dog_app

June 27, 2021

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

In this notebook, some template code has already been provided for you, and you will need to implement additional functionality to successfully complete this project. You will not need to modify the included code beyond what is requested. Sections that begin with **'(IMPLEMENTATION)'** in the header indicate that the following block of code will require additional functionality which you must provide. Instructions will be provided for each section, and the specifics of the implementation are marked in the code block with a 'TODO' statement. Please be sure to read the instructions carefully!

> **Note**: Once you have completed all of the code implementations, you need to finalize your work by exporting the Jupyter Notebook as an HTML document. Before exporting the notebook to html, all of the code cells need to have been run so that reviewers can see the final implementation and output. You can then export the notebook by using the menu above and navigating to **File -> Download as -> HTML (.html)**. Include the finished document along with this notebook as your submission.

In addition to implementing code, there will be questions that you must answer which relate to the project and your implementation. Each section where you will answer a question is preceded by a **'Question X'** header. Carefully read each question and provide thorough answers in the following text boxes that begin with **'Answer:'**. Your project submission will be evaluated based on your answers to each of the questions and the implementation you provide.

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

The rubric contains *optional* "Stand Out Suggestions" for enhancing the project beyond the minimum requirements. If you decide to pursue the "Stand Out Suggestions", you should include the code in this Jupyter notebook.

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("/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

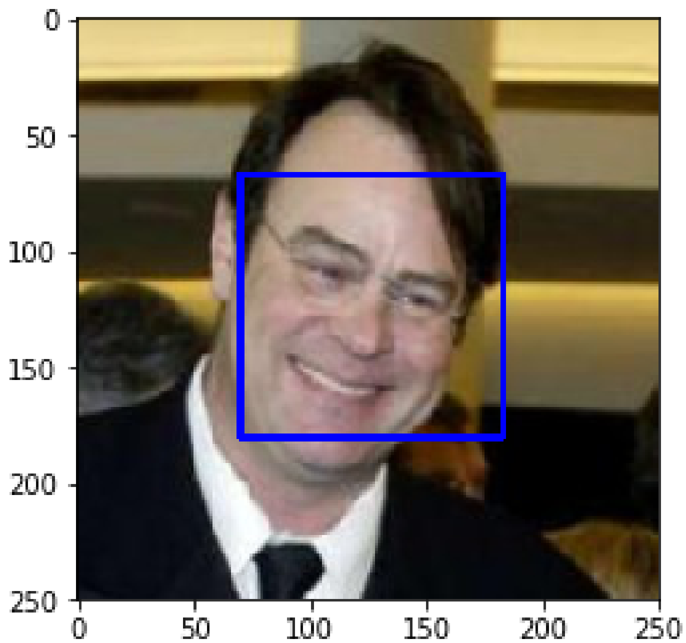
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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))
```

```python
# get bounding box for each detected face
for (x,y,w,h) in faces:
    # add bounding box to color image
    cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)

# convert BGR image to RGB for plotting
cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

# display the image, along with bounding box
plt.imshow(cv_rgb)
plt.show()
```

```
Number of faces detected: 1
```



Before using any of the face detectors, it is standard procedure to convert the images to grayscale. The `detectMultiScale` function executes the classifier stored in `face_cascade` and takes the grayscale image as a parameter.

In the above code, `faces` is a numpy array of detected faces, where each row corresponds to a detected face. Each detected face is a 1D array with four entries that specifies the bounding box of the detected face. The first two entries in the array (extracted in the above code as `x` and `y`) specify the horizontal and vertical positions of the top left corner of the bounding box. The last two entries in the array (extracted here as `w` and `h`) specify the width and height of the box.

1.1.1 Write a Human Face Detector

We can use this procedure to write a function that returns `True` if a human face is detected in an image and `False` otherwise. This function, aptly named `face_detector`, takes a string-valued file path to an image as input and appears in the code block below.

```
In [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: 98% of the human faces were detected in human files but only about 17% of the dog faces were recognized.

```
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.
        num_human = 0
        for img in human_files_short:
            isHuman = face_detector(img)
            if isHuman:
                num_human += 1
        percentage = (num_human/len(human_files_short)) * 100
        print('Percentage of humans correctly classified as people: {}%'.format(percentage))

        num_dog = 0
        for img in dog_files_short:
            isHuman = face_detector(img)
            if isHuman:
                num_dog += 1
        percentage = (num_dog/len(dog_files_short)) * 100
        print('Percentage of dogs misclassified as people: {}%'.format(percentage))
```

```
Percentage of humans correctly classified as people: 98.0%
Percentage of dogs misclassified as people: 17.0%
```

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

```
In [5]: ### (Optional)
        ### TODO: Test performance of anotherface detection algorithm.
        ### Feel free to use as many code cells as needed.
```

Step 2: Detect Dogs
In this section, we use a pre-trained model to detect dogs in images.

