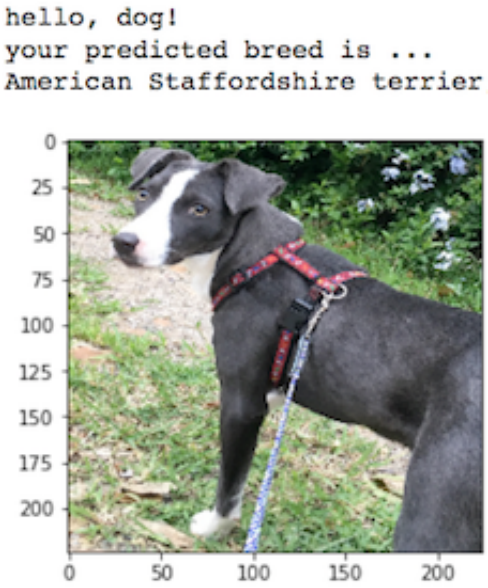
> **Note:** Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. Markdown cells can be edited by double-clicking the cell to enter edit mode.

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.

1.1.1 Why We're Here

In this notebook, you will make the first steps towards developing an algorithm that could be used as part of a mobile or web app. At the end of this project, your code will accept any user-supplied

image as input. If a dog is detected in the image, it will provide an estimate of the dog's breed. If a human is detected, it will provide an estimate of the dog breed that is most resembling. The image below displays potential sample output of your finished project (... but we expect that each student's algorithm will behave differently!).



Sample Dog Output

In this real-world setting, you will need to piece together a series of models to perform different tasks; for instance, the algorithm that detects humans in an image will be different from the CNN that infers dog breed. There are many points of possible failure, and no perfect algorithm exists. Your imperfect solution will nonetheless create a fun user experience!

1.1.2 The Road Ahead

We break the notebook into separate steps. Feel free to use the links below to navigate the notebook.

- Section **??**: Import Datasets
- Section **??**: Detect Humans
- Section **??**: Detect Dogs
- Section **??**: Create a CNN to Classify Dog Breeds (from Scratch)
- Section **??**: Create a CNN to Classify Dog Breeds (using Transfer Learning)
- Section **??**: Write your Algorithm
- Section **??**: Test Your Algorithm

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 `/dogImages`.

- 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`.

1.2 Get data on Colab

I used Colab to run my project, the following cells are just to set up the environment.

1.3 Mount Google Drive

[ ]:
```
from google.colab import drive
drive.mount('/content/drive')

%cd /content
!ls -alh
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
/content
total 24K
drwxr-xr-x 1 root root 4.0K Jan  9 18:17 .
drwxr-xr-x 1 root root 4.0K Jan  9 18:14 ..
drwxr-xr-x 1 root root 4.0K Jan  6 18:10 .config
drwx------ 5 root root 4.0K Jan  9 18:17 drive
drwxr-xr-x 7 root root 4.0K Jan  9 18:19 project-dog-classification
drwxr-xr-x 1 root root 4.0K Jan  6 18:10 sample_data
```

1.4 Get the data

[ ]:
```
%mkdir project-dog-classification
%cd project-dog-classification

! cp -a /content/drive/MyDrive/Udacity/deep-learning-v2-pytorch/
 ↪project-dog-classification/* .

! wget https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/dogImages.zip
! wget https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/lfw.zip

! unzip -qq dogImages.zip
! rm dogImages.zip

! unzip -qq lfw.zip
! rm lfw.zip
! rm -rf __MACOSX

! ls -alh
```

```
mkdir: cannot create directory project-dog-classification: File exists
/content/project-dog-classification
cp: cannot open '/content/drive/MyDrive/Udacity/deep-
learning-v2-pytorch/project-dog-classification/Report.gdoc' for reading:
Operation not supported
```

1.5 Check GPU assigned

[ ]:
```
import torch

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

if use_cuda:
  print ("Training on GPU...")
else:
  print ("Training on CPU...")

# check GPU type
gpu_info = !nvidia-smi
gpu_info = '\n'.join(gpu_info)
if gpu_info.find('failed') >= 0:
  print('Select the Runtime > "Change runtime type" menu to enable a GPU␣
 ↪accelerator, ')
  print('and then re-execute this cell.')
else:
  print(gpu_info)
```

```
Training on GPU...
Sun Jan 10 02:10:32 2021
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 460.27.04    Driver Version: 418.67       CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla V100-SXM2...  Off  | 00000000:00:04.0 Off |                    0 |
| N/A   39C    P0    24W / 300W |     10MiB / 16130MiB |      0%      Default |
|                               |                      |                 ERR! |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
```

```
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

1.6 Check memory

[ ]:
```
from psutil import virtual_memory
ram_gb = virtual_memory().total / 1e9
print('Your runtime has {:.1f} gigabytes of available RAM\n'.format(ram_gb))

if ram_gb < 20:
  print('To enable a high-RAM runtime, select the Runtime > "Change runtime␣
 ↪type"')
  print('menu, and then select High-RAM in the Runtime shape dropdown. Then, ')
  print('re-execute this cell.')
else:
  print('You are using a high-RAM runtime!')
```

```
Your runtime has 27.4 gigabytes of available RAM

You are using a high-RAM runtime!
```

1.7 Project resumes here

[ ]:
```
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

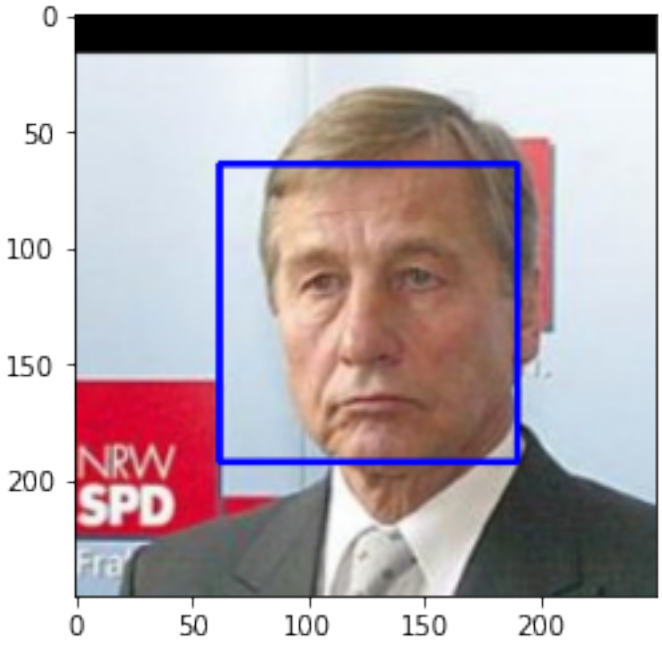
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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.

