Dog-breed-classifier_app

May 28, 2020

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

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

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

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

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

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

Step 0: Import Datasets

Make sure that you've downloaded the required human and dog datasets:

Note: if you are using the Udacity workspace, you *DONOT* 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

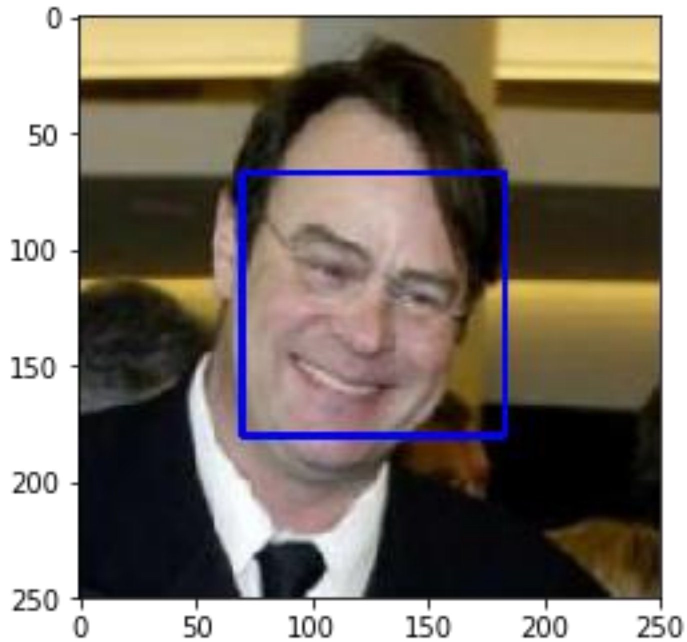
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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)) # 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: (You can print out your results and/or write your percentages in this cell)
Human faces correctly classified: 98%
Dog faces mistakenly classified as human faces: 17%

```
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.
        human_face_count = 0 dog_face_count =
        0

        for img in human_files_short:
            if face_detector(img) == True: human_face_count +=1

        for img in dog_files_short:
            if face_detector(img) == True:
                dog_face_count +=1
```

```
print ("Correctly Detected Human Faces: ", human_face_count) print ("Images
wrongly classified as human faces: ", dog_face_count) Correctly Detected Human
Faces: 98
Images wrongly classified as human faces: 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 [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 from torchvision.models.vgg import
        model_urls model_urls['vgg16'] = model_urls['vgg16'].replace('https://', 'http://')

        # 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: "http://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg1
100%|| 553433881/553433881 [00:13<00:00, 40024713.82it/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, ImageFile import
torchvision.transforms as transforms
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 image = Image.open(img_path).convert('RGB') normalize =

    transforms.Normalize(mean=[0.485, 0.456, 0.406],std=[0.229, 0.224, 0.225

    transformations = transforms.Compose([transforms.Resize(size=(224, 224)),
                                          transforms.ToTensor(), normalize])

    transformed_image = transformations(image)[:3,:,:].unsqueeze(0)

    if use_cuda:
        transformed_image = transformed_image.cuda() output =

    VGG16(transformed_image) return torch.max(output,1)[1].item() #

    predicted class index
```

1.1.5 (IMPLEMENTATION) Write a Dog Detector

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

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

```
In [8]: ### returns "True" if a dog is detected in the image stored at img_path def
        dog_detector(img_path): ## TODO: Complete the function.

            predicted_index = VGG16_predict(img_path) result = predicted_index

            >=151 and predicted_index <=268 return result # 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 dog_files_short, all dog faces are correctly detected - 100% In human_files_short, 1% of images is misclassified.

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

        dog_detector_dogs = 0 dog_detector_humans = 0

        for img in human_files_short: if
            dog_detector(img) == True:
            dog_detector_humans +=1

        for img in dog_files_short:
            if dog_detector(img) == True:
                dog_detector_dogs +=1

        print ("Correctly Detected Dog Faces: ", dog_detector_dogs) print ("Images wrongly
        classified in human faces: ", dog_detector_humans)
```

```
Correctly Detected Dog Faces: 100
Images wrongly classified in human faces: 1
```