1.1.3 Obtain Pre-trained VGG-16 Model

The code cell below downloads the VGG-16 model, along with weights that have been trained on ImageNet, a very large, very popular dataset used for image classification and other vision tasks. ImageNet contains over 10 million URLs, each linking to an image containing an object from one of 1000 categories.

```
In [6]: import torch
        import torchvision.models as models

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

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

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

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:09<00:00, 56234679.37it/s]
```

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

1.1.4 (IMPLEMENTATION) Making Predictions with a Pre-trained Model

In the next code cell, you will write a function that accepts a path to an image (such as `'dogImages/train/001.Affenpinscher/Affenpinscher_00001.jpg'`) as input and returns the index corresponding to the ImageNet class that is predicted by the pre-trained VGG-16 model. The output should always be an integer between 0 and 999, inclusive.

Before writing the function, make sure that you take the time to learn how to appropriately pre-process tensors for pre-trained models in the PyTorch documentation.

In [7]:
```
from PIL import Image
import torchvision.transforms as transforms

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

    Args:
        img_path: path to an image

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

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

    img = Image.open(img_path)

    transform_pipeline = transforms.Compose([transforms.RandomResizedCrop(250),
                                              transforms.ToTensor()])

    img_tensor = transform_pipeline(img)
    img_tensor = img_tensor.unsqueeze(0)

    if torch.cuda.is_available():
        img_tensor = img_tensor.cuda()

    prediction = VGG16(img_tensor)

    if torch.cuda.is_available():
        prediction = prediction.cpu()

    index = prediction.data.numpy().argmax()

    return index
```

1.1.5 (IMPLEMENTATION) Write a Dog Detector

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

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

```
In [8]: ### returns "True" if a dog is detected in the image stored at img_path
        def dog_detector(img_path):
            ## TODO: Complete the function.
            index = VGG16_predict(img_path)
            if index >= 151 and index <= 268:
                return True
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

Question 2: Use the code cell below to test the performance of your `dog_detector` function.
- What percentage of the images in `human_files_short` have a detected dog?
- What percentage of the images in `dog_files_short` have a detected dog?

Answer: 1% of the humans were misclassified whereas 74% of the dogs were correctly classified

```
In [9]: ### TODO: Test the performance of the dog_detector function
        ### on the images in human_files_short and dog_files_short.
        num_human = 0
        for img in human_files_short:
            isHuman = dog_detector(img)
            if isHuman:
                num_human += 1
        percentage = (num_human/len(human_files_short)) * 100
        print('Percentage of humans misclassified as people: {}%'.format(percentage))

        num_dog = 0
        for img in dog_files_short:
            isHuman = dog_detector(img)
            if isHuman:
                num_dog += 1
        percentage = (num_dog/len(dog_files_short)) * 100
        print('Percentage of correctly classified as people: {}%'.format(percentage))
```

```
Percentage of humans misclassified as people: 1.0%
Percentage of correctly classified as people: 74.0%
```

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

```
In [10]: ### (Optional)
         ### TODO: Report the performance of another pre-trained network.
         ### Feel free to use as many code cells as needed.
```

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

Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of at least 10%. In Step 4 of this notebook, you will have the opportunity to use transfer learning to create a CNN that attains greatly improved accuracy.

We mention that the task of assigning breed to dogs from images is considered exceptionally challenging. To see why, consider that *even a human* would have trouble distinguishing between a Brittany and a Welsh Springer Spaniel.

Brittany	Welsh Springer Spaniel

It is not difficult to find other dog breed pairs with minimal inter-class variation (for instance, Curly-Coated Retrievers and American Water Spaniels).

Curly-Coated Retriever	American Water Spaniel

Likewise, recall that labradors come in yellow, chocolate, and black. Your vision-based algorithm will have to conquer this high intra-class variation to determine how to classify all of these different shades as the same breed.

Yellow Labrador	Chocolate Labrador

We also mention that random chance presents an exceptionally low bar: setting aside the fact that the classes are slightly imabalanced, a random guess will provide a correct answer roughly 1 in 133 times, which corresponds to an accuracy of less than 1%.

Remember that the practice is far ahead of the theory in deep learning. Experiment with many different architectures, and trust your intuition. And, of course, have fun!

1.1.7 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

Use the code cell below to write three separate data loaders for the training, validation, and test datasets of dog images (located at `dog_images/train`, `dog_images/valid`, and `dog_images/test`, respectively). You may find this documentation on custom datasets to be a useful resource. If you are interested in augmenting your training and/or validation data, check out the wide variety of transforms!

