dog_app

May 2, 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 [1]: import numpy as np
        from glob import glob

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

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

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

## Step 1: Detect Humans

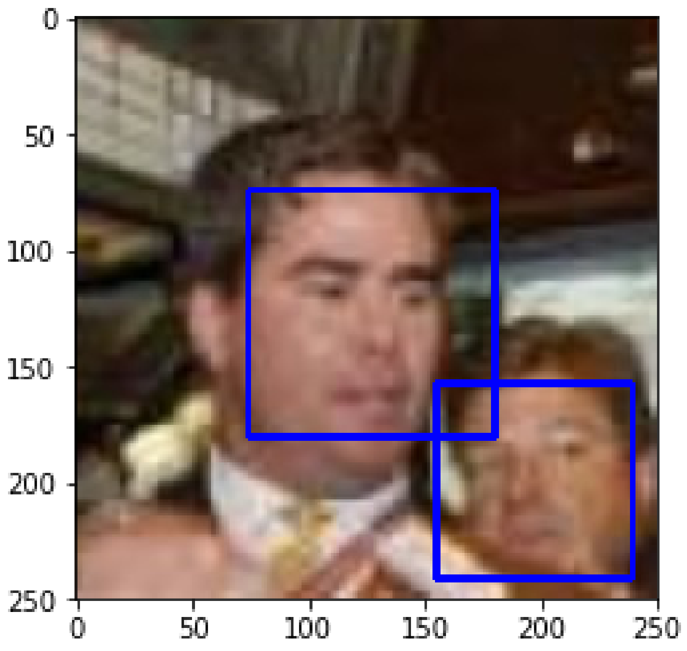
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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

```
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: 2
```



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]: 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 [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.

        def face_detection_test(files):
            detection_cnt = 0;
            total_cnt = len(files)
            for file in files:
                detection_cnt += face_detector(file)
            return detection_cnt, total_cnt
```

```
In [5]: print("detect face in human_files: {} / {}".format(face_detection_test(human_files_short
        print("detect face in dog_files: {} / {}".format(face_detection_test(dog_files_short)[0]
```

```
detect face in human_files: 99 / 100
detect face in dog_files: 14 / 100
```

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 [6]: ### (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 [7]: import torch
        import torchvision.models as models

        # check if CUDA is available
        use_cuda = torch.cuda.is_available()
        print("cuda available? {0}".format(use_cuda))
```

```
cuda available? True
```

```
In [8]: # define VGG16 model
        VGG16 = models.vgg16(pretrained=True)

        # 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 [9]: from PIL import Image
        import torchvision.transforms as transforms

        def load_image(img_path):
            image = Image.open(img_path).convert('RGB')
            # resize to (244, 244) because VGG16 accept this shape
            in_transform = transforms.Compose([
                                transforms.Resize(size=(244, 244)),
                                transforms.ToTensor()]) # normalizaiton parameters from pytorch

            # discard the transparent, alpha channel (that's the :3) and add the batch dimension
            image = in_transform(image)[:3,:,:].unsqueeze(0)
            return image
```

```
In [10]: def VGG16_predict(img_path):
             '''
             Use pre-trained VGG-16 model to obtain index corresponding to
             predicted ImageNet class for image at specified path

             Args:
                 img_path: path to an image

             Returns:
                 Index corresponding to VGG-16 model's prediction
             '''

             ## TODO: Complete the function.
             ## Load and pre-process an image from the given img_path
             ## Return the *index* of the predicted class for that image
             img = load_image(img_path)
             if use_cuda:
                 img = img.cuda()
             ret = VGG16(img)
             return torch.max(ret,1)[1].item() # predicted class index
```

```
In [11]: # predict dog using ImageNet class
         VGG16_predict(dog_files_short[0])
```

```
Out[11]: 719
```

1.1.5 (IMPLEMENTATION) Write a Dog Detector

While looking at the dictionary, you will notice that the categories corresponding to dogs appear in an uninterrupted sequence and correspond to dictionary keys 151-268, inclusive, to include all categories from `'Chihuahua'` to `'Mexican hairless'`. Thus, in order to check to see if an image is predicted to contain a dog by the pre-trained VGG-16 model, we need only check if the pre-trained model predicts an index between 151 and 268 (inclusive).

Use these ideas to complete the `dog_detector` function below, which returns `True` if a dog is detected in an image (and `False` if not).

```
In [12]: ### returns "True" if a dog is detected in the image stored at img_path
         def dog_detector(img_path):
             ## TODO: Complete the function.
             idx = VGG16_predict(img_path)
             return idx >= 151 and idx <= 268 # true/false
```

```
In [13]: print(dog_detector(dog_files_short[0]))
         print(dog_detector(human_files_short[0]))
```

```
False
False
```

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 [14]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.
         def dog_detector_test(files):
             detection_cnt = 0;
             total_cnt = len(files)
             for file in files:
                 detection_cnt += dog_detector(file)
             return detection_cnt, total_cnt
```

```
In [15]: print("detect a dog in human_files: {} / {}".format(dog_detector_test(human_files_short
         print("detect a dog in dog_files: {} / {}".format(dog_detector_test(dog_files_short)[0]
```

```
detect a dog in human_files: 0 / 100
detect a dog in dog_files: 89 / 100
```

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 [16]: ### (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!

1.2 Preprocessing

```
In [17]: import os
         from torchvision import datasets
         import torchvision.transforms as transforms
         import torch
         import numpy as np
         from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True
```

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

batch_size = 20
num_workers = 0

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

1.3 User Standard Normalization Value

```
In [18]: standard_normalization = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                       std=[0.229, 0.224, 0.225])
         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])
                           }
```

1.4 Use ImageFolder to Load image_dataset

```
In [19]: 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'])
         train_loader = torch.utils.data.DataLoader(train_data,
                                                    batch_size=batch_size,
                                                    num_workers=num_workers,
                                                    shuffle=True)
         valid_loader = torch.utils.data.DataLoader(valid_data,
                                                    batch_size=batch_size,
                                                    num_workers=num_workers,
                                                    shuffle=False)
         test_loader = torch.utils.data.DataLoader(test_data,
                                                    batch_size=batch_size,
                                                    num_workers=num_workers,
                                                    shuffle=False)

         loaders_scratch = {
             'train': train_loader,
    '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: I've applied RandomResizedCrop & RandomHorizontalFlip to just train_data. This will do both image augmentations and resizing jobs. Image augmentation will give randomness to the dataset so, it prevents overfitting and I can expect better performance of model when it's predicting toward test_data. On the other hand, I've done Resize of (256) and then, center crop to make 224 X 224. Since valid_data will be used for validation check, I will not do image augmentations. For the test_data, I've applied only image resizing.