[ ]:
```
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.

1.7.1 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.

[ ]:
```
# 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
```

1.7.2 (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: 99% for humans, 14% for dogs

[ ]:
```
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.

n_humans_detected = 0
n_dogs_detected_as_humans = 0
for human, dog in zip(human_files_short, dog_files_short):
  if face_detector(human):
```

```
    n_humans_detected += 1
  if face_detector(dog):
    n_dogs_detected_as_humans += 1

print('Percentage human faces detected: {:.2f}'.format(n_humans_detected / 100.
 ↪))
print('Percentage dogs detected as human faces: {:.2f}'.
 ↪format(n_dogs_detected_as_humans / 100.))
```

```
Percentage human faces detected: 0.99
Percentage dogs detected as human faces: 0.14
```

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`.

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

1.7.3 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.

[ ]:
```
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()

if use_cuda:
  print ("Training on GPU...")
else:
  print ("Training on CPU...")

# move model to GPU if CUDA is available
```

```
if use_cuda:
    VGG16 = VGG16.cuda()

#print(VGG16)
```

```
Training on GPU...
```

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.

1.7.4 (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.

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

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

def VGG16_predict(img_path):
    '''
    Use pre-trained VGG-16 model to obtain index corresponding to
    predicted ImageNet class for image at specified path

    Args:
        img_path: path to an image

    Returns:
        Index corresponding to VGG-16 model's prediction
    '''
    # pre-process images
    transform = T.Compose([T.RandomResizedCrop(224),
                           T.ToTensor(),
                           T.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,␣
 ↪0.224, 0.225])])

    image = Image.open(img_path)
    x = transform(image)[np.newaxis, :]
    if use_cuda:
      x = x.cuda()
    return torch.argmax(VGG16(x))
```

1.7.5 (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).

[ ]:
```
### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):

    p = VGG16_predict(img_path)
    return p >= 151 and p <= 268  # ImageNet dog's categories as per␣
 ↪documentation above
```

1.7.6 (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 humans detected as dog: 2% Percentage dogs detected as dogs: 97%

[ ]:
```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.

n_humans = 0
n_dogs = 0

for human, dog in zip(human_files_short, dog_files_short):
  if dog_detector(human):
    n_humans += 1
  if dog_detector(dog):
    n_dogs += 1

print('Percentage humans detected as dog: {:.2f}'.format(n_humans / 100.))
print('Percentage dogs detected as dogs: {:.2f}'.format(n_dogs / 100.))
```

```
Percentage humans detected as dog: 0.02
Percentage dogs detected as dogs: 0.97
```

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`.

[ ]:
```
### (Optional)
### TODO: Report the performance of another pre-trained network.
```

```
### Feel free to use as many code cells as needed.
```

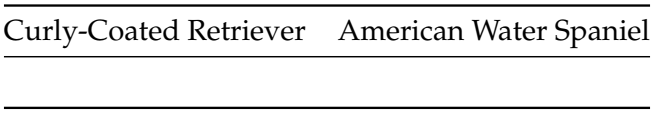
Step 3: Create a CNN to Classify Dog Breeds (from Scratch)

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.

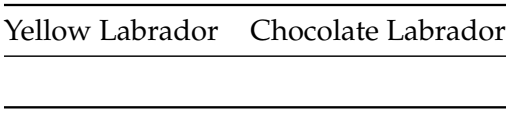
Brittany Welsh Springer Spaniel

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!

1.7.7 (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 `dogImages/train`, `dogImages/valid`, and `dogImages/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!

[ ]:
```
import torch

from torchvision import datasets
from torchvision import transforms

def get_mean_std(transforms_array, image_dir):
  """
    return the mean and std for the images in the specified dir passed in,
    after applying the specified transforms
  """
  transform = T.Compose(transforms_array + [transforms.ToTensor()])
  dataset = datasets.ImageFolder(image_dir, transform=transform)

  means = []
  stds = []
  for img,_ in dataset:
    means.append(torch.mean(img))
    stds.append(torch.std(img))

  mean = torch.mean(torch.tensor(means)).item()
  std = torch.mean(torch.tensor(stds)).item()
  return mean, std
```

[ ]:
```
import os
import torch

from torchvision import datasets
from torchvision import transforms

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

data_dir = 'dogImages/'
train_dir = os.path.join(data_dir, 'train/')
valid_dir = os.path.join(data_dir, 'valid/')
test_dir = os.path.join(data_dir, 'test/')

batch_size = 32
num_workers = 0

base_transforms = [transforms.Resize(160), transforms.CenterCrop(128)]
data_aug_transforms = [transforms.RandomHorizontalFlip(), transforms.
 ↪RandomRotation(10)]

#mean, std = get_mean_std(base_transforms, train_dir)
#print('Training data mean {:.3f}, std {:.4f}'.format(mean, std))
```

```
mean, std = (0.442, 0.2307)

normalize_transforms = [transforms.ToTensor(),
                        transforms.Normalize(mean=(mean, mean, mean), std=(std,␣
 ↪std, std))]

train_transforms = transforms.Compose(base_transforms +
                                data_aug_transforms +
                                normalize_transforms)

# No data augmentation for valid and test data
valid_transforms = transforms.Compose(base_transforms + normalize_transforms)

train_data = datasets.ImageFolder(train_dir, transform=train_transforms)
valid_data = datasets.ImageFolder(valid_dir, transform=valid_transforms)
test_data = datasets.ImageFolder(test_dir, transform=valid_transforms)

print('Num training images: ', len(train_data))
print('Num valid images: ', len(valid_data))
print('Num test images: ', len(test_data))

train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,
                                           num_workers=num_workers,␣
 ↪shuffle=True)

valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=batch_size,
                                           num_workers=num_workers,␣
 ↪shuffle=False)

test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size,
                                           num_workers=num_workers,␣
 ↪shuffle=False)

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

```
Num training images:  6680
Num valid images:  835
Num test images:  836
```

[ ]:
```
# Visualize some sample data
import matplotlib.pyplot as plt
%matplotlib inline

# obtain one batch of training images
dataiter = iter(train_loader)
```

```
images, labels = dataiter.next()
images = images.numpy() # convert images to numpy for display

def imshow(img):
    img = img * std + mean # unnormalize
    plt.imshow(np.transpose(img, (1, 2, 0)))  # convert from Tensor image

# plot the images in the batch, along with the corresponding labels
fig = plt.figure(figsize=(25, 4))
for idx in np.arange(20):
    ax = fig.add_subplot(2, 20/2, idx+1, xticks=[], yticks=[])
    imshow(images[idx])
    ax.set_title(train_data.classes[labels[idx]][5:])
```

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:

The code first resizes the images to 160, a value slightly larger than the crop size, and then center-crops to 128. I picked 128 in order to have a size that is not too large but at the same time keeps a reasonable resolution, and is a power of 2. I resized first in order to avoid losing too much information from the picture.

I've augmented the data set with random horizontal flips and rotations of up to 10 degrees.

I've also calculated the mean and std of the training data in order to have more accurate normalization, rather than using the VGG parameters, which were obtained from a subset of ImageNet, or a more generic 0.5.

1.7.8 (IMPLEMENTATION) Model Architecture

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

[ ]:
```
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__()
```

