dog_app

February 21, 2019

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

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

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

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

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

The rubric contains *optional* "Stand Out Suggestions" for enhancing the project beyond the minimum requirements. If you decide to pursue the "Stand Out Suggestions", you should include the code in this Jupyter notebook.
## Step 0: Import Datasets
Make sure that you've downloaded the required human and dog datasets: * Download the dog dataset. Unzip the folder and place it in this project's home directory, at the location `/dog_images`.

- Download the human dataset. Unzip the folder and place it in the home directory, at location `/lfw`.

Note: If you are using a Windows machine, you are encouraged to use 7zip to extract the folder.

In the code cell below, we save the file paths for both the human (LFW) dataset and dog dataset in the numpy arrays `human_files` and `dog_files`.

```
In [190]: 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

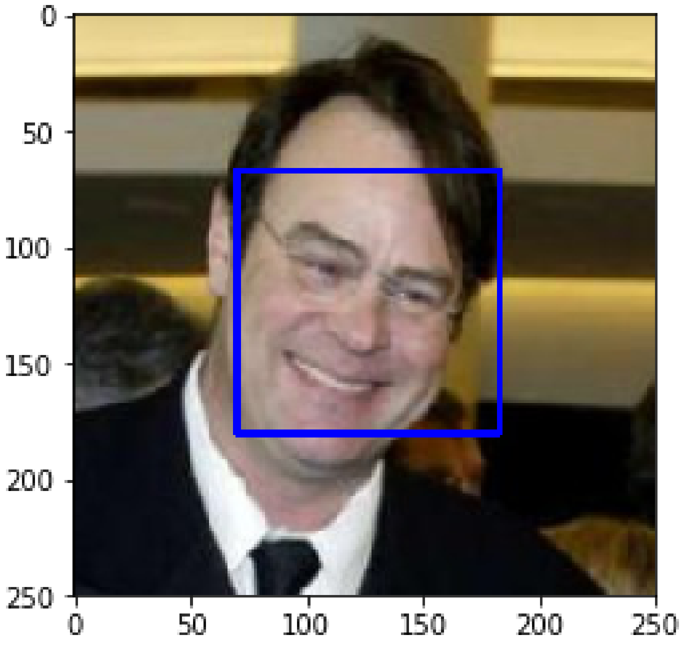
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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

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

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

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

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

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

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

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

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



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

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

1.1.1 Write a Human Face Detector

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

```
In [192]: # returns "True" if face is detected in image stored at img_path
          def face_detector(img_path):
```

```
    img = cv2.imread(img_path)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    faces = face_cascade.detectMultiScale(gray)
    return len(faces) > 0
```

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

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

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

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

```
In [193]: #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.
          no_of_faces_detected_in_human_files = 0
          for human_image in human_files_short:
              if face_detector(human_image):
                  no_of_faces_detected_in_human_files += 1

          no_of_faces_detected_in_dog_files = 0
          for dog_image in dog_files_short:
              if face_detector(dog_image):
                  no_of_faces_detected_in_dog_files += 1

          print("No of human images in which human face is detected are :"+str(no_of_faces_detec
          print("% of human images in which human face is detected are :"+str(1.*no_of_faces_det
          print("No of dog images in which human face is detected are :"+str(no_of_faces_detecte
          print("% of dog images in which human face is detected are :"+str(1.*no_of_faces_detec
```

```
No of human images in which human face is detected are :98
% of human images in which human face is detected are :0.98
No of dog images in which human face is detected are :17
% of dog images in which human face is detected are :0.17
```

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 [194]: ### (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 [195]: import torch
          import torchvision.models as models

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

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

          # move model to GPU if CUDA is available

          if use_cuda:
              VGG16 = VGG16.cuda()
```