```
In [11]: import os
         from torchvision import datasets

         ### TODO: Write data loaders for training, validation, and test sets
         ## Specify appropriate transforms, and batch_sizes
         param_transform_resize = 224
         param_transform_crop = 224
         param_data_directory = "/data/dog_images"

         # define transforms for the training data and testing data
         train_transforms = transforms.Compose([transforms.Resize(param_transform_resize),
                                               transforms.CenterCrop(param_transform_crop),
                                               transforms.RandomHorizontalFlip(),
                                               transforms.RandomVerticalFlip(),
                                               transforms.RandomRotation(20),
                                               transforms.ToTensor(),
                                               transforms.Normalize([0.485, 0.456, 0.406],
                                                                    [0.229, 0.224, 0.225])])

         test_transforms = transforms.Compose([transforms.Resize(param_transform_resize),
                                              transforms.CenterCrop(param_transform_crop),
                                              transforms.ToTensor(),
                                              transforms.Normalize([0.485, 0.456, 0.406],
                                                                   [0.229, 0.224, 0.225])])


         # pass transforms in here, then run the next cell to see how the transforms look
         train_data = datasets.ImageFolder( param_data_directory + '/train', transform=train_tra
         test_data = datasets.ImageFolder( param_data_directory + '/test', transform=test_transf
         valid_data = datasets.ImageFolder( param_data_directory + '/valid', transform=test_tran

         trainloader = torch.utils.data.DataLoader( train_data, batch_size=32, shuffle=True, num
         testloader = torch.utils.data.DataLoader( test_data, batch_size=16, shuffle = False,  n
         validloader = torch.utils.data.DataLoader( valid_data, batch_size=16, shuffle = False,

         # create dictionary for all loaders in one
         loaders_scratch = {
             'train': trainloader,
             'valid': validloader,
             'test': testloader
         }
```

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: Loaded the training, testing and validation datasets and then created data loaders for the same. After this, I resized all image to center cropped, 224 pixel. Randomly adding rotation, horizontal and vertical flips also helps with overfitting the data.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [60]: import torch.nn as nn
         import torch.nn.functional as F

         # define the CNN architecture
         class Net(nn.Module):
             ### TODO: choose an architecture, and complete the class

             def __init__(self):
                 super(Net, self).__init__()
                 ## Define layers of a CNN

                 self.conv1 = nn.Conv2d(3, 16, 3, padding=1)

                 self.conv2 = nn.Conv2d(16, 32, 3, padding=1)

                 self.conv3 = nn.Conv2d(32, 64, 3, padding=1)

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

                 self.fc1 = nn.Linear(28 * 28 * 64, 500)
                 self.fc2 = nn.Linear(500, len(train_data.classes))
                 self.dropout = nn.Dropout(0.25)
                 self.batch_norm = nn.BatchNorm1d(num_features=500)

             def forward(self, x):
                 ## Define forward behavior
                 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(x.size(0), -1)
                 x = self.dropout(x)
                 x = F.relu(self.batch_norm( 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, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(32, 64, 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=50176, out_features=500, bias=True)
  (fc2): Linear(in_features=500, out_features=133, bias=True)
  (dropout): Dropout(p=0.25)
  (batch_norm): BatchNorm1d(500, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
)
```

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

Answer:

I have used three convoluted layers, the last layer outputting 133 dog classes and 2D maxpooling. The convoluted layers I have used are nn.Conv2d(3, 16, 3, padding=1), nn.Conv2d(16, 32, 3, padding=1), nn.Conv2d(32, 64, 3, padding=1).

I chose 2D maxpooling to down-sample because it is simple and a choice used often. This will also help in preventing overfitting of features in each layer. For maxpooling, I have gone with nn.MaxPool2d(2, 2) and is effective because it will down-sample x and y's dimensions by a factor of 2.

I also add a linear layer at the end to product a 133 dimension output as we need and dropout to prevent overfitting. The dropout layer was implemented to add a bit of randomness to the model since the dropout layer will randomly drop weights into the layer and causes the outputs to be scaled by a factor of p/(1-p) where in this case, we have used a p = 0.25. The dropout method has proven to be an effective technique for regularization and preventing the co-adaptation of neurons.

Then finally, the forward function dictates the forward behavior of the model.

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 [14]: import torch.optim as optim

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

         ### TODO: select optimizer
```

```
optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=0.01)

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

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 [16]: def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
             """returns trained model"""
             print("start training for {} epochs ...".format(n_epochs))
             # initialize tracker for minimum validation loss
             valid_loss_min = np.Inf

             # exist save-file, load save file
             if os.path.exists(save_path):
                 print("load previous saved model ...")
                 model.load_state_dict(torch.load(save_path))

             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()    # --- set model to train mode
                 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
                     # -----------------------------
                     # clear the gradients of all optimized variables
                     optimizer.zero_grad()
                     # forward pass: compute predicted outputs by passing inputs to the model
                     output = model(data)
                     # calculate the batch loss
                     loss = criterion(output, target)
                     # backward pass: compute gradient of the loss with respect to model paramet
                     loss.backward()
                     # perform a single optimization step (parameter update)
                     optimizer.step()
```

```
            # update training loss
            #train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss
            train_loss += loss.item()*data.size(0)
            # ----------------------------