```
        # Conv Layer block 1 - width/height after MP: 128 -> 64, channels: 3 ->␣
↪32
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3,␣
↪padding=1)

        # Conv Layer block 2 - width/height after MP: 64 -> 32, channels: 32 ->␣
↪64
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3,␣
↪padding=1)

        # Conv Layer block 3 - width/height after MP: 32 -> 16, channels: 64 ->␣
↪128
        self.conv3 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3,␣
↪padding=1)

        # Conv Layer block 4 - width/height after MP: 16 -> 8, channels: 128 ->␣
↪256
        self.conv4 = nn.Conv2d(in_channels=128, out_channels=256,␣
↪kernel_size=3, padding=1)

        self.maxPool = nn.MaxPool2d(kernel_size=2, stride=2)

        self.fc1 = nn.Linear(16384, 4096)
        self.fc2 = nn.Linear(4096, 4096)
        self.fc3 = nn.Linear(4096, 133)

        self.batchNorm = nn.BatchNorm1d(16384)
        self.dropOut = nn.Dropout(p=0.5)

        # use xavier weights initialization as recommended in forums
        #for m in self.modules():
        #  if isinstance(m, nn.Linear) or isinstance(m, nn.Conv2d):
        #    nn.init.xavier_uniform_(m.weight)

    def forward(self, x):
        # conv layers
        x = self.maxPool(F.relu(self.conv1(x)))
        x = self.maxPool(F.relu(self.conv2(x)))
        x = self.maxPool(F.relu(self.conv3(x)))
        x = self.maxPool(F.relu(self.conv4(x)))

        #print(x.shape)

        # flatten + batch norm
        x = x.view(x.size(0), -1)
        x = self.batchNorm(x)
```

```
        x = self.dropOut(x)
        x = F.relu(self.fc1(x))

        x = self.dropOut(x)
        x = F.relu(self.fc2(x))

        x = self.dropOut(x)
        x = self.fc3(x)
        #print(x.shape)

        return x

#-#-# You do NOT have to modify the code below this line. #-#-#

# instantiate the CNN
model_scratch = Net()
print(model_scratch)

# move tensors to GPU if CUDA is available
if use_cuda:
    model_scratch.cuda()
    print(torch.cuda.memory_summary(device=None, abbreviated=False))
```

```
Net(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv4): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (maxPool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (fc1): Linear(in_features=16384, out_features=4096, bias=True)
  (fc2): Linear(in_features=4096, out_features=4096, bias=True)
  (fc3): Linear(in_features=4096, out_features=133, bias=True)
  (batchNorm): BatchNorm1d(16384, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (dropOut): Dropout(p=0.5, inplace=False)
)
|===========================================================================|
|                  PyTorch CUDA memory summary, device ID 0                 |
|---------------------------------------------------------------------------|
|            CUDA OOMs: 0            |        cudaMalloc retries: 0         |
|===========================================================================|
|        Metric         | Cur Usage  | Peak Usage | Tot Alloc  | Tot Freed  |
|---------------------------------------------------------------------------|
| Allocated memory      |     853 MB |     853 MB |   47403 MB |   46549 MB |
|       from large pool |     852 MB |     852 MB |   46290 MB |   45438 MB |
```

```
|       from small pool |       1 MB |       5 MB |    1112 MB |    1111 MB |
|---------------------------------------------------------------------------|
| Active memory         |     853 MB |     853 MB |   47403 MB |   46549 MB |
|       from large pool |     852 MB |     852 MB |   46290 MB |   45438 MB |
|       from small pool |       1 MB |       5 MB |    1112 MB |    1111 MB |
|---------------------------------------------------------------------------|
| GPU reserved memory   |     972 MB |     972 MB |     972 MB |       0 B  |
|       from large pool |     966 MB |     966 MB |     966 MB |       0 B  |
|       from small pool |       6 MB |       6 MB |       6 MB |       0 B  |
|---------------------------------------------------------------------------|
| Non-releasable memory |   14580 KB |   29123 KB |   21440 MB |   21425 MB |
|       from large pool |   14256 KB |   26688 KB |   19502 MB |   19488 MB |
|       from small pool |     324 KB |    3027 KB |    1937 MB |    1937 MB |
|---------------------------------------------------------------------------|
| Allocations           |      51    |      65    |    9649    |    9598    |
|       from large pool |      16    |      28    |    5016    |    5000    |
|       from small pool |      35    |      38    |    4633    |    4598    |
|---------------------------------------------------------------------------|
| Active allocs         |      51    |      65    |    9649    |    9598    |
|       from large pool |      16    |      28    |    5016    |    5000    |
|       from small pool |      35    |      38    |    4633    |    4598    |
|---------------------------------------------------------------------------|
| GPU reserved segments |      17    |      17    |      17    |       0    |
|       from large pool |      14    |      14    |      14    |       0    |
|       from small pool |       3    |       3    |       3    |       0    |
|---------------------------------------------------------------------------|
| Non-releasable allocs |       3    |       9    |    4205    |    4202    |
|       from large pool |       2    |       6    |    2604    |    2602    |
|       from small pool |       1    |       4    |    1601    |    1600    |
|===========================================================================|
```

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

Answer:

I've created a architecture with 4 convolutional layers and 3 fully connected linear layers.

The convolutional layers use kernel size set to 3 and padding set to 1 so as to preserve the dimensions, whereas the max pool layers use kernel size and stride set to 2 so as to half the dimensions. The four convolutional layers therefore reduce the dimentisons from 128 to 8. The channels are instead increased from 3 to 256 with steps of 32, 64, 128 and 256.

The fully connected layers have an intermediate layer at 4096. I've also added batch normalization and drop out.

I experimented with initial parameters initialization but had no improvement and hence reverted to default initialization.

The dropout is quite high because I noticed overfitting, although that was before adding batch normalization so it's possible that it could be reduced somewhat.

I got 18% accuracy with this architecture and therefore stopped experimenting futher.

1.7.9 (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.

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

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

### TODO: select optimizer
optimizer_scratch = optim.SGD(model_scratch.parameters(), lr = 0.001)
#optimizer_scratch = optim.Adadelta(model_scratch.parameters())
```

1.7.10 (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'`.

