dog_app

February 19, 2019

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 [1]: import numpy as np
        import random
        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

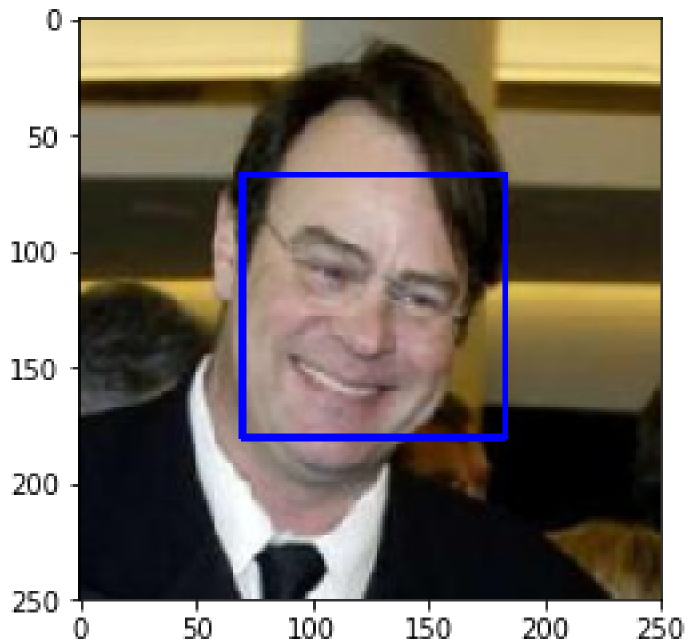
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [2]: import cv2
        import matplotlib.pyplot as plt
        %matplotlib inline

        # extract pre-trained face detector
        face_cascade = cv2.CascadeClassifier('haarcascades/haarcascade_frontalface_alt.xml')

        # load color (BGR) image
        img = cv2.imread(human_files[0])

        # Wrong detection examples (Human face in dog images)
        # img = cv2.imread('/data/dog_images/train/103.Mastiff/Mastiff_06844.jpg')
        # img = cv2.imread('/data/dog_images/train/059.Doberman_pinscher/Doberman_pinscher_04157

        # 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 [3]: # returns "True" if face is detected in image stored at img_path
        def face_detector(img_path):
            img = cv2.imread(img_path)
            gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
            faces = face_cascade.detectMultiScale(gray)
            return len(faces) > 0

        def display_image(img_path):
            # load color (BGR) image
            img = cv2.imread(img_path)

            # 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()
```

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 human_files_short with detected human face: **98.00%** - Percentage of dog_files_short with detected human face: **17.00%**

```
In [4]: from tqdm import tqdm

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

        #-#-# Do NOT modify the code above this line. #-#-#

        ## TODO: Test the performance of the face_detector algorithm
        ## on the images in human_files_short and dog_files_short.
        ## TODO
        debug = False
        human_count = 0
        for human_file in human_files_short:
```

```
    if face_detector(human_file):
        human_count += 1
    elif debug:
        display_image(human_file)
print("Percentage of human_files_short with detected human face: {:.2f}%".format(100*hum

human_count = 0
for dog_file in dog_files_short:
    if face_detector(dog_file):
        human_count += 1
        if debug:
            #print('dog_file: ', dog_file)
            display_image(dog_file)
print("Percentage of dog_files_short with detected human face: {:.2f}%".format(100*human
```

```
Percentage of human_files_short with detected human face: 98.00%
Percentage of dog_files_short with detected human face: 17.00%
```

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 [5]:
```
### (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 [6]:
```
import torch
import torchvision.models as models

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

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

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

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:04<00:00, 116403068.07it/s]
```

Given an image, this pre-trained VGG-16 model returns a prediction (derived from the 1000 possible categories in ImageNet) for the object that is contained in the image.

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 [7]: from PIL import Image, ImageFile
        ImageFile.LOAD_TRUNCATED_IMAGES = True
        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
            ## Return the *index* of the predicted class for that image

            # Load Image
            img_bgr = Image.open(img_path)

            # Define Image Transform to make it compatible with VGG-16
            data_transform = transforms.Compose([ transforms.Resize([224, 224]),
                                                  transforms.ToTensor(),
                                                  transforms.Normalize(mean=[0.485, 0.456, 0.406
                                                     std=[0.229, 0.224, 0.225])])
```

```
    # Transform Image
    # unsqueeze - Feed input as a batch of size 1 (1 image in the batch)
    # img_tensor = data_transform(img_bgr).unsqueeze(0)
    # Discard the transparent, alpha channel (that's the :3) and add the batch dimension
    img_tensor = data_transform(img_bgr)[:3,:,:].unsqueeze(0)

    # Move input to GPU if available
    if use_cuda:
        img_tensor = img_tensor.cuda()

    # Feed Forward the Image via VGG-16
    output = VGG16(img_tensor)


    # Get class with max probability
    _, pred = torch.max(output, 1)

    #if debug:
        #print(f"Prediction for {img_path} is: ", pred)
        #plt.imshow(img_bgr)

    return pred # 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 [8]: ### returns "True" if a dog is detected in the image stored at img_path
        # Return true/false
        def dog_detector(img_path):
            ## TODO: Complete the function.
            predicted_class = VGG16_predict(img_path)
            if predicted_class >= 151 and predicted_class <=268:
                return True
            return False

        test_file = dog_files[random.randint(0,99)]
        print(f"dog_detector({test_file}):", dog_detector(test_file))
```

```
dog_detector(/data/dog_images/train/103.Mastiff/Mastiff_06828.jpg): True
```

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 human_files_short with detected dog: **1.00%** - Percentage of dog_files_short with detected dog: **100.00%**

```
In [9]: ### TODO: Test the performance of the dog_detector function
        ### on the images in human_files_short and dog_files_short.

        dog_count = 0
        for human_file in human_files_short:
            if dog_detector(human_file):
                dog_count += 1
                if debug:
                    display_image(human_file)
        print("Percentage of human_files_short with detected dog: {:.2f}%".format(100*dog_count/

        dog_count = 0
        for dog_file in dog_files_short:
            if dog_detector(dog_file):
                dog_count += 1
            else:
                if debug:
                    display_image(dog_file)
        print("Percentage of dog_files_short with detected dog: {:.2f}%".format(100*dog_count/le
```

```
Percentage of human_files_short with detected dog: 1.00%
Percentage of dog_files_short with detected dog: 100.00%
```

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 [10]: ### (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 [11]: import os
         import numpy as np
         import torchvision.transforms as transforms
         import torch

         from torchvision import datasets
         from PIL import Image
         from torch.utils.data.sampler import SubsetRandomSampler

         ### TODO: Write data loaders for training, validation, and test sets
         ## Specify appropriate transforms, and batch_sizes
         data_dir = '/data/dog_images/'
```

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

# Get the dog class labels
classes = os.listdir(train_dir)

# Transform - Random Resized Crop
data_transform1 = transforms.Compose([transforms.RandomResizedCrop(224),
                                     transforms.RandomHorizontalFlip(),
                                     transforms.ToTensor(),
                                     transforms.Normalize(mean=[0.485, 0.456, 0.406],st
data_transform2 = transforms.Compose([transforms.Resize([224, 224]),
                                     transforms.ToTensor(),
                                     transforms.Normalize(mean=[0.485, 0.456, 0.406],st


# get data sets
train_data = datasets.ImageFolder(train_dir, transform=data_transform1)
valid_data = datasets.ImageFolder(valid_dir, transform=data_transform2)
test_data = datasets.ImageFolder(test_dir, transform=data_transform2)

# print out some data stats
num_train = len(train_data)
num_valid = len(valid_data)
num_test = len(test_data)
num_classes = len(classes)
print('Num training images: ', num_train)
print('Num validation images: ', num_valid)
print('Num test images: ', num_test)
print('Num classes: ', num_classes)

# define dataloader parameters
batch_size = 20
num_workers = 0

# setup the loaders
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, num_worke
valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=batch_size, num_worke
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, num_workers
loaders_scratch = {'train': train_loader,
                  'valid': valid_loader,
                  'test': test_loader}
```

```
Num training images:  6680
Num validation images:  835
Num test images:  836
Num classes:  133
```

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: - Resizing - **Training**: I used RandomResizedCrop(224) based on what is used in several other pretrained torchvision models. I also normalized the images using the mean and standard deviation used by the pretrained models. - **Validation/Testing**: I used transforms.Resize([224, 224]) so as to lose as little information (vs RandomCrop) as possible during evaluation. - Tensor Size - I used a **224x224** image size again based on the what was used by torchvision models like VGG16. The size was not too small and allowed to me to still get RandomCrops of the original image with enough variety to help with dataset augmentation. - Dataset Augmentation - **Yes**. - I used RandomResizedCrop and RandomHorizontalFlip to augment the data (training on more variations of the dataset) and also to avoid overfitting. - In addition, I tried using RandomRotation(10) on the training set datatransform. But, the model accuracy decreased! Most likely because I did not train the model for a long time (>30 epochs).

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [12]: import torch.nn as nn
         import torch.nn.functional as F

         # define the CNN architecture
         class Net(nn.Module):
             ### TODO: choose an architecture, and complete the class
             def __init__(self):
                 super(Net, self).__init__()
                 ## Define layers of a CNN
                 self.conv1 = nn.Conv2d(3, 32, 3, stride=2, padding=1)
                 self.conv2 = nn.Conv2d(32, 64, 3, stride=2, padding=1)
                 self.conv3 = nn.Conv2d(64, 128, 3, stride=1, padding=1)
                 self.pool = nn.MaxPool2d(2, 2)
                 self.fc1 = nn.Linear(7 * 7 * 128, 512)
                 self.fc2 = nn.Linear(512, num_classes)
                 self.dropout = nn.Dropout(p=0.2)

             def forward(self, x):
                 ## Define forward behavior

                 # convolution layers
                 x = self.pool(F.relu((self.conv1(x))))
                 x = self.pool(F.relu((self.conv2(x))))
                 x = self.pool(F.relu((self.conv3(x))))
                 #print('Shape: ',x.shape)
```

```python
        # flatten
        x = x.view(-1, 7 * 7 * 128)

        # pass to fully connected layers
        x = self.dropout(x)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x

#-#-# You so 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()
```

```
Net(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=6272, out_features=512, bias=True)
  (fc2): Linear(in_features=512, out_features=133, bias=True)
  (dropout): Dropout(p=0.2)
)
```

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

Answer: - Input Image Size: - 224x224 - Convolution Layers: - Wanted to reduce the (x, y) dimensions to (7x7). Inline with well established models. - Depth: Tried 64 and later increased it to 128 to get over 10% accuracy. - Layer 1: Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1)) - MaxPool: MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False) - Layer 2: Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1)) - MaxPool: MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False) - Layer 3: Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) - Modified stride here so as to achieve the desired (x, y) dimensions of (7, 7) after all the convolution layers. Reducing (x,y) to (3, 3) did not train well. - MaxPool: MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
- Fully Connected Layers: - Used 2 fully connected layers with ReLu activation and a dropout of 20% - Layer 1: (fc1): Linear(in_features=6272, out_features=512, bias=True) - Layer 2: (fc2): Linear(in_features=512, out_features=133, bias=True) - Loss: CrossEntropyLoss - Optimizer: SGD (with learning rate = 0.05). Tried Adam too which worked equally well. - Output Size (num classes): - 133 - **Test Accuracy (on 1 run)**: 20% (171/836)

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 [13]: import torch.optim as optim