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

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

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

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

```
In [196]: from PIL import Image
          import torchvision.transforms as transforms
          from torchvision import datasets
from torch.autograd import Variable

# Set PIL to be tolerant of image files that are truncated.
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

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
    normalize = transforms.Normalize(
            mean=[0.485, 0.456, 0.406],
            std=[0.229, 0.224, 0.225]
    )
    preprocess = transforms.Compose([
       transforms.Scale(256),
       transforms.CenterCrop(224),
       transforms.ToTensor(),
       normalize
    ])

    img = Image.open(img_path)
    img = preprocess(img)
    img = img.unsqueeze_(0)

    img_variable = Variable(img)
    if use_cuda:
        img_variable = img_variable.cuda()
    fc_out = VGG16(img_variable)

    return fc_out # predicted class index
```

1.1.5 (IMPLEMENTATION) Write a Dog Detector

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

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

```
In [197]: ### returns "True" if a dog is detected in the image stored at img_path
          def dog_detector_vgg16(img_path):
              ## TODO: Complete the function.
              prediction =  VGG16_predict(img_path)
              prediction = prediction.cpu().data.numpy().argmax()
              if prediction >=151 and prediction <=268:
                  return True
              return False # true/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 [198]: ### TODO: Test the performance of the dog_detector function
          ### on the images in human_files_short and dog_files_short.
          no_of_faces_detected_in_human_files = 0
          for human_image in human_files_short:
              if dog_detector_vgg16(human_image):
                  no_of_faces_detected_in_human_files += 1

          no_of_faces_detected_in_dog_files = 0
          for dog_image in dog_files_short:
              if dog_detector_vgg16(dog_image):
                  no_of_faces_detected_in_dog_files += 1

          print("No of human images in which dog is detected are :"+str(no_of_faces_detected_in_
          print("% of human images in which dog is detected are :"+str(1.*no_of_faces_detected_i
          print("No of dog images in which dog is detected are :"+str(no_of_faces_detected_in_do
          print("% of dog images in which dog is detected are :"+str(1.*no_of_faces_detected_in_
```

```
/opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-py3.6.egg/torchvision/transforms/transf
  "please use transforms.Resize instead.")


No of human images in which dog is detected are :0
% of human images in which dog is detected are :0.0
No of dog images in which dog is detected are :100
% of dog images in which dog is detected are :1.0
```

```
In [199]: dog_detector_vgg16(dog_files_short[0])
```

```
/opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-py3.6.egg/torchvision/transforms/transf
  "please use transforms.Resize instead.")
```

```
Out[199]: True
```

```
In [200]: VGG16_predict(dog_files_short[0]).cpu().data.numpy().argmax()
```

```
/opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-py3.6.egg/torchvision/transforms/transf
  "please use transforms.Resize instead.")
```

```
Out[200]: 243
```

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

          # define ResNet50 model
          resnet50 = models.resnet50(pretrained=True)

          if use_cuda:
              resnet50 = resnet50.cuda()

          def ResNet_50_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
              normalize = transforms.Normalize(
                      mean=[0.485, 0.456, 0.406],
                      std=[0.229, 0.224, 0.225]
              )
              preprocess = transforms.Compose([
            transforms.Scale(256),
            transforms.CenterCrop(224),
            transforms.ToTensor(),
            normalize
         ])

         img = Image.open(img_path)
         img_tensor = preprocess(img)
         img_tensor = img_tensor.unsqueeze_(0)

         img_variable = Variable(img_tensor)
         if use_cuda:
             img_variable = img_variable.cuda()
         fc_out = resnet50(img_variable)

         return fc_out # predicted class index
```

```
In [202]: def dog_detector_resnet50(img_path):
              ## TODO: Complete the function.
              prediction =  ResNet_50_predict(img_path)
              prediction = prediction.cpu().data.numpy().argmax()
              if prediction >=151 and prediction <=268:
                  return True
              return False # true/false
```

```
In [203]: dog_detector_resnet50(dog_files_short[8])
```

```
/opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-py3.6.egg/torchvision/transforms/transf
  "please use transforms.Resize instead.")
```