[ ]:
```
# the following import is required for training to be robust to truncated␣
 ↪images

from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

from datetime import datetime

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

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

        ###################
        # train the model #
        ###################
        model.train()
        for batch_idx, (data, target) in enumerate(loaders['train']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()

            ## find the loss and update the model parameters accordingly
            optimizer.zero_grad()
            output = model(data)
```

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

            ## record the average training loss
            ##train_loss += ((1 / (batch_idx + 1)) * (loss.data - train_loss))
            train_loss += loss.item() * data.size(0)


        ######################
        # validate the model #
        ######################
        model.eval()
        for batch_idx, (data, target) in enumerate(loaders['valid']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## update the average validation loss
            output = model(data)
            loss = criterion(output, target)
            #valid_loss += ((1 / (batch_idx + 1)) * (loss.data - valid_loss))
            valid_loss += loss.item()*data.size(0)

        # calculate average losses
        train_loss = train_loss/len(train_loader.sampler)
        valid_loss = valid_loss/len(valid_loader.sampler)

        # 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 .
↪..'.format(
          #  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(100, loaders_scratch, model_scratch, optimizer_scratch,
                      criterion_scratch, use_cuda, 'model_scratch.pt')

%cp model_scratch.pt /content/drive/MyDrive/Udacity/deep-learning-v2-pytorch/
 ↪project-dog-classification

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

```
Epoch: 1        Training Loss: 4.213442         Validation Loss: 4.275726
Epoch: 2        Training Loss: 4.209490         Validation Loss: 4.264143
Epoch: 3        Training Loss: 4.186931         Validation Loss: 4.241871
Epoch: 4        Training Loss: 4.166542         Validation Loss: 4.227767
Epoch: 5        Training Loss: 4.152678         Validation Loss: 4.213223
Epoch: 6        Training Loss: 4.122827         Validation Loss: 4.193310
Epoch: 7        Training Loss: 4.114234         Validation Loss: 4.175648
Epoch: 8        Training Loss: 4.089961         Validation Loss: 4.167672
Epoch: 9        Training Loss: 4.082011         Validation Loss: 4.147907
Epoch: 10       Training Loss: 4.043008         Validation Loss: 4.151161
Epoch: 11       Training Loss: 4.033876         Validation Loss: 4.120167
Epoch: 12       Training Loss: 4.015846         Validation Loss: 4.103840
Epoch: 13       Training Loss: 3.998983         Validation Loss: 4.094554
Epoch: 14       Training Loss: 3.980329         Validation Loss: 4.073943
Epoch: 15       Training Loss: 3.967782         Validation Loss: 4.078675
Epoch: 16       Training Loss: 3.948219         Validation Loss: 4.057493
Epoch: 17       Training Loss: 3.937939         Validation Loss: 4.052180
Epoch: 18       Training Loss: 3.904809         Validation Loss: 4.024696
Epoch: 19       Training Loss: 3.899450         Validation Loss: 4.011857
Epoch: 20       Training Loss: 3.892359         Validation Loss: 4.038985
Epoch: 21       Training Loss: 3.867366         Validation Loss: 4.020607
Epoch: 22       Training Loss: 3.841542         Validation Loss: 3.989343
Epoch: 23       Training Loss: 3.833877         Validation Loss: 3.957730
Epoch: 24       Training Loss: 3.809459         Validation Loss: 3.953715
Epoch: 25       Training Loss: 3.794811         Validation Loss: 3.930320
Epoch: 26       Training Loss: 3.770808         Validation Loss: 3.921543
Epoch: 27       Training Loss: 3.762427         Validation Loss: 4.013195
Epoch: 28       Training Loss: 3.751422         Validation Loss: 3.895144
Epoch: 29       Training Loss: 3.723111         Validation Loss: 3.920633
Epoch: 30       Training Loss: 3.717900         Validation Loss: 3.882482
Epoch: 31       Training Loss: 3.698409         Validation Loss: 3.905858
Epoch: 32       Training Loss: 3.674988         Validation Loss: 3.927136
Epoch: 33       Training Loss: 3.663818         Validation Loss: 3.856540
Epoch: 34       Training Loss: 3.665636         Validation Loss: 3.842077
Epoch: 35       Training Loss: 3.631317         Validation Loss: 3.828759
Epoch: 36       Training Loss: 3.625916         Validation Loss: 3.824209
Epoch: 37       Training Loss: 3.594484         Validation Loss: 3.880293
```

```
Epoch: 38       Training Loss: 3.575470         Validation Loss: 3.832881
Epoch: 39       Training Loss: 3.573345         Validation Loss: 3.838829
Epoch: 40       Training Loss: 3.571329         Validation Loss: 3.903978
Epoch: 41       Training Loss: 3.532394         Validation Loss: 3.778379
Epoch: 42       Training Loss: 3.537697         Validation Loss: 3.773724
Epoch: 43       Training Loss: 3.519698         Validation Loss: 3.765071
Epoch: 44       Training Loss: 3.511332         Validation Loss: 3.764916
Epoch: 45       Training Loss: 3.474725         Validation Loss: 3.730516
Epoch: 46       Training Loss: 3.462928         Validation Loss: 3.754481
Epoch: 47       Training Loss: 3.462978         Validation Loss: 3.715966
Epoch: 48       Training Loss: 3.428128         Validation Loss: 3.729748
Epoch: 49       Training Loss: 3.419787         Validation Loss: 3.707428
Epoch: 50       Training Loss: 3.397734         Validation Loss: 3.792447
Epoch: 51       Training Loss: 3.387007         Validation Loss: 3.657484
Epoch: 52       Training Loss: 3.369203         Validation Loss: 3.675496
Epoch: 53       Training Loss: 3.369098         Validation Loss: 3.659058
Epoch: 54       Training Loss: 3.338314         Validation Loss: 3.651053
Epoch: 55       Training Loss: 3.344469         Validation Loss: 3.641096
Epoch: 56       Training Loss: 3.310745         Validation Loss: 3.610890
Epoch: 57       Training Loss: 3.285555         Validation Loss: 3.630297
Epoch: 58       Training Loss: 3.282943         Validation Loss: 3.605442
Epoch: 59       Training Loss: 3.260860         Validation Loss: 3.589093
Epoch: 60       Training Loss: 3.248093         Validation Loss: 3.604250
Epoch: 61       Training Loss: 3.239595         Validation Loss: 3.556570
Epoch: 62       Training Loss: 3.214210         Validation Loss: 3.551086
Epoch: 63       Training Loss: 3.197039         Validation Loss: 3.569344
Epoch: 64       Training Loss: 3.175313         Validation Loss: 3.535691
Epoch: 65       Training Loss: 3.179964         Validation Loss: 3.644741
Epoch: 66       Training Loss: 3.151308         Validation Loss: 3.597156
Epoch: 67       Training Loss: 3.142454         Validation Loss: 3.725059
Epoch: 68       Training Loss: 3.128034         Validation Loss: 3.539426
Epoch: 69       Training Loss: 3.117319         Validation Loss: 3.581037
Epoch: 70       Training Loss: 3.108791         Validation Loss: 3.562429
Epoch: 71       Training Loss: 3.076000         Validation Loss: 3.501242
Epoch: 72       Training Loss: 3.078617         Validation Loss: 3.558385
Epoch: 73       Training Loss: 3.059233         Validation Loss: 3.488347
Epoch: 74       Training Loss: 3.054356         Validation Loss: 3.681603
Epoch: 75       Training Loss: 3.039527         Validation Loss: 3.543621
Epoch: 76       Training Loss: 3.022353         Validation Loss: 3.485885
Epoch: 77       Training Loss: 3.006807         Validation Loss: 3.496915
Epoch: 78       Training Loss: 3.000648         Validation Loss: 3.494311
Epoch: 79       Training Loss: 2.978202         Validation Loss: 3.469681
Epoch: 80       Training Loss: 2.952380         Validation Loss: 3.450315
Epoch: 81       Training Loss: 2.954918         Validation Loss: 3.440937
Epoch: 82       Training Loss: 2.939680         Validation Loss: 3.598203
Epoch: 83       Training Loss: 2.929989         Validation Loss: 3.455385
Epoch: 84       Training Loss: 2.911360         Validation Loss: 3.481536
Epoch: 85       Training Loss: 2.893280         Validation Loss: 3.471070
```

```
Epoch: 86       Training Loss: 2.883705         Validation Loss: 3.380924
Epoch: 87       Training Loss: 2.869690         Validation Loss: 3.451198
Epoch: 88       Training Loss: 2.855110         Validation Loss: 3.387254
Epoch: 89       Training Loss: 2.846319         Validation Loss: 3.395308
Epoch: 90       Training Loss: 2.823235         Validation Loss: 3.575538
Epoch: 91       Training Loss: 2.818140         Validation Loss: 3.479967
Epoch: 92       Training Loss: 2.803805         Validation Loss: 3.381556
Epoch: 93       Training Loss: 2.805637         Validation Loss: 3.440576
Epoch: 94       Training Loss: 2.772539         Validation Loss: 3.467289
Epoch: 95       Training Loss: 2.779428         Validation Loss: 3.387725
Epoch: 96       Training Loss: 2.758164         Validation Loss: 3.382800
Epoch: 97       Training Loss: 2.736392         Validation Loss: 3.429862
Epoch: 98       Training Loss: 2.728565         Validation Loss: 3.383054
Epoch: 99       Training Loss: 2.719252         Validation Loss: 3.417264
Epoch: 100      Training Loss: 2.678961         Validation Loss: 3.399731
```

