dog_app

March 22, 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
        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

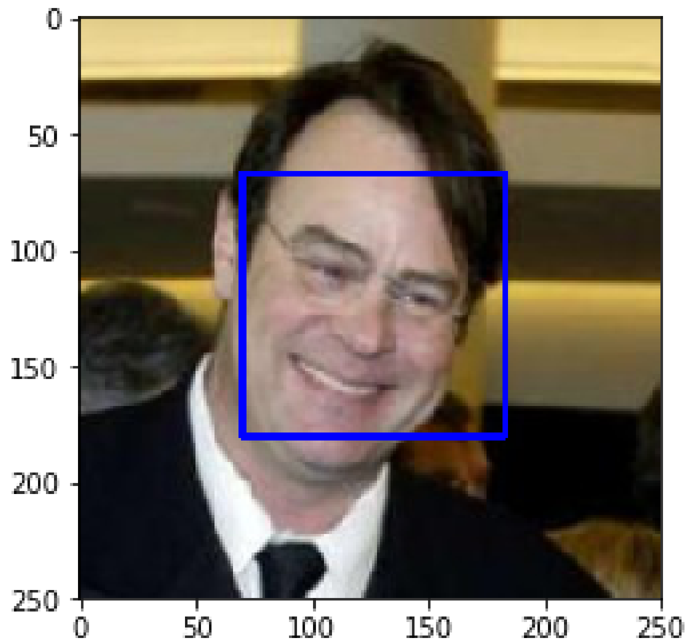
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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])

        # 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
```

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:

```
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.
        all_face = np.vectorize(face_detector)
        human_faces = all_face(human_files_short)
        dog_faces = all_face(dog_files_short)
        print('Percentage from human_files_short with detected human face: {:.2f}%'.format((sum(
        print('Percentage from dog_files_short with detected human face: {:.2f}%'.format((sum(do
```

```
Percentage from human_files_short with detected human face: 98.00%
Percentage from 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 [ ]: ### (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 [5]: import torch
        import torchvision.models as models

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

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

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

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:06<00:00, 89882071.89it/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 [6]: from PIL import Image
        import torchvision.transforms as transforms

        # Get image path
        def image_path(img_path):
            image = Image.open(img_path).convert('RGB')
            trans = transforms.Compose([transforms.Resize(size=(244, 244)), transforms.ToTensor(
            image = trans(image)[:3,:,:].unsqueeze(0)
            return image

        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
    '''

    ## Load and pre-process an image from the given img_path
    ## Return the *index* of the predicted class for that image

    img = image_path(img_path)
    if use_cuda:
        img = img.cuda()
    ret = VGG16(img)

    return torch.max(ret,1)[1].item() # 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 [7]: ### returns "True" if a dog is detected in the image stored at img_path
        def dog_detector(img_path):
            class_id = VGG16_predict(img_path)
            return (class_id in range(151,269))
```

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

        def dog_counter(files):
            cnt = 0;
            for file in files:
                cnt += dog_detector(file)
            return cnt
```

```
print("Percentage from human_files_short with detected dog: {:.2f}%".format(100*dog_coun
print("Percentage from dog_files_short with detected dog: {:.2f}%".format(100*dog_counte
```

```
Percentage from human_files_short with detected dog: 0.00%
Percentage from 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 [ ]: ### (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 [10]:
```
import os
import numpy as np
import torch
import torchvision.transforms as transforms
from torchvision import datasets
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

### Data loaders for training, validation, and test sets

## Specification for appropriate transforms, and batch_sizes

batch_size = 20
num_workers = 0

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/')

# Normalization
standard_normalization = transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0

# Transform - Random Resized Crop
data_transforms = {'train': transforms.Compose([transforms.RandomResizedCrop(224),
                                     transforms.RandomHorizontalFlip(),
                                     transforms.ToTensor(),
                                     standard_normalization]),
                   'val': transforms.Compose([transforms.Resize(256),
                                     transforms.CenterCrop(224),
                                     transforms.ToTensor(),
                                     standard_normalization]),
                   'test': transforms.Compose([transforms.Resize(size=(224,224)),
                                     transforms.ToTensor(),
                                     standard_normalization])
```

```
                    }

    # Data sets
    train_data = datasets.ImageFolder(train_dir, transform=data_transforms['train'])
    valid_data = datasets.ImageFolder(valid_dir, transform=data_transforms['val'])
    test_data = datasets.ImageFolder(test_dir, transform=data_transforms['test'])

    # 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}
```

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

Answer:

- For resizing, RandomResizedCrop was used for training with also normalization on the images via the mean and standard deviation while transforms.Resiz was used for testing. Also, Tensor size of 224x224 was applied on image size to allow proper RandomCrops of the original image.
- Dataset augment was decided using RandomResizedCrop and RandomHorizontalFlip to traine more variations of the dataset and avoid overfitting.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [11]: import torch.nn as nn
         import torch.nn.functional as F
         import numpy as np
         ImageFile.LOAD_TRUNCATED_IMAGES = True

         # define the CNN architecture
         num_classes = 133
         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):
        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 = 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:

- Set Input Image Size to 224x224.
- For Convolution Layers, reduce hte dimensions of (x, y) to (7x7) to inline with well established models.
- Depth was eventually increased to 128 to get as high as 10% accuracy.
- Layer 1 is Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1)) with MaxPool: MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
- Layer 2 is Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1)) with MaxPool: MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
- Layer 3 is Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) with MaxPool: MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)

- Layers are connected fully with ReLu activation and a dropout of 20%
- Layer 1 is (fc1): Linear(in_features=6272, out_features=512, bias=True) while Layer 2 is (fc2): Linear(in_features=512, out_features=133, bias=True)
- Loss is set to CrossEntropyLoss
- Optimizer is SGD (with learning rate = 0.05), and Output Size (num classes) is 133

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 [12]: import torch.optim as optimization

         ### loss function
         criterion_scratch = nn.CrossEntropyLoss()

         ### select optimizer
         optimizer_scratch = optimization.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 [15]: 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_after = 100

             print("Training is now starting! Get some popcorn ;)\n")

             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
```

```
            # initialize weights to zero
            optimizer.zero_grad()

            output = model(data)

            # calculate loss
            loss = criterion(output, target)

            # back prop
            loss.backward()

            # grad
            optimizer.step()

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

            if batch_idx % print_after == 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()

            output = model(data)
            loss = criterion(output, target)
            valid_loss += ((1 / (batch_idx + 1)) * (loss.data - valid_loss))

        print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(epoc

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

    # return trained model
    return model
```

```python
# train the model
trained_epochs = 33
model_file = 'model_scratch.pt'

model_scratch = train(trained_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("Training is now completed! Put the popcorn away ;)")
```

```
Training is now starting! Get some popcorn ;)

        Epoch #1, Iteration #1, Loss: 4.49035120010376
        Epoch #1, Iteration #101, Loss: 4.563215255737305
        Epoch #1, Iteration #201, Loss: 4.570560455322266
        Epoch #1, Iteration #301, Loss: 4.545999526977539
Epoch: 1        Training Loss: 4.545671         Validation Loss: 4.370954
Validation loss decreased (inf --> 4.370954).  Saving model ...


        Epoch #2, Iteration #1, Loss: 4.632518768310547
        Epoch #2, Iteration #101, Loss: 4.476144790649414
        Epoch #2, Iteration #201, Loss: 4.474935054779053
        Epoch #2, Iteration #301, Loss: 4.466214179992676
Epoch: 2        Training Loss: 4.463962         Validation Loss: 4.343661
Validation loss decreased (4.370954 --> 4.343661).  Saving model ...


        Epoch #3, Iteration #1, Loss: 4.192032814025879
        Epoch #3, Iteration #101, Loss: 4.409173488616943
        Epoch #3, Iteration #201, Loss: 4.406735420227051
        Epoch #3, Iteration #301, Loss: 4.409986972808838
Epoch: 3        Training Loss: 4.409357         Validation Loss: 4.202763
Validation loss decreased (4.343661 --> 4.202763).  Saving model ...


        Epoch #4, Iteration #1, Loss: 4.2416486740112305
        Epoch #4, Iteration #101, Loss: 4.352278232574463
        Epoch #4, Iteration #201, Loss: 4.364720821380615
        Epoch #4, Iteration #301, Loss: 4.366661548614502
Epoch: 4        Training Loss: 4.363083         Validation Loss: 4.118946
Validation loss decreased (4.202763 --> 4.118946).  Saving model ...


        Epoch #5, Iteration #1, Loss: 3.754354953765869
        Epoch #5, Iteration #101, Loss: 4.276883125305176
```

```
        Epoch #5, Iteration #201, Loss: 4.277930736541748
        Epoch #5, Iteration #301, Loss: 4.271509647369385
Epoch: 5        Training Loss: 4.271475         Validation Loss: 4.045333
Validation loss decreased (4.118946 --> 4.045333).  Saving model ...


        Epoch #6, Iteration #1, Loss: 4.438803672790527
        Epoch #6, Iteration #101, Loss: 4.242791175842285
        Epoch #6, Iteration #201, Loss: 4.214855670928955
        Epoch #6, Iteration #301, Loss: 4.2111687660217285
Epoch: 6        Training Loss: 4.202993         Validation Loss: 4.026919
Validation loss decreased (4.045333 --> 4.026919).  Saving model ...


        Epoch #7, Iteration #1, Loss: 4.108998775482178
        Epoch #7, Iteration #101, Loss: 4.160248279571533
        Epoch #7, Iteration #201, Loss: 4.16623592376709
        Epoch #7, Iteration #301, Loss: 4.152754306793213
Epoch: 7        Training Loss: 4.148472         Validation Loss: 3.982505
Validation loss decreased (4.026919 --> 3.982505).  Saving model ...


        Epoch #8, Iteration #1, Loss: 4.197998523712158
        Epoch #8, Iteration #101, Loss: 4.085809230804443
        Epoch #8, Iteration #201, Loss: 4.083142280578613
        Epoch #8, Iteration #301, Loss: 4.096522808074951
Epoch: 8        Training Loss: 4.098329         Validation Loss: 3.911072
Validation loss decreased (3.982505 --> 3.911072).  Saving model ...


        Epoch #9, Iteration #1, Loss: 3.7527194023132324
        Epoch #9, Iteration #101, Loss: 4.004271507263184
        Epoch #9, Iteration #201, Loss: 4.042060852050781
        Epoch #9, Iteration #301, Loss: 4.034053325653076
Epoch: 9        Training Loss: 4.027389         Validation Loss: 3.885652
Validation loss decreased (3.911072 --> 3.885652).  Saving model ...


        Epoch #10, Iteration #1, Loss: 3.7918620109558105
        Epoch #10, Iteration #101, Loss: 3.9462153911590576
        Epoch #10, Iteration #201, Loss: 3.9648027420043945
        Epoch #10, Iteration #301, Loss: 3.980725049972534
Epoch: 10        Training Loss: 3.987282         Validation Loss: 4.015882


        Epoch #11, Iteration #1, Loss: 3.7799296379089355
        Epoch #11, Iteration #101, Loss: 3.9229090213775635
        Epoch #11, Iteration #201, Loss: 3.90301775932312
```

```
        Epoch #11, Iteration #301, Loss: 3.917524576187134
Epoch: 11         Training Loss: 3.919889         Validation Loss: 3.751864
Validation loss decreased (3.885652 --> 3.751864).  Saving model ...


        Epoch #12, Iteration #1, Loss: 3.5387673377990723
        Epoch #12, Iteration #101, Loss: 3.8777763843536377
        Epoch #12, Iteration #201, Loss: 3.868990182876587
        Epoch #12, Iteration #301, Loss: 3.864940881729126
Epoch: 12         Training Loss: 3.869295         Validation Loss: 3.697682
Validation loss decreased (3.751864 --> 3.697682).  Saving model ...


        Epoch #13, Iteration #1, Loss: 3.017728567123413
        Epoch #13, Iteration #101, Loss: 3.7979660034179688
        Epoch #13, Iteration #201, Loss: 3.8244755268096924
        Epoch #13, Iteration #301, Loss: 3.817145586013794
Epoch: 13         Training Loss: 3.815827         Validation Loss: 3.673929
Validation loss decreased (3.697682 --> 3.673929).  Saving model ...


        Epoch #14, Iteration #1, Loss: 3.642611265182495
        Epoch #14, Iteration #101, Loss: 3.7233481407165527
        Epoch #14, Iteration #201, Loss: 3.749748945236206
        Epoch #14, Iteration #301, Loss: 3.7545359134674072
Epoch: 14         Training Loss: 3.759728         Validation Loss: 3.639833
Validation loss decreased (3.673929 --> 3.639833).  Saving model ...


        Epoch #15, Iteration #1, Loss: 3.071089506149292
        Epoch #15, Iteration #101, Loss: 3.7255918979644775
        Epoch #15, Iteration #201, Loss: 3.7178494930267334
        Epoch #15, Iteration #301, Loss: 3.720290422439575
Epoch: 15         Training Loss: 3.722321         Validation Loss: 3.574137
Validation loss decreased (3.639833 --> 3.574137).  Saving model ...


        Epoch #16, Iteration #1, Loss: 3.86488676071167
        Epoch #16, Iteration #101, Loss: 3.6759872436523438
        Epoch #16, Iteration #201, Loss: 3.653087854385376
        Epoch #16, Iteration #301, Loss: 3.677933931350708
Epoch: 16         Training Loss: 3.672444         Validation Loss: 3.624585


        Epoch #17, Iteration #1, Loss: 3.286264419555664
        Epoch #17, Iteration #101, Loss: 3.5914478302001953
        Epoch #17, Iteration #201, Loss: 3.5900473594665527
        Epoch #17, Iteration #301, Loss: 3.6135873794555664
```

```
Epoch: 17        Training Loss: 3.610248        Validation Loss: 3.500738
Validation loss decreased (3.574137 --> 3.500738).  Saving model ...


        Epoch #18, Iteration #1, Loss: 3.5148119926452637
        Epoch #18, Iteration #101, Loss: 3.5487215518951416
        Epoch #18, Iteration #201, Loss: 3.551593065261841
        Epoch #18, Iteration #301, Loss: 3.5640525817871094
Epoch: 18        Training Loss: 3.558341        Validation Loss: 3.582408


        Epoch #19, Iteration #1, Loss: 3.136556386947632
        Epoch #19, Iteration #101, Loss: 3.4715702533721924
        Epoch #19, Iteration #201, Loss: 3.5059027671813965
        Epoch #19, Iteration #301, Loss: 3.503375291824341
Epoch: 19        Training Loss: 3.515627        Validation Loss: 3.479233
Validation loss decreased (3.500738 --> 3.479233).  Saving model ...


        Epoch #20, Iteration #1, Loss: 3.757024049758911
        Epoch #20, Iteration #101, Loss: 3.4652233123779297
        Epoch #20, Iteration #201, Loss: 3.4958155155181885
        Epoch #20, Iteration #301, Loss: 3.4950413703918457
Epoch: 20        Training Loss: 3.499774        Validation Loss: 3.489099


        Epoch #21, Iteration #1, Loss: 3.335376262664795
        Epoch #21, Iteration #101, Loss: 3.450730800628662
        Epoch #21, Iteration #201, Loss: 3.4476935863494873
        Epoch #21, Iteration #301, Loss: 3.442173719406128
Epoch: 21        Training Loss: 3.449567        Validation Loss: 3.475384
Validation loss decreased (3.479233 --> 3.475384).  Saving model ...


        Epoch #22, Iteration #1, Loss: 3.5338141918182373
        Epoch #22, Iteration #101, Loss: 3.324195146560669
        Epoch #22, Iteration #201, Loss: 3.38456392288208
        Epoch #22, Iteration #301, Loss: 3.3966166973114014
Epoch: 22        Training Loss: 3.407851        Validation Loss: 3.378596
Validation loss decreased (3.475384 --> 3.378596).  Saving model ...


        Epoch #23, Iteration #1, Loss: 3.4202163219451904
        Epoch #23, Iteration #101, Loss: 3.316668748855591
        Epoch #23, Iteration #201, Loss: 3.3444066047668457
        Epoch #23, Iteration #301, Loss: 3.384953260421753
Epoch: 23        Training Loss: 3.392874        Validation Loss: 3.405930
```

```
        Epoch #24, Iteration #1, Loss: 2.569550037384033
        Epoch #24, Iteration #101, Loss: 3.273745536804199
        Epoch #24, Iteration #201, Loss: 3.3321280479431152
        Epoch #24, Iteration #301, Loss: 3.3390440940856934
Epoch: 24         Training Loss: 3.341266         Validation Loss: 3.423422


        Epoch #25, Iteration #1, Loss: 3.320918560028076
        Epoch #25, Iteration #101, Loss: 3.267322540283203
        Epoch #25, Iteration #201, Loss: 3.3025524616241455
        Epoch #25, Iteration #301, Loss: 3.299772262573242
Epoch: 25         Training Loss: 3.302087         Validation Loss: 3.496745


        Epoch #26, Iteration #1, Loss: 2.8111181259155273
        Epoch #26, Iteration #101, Loss: 3.241466760635376
        Epoch #26, Iteration #201, Loss: 3.227729082107544
        Epoch #26, Iteration #301, Loss: 3.2433764934539795
Epoch: 26         Training Loss: 3.259964         Validation Loss: 3.417350


        Epoch #27, Iteration #1, Loss: 3.330902099609375
        Epoch #27, Iteration #101, Loss: 3.1920969486236572
        Epoch #27, Iteration #201, Loss: 3.1992762088775635
        Epoch #27, Iteration #301, Loss: 3.205310821533203
Epoch: 27         Training Loss: 3.209116         Validation Loss: 3.480667


        Epoch #28, Iteration #1, Loss: 2.362097978591919
        Epoch #28, Iteration #101, Loss: 3.160122871398926
        Epoch #28, Iteration #201, Loss: 3.1938650608062744
        Epoch #28, Iteration #301, Loss: 3.206001043319702
Epoch: 28         Training Loss: 3.205211         Validation Loss: 3.284253
Validation loss decreased (3.378596 --> 3.284253).  Saving model ...


        Epoch #29, Iteration #1, Loss: 2.4436161518096924
        Epoch #29, Iteration #101, Loss: 3.122122287750244
        Epoch #29, Iteration #201, Loss: 3.157163619995117
        Epoch #29, Iteration #301, Loss: 3.149078130722046
Epoch: 29         Training Loss: 3.149392         Validation Loss: 3.329807


        Epoch #30, Iteration #1, Loss: 1.9952293634414673
        Epoch #30, Iteration #101, Loss: 3.0769922733306885
        Epoch #30, Iteration #201, Loss: 3.09073805809021
        Epoch #30, Iteration #301, Loss: 3.1077113151550293
```

```
Epoch: 30 	Training Loss: 3.116377 	Validation Loss: 3.338471


	Epoch #31, Iteration #1, Loss: 2.6330947875976562
	Epoch #31, Iteration #101, Loss: 3.0712692737579346
	Epoch #31, Iteration #201, Loss: 3.0889105796813965
	Epoch #31, Iteration #301, Loss: 3.0796284675598145
Epoch: 31 	Training Loss: 3.098001 	Validation Loss: 3.294354


	Epoch #32, Iteration #1, Loss: 3.588583469390869
	Epoch #32, Iteration #101, Loss: 3.0308022499084473
	Epoch #32, Iteration #201, Loss: 3.053354024887085
	Epoch #32, Iteration #301, Loss: 3.0491809844970703
Epoch: 32 	Training Loss: 3.050368 	Validation Loss: 3.410954


	Epoch #33, Iteration #1, Loss: 2.7967026233673096
	Epoch #33, Iteration #101, Loss: 2.9777023792266846
	Epoch #33, Iteration #201, Loss: 2.982584238052368
	Epoch #33, Iteration #301, Loss: 3.013981819152832
Epoch: 33 	Training Loss: 3.019012 	Validation Loss: 3.281077
Validation loss decreased (3.284253 --> 3.281077).  Saving model ...


Training is now completed! Put the popcorn away ;)
```

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 [16]:
```python
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)

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

```
Test Loss: 3.489491


Test Accuracy: 18% (154/836)
```

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]: # Specify data loaders
         loaders_transfer = 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 [18]: import torchvision.models as models
         import torch.nn as nn
         #import torch.optim as optim
```

```
# Specify model architecture
model_transfer = models.resnet50(pretrained=True)

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

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

for p in fc_parameters:
    p.requires_grad = True

print (model_transfer)

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

```
Downloading: "https://download.pytorch.org/models/resnet50-19c8e357.pth" to /root/.torch/models/
100%|| 102502400/102502400 [00:01<00:00, 78738428.06it/s]


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=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (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=True)
      (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=True)
      )
    )
    (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=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (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=True)
      (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=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (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=True)
      (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=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (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=True)
      (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=True)
      )
    )
    (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=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (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=True)
      (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=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (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=True)
      (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=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

```
      (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=True)
      (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=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (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_stats=True)
      (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_stats=True)
      )
    )
    (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=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (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_stats=True)
      (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=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (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_stats=True)
      (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=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (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_stats=True)
      (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=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (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_stats=True)
      (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=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (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_stats=True)
      (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=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (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_stats=True)
      (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_stats=True)
      )
    )
    (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=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (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_stats=True)
      (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=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (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_stats=True)
      (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: The Resnet model was chosen for the sake of classification as a poper candidate for image classifcation.

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()
         optimizer_transfer = optimization.SGD(model_transfer.fc.parameters(), lr=0.001)
```

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]: def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
             """returns trained model"""

             valid_loss_min = np.Inf # initialization
             print_after = 100

             print("Training is now starting! Where is your popcorn :}\n")


             for epoch in range(1, n_epochs+1):
                 train_loss = 0.0
                 valid_loss = 0.0

                 ###################
                 # train the model #
                 ###################
                 model.train()

                 for batch_idx, (data, target) in enumerate(loaders['train']):

                     if use_cuda:
                         data, target = data.cuda(), target.cuda()

                     optimizer.zero_grad()
```

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

            if batch_idx % print_after == 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']):

            if use_cuda:
                data, target = data.cuda(), target.cuda()

            output = model(data)
            loss = criterion(output, target)
            valid_loss += ((1 / (batch_idx + 1)) * (loss.data - valid_loss))

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

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

    return model


# train the model
trained_epochs = 33
model_file = 'model_transfer.pt'

train(trained_epochs, loaders_transfer, model_transfer, optimizer_transfer, criterion_t

print("Training is now completed! Break is over!")
```

```
Training is now starting! Where is your popcorn :}
```

```
        Epoch #1, Iteration #1, Loss: 4.871053218841553
        Epoch #1, Iteration #101, Loss: 4.906985759735107
        Epoch #1, Iteration #201, Loss: 4.86801815032959
        Epoch #1, Iteration #301, Loss: 4.8347578048706055
Epoch: 1        Training Loss: 4.822406         Validation Loss: 4.641018
Validation loss decreased (inf --> 4.641018).  Saving model ...


        Epoch #2, Iteration #1, Loss: 4.727107048034668
        Epoch #2, Iteration #101, Loss: 4.664030075073242
        Epoch #2, Iteration #201, Loss: 4.637318134307861
        Epoch #2, Iteration #301, Loss: 4.606761932373047
Epoch: 2        Training Loss: 4.596100         Validation Loss: 4.392936
Validation loss decreased (4.641018 --> 4.392936).  Saving model ...


        Epoch #3, Iteration #1, Loss: 4.631146430969238
        Epoch #3, Iteration #101, Loss: 4.442633628845215
        Epoch #3, Iteration #201, Loss: 4.42738151550293
        Epoch #3, Iteration #301, Loss: 4.404378890991211
Epoch: 3        Training Loss: 4.398647         Validation Loss: 4.143116
Validation loss decreased (4.392936 --> 4.143116).  Saving model ...


        Epoch #4, Iteration #1, Loss: 4.254113674163818
        Epoch #4, Iteration #101, Loss: 4.271188259124756
        Epoch #4, Iteration #201, Loss: 4.251242160797119
        Epoch #4, Iteration #301, Loss: 4.222027778625488
Epoch: 4        Training Loss: 4.212747         Validation Loss: 3.908545
Validation loss decreased (4.143116 --> 3.908545).  Saving model ...


        Epoch #5, Iteration #1, Loss: 4.1742753982543945
        Epoch #5, Iteration #101, Loss: 4.111791133880615
        Epoch #5, Iteration #201, Loss: 4.079235076904297
        Epoch #5, Iteration #301, Loss: 4.048720836639404
Epoch: 5        Training Loss: 4.040592         Validation Loss: 3.701074
Validation loss decreased (3.908545 --> 3.701074).  Saving model ...


        Epoch #6, Iteration #1, Loss: 4.139285564422607
        Epoch #6, Iteration #101, Loss: 3.9245238304138184
        Epoch #6, Iteration #201, Loss: 3.9014439582824707
        Epoch #6, Iteration #301, Loss: 3.8797218799591064
Epoch: 6        Training Loss: 3.872854         Validation Loss: 3.510882
Validation loss decreased (3.701074 --> 3.510882).  Saving model ...
```

```
        Epoch #7, Iteration #1, Loss: 3.7922465801239014
        Epoch #7, Iteration #101, Loss: 3.7299368381500244
        Epoch #7, Iteration #201, Loss: 3.731947898864746
        Epoch #7, Iteration #301, Loss: 3.716610908508301
Epoch: 7        Training Loss: 3.707788         Validation Loss: 3.314846
Validation loss decreased (3.510882 --> 3.314846).  Saving model ...


        Epoch #8, Iteration #1, Loss: 3.778322219848633
        Epoch #8, Iteration #101, Loss: 3.6130785942077637
        Epoch #8, Iteration #201, Loss: 3.5760936737060547
        Epoch #8, Iteration #301, Loss: 3.564549684524536
Epoch: 8        Training Loss: 3.554457         Validation Loss: 3.132590
Validation loss decreased (3.314846 --> 3.132590).  Saving model ...


        Epoch #9, Iteration #1, Loss: 3.547138214111328
        Epoch #9, Iteration #101, Loss: 3.457639455795288
        Epoch #9, Iteration #201, Loss: 3.4368507862091064
        Epoch #9, Iteration #301, Loss: 3.417843818664551
Epoch: 9        Training Loss: 3.418759         Validation Loss: 2.957952
Validation loss decreased (3.132590 --> 2.957952).  Saving model ...


        Epoch #10, Iteration #1, Loss: 3.3525230884552
        Epoch #10, Iteration #101, Loss: 3.3215994834899902
        Epoch #10, Iteration #201, Loss: 3.308624505996704
        Epoch #10, Iteration #301, Loss: 3.2894132137298584
Epoch: 10        Training Loss: 3.284708         Validation Loss: 2.781582
Validation loss decreased (2.957952 --> 2.781582).  Saving model ...


        Epoch #11, Iteration #1, Loss: 3.0038931369781494
        Epoch #11, Iteration #101, Loss: 3.223510503768921
        Epoch #11, Iteration #201, Loss: 3.1848623752593994
        Epoch #11, Iteration #301, Loss: 3.1724538803100586
Epoch: 11        Training Loss: 3.166575         Validation Loss: 2.639552
Validation loss decreased (2.781582 --> 2.639552).  Saving model ...


        Epoch #12, Iteration #1, Loss: 2.9938621520996094
        Epoch #12, Iteration #101, Loss: 3.0716328620910645
        Epoch #12, Iteration #201, Loss: 3.0488555431365967
        Epoch #12, Iteration #301, Loss: 3.0349502563476562
Epoch: 12        Training Loss: 3.035010         Validation Loss: 2.518097
Validation loss decreased (2.639552 --> 2.518097).  Saving model ...
```

```
        Epoch #13, Iteration #1, Loss: 3.085958242416382
        Epoch #13, Iteration #101, Loss: 2.9398319721221924
        Epoch #13, Iteration #201, Loss: 2.9269754886627197
        Epoch #13, Iteration #301, Loss: 2.9255728721618652
Epoch: 13        Training Loss: 2.925945        Validation Loss: 2.404809
Validation loss decreased (2.518097 --> 2.404809).  Saving model ...


        Epoch #14, Iteration #1, Loss: 2.5344018936157227
        Epoch #14, Iteration #101, Loss: 2.83162522315979
        Epoch #14, Iteration #201, Loss: 2.8394277095794678
        Epoch #14, Iteration #301, Loss: 2.833806037902832
Epoch: 14        Training Loss: 2.832696        Validation Loss: 2.282537
Validation loss decreased (2.404809 --> 2.282537).  Saving model ...


        Epoch #15, Iteration #1, Loss: 2.778444290161133
        Epoch #15, Iteration #101, Loss: 2.713353395462036
        Epoch #15, Iteration #201, Loss: 2.742091655731201
        Epoch #15, Iteration #301, Loss: 2.726854085922241
Epoch: 15        Training Loss: 2.732027        Validation Loss: 2.167162
Validation loss decreased (2.282537 --> 2.167162).  Saving model ...


        Epoch #16, Iteration #1, Loss: 2.789745569229126
        Epoch #16, Iteration #101, Loss: 2.6605262756347656
        Epoch #16, Iteration #201, Loss: 2.6723146438598633
        Epoch #16, Iteration #301, Loss: 2.66235613822937
Epoch: 16        Training Loss: 2.656143        Validation Loss: 2.088530
Validation loss decreased (2.167162 --> 2.088530).  Saving model ...


        Epoch #17, Iteration #1, Loss: 2.668334484100342
        Epoch #17, Iteration #101, Loss: 2.5816774368286133
        Epoch #17, Iteration #201, Loss: 2.566585063934326
        Epoch #17, Iteration #301, Loss: 2.5545690059661865
Epoch: 17        Training Loss: 2.554657        Validation Loss: 1.977954
Validation loss decreased (2.088530 --> 1.977954).  Saving model ...


        Epoch #18, Iteration #1, Loss: 2.268298625946045
        Epoch #18, Iteration #101, Loss: 2.5113885402679443
        Epoch #18, Iteration #201, Loss: 2.506760358810425
        Epoch #18, Iteration #301, Loss: 2.48972749710083
Epoch: 18        Training Loss: 2.489397        Validation Loss: 1.886684
Validation loss decreased (1.977954 --> 1.886684).  Saving model ...
```

```
        Epoch #19, Iteration #1, Loss: 2.3887739181518555
        Epoch #19, Iteration #101, Loss: 2.441436529159546
        Epoch #19, Iteration #201, Loss: 2.429358720779419
        Epoch #19, Iteration #301, Loss: 2.4147322177886963
Epoch: 19        Training Loss: 2.410990        Validation Loss: 1.839738
Validation loss decreased (1.886684 --> 1.839738).  Saving model ...


        Epoch #20, Iteration #1, Loss: 2.40118145942688
        Epoch #20, Iteration #101, Loss: 2.367600440979004
        Epoch #20, Iteration #201, Loss: 2.369434356689453
        Epoch #20, Iteration #301, Loss: 2.3599424362182617
Epoch: 20        Training Loss: 2.355678        Validation Loss: 1.743631
Validation loss decreased (1.839738 --> 1.743631).  Saving model ...


        Epoch #21, Iteration #1, Loss: 2.154035806655884
        Epoch #21, Iteration #101, Loss: 2.303112030029297
        Epoch #21, Iteration #201, Loss: 2.2751400470733643
        Epoch #21, Iteration #301, Loss: 2.2692527770996094
Epoch: 21        Training Loss: 2.271598        Validation Loss: 1.691054
Validation loss decreased (1.743631 --> 1.691054).  Saving model ...


        Epoch #22, Iteration #1, Loss: 2.380399465560913
        Epoch #22, Iteration #101, Loss: 2.24202561378479
        Epoch #22, Iteration #201, Loss: 2.215045928955078
        Epoch #22, Iteration #301, Loss: 2.2346031665802
Epoch: 22        Training Loss: 2.217704        Validation Loss: 1.633183
Validation loss decreased (1.691054 --> 1.633183).  Saving model ...


        Epoch #23, Iteration #1, Loss: 2.246616840362549
        Epoch #23, Iteration #101, Loss: 2.166900873184204
        Epoch #23, Iteration #201, Loss: 2.159238576889038
        Epoch #23, Iteration #301, Loss: 2.168975353240967
Epoch: 23        Training Loss: 2.164917        Validation Loss: 1.560359
Validation loss decreased (1.633183 --> 1.560359).  Saving model ...


        Epoch #24, Iteration #1, Loss: 2.1551430225372314
        Epoch #24, Iteration #101, Loss: 2.0878403186798096
        Epoch #24, Iteration #201, Loss: 2.1098127365112305
        Epoch #24, Iteration #301, Loss: 2.0934946537017822
Epoch: 24        Training Loss: 2.091671        Validation Loss: 1.502749
Validation loss decreased (1.560359 --> 1.502749).  Saving model ...
```

```
        Epoch #25, Iteration #1, Loss: 2.0316720008850098
        Epoch #25, Iteration #101, Loss: 2.043164014816284
        Epoch #25, Iteration #201, Loss: 2.0630781650543213
        Epoch #25, Iteration #301, Loss: 2.063009738922119
Epoch: 25         Training Loss: 2.057938         Validation Loss: 1.465889
Validation loss decreased (1.502749 --> 1.465889).  Saving model ...


        Epoch #26, Iteration #1, Loss: 2.0052175521850586
        Epoch #26, Iteration #101, Loss: 2.027733325958252
        Epoch #26, Iteration #201, Loss: 2.0338637828826904
        Epoch #26, Iteration #301, Loss: 2.0387582778930664
Epoch: 26         Training Loss: 2.033662         Validation Loss: 1.407057
Validation loss decreased (1.465889 --> 1.407057).  Saving model ...


        Epoch #27, Iteration #1, Loss: 1.824806571006775
        Epoch #27, Iteration #101, Loss: 2.0029516220092773
        Epoch #27, Iteration #201, Loss: 1.9999046325683594
        Epoch #27, Iteration #301, Loss: 1.983224630355835
Epoch: 27         Training Loss: 1.981071         Validation Loss: 1.370805
Validation loss decreased (1.407057 --> 1.370805).  Saving model ...


        Epoch #28, Iteration #1, Loss: 2.419095516204834
        Epoch #28, Iteration #101, Loss: 1.9368305206298828
        Epoch #28, Iteration #201, Loss: 1.938778042793274
        Epoch #28, Iteration #301, Loss: 1.949415683746338
Epoch: 28         Training Loss: 1.942551         Validation Loss: 1.322099
Validation loss decreased (1.370805 --> 1.322099).  Saving model ...


        Epoch #29, Iteration #1, Loss: 2.0739150047302246
        Epoch #29, Iteration #101, Loss: 1.8892093896865845
        Epoch #29, Iteration #201, Loss: 1.879821538925171
        Epoch #29, Iteration #301, Loss: 1.8724563121795654
Epoch: 29         Training Loss: 1.877446         Validation Loss: 1.279112
Validation loss decreased (1.322099 --> 1.279112).  Saving model ...


        Epoch #30, Iteration #1, Loss: 2.053122043609619
        Epoch #30, Iteration #101, Loss: 1.8621724843978882
        Epoch #30, Iteration #201, Loss: 1.8656651973724365
        Epoch #30, Iteration #301, Loss: 1.8602133989334106
Epoch: 30         Training Loss: 1.863457         Validation Loss: 1.239220
Validation loss decreased (1.279112 --> 1.239220).  Saving model ...
```

```
        Epoch #31, Iteration #1, Loss: 2.3706531524658203
        Epoch #31, Iteration #101, Loss: 1.8474818468093872
        Epoch #31, Iteration #201, Loss: 1.8314679861068726
        Epoch #31, Iteration #301, Loss: 1.8264859914779663
Epoch: 31         Training Loss: 1.822910         Validation Loss: 1.226833
Validation loss decreased (1.239220 --> 1.226833).  Saving model ...


        Epoch #32, Iteration #1, Loss: 1.4818055629730225
        Epoch #32, Iteration #101, Loss: 1.7899476289749146
        Epoch #32, Iteration #201, Loss: 1.7933275699615479
        Epoch #32, Iteration #301, Loss: 1.8014802932739258
Epoch: 32         Training Loss: 1.794790         Validation Loss: 1.195498
Validation loss decreased (1.226833 --> 1.195498).  Saving model ...


        Epoch #33, Iteration #1, Loss: 2.1339802742004395
        Epoch #33, Iteration #101, Loss: 1.7751797437667847
        Epoch #33, Iteration #201, Loss: 1.778869390487671
        Epoch #33, Iteration #301, Loss: 1.7815287113189697
Epoch: 33         Training Loss: 1.780451         Validation Loss: 1.145228
Validation loss decreased (1.195498 --> 1.145228).  Saving model ...


Training is now completed! Break is over!
```

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

```
Test Loss: 1.211861


Test Accuracy: 78% (653/836)
```

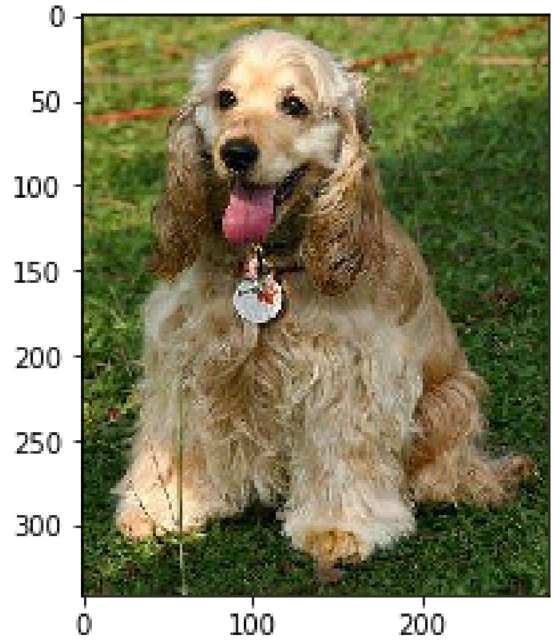
1.1.17 (IMPLEMENTATION) Predict Dog Breed with the Model

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.

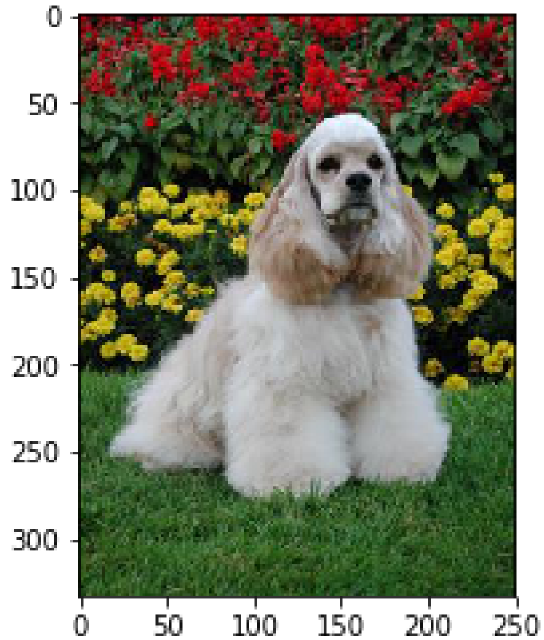
```
In [35]: #from PIL import Image
         #import torchvision.transforms as transforms
```

```
#import cv2
#import matplotlib.pyplot as plt
#%matplotlib inline

class_names = [item[4:].replace("_", " ") for item in loaders_transfer['train'].dataset
loaders_transfer['train'].dataset.classes[:10]
class_names[:10]

def get_image(img_path):
    img = Image.open(img_path).convert('RGB')
    prediction_transform = transforms.Compose([transforms.Resize(size=(224, 224)),trans
    img = prediction_transform(img)[:3,:,:].unsqueeze(0)
    return img


def predict_breed_transfer(model, class_names, img_path):
    img = get_image(img_path)
    pred = torch.argmax(model(img))
    return class_names[pred]
```

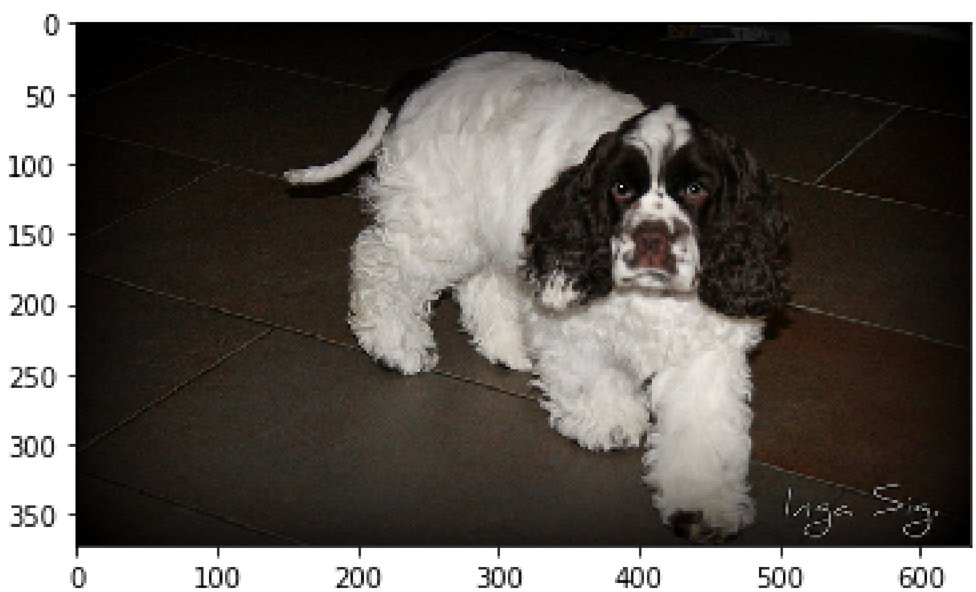
In [36]:
```
# test
for img_file in os.listdir('/data/dog_images/test/053.Cocker_spaniel/'):
    img = cv2.imread(img_path)
    cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    plt.imshow(cv_rgb)
    plt.show()
    img_path = os.path.join('/data/dog_images/test/053.Cocker_spaniel/', img_file)
    predition = predict_breed_transfer(model_transfer, class_names, img_path)
    print("predition breed: {}".format(predition))
```



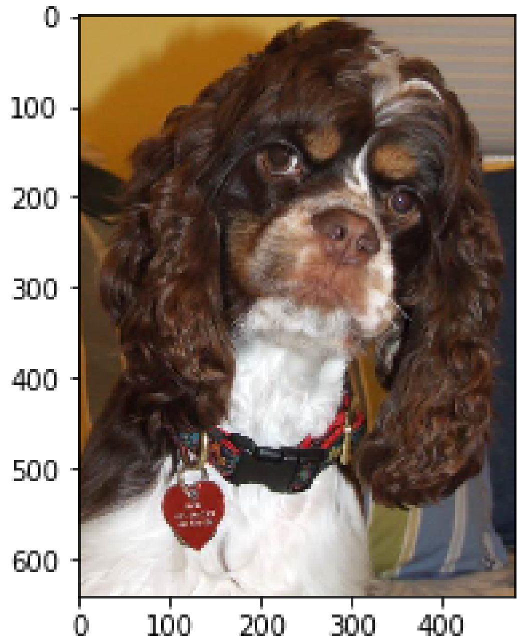
prediton breed: Cocker spaniel



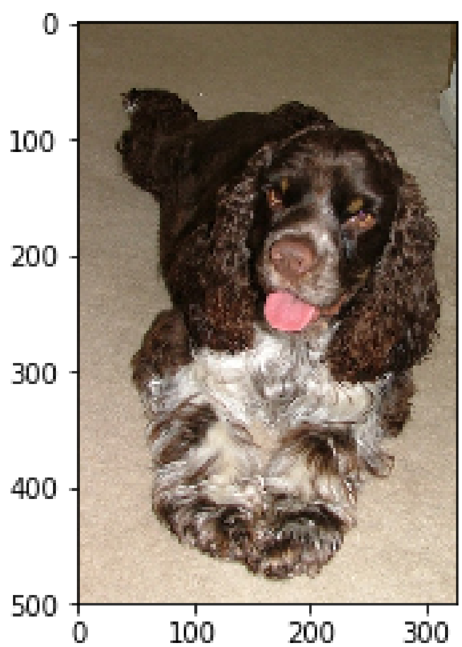
prediton breed: Cocker spaniel



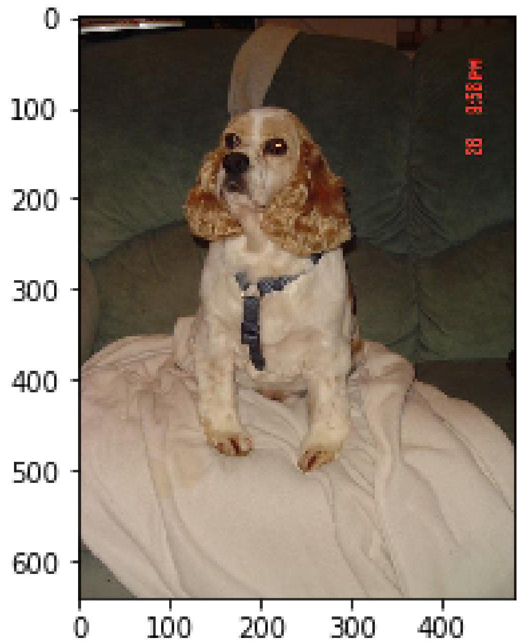
```
predition breed: Cavalier king charles spaniel
```




```
predition breed: English cocker spaniel
```



```
predition breed: English cocker spaniel
```



```
predition breed: English cocker spaniel
```

Step 5: Write your Algorithm

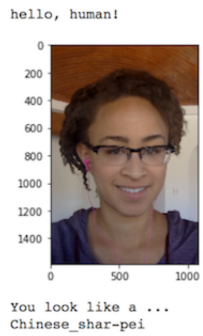
Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then, - if a **dog** is detected in the image, return the predicted breed. - if a **human** is detected in the image, return the resembling dog breed. - if **neither** is 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 [46]: def dog_app(img_path):
             img = Image.open(img_path)
```



Sample Human Output

```
    plt.imshow(img)
    plt.show()
    if dog_detector(img_path) is True:
        #prediction = predict_breed_transfer(model_transfer, class_names, img_path)
        print("This dog looks like a {0}".format(predict_breed_transfer(model_transfer,
    elif face_detector(img_path) > 0:
        prediction = predict_breed_transfer(model_transfer, class_names, img_path)
        print("This human looks like the dog breed {0}".format(prediction))
    else:
        print("Neither dog nor human was detected by the model")
```

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?

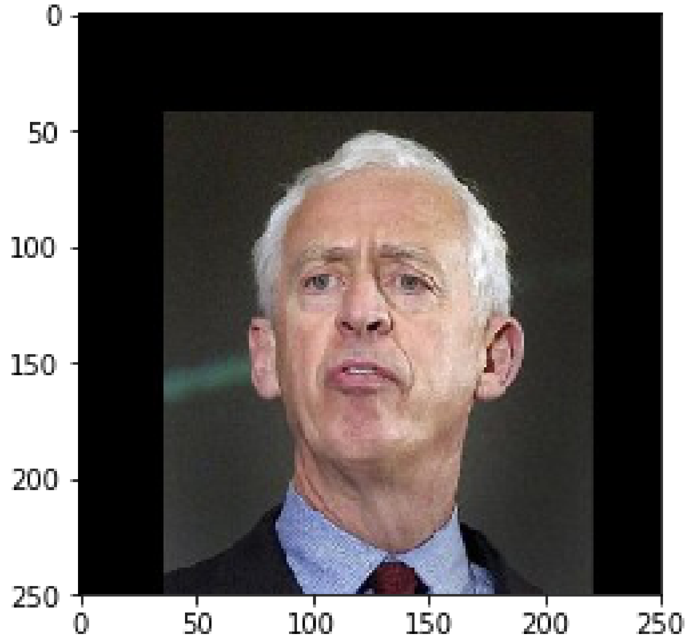
1.1.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

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.

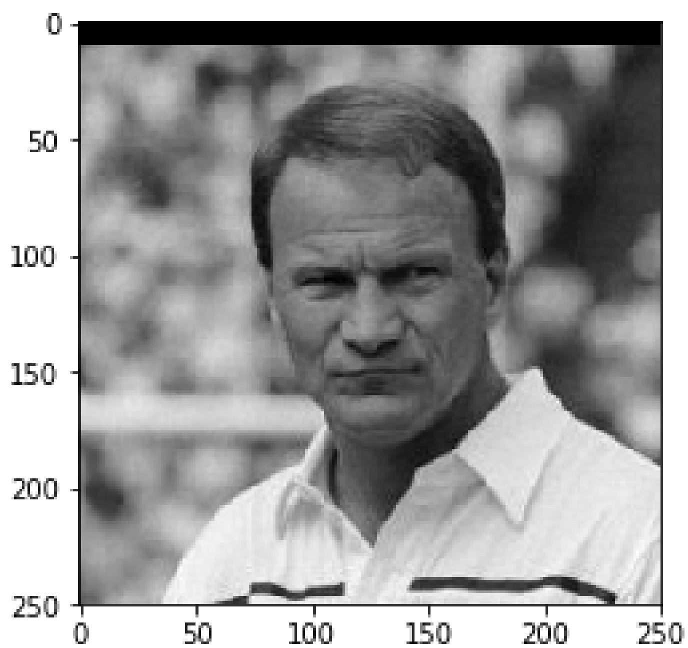
Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer: (Three possible points for improvement): more image for train the model, incrase the overall performance of the model, and parameterize the model more.

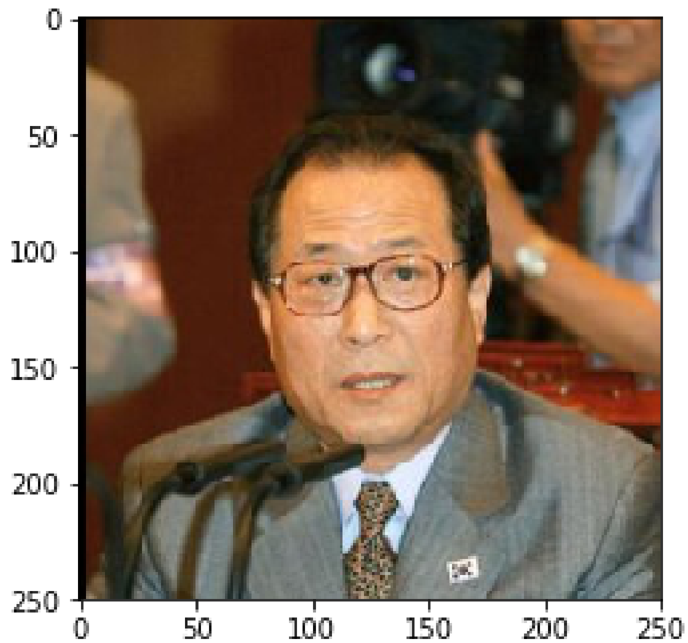
```
In [60]: num_human_img = 3
         num_dog_img = 4
         index_slice = 3
         for file in np.hstack((human_files[index_slice:(num_human_img+index_slice)], dog_files[
             dog_app(file)
```



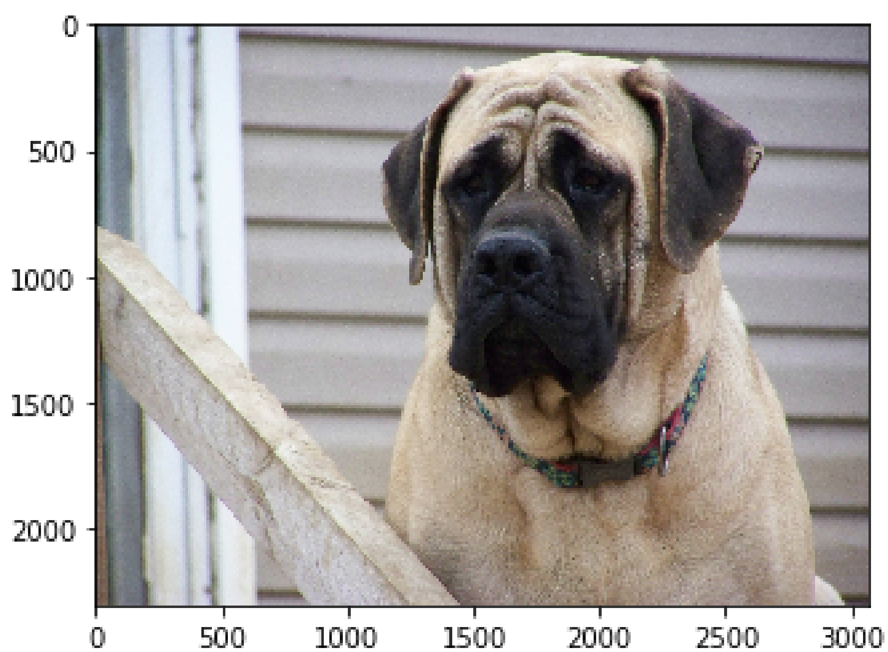
This human looks like the dog breed Poodle



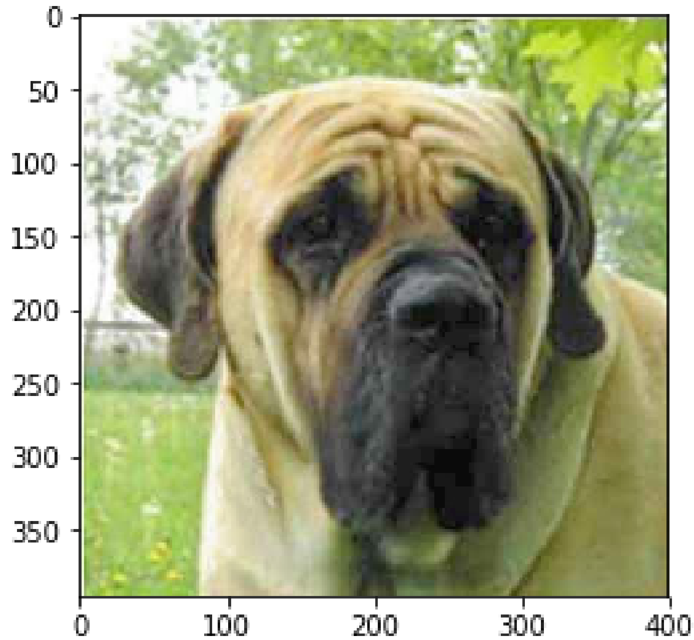
This human looks like the dog breed German pinscher



This human looks like the dog breed Dachshund

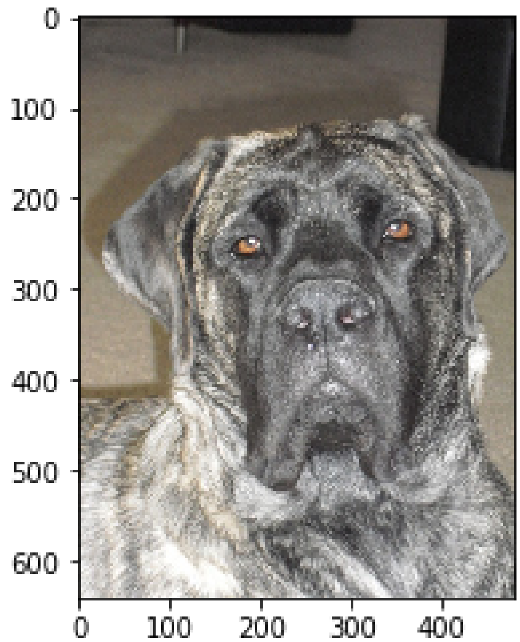


This dog looks like a Mastiff



This dog looks like a Mastiff

This dog looks like a Mastiff



This dog looks like a Mastiff

```
In [ ]:
```