        ######################
        # validate the model #
        ######################
        model.eval()        # ---- set model to evaluation mode
        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
            # ----------------------------
            # forward pass: compute predicted outputs by passing inputs to the model
            output = model(data)
            # calculate the batch loss
            loss = criterion(output, target)
            # update average validation loss
            valid_loss += loss.item() * data.size(0)
            # ----------------------------

        # ----------------------------
        # calculate average losses
        train_loss = train_loss / len(loaders['train'].dataset)
        valid_loss = valid_loss / len(loaders['valid'].dataset)
        # ----------------------------

        # print training/validation statistics
        print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format( epo

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

    print("done")
    # return trained model
    return model
```

In [17]:
```
# ---Defining Param-----
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

param_epochs = 50

# train the model
model_scratch = train(param_epochs, loaders_scratch, model_scratch, optimizer_scratch,
                      criterion_scratch, use_cuda, 'model_scratch.pt')
```

```
start training for 50 epochs ...
Epoch: 1        Training Loss: 4.801988        Validation Loss: 4.677920  Saving model ...
Epoch: 2        Training Loss: 4.598128        Validation Loss: 4.563519  Saving model ...
Epoch: 3        Training Loss: 4.469912        Validation Loss: 4.473160  Saving model ...
Epoch: 4        Training Loss: 4.372150        Validation Loss: 4.379837  Saving model ...
Epoch: 5        Training Loss: 4.287302        Validation Loss: 4.327372  Saving model ...
Epoch: 6        Training Loss: 4.215868        Validation Loss: 4.295828  Saving model ...
Epoch: 7        Training Loss: 4.146123        Validation Loss: 4.223012  Saving model ...
Epoch: 8        Training Loss: 4.086580        Validation Loss: 4.209549  Saving model ...
Epoch: 9        Training Loss: 4.018554        Validation Loss: 4.206627  Saving model ...
Epoch: 10        Training Loss: 3.957132        Validation Loss: 4.117373  Saving model ...
Epoch: 11        Training Loss: 3.897040        Validation Loss: 4.100388  Saving model ...
Epoch: 12        Training Loss: 3.846616        Validation Loss: 4.040172  Saving model ...
Epoch: 13        Training Loss: 3.773540        Validation Loss: 4.017182  Saving model ...
Epoch: 14        Training Loss: 3.729786        Validation Loss: 3.966501  Saving model ...
Epoch: 15        Training Loss: 3.657783        Validation Loss: 3.896129  Saving model ...
Epoch: 16        Training Loss: 3.615992        Validation Loss: 3.906564
Epoch: 17        Training Loss: 3.550210        Validation Loss: 3.943464
Epoch: 18        Training Loss: 3.499192        Validation Loss: 3.820334  Saving model ...
Epoch: 19        Training Loss: 3.449738        Validation Loss: 3.879319
Epoch: 20        Training Loss: 3.396656        Validation Loss: 3.826475
Epoch: 21        Training Loss: 3.344506        Validation Loss: 3.836145
Epoch: 22        Training Loss: 3.283201        Validation Loss: 3.776577  Saving model ...
Epoch: 23        Training Loss: 3.231855        Validation Loss: 3.731792  Saving model ...
Epoch: 24        Training Loss: 3.184089        Validation Loss: 3.745505
Epoch: 25        Training Loss: 3.136433        Validation Loss: 3.862862
Epoch: 26        Training Loss: 3.080465        Validation Loss: 3.750353
Epoch: 27        Training Loss: 3.023791        Validation Loss: 3.847730
Epoch: 28        Training Loss: 2.973949        Validation Loss: 3.700025  Saving model ...
Epoch: 29        Training Loss: 2.935419        Validation Loss: 3.648092  Saving model ...
Epoch: 30        Training Loss: 2.872530        Validation Loss: 3.679000
Epoch: 31        Training Loss: 2.818180        Validation Loss: 3.792285
Epoch: 32        Training Loss: 2.746999        Validation Loss: 3.624864  Saving model ...
Epoch: 33        Training Loss: 2.707788        Validation Loss: 3.550603  Saving model ...
Epoch: 34        Training Loss: 2.674686        Validation Loss: 3.615180
Epoch: 35        Training Loss: 2.615340        Validation Loss: 3.544678  Saving model ...
Epoch: 36        Training Loss: 2.555373        Validation Loss: 3.618633
Epoch: 37        Training Loss: 2.520397        Validation Loss: 3.710036
```

```
Epoch: 38       Training Loss: 2.462733      Validation Loss: 3.517357  Saving model ...
Epoch: 39       Training Loss: 2.417157      Validation Loss: 3.640233
Epoch: 40       Training Loss: 2.374699      Validation Loss: 3.569021
Epoch: 41       Training Loss: 2.314197      Validation Loss: 3.624237
Epoch: 42       Training Loss: 2.256395      Validation Loss: 3.538845
Epoch: 43       Training Loss: 2.216783      Validation Loss: 3.530317
Epoch: 44       Training Loss: 2.167819      Validation Loss: 3.545329
Epoch: 45       Training Loss: 2.102751      Validation Loss: 3.609093
Epoch: 46       Training Loss: 2.078983      Validation Loss: 3.575150
Epoch: 47       Training Loss: 2.043333      Validation Loss: 3.529112
Epoch: 48       Training Loss: 1.985230      Validation Loss: 3.545127
Epoch: 49       Training Loss: 1.937228      Validation Loss: 3.499937  Saving model ...
Epoch: 50       Training Loss: 1.875049      Validation Loss: 3.565478
done
```

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 [18]: 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 = 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: 3.620545


Test Accuracy: 16% (141/836)
```