[ ]:
```
<All keys matched successfully>
```

1.7.11 (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%.

[ ]:
```python
def test(loaders, model, criterion, use_cuda):

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

    with torch.no_grad():
      model.eval()
      for batch_idx, (data, target) in enumerate(loaders['test']):
          # move to GPU
          if use_cuda:
              data, target = data.cuda(), target.cuda()
          # forward pass: compute predicted outputs by passing inputs to the␣
 ↪model
          output = model(data)
          # calculate the loss
          loss = criterion(output, target)
          # update average test loss
          #test_loss += ((1 / (batch_idx + 1)) * (loss.data - test_loss))
          test_loss += loss.item()*data.size(0)
          # 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)

    test_loss = test_loss/len(loaders['test'].sampler)
    print('Test Loss: {:.6f}\n'.format(test_loss))

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

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

# call test function
test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.367971


Test Accuracy: 21% (177/836)
```

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.

1.7.12 (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.

[ ]:
```
## TODO: Specify data loaders for transfer learning solution, similar to what's␣
 ↪done in step3, just change
## image preprocessing and increase batch size as per paper (https://arxiv.org/
 ↪pdf/1409.1556.pdf)

import os
import torch

from torchvision import datasets
from torchvision import transforms

data_dir = 'dogImages/'
train_dir = os.path.join(data_dir, 'train/')
valid_dir = os.path.join(data_dir, 'valid/')
```

```
test_dir = os.path.join(data_dir, 'test/')

batch_size = 64
num_workers = 0

transform = T.Compose([T.Resize(256),
                       T.CenterCrop(224),
                       T.ToTensor(),
                       T.Normalize(mean=[0.485, 0.456, 0.406],
                                   std=[0.229, 0.224, 0.225])])

train_data = datasets.ImageFolder(train_dir, transform=transform)
valid_data = datasets.ImageFolder(valid_dir, transform=transform)
test_data = datasets.ImageFolder(test_dir, transform=transform)

print('Num training images: ', len(train_data))
print('Num valid images: ', len(valid_data))
print('Num test images: ', len(test_data))

train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,
                                           num_workers=num_workers,␣
 ↪shuffle=True)

valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=batch_size,
                                           num_workers=num_workers,␣
 ↪shuffle=False)

test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size,
                                           num_workers=num_workers,␣
 ↪shuffle=False)

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

```
Num training images:  6680
Num valid images:  835
Num test images:  836
```

1.7.13 (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`.

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

model_transfer = models.vgg19(pretrained=True)
```

```
print(model_transfer)

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

n_inputs = model_transfer.classifier[6].in_features
last_layer = nn.Linear(n_inputs, 133)
model_transfer.classifier[6] = last_layer

print("Modified classifier: {}".format(model_transfer.classifier))

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

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (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=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (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=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (17): ReLU(inplace=True)
    (18): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (19): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (24): ReLU(inplace=True)
    (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (26): ReLU(inplace=True)
    (27): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
```

```
ceil_mode=False)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (31): ReLU(inplace=True)
    (32): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (33): ReLU(inplace=True)
    (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (35): ReLU(inplace=True)
    (36): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
Modified classifier: Sequential(
  (0): Linear(in_features=25088, out_features=4096, bias=True)
  (1): ReLU(inplace=True)
  (2): Dropout(p=0.5, inplace=False)
  (3): Linear(in_features=4096, out_features=4096, bias=True)
  (4): ReLU(inplace=True)
  (5): Dropout(p=0.5, inplace=False)
  (6): Linear(in_features=4096, out_features=133, bias=True)
)
```

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:

Very deep neural networks with small filter should perform well with this task given that they were the winners for ImageNet in 2014, so I picked the deepest, VGG19 and decided to retrain the fully connected layers to specialize it on dog breeds.

1.7.14 (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.

[ ]:
```
criterion_transfer = nn.CrossEntropyLoss()
optimizer_transfer = optim.SGD(model_transfer.classifier.parameters(), lr=0.001)
```

1.7.15 (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'`.