1.4.1 (IMPLEMENTATION) Model Architecture

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

```
In [20]: from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True

         num_classes = 133 # total classes of dog breeds

In [21]: import torch.nn as nn
         import torch.nn.functional as F
         import numpy as np

         # define the CNN architecture
         class Net(nn.Module):
             ### TODO: choose an architecture, and complete the class
             def __init__(self):
                 super(Net, self).__init__()
                 ## Define layers of a CNN
                 self.conv1 = nn.Conv2d(3, 32, 3, stride=2, padding=1)
                 self.conv2 = nn.Conv2d(32, 64, 3, stride=2, padding=1)
                 self.conv3 = nn.Conv2d(64, 128, 3, padding=1)

                 # pool
                 self.pool = nn.MaxPool2d(2, 2)

                 # fully-connected
                 self.fc1 = nn.Linear(7*7*128, 500)
                 self.fc2 = nn.Linear(500, num_classes)

                 # drop-out
                 self.dropout = nn.Dropout(0.3)

             def forward(self, x):
                ## Define forward behavior
                x = F.relu(self.conv1(x))
                x = self.pool(x)
                x = F.relu(self.conv2(x))
                x = self.pool(x)
                x = F.relu(self.conv3(x))
                x = self.pool(x)

                # flatten
                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=500, bias=True)
  (fc2): Linear(in_features=500, out_features=133, bias=True)
  (dropout): Dropout(p=0.3)
)
```

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

Answer: (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
activation: relu
(pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
activation: relu
(conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
activation: relu
(pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)

(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)
(dropout): Dropout(p=0.3)
(fc1): Linear(in_features=6272, out_features=500, bias=True)
(dropout): Dropout(p=0.3)
(fc2): Linear(in_features=500, out_features=133, bias=True)

explanations First 2 conv layers I've applied kernel_size of 3 with stride 2, this will lead to downsize of input image by 2. after 2 conv layers, maxpooling with stride 2 is placed and this will lead to downsize of input image by 2. The 3rd conv layers is consist of kernel_size of 3 with stride 1, and this will not reduce input image. after final maxpooling with stride 2, the total output image size is downsized by factor of 32 and the depth will be 128. I've applied dropout of 0.3 in order to prevent overfitting. Fully-connected layer is placed and then, 2nd fully-connected layer is intended to produce final output_size which predicts classes of breeds.

1.4.2 (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 [22]: import torch.optim as optim

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

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

1.4.3 (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 [23]: def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path, last_val
             """returns trained model"""
             # initialize tracker for minimum validation loss
             if last_validation_loss is not None:
                 valid_loss_min = last_validation_loss
             else:
                 valid_loss_min = np.Inf

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

                 ###################
                 # train the model #
                 ###################
```

```
model.train()
for batch_idx, (data, target) in enumerate(loaders['train']):
    # move to GPU
    if use_cuda:
        data, target = data.cuda(), target.cuda()
    ## find the loss and update the model parameters accordingly
    ## 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 = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss)

    if batch_idx % 100 == 0:
        print('Epoch %d, Batch %d loss: %.6f' %
          (epoch, batch_idx + 1, train_loss))

######################
# validate the model #
######################
model.eval()
for batch_idx, (data, target) in enumerate(loaders['valid']):
    # move to GPU
    if use_cuda:
        data, target = data.cuda(), target.cuda()
    ## update the average validation loss
    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 decreased ({:.6f} --> {:.6f}).  Saving model ...'.fo
                    valid_loss_min,
                    valid_loss))
                    valid_loss_min = valid_loss

            # return trained model
            return model
```

```
In [24]: # train the model
         model_scratch = train(20, loaders_scratch, model_scratch, optimizer_scratch,
                               criterion_scratch, use_cuda, 'model_scratch.pt')
```