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

         ### TODO: select optimizer
         optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=0.05)
```

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 [14]: 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
             print_every = 100

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

                     # Reset gradients
                     optimizer.zero_grad()

                     # Feed forward and compute loss
                     output = model(data)

                     # compute loss
                     loss = criterion(output, target)
```

```
            # optimzer step
            loss.backward()
            optimizer.step()

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

            if debug and batch_idx % print_every == 0:
                print(f'\tEpoch #{epoch}, Iteration #{batch_idx+1}, Loss: {train_loss}'

        ######################
        # validate the model #
        ######################
        model.eval()
        for batch_idx, (data, target) in enumerate(loaders['valid']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## update the average validation loss

            # Feed forward and compute loss
            output = model(data)
            loss = criterion(output, target)
            valid_loss += ((1 / (batch_idx + 1)) * (loss.data - valid_loss))

        # print training/validation statistics
        print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
            epoch,
            train_loss,
            valid_loss
            ))

        ## TODO: save the model if validation loss has decreased
        if valid_loss < valid_loss_min:
            print('\tValidation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.
                    valid_loss_min,
                    valid_loss))
            valid_loss_min = valid_loss
            torch.save(model.state_dict(), save_path)

        print("\n")

    # return trained model
    return model

# train the model
num_epochs = 30
debug=True
```

```python
model_file = 'models/model_scratch.pt'

# load the previously saved model if available, to get a warm start
if os.path.isfile(model_file):
    model_scratch.load_state_dict(torch.load(model_file))

model_scratch = train(num_epochs, loaders_scratch, model_scratch, optimizer_scratch,
                      criterion_scratch, use_cuda, model_file)


# load the model that got the best validation accuracy
model_scratch.load_state_dict(torch.load(model_file))

print('Done Training!')
```

```
        Epoch #1, Iteration #1, Loss: 3.222787857055664
        Epoch #1, Iteration #101, Loss: 3.415236473083496
        Epoch #1, Iteration #201, Loss: 3.4306204319000244
        Epoch #1, Iteration #301, Loss: 3.431774377822876
Epoch: 1         Training Loss: 3.416890         Validation Loss: 3.859243
        Validation loss decreased (inf --> 3.859243).  Saving model ...


        Epoch #2, Iteration #1, Loss: 2.9762368202209473
        Epoch #2, Iteration #101, Loss: 3.3477654457092285
        Epoch #2, Iteration #201, Loss: 3.3717944622039795
        Epoch #2, Iteration #301, Loss: 3.3874828815460205
Epoch: 2         Training Loss: 3.391319         Validation Loss: 3.578955
        Validation loss decreased (3.859243 --> 3.578955).  Saving model ...


        Epoch #3, Iteration #1, Loss: 2.5569567680358887
        Epoch #3, Iteration #101, Loss: 3.248321533203125
        Epoch #3, Iteration #201, Loss: 3.2913012504577637
        Epoch #3, Iteration #301, Loss: 3.321340322494507
Epoch: 3         Training Loss: 3.346957         Validation Loss: 3.736090


        Epoch #4, Iteration #1, Loss: 3.172574520111084
        Epoch #4, Iteration #101, Loss: 3.3154990673065186
        Epoch #4, Iteration #201, Loss: 3.2750895023345947
        Epoch #4, Iteration #301, Loss: 3.28889536857605
Epoch: 4         Training Loss: 3.301481         Validation Loss: 3.657772


        Epoch #5, Iteration #1, Loss: 3.534672260284424
        Epoch #5, Iteration #101, Loss: 3.2341620922088623
        Epoch #5, Iteration #201, Loss: 3.243776321411133
```

```
        Epoch #5, Iteration #301, Loss: 3.2569363117218018
Epoch: 5        Training Loss: 3.258054        Validation Loss: 3.548971
        Validation loss decreased (3.578955 --> 3.548971).  Saving model ...


        Epoch #6, Iteration #1, Loss: 2.587562084197998
        Epoch #6, Iteration #101, Loss: 3.173037052154541
        Epoch #6, Iteration #201, Loss: 3.1951847076416016
        Epoch #6, Iteration #301, Loss: 3.2215211391448975
Epoch: 6        Training Loss: 3.220563        Validation Loss: 3.570187


        Epoch #7, Iteration #1, Loss: 3.163062572479248
        Epoch #7, Iteration #101, Loss: 3.1206462383270264
        Epoch #7, Iteration #201, Loss: 3.1501362323760986
        Epoch #7, Iteration #301, Loss: 3.1777756214141846
Epoch: 7        Training Loss: 3.177391        Validation Loss: 3.596332


        Epoch #8, Iteration #1, Loss: 2.9891936779022217
        Epoch #8, Iteration #101, Loss: 3.1256346702575684
        Epoch #8, Iteration #201, Loss: 3.15085506439209
        Epoch #8, Iteration #301, Loss: 3.1663148403167725
Epoch: 8        Training Loss: 3.169110        Validation Loss: 3.546840
        Validation loss decreased (3.548971 --> 3.546840).  Saving model ...


        Epoch #9, Iteration #1, Loss: 3.3368678092956543
        Epoch #9, Iteration #101, Loss: 3.061596632003784
        Epoch #9, Iteration #201, Loss: 3.0721092224121094
        Epoch #9, Iteration #301, Loss: 3.0885891914367676
Epoch: 9        Training Loss: 3.096459        Validation Loss: 3.571249


        Epoch #10, Iteration #1, Loss: 3.0263917446136475
        Epoch #10, Iteration #101, Loss: 3.0734403133392334
        Epoch #10, Iteration #201, Loss: 3.0812151432037354
        Epoch #10, Iteration #301, Loss: 3.092982053756714
Epoch: 10        Training Loss: 3.097695        Validation Loss: 3.786886


        Epoch #11, Iteration #1, Loss: 2.676487445831299
        Epoch #11, Iteration #101, Loss: 3.042048454284668
        Epoch #11, Iteration #201, Loss: 3.037799835205078
        Epoch #11, Iteration #301, Loss: 3.04154634475708
Epoch: 11        Training Loss: 3.040271        Validation Loss: 3.558982
```

```
        Epoch #12, Iteration #1, Loss: 2.902878522872925
        Epoch #12, Iteration #101, Loss: 2.993830442428589
        Epoch #12, Iteration #201, Loss: 3.0025572776794434
        Epoch #12, Iteration #301, Loss: 3.0164196491241455
Epoch: 12        Training Loss: 3.025769        Validation Loss: 3.443315
        Validation loss decreased (3.546840 --> 3.443315).  Saving model ...


        Epoch #13, Iteration #1, Loss: 2.888326406478882
        Epoch #13, Iteration #101, Loss: 2.921394109725952
        Epoch #13, Iteration #201, Loss: 2.944779634475708
        Epoch #13, Iteration #301, Loss: 2.953549385070801
Epoch: 13        Training Loss: 2.958822        Validation Loss: 3.531041


        Epoch #14, Iteration #1, Loss: 3.128679037094116
        Epoch #14, Iteration #101, Loss: 2.943382501602173
        Epoch #14, Iteration #201, Loss: 2.9561848640441895
        Epoch #14, Iteration #301, Loss: 2.965667486190796
Epoch: 14        Training Loss: 2.973637        Validation Loss: 3.507233


        Epoch #15, Iteration #1, Loss: 2.4618849754333496
        Epoch #15, Iteration #101, Loss: 2.951514959335327
        Epoch #15, Iteration #201, Loss: 2.9548890590667725
        Epoch #15, Iteration #301, Loss: 2.9454963207244873
Epoch: 15        Training Loss: 2.946701        Validation Loss: 3.532644


        Epoch #16, Iteration #1, Loss: 2.6506834030151367
        Epoch #16, Iteration #101, Loss: 2.831879138946533
        Epoch #16, Iteration #201, Loss: 2.8525216579437256
        Epoch #16, Iteration #301, Loss: 2.8761136531829834
Epoch: 16        Training Loss: 2.890655        Validation Loss: 3.808220


        Epoch #17, Iteration #1, Loss: 3.4228038787841797
        Epoch #17, Iteration #101, Loss: 2.8707993030548096
        Epoch #17, Iteration #201, Loss: 2.8619353771209717
        Epoch #17, Iteration #301, Loss: 2.887681007385254
Epoch: 17        Training Loss: 2.891411        Validation Loss: 3.538228


        Epoch #18, Iteration #1, Loss: 2.1164374351501465
        Epoch #18, Iteration #101, Loss: 2.794858694076538
        Epoch #18, Iteration #201, Loss: 2.8487281799316406
        Epoch #18, Iteration #301, Loss: 2.8851191997528076
Epoch: 18        Training Loss: 2.894778        Validation Loss: 3.632461
```

```
        Epoch #19, Iteration #1, Loss: 3.13734769821167
        Epoch #19, Iteration #101, Loss: 2.733853578567505
        Epoch #19, Iteration #201, Loss: 2.790557622909546
        Epoch #19, Iteration #301, Loss: 2.803269147872925
Epoch: 19        Training Loss: 2.814364        Validation Loss: 3.555188


        Epoch #20, Iteration #1, Loss: 2.2411577701568604
        Epoch #20, Iteration #101, Loss: 2.853034496307373
        Epoch #20, Iteration #201, Loss: 2.8291540145874023
        Epoch #20, Iteration #301, Loss: 2.823007822036743
Epoch: 20        Training Loss: 2.828866        Validation Loss: 3.516930


        Epoch #21, Iteration #1, Loss: 3.198730945587158
        Epoch #21, Iteration #101, Loss: 2.759795904159546
        Epoch #21, Iteration #201, Loss: 2.756319046020508
        Epoch #21, Iteration #301, Loss: 2.7767271995544434
Epoch: 21        Training Loss: 2.794479        Validation Loss: 3.516197


        Epoch #22, Iteration #1, Loss: 2.129220962524414
        Epoch #22, Iteration #101, Loss: 2.702435255050659
        Epoch #22, Iteration #201, Loss: 2.74430775642395
        Epoch #22, Iteration #301, Loss: 2.77054500579834
Epoch: 22        Training Loss: 2.781784        Validation Loss: 3.528630


        Epoch #23, Iteration #1, Loss: 2.7940332889556885
        Epoch #23, Iteration #101, Loss: 2.6588544845581055
        Epoch #23, Iteration #201, Loss: 2.6730949878692627
        Epoch #23, Iteration #301, Loss: 2.701401710510254
Epoch: 23        Training Loss: 2.716979        Validation Loss: 3.605093


        Epoch #24, Iteration #1, Loss: 2.7360739707946777
        Epoch #24, Iteration #101, Loss: 2.608131170272827
        Epoch #24, Iteration #201, Loss: 2.6754415035247803
        Epoch #24, Iteration #301, Loss: 2.7130370140075684
Epoch: 24        Training Loss: 2.712008        Validation Loss: 3.712774


        Epoch #25, Iteration #1, Loss: 3.4913086891174316
        Epoch #25, Iteration #101, Loss: 2.632678985595703
        Epoch #25, Iteration #201, Loss: 2.6637189388275146
        Epoch #25, Iteration #301, Loss: 2.691129446029663
```

```
Epoch: 25         Training Loss: 2.695329         Validation Loss: 3.613812


        Epoch #26, Iteration #1, Loss: 3.2120487689971924
        Epoch #26, Iteration #101, Loss: 2.65832781791687
        Epoch #26, Iteration #201, Loss: 2.655308961868286
        Epoch #26, Iteration #301, Loss: 2.677370309829712
Epoch: 26         Training Loss: 2.684457         Validation Loss: 3.471386


        Epoch #27, Iteration #1, Loss: 1.8021968603134155
        Epoch #27, Iteration #101, Loss: 2.688344717025757
        Epoch #27, Iteration #201, Loss: 2.675527811050415
        Epoch #27, Iteration #301, Loss: 2.6795949935913086
Epoch: 27         Training Loss: 2.672729         Validation Loss: 3.577964


        Epoch #28, Iteration #1, Loss: 2.474621295928955
        Epoch #28, Iteration #101, Loss: 2.513354778289795
        Epoch #28, Iteration #201, Loss: 2.560866117477417
        Epoch #28, Iteration #301, Loss: 2.5960423946380615
Epoch: 28         Training Loss: 2.616052         Validation Loss: 3.627453


        Epoch #29, Iteration #1, Loss: 1.9054960012435913
        Epoch #29, Iteration #101, Loss: 2.5672268867492676
        Epoch #29, Iteration #201, Loss: 2.590578556060791
        Epoch #29, Iteration #301, Loss: 2.5955138206481934
Epoch: 29         Training Loss: 2.606864         Validation Loss: 3.620353


        Epoch #30, Iteration #1, Loss: 2.1952805519104004
        Epoch #30, Iteration #101, Loss: 2.505970001220703
        Epoch #30, Iteration #201, Loss: 2.549293279647827
        Epoch #30, Iteration #301, Loss: 2.5775768756866455
Epoch: 30         Training Loss: 2.594254         Validation Loss: 3.672189


Done Training!
```