[ ]:
```
# train the model

# The model parameters will be saved to this file
model_transfer = train(50, loaders_transfer, model_transfer,␣
 ↪optimizer_transfer, criterion_transfer, use_cuda, 'model_transfer.pt')
%cp model_transfer.pt /content/drive/MyDrive/Udacity/deep-learning-v2-pytorch/
 ↪project-dog-classification

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

```
Epoch: 1        Training Loss: 4.608812         Validation Loss: 4.015163
Epoch: 2        Training Loss: 3.688275         Validation Loss: 3.118357
Epoch: 3        Training Loss: 2.822104         Validation Loss: 2.268434
Epoch: 4        Training Loss: 2.100472         Validation Loss: 1.619323
Epoch: 5        Training Loss: 1.575391         Validation Loss: 1.213565
Epoch: 6        Training Loss: 1.234933         Validation Loss: 0.961211
Epoch: 7        Training Loss: 1.038029         Validation Loss: 0.809496
Epoch: 8        Training Loss: 0.900965         Validation Loss: 0.709751
Epoch: 9        Training Loss: 0.818150         Validation Loss: 0.641758
Epoch: 10       Training Loss: 0.738214         Validation Loss: 0.595038
Epoch: 11       Training Loss: 0.687683         Validation Loss: 0.553164
Epoch: 12       Training Loss: 0.639502         Validation Loss: 0.524457
Epoch: 13       Training Loss: 0.613993         Validation Loss: 0.499590
Epoch: 14       Training Loss: 0.573933         Validation Loss: 0.485077
Epoch: 15       Training Loss: 0.545640         Validation Loss: 0.466856
Epoch: 16       Training Loss: 0.516461         Validation Loss: 0.454195
Epoch: 17       Training Loss: 0.504500         Validation Loss: 0.437418
Epoch: 18       Training Loss: 0.483047         Validation Loss: 0.430763
Epoch: 19       Training Loss: 0.461202         Validation Loss: 0.416344
Epoch: 20       Training Loss: 0.444239         Validation Loss: 0.407965
Epoch: 21       Training Loss: 0.432330         Validation Loss: 0.404256
Epoch: 22       Training Loss: 0.426797         Validation Loss: 0.396348
Epoch: 23       Training Loss: 0.396784         Validation Loss: 0.392552
Epoch: 24       Training Loss: 0.401540         Validation Loss: 0.385789
Epoch: 25       Training Loss: 0.379725         Validation Loss: 0.383760
Epoch: 26       Training Loss: 0.373309         Validation Loss: 0.380584
Epoch: 27       Training Loss: 0.361192         Validation Loss: 0.373664
Epoch: 28       Training Loss: 0.362099         Validation Loss: 0.369391
Epoch: 29       Training Loss: 0.349253         Validation Loss: 0.369375
Epoch: 30       Training Loss: 0.332869         Validation Loss: 0.365187
Epoch: 31       Training Loss: 0.325682         Validation Loss: 0.358383
Epoch: 32       Training Loss: 0.323413         Validation Loss: 0.363829
Epoch: 33       Training Loss: 0.314641         Validation Loss: 0.360208
```

```
Epoch: 34       Training Loss: 0.305631         Validation Loss: 0.353269
Epoch: 35       Training Loss: 0.297706         Validation Loss: 0.353830
Epoch: 36       Training Loss: 0.286893         Validation Loss: 0.351141
Epoch: 37       Training Loss: 0.282620         Validation Loss: 0.350017
Epoch: 38       Training Loss: 0.281323         Validation Loss: 0.349990
Epoch: 39       Training Loss: 0.284985         Validation Loss: 0.342053
Epoch: 40       Training Loss: 0.274616         Validation Loss: 0.341172
Epoch: 41       Training Loss: 0.264713         Validation Loss: 0.341487
Epoch: 42       Training Loss: 0.262014         Validation Loss: 0.344237
Epoch: 43       Training Loss: 0.260041         Validation Loss: 0.341495
Epoch: 44       Training Loss: 0.248595         Validation Loss: 0.344406
Epoch: 45       Training Loss: 0.250119         Validation Loss: 0.340515
Epoch: 46       Training Loss: 0.241131         Validation Loss: 0.333723
Epoch: 47       Training Loss: 0.236796         Validation Loss: 0.330791


      ␣
↪------------------------------------------------------------------------

      KeyboardInterrupt                         Traceback (most recent call␣
↪last)

      <ipython-input-71-bebc742afbe5> in <module>()
        2
        3 # The model parameters will be saved to this file
   ----> 4 model_transfer = train(100, loaders_transfer, model_transfer,␣
↪optimizer_transfer, criterion_transfer, use_cuda, 'model_transfer.pt')
        5 get_ipython().magic('cp model_transfer.pt /content/drive/MyDrive/
↪Udacity/deep-learning-v2-pytorch/project-dog-classification')
        6


      <ipython-input-56-702892e84b8d> in train(n_epochs, loaders, model,␣
↪optimizer, criterion, use_cuda, save_path)
       20         ###################
       21         model.train()
   ---> 22         for batch_idx, (data, target) in enumerate(loaders['train']):
       23             # move to GPU
       24             if use_cuda:


      /usr/local/lib/python3.6/dist-packages/torch/utils/data/dataloader.py in␣
↪__next__(self)
      433         if self._sampler_iter is None:
      434             self._reset()
   --> 435         data = self._next_data()
      436         self._num_yielded += 1
      437         if self._dataset_kind == _DatasetKind.Iterable and \
```

```
      /usr/local/lib/python3.6/dist-packages/torch/utils/data/dataloader.py in␣
↪_next_data(self)
      473     def _next_data(self):
      474         index = self._next_index()  # may raise StopIteration
  --> 475         data = self._dataset_fetcher.fetch(index)  # may raise␣
↪StopIteration
      476         if self._pin_memory:
      477             data = _utils.pin_memory.pin_memory(data)


      /usr/local/lib/python3.6/dist-packages/torch/utils/data/_utils/fetch.py␣
↪in fetch(self, possibly_batched_index)
       42     def fetch(self, possibly_batched_index):
       43         if self.auto_collation:
  ---> 44             data = [self.dataset[idx] for idx in␣
↪possibly_batched_index]
       45         else:
       46             data = self.dataset[possibly_batched_index]


      /usr/local/lib/python3.6/dist-packages/torch/utils/data/_utils/fetch.py␣
↪in <listcomp>(.0)
       42     def fetch(self, possibly_batched_index):
       43         if self.auto_collation:
  ---> 44             data = [self.dataset[idx] for idx in␣
↪possibly_batched_index]
       45         else:
       46             data = self.dataset[possibly_batched_index]


      /usr/local/lib/python3.6/dist-packages/torchvision/datasets/folder.py in␣
↪__getitem__(self, index)
      149         """
      150         path, target = self.samples[index]
  --> 151         sample = self.loader(path)
      152         if self.transform is not None:
      153             sample = self.transform(sample)