```
Epoch 1, Batch 1 loss: 4.877383
Epoch 1, Batch 101 loss: 4.887667
Epoch 1, Batch 201 loss: 4.879937
Epoch 1, Batch 301 loss: 4.873069
Epoch: 1         Training Loss: 4.869494         Validation Loss: 4.805907
Validation loss decreased (inf --> 4.805907).  Saving model ...
Epoch 2, Batch 1 loss: 4.804860
Epoch 2, Batch 101 loss: 4.786738
Epoch 2, Batch 201 loss: 4.759734
Epoch 2, Batch 301 loss: 4.737430
Epoch: 2         Training Loss: 4.733724         Validation Loss: 4.563401
Validation loss decreased (4.805907 --> 4.563401).  Saving model ...
Epoch 3, Batch 1 loss: 4.694051
Epoch 3, Batch 101 loss: 4.609666
Epoch 3, Batch 201 loss: 4.620006
Epoch 3, Batch 301 loss: 4.613776
Epoch: 3         Training Loss: 4.612876         Validation Loss: 4.439207
Validation loss decreased (4.563401 --> 4.439207).  Saving model ...
Epoch 4, Batch 1 loss: 4.632480
Epoch 4, Batch 101 loss: 4.552093
Epoch 4, Batch 201 loss: 4.562886
Epoch 4, Batch 301 loss: 4.562861
Epoch: 4         Training Loss: 4.556470         Validation Loss: 4.449041
Epoch 5, Batch 1 loss: 4.254582
Epoch 5, Batch 101 loss: 4.497319
Epoch 5, Batch 201 loss: 4.497112
Epoch 5, Batch 301 loss: 4.489702
Epoch: 5         Training Loss: 4.488409         Validation Loss: 4.296841
Validation loss decreased (4.439207 --> 4.296841).  Saving model ...
Epoch 6, Batch 1 loss: 4.767724
Epoch 6, Batch 101 loss: 4.428242
Epoch 6, Batch 201 loss: 4.423066
Epoch 6, Batch 301 loss: 4.436029
Epoch: 6         Training Loss: 4.438098         Validation Loss: 4.227551
Validation loss decreased (4.296841 --> 4.227551).  Saving model ...
Epoch 7, Batch 1 loss: 4.086329
Epoch 7, Batch 101 loss: 4.344385
Epoch 7, Batch 201 loss: 4.359696
Epoch 7, Batch 301 loss: 4.375801
Epoch: 7         Training Loss: 4.378317         Validation Loss: 4.150788
Validation loss decreased (4.227551 --> 4.150788).  Saving model ...
Epoch 8, Batch 1 loss: 4.599877
Epoch 8, Batch 101 loss: 4.297233
Epoch 8, Batch 201 loss: 4.314770
Epoch 8, Batch 301 loss: 4.312819
Epoch: 8         Training Loss: 4.309750         Validation Loss: 4.083044
Validation loss decreased (4.150788 --> 4.083044).  Saving model ...
Epoch 9, Batch 1 loss: 4.375301
Epoch 9, Batch 101 loss: 4.282229
Epoch 9, Batch 201 loss: 4.249982
Epoch 9, Batch 301 loss: 4.256672
Epoch: 9         Training Loss: 4.258027         Validation Loss: 4.085121
Epoch 10, Batch 1 loss: 4.383717
Epoch 10, Batch 101 loss: 4.179821
Epoch 10, Batch 201 loss: 4.213851
Epoch 10, Batch 301 loss: 4.206070
Epoch: 10         Training Loss: 4.205813         Validation Loss: 3.982710
Validation loss decreased (4.083044 --> 3.982710).  Saving model ...
Epoch 11, Batch 1 loss: 3.983536
Epoch 11, Batch 101 loss: 4.180573
Epoch 11, Batch 201 loss: 4.174407
Epoch 11, Batch 301 loss: 4.166645
Epoch: 11         Training Loss: 4.164503         Validation Loss: 3.905050
Validation loss decreased (3.982710 --> 3.905050).  Saving model ...
Epoch 12, Batch 1 loss: 3.827942
Epoch 12, Batch 101 loss: 4.075122
Epoch 12, Batch 201 loss: 4.083336
Epoch 12, Batch 301 loss: 4.085881
Epoch: 12         Training Loss: 4.088293         Validation Loss: 3.926172
Epoch 13, Batch 1 loss: 3.625863
Epoch 13, Batch 101 loss: 4.063413
Epoch 13, Batch 201 loss: 4.063286
Epoch 13, Batch 301 loss: 4.062962
Epoch: 13         Training Loss: 4.055988         Validation Loss: 3.860309
Validation loss decreased (3.905050 --> 3.860309).  Saving model ...
Epoch 14, Batch 1 loss: 3.578565
Epoch 14, Batch 101 loss: 3.999750
Epoch 14, Batch 201 loss: 3.999737
Epoch 14, Batch 301 loss: 4.002148
```

```
Epoch: 14         Training Loss: 3.997995         Validation Loss: 3.784807
Validation loss decreased (3.860309 --> 3.784807).  Saving model ...
Epoch 15, Batch 1 loss: 3.642764
Epoch 15, Batch 101 loss: 3.975488
Epoch 15, Batch 201 loss: 3.947407
Epoch 15, Batch 301 loss: 3.965880
Epoch: 15         Training Loss: 3.964724         Validation Loss: 3.755694
Validation loss decreased (3.784807 --> 3.755694).  Saving model ...
Epoch 16, Batch 1 loss: 2.980817
Epoch 16, Batch 101 loss: 3.907072
Epoch 16, Batch 201 loss: 3.912779
Epoch 16, Batch 301 loss: 3.903858
Epoch: 16         Training Loss: 3.914926         Validation Loss: 3.680523
Validation loss decreased (3.755694 --> 3.680523).  Saving model ...
Epoch 17, Batch 1 loss: 3.342377
Epoch 17, Batch 101 loss: 3.802735
Epoch 17, Batch 201 loss: 3.830981
Epoch 17, Batch 301 loss: 3.853065
Epoch: 17         Training Loss: 3.853230         Validation Loss: 3.712467
Epoch 18, Batch 1 loss: 3.602098
Epoch 18, Batch 101 loss: 3.811386
Epoch 18, Batch 201 loss: 3.813449
Epoch 18, Batch 301 loss: 3.831213
Epoch: 18         Training Loss: 3.837327         Validation Loss: 3.604288
Validation loss decreased (3.680523 --> 3.604288).  Saving model ...
Epoch 19, Batch 1 loss: 3.467219
Epoch 19, Batch 101 loss: 3.722307
Epoch 19, Batch 201 loss: 3.744970
Epoch 19, Batch 301 loss: 3.766528
Epoch: 19         Training Loss: 3.768907         Validation Loss: 3.581593
Validation loss decreased (3.604288 --> 3.581593).  Saving model ...
Epoch 20, Batch 1 loss: 3.635132
Epoch 20, Batch 101 loss: 3.692074
Epoch 20, Batch 201 loss: 3.726495
Epoch 20, Batch 301 loss: 3.742584
Epoch: 20         Training Loss: 3.743923         Validation Loss: 3.672941
```

```
In [25]: # load the model that got the best validation accuracy
         model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

1.4.4 (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 [26]: def test(loaders, model, criterion, use_cuda):
```

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

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

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

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

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

```
Test Loss: 3.720266


Test Accuracy: 13% (117/836)
```

## 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.

1.4.5 (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 [27]: ## TODO: Specify data loaders
         loaders_transfer = loaders_scratch.copy()
```

1.4.6 (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 [28]: import torchvision.models as models
         import torch.nn as nn

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

In [29]: for param in model_transfer.parameters():
             param.requires_grad = False

In [30]: model_transfer.fc = nn.Linear(2048, 133, bias=True)

In [31]: fc_parameters = model_transfer.fc.parameters()

In [32]: for param in fc_parameters:
             param.requires_grad = True

In [33]: model_transfer

Out[33]: 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)
    )
```

```
In [34]: if use_cuda:
             model_transfer = model_transfer.cuda()
```

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 picked ResNet as a transfer model because it performed outstanding on Image Classification. I looked into the structure and functions of ResNet. The core idea of ResNet is introducing a so-called "identity shortcut connection" that skips one or more layers. I guess this prevents overfitting when it's training.

I've pull out the final Fully-connected layer and replaced with Fully-connected layer with output of 133 (dog br

1.4.7 (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 [35]: criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.SGD(model_transfer.fc.parameters(), lr=0.001)
```

1.4.8 (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 [36]: # train the model
         # train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer, criterion_trans

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

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

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

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

    if batch_idx % 100 == 0:
        print('Epoch %d, Batch %d loss: %.6f' %
          (epoch, batch_idx + 1, train_loss))