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

batch_size = 20 num_workers = 0
data_dir = '/data/dog_images/'

train_path = data_dir + 'train' val_path =
data_dir + 'valid' test_path = data_dir + 'test'

normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

train_dataset = datasets.ImageFolder(train_path, transforms.Compose([
                        transforms.RandomResizedCrop(224),
                        transforms.RandomHorizontalFlip(),
                        transforms.RandomRotation(15),
                        transforms.ToTensor(), normalize,
                ]))

val_dataset = datasets.ImageFolder(val_path, transforms.Compose([
                        transforms.Resize(size=(224,224)),
                        transforms.ToTensor(), normalize,
                ]))

test_dataset = datasets.ImageFolder(test_path, transforms.Compose([
                         transforms.Resize(size=(224,224)),
                        transforms.ToTensor(), normalize,
                ]))

 train_loader = torch.utils.data.DataLoader(train_dataset, batch_size= batch_size, num_w val_loader =

       torch.utils.data.DataLoader(val_dataset, batch_size= batch_size, num_worke test_loader =

              torch.utils.data.DataLoader(test_dataset, batch_size= batch_size, num_wor


loaders_scratch = {
    'train': train_loader,
    'valid': val_loader,
```

```
'test': test_loader
}
```

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

Answer:

Most of the Pre trained models like VGG16 takes the size (224,224) as input, so I have used this size.

For train data, I have done image augmentation to avoid overfitting the model. Transforms used: Random resize crop to 224, random flipping and random rotation.

For validation and test data, I have done only image resizing.

I have applied normalization to all three datasets.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [12]: 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, 36, 3, padding=1) self.conv2 =
             nn.Conv2d(36, 64, 3, padding=1) self.conv3 = nn.Conv2d(64,
             128, 3, padding=1) self.pool = nn.MaxPool2d(2, 2) self.fc1 =
             nn.Linear(28*28*128, 512) self.fc2 = nn.Linear(512, 133)
             self.dropout = nn.Dropout(0.25) self.batch_norm =
             nn.BatchNorm1d(512) def forward(self, x):

                 x = self.pool(F.relu(self.conv1(x))) x =
                 self.pool(F.relu(self.conv2(x))) x =
                 self.pool(F.relu(self.conv3(x))) x = x.view(-1,
                 28*28*128)
                 x = F.relu(self.batch_norm(self.fc1(x))) x =
                 self.dropout(x) x = F.relu(self.fc2(x)) return x

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

```
# instantiate the CNN model_scratch =

Net() print(model_scratch)

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

```
Net(
  (conv1): Conv2d(3, 36, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(36, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=100352, out_features=512, bias=True)
  (fc2): Linear(in_features=512, out_features=133, bias=True)
  (dropout): Dropout(p=0.25)
  (batch_norm): BatchNorm1d(512, 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:

The model has 3 convolutional layers. All convolutional layers has kernal size of 3 and stride 1. The first conv layer (conv1) have in_channels =3 and the final conv layer (conv3) produces an output size of 128.

ReLU activation function is used here. The pooling layer of (2,2) is used which will reduce the input size by 2. We have two fully connected layers that finally produces 133 dimensional output.
A dropout of 0.25 is added to avoid overfitting.

1.1.9 (IMPLEMENTATION) Specify Loss Function and Optimizer

Use the next code cell to specify a loss function and optimizer. Save the chosen loss function as criterion_scratch, and the optimizer as optimizer_scratch below.

In [13]:
```
import torch.optim as optim

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

### TODO: select optimizer optimizer_scratch =
optim.SGD(model_scratch.parameters(), lr=0.02)
```

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 [14]: 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_lo

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

                ###################### # validate the model #
                ###################### model.eval() for batch_idx, (data, target) in
                enumerate(loaders['valid']):
                    # move to GPU
                    if use_cuda:
                        data, target = data.cuda(), target.cuda() ## update
                    the average validation loss output = model(data) loss =
                    criterion(output, target)
                                    valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_loss)

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

                ## TODO: save the model if validation loss has decreased if valid_loss <
                valid_loss_min:
```

```
print('Validation loss decreased ({:.6f} --> {:.6f}). Saving the model'.for
torch.save(model.state_dict(), save_path) valid_loss_min = valid_loss

# return trained model return
model


# train the model
model_scratch = train(15, 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: 4.826710    Validation Loss: 4.753974
Validation loss decreased (inf --> 4.753974). Saving the model
Epoch: 2    Training Loss: 4.685363    Validation Loss: 4.611388
Validation loss decreased (4.753974 --> 4.611388). Saving the model
Epoch: 3    Training Loss: 4.575179    Validation Loss: 4.515163
Validation loss decreased (4.611388 --> 4.515163). Saving the model
Epoch: 4    Training Loss: 4.480220    Validation Loss: 4.392666
Validation loss decreased (4.515163 --> 4.392666). Saving the model
Epoch: 5    Training Loss: 4.403322    Validation Loss: 4.324432
Validation loss decreased (4.392666 --> 4.324432). Saving the model
Epoch: 6    Training Loss: 4.336186    Validation Loss: 4.246064
Validation loss decreased (4.324432 --> 4.246064). Saving the model
Epoch: 7    Training Loss: 4.249739    Validation Loss: 4.170219
Validation loss decreased (4.246064 --> 4.170219). Saving the model
Epoch: 8    Training Loss: 4.201498    Validation Loss: 4.101915
Validation loss decreased (4.170219 --> 4.101915). Saving the model
Epoch: 9    Training Loss: 4.124438    Validation Loss: 4.049810
Validation loss decreased (4.101915 --> 4.049810). Saving the model
Epoch: 10    Training Loss: 4.047789    Validation Loss: 4.037409
Validation loss decreased (4.049810 --> 4.037409). Saving the model
Epoch: 11    Training Loss: 4.013179    Validation Loss: 4.087639
Epoch: 12    Training Loss: 3.928880    Validation Loss: 4.076640
Epoch: 13    Training Loss: 3.899233    Validation Loss: 3.808015
Validation loss decreased (4.037409 --> 3.808015). Saving the model
Epoch: 14    Training Loss: 3.825022    Validation Loss: 3.824240
Epoch: 15    Training Loss: 3.785158    Validation Loss: 3.807190
Validation loss decreased (3.808015 --> 3.807190). Saving the model
```

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 [15]:
```
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: 3.735869


Test Accuracy: 13% (112/836)
```

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 [16]: ## 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 [17]: import torchvision.models as models import torch.nn
         as nn

         ## TODO: Specify model architecture model_transfer =

         models.resnet101(pretrained=True)

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

```
Downloading: "https://download.pytorch.org/models/resnet101-5d3b4d8f.pth" to /root/.torch/models
100%|| 178728960/178728960 [00:02<00:00, 66387869.64it/s]
```

```
In [18]: 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)
  )
  (6): 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) )
  (7): 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)
  )
  (8): 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)
  )
  (9): 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)
  )
  (10): 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)
  )
  (11): 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)
  )
  (12): 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)
  )
  (13): 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)
  )
  (14): 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)
  )
  (15): 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)
  )
  (16): 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)
  )
  (17): 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)
  )
  (18): 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)
  )
  (19): 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)
  )
  (20): 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)
  )
```

```
(21): 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)
)
(22): 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=1000, bias=True)
)
```

```
In [19]: for param in model_transfer.parameters(): param.requires_grad =
             False model_transfer.fc = nn.Linear(2048, 133, bias=True)

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

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 decided to use the resnet101 architecture which is pre-trained on Imagenet dataset, The architecture is 101 layers deep, within just 5 epochs, the model got 81% accuracy. If we train for more epochs, the accuracy can be significantly improved. Steps:

1. Import pre-trained resnet101 model
2. Change the out_features of fully connected layer to 133 to solve the classification problem 3. CrossEntropy loss function is chosen as loss function.

Trained for 5 epochs and got 81% accuracy.

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 [20]: criterion_transfer = nn.CrossEntropyLoss() optimizer_transfer =
             optim.SGD(model_transfer.fc.parameters(), lr=0.02)
```

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 [21]: # train the model model_transfer = train(5, loaders_transfer, model_transfer, optimizer_transfer, criter
```

```
# 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: 3.544116 Validation Loss: 1.942998
Validation loss decreased (inf --> 1.942998). Saving the model
Epoch: 2 Training Loss: 2.061800 Validation Loss: 1.126173
Validation loss decreased (1.942998 --> 1.126173). Saving the model
Epoch: 3 Training Loss: 1.555040 Validation Loss: 0.886491
Validation loss decreased (1.126173 --> 0.886491). Saving the model
Epoch: 4 Training Loss: 1.378631 Validation Loss: 0.758096
Validation loss decreased (0.886491 --> 0.758096). Saving the model
Epoch: 5 Training Loss: 1.211058 Validation Loss: 0.682205
Validation loss decreased (0.758096 --> 0.682205). Saving the model

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 [22]:
```
test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

Test Loss: 0.721415

Test Accuracy: 81% (680/836)

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 [23]:
```
### TODO: Write a function that takes a path to an image as input ### and returns the
dog breed that is predicted by the model.
data_transfer = loaders_transfer
# 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 data_transfer['train'].dataset.cl

def predict_breed_transfer(img_path):
    # load the image and return the predicted breed image =
    Image.open(img_path).convert('RGB')
    normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],std=[0.229, 0.224, 0.22 transformations =
    transforms.Compose([transforms.Resize(size=(224, 224)),
                                         transforms.ToTensor(), normalize])
```

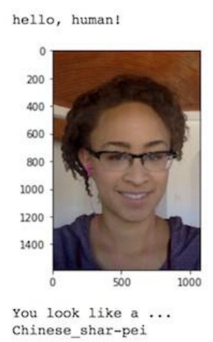
```
transformed_image = transformations(image)[:3,:,:].unsqueeze(0)

if use_cuda:
    transformed_image = transformed_image.cuda()

output = model_transfer(transformed_image) pred_index =

torch.max(output,1)[1].item() return

class_names[pred_index]
```

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!



Sample Human Output

1.1.18 (IMPLEMENTATION) Write your Algorithm

```
In [24]: ### TODO: Write your algorithm.
         ### Feel free to use as many code cells as needed.
         def load_image(img_path):
             img = Image.open(img_path)
             plt.imshow(img) plt.show()

         def run_app(img_path):
             ## handle cases for a human face, dog, and neither
```

```
if face_detector(img_path): print
    ("Hello Human!")
    predicted_breed = predict_breed_transfer(img_path) print("Predicted breed:
    ",predicted_breed) load_image(img_path)

elif dog_detector(img_path): print
    ("Hello Dog!")
    predicted_breed = predict_breed_transfer(img_path) print("Predicted breed:
    ",predicted_breed) load_image(img_path)

else: print ("Invalid 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.

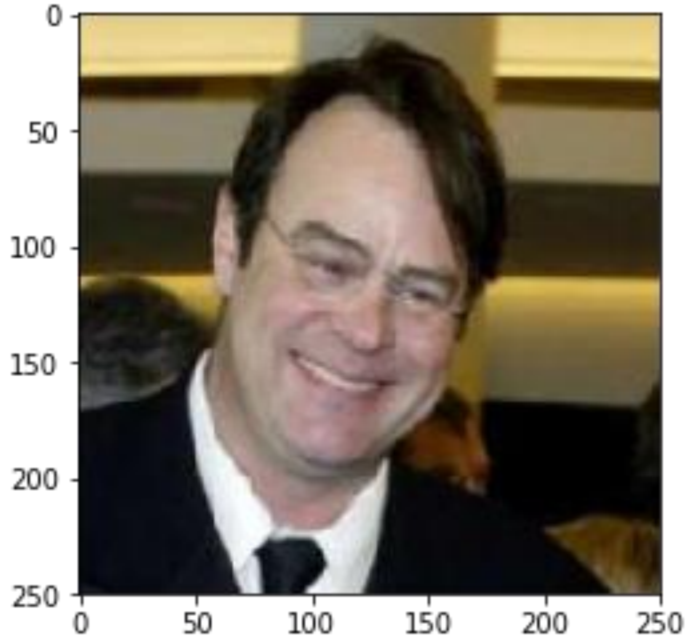
Answer: (Three possible points for improvement)

I think the model created using transfer learning performed very well. Improvement areas:

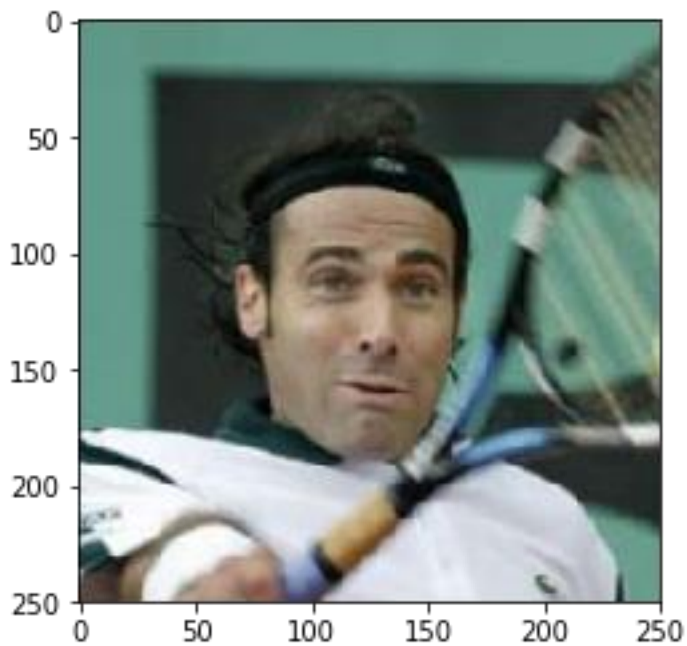
1. More training data will help in model improvement.
2. Hyper parameter tuning will also help in improving performance.
3. More image augmentation can be tried to improve accuracy.

```
In [25]: ## 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)
```

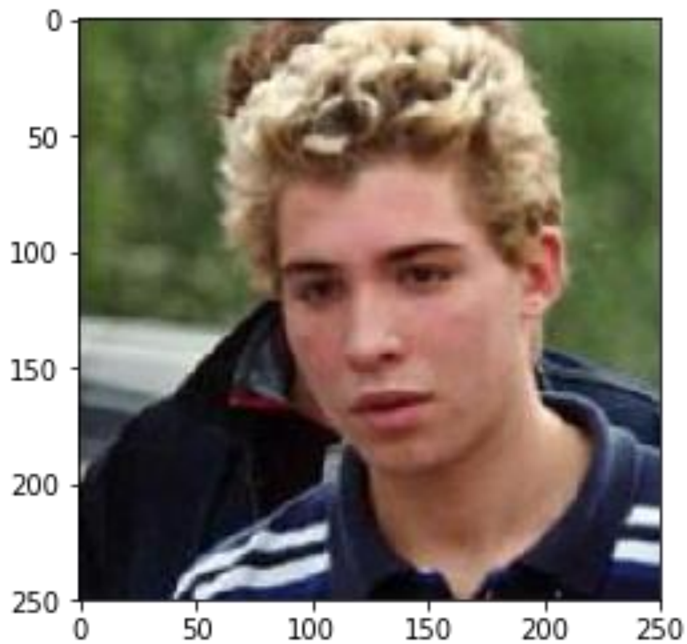
```
Hello Human!
Predicted breed: Dachshund
```



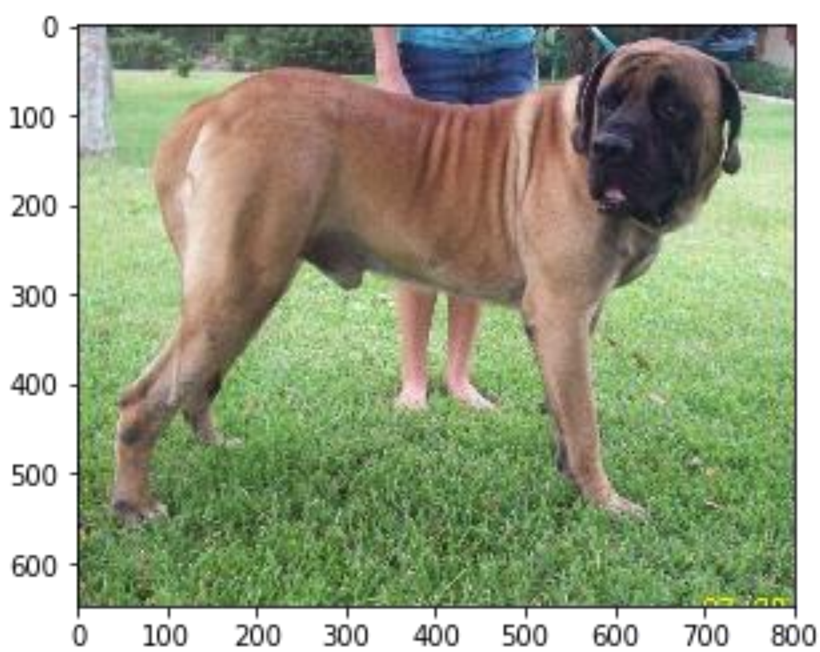
Hello Human!
Predicted breed: Parson russell terrier



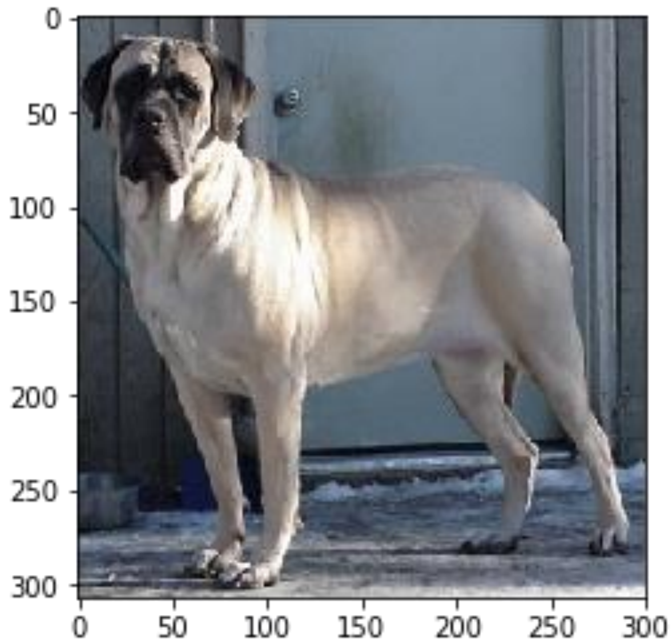
Hello Human!
Predicted breed: German shepherd dog