```
Out[203]: False
```

```
In [204]: ### Test the performance of the dog_detector function
          ### on the images in human_files_short and dog_files_short.
          no_of_faces_detected_in_human_files = 0
          for human_image in human_files_short:
              if dog_detector_resnet50(human_image):
                  no_of_faces_detected_in_human_files += 1

          no_of_faces_detected_in_dog_files = 0
          for dog_image in dog_files_short:
              if dog_detector_resnet50(dog_image):
                  no_of_faces_detected_in_dog_files += 1

          print("No of human images in which dog is detected are :"+str(no_of_faces_detected_in_
          print("% of human images in which dog is detected are :"+str(1.*no_of_faces_detected_i
          print("No of dog images in which dog is detected are :"+str(no_of_faces_detected_in_do
          print("% of dog images in which dog is detected are :"+str(1.*no_of_faces_detected_in_
/opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-py3.6.egg/torchvision/transforms/transf
  "please use transforms.Resize instead.")


No of human images in which dog is detected are :0
% of human images in which dog is detected are :0.0
No of dog images in which dog is detected are :0
% of dog images in which dog is detected are :0.0
```

## 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 [205]: import os
          from torchvision import datasets

          ### TODO: Write data loaders for training, validation, and test sets
          ## Specify appropriate transforms, and batch_sizes
          data_transforms = {
              'train': transforms.Compose([
                  transforms.RandomResizedCrop(224),
                  transforms.RandomHorizontalFlip(),
                  transforms.ToTensor(),
                  transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
              ]),
              'valid': transforms.Compose([
                  transforms.Resize(256),
                  transforms.CenterCrop(224),
                  transforms.ToTensor(),
                  transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
              ]),
              'test' : transforms.Compose([
                  transforms.Resize(256),
                  transforms.CenterCrop(224),
                  transforms.ToTensor(),
                  transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
              ]),
          }

          data_dir = '/data/dog_images'
          image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x),
                                                    data_transforms[x])
                            for x in ['train','valid','test']}
          loaders_scratch = {x: torch.utils.data.DataLoader(image_datasets[x], batch_size=16,
                                                       shuffle=True, num_workers=0)
                        for x in ['train','valid','test']}
          dataset_sizes = {x: len(image_datasets[x]) for x in ['train','valid','test']}
```

```
In [206]: print(dataset_sizes)
```

```
{'train': 6680, 'valid': 835, 'test': 836}
```

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?

My preprocessing of data includes resizing the images and then taking the center crop of the image. Since VGG takes 224 x 224 images as input, I have resized the images to a little higher dimension to 224 i.e 256 and then took the center crop of 224 x 224. From there I convert it to a tensor which is required for pytorch and then normalize using the values specified in the pytorch documentation. Coming to the augmention part, I have augmented the training data using RandomResizedCrop and RandomHorizontalFlip which does data augmentation by including randomization into the dataset so that it will not overfit to the training data and generalize well to data outside the training set.

Answer:

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [207]: 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
                  # convolutional layer (sees 224x224x3 image tensor)
                  self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
                  # convolutional layer (sees 112x112x3 image tensor)
                  self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
                  # convolutional layer (sees 56x56x3 image tensor)
                  self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
                  # convolutional layer (sees 28x28x3 image tensor)
                  self.conv4 = nn.Conv2d(64, 128, 3, padding=1)

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

                  # linear layer (128 * 14 * 14 -> 500)
                  self.fc1 = nn.Linear(128 * 14 * 14, 1000)
                  # linear layer (500 -> 10)
                  self.fc2 = nn.Linear(1000, 133)
                  self.dropout = nn.Dropout(0.25)


              def forward(self, x):
                  ## Define forward behavior
                  # add sequence of convolutional and max pooling layers
                  x = self.pool(F.relu(self.conv1(x)))
```

```
                x = self.pool(F.relu(self.conv2(x)))
                x = self.pool(F.relu(self.conv3(x)))
                x = self.pool(F.relu(self.conv4(x)))
                # flatten image input
                x = x.view(-1, 128 * 14 * 14)
                # add dropout layer
                x = self.dropout(x)
                # add 1st hidden layer, with relu activation function
                x = F.relu(self.fc1(x))
                # add dropout layer
                x = self.dropout(x)
                # add 2nd hidden layer, with relu activation function
                x = self.fc2(x)
                return x

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

        # instantiate the CNN
        model_scratch = Net()

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

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

Answer: Since the image size is 224 X 224 and since it needs to differentiate among 133 different set of classes, there is a need for complex architecture model than a simple architecture. So I have decided to go with 4 Conv layers followed by 2 fc layers. Choosing 2 fc layers in the end is pretty common standard followed in all of the famous architectures. I have decided to go with 4 Conv layers so that it would not be so simple that the network would not be able to find patterns so that it can classify 133 different classes and at the same time it is not too big to be trained in resonable amount of time on GPU's. I have also added dropout in the FC layers so that the network will not overfit especially with lot of paramenters(weights) in the fc layer and generalizes well to other data apart from the training data.

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 [208]: import torch.optim as optim

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

          ### TODO: select optimizer
          optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=0.06)
```

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 [209]: # the following import is required for training to be robust to truncated images
          from PIL import ImageFile
          ImageFile.LOAD_TRUNCATED_IMAGES = True

          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()
                      ## 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_l
                      # 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 parame
                      loss.backward()
                      # perform a single optimization step (parameter update)
                      optimizer.step()
                      # update training loss
                      train_loss += ((1 / (batch_idx + 1)) * (loss.data - 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()
                # 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 += ((1 / (batch_idx + 1)) * (loss.data - valid_loss))

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

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

        # return trained model
        return model


    # train the model
    model_scratch = train(20, loaders_scratch, model_scratch, optimizer_scratch,
                          criterion_scratch, use_cuda, 'model_scratch.pt')

    # load the model that got the best validation accuracy
    model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

```
Epoch: 1         Training Loss: 4.864626         Validation Loss: 4.826707
Validation loss decreased (inf --> 4.826707).  Saving model ...
Epoch: 2         Training Loss: 4.725391         Validation Loss: 4.564947
Validation loss decreased (4.826707 --> 4.564947).  Saving model ...
Epoch: 3         Training Loss: 4.619855         Validation Loss: 4.427682
Validation loss decreased (4.564947 --> 4.427682).  Saving model ...
Epoch: 4         Training Loss: 4.532874         Validation Loss: 4.351610
Validation loss decreased (4.427682 --> 4.351610).  Saving model ...
Epoch: 5         Training Loss: 4.496012         Validation Loss: 4.276646
Validation loss decreased (4.351610 --> 4.276646).  Saving model ...
Epoch: 6         Training Loss: 4.408587         Validation Loss: 4.187097
Validation loss decreased (4.276646 --> 4.187097).  Saving model ...
Epoch: 7         Training Loss: 4.356696         Validation Loss: 4.128305
```