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 [19]: ## TODO: 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 [25]: import torchvision.models as models
         import torch.nn as nn

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

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

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

         fc_parameters = model_transfer.fc.parameters()

         for parameter in fc_parameters:
             parameter.requires_grad = True
```

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

```
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: ResNet was used as it is a very effective method as it the previous layers are used to feed data into the next layers. This makes the features more powerful and accuracy is improved.

Here I have used transfer learning to train a network that can classify dog images. The classifer part of the model I have used is a Linear model for the fully connected layer with features nn.Linear(2048, 133, bias=True). Seeing as that the fully connected layer is predominantly used for the forward behavior, nn.Linear is applicable since it will use 2048 inputs and create 133 outputs for the 133 breeds.

ResNet50 architecture is an excellent model because ResNet network uses a 34-layer plain network architecture which was developed as a result of VGG-19. Shortcut connections are also added and these shortcut connections then convert the architecture into the residual network.

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 [27]: criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.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 [28]: # train the model
         model_transfer = train(param_epochs, loaders_transfer, model_transfer, optimizer_transf
                                criterion_transfer, use_cuda, 'model_transfer.pt')

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

```
start training for 50 epochs ...
Epoch: 1         Training Loss: 4.500599        Validation Loss: 3.817800  Saving model ...
Epoch: 2         Training Loss: 3.735618        Validation Loss: 2.950638  Saving model ...
Epoch: 3         Training Loss: 3.177112        Validation Loss: 2.415431  Saving model ...
Epoch: 4         Training Loss: 2.748735        Validation Loss: 1.981119  Saving model ...
Epoch: 5         Training Loss: 2.437793        Validation Loss: 1.679195  Saving model ...
Epoch: 6         Training Loss: 2.229697        Validation Loss: 1.495493  Saving model ...
Epoch: 7         Training Loss: 2.051385        Validation Loss: 1.313014  Saving model ...
Epoch: 8         Training Loss: 1.900481        Validation Loss: 1.192562  Saving model ...
Epoch: 9         Training Loss: 1.795679        Validation Loss: 1.099309  Saving model ...
Epoch: 10         Training Loss: 1.710554        Validation Loss: 1.037682  Saving model ...
Epoch: 11         Training Loss: 1.624440        Validation Loss: 0.959847  Saving model ...
Epoch: 12         Training Loss: 1.586273        Validation Loss: 0.910881  Saving model ...
Epoch: 13         Training Loss: 1.497395        Validation Loss: 0.878000  Saving model ...
Epoch: 14         Training Loss: 1.444137        Validation Loss: 0.837889  Saving model ...
Epoch: 15         Training Loss: 1.414137        Validation Loss: 0.807969  Saving model ...
Epoch: 16         Training Loss: 1.363771        Validation Loss: 0.793939  Saving model ...
Epoch: 17         Training Loss: 1.352430        Validation Loss: 0.749769  Saving model ...
Epoch: 18         Training Loss: 1.310754        Validation Loss: 0.735718  Saving model ...
Epoch: 19         Training Loss: 1.277903        Validation Loss: 0.731844  Saving model ...
Epoch: 20         Training Loss: 1.246061        Validation Loss: 0.707495  Saving model ...
Epoch: 21         Training Loss: 1.242312        Validation Loss: 0.682573  Saving model ...
Epoch: 22         Training Loss: 1.219294        Validation Loss: 0.672332  Saving model ...
Epoch: 23         Training Loss: 1.190118        Validation Loss: 0.668709  Saving model ...
Epoch: 24         Training Loss: 1.169959        Validation Loss: 0.650566  Saving model ...
Epoch: 25         Training Loss: 1.143648        Validation Loss: 0.654021
```

```
Epoch: 26         Training Loss: 1.125292         Validation Loss: 0.645956  Saving model ...
Epoch: 27         Training Loss: 1.113363         Validation Loss: 0.629227  Saving model ...
Epoch: 28         Training Loss: 1.088679         Validation Loss: 0.622860  Saving model ...
Epoch: 29         Training Loss: 1.084683         Validation Loss: 0.613485  Saving model ...
Epoch: 30         Training Loss: 1.081354         Validation Loss: 0.607068  Saving model ...
Epoch: 31         Training Loss: 1.067878         Validation Loss: 0.612292
Epoch: 32         Training Loss: 1.035227         Validation Loss: 0.595983  Saving model ...
Epoch: 33         Training Loss: 1.031238         Validation Loss: 0.580314  Saving model ...
Epoch: 34         Training Loss: 1.018086         Validation Loss: 0.584460
Epoch: 35         Training Loss: 1.024919         Validation Loss: 0.586371
Epoch: 36         Training Loss: 1.000058         Validation Loss: 0.573042  Saving model ...
Epoch: 37         Training Loss: 0.991055         Validation Loss: 0.574018
Epoch: 38         Training Loss: 0.975827         Validation Loss: 0.571047  Saving model ...
Epoch: 39         Training Loss: 0.977063         Validation Loss: 0.559331  Saving model ...
Epoch: 40         Training Loss: 0.956751         Validation Loss: 0.561287
Epoch: 41         Training Loss: 0.955011         Validation Loss: 0.551844  Saving model ...
Epoch: 42         Training Loss: 0.932747         Validation Loss: 0.546677  Saving model ...
Epoch: 43         Training Loss: 0.934123         Validation Loss: 0.540680  Saving model ...
Epoch: 44         Training Loss: 0.930275         Validation Loss: 0.545332
Epoch: 45         Training Loss: 0.908657         Validation Loss: 0.555367
Epoch: 46         Training Loss: 0.921612         Validation Loss: 0.536732  Saving model ...
Epoch: 47         Training Loss: 0.898707         Validation Loss: 0.547314
Epoch: 48         Training Loss: 0.890394         Validation Loss: 0.537725
Epoch: 49         Training Loss: 0.869518         Validation Loss: 0.535938  Saving model ...
Epoch: 50         Training Loss: 0.890750         Validation Loss: 0.532726  Saving model ...
done
```