######################
# validate the model #
######################
model.eval()
for batch_idx, (data, target) in enumerate(loaders['valid']):
    # move to GPU
    if use_cuda:
        data, target = data.cuda(), target.cuda()
    ## update the average validation loss
    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 decreased ({:.6f} --> {:.6f}).  Saving model ...'.fo
                valid_loss_min,
                valid_loss))
                valid_loss_min = valid_loss

        # return trained model
        return model
```

```
In [37]: train(20, loaders_transfer, model_transfer, optimizer_transfer, criterion_transfer, use
```

```
Epoch 1, Batch 1 loss: 4.829136
Epoch 1, Batch 101 loss: 4.899455
Epoch 1, Batch 201 loss: 4.864970
Epoch 1, Batch 301 loss: 4.830385
Epoch: 1        Training Loss: 4.819628         Validation Loss: 4.634914
Validation loss decreased (inf --> 4.634914).  Saving model ...
Epoch 2, Batch 1 loss: 4.647561
Epoch 2, Batch 101 loss: 4.651802
Epoch 2, Batch 201 loss: 4.626887
Epoch 2, Batch 301 loss: 4.604084
Epoch: 2        Training Loss: 4.595002         Validation Loss: 4.385592
Validation loss decreased (4.634914 --> 4.385592).  Saving model ...
Epoch 3, Batch 1 loss: 4.636754
Epoch 3, Batch 101 loss: 4.454840
Epoch 3, Batch 201 loss: 4.428420
Epoch 3, Batch 301 loss: 4.408909
Epoch: 3        Training Loss: 4.400160         Validation Loss: 4.132410
Validation loss decreased (4.385592 --> 4.132410).  Saving model ...
Epoch 4, Batch 1 loss: 4.184312
Epoch 4, Batch 101 loss: 4.260130
Epoch 4, Batch 201 loss: 4.248450
Epoch 4, Batch 301 loss: 4.221186
Epoch: 4        Training Loss: 4.216831         Validation Loss: 3.907697
Validation loss decreased (4.132410 --> 3.907697).  Saving model ...
Epoch 5, Batch 1 loss: 3.998448
Epoch 5, Batch 101 loss: 4.089659
Epoch 5, Batch 201 loss: 4.068702
Epoch 5, Batch 301 loss: 4.046999
Epoch: 5        Training Loss: 4.041185         Validation Loss: 3.704570
Validation loss decreased (3.907697 --> 3.704570).  Saving model ...
Epoch 6, Batch 1 loss: 3.959725
Epoch 6, Batch 101 loss: 3.925986
Epoch 6, Batch 201 loss: 3.908130
Epoch 6, Batch 301 loss: 3.876422
Epoch: 6         Training Loss: 3.868621         Validation Loss: 3.494985
Validation loss decreased (3.704570 --> 3.494985).  Saving model ...
Epoch 7, Batch 1 loss: 3.879548
Epoch 7, Batch 101 loss: 3.754710
Epoch 7, Batch 201 loss: 3.733770
Epoch 7, Batch 301 loss: 3.720606
Epoch: 7         Training Loss: 3.713799         Validation Loss: 3.297140
Validation loss decreased (3.494985 --> 3.297140).  Saving model ...
Epoch 8, Batch 1 loss: 3.750188
Epoch 8, Batch 101 loss: 3.606280
Epoch 8, Batch 201 loss: 3.582747
Epoch 8, Batch 301 loss: 3.574255
Epoch: 8         Training Loss: 3.567230         Validation Loss: 3.107974
Validation loss decreased (3.297140 --> 3.107974).  Saving model ...
Epoch 9, Batch 1 loss: 3.387908
Epoch 9, Batch 101 loss: 3.480219
Epoch 9, Batch 201 loss: 3.442692
Epoch 9, Batch 301 loss: 3.428284
Epoch: 9         Training Loss: 3.418375         Validation Loss: 2.969525
Validation loss decreased (3.107974 --> 2.969525).  Saving model ...
Epoch 10, Batch 1 loss: 3.401311
Epoch 10, Batch 101 loss: 3.318053
Epoch 10, Batch 201 loss: 3.309029
Epoch 10, Batch 301 loss: 3.295139
Epoch: 10         Training Loss: 3.286289         Validation Loss: 2.816649
Validation loss decreased (2.969525 --> 2.816649).  Saving model ...
Epoch 11, Batch 1 loss: 2.923185
Epoch 11, Batch 101 loss: 3.206723
Epoch 11, Batch 201 loss: 3.191664
Epoch 11, Batch 301 loss: 3.174690
Epoch: 11         Training Loss: 3.167624         Validation Loss: 2.665425
Validation loss decreased (2.816649 --> 2.665425).  Saving model ...
Epoch 12, Batch 1 loss: 3.219001
Epoch 12, Batch 101 loss: 3.075130
Epoch 12, Batch 201 loss: 3.066425
Epoch 12, Batch 301 loss: 3.055042
Epoch: 12         Training Loss: 3.046299         Validation Loss: 2.521117
Validation loss decreased (2.665425 --> 2.521117).  Saving model ...
Epoch 13, Batch 1 loss: 2.918623
Epoch 13, Batch 101 loss: 3.015089
Epoch 13, Batch 201 loss: 2.965743
Epoch 13, Batch 301 loss: 2.951859
Epoch: 13         Training Loss: 2.946013         Validation Loss: 2.396902
Validation loss decreased (2.521117 --> 2.396902).  Saving model ...
Epoch 14, Batch 1 loss: 3.131765
Epoch 14, Batch 101 loss: 2.858148
Epoch 14, Batch 201 loss: 2.854010
Epoch 14, Batch 301 loss: 2.842777
Epoch: 14         Training Loss: 2.836573         Validation Loss: 2.270243
Validation loss decreased (2.396902 --> 2.270243).  Saving model ...
Epoch 15, Batch 1 loss: 3.002994
Epoch 15, Batch 101 loss: 2.781622
Epoch 15, Batch 201 loss: 2.760726
Epoch 15, Batch 301 loss: 2.748703
Epoch: 15         Training Loss: 2.742063         Validation Loss: 2.166887
Validation loss decreased (2.270243 --> 2.166887).  Saving model ...
Epoch 16, Batch 1 loss: 2.648213
Epoch 16, Batch 101 loss: 2.674063
Epoch 16, Batch 201 loss: 2.650923
Epoch 16, Batch 301 loss: 2.651308
Epoch: 16         Training Loss: 2.647260         Validation Loss: 2.091175
Validation loss decreased (2.166887 --> 2.091175).  Saving model ...
Epoch 17, Batch 1 loss: 2.255731
Epoch 17, Batch 101 loss: 2.598961
Epoch 17, Batch 201 loss: 2.595691
Epoch 17, Batch 301 loss: 2.587842
Epoch: 17         Training Loss: 2.582735         Validation Loss: 1.972644
Validation loss decreased (2.091175 --> 1.972644).  Saving model ...
Epoch 18, Batch 1 loss: 2.594086
Epoch 18, Batch 101 loss: 2.519211
Epoch 18, Batch 201 loss: 2.522317
Epoch 18, Batch 301 loss: 2.514375
Epoch: 18         Training Loss: 2.505054         Validation Loss: 1.883675
Validation loss decreased (1.972644 --> 1.883675).  Saving model ...
Epoch 19, Batch 1 loss: 2.776507
Epoch 19, Batch 101 loss: 2.416139
Epoch 19, Batch 201 loss: 2.400115
Epoch 19, Batch 301 loss: 2.405025
Epoch: 19         Training Loss: 2.408270         Validation Loss: 1.814252
Validation loss decreased (1.883675 --> 1.814252).  Saving model ...
Epoch 20, Batch 1 loss: 2.458050
Epoch 20, Batch 101 loss: 2.398591
Epoch 20, Batch 201 loss: 2.363987
Epoch 20, Batch 301 loss: 2.343486
Epoch: 20         Training Loss: 2.341146         Validation Loss: 1.746544
Validation loss decreased (1.814252 --> 1.746544).  Saving model ...
```