      /usr/local/lib/python3.6/dist-packages/torchvision/datasets/folder.py in␣
↪default_loader(path)
      186         return accimage_loader(path)
      187     else:
  --> 188         return pil_loader(path)
      189
```

```
      190


      /usr/local/lib/python3.6/dist-packages/torchvision/datasets/folder.py in␣
↪pil_loader(path)
      168     with open(path, 'rb') as f:
      169         img = Image.open(f)
  --> 170         return img.convert('RGB')
      171
      172


      /usr/local/lib/python3.6/dist-packages/PIL/Image.py in convert(self,␣
↪mode, matrix, dither, palette, colors)
      871         """
      872
  --> 873         self.load()
      874
      875         if not mode and self.mode == "P":


      /usr/local/lib/python3.6/dist-packages/PIL/ImageFile.py in load(self)
      249
      250                             b = b + s
  --> 251                             n, err_code = decoder.decode(b)
      252                             if n < 0:
      253                                 break


      KeyboardInterrupt:
```

1.7.16 (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%.

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

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

```
Test Loss: 0.398667


Test Accuracy: 89% (745/836)
```

1.7.17 (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.

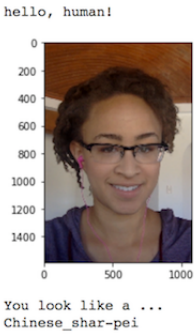
[ ]:
```
### TODO: Write a function that takes a path to an image as input
### and returns the dog breed that is predicted by the model.

from PIL import Image
import torchvision.transforms as T

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

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

def predict_breed_transfer(img_path):
    # load the image and return the predicted breed
    image = Image.open(img_path)
    x = transform(image)[np.newaxis, :]
    if use_cuda:
      x = x.cuda()
    return class_names[torch.argmax(model_transfer(x))]
```

Step 5: Write your Algorithm

Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then, - if a **dog** is detected in the image, return the predicted breed. - if a **human** is detected in the image, return the resembling dog breed. - if **neither** is detected in the image, provide output that indicates an error.

You are welcome to write your own functions for detecting humans and dogs in images, but feel free to use the `face_detector` and `dog_detector` functions developed above. You are **required** to use your CNN from Step 4 to predict dog breed.

Some sample output for our algorithm is provided below, but feel free to design your own user experience!



Sample Human Output

1.7.18 (IMPLEMENTATION) Write your Algorithm

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

def show_image(img_path):
    #print("Image path: {}".format(img_path))
    img = cv2.imread(img_path)
    cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    plt.imshow(cv_rgb)
    plt.show()

def run_app(img_path):
    if dog_detector(img_path):
      print("Hello k9!")
      show_image(img_path)
      print("Predicted dog breed is {}".
 ↪format(predict_breed_transfer(img_path)))
    elif face_detector(img_path):
      print("Hello human!")
      show_image(img_path)
      print("Resembling dog breed is {}".
 ↪format(predict_breed_transfer(img_path)))
    else:
      print("Hello who are you?")
      show_image(img_path)
      print("Resembling dog breed is {}".
 ↪format(predict_breed_transfer(img_path)))

    print("\n")
```

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?

1.7.19 (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: Object detection should be improved, for example by using some algorightms such as Mask or Faster R-CNN or even YOLO. One possibile tutorial is here: https://pytorch.org/tutorials/intermediate/torchvision_tutorial.html. At the moment, the object detection algorithm mistakenly classified a dog and a cat, 2 errors out of 6 is a bit too high.

The algorithm could print the top 5 breed probabilities, rather than only the top one. This would work much better for the two dogs tested, since they are both mixed-breed.

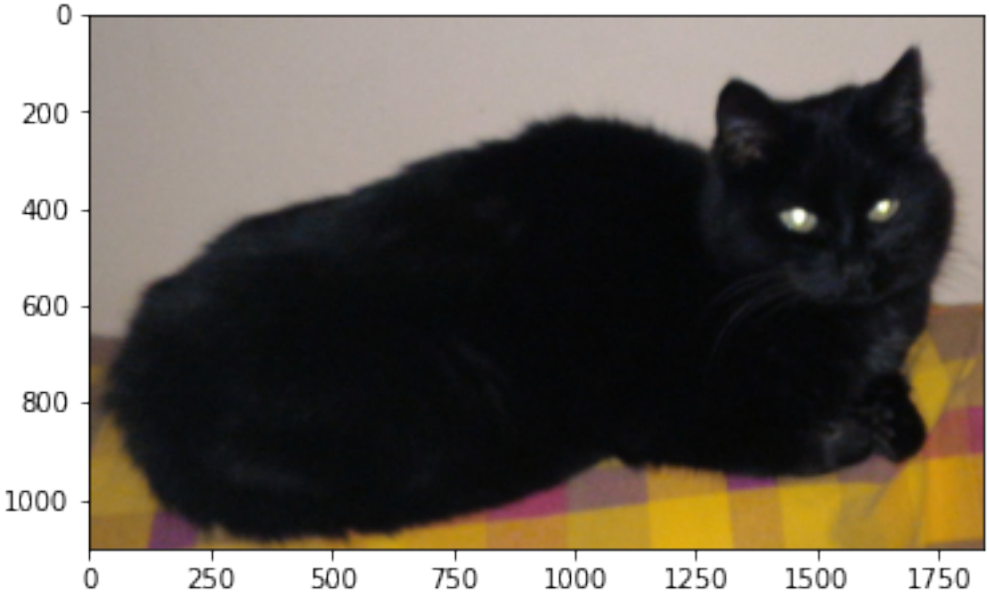
The learning rate and optimizer I used for training the networks, both the one from scratch and VGG19, could be tuned further. Dropout is possibly too high but I observed significant overfitting with a dropout of 0.1, so I chose 0.5 similarly to the VGG paper. I did not have any luck with explicity parameter initialization in the network from scratch.

[ ]:
```
## 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.

own_files = np.array(glob("own_images/*"))

## suggested code, below
for file in own_files:
    run_app(file)
```

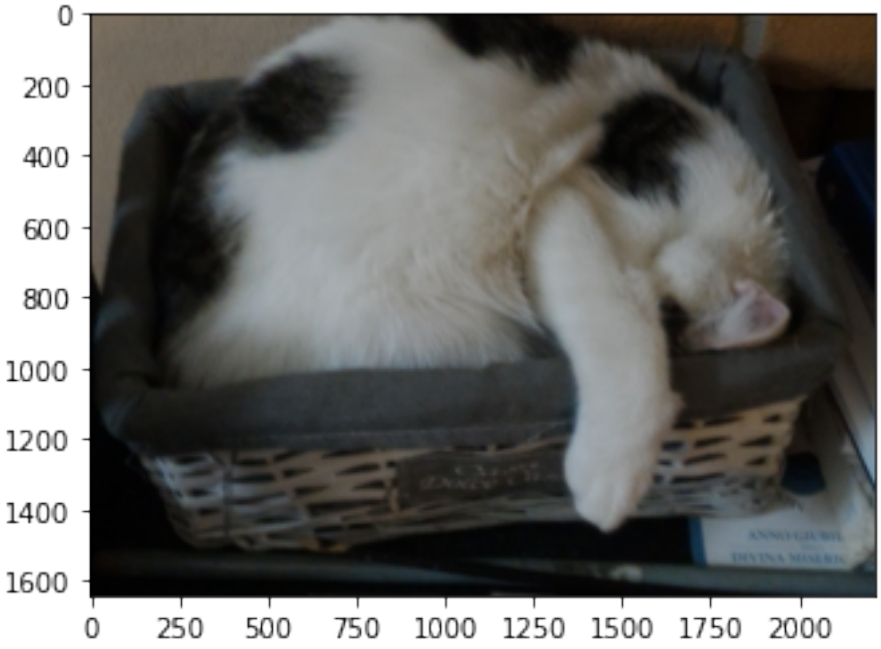
```
Hello k9!
```


```
Predicted dog breed is Black and tan coonhound
```

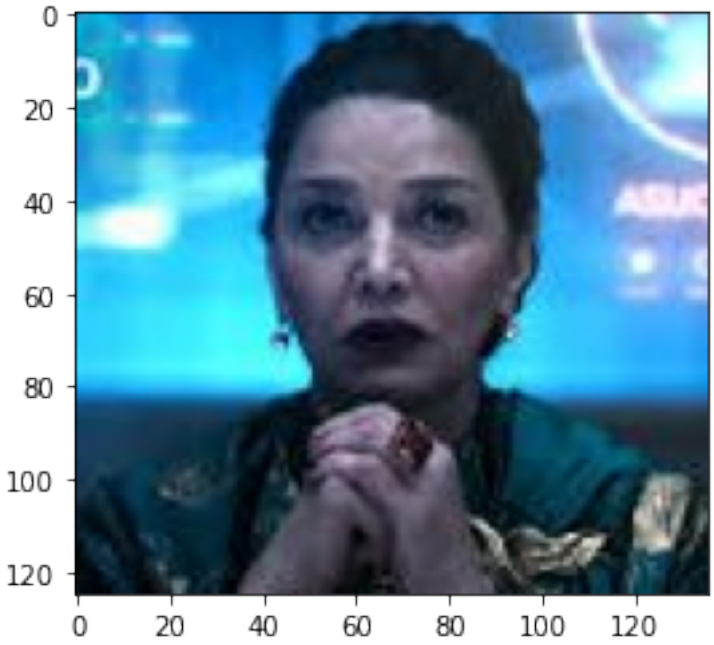
```
Hello who are you?
```



```
Resembling dog breed is Cardigan welsh corgi


Hello k9!
```

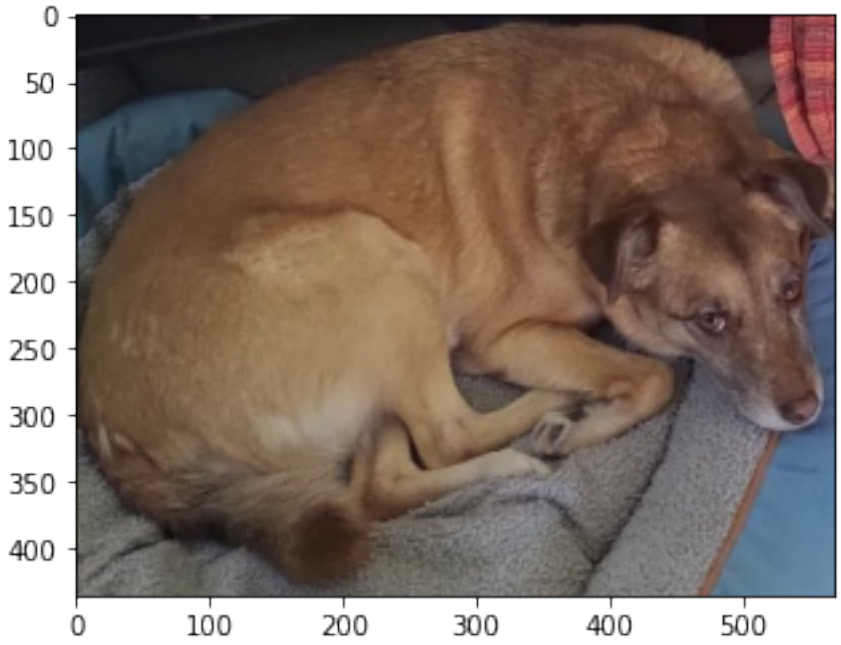



```
Predicted dog breed is Alaskan malamute


Hello human!
```



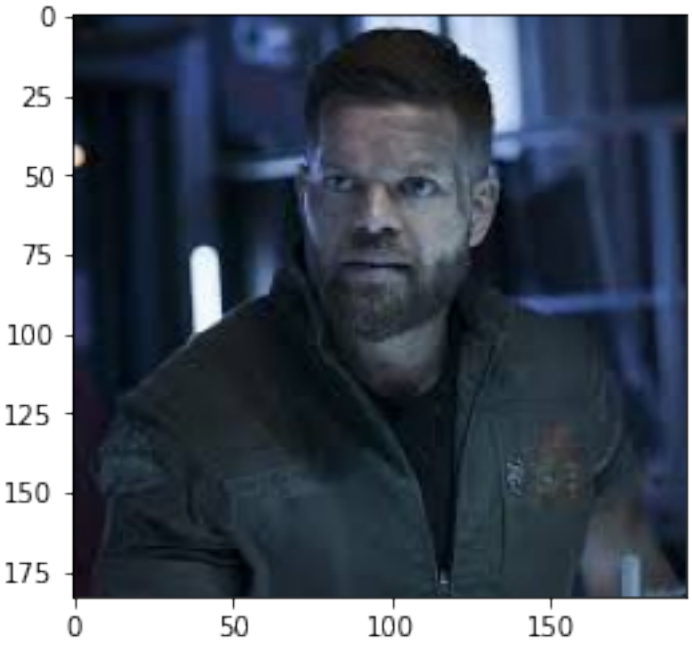

```
Resembling dog breed is Xoloitzcuintli
```

```
Hello who are you?
```



```
Resembling dog breed is Finnish spitz


Hello human!
```


```
Resembling dog breed is Basenji


Hello human!
```




```
Resembling dog breed is Collie
```