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

```
Test Loss: 0.549265


Test Accuracy: 84% (707/836)
```

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 [55]: ### 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 train_data.classes]

def predict_breed_transfer(img_path):
    # load the image and return the predicted breed
    image_tensor = image_convert_tensor(img_path)

    if use_cuda:
        image_tensor = image_tensor.cuda()

    y = model_transfer(image_tensor)

    _, preds_tensor = torch.max(y, 1)
    if not use_cuda:
        prediction = np.squeeze(preds_tensor.numpy())
    else:
        prediction = np.squeeze(preds_tensor.cpu().numpy())

    return class_names[prediction]

def print_image(img_path, title="Title"):
    image = Image.open(img_path)
    plt.title(title)
    plt.imshow(image)
    plt.show()
```

Step 5: Write your Algorithm

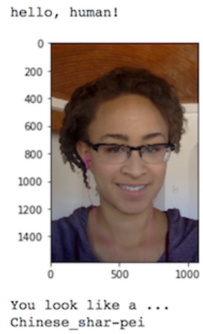
Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then, - if a **dog** is detected in the image, return the predicted breed. - if a **human** is detected in the image, return the resembling dog breed. - if **neither** is 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 [56]:
```
### 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
    if (face_detector(img_path)):
        print("Hello, human!")
```



Sample Human Output

```
        predicted_breed = predict_breed_transfer(img_path)
        print_image(img_path, title="Predicted breed: {}".format(predicted_breed) )

        print("You most closely resemble-")
        print(predicted_breed)
    elif (dog_detector(img_path)):
        print("Hi, dog!")
        predicted_breed = predict_breed_transfer(img_path)
        print_image(img_path, title="Predicted breed: {}".format(predicted_breed) )

        print("Your breed is-")
        print(predicted_breed)
    else:
        print("Oh no! Looks like we weren't able to predict what type of breed you are!
        print_image(img_path, title="...")
        print("Please try again :)")
```

```
In [57]: def image_convert_tensor(image):
             prediction_transforms = transforms.Compose([transforms.Resize(param_transform_resiz
                                                         transforms.CenterCrop(param_transform_crop),
                                                         transforms.ToTensor(),
                                                         transforms.Normalize([0.485, 0.456, 0.406],
                                                                              [0.229, 0.224, 0.225])])

             img_pil = Image.open( image ).convert('RGB')
             img_tensor = prediction_transforms( img_pil )[:3,:,:].unsqueeze(0)

             return img_tensor

         def image_convert(tensor):
             """ Display a tensor as an image. """

             image = tensor.to("cpu").clone().detach()
             image = image.numpy().squeeze()
             image = image.transpose(1,2,0)
             image = image * np.array((0.229, 0.224, 0.225)) + np.array((0.485, 0.456, 0.406))
```