```
Out[37]: 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)
        )
```

```
In [38]: # load the model that got the best validation accuracy
         model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

1.4.9 (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 [ ]:
```

```
In [39]: test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 1.822232


Test Accuracy: 72% (609/836)
```

1.4.10 (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 [40]: ### TODO: Write a function that takes a path to an image as input
         ### and returns the dog breed that is predicted by the model.

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

```
In [41]: loaders_transfer['train'].dataset.classes[:10]
```

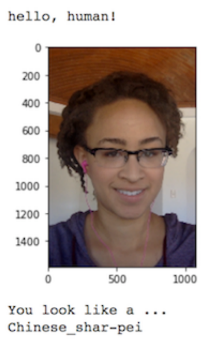
```
Out[41]: ['001.Affenpinscher',
          '002.Afghan_hound',
          '003.Airedale_terrier',
          '004.Akita',
          '005.Alaskan_malamute',
          '006.American_eskimo_dog',
          '007.American_foxhound',
          '008.American_staffordshire_terrier',
          '009.American_water_spaniel',
          '010.Anatolian_shepherd_dog']
```

```
In [42]: class_names[:10]
```

```
Out[42]: ['Affenpinscher',
          'Afghan hound',
          'Airedale terrier',
          'Akita',
          'Alaskan malamute',
          'American eskimo dog',
          'American foxhound',
          'American staffordshire terrier',
          'American water spaniel',
          'Anatolian shepherd dog']
```

```
In [43]: from PIL import Image
         import torchvision.transforms as transforms

         def load_input_image(img_path):
             image = Image.open(img_path).convert('RGB')
             prediction_transform = transforms.Compose([transforms.Resize(size=(224, 224)),
                                                  transforms.ToTensor(),
                                                  standard_normalization])

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

```
In [44]: def predict_breed_transfer(model, class_names, img_path):
             # load the image and return the predicted breed
             img = load_input_image(img_path)
             model = model.cpu()
```



```
hello, human!
```

```
You look like a ...
Chinese_shar-pei
```

Sample Human Output

```
            model.eval()
            idx = torch.argmax(model(img))
            return class_names[idx]
```

```
In [45]: for img_file in os.listdir('./images'):
             img_path = os.path.join('./images', img_file)
             predition = predict_breed_transfer(model_transfer, class_names, img_path)
             print("image_file_name: {0}, \t predition breed: {1}".format(img_path, predition))
```

```
image_file_name: ./images/Welsh_springer_spaniel_08203.jpg,          predition breed: Welsh spri
image_file_name: ./images/sample_human_output.png,          predition breed: Dogue de bordeaux
image_file_name: ./images/Labrador_retriever_06457.jpg,          predition breed: Golden retriev
image_file_name: ./images/Curly-coated_retriever_03896.jpg,          predition breed: Curly-coat
image_file_name: ./images/sample_cnn.png,          predition breed: American eskimo dog
image_file_name: ./images/Brittany_02625.jpg,          predition breed: Brittany
image_file_name: ./images/Labrador_retriever_06449.jpg,          predition breed: Flat-coated re
image_file_name: ./images/American_water_spaniel_00648.jpg,          predition breed: Irish wate
image_file_name: ./images/sample_dog_output.png,          predition breed: Italian greyhound
image_file_name: ./images/Labrador_retriever_06455.jpg,          predition breed: Chesapeake bay
```

## 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.

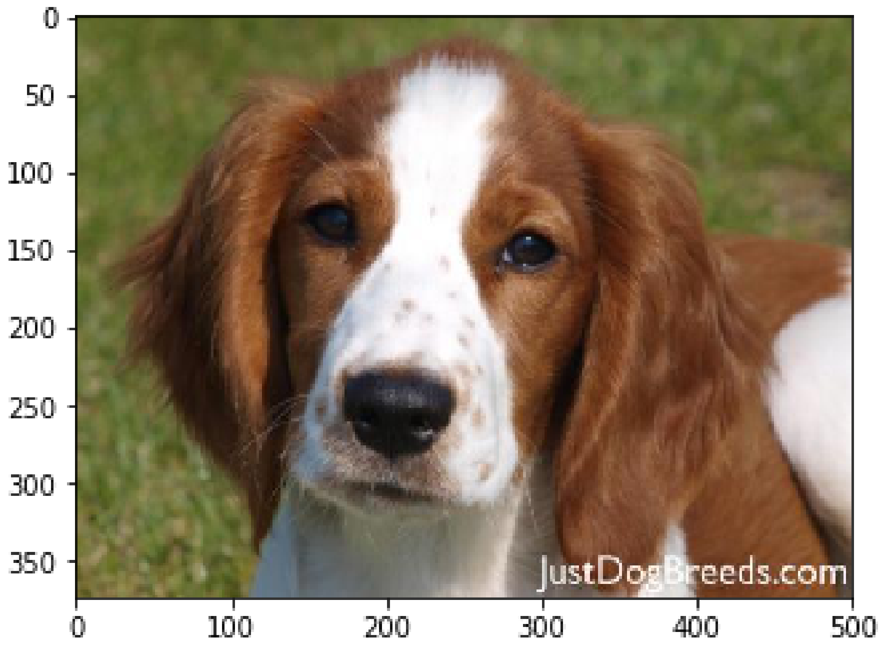
Some sample output for our algorithm is provided below, but feel free to design your own user experience!

1.4.11 (IMPLEMENTATION) Write your Algorithm

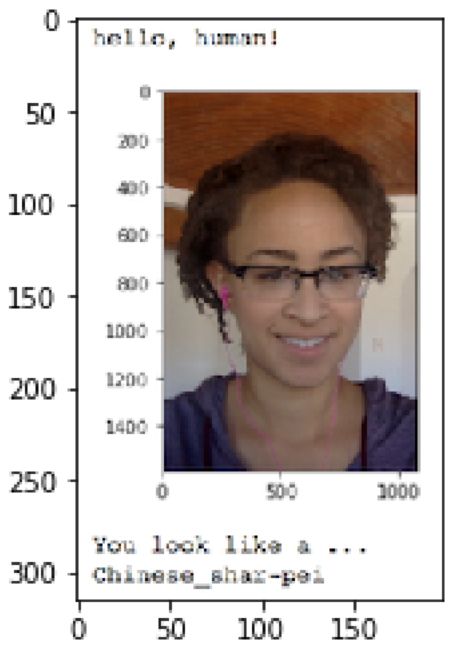
```
In [46]: ### TODO: Write your algorithm.
         ### Feel free to use as many code cells as needed.
```

```
        def run_app(img_path):
            ## handle cases for a human face, dog, and neither
            img = Image.open(img_path)
            plt.imshow(img)
            plt.show()
            if dog_detector(img_path) is True:
                prediction = predict_breed_transfer(model_transfer, class_names, img_path)
                print("Dogs Detected!\nIt looks like a {0}".format(prediction))
            elif face_detector(img_path) > 0:
                prediction = predict_breed_transfer(model_transfer, class_names, img_path)
                print("Hello, human!\nIf you were a dog..You may look like a {0}".format(predic
            else:
                print("Error! Can't detect anything..")
```

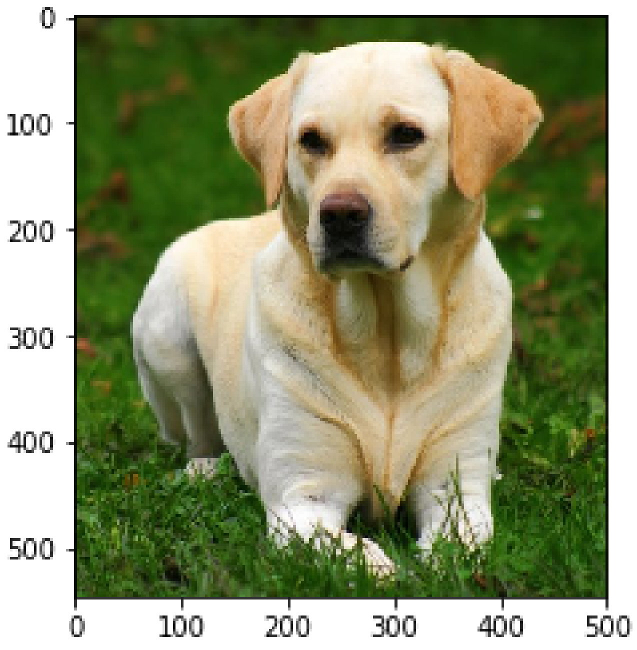
```
In [47]: for img_file in os.listdir('./images'):
            img_path = os.path.join('./images', img_file)
            run_app(img_path)
```



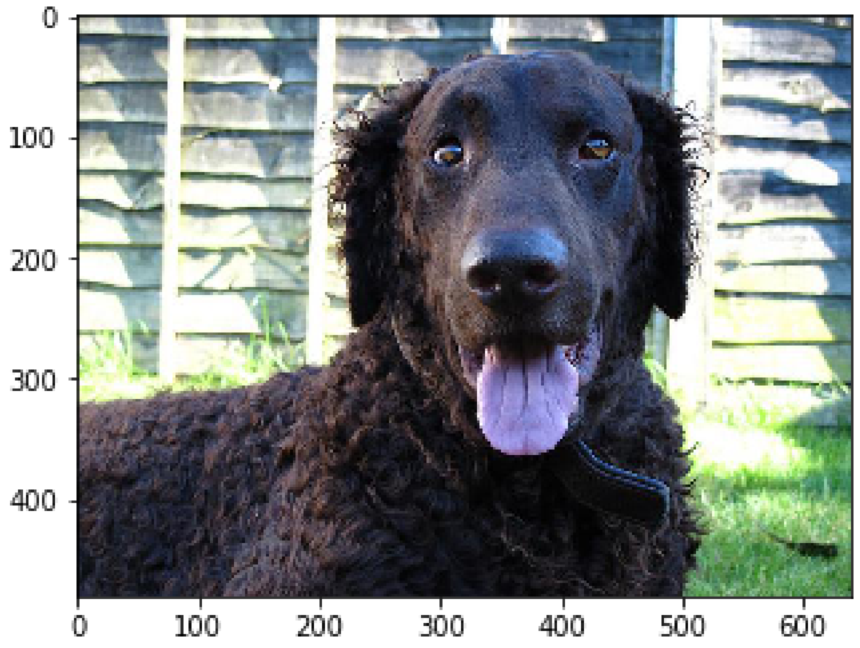
```
Dogs Detected!
It looks like a Welsh springer spaniel
```



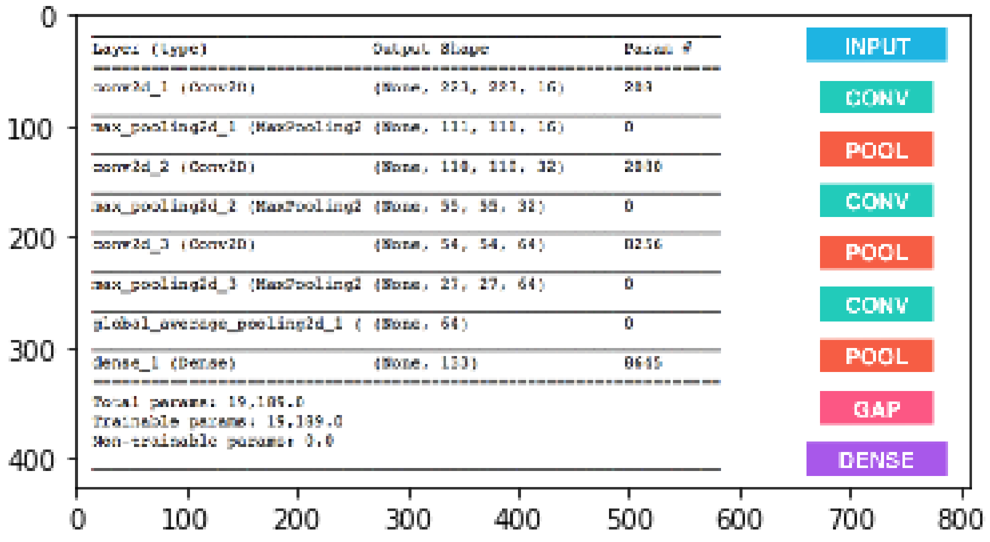
```
Hello, human!
If you were a dog..You may look like a Dogue de bordeaux
```



```
Dogs Detected!
It looks like a Golden retriever
```

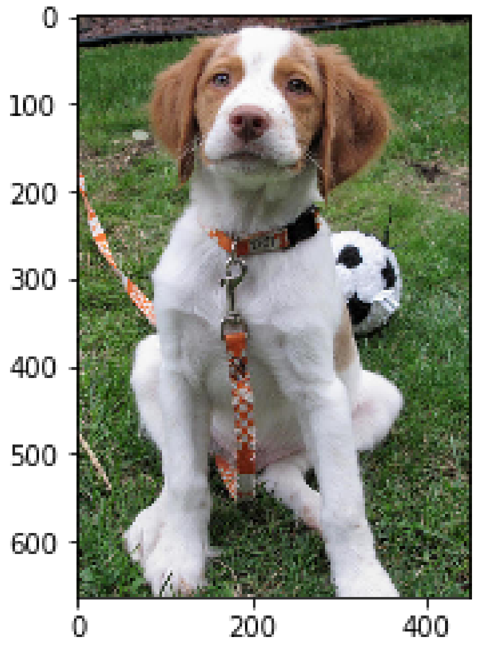


```
Dogs Detected!
It looks like a Curly-coated retriever
```



```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 223, 223, 16)      208
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 111, 111, 16)      0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 110, 110, 32)      2080
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 55, 55, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 54, 54, 64)        8256
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 27, 27, 64)        0
_________________________________________________________________
global_average_pooling2d_1 ( (None, 64)                0
_________________________________________________________________
dense_1 (Dense)              (None, 133)               8645
=================================================================
Total params: 19,189.0
Trainable params: 19,189.0
Non-trainable params: 0.0
_________________________________________________________________
```

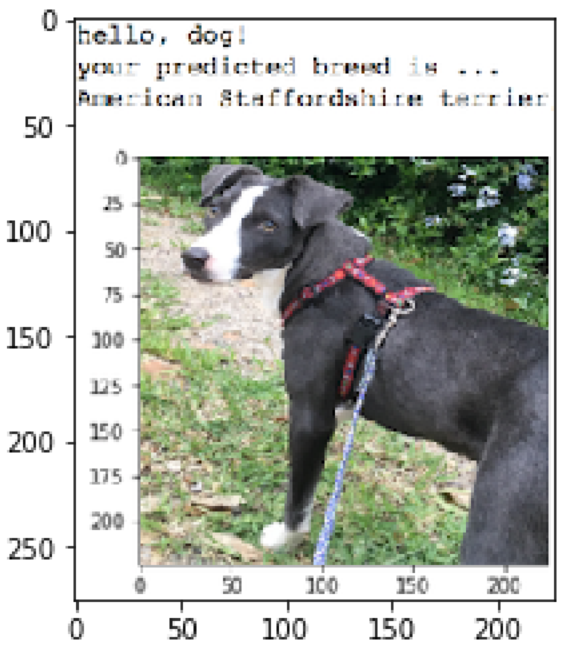
```
Error! Can't detect anything..
```



```
Dogs Detected!
It looks like a Brittany
```

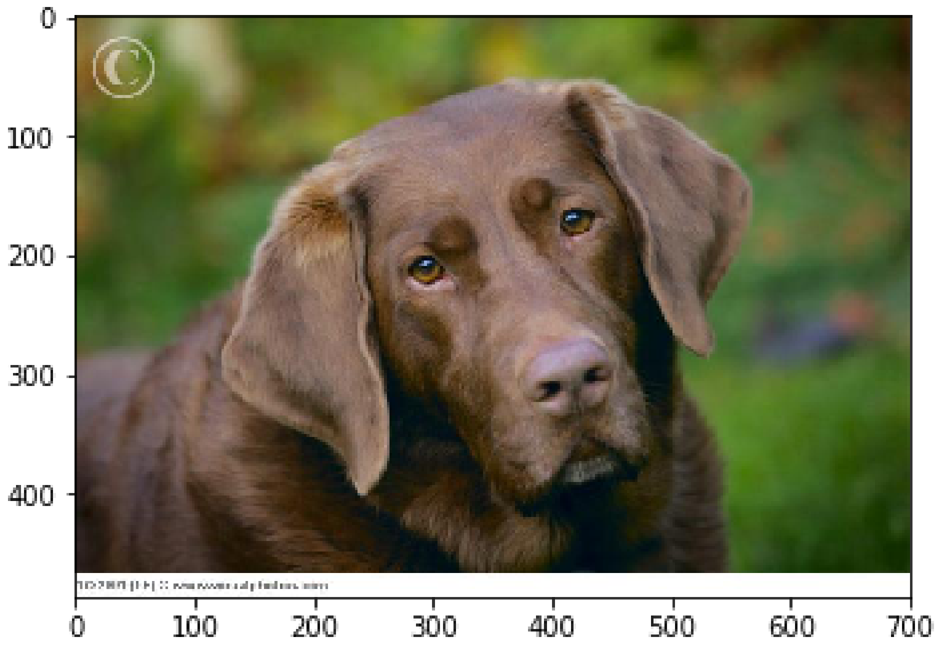
```
Dogs Detected!
It looks like a Flat-coated retriever
```

```
Dogs Detected!
It looks like a Irish water spaniel
```



```
hello, dog!
your predicted breed is ...
American Staffordshire terrier
```

```
Dogs Detected!
It looks like a Italian greyhound
```



```
Dogs Detected!
It looks like a Chesapeake bay retriever
```

## 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?

1.4.12 (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.

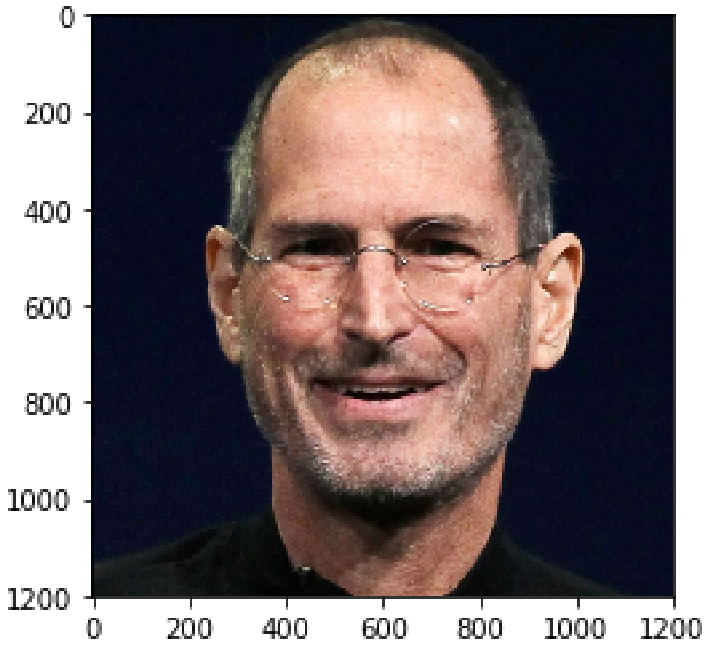
Answer: (Three possible points for improvement)

1. More image datasets of dogs will improve training models. Also, more image augmentations trials (flipping vertically, move left or right, etc.) will improve performance on test data.
2. Hyper-parameter tunings: weight initializings, learning rates, drop-outs, batch_sizes, and optimizers will be helpful to improve performances.
3. Ensembles of models

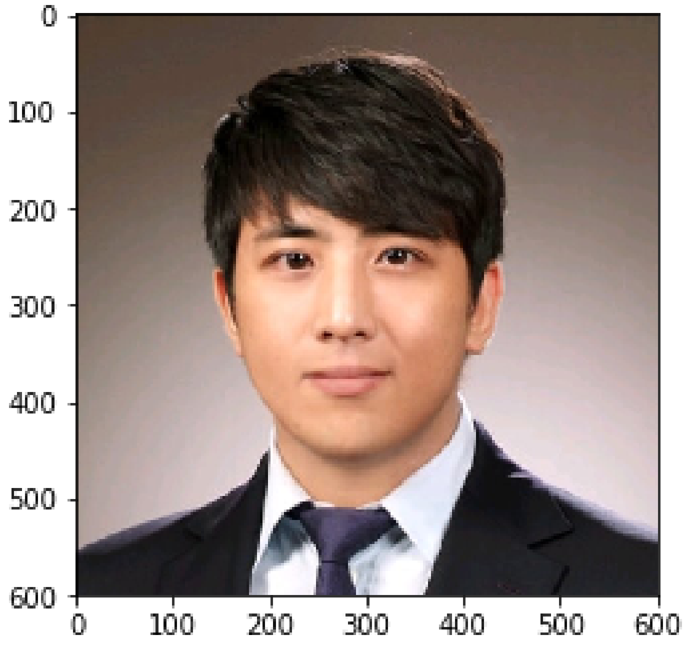
```
In [48]: ## TODO: Execute your algorithm from Step 6 on
         ## at least 6 images on your computer.
         ## Feel free to use as many code cells as needed.

         ## suggested code, below
```

```
In [49]: my_human_files = ['./my_images/human_2.jpg', './my_images/human_3.jpg' ]
         my_dog_files = ['./my_images/dog_shiba.jpeg', './my_images/dog_yorkshire.jpg']
```

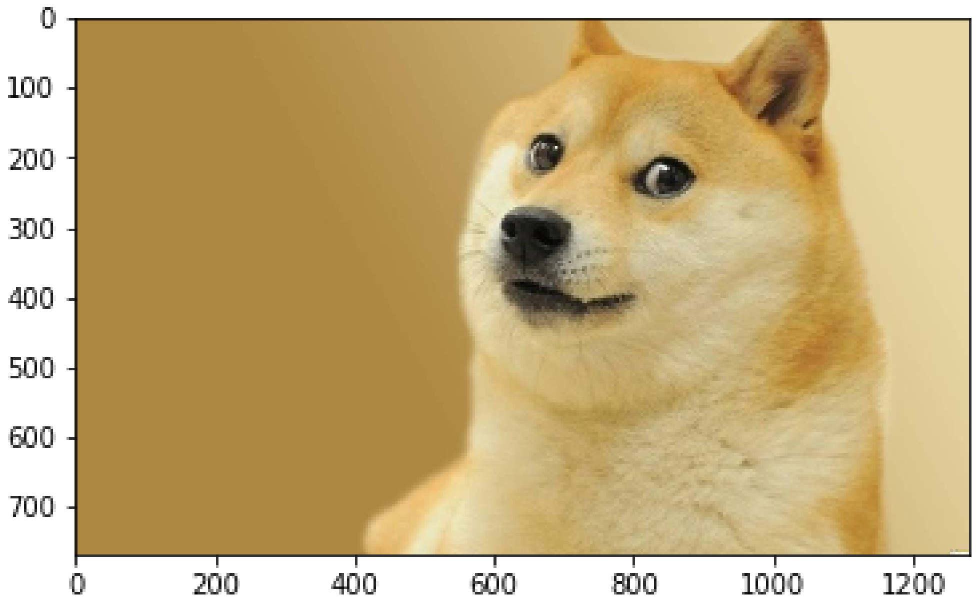
```
In [50]: ## suggested code, below
         for file in np.hstack((my_human_files, my_dog_files)):
             run_app(file)
```



```
Hello, human!
If you were a dog..You may look like a Bull terrier
```



```
Hello, human!
If you were a dog..You may look like a Poodle
```



```
Dogs Detected!
It looks like a Akita
```

```
Dogs Detected!
It looks like a Australian terrier
```

```
In [ ]:
```