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

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

    model.eval()
    for batch_idx, (data, target) in enumerate(loaders['test']):
        # move to GPU
        if use_cuda:
            data, target = data.cuda(), target.cuda()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the loss
        loss = criterion(output, target)
        # update average test loss
        test_loss += ((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))
```

In [16]:
```
# call test function
test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.390768


Test Accuracy: 20% (171/836)
```

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 [17]:
```
import os
import numpy as np
import torchvision.transforms as transforms
import torch

from torchvision import datasets
from PIL import Image, ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True
from torch.utils.data.sampler import SubsetRandomSampler

### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes
data_dir = '/data/dog_images/'
train_dir = os.path.join(data_dir, 'train/')
valid_dir = os.path.join(data_dir, 'valid/')
test_dir = os.path.join(data_dir, 'test/')

# Get the dog class labels
classes = os.listdir(train_dir)

# Transform - Random Resized Crop
data_transform_train = transforms.Compose([transforms.RandomResizedCrop(299),
                                    transforms.RandomHorizontalFlip(),
                                    transforms.ToTensor(),
                                    transforms.Normalize(mean=[0.485, 0.456, 0.406],st
data_transform_test = transforms.Compose([transforms.Resize([299, 299]),
                                    transforms.ToTensor(),
                                    transforms.Normalize(mean=[0.485, 0.456, 0.406],st


# get data sets
train_data = datasets.ImageFolder(train_dir, transform=data_transform_train)
valid_data = datasets.ImageFolder(valid_dir, transform=data_transform_test)
test_data = datasets.ImageFolder(test_dir, transform=data_transform_test)

# print out some data stats
num_train = len(train_data)
num_valid = len(valid_data)
num_test = len(test_data)
num_classes = len(classes)
print('Num training images: ', num_train)
print('Num validation images: ', num_valid)
print('Num test images: ', num_test)
print('Num classes: ', num_classes)

# define dataloader parameters
batch_size = 20
num_workers = 0
```

```
# setup the loaders
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, num_worke
valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=batch_size, num_worke
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, num_workers
loaders_transfer = {'train': train_loader,
                    'valid': valid_loader,
                    'test': test_loader}
```

```
Num training images:  6680
Num validation images:  835
Num test images:  836
Num classes:  133
```

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 [18]:
```
import torch
import torchvision.models as models
import torch.nn as nn