```
Validation loss decreased (4.187097 --> 4.128305).  Saving model ...
Epoch: 8         Training Loss: 4.286060        Validation Loss: 4.160738
Epoch: 9         Training Loss: 4.223168        Validation Loss: 4.132854
Epoch: 10         Training Loss: 4.159286        Validation Loss: 4.044314
Validation loss decreased (4.128305 --> 4.044314).  Saving model ...
Epoch: 11         Training Loss: 4.143911        Validation Loss: 3.956875
Validation loss decreased (4.044314 --> 3.956875).  Saving model ...
Epoch: 12         Training Loss: 4.046477        Validation Loss: 3.944278
Validation loss decreased (3.956875 --> 3.944278).  Saving model ...
Epoch: 13         Training Loss: 3.996794        Validation Loss: 4.109175
Epoch: 14         Training Loss: 3.939649        Validation Loss: 3.774172
Validation loss decreased (3.944278 --> 3.774172).  Saving model ...
Epoch: 15         Training Loss: 3.873673        Validation Loss: 3.816752
Epoch: 16         Training Loss: 3.839475        Validation Loss: 4.040339
Epoch: 17         Training Loss: 3.764949        Validation Loss: 3.612114
Validation loss decreased (3.774172 --> 3.612114).  Saving model ...
Epoch: 18         Training Loss: 3.733633        Validation Loss: 4.089422
Epoch: 19         Training Loss: 3.684084        Validation Loss: 3.871180
Epoch: 20         Training Loss: 3.629574        Validation Loss: 3.794403
```

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 [210]: 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: 3.620356


Test Accuracy: 14% (119/836)
```

## 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 [211]: ## 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 [212]: import torchvision.models as models
          import torch.nn as nn

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

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

```
In [213]: model_transfer
```

```
Out[213]: 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=Tru
            (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=
                (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_stat
                (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_stat
                )
              )
              (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=
                (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_stat
                (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=
                (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_stat
                (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_stat
                (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bia
                (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stat
                (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_stat
                (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_stat
      )
    )
    (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_stat
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bia
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      (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_stat
      (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_stat
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bia
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      (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_stat
      (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_stat
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bia
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      (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_stat
      (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_stat
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bia
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      (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_sta
      (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_sta
      )
    )
    (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_stat
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bia
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      (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_sta
      (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_stat
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bia
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      (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_sta
      (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_stat
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bia
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      (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_sta
      (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_stat
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bia
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      (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_sta
      (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_stat
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bia
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      (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_sta
      (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_stat
          (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bia
          (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stat
          (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_sta
          (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_sta
          )
        )
        (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_stat
          (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bia
          (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stat
          (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_sta
          (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_stat
          (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bia
          (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stat
          (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_sta
          (relu): ReLU(inplace)
        )
      )
      (avgpool): AvgPool2d(kernel_size=7, stride=1, padding=0)
      (fc): Linear(in_features=2048, out_features=1000, bias=True)
    )
```

```
In [214]: for param in model_transfer.parameters():
              param.requires_grad = False

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

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

Question 5: Outline the steps you took to get to your final CNN architecture and your reasoning at each step. Describe why you think the architecture is suitable for the current problem.

Answer:

I have chosen resnet50, developed by Microsoft as it is one of the standard architectures and have shown to perform well on Imagenet dataset. Since the datasets are similar, we can be confident that the parameters learned for resnet on Imagenet dataset would generalie well for our dataset too. Infact any architecture trained on Imagenet dataset can be used here to apply transfer learning as Imagenet has dog Images. As the dataset is small and since dog pictures are already present in the Imagenet dataset on which the resent was trained on, we can freeze the conv layers and change the fc layer in the end and train only that. In order to freeze that I have set the requires grad parameter to false. And then I replaced the FC layer so that now it has 133 nodes in the end instead of having 1000. Then I have set the requires grad param for the fc layer to true so that these weights can be learned from training.

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 [215]: criterion_transfer = nn.CrossEntropyLoss()
          optimizer_transfer = optim.Adam(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 [216]: # train the model
          if use_cuda:
              model_transfer = model_transfer.cuda()

          n_epochs = 6
          model_transfer = train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer,

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

```
Epoch: 1         Training Loss: 2.524130         Validation Loss: 0.857702
Validation loss decreased (inf --> 0.857702).  Saving model ...
Epoch: 2         Training Loss: 1.301344         Validation Loss: 0.704424
Validation loss decreased (0.857702 --> 0.704424).  Saving model ...
Epoch: 3         Training Loss: 1.149909         Validation Loss: 0.575864
Validation loss decreased (0.704424 --> 0.575864).  Saving model ...
Epoch: 4         Training Loss: 1.085534         Validation Loss: 0.611777
Epoch: 5         Training Loss: 1.065360         Validation Loss: 0.563661
Validation loss decreased (0.575864 --> 0.563661).  Saving model ...
Epoch: 6         Training Loss: 0.981925         Validation Loss: 0.573086
```

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

```
Test Loss: 0.584704


Test Accuracy: 82% (690/836)
```

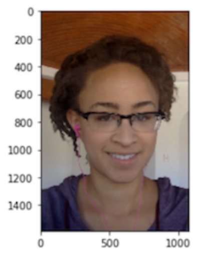
1.1.17 (IMPLEMENTATION) Predict Dog Breed with the Model

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 [218]:
```
### 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'].datase

import matplotlib.pyplot as plt

def predict_breed_transfer(model, class_names, img_path):
    # load the image and return the predicted breed
    normalize = transforms.Normalize(
            mean=[0.485, 0.456, 0.406],
            std=[0.229, 0.224, 0.225]
    )
    preprocess = transforms.Compose([
       transforms.Scale(256),
       transforms.CenterCrop(224),
       transforms.ToTensor(),
       normalize
    ])
    img = Image.open(img_path)
    plt.imshow(img)
    plt.show()
    img = preprocess(img)
    img = img.unsqueeze_(0)
    model = model.cpu()
    model.eval()
    idx = torch.argmax(model(img))
    return class_names[idx]
```

## Step 5: Write your Algorithm
Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then, - if a **dog** is detected in the image, return the predicted 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.

```
hello, human!
```



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

Sample Human Output

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 [219]: ### 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)
              if dog_detector_vgg16(img_path) is True:
                  prediction = predict_breed_transfer(model_transfer, class_names, img_path)
                  print("Dog is Detected and the breed is {0}".format(prediction))
              elif face_detector(img_path) > 0:
                  prediction = predict_breed_transfer(model_transfer, class_names, img_path)
                  print("Human face is detected. The resembling dog breed is a {0}".format(predi
              else:
                  print("Error...Cannot detect anything")
```

## Step 6: Test Your Algorithm

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

1.1.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

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

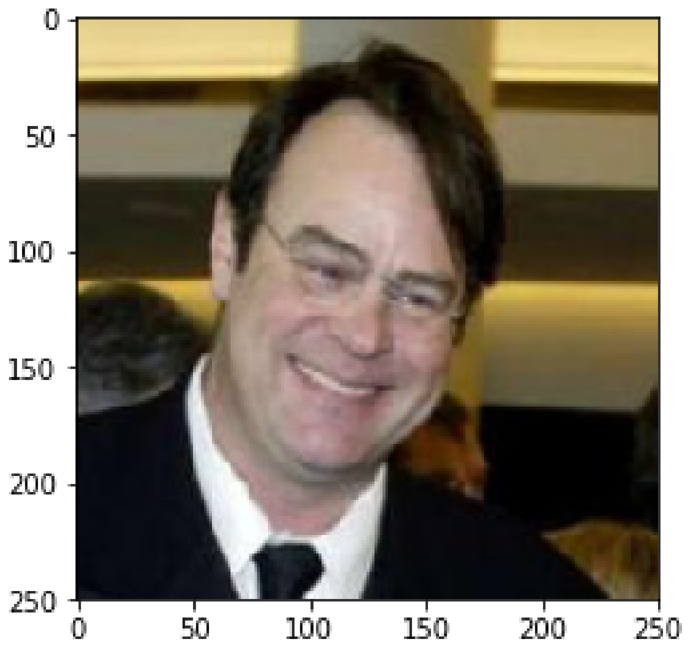
Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer: (Three possible points for improvement)

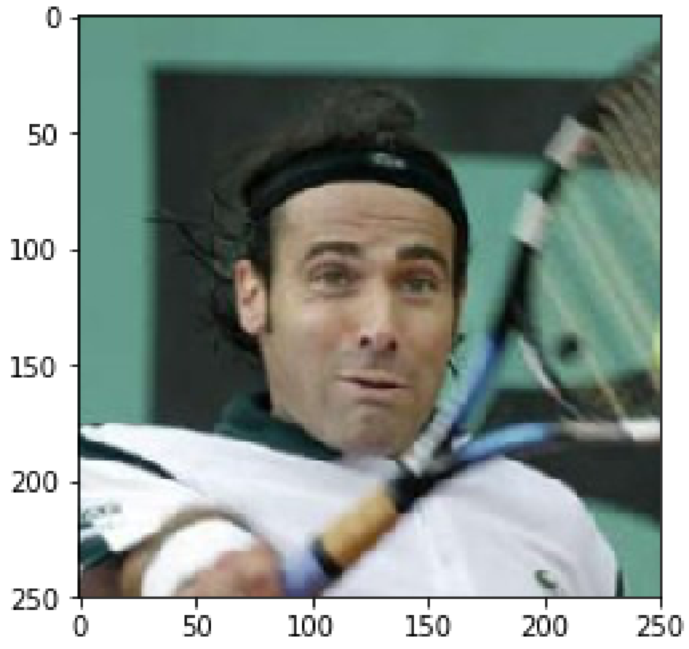
1) Having larger dataset, we can train the classic architectures from scratch that will give better performance as they learn specific filters to dataset rathern than generalized filters.
2) Tuning hyperparameters by using techniques like grid search and also experimenting out different optimizers.
3) More aggressive data augmentaion would give us better results.

```
In [221]: ## 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)
```

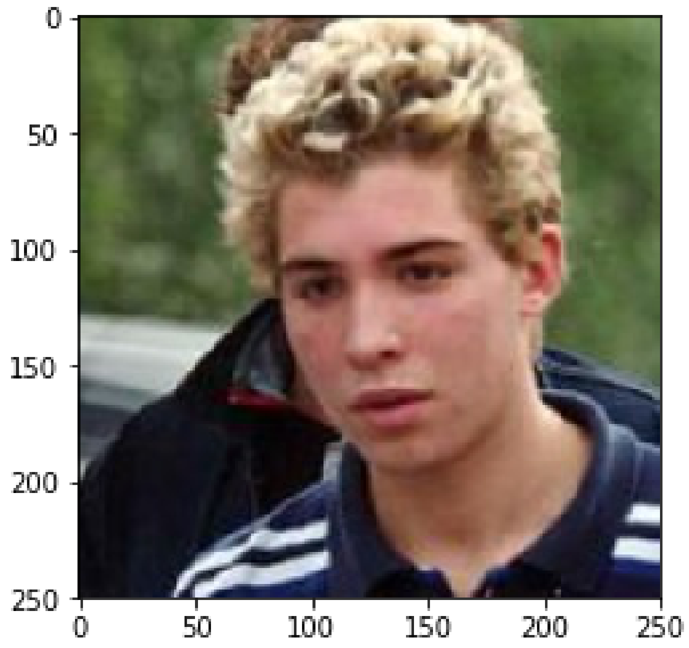
```
/opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-py3.6.egg/torchvision/transforms/transf
  "please use transforms.Resize instead.")
```



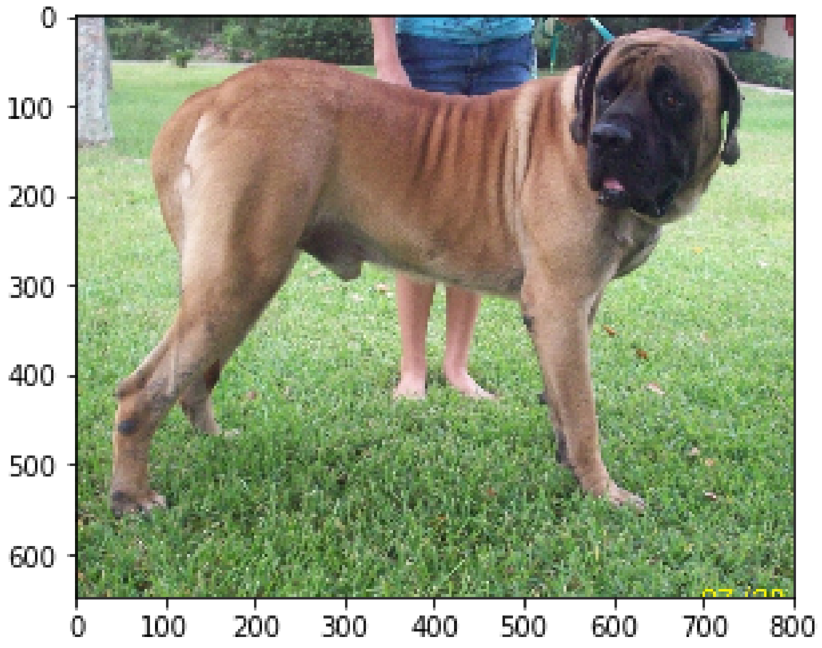
```
Human face is detected. The resembling dog breed is a Beagle
```



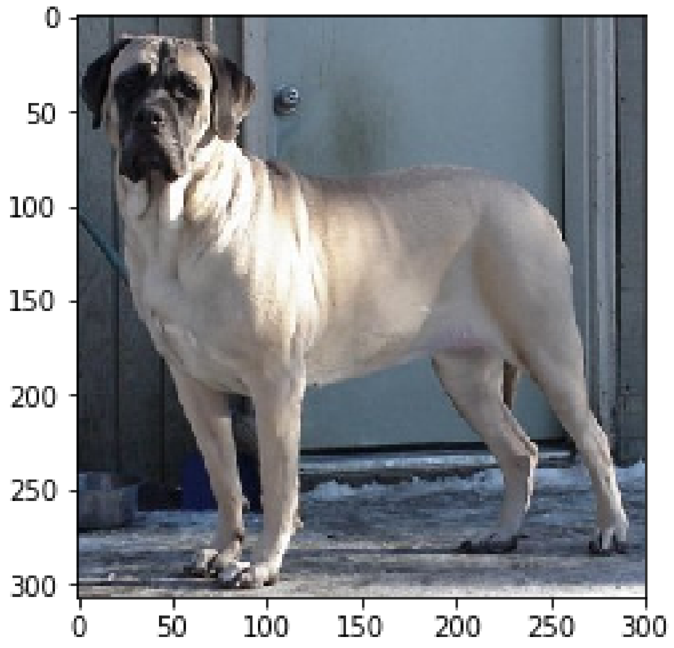
```
Human face is detected. The resembling dog breed is a Beagle
```



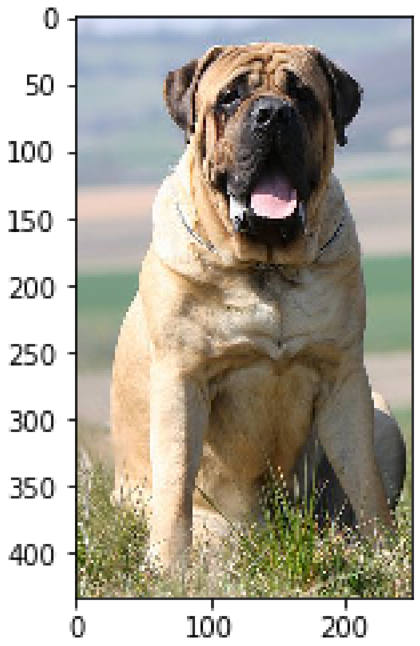
```
Human face is detected. The resembling dog breed is a American water spaniel
```



```
Dog is Detected and the breed is Mastiff
```



```
Dog is Detected and the breed is Mastiff
```



```
Dog is Detected and the breed is Mastiff
```