```
        image = image.clip(0, 1)

        return image
```

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!

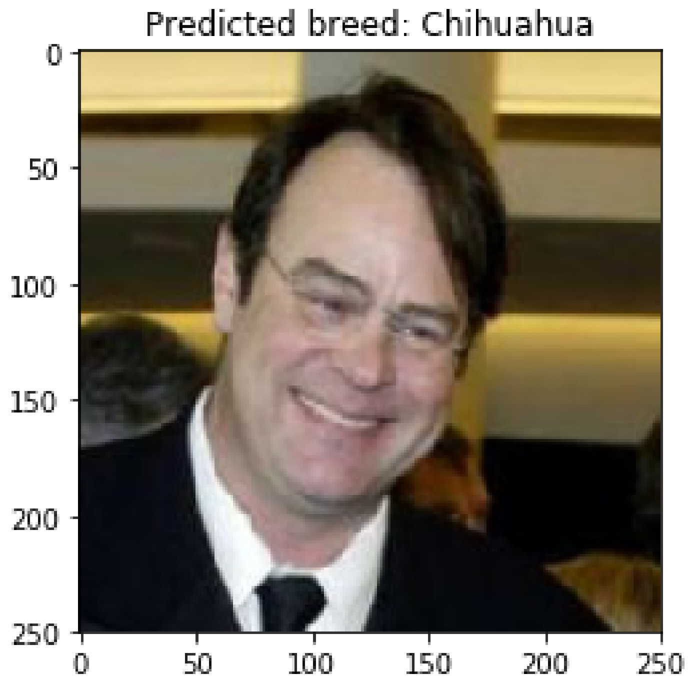
Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

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

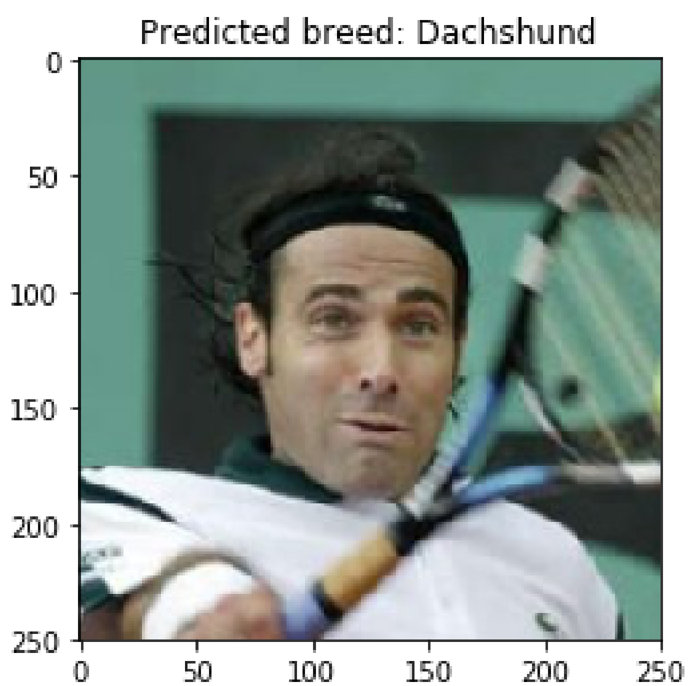
Answer: The output is better than I expected. Since the model had a test accuracy of about 84%, I thought the model would make some mistakes but there were none made. One area for improvement could be increasing the number of epochs run, this would also increase the time of training but we could iterate through various number of epochs to find the optimum. Another aspect would be to increase the classes of dogs since this will help in improving model accuracy. Finally, I think using fully connected layers will also help.