import torch.optim as optim

## TODO: Specify model architecture
model_transfer = models.inception_v3(pretrained=True)

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

# Freeze training for all "features" layers
for param in model_transfer.parameters():
    param.requires_grad = False

# Replace fully connected layer
model_transfer.fc = nn.Linear(2048, num_classes)

# Replace Auxillary Classifier layer
model_transfer.AuxLogits.fc = nn.Linear(768, num_classes)

# Print modified model
print(model_transfer)

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

```
Downloading: "https://download.pytorch.org/models/inception_v3_google-1a9a5a14.pth" to /root/.to
100%|| 108857766/108857766 [00:01<00:00, 55897450.75it/s]
```

```
Inception3(
  (Conv2d_1a_3x3): BasicConv2d(
    (conv): Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2), bias=False)
    (bn): BatchNorm2d(32, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
  )
  (Conv2d_2a_3x3): BasicConv2d(
    (conv): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), bias=False)
    (bn): BatchNorm2d(32, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
  )
  (Conv2d_2b_3x3): BasicConv2d(
    (conv): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn): BatchNorm2d(64, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
  )
  (Conv2d_3b_1x1): BasicConv2d(
    (conv): Conv2d(64, 80, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn): BatchNorm2d(80, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
  )
  (Conv2d_4a_3x3): BasicConv2d(
    (conv): Conv2d(80, 192, kernel_size=(3, 3), stride=(1, 1), bias=False)
    (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
  )
  (Mixed_5b): InceptionA(
    (branch1x1): BasicConv2d(
      (conv): Conv2d(192, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(64, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch5x5_1): BasicConv2d(
      (conv): Conv2d(192, 48, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(48, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch5x5_2): BasicConv2d(
      (conv): Conv2d(48, 64, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2), bias=False)
      (bn): BatchNorm2d(64, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch3x3dbl_1): BasicConv2d(
      (conv): Conv2d(192, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(64, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch3x3dbl_2): BasicConv2d(
      (conv): Conv2d(64, 96, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn): BatchNorm2d(96, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch3x3dbl_3): BasicConv2d(
      (conv): Conv2d(96, 96, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
```

```
      (bn): BatchNorm2d(96, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch_pool): BasicConv2d(
      (conv): Conv2d(192, 32, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(32, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (Mixed_5c): InceptionA(
    (branch1x1): BasicConv2d(
      (conv): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(64, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch5x5_1): BasicConv2d(
      (conv): Conv2d(256, 48, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(48, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch5x5_2): BasicConv2d(
      (conv): Conv2d(48, 64, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2), bias=False)
      (bn): BatchNorm2d(64, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch3x3dbl_1): BasicConv2d(
      (conv): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(64, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch3x3dbl_2): BasicConv2d(
      (conv): Conv2d(64, 96, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn): BatchNorm2d(96, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch3x3dbl_3): BasicConv2d(
      (conv): Conv2d(96, 96, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn): BatchNorm2d(96, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch_pool): BasicConv2d(
      (conv): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(64, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (Mixed_5d): InceptionA(
    (branch1x1): BasicConv2d(
      (conv): Conv2d(288, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(64, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch5x5_1): BasicConv2d(
      (conv): Conv2d(288, 48, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(48, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch5x5_2): BasicConv2d(
      (conv): Conv2d(48, 64, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2), bias=False)
```

```
      (bn): BatchNorm2d(64, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch3x3dbl_1): BasicConv2d(
      (conv): Conv2d(288, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(64, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch3x3dbl_2): BasicConv2d(
      (conv): Conv2d(64, 96, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn): BatchNorm2d(96, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch3x3dbl_3): BasicConv2d(
      (conv): Conv2d(96, 96, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn): BatchNorm2d(96, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch_pool): BasicConv2d(
      (conv): Conv2d(288, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(64, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (Mixed_6a): InceptionB(
    (branch3x3): BasicConv2d(
      (conv): Conv2d(288, 384, kernel_size=(3, 3), stride=(2, 2), bias=False)
      (bn): BatchNorm2d(384, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch3x3dbl_1): BasicConv2d(
      (conv): Conv2d(288, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(64, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch3x3dbl_2): BasicConv2d(
      (conv): Conv2d(64, 96, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn): BatchNorm2d(96, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch3x3dbl_3): BasicConv2d(
      (conv): Conv2d(96, 96, kernel_size=(3, 3), stride=(2, 2), bias=False)
      (bn): BatchNorm2d(96, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (Mixed_6b): InceptionC(
    (branch1x1): BasicConv2d(
      (conv): Conv2d(768, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch7x7_1): BasicConv2d(
      (conv): Conv2d(768, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(128, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch7x7_2): BasicConv2d(
      (conv): Conv2d(128, 128, kernel_size=(1, 7), stride=(1, 1), padding=(0, 3), bias=False)
```

```
      (bn): BatchNorm2d(128, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch7x7_3): BasicConv2d(
      (conv): Conv2d(128, 192, kernel_size=(7, 1), stride=(1, 1), padding=(3, 0), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch7x7dbl_1): BasicConv2d(
      (conv): Conv2d(768, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(128, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch7x7dbl_2): BasicConv2d(
      (conv): Conv2d(128, 128, kernel_size=(7, 1), stride=(1, 1), padding=(3, 0), bias=False)
      (bn): BatchNorm2d(128, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch7x7dbl_3): BasicConv2d(
      (conv): Conv2d(128, 128, kernel_size=(1, 7), stride=(1, 1), padding=(0, 3), bias=False)
      (bn): BatchNorm2d(128, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch7x7dbl_4): BasicConv2d(
      (conv): Conv2d(128, 128, kernel_size=(7, 1), stride=(1, 1), padding=(3, 0), bias=False)
      (bn): BatchNorm2d(128, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch7x7dbl_5): BasicConv2d(
      (conv): Conv2d(128, 192, kernel_size=(1, 7), stride=(1, 1), padding=(0, 3), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch_pool): BasicConv2d(
      (conv): Conv2d(768, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (Mixed_6c): InceptionC(
    (branch1x1): BasicConv2d(
      (conv): Conv2d(768, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch7x7_1): BasicConv2d(
      (conv): Conv2d(768, 160, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(160, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch7x7_2): BasicConv2d(
      (conv): Conv2d(160, 160, kernel_size=(1, 7), stride=(1, 1), padding=(0, 3), bias=False)
      (bn): BatchNorm2d(160, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch7x7_3): BasicConv2d(
      (conv): Conv2d(160, 192, kernel_size=(7, 1), stride=(1, 1), padding=(3, 0), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
```

```
      (branch7x7dbl_1): BasicConv2d(
        (conv): Conv2d(768, 160, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn): BatchNorm2d(160, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      )
      (branch7x7dbl_2): BasicConv2d(
        (conv): Conv2d(160, 160, kernel_size=(7, 1), stride=(1, 1), padding=(3, 0), bias=False)
        (bn): BatchNorm2d(160, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      )
      (branch7x7dbl_3): BasicConv2d(
        (conv): Conv2d(160, 160, kernel_size=(1, 7), stride=(1, 1), padding=(0, 3), bias=False)
        (bn): BatchNorm2d(160, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      )
      (branch7x7dbl_4): BasicConv2d(
        (conv): Conv2d(160, 160, kernel_size=(7, 1), stride=(1, 1), padding=(3, 0), bias=False)
        (bn): BatchNorm2d(160, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      )
      (branch7x7dbl_5): BasicConv2d(
        (conv): Conv2d(160, 192, kernel_size=(1, 7), stride=(1, 1), padding=(0, 3), bias=False)
        (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      )
      (branch_pool): BasicConv2d(
        (conv): Conv2d(768, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (Mixed_6d): InceptionC(
      (branch1x1): BasicConv2d(
        (conv): Conv2d(768, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      )
      (branch7x7_1): BasicConv2d(
        (conv): Conv2d(768, 160, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn): BatchNorm2d(160, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      )
      (branch7x7_2): BasicConv2d(
        (conv): Conv2d(160, 160, kernel_size=(1, 7), stride=(1, 1), padding=(0, 3), bias=False)
        (bn): BatchNorm2d(160, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      )
      (branch7x7_3): BasicConv2d(
        (conv): Conv2d(160, 192, kernel_size=(7, 1), stride=(1, 1), padding=(3, 0), bias=False)
        (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      )
      (branch7x7dbl_1): BasicConv2d(
        (conv): Conv2d(768, 160, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn): BatchNorm2d(160, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      )
      (branch7x7dbl_2): BasicConv2d(
        (conv): Conv2d(160, 160, kernel_size=(7, 1), stride=(1, 1), padding=(3, 0), bias=False)
```

```
        (bn): BatchNorm2d(160, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      )
      (branch7x7dbl_3): BasicConv2d(
        (conv): Conv2d(160, 160, kernel_size=(1, 7), stride=(1, 1), padding=(0, 3), bias=False)
        (bn): BatchNorm2d(160, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      )
      (branch7x7dbl_4): BasicConv2d(
        (conv): Conv2d(160, 160, kernel_size=(7, 1), stride=(1, 1), padding=(3, 0), bias=False)
        (bn): BatchNorm2d(160, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      )
      (branch7x7dbl_5): BasicConv2d(
        (conv): Conv2d(160, 192, kernel_size=(1, 7), stride=(1, 1), padding=(0, 3), bias=False)
        (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      )
      (branch_pool): BasicConv2d(
        (conv): Conv2d(768, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (Mixed_6e): InceptionC(
      (branch1x1): BasicConv2d(
        (conv): Conv2d(768, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      )
      (branch7x7_1): BasicConv2d(
        (conv): Conv2d(768, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      )
      (branch7x7_2): BasicConv2d(
        (conv): Conv2d(192, 192, kernel_size=(1, 7), stride=(1, 1), padding=(0, 3), bias=False)
        (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      )
      (branch7x7_3): BasicConv2d(
        (conv): Conv2d(192, 192, kernel_size=(7, 1), stride=(1, 1), padding=(3, 0), bias=False)
        (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      )
      (branch7x7dbl_1): BasicConv2d(
        (conv): Conv2d(768, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      )
      (branch7x7dbl_2): BasicConv2d(
        (conv): Conv2d(192, 192, kernel_size=(7, 1), stride=(1, 1), padding=(3, 0), bias=False)
        (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      )
      (branch7x7dbl_3): BasicConv2d(
        (conv): Conv2d(192, 192, kernel_size=(1, 7), stride=(1, 1), padding=(0, 3), bias=False)
        (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
      )
```

```
    (branch7x7dbl_4): BasicConv2d(
      (conv): Conv2d(192, 192, kernel_size=(7, 1), stride=(1, 1), padding=(3, 0), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch7x7dbl_5): BasicConv2d(
      (conv): Conv2d(192, 192, kernel_size=(1, 7), stride=(1, 1), padding=(0, 3), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch_pool): BasicConv2d(
      (conv): Conv2d(768, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (AuxLogits): InceptionAux(
    (conv0): BasicConv2d(
      (conv): Conv2d(768, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(128, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (conv1): BasicConv2d(
      (conv): Conv2d(128, 768, kernel_size=(5, 5), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(768, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (fc): Linear(in_features=768, out_features=133, bias=True)
  )
  (Mixed_7a): InceptionD(
    (branch3x3_1): BasicConv2d(
      (conv): Conv2d(768, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch3x3_2): BasicConv2d(
      (conv): Conv2d(192, 320, kernel_size=(3, 3), stride=(2, 2), bias=False)
      (bn): BatchNorm2d(320, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch7x7x3_1): BasicConv2d(
      (conv): Conv2d(768, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch7x7x3_2): BasicConv2d(
      (conv): Conv2d(192, 192, kernel_size=(1, 7), stride=(1, 1), padding=(0, 3), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch7x7x3_3): BasicConv2d(
      (conv): Conv2d(192, 192, kernel_size=(7, 1), stride=(1, 1), padding=(3, 0), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch7x7x3_4): BasicConv2d(
      (conv): Conv2d(192, 192, kernel_size=(3, 3), stride=(2, 2), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
```

```
    )
  )
  (Mixed_7b): InceptionE(
    (branch1x1): BasicConv2d(
      (conv): Conv2d(1280, 320, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(320, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch3x3_1): BasicConv2d(
      (conv): Conv2d(1280, 384, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(384, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch3x3_2a): BasicConv2d(
      (conv): Conv2d(384, 384, kernel_size=(1, 3), stride=(1, 1), padding=(0, 1), bias=False)
      (bn): BatchNorm2d(384, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch3x3_2b): BasicConv2d(
      (conv): Conv2d(384, 384, kernel_size=(3, 1), stride=(1, 1), padding=(1, 0), bias=False)
      (bn): BatchNorm2d(384, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch3x3dbl_1): BasicConv2d(
      (conv): Conv2d(1280, 448, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(448, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch3x3dbl_2): BasicConv2d(
      (conv): Conv2d(448, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn): BatchNorm2d(384, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch3x3dbl_3a): BasicConv2d(
      (conv): Conv2d(384, 384, kernel_size=(1, 3), stride=(1, 1), padding=(0, 1), bias=False)
      (bn): BatchNorm2d(384, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch3x3dbl_3b): BasicConv2d(
      (conv): Conv2d(384, 384, kernel_size=(3, 1), stride=(1, 1), padding=(1, 0), bias=False)
      (bn): BatchNorm2d(384, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch_pool): BasicConv2d(
      (conv): Conv2d(1280, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (Mixed_7c): InceptionE(
    (branch1x1): BasicConv2d(
      (conv): Conv2d(2048, 320, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(320, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch3x3_1): BasicConv2d(
      (conv): Conv2d(2048, 384, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(384, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
```

```
    )
    (branch3x3_2a): BasicConv2d(
      (conv): Conv2d(384, 384, kernel_size=(1, 3), stride=(1, 1), padding=(0, 1), bias=False)
      (bn): BatchNorm2d(384, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch3x3_2b): BasicConv2d(
      (conv): Conv2d(384, 384, kernel_size=(3, 1), stride=(1, 1), padding=(1, 0), bias=False)
      (bn): BatchNorm2d(384, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch3x3dbl_1): BasicConv2d(
      (conv): Conv2d(2048, 448, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(448, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch3x3dbl_2): BasicConv2d(
      (conv): Conv2d(448, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn): BatchNorm2d(384, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch3x3dbl_3a): BasicConv2d(
      (conv): Conv2d(384, 384, kernel_size=(1, 3), stride=(1, 1), padding=(0, 1), bias=False)
      (bn): BatchNorm2d(384, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch3x3dbl_3b): BasicConv2d(
      (conv): Conv2d(384, 384, kernel_size=(3, 1), stride=(1, 1), padding=(1, 0), bias=False)
      (bn): BatchNorm2d(384, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch_pool): BasicConv2d(
      (conv): Conv2d(2048, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (fc): Linear(in_features=2048, 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: I wanted to try some new architectures and chose Inception_v3 as it is known to perform well, heard it mentioned several times but actually not seen it in action. I soon relaized that this model had an **auxiliary fully connected layer** that was used only in training. I replaced both the output and auxiliary fully connected layers with my own that matched with the number of output classes for our classification problem. I froze all model parameter other than the ones relating to the 2 fully connected layers that I added.

- Tensor Size
- I had to use 299x299 image size based on the input requirements for Inception_v3 model.
- Training: I used RandomResizedCrop(229). I also normalized the images using the mean and standard deviation used by the pretrained models.
- Validation/Testing: I used transforms.Resize([299, 299]) so as to lose as little information (vs RandomCrop) as possible during evaluation.

- Dataset Augmentation
  - Yes.
  - I used RandomResizedCrop and RandomHorizontalFlip to augment the data (training on more variations of the dataset) and also to avoid overfitting.
- **Test Accuracy (on 1 run)**: 78% (659/836)

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 [19]: criterion_transfer = nn.CrossEntropyLoss()
         params_to_update = []
         for param in model_transfer.parameters():
             if param.requires_grad:
                 params_to_update.append(param)
         optimizer_transfer = optim.Adam(params_to_update, lr=0.01)
```

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 [20]: import os.path

         def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path, is_incep
             """returns trained model"""
             # initialize tracker for minimum validation loss
             valid_loss_min = np.Inf
             print_every = 100

             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
                     ## record the average training loss, using something like
                     ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_lo
```

```
        # Reset gradients
        optimizer.zero_grad()

        # Feed forward and compute loss
        if is_inception:
            # handle auxillary output from inception
            # https://pytorch.org/tutorials/beginner/finetuning_torchvision_models_
            output, aux_output = model(data)
            loss1 = criterion(output, target)
            loss2 = criterion(aux_output, target)
            loss = loss1 + 0.4*loss2
        else:
            output = model(data)
            loss = criterion(output, target)

        # optimizer step
        loss.backward()
        optimizer.step()

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

        if debug and batch_idx % print_every == 0:
            print(f'\tEpoch #{epoch}, Iteration #{batch_idx+1}, Loss: {train_loss}'

    ######################
    # validate the model #
    ######################
    model.eval()
    for batch_idx, (data, target) in enumerate(loaders['valid']):
        # move to GPU
        if use_cuda:
            data, target = data.cuda(), target.cuda()
        ## update the average validation loss

        # Feed forward and compute loss
        # Even in inception, there is no auxillary output during model evaluation
        # https://github.com/pytorch/vision/blob/master/torchvision/models/inceptio
        output = model(data)
        loss = criterion(output, target)

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

    # print training/validation statistics
    print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
        epoch,
        train_loss,
        valid_loss
```

```
                ))

            ## TODO: save the model if validation loss has decreased
            if valid_loss < valid_loss_min:
                print('\tValidation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.
                        valid_loss_min,
                        valid_loss))
                valid_loss_min = valid_loss
                torch.save(model.state_dict(), save_path)

            print("\n")

        # return trained model
        return model
```

In [21]:
```
# train the model
debug=True
num_epochs = 30
model_file = 'models/model_transfer.pt'

# load the previously saved model if available, to get a warm start
if os.path.isfile(model_file):
    model_transfer.load_state_dict(torch.load(model_file))

model_transfer = train(num_epochs, loaders_transfer, model_transfer, optimizer_transfer
                       criterion_transfer, use_cuda, model_file, True)

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

print('Done Training!')
```

```
        Epoch #1, Iteration #1, Loss: 2.5857138633728027
        Epoch #1, Iteration #101, Loss: 6.0715532302856445
        Epoch #1, Iteration #201, Loss: 6.115310192108154
        Epoch #1, Iteration #301, Loss: 6.2081217765808105
Epoch: 1         Training Loss: 6.268450         Validation Loss: 2.021797
        Validation loss decreased (inf --> 2.021797).  Saving model ...


        Epoch #2, Iteration #1, Loss: 7.233248710632324
        Epoch #2, Iteration #101, Loss: 5.946088790893555
        Epoch #2, Iteration #201, Loss: 6.140623569488525
        Epoch #2, Iteration #301, Loss: 6.265612602233887
Epoch: 2         Training Loss: 6.317700         Validation Loss: 2.139870


        Epoch #3, Iteration #1, Loss: 6.745226860046387
```

```
        Epoch #3, Iteration #101, Loss: 6.348630428314209
        Epoch #3, Iteration #201, Loss: 6.434607982635498
        Epoch #3, Iteration #301, Loss: 6.634353160858154
Epoch: 3        Training Loss: 6.648694        Validation Loss: 2.014069
        Validation loss decreased (2.021797 --> 2.014069).  Saving model ...


        Epoch #4, Iteration #1, Loss: 4.540031433105469
        Epoch #4, Iteration #101, Loss: 6.114386081695557
        Epoch #4, Iteration #201, Loss: 6.2734503746032715
        Epoch #4, Iteration #301, Loss: 6.387295722961426
Epoch: 4        Training Loss: 6.471759        Validation Loss: 1.817088
        Validation loss decreased (2.014069 --> 1.817088).  Saving model ...


        Epoch #5, Iteration #1, Loss: 9.699653625488281
        Epoch #5, Iteration #101, Loss: 6.729641437530518
        Epoch #5, Iteration #201, Loss: 6.790037155151367
        Epoch #5, Iteration #301, Loss: 6.856613636016846
Epoch: 5        Training Loss: 6.972218        Validation Loss: 1.965765


        Epoch #6, Iteration #1, Loss: 9.631362915039062
        Epoch #6, Iteration #101, Loss: 6.873438835144043
        Epoch #6, Iteration #201, Loss: 7.3469438552856445
        Epoch #6, Iteration #301, Loss: 7.179884433746338
Epoch: 6        Training Loss: 7.249672        Validation Loss: 1.930143


        Epoch #7, Iteration #1, Loss: 4.078765392303467
        Epoch #7, Iteration #101, Loss: 7.176174163818359
        Epoch #7, Iteration #201, Loss: 7.156558036804199
        Epoch #7, Iteration #301, Loss: 7.09710168838501
Epoch: 7        Training Loss: 7.160083        Validation Loss: 2.118811


        Epoch #8, Iteration #1, Loss: 6.949888229370117
        Epoch #8, Iteration #101, Loss: 6.753422737121582
        Epoch #8, Iteration #201, Loss: 6.999551773071289
        Epoch #8, Iteration #301, Loss: 7.178799629211426
Epoch: 8        Training Loss: 7.180485        Validation Loss: 2.223028


        Epoch #9, Iteration #1, Loss: 13.026325225830078
        Epoch #9, Iteration #101, Loss: 6.869229316711426
        Epoch #9, Iteration #201, Loss: 7.133728981018066
        Epoch #9, Iteration #301, Loss: 7.373859405517578
Epoch: 9        Training Loss: 7.469930        Validation Loss: 2.613167
```

```
        Epoch #10, Iteration #1, Loss: 6.99279260635376
        Epoch #10, Iteration #101, Loss: 7.021251201629639
        Epoch #10, Iteration #201, Loss: 7.2922821044921875
        Epoch #10, Iteration #301, Loss: 7.384606838226318
Epoch: 10        Training Loss: 7.462847        Validation Loss: 2.484450


        Epoch #11, Iteration #1, Loss: 6.730443000793457
        Epoch #11, Iteration #101, Loss: 7.525222301483154
        Epoch #11, Iteration #201, Loss: 7.333653926849365
        Epoch #11, Iteration #301, Loss: 7.407731056213379
Epoch: 11        Training Loss: 7.322731        Validation Loss: 2.490700


        Epoch #12, Iteration #1, Loss: 9.245223045349121
        Epoch #12, Iteration #101, Loss: 7.923200607299805
        Epoch #12, Iteration #201, Loss: 7.800239086151123
        Epoch #12, Iteration #301, Loss: 7.8751091957092285
Epoch: 12        Training Loss: 7.812053        Validation Loss: 2.266948


        Epoch #13, Iteration #1, Loss: 3.8581244945526123
        Epoch #13, Iteration #101, Loss: 7.609640598297119
        Epoch #13, Iteration #201, Loss: 7.722017288208008
        Epoch #13, Iteration #301, Loss: 7.710282325744629
Epoch: 13        Training Loss: 7.670198        Validation Loss: 2.213556


        Epoch #14, Iteration #1, Loss: 8.212874412536621
        Epoch #14, Iteration #101, Loss: 7.797274589538574
        Epoch #14, Iteration #201, Loss: 7.799099445343018
        Epoch #14, Iteration #301, Loss: 8.031720161437988
Epoch: 14        Training Loss: 8.115748        Validation Loss: 2.375376


        Epoch #15, Iteration #1, Loss: 3.9355766773223877
        Epoch #15, Iteration #101, Loss: 7.430359363555908
        Epoch #15, Iteration #201, Loss: 7.507734298706055
        Epoch #15, Iteration #301, Loss: 7.606751441955566
Epoch: 15        Training Loss: 7.670272        Validation Loss: 2.337484


        Epoch #16, Iteration #1, Loss: 10.928091049194336
        Epoch #16, Iteration #101, Loss: 7.7007365226745605
        Epoch #16, Iteration #201, Loss: 7.755185127258301
        Epoch #16, Iteration #301, Loss: 7.874589443206787
```

```
Epoch: 16        Training Loss: 7.879434        Validation Loss: 2.383183


        Epoch #17, Iteration #1, Loss: 11.433257102966309
        Epoch #17, Iteration #101, Loss: 8.118145942687988
        Epoch #17, Iteration #201, Loss: 7.930103302001953
        Epoch #17, Iteration #301, Loss: 8.17491340637207
Epoch: 17        Training Loss: 8.165871        Validation Loss: 2.447206


        Epoch #18, Iteration #1, Loss: 9.608397483825684
        Epoch #18, Iteration #101, Loss: 7.685166835784912
        Epoch #18, Iteration #201, Loss: 7.895205974578857
        Epoch #18, Iteration #301, Loss: 8.220281600952148
Epoch: 18        Training Loss: 8.183845        Validation Loss: 2.500071


        Epoch #19, Iteration #1, Loss: 5.463583469390869
        Epoch #19, Iteration #101, Loss: 7.917751312255859
        Epoch #19, Iteration #201, Loss: 8.120160102844238
        Epoch #19, Iteration #301, Loss: 8.120307922363281
Epoch: 19        Training Loss: 8.105686        Validation Loss: 2.584613


        Epoch #20, Iteration #1, Loss: 8.257987022399902
        Epoch #20, Iteration #101, Loss: 7.950231075286865
        Epoch #20, Iteration #201, Loss: 8.159234046936035
        Epoch #20, Iteration #301, Loss: 8.262646675109863
Epoch: 20        Training Loss: 8.242275        Validation Loss: 2.808084


        Epoch #21, Iteration #1, Loss: 8.799139022827148
        Epoch #21, Iteration #101, Loss: 7.865368366241455
        Epoch #21, Iteration #201, Loss: 7.935146331787109
        Epoch #21, Iteration #301, Loss: 8.023098945617676
Epoch: 21        Training Loss: 8.080071        Validation Loss: 2.623898


        Epoch #22, Iteration #1, Loss: 7.115983009338379
        Epoch #22, Iteration #101, Loss: 7.763071060180664
        Epoch #22, Iteration #201, Loss: 7.8278093338012695
        Epoch #22, Iteration #301, Loss: 8.110808372497559
Epoch: 22        Training Loss: 8.100648        Validation Loss: 2.758134


        Epoch #23, Iteration #1, Loss: 8.458528518676758
        Epoch #23, Iteration #101, Loss: 8.57793140411377
        Epoch #23, Iteration #201, Loss: 8.696394920349121
```

```
        Epoch #23, Iteration #301, Loss: 8.624100685119629
Epoch: 23         Training Loss: 8.616224         Validation Loss: 2.863217


        Epoch #24, Iteration #1, Loss: 2.8071908950805664
        Epoch #24, Iteration #101, Loss: 8.155203819274902
        Epoch #24, Iteration #201, Loss: 8.369111061096191
        Epoch #24, Iteration #301, Loss: 8.24611759185791
Epoch: 24         Training Loss: 8.352648         Validation Loss: 3.169943


        Epoch #25, Iteration #1, Loss: 6.092153549194336
        Epoch #25, Iteration #101, Loss: 8.009735107421875
        Epoch #25, Iteration #201, Loss: 8.258747100830078
        Epoch #25, Iteration #301, Loss: 8.280364036560059
Epoch: 25         Training Loss: 8.277253         Validation Loss: 3.206868


        Epoch #26, Iteration #1, Loss: 4.326980113983154
        Epoch #26, Iteration #101, Loss: 8.205707550048828
        Epoch #26, Iteration #201, Loss: 7.744072437286377
        Epoch #26, Iteration #301, Loss: 7.997096061706543
Epoch: 26         Training Loss: 8.133477         Validation Loss: 2.451951


        Epoch #27, Iteration #1, Loss: 6.755497455596924
        Epoch #27, Iteration #101, Loss: 8.681264877319336
        Epoch #27, Iteration #201, Loss: 8.749443054199219
        Epoch #27, Iteration #301, Loss: 8.931979179382324
Epoch: 27         Training Loss: 8.934000         Validation Loss: 2.904748


        Epoch #28, Iteration #1, Loss: 8.173095703125
        Epoch #28, Iteration #101, Loss: 8.541572570800781
        Epoch #28, Iteration #201, Loss: 8.58267879486084
        Epoch #28, Iteration #301, Loss: 8.554693222045898
Epoch: 28         Training Loss: 8.633287         Validation Loss: 2.707103


        Epoch #29, Iteration #1, Loss: 7.8459625244140625
        Epoch #29, Iteration #101, Loss: 8.552529335021973
        Epoch #29, Iteration #201, Loss: 8.319947242736816
        Epoch #29, Iteration #301, Loss: 8.400060653686523
Epoch: 29         Training Loss: 8.385900         Validation Loss: 2.662087


        Epoch #30, Iteration #1, Loss: 8.343340873718262
        Epoch #30, Iteration #101, Loss: 8.973581314086914
```

```
        Epoch #30, Iteration #201, Loss: 9.019980430603027
        Epoch #30, Iteration #301, Loss: 8.91418170928955
Epoch: 30         Training Loss: 8.946994         Validation Loss: 2.459717


Done Training!
```

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

```
Test Loss: 2.109863


Test Accuracy: 78% (659/836)
```

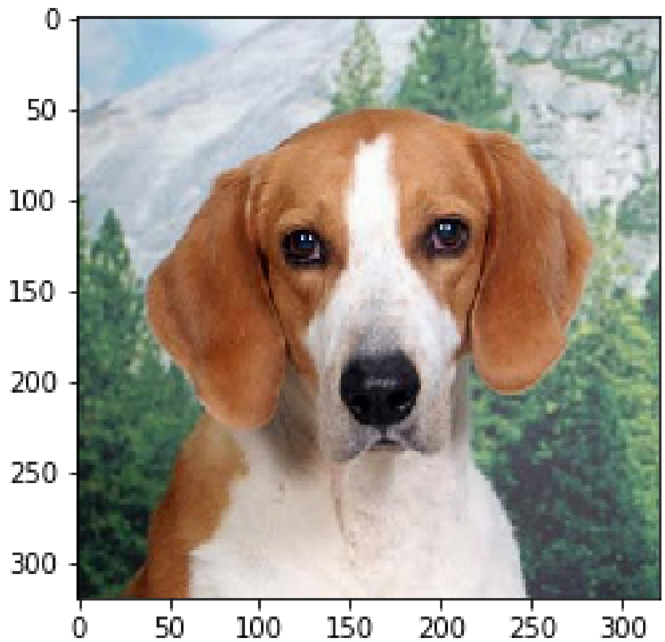
1.1.17 (IMPLEMENTATION) Predict Dog Breed with the Model

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.

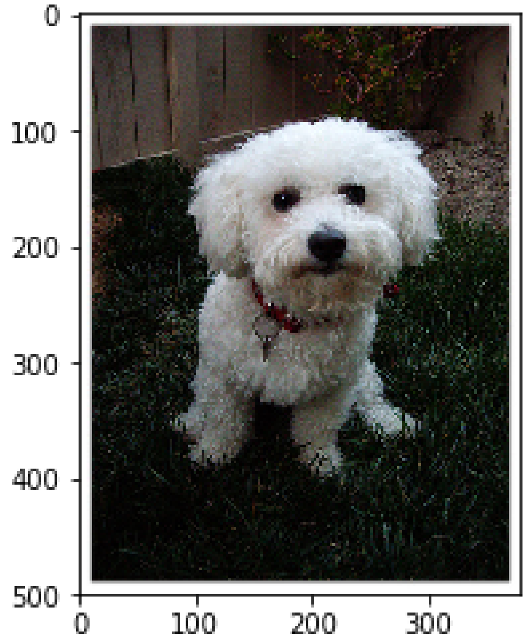
```
In [30]: def load_input_image_as_tensor(img_path):
             image = Image.open(img_path).convert('RGB')

             # discard the transparent, alpha channel (that's the :3) and add the batch dimensio
             image_tensor = data_transform_test(image)[:3,:,:].unsqueeze(0)

             return image_tensor
```

```
In [38]: import cv2
         import matplotlib.pyplot as plt
         %matplotlib inline

         ### 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 = [item[4:].replace("_", " ") for item in loaders_transfer['train'].dataset

         def predict_breed_transfer(img_path):
             # load the image and return the predicted breed
             image_tensor = load_input_image_as_tensor(img_path)
             if use_cuda:
                image_tensor = image_tensor.cuda()
```

```
    output = model_transfer(image_tensor)

    # display the image
    img = cv2.imread(img_path)
    # convert BGR image to RGB for plotting
    cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    plt.imshow(cv_rgb)
    plt.show()

    # class index
    pred = output.data.max(1, keepdim=True)[1]

    return class_names[pred]

num_samples = 5
for i in range(0, num_samples):
    test_file = dog_files[random.randint(0,len(dog_files))]
    print(f"^^^predict_breed_transfer({test_file}):", predict_breed_transfer(test_file)
```

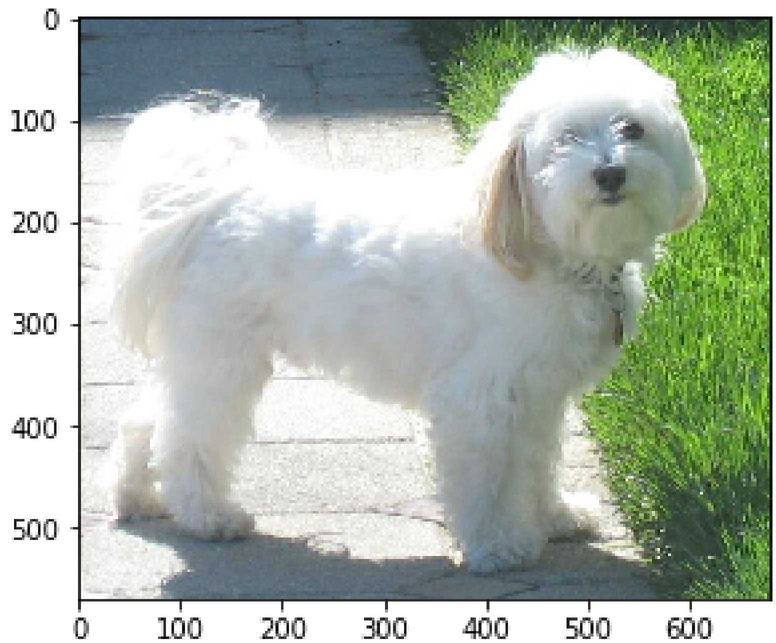



```
^^^predict_breed_transfer(/data/dog_images/train/007.American_foxhound/American_foxhound_00474.j
```

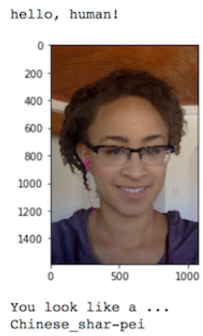



```
^^^predict_breed_transfer(/data/dog_images/train/024.Bichon_frise/Bichon_frise_01698.jpg): Bicho
```

```
^^^predict_breed_transfer(/data/dog_images/train/090.Italian_greyhound/Italian_greyhound_06132.j
```




```
^^^predict_breed_transfer(/data/dog_images/train/082.Havanese/Havanese_05610.jpg): Havanese
```



Sample Human Output

^^^predict_breed_transfer(/data/dog_images/train/021.Belgian_sheepdog/Belgian_sheepdog_01506.jpg

---

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 [39]:
```
### 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_breed = predict_breed_transfer(img_path)
    if dog_detector(img_path):
        print(f'Dog in "{img_path}" is a "{dog_breed}"')
    elif face_detector(img_path):
        print(f'Human in "{img_path}" resembles the dog breed "{dog_breed}"')
    else:
        print(f'Oops! Neither dog nor human is detected in file "{img_path}"')
```

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?

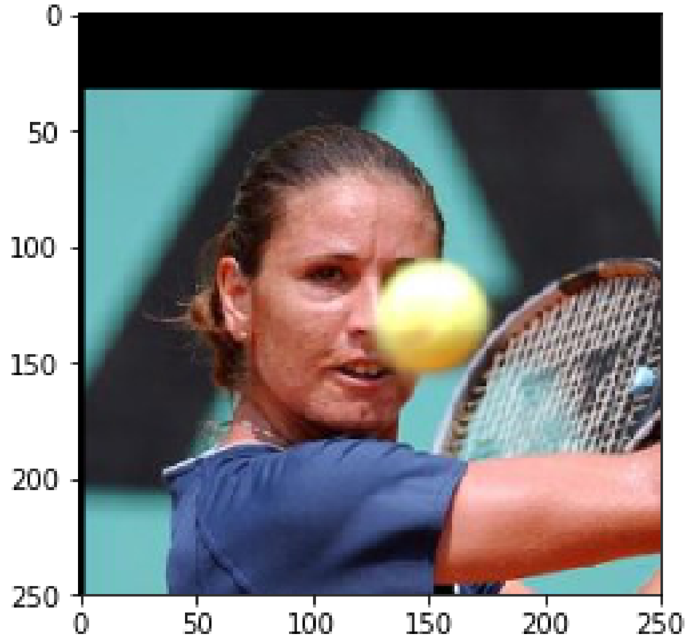
1.1.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

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.

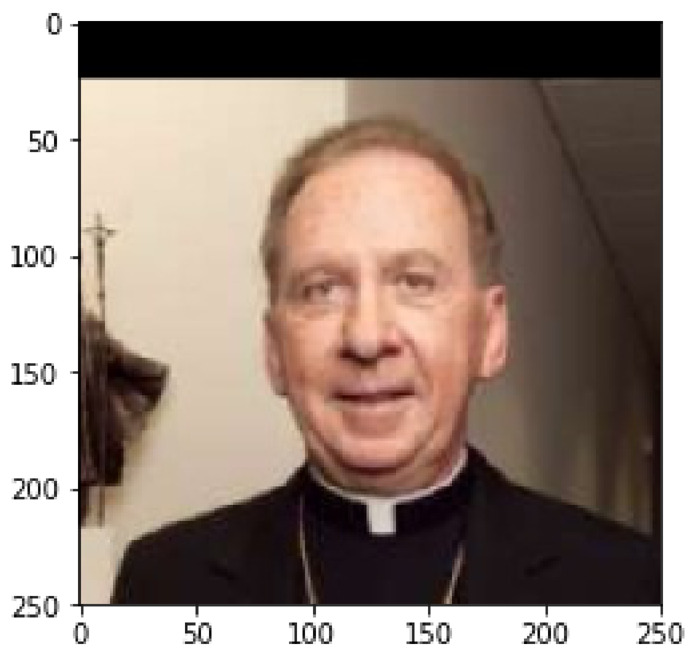
Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer: (Three possible points for improvement) - Ouput Expectation - On the task of dog vs human face detection - model perfomed well and in line with my expectation (based on evaluations in the beginning of this notebook). - On the task of dog breed classification, the model seems to be not doing great (only ~70% accuracy) and performed worse than my expectation.
- Areas for improvement - **Train on bigger dataset.** This dataset has only 6680/133 =~ 50 samples per dog breed and some of these breeds are so close to each other that even humans will have difficulty with the task of breed identification. - **Data augmentation.** Try more data augmentation techniques (eg: rotation) and train for longer duration for convergence. - **Tuning Hyperparameters.** Learning rate, optimizer, dropout, weight initialization, normalization etc - **Ensemble Models.** Combine the results from multiple classification models (as a voting scheme) and see if this approach does better.

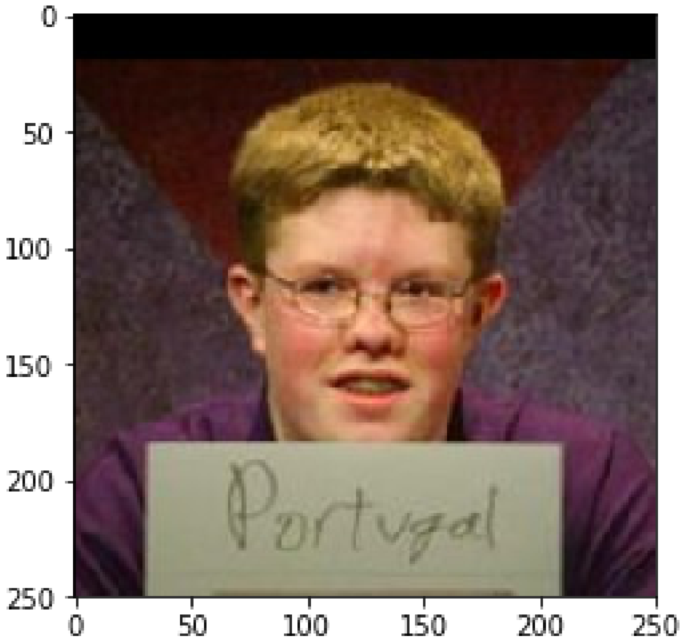
```
In [40]: ## TODO: Execute your algorithm from Step 6 on
         ## at least 6 images on your computer.
         ## Feel free to use as many code cells as needed.

         ## suggested code, below
         num_samples = 3
         for i in range(0, num_samples):
             test_file = human_files[random.randint(0,len(human_files))]
             run_app(test_file)
         for i in range(0, num_samples):
             test_file = dog_files[random.randint(0,len(dog_files))]
             run_app(test_file)
```



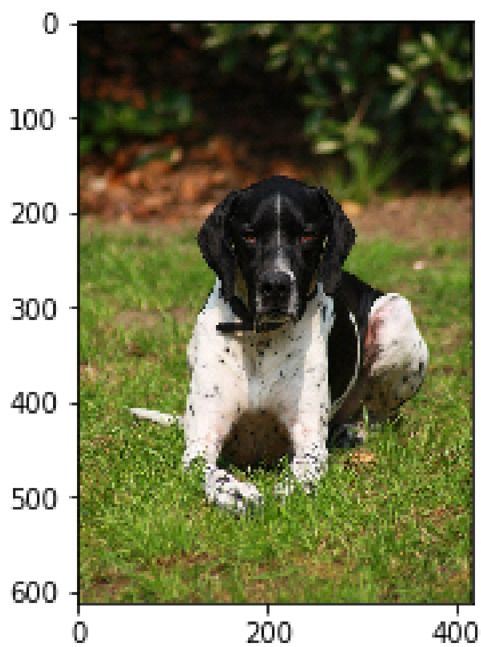
Human in "/data/lfw/Rita_Grande/Rita_Grande_0002.jpg" resembles the dog breed "Boykin spaniel"



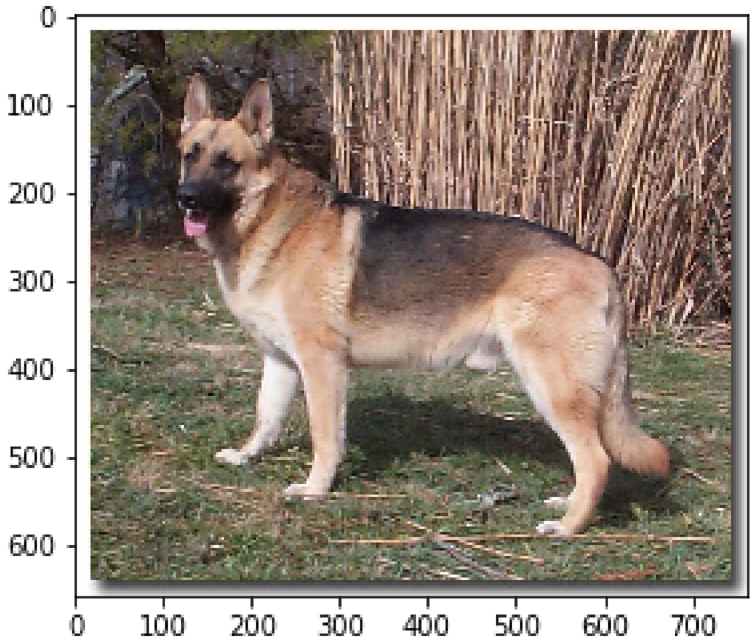
Human in "/data/lfw/Thomas_OBrien/Thomas_OBrien_0003.jpg" resembles the dog breed "English sprin



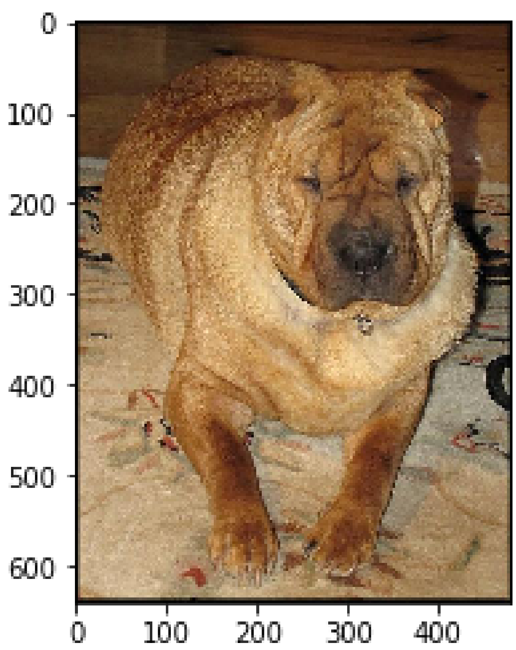
Human in "/data/lfw/James_Williams/James_Williams_0001.jpg" resembles the dog breed "Afghan houn



Dog in "/data/dog_images/train/122.Pointer/Pointer_07835.jpg" is a "Pointer"

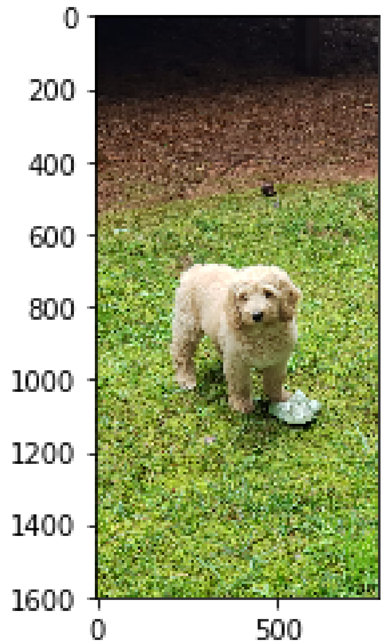


Dog in "/data/dog_images/train/071.German_shepherd_dog/German_shepherd_dog_04944.jpg" is a "Germ

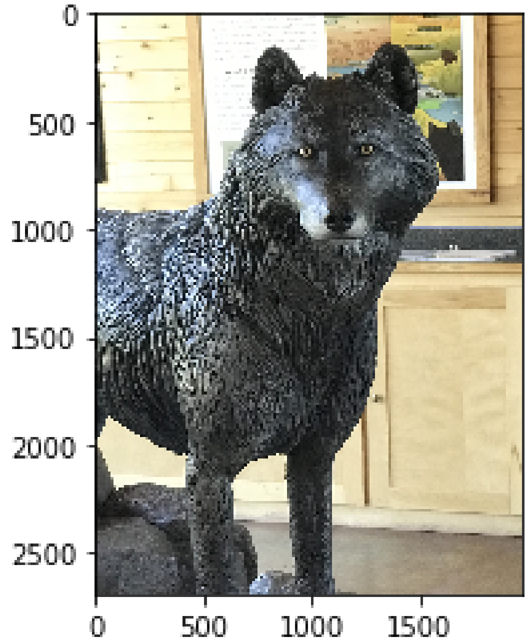


```
Dog in "/data/dog_images/train/050.Chinese_shar-pei/Chinese_shar-pei_03542.jpg" is a "Chinese sh
```

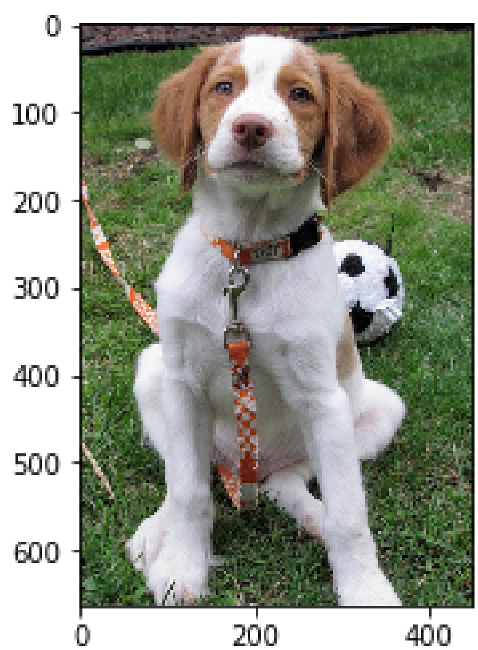
```
In [41]: for img_file in os.listdir('images'):
             img_path = os.path.join('images', img_file)
             run_app(img_path)
```



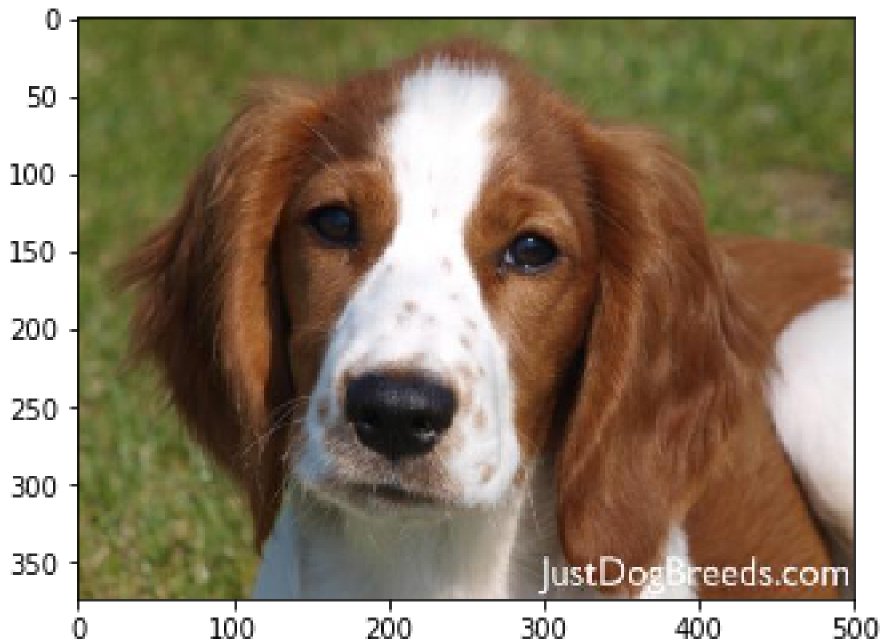

```
Dog in "images/73100552-FC55-4669-80B3-B6482F6E0CA8.JPG" is a "Bichon frise"
```



```
Human in "images/511F2544-99AD-48FF-88D3-482E59F6DCF9.jpg" resembles the dog breed "Kerry blue t
```

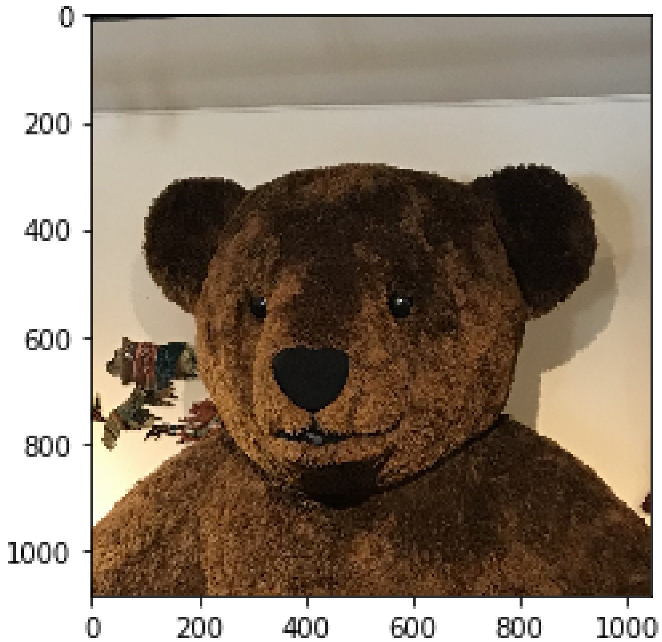


Dog in "images/Brittany_02625.jpg" is a "Brittany"

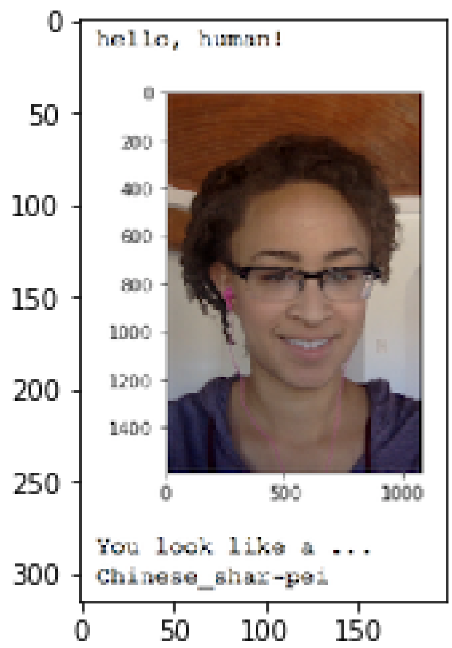


Dog in "images/Welsh_springer_spaniel_08203.jpg" is a "Welsh springer spaniel"

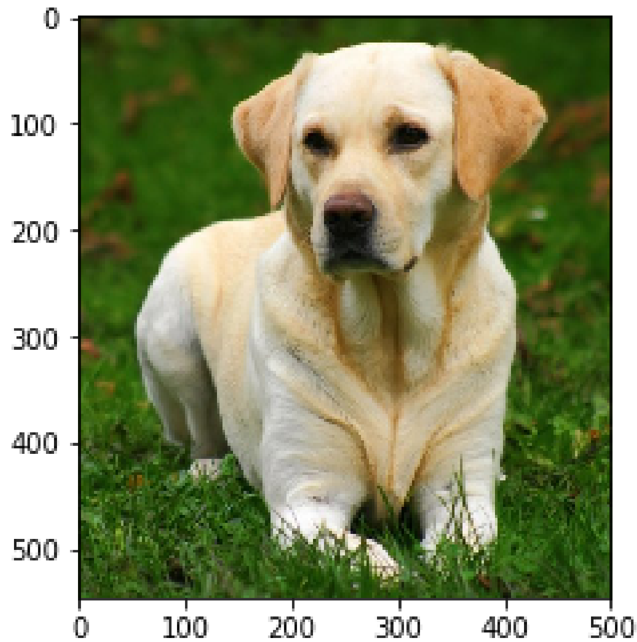
```
Dog in "images/Labrador_retriever_06449.jpg" is a "Labrador retriever"
```



```
Oops! Neither dog nor human is detected in file "images/F10C0AA7-8859-452D-B31C-E413C67AA03C.jpg
```

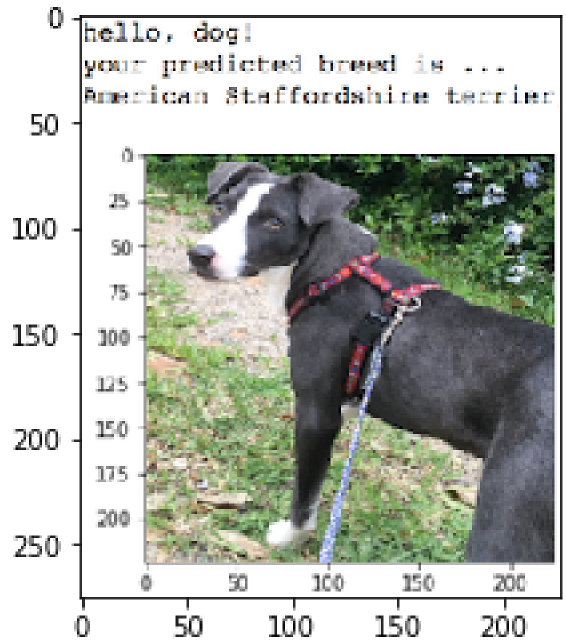


Human in "images/sample_human_output.png" resembles the dog breed "Basenji"

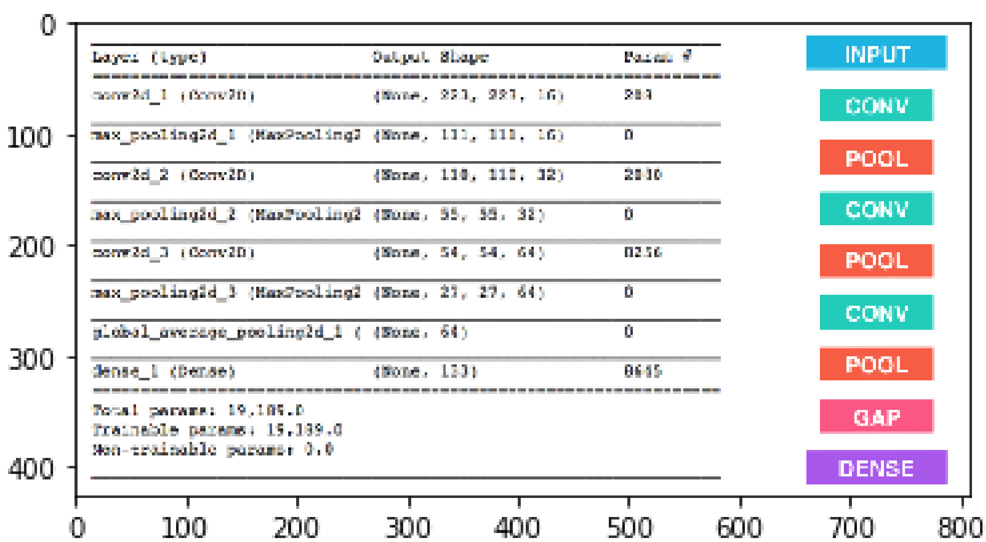


Dog in "images/Labrador_retriever_06457.jpg" is a "Labrador retriever"

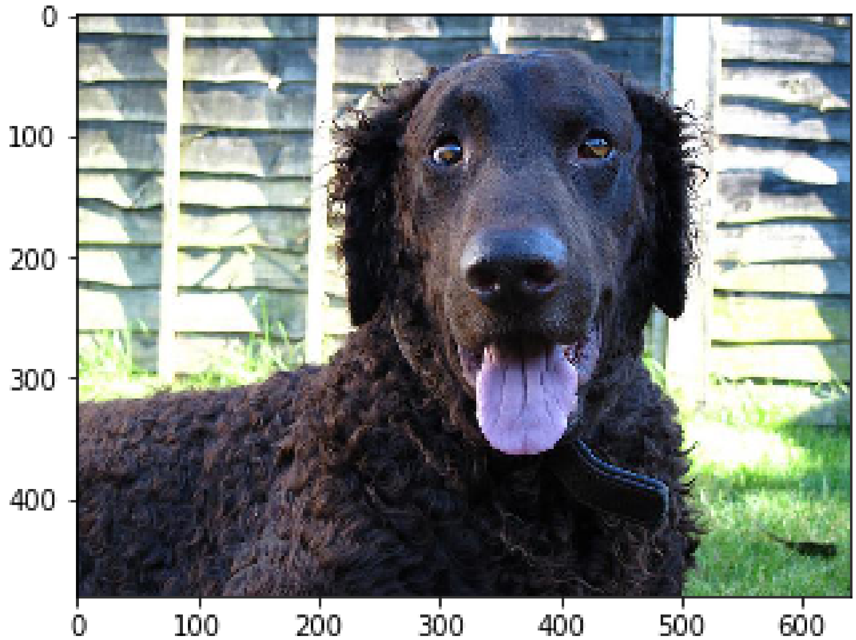
```
Dog in "images/American_water_spaniel_00648.jpg" is a "Boykin spaniel"
```



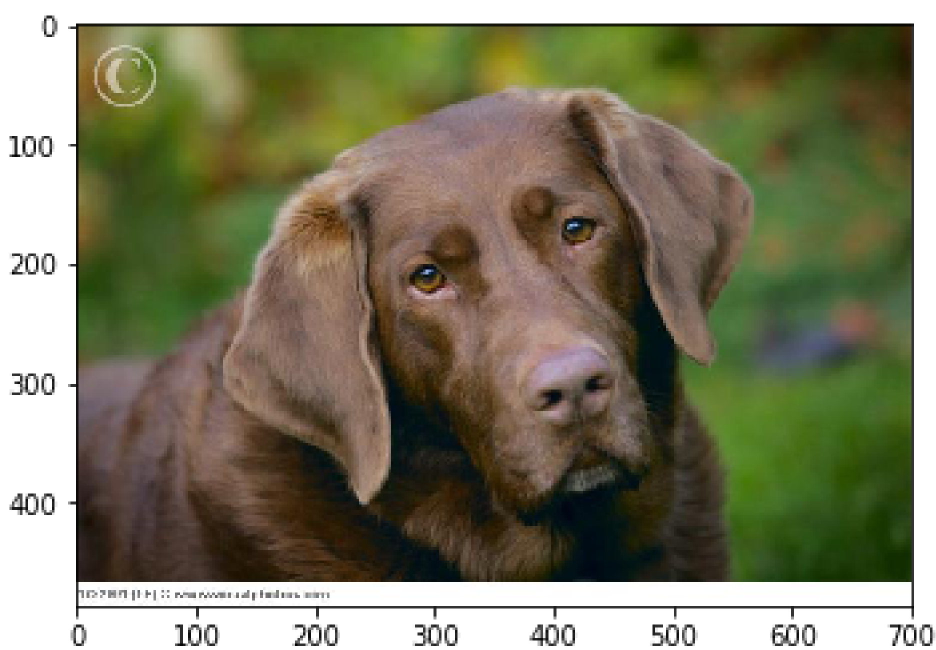

```
Dog in "images/sample_dog_output.png" is a "Smooth fox terrier"
```



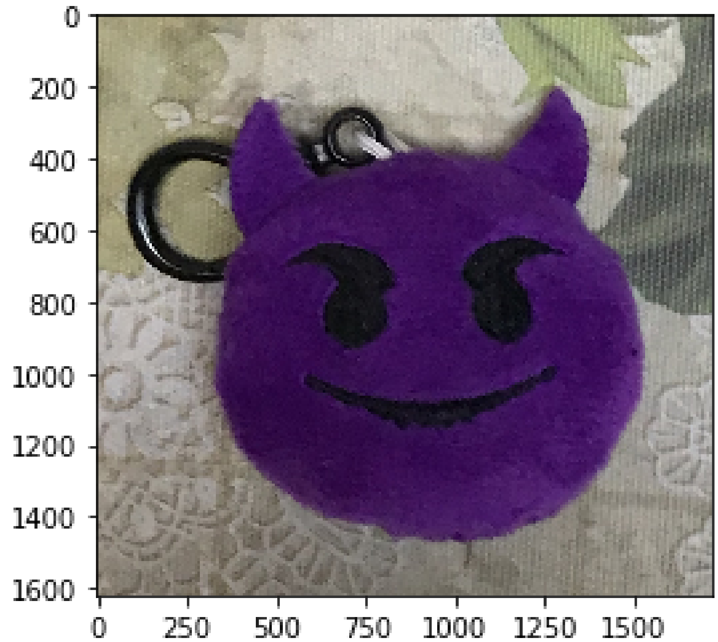
```
Oops! Neither dog nor human is detected in file "images/sample_cnn.png""
```



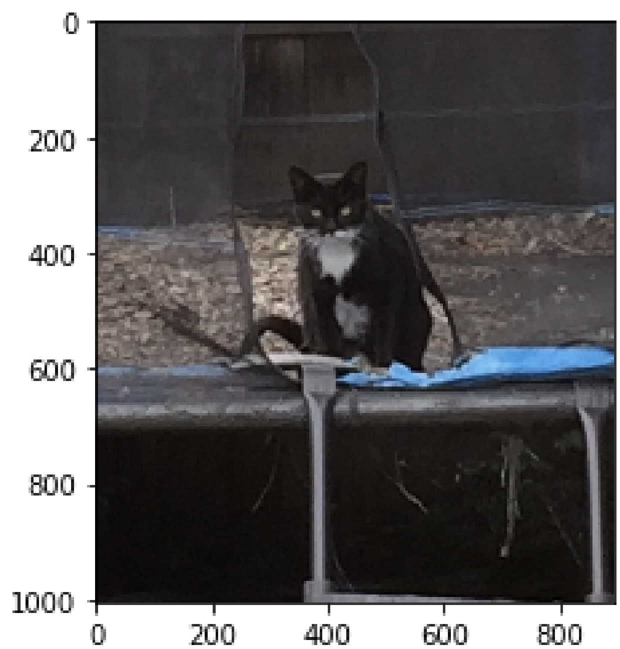

```
Dog in "images/Curly-coated_retriever_03896.jpg" is a "Curly-coated retriever"
```



```
Dog in "images/Labrador_retriever_06455.jpg" is a "Labrador retriever"
```




```
Oops! Neither dog nor human is detected in file "images/CC0E0128-09CF-412A-AB4F-912C7E847880.JPE
```



```
Dog in "images/D782B1C8-7A59-4B16-B8A4-CE4D99433C50.jpg" is a "Border collie"
```

```
In [ ]:
```