dog_app

September 8, 2020

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:

Note: if you are using the Udacity workspace, you *DO NOT* need to re-download these - they can be found in the `/data` folder as noted in the cell below.

- Download the dog dataset. 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 [65]: import numpy as np
         from glob import glob

         # load filenames for human and dog images
         human_files = np.array(glob("/data/lfw/*/*"))
         dog_files = np.array(glob("/data/dog_images/*/*/*"))

         # print number of images in each dataset
         print('There are %d total human images.' % len(human_files))
         print('There are %d total dog images.' % len(dog_files))
```

```
There are 13233 total human images.
There are 8351 total dog images.
```

Step 1: Detect Humans

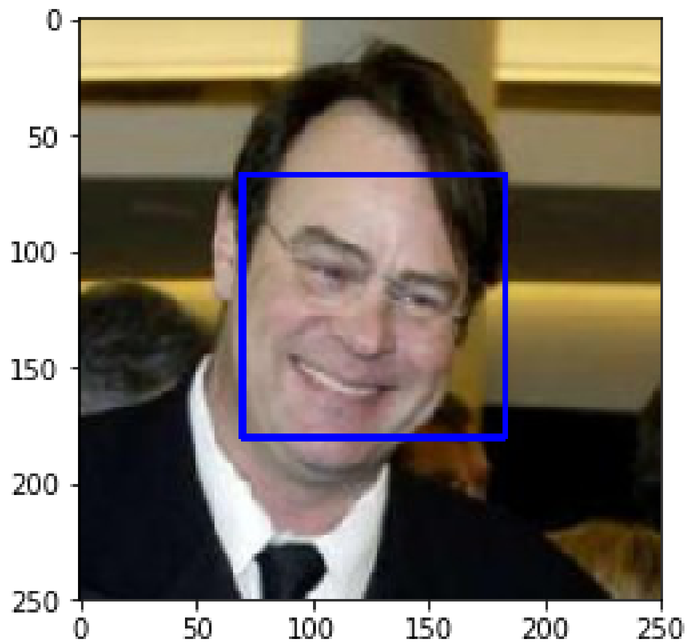
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

OpenCV provides many pre-trained face detectors, stored as XML files on github. We have downloaded one of these detectors and stored it in the `haarcascades` directory. In the next code cell, we demonstrate how to use this detector to find human faces in a sample image.

```
In [66]: import cv2
         import matplotlib.pyplot as plt
         %matplotlib inline

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

         # load color (BGR) image
         img = cv2.imread(human_files[0])
         # convert BGR image to grayscale
         gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

         # find faces in image
         faces = face_cascade.detectMultiScale(gray)

         # print number of faces detected in the image
         print('Number of faces detected:', len(faces))
```

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

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

Question 1: Use the code cell below to test the performance of the `face_detector` function.
- What percentage of the first 100 images in `human_files` have a detected human face?
- What percentage of the first 100 images in `dog_files` have a detected human face?

Ideally, we would like 100% of human images with a detected face and 0% of dog images with a detected face. You will see that our algorithm falls short of this goal, but still gives acceptable performance. We extract the file paths for the first 100 images from each of the datasets and store them in the numpy arrays `human_files_short` and `dog_files_short`.

Answer: (You can print out your results and/or write your percentages in this cell)

```
In [68]: 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.

         Count_faces=0
         Count_dogs=0
         for i in range(len(human_files_short)):
             test_face = human_files_short[i]
             if face_detector(test_face) is True:
                 Count_faces+=1
         percentage_faces=100*Count_faces/len(human_files_short)
         print('Face detection accuracy is '+str(percentage_faces)+'%')

         for i in range(len(dog_files_short)):
             test_dog = dog_files_short[i]
             if face_detector(test_dog) is True:
                 Count_dogs+=1
         percentage_dogs=100*Count_dogs/len(dog_files_short)
         print('Dog detection accuracy is '+str(percentage_dogs)+'%')
```

```
Face detection accuracy is 98.0%
Dog detection accuracy is 17.0%
```

We suggest the face detector from OpenCV as a potential way to detect human images in your algorithm, but you are free to explore other approaches, especially approaches that make use of deep learning :). Please use the code cell below to design and test your own face detection algorithm. If you decide to pursue this *optional* task, report performance on `human_files_short` and `dog_files_short`.

```
In [ ]: ### (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 [69]: import torch
         import torchvision.models as models

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

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

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

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

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 [78]: from PIL import Image
         import torchvision.transforms as transforms

         def VGG16_predict(img_path):
         #    '''
         ##    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

             dog_image = Image.open(img_path).convert('RGB')
             transform = transforms.Compose([transforms.Resize((224, 224)),
                                             transforms.ToTensor()])
             transformed_image=transform(dog_image)[:3,:,:].unsqueeze(0)

             if use_cuda:
                 transformed_image=transformed_image.cuda()
             neural_output=VGG16(transformed_image)
             return torch.max(neural_output, 1)[1].item()

In [63]:

In [79]: VGG16_predict(dog_files[12])

Out[79]: 243
```

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

```
print(dog_detector(human_files[6]))
print(dog_detector(dog_files[5]))
```

```
False
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:

```
In [81]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.

         Count_faces=0
         Count_dogs=0
         for i in range(len(human_files_short)):
             if dog_detector(human_files_short[i]) is True:
                 Count_faces+=1
         percentage_faces=100*Count_faces/len(human_files_short)
         print('Face detection accuracy is '+str(percentage_faces)+'%')

         for i in range(len(dog_files_short)):
             if dog_detector(dog_files_short[i]) is True:
                 Count_dogs+=1
         percentage_dogs=100*Count_dogs/len(dog_files_short)
         print('Dog detection accuracy is '+str(percentage_dogs)+'%')
```

```
Face detection accuracy is 0.0%
Dog detection accuracy is 94.0%
```

We suggest VGG-16 as a potential network to detect dog images in your algorithm, but you are free to explore other pre-trained networks (such as Inception-v3, ResNet-50, etc). Please use the code cell below to test other pre-trained PyTorch models. If you decide to pursue this *optional* task, report performance on `human_files_short` and `dog_files_short`.

```
In [ ]: ### (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 [82]:
```
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True
import torch
from torchvision import datasets, transforms


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

```
transform_validation = transforms.Compose([transforms.Resize((255, 255)),
                                     transforms.CenterCrop(224),
                                     transforms.ToTensor(),
                                     transforms.Normalize((0.5, 0.5, 0.5), (0.25, 0.25
transform_training = transforms.Compose([transforms.RandomResizedCrop(224),
                                     transforms.RandomRotation(90),
                                     transforms.ToTensor(),
                                     transforms.Normalize((0.5, 0.5, 0.5), (0.25, 0.25

transform_testing = transforms.Compose([transforms.Resize((224, 224)),
                                    transforms.ToTensor(),
                                    transforms.Normalize((0.5, 0.5, 0.5), (0.25, 0.25,

# choose the train, valid and test datasets
validation_dataset = datasets.ImageFolder('/data/dog_images/valid/', transform = transf
training_dataset = datasets.ImageFolder('/data/dog_images/train/', transform = transfor
testing_dataset = datasets.ImageFolder('/data/dog_images/test/', transform = transform_

# prepare data loaders
train_loader = torch.utils.data.DataLoader(training_dataset, batch_size = 10, num_worke
valid_loader = torch.utils.data.DataLoader(validation_dataset, batch_size = 10, num_wor
test_loader = torch.utils.data.DataLoader(testing_dataset, batch_size =10, num_workers

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

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

I first created the transforms for validation, training and testing. The training data consists of images resized to up to 224px, rotated between -90 to 90 degrees and normalized. I decided to use 224x224px as centercrop because 224px is the image size that many CNNs were trained with, including VGG and Resnet.

I tried to apply other transforms, like randomized operations, but I did not manage to make it work. Maybe the code was wrong, or the classes depricated. I decided to use these training augmentations to improve the quality of the training steps and to avoid overfitting.

The validation data is resized, cropped and normalized to make the validation step faster. The testing data is just resized.

I then create the datasets by applying the transforms.

Since I got an error message when attempting to use Cuda, I decided to keep the batch size relatively small. Hopefully this will speed up the training iterations, though at the detriment of training loss convergence.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [83]: import torch
         import numpy as np
```

```python
import torch.nn as nn
import torch.nn.functional as F

use_cuda = torch.cuda.is_available()
if use_cuda:
    torch.cuda.set_device(0)

class Net(nn.Module):
    ### TODO: choose an architecture, and complete the class
    def __init__(self):
        super(Net, self).__init__()

        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 , 600)
        self.fc2 = nn.Linear(600, 133)

        self.dropout = nn.Dropout(p=0.20)

    def forward(self, x):

        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))

        x = x.view(-1, 7*7*128)

        x = self.dropout(x)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.relu(self.fc2(x))
        return x

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

# instantiate the CNN
model_scratch = Net()

# move tensors to GPU if CUDA is available
if use_cuda:
    model_scratch.cuda()
```

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

Answer: I created 3 convolutional layers with pooling layers in between. I thought kernel size 3 was adequate for all layers to keep it simple. 1px padding was used so that only the layer depth changed, rather than lateral dimensions between convolutional layers.

Maxpooling was used to reduce dimensionality and prevent overfitting. I decided to not reduce too much, so window size 2 and stride 2 seemed ok. Finally, I forwarded the flattened output from the convolutional layer through 2 connected layers that were activated by relu. I used dropout with p=0.2 to prevent overfitting.

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 [84]: import torch.optim as optim

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

         ### TODO: select optimizer
         optimizer_scratch = optim.SGD(model_scratch.parameters(), lr = 0.04)
```

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

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

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

```
                optimizer.zero_grad()
                output = model(data)
                loss = criterion(output, target)
                loss.backward()
                optimizer.step()
                train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss)

                if batch_idx % 100 == 0:
                    print('Epoch: %d \tBatch: %d \tTraining Loss: %.6f' %(epoch, batch_idx

            ######################
            # validate the model #
            ######################
            model.eval()
            for batch_idx, (data, target) in enumerate(loaders['valid']):
                # move to GPU
                if use_cuda:
                    data, target = data.cuda(), target.cuda()
                ## update the average validation loss
                output = model(data)
                loss = criterion(output, target)
                valid_loss = 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:
                torch.save(model.state_dict(), save_path)
                print('Validation loss changed from {:.4f} to {:.4f}.  New value is {:4}'.f
                valid_loss_min = valid_loss
        # return trained model
        return model
```

```
In [86]: model_scratch = train(30, loaders_scratch, model_scratch, optimizer_scratch,
                               criterion_scratch, use_cuda, 'model_scratch.pt')
```

```
Epoch: 1        Batch: 1        Training Loss: 4.876195
Epoch: 1        Batch: 101        Training Loss: 4.890837
Epoch: 1        Batch: 201        Training Loss: 4.890780
Epoch: 1        Batch: 301        Training Loss: 4.890290
Epoch: 1        Batch: 401        Training Loss: 4.889462
Epoch: 1        Batch: 501        Training Loss: 4.887536
Epoch: 1        Batch: 601        Training Loss: 4.884293
```

```
Epoch: 1         Training Loss: 4.882493         Validation Loss: 4.856354
Validation loss changed from inf to 4.8564.  New value is 4.856353759765625
Epoch: 2         Batch: 1         Training Loss: 4.943690
Epoch: 2         Batch: 101         Training Loss: 4.856686
Epoch: 2         Batch: 201         Training Loss: 4.847610
Epoch: 2         Batch: 301         Training Loss: 4.833727
Epoch: 2         Batch: 401         Training Loss: 4.830675
Epoch: 2         Batch: 501         Training Loss: 4.826552
Epoch: 2         Batch: 601         Training Loss: 4.823737
Epoch: 2         Training Loss: 4.819830         Validation Loss: 4.760084
Validation loss changed from 4.8564 to 4.7601.  New value is 4.76008415222168
Epoch: 3         Batch: 1         Training Loss: 4.915740
Epoch: 3         Batch: 101         Training Loss: 4.763247
Epoch: 3         Batch: 201         Training Loss: 4.763479
Epoch: 3         Batch: 301         Training Loss: 4.757942
Epoch: 3         Batch: 401         Training Loss: 4.757942
Epoch: 3         Batch: 501         Training Loss: 4.759677
Epoch: 3         Batch: 601         Training Loss: 4.753852
Epoch: 3         Training Loss: 4.752524         Validation Loss: 4.647519
Validation loss changed from 4.7601 to 4.6475.  New value is 4.647518634796143
Epoch: 4         Batch: 1         Training Loss: 4.853993
Epoch: 4         Batch: 101         Training Loss: 4.739142
Epoch: 4         Batch: 201         Training Loss: 4.715018
Epoch: 4         Batch: 301         Training Loss: 4.715546
Epoch: 4         Batch: 401         Training Loss: 4.710485
Epoch: 4         Batch: 501         Training Loss: 4.700609
Epoch: 4         Batch: 601         Training Loss: 4.699837
Epoch: 4         Training Loss: 4.699611         Validation Loss: 4.647601
Epoch: 5         Batch: 1         Training Loss: 4.642446
Epoch: 5         Batch: 101         Training Loss: 4.671688
Epoch: 5         Batch: 201         Training Loss: 4.658917
Epoch: 5         Batch: 301         Training Loss: 4.663818
Epoch: 5         Batch: 401         Training Loss: 4.668627
Epoch: 5         Batch: 501         Training Loss: 4.669023
Epoch: 5         Batch: 601         Training Loss: 4.669760
Epoch: 5         Training Loss: 4.665609         Validation Loss: 4.530779
Validation loss changed from 4.6475 to 4.5308.  New value is 4.530778884887695
Epoch: 6         Batch: 1         Training Loss: 4.625591
Epoch: 6         Batch: 101         Training Loss: 4.644721
Epoch: 6         Batch: 201         Training Loss: 4.617912
Epoch: 6         Batch: 301         Training Loss: 4.630308
Epoch: 6         Batch: 401         Training Loss: 4.627867
Epoch: 6         Batch: 501         Training Loss: 4.623286
Epoch: 6         Batch: 601         Training Loss: 4.619180
Epoch: 6         Training Loss: 4.620897         Validation Loss: 4.523046
Validation loss changed from 4.5308 to 4.5230.  New value is 4.523046493530273
Epoch: 7         Batch: 1         Training Loss: 4.771703
Epoch: 7         Batch: 101         Training Loss: 4.594176
```

```
Epoch: 7         Batch: 201        Training Loss: 4.571884
Epoch: 7         Batch: 301        Training Loss: 4.576384
Epoch: 7         Batch: 401        Training Loss: 4.566557
Epoch: 7         Batch: 501        Training Loss: 4.565024
Epoch: 7         Batch: 601        Training Loss: 4.560324
Epoch: 7         Training Loss: 4.559960         Validation Loss: 4.437730
Validation loss changed from 4.5230 to 4.4377.  New value is 4.437729835510254
Epoch: 8         Batch: 1        Training Loss: 5.093839
Epoch: 8         Batch: 101        Training Loss: 4.527417
Epoch: 8         Batch: 201        Training Loss: 4.514925
Epoch: 8         Batch: 301        Training Loss: 4.506995
Epoch: 8         Batch: 401        Training Loss: 4.515676
Epoch: 8         Batch: 501        Training Loss: 4.512192
Epoch: 8         Batch: 601        Training Loss: 4.515654
Epoch: 8         Training Loss: 4.517395         Validation Loss: 4.319198
Validation loss changed from 4.4377 to 4.3192.  New value is 4.319198131561279
Epoch: 9         Batch: 1        Training Loss: 4.815304
Epoch: 9         Batch: 101        Training Loss: 4.472959
Epoch: 9         Batch: 201        Training Loss: 4.476808
Epoch: 9         Batch: 301        Training Loss: 4.470770
Epoch: 9         Batch: 401        Training Loss: 4.452850
Epoch: 9         Batch: 501        Training Loss: 4.450154
Epoch: 9         Batch: 601        Training Loss: 4.455259
Epoch: 9         Training Loss: 4.453271         Validation Loss: 4.327108
Epoch: 10         Batch: 1        Training Loss: 4.539146
Epoch: 10         Batch: 101        Training Loss: 4.398484
Epoch: 10         Batch: 201        Training Loss: 4.362986
Epoch: 10         Batch: 301        Training Loss: 4.380854
Epoch: 10         Batch: 401        Training Loss: 4.379040
Epoch: 10         Batch: 501        Training Loss: 4.388360
Epoch: 10         Batch: 601        Training Loss: 4.389386
Epoch: 10         Training Loss: 4.395785         Validation Loss: 4.291129
Validation loss changed from 4.3192 to 4.2911.  New value is 4.291128635406494
Epoch: 11         Batch: 1        Training Loss: 4.318161
Epoch: 11         Batch: 101        Training Loss: 4.336158
Epoch: 11         Batch: 201        Training Loss: 4.334092
Epoch: 11         Batch: 301        Training Loss: 4.345280
Epoch: 11         Batch: 401        Training Loss: 4.341688
Epoch: 11         Batch: 501        Training Loss: 4.349188
Epoch: 11         Batch: 601        Training Loss: 4.345989
Epoch: 11         Training Loss: 4.343179         Validation Loss: 4.230071
Validation loss changed from 4.2911 to 4.2301.  New value is 4.230071067810059
Epoch: 12         Batch: 1        Training Loss: 4.290127
Epoch: 12         Batch: 101        Training Loss: 4.317916
Epoch: 12         Batch: 201        Training Loss: 4.307762
Epoch: 12         Batch: 301        Training Loss: 4.307004
Epoch: 12         Batch: 401        Training Loss: 4.304451
Epoch: 12         Batch: 501        Training Loss: 4.299220
```

```
Epoch: 12        Batch: 601        Training Loss: 4.306761
Epoch: 12        Training Loss: 4.299767        Validation Loss: 4.127627
Validation loss changed from 4.2301 to 4.1276.  New value is 4.127627372741699
Epoch: 13        Batch: 1        Training Loss: 4.236799
Epoch: 13        Batch: 101        Training Loss: 4.252677
Epoch: 13        Batch: 201        Training Loss: 4.217273
Epoch: 13        Batch: 301        Training Loss: 4.219673
Epoch: 13        Batch: 401        Training Loss: 4.232332
Epoch: 13        Batch: 501        Training Loss: 4.227416
Epoch: 13        Batch: 601        Training Loss: 4.226707
Epoch: 13        Training Loss: 4.230302        Validation Loss: 4.155729
Epoch: 14        Batch: 1        Training Loss: 4.726972
Epoch: 14        Batch: 101        Training Loss: 4.215702
Epoch: 14        Batch: 201        Training Loss: 4.228807
Epoch: 14        Batch: 301        Training Loss: 4.208229
Epoch: 14        Batch: 401        Training Loss: 4.197175
Epoch: 14        Batch: 501        Training Loss: 4.204523
Epoch: 14        Batch: 601        Training Loss: 4.201984
Epoch: 14        Training Loss: 4.204855        Validation Loss: 4.100717
Validation loss changed from 4.1276 to 4.1007.  New value is 4.100716590881348
Epoch: 15        Batch: 1        Training Loss: 4.265543
Epoch: 15        Batch: 101        Training Loss: 4.110864
Epoch: 15        Batch: 201        Training Loss: 4.155467
Epoch: 15        Batch: 301        Training Loss: 4.158370
Epoch: 15        Batch: 401        Training Loss: 4.148656
Epoch: 15        Batch: 501        Training Loss: 4.152812
Epoch: 15        Batch: 601        Training Loss: 4.162902
Epoch: 15        Training Loss: 4.164909        Validation Loss: 4.035397
Validation loss changed from 4.1007 to 4.0354.  New value is 4.035397052764893
Epoch: 16        Batch: 1        Training Loss: 4.524873
Epoch: 16        Batch: 101        Training Loss: 4.078904
Epoch: 16        Batch: 201        Training Loss: 4.094869
Epoch: 16        Batch: 301        Training Loss: 4.120989
Epoch: 16        Batch: 401        Training Loss: 4.121828
Epoch: 16        Batch: 501        Training Loss: 4.115223
Epoch: 16        Batch: 601        Training Loss: 4.112271
Epoch: 16        Training Loss: 4.114011        Validation Loss: 3.944397
Validation loss changed from 4.0354 to 3.9444.  New value is 3.94439697265625
Epoch: 17        Batch: 1        Training Loss: 3.836016
Epoch: 17        Batch: 101        Training Loss: 4.089679
Epoch: 17        Batch: 201        Training Loss: 4.078568
Epoch: 17        Batch: 301        Training Loss: 4.075285
Epoch: 17        Batch: 401        Training Loss: 4.082277
Epoch: 17        Batch: 501        Training Loss: 4.072808
Epoch: 17        Batch: 601        Training Loss: 4.069186
Epoch: 17        Training Loss: 4.074474        Validation Loss: 3.888692
Validation loss changed from 3.9444 to 3.8887.  New value is 3.8886921405792236
Epoch: 18        Batch: 1        Training Loss: 3.134437
```

```
Epoch: 18         Batch: 101         Training Loss: 4.051461
Epoch: 18         Batch: 201         Training Loss: 4.020488
Epoch: 18         Batch: 301         Training Loss: 4.039019
Epoch: 18         Batch: 401         Training Loss: 4.040560
Epoch: 18         Batch: 501         Training Loss: 4.031721
Epoch: 18         Batch: 601         Training Loss: 4.033983
Epoch: 18         Training Loss: 4.045272         Validation Loss: 3.869316
Validation loss changed from 3.8887 to 3.8693.  New value is 3.8693156242370605
Epoch: 19         Batch: 1         Training Loss: 3.663805
Epoch: 19         Batch: 101         Training Loss: 4.040644
Epoch: 19         Batch: 201         Training Loss: 4.041574
Epoch: 19         Batch: 301         Training Loss: 4.035224
Epoch: 19         Batch: 401         Training Loss: 4.035057
Epoch: 19         Batch: 501         Training Loss: 4.024794
Epoch: 19         Batch: 601         Training Loss: 4.011472
Epoch: 19         Training Loss: 4.012155         Validation Loss: 3.913127
Epoch: 20         Batch: 1         Training Loss: 3.683182
Epoch: 20         Batch: 101         Training Loss: 3.982702
Epoch: 20         Batch: 201         Training Loss: 3.987751
Epoch: 20         Batch: 301         Training Loss: 3.977424
Epoch: 20         Batch: 401         Training Loss: 3.978517
Epoch: 20         Batch: 501         Training Loss: 3.971265
Epoch: 20         Batch: 601         Training Loss: 3.978771
Epoch: 20         Training Loss: 3.970600         Validation Loss: 3.996089
Epoch: 21         Batch: 1         Training Loss: 4.840940
Epoch: 21         Batch: 101         Training Loss: 3.935676
Epoch: 21         Batch: 201         Training Loss: 3.907315
Epoch: 21         Batch: 301         Training Loss: 3.934448
Epoch: 21         Batch: 401         Training Loss: 3.930236
Epoch: 21         Batch: 501         Training Loss: 3.939122
Epoch: 21         Batch: 601         Training Loss: 3.939350
Epoch: 21         Training Loss: 3.943223         Validation Loss: 3.894447
Epoch: 22         Batch: 1         Training Loss: 3.653865
Epoch: 22         Batch: 101         Training Loss: 3.903923
Epoch: 22         Batch: 201         Training Loss: 3.897068
Epoch: 22         Batch: 301         Training Loss: 3.912306
Epoch: 22         Batch: 401         Training Loss: 3.923211
Epoch: 22         Batch: 501         Training Loss: 3.927507
Epoch: 22         Batch: 601         Training Loss: 3.920169
Epoch: 22         Training Loss: 3.931327         Validation Loss: 3.839771
Validation loss changed from 3.8693 to 3.8398.  New value is 3.839771270751953
Epoch: 23         Batch: 1         Training Loss: 3.859473
Epoch: 23         Batch: 101         Training Loss: 3.889431
Epoch: 23         Batch: 201         Training Loss: 3.920490
Epoch: 23         Batch: 301         Training Loss: 3.904346
Epoch: 23         Batch: 401         Training Loss: 3.899681
Epoch: 23         Batch: 501         Training Loss: 3.899548
Epoch: 23         Batch: 601         Training Loss: 3.887300
```

```
Epoch: 23         Training Loss: 3.890280        Validation Loss: 3.894041
Epoch: 24         Batch: 1        Training Loss: 3.670111
Epoch: 24         Batch: 101        Training Loss: 3.750240
Epoch: 24         Batch: 201        Training Loss: 3.789108
Epoch: 24         Batch: 301        Training Loss: 3.824750
Epoch: 24         Batch: 401        Training Loss: 3.840643
Epoch: 24         Batch: 501        Training Loss: 3.840483
Epoch: 24         Batch: 601        Training Loss: 3.852867
Epoch: 24         Training Loss: 3.857211        Validation Loss: 3.897867
Epoch: 25         Batch: 1        Training Loss: 4.592988
Epoch: 25         Batch: 101        Training Loss: 3.865927
Epoch: 25         Batch: 201        Training Loss: 3.838220
Epoch: 25         Batch: 301        Training Loss: 3.840429
Epoch: 25         Batch: 401        Training Loss: 3.830123
Epoch: 25         Batch: 501        Training Loss: 3.843539
Epoch: 25         Batch: 601        Training Loss: 3.850296
Epoch: 25         Training Loss: 3.849471        Validation Loss: 3.727453
Validation loss changed from 3.8398 to 3.7275.  New value is 3.7274529933929443
Epoch: 26         Batch: 1        Training Loss: 3.963204
Epoch: 26         Batch: 101        Training Loss: 3.832067
Epoch: 26         Batch: 201        Training Loss: 3.789786
Epoch: 26         Batch: 301        Training Loss: 3.788140
Epoch: 26         Batch: 401        Training Loss: 3.802006
Epoch: 26         Batch: 501        Training Loss: 3.807149
Epoch: 26         Batch: 601        Training Loss: 3.805319
Epoch: 26         Training Loss: 3.795683        Validation Loss: 3.754413
Epoch: 27         Batch: 1        Training Loss: 3.324042
Epoch: 27         Batch: 101        Training Loss: 3.658386
Epoch: 27         Batch: 201        Training Loss: 3.721763
Epoch: 27         Batch: 301        Training Loss: 3.743671
Epoch: 27         Batch: 401        Training Loss: 3.793220
Epoch: 27         Batch: 501        Training Loss: 3.802125
Epoch: 27         Batch: 601        Training Loss: 3.802186
Epoch: 27         Training Loss: 3.805079        Validation Loss: 3.627789
Validation loss changed from 3.7275 to 3.6278.  New value is 3.627788543701172
Epoch: 28         Batch: 1        Training Loss: 3.282414
Epoch: 28         Batch: 101        Training Loss: 3.673721
Epoch: 28         Batch: 201        Training Loss: 3.709862
Epoch: 28         Batch: 301        Training Loss: 3.710604
Epoch: 28         Batch: 401        Training Loss: 3.719113
Epoch: 28         Batch: 501        Training Loss: 3.748349
Epoch: 28         Batch: 601        Training Loss: 3.754121
Epoch: 28         Training Loss: 3.752730        Validation Loss: 3.751163
Epoch: 29         Batch: 1        Training Loss: 4.324855
Epoch: 29         Batch: 101        Training Loss: 3.713336
Epoch: 29         Batch: 201        Training Loss: 3.777412
Epoch: 29         Batch: 301        Training Loss: 3.790137
Epoch: 29         Batch: 401        Training Loss: 3.778344
```

```
Epoch: 29        Batch: 501        Training Loss: 3.775936
Epoch: 29        Batch: 601        Training Loss: 3.763662
Epoch: 29        Training Loss: 3.759918        Validation Loss: 3.729019
Epoch: 30        Batch: 1        Training Loss: 3.230650
Epoch: 30        Batch: 101        Training Loss: 3.666762
Epoch: 30        Batch: 201        Training Loss: 3.668352
Epoch: 30        Batch: 301        Training Loss: 3.662665
Epoch: 30        Batch: 401        Training Loss: 3.674182
Epoch: 30        Batch: 501        Training Loss: 3.694981
Epoch: 30        Batch: 601        Training Loss: 3.701793
Epoch: 30        Training Loss: 3.705421        Validation Loss: 3.779094
```

In [87]:
```
model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

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 [88]:
```
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 = data.to('cuda')
            target = target.to('cuda')
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the loss
        loss = criterion(output, target)
        # update average test loss
        test_loss = test_loss + ((1 / (batch_idx + 1)) * (loss.data - test_loss))
        # convert output probabilities to predicted class
        pred = output.data.max(1, keepdim=True)[1]
        # compare predictions to true label
        correct += np.sum(np.squeeze(pred.eq(target.data.view_as(pred))).cpu().numpy())
        total += data.size(0)

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

    print('\nTest Accuracy: %2d%% (%2d/%2d)' % (
```

```
                100. * correct / total, correct, total))

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

```
Test Loss: 3.705229


Test Accuracy: 13% (111/836)
```

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 [89]: ## TODO: Specify data loaders
         transferred_loader = loaders_scratch.copy()
```

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

         ## TODO: Specify model architecture

         model_transfer = models.resnet50(pretrained = True)

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

         model_transfer.fc = nn.Linear(2048, 133, bias = True)

         if use_cuda:
             model_transfer = model_transfer.cuda()

         model_transfer
```

```
Out[90]: ResNet(
           (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
           (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
           (relu): ReLU(inplace)
           (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
           (layer1): Sequential(
             (0): Bottleneck(
               (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
               (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
               (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
               (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
               (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
               (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
               (relu): ReLU(inplace)
               (downsample): Sequential(
                 (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
                 (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
               )
             )
             (1): Bottleneck(
               (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
               (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
               (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
               (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
               (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
               (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
               (relu): ReLU(inplace)
             )
             (2): Bottleneck(
               (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
               (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
               (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
               (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
               (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
               (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
               (relu): ReLU(inplace)
             )
           )
           (layer2): Sequential(
             (0): Bottleneck(
               (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
               (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
               (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias
               (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
               (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
               (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
               (relu): ReLU(inplace)
               (downsample): Sequential(
```

```
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (relu): ReLU(inplace)
    )
    (2): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (relu): ReLU(inplace)
    )
    (3): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (relu): ReLU(inplace)
    )
  )
  (layer3): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      (relu): ReLU(inplace)
      (downsample): Sequential(
        (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

```
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
        (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
        (relu): ReLU(inplace)
      )
      (2): Bottleneck(
        (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
        (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
        (relu): ReLU(inplace)
      )
      (3): Bottleneck(
        (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
        (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
        (relu): ReLU(inplace)
      )
      (4): Bottleneck(
        (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
        (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
        (relu): ReLU(inplace)
      )
      (5): Bottleneck(
        (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
        (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
        (relu): ReLU(inplace)
      )
    )
    (layer4): Sequential(
      (0): Bottleneck(
        (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
```

```
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
        (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stat
        (relu): ReLU(inplace)
        (downsample): Sequential(
          (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
          (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stat
        )
      )
      (1): Bottleneck(
        (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
        (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stat
        (relu): ReLU(inplace)
      )
      (2): Bottleneck(
        (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
        (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stat
        (relu): ReLU(inplace)
      )
    )
    (avgpool): AvgPool2d(kernel_size=7, stride=1, padding=0)
    (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 decided to use Resnet50 since it is sufficiently deep to process the convoluted data in more detail while only needing a single functional layer at the end to get the final output. I tried to use other networks like Alexnet, Deepnet and others. In most cases I had to set requires_grad to True, which made the networks inefficient. In other to make Resnet50 work, I only had to make sure the dimensions of the output from the final convolutional layer and the fully connected layer are aligned.

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 [91]: criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.SGD(model_transfer.fc.parameters(), lr = 0.015)
```

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 [92]:
```
# train the model
train(10, transferred_loader, model_transfer, optimizer_transfer, criterion_transfer, u

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

```
Epoch: 1         Batch: 1        Training Loss: 4.977014
Epoch: 1         Batch: 101        Training Loss: 4.871393
Epoch: 1         Batch: 201        Training Loss: 4.713460
Epoch: 1         Batch: 301        Training Loss: 4.555293
Epoch: 1         Batch: 401        Training Loss: 4.415236
Epoch: 1         Batch: 501        Training Loss: 4.294582
Epoch: 1         Batch: 601        Training Loss: 4.173687
Epoch: 1         Training Loss: 4.100103         Validation Loss: 2.472441
Validation loss changed from inf to 2.4724.  New value is 2.4724414348602295
Epoch: 2         Batch: 1        Training Loss: 3.553586
Epoch: 2         Batch: 101        Training Loss: 3.279460
Epoch: 2         Batch: 201        Training Loss: 3.181521
Epoch: 2         Batch: 301        Training Loss: 3.130298
Epoch: 2         Batch: 401        Training Loss: 3.078172
Epoch: 2         Batch: 501        Training Loss: 3.031755
Epoch: 2         Batch: 601        Training Loss: 2.991960
Epoch: 2         Training Loss: 2.963060         Validation Loss: 1.529848
Validation loss changed from 2.4724 to 1.5298.  New value is 1.5298482179641724
Epoch: 3         Batch: 1        Training Loss: 2.564997
Epoch: 3         Batch: 101        Training Loss: 2.494681
Epoch: 3         Batch: 201        Training Loss: 2.508966
Epoch: 3         Batch: 301        Training Loss: 2.482943
Epoch: 3         Batch: 401        Training Loss: 2.470545
Epoch: 3         Batch: 501        Training Loss: 2.465252
Epoch: 3         Batch: 601        Training Loss: 2.435211
Epoch: 3         Training Loss: 2.422454         Validation Loss: 1.252477
Validation loss changed from 1.5298 to 1.2525.  New value is 1.2524768114089966
Epoch: 4         Batch: 1        Training Loss: 2.858080
Epoch: 4         Batch: 101        Training Loss: 2.235434
Epoch: 4         Batch: 201        Training Loss: 2.230567
Epoch: 4         Batch: 301        Training Loss: 2.228891
Epoch: 4         Batch: 401        Training Loss: 2.245988
Epoch: 4         Batch: 501        Training Loss: 2.240542
Epoch: 4         Batch: 601        Training Loss: 2.215743
Epoch: 4         Training Loss: 2.200469         Validation Loss: 1.002543
Validation loss changed from 1.2525 to 1.0025.  New value is 1.0025429725646973
Epoch: 5         Batch: 1        Training Loss: 2.450544
Epoch: 5         Batch: 101        Training Loss: 2.109457
```

```
Epoch: 5        Batch: 201        Training Loss: 2.093014
Epoch: 5        Batch: 301        Training Loss: 2.085296
Epoch: 5        Batch: 401        Training Loss: 2.061990
Epoch: 5        Batch: 501        Training Loss: 2.057966
Epoch: 5        Batch: 601        Training Loss: 2.053716
Epoch: 5        Training Loss: 2.057136        Validation Loss: 0.907861
Validation loss changed from 1.0025 to 0.9079.  New value is 0.9078614115715027
Epoch: 6        Batch: 1        Training Loss: 1.822885
Epoch: 6        Batch: 101        Training Loss: 2.050828
Epoch: 6        Batch: 201        Training Loss: 1.996058
Epoch: 6        Batch: 301        Training Loss: 2.021533
Epoch: 6        Batch: 401        Training Loss: 2.007704
Epoch: 6        Batch: 501        Training Loss: 1.999653
Epoch: 6        Batch: 601        Training Loss: 1.987286
Epoch: 6        Training Loss: 1.976605        Validation Loss: 0.806183
Validation loss changed from 0.9079 to 0.8062.  New value is 0.8061829805374146
Epoch: 7        Batch: 1        Training Loss: 2.527228
Epoch: 7        Batch: 101        Training Loss: 1.914014
Epoch: 7        Batch: 201        Training Loss: 1.903932
Epoch: 7        Batch: 301        Training Loss: 1.889925
Epoch: 7        Batch: 401        Training Loss: 1.891588
Epoch: 7        Batch: 501        Training Loss: 1.882888
Epoch: 7        Batch: 601        Training Loss: 1.883925
Epoch: 7        Training Loss: 1.882582        Validation Loss: 0.784797
Validation loss changed from 0.8062 to 0.7848.  New value is 0.7847970128059387
Epoch: 8        Batch: 1        Training Loss: 2.084092
Epoch: 8        Batch: 101        Training Loss: 1.751248
Epoch: 8        Batch: 201        Training Loss: 1.770309
Epoch: 8        Batch: 301        Training Loss: 1.772175
Epoch: 8        Batch: 401        Training Loss: 1.781812
Epoch: 8        Batch: 501        Training Loss: 1.780459
Epoch: 8        Batch: 601        Training Loss: 1.796677
Epoch: 8        Training Loss: 1.797799        Validation Loss: 0.729054
Validation loss changed from 0.7848 to 0.7291.  New value is 0.7290537357330322
Epoch: 9        Batch: 1        Training Loss: 1.633300
Epoch: 9        Batch: 101        Training Loss: 1.771263
Epoch: 9        Batch: 201        Training Loss: 1.792794
Epoch: 9        Batch: 301        Training Loss: 1.814044
Epoch: 9        Batch: 401        Training Loss: 1.801196
Epoch: 9        Batch: 501        Training Loss: 1.794491
Epoch: 9        Batch: 601        Training Loss: 1.786113
Epoch: 9        Training Loss: 1.785018        Validation Loss: 0.738573
Epoch: 10        Batch: 1        Training Loss: 1.842127
Epoch: 10        Batch: 101        Training Loss: 1.767758
Epoch: 10        Batch: 201        Training Loss: 1.747053
Epoch: 10        Batch: 301        Training Loss: 1.746160
Epoch: 10        Batch: 401        Training Loss: 1.759368
Epoch: 10        Batch: 501        Training Loss: 1.749673
```

```
Epoch: 10         Batch: 601        Training Loss: 1.749084
Epoch: 10         Training Loss: 1.754589         Validation Loss: 0.684340
Validation loss changed from 0.7291 to 0.6843.  New value is 0.6843398809432983
```

In [ ]:

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 [93]:
```
test(transferred_loader, model_transfer, criterion_transfer, use_cuda)
```

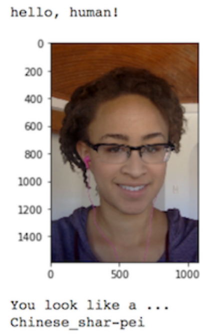
```
Test Loss: 0.732599


Test Accuracy: 79% (663/836)
```

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 [94]:
```
### 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 transferred_loader['train'].datas


def predict_doggo(img_path):
    # load the image and return the predicted breed
    transform = transforms.Compose([transforms.Resize((224, 224)),
                                    transforms.ToTensor(),
                                    transforms.Normalize((0.5, 0.5, 0.5), (0.25, 0.25,
    photo = Image.open(img_path).convert('RGB')
    transformed_photo=transform(photo)[:3,:,:].unsqueeze(0)
    model = model_transfer.cpu()
    model.eval()
    idx = torch.argmax(model(transformed_photo))
    return class_names[idx]
```

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



Sample Human Output

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.

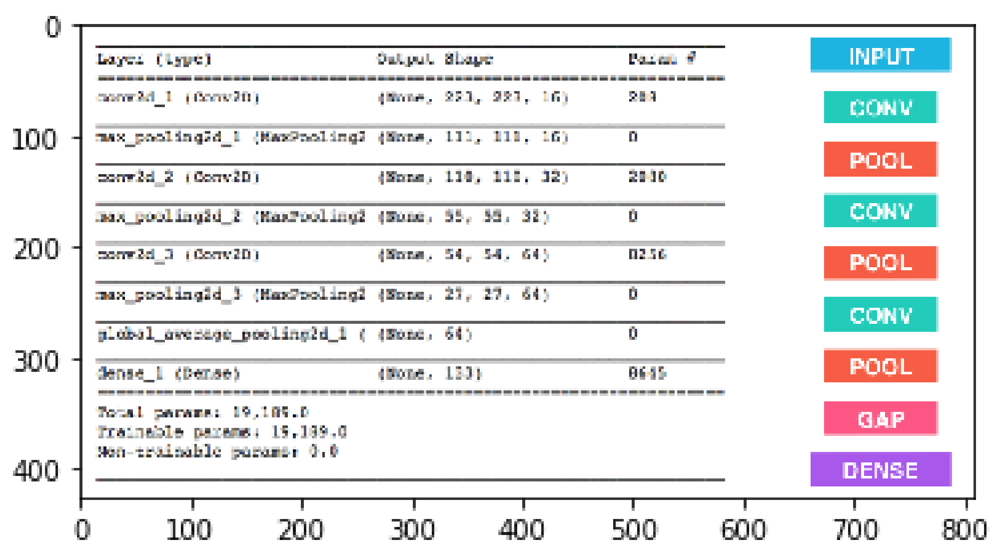
Some sample output for our algorithm is provided below, but feel free to design your own user experience!

1.1.18 (IMPLEMENTATION) Write your Algorithm

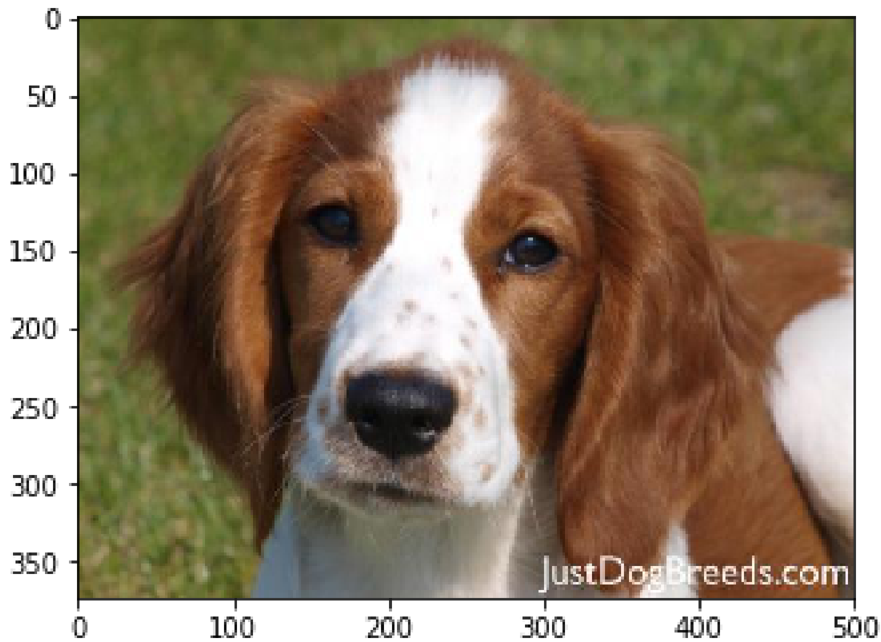
```
In [95]: ### TODO: Write your algorithm.
         ### Feel free to use as many code cells as needed.
         import os
         def run_app(img_path):
             ## handle cases for a human face, dog, and neither

             photo = Image.open(img_path)

             if dog_detector(img_path):
                 plt.imshow(photo)
                 plt.show()
                 guess = predict_doggo(img_path)
                 print('Your code detects that the above photo is a goodest boy :3. Probably a {
             elif face_detector(img_path) > 0:
                 plt.imshow(photo)
                 plt.show()
                 guess = predict_doggo(img_path)
                 print("You are probably a human and not a doggo. You would make a nice {}.".for
             else:
                 plt.imshow(photo)
                 plt.show()
                 print("This does not look like anything to me. Needs more doggo.")
             print()

In [96]: for image_file in os.listdir('./images'):
             img_path = os.path.join('./images', image_file)
```

```
run_app(img_path)
```

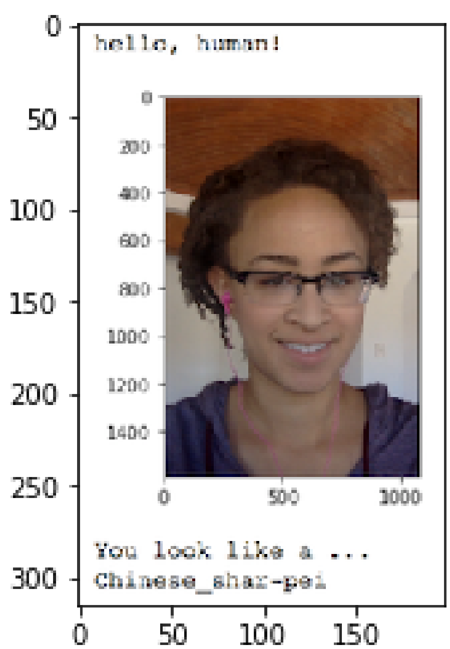
Your code detects that the above photo is a goodest boy :3. Probably a Boykin spaniel.



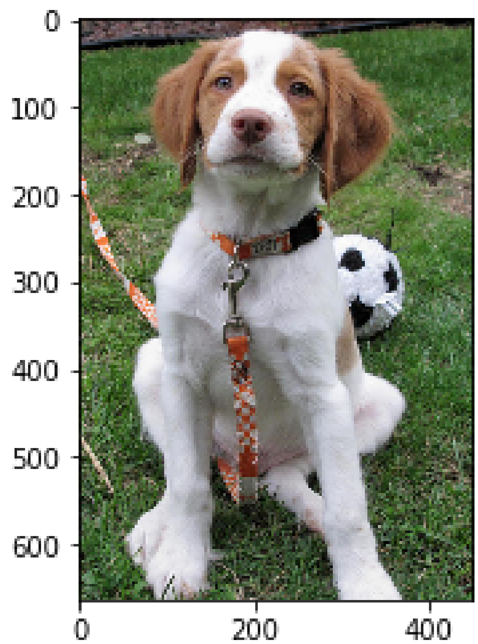
This does not look like anything to me. Needs more doggo.



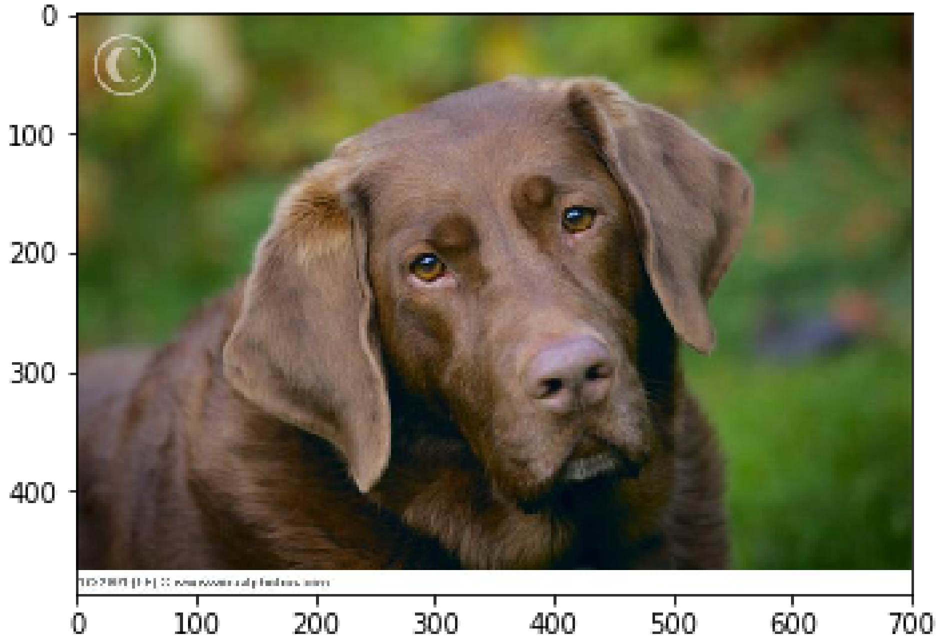
Your code detects that the above photo is a goodest boy :3. Probably a Irish red and white sette



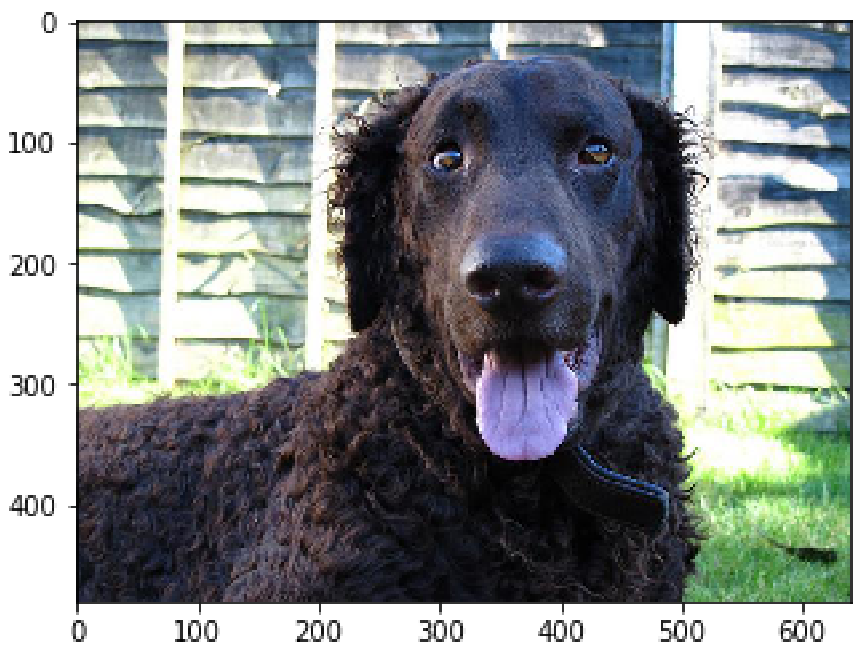
You are probably a human and not a doggo. You would make a nice Dogue de bordeaux.



Your code detects that the above photo is a goodest boy :3. Probably a Brittany.




```
Your code detects that the above photo is a goodest boy :3. Probably a Chesapeake bay retriever.
```



```
Your code detects that the above photo is a goodest boy :3. Probably a Curly-coated retriever.
```

```
---------------------------------------------------------------------------

IsADirectoryError                         Traceback (most recent call last)

<ipython-input-96-301fffbd3eb8> in <module>()
      1 for image_file in os.listdir('./images'):
      2     img_path = os.path.join('./images', image_file)
----> 3     run_app(img_path)


<ipython-input-95-c86bff802271> in run_app(img_path)
      5     ## handle cases for a human face, dog, and neither
      6
----> 7     photo = Image.open(img_path)
      8
      9     if dog_detector(img_path):


/opt/conda/lib/python3.6/site-packages/PIL/Image.py in open(fp, mode)
   2578
   2579     if filename:
-> 2580         fp = builtins.open(filename, "rb")
   2581         exclusive_fp = True
   2582


IsADirectoryError: [Errno 21] Is a directory: './images/.ipynb_checkpoints'
```

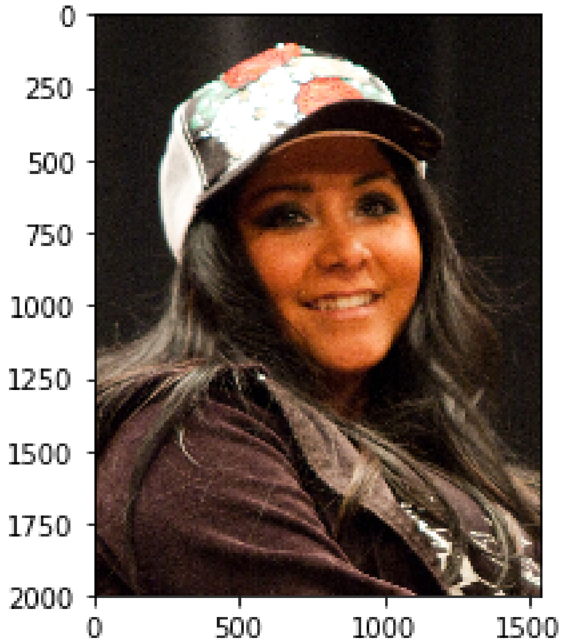
Step 6: Test Your Algorithm
In this section, you will take your new algorithm for a spin! What kind of dog does the algorithm think that *you* look like? If you have a dog, does it predict your dog's breed accurately? If you have a cat, does it mistakenly think that your cat is a dog?

1.1.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

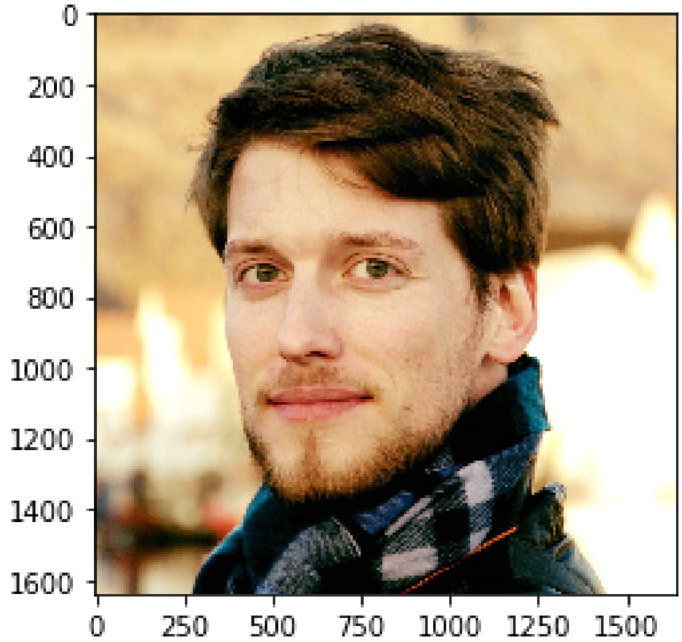
Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

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

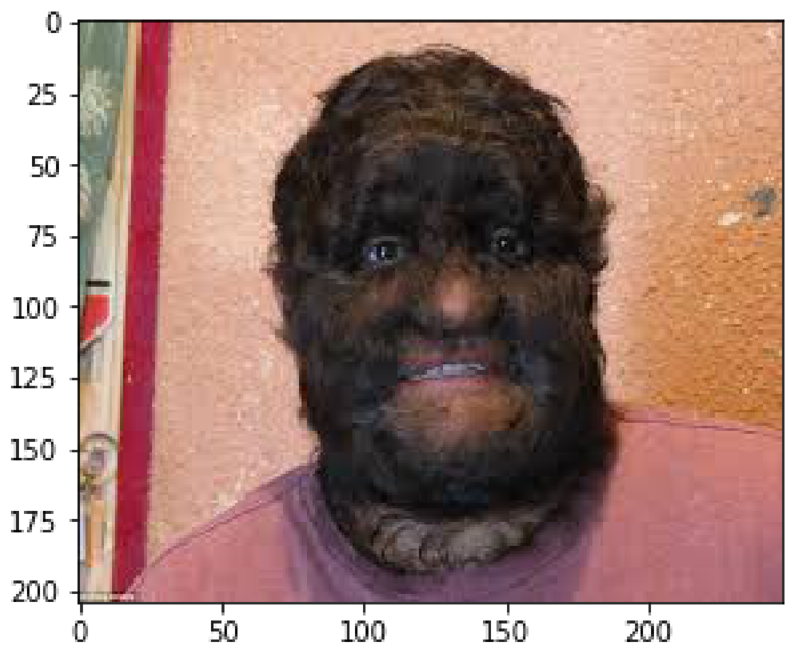
In [97]:
```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.

## suggested code, below

    ## 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.
human_files = ['./Test_images/Snooki.jpg','./Test_images/sleepy_man.jpg','./Test_images

dog_files = ['./Test_images/Shiba.jpg', './Test_images/Chihuahua.jpg', './Test_images/H
## suggested code, below
for file in np.hstack((human_files[:3], dog_files[:3])):
    run_app(file)
```



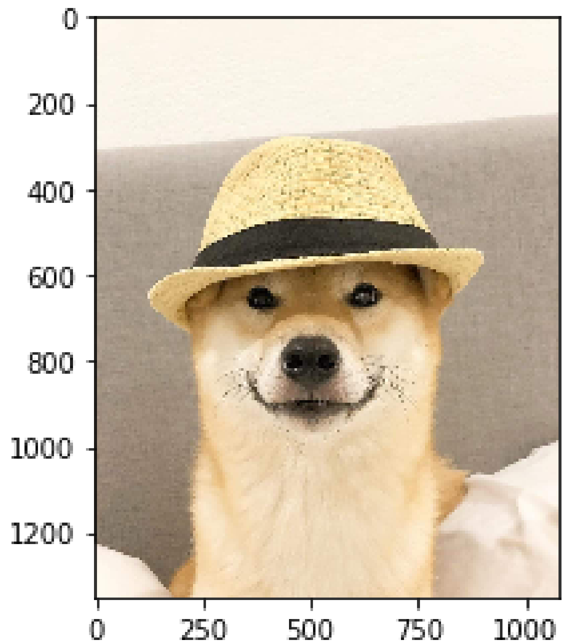

```
You are probably a human and not a doggo. You would make a nice Bearded collie.
```



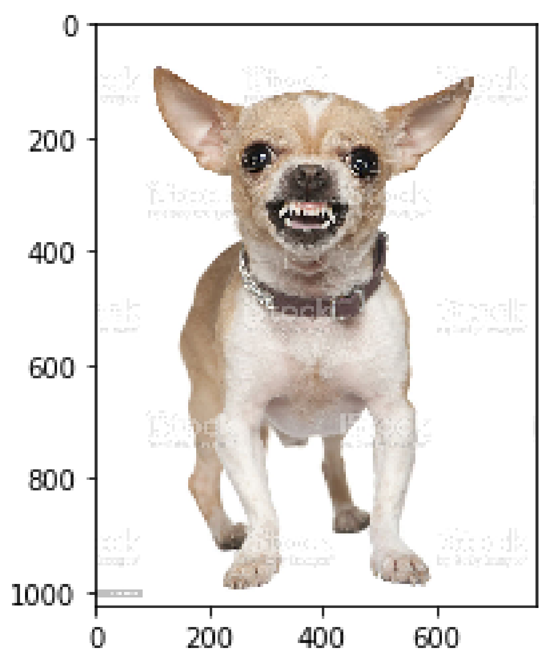
You are probably a human and not a doggo. You would make a nice Norwich terrier.



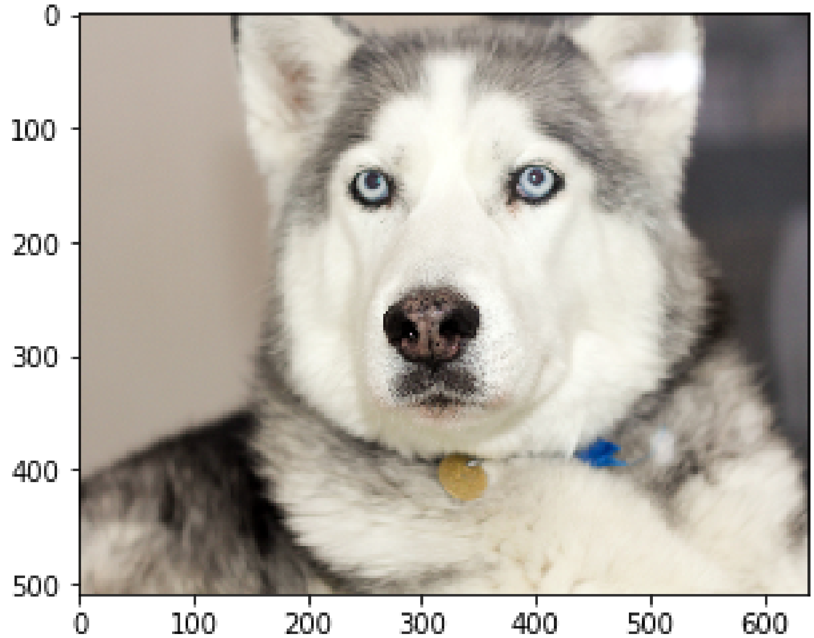
```
This does not look like anything to me. Needs more doggo.
```




```
This does not look like anything to me. Needs more doggo.
```



```
Your code detects that the above photo is a goodest boy :3. Probably a Chihuahua.
```




```
Your code detects that the above photo is a goodest boy :3. Probably a Alaskan malamute.
```

In [ ]:

Answer: (Three possible points for improvement)

The result was not quite as expected. While some breeds were easy to identify, such as Alaskan malamute and Chihuahua, others were easily confused with similar looking breeds. This was especially the case for the Spaniel and Labrador breeds, though my image of the Shiba Inu wearing a hat also remained undetected by the algorithm.

I am quite content with the learning rate since it was quite high but managed to avoid false gradient minima and steadily decreased the validation/training losses.

My main points for improvement would be:

1) Increase accuracy of the prediction by letting the model train for longer, e.g. increase epochs from 30 to 60 or 80.

2) Increase the dataset to obtain convolutional layers with more fine-tuned features. Alternatively, increase the batch size for each epoch. I used only 10 because I'm impatient, though 20-30 would perhaps allows more features to be extracted which could help distinguishing between similar breeds.

3) Introduce more variance in the dataset. I could have tried to apply more transforms when loading the images to train the model harder. However, in order to improve classification in general it would be helpful to supplement the dataset with more atypical images of dogs. For example, complementing the dataset with more images of dogs wearing hats would likely prevent the model from relying on ear shape as an indicative feature. Similarly, using images in which the dogs face slightly away from the camera could improve detection based on facial features which are not readily detectable when the dog is looking at the camera. This could be features such as nose length and head size. It may be necessary to create a separate detection algorithm to this end.

```
In [ ]:
```