```
In [58]: ## 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
         for file in np.hstack((human_files[:3], dog_files[:3])):
             run_app(file)
```

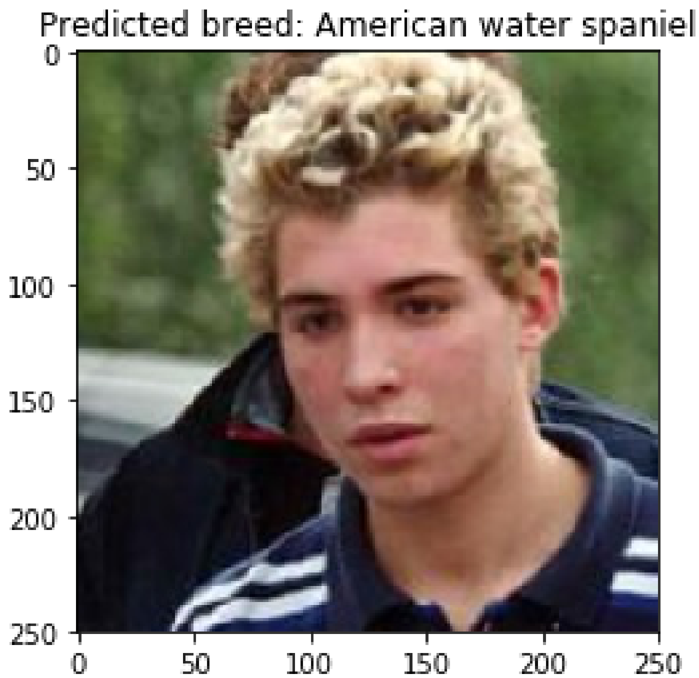
```
Hello, human!
```



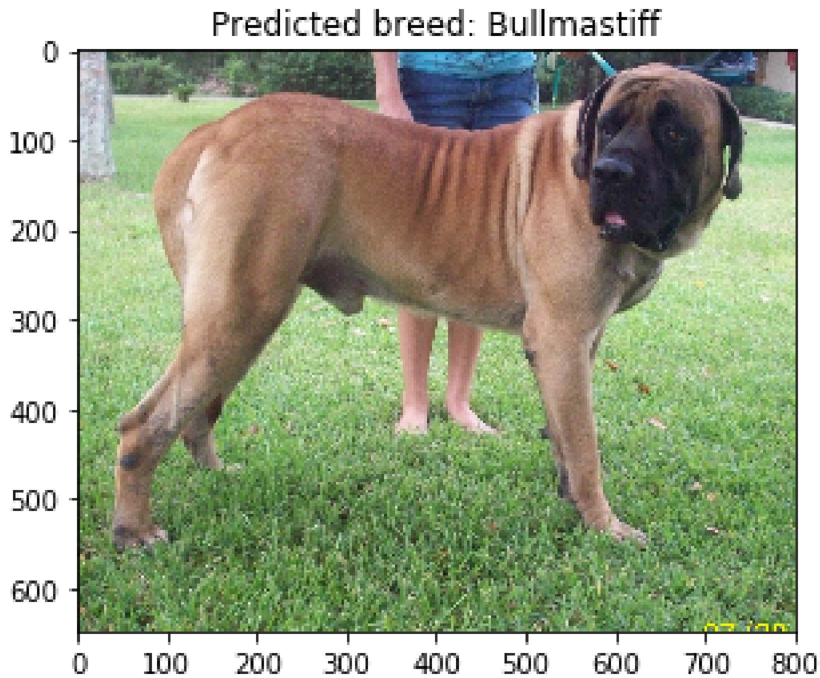

```
You most closely resemble-
Chihuahua
Hello, human!
```



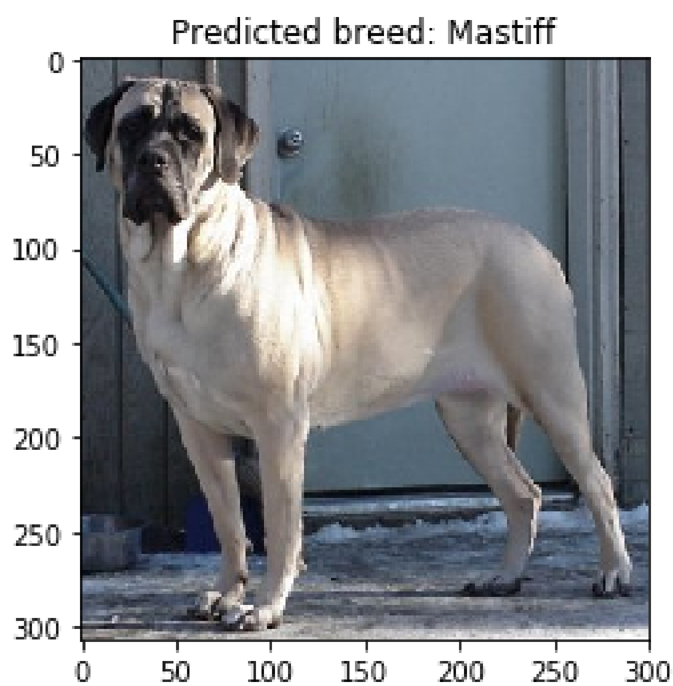
```
You most closely resemble-
Dachshund
Hello, human!
```



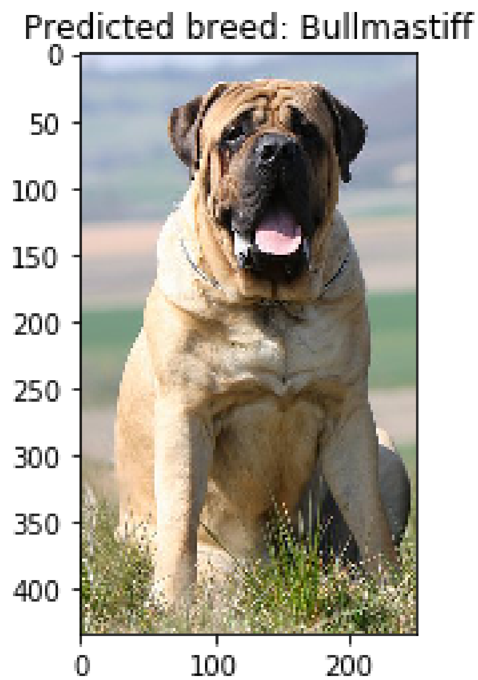

```
You most closely resemble-
American water spaniel
Hi, dog!
```




```
Your breed is-
Bullmastiff
Hi, dog!
```



```
Your breed is-
Mastiff
Hi, dog!
```




```
Your breed is-
Bullmastiff
```

In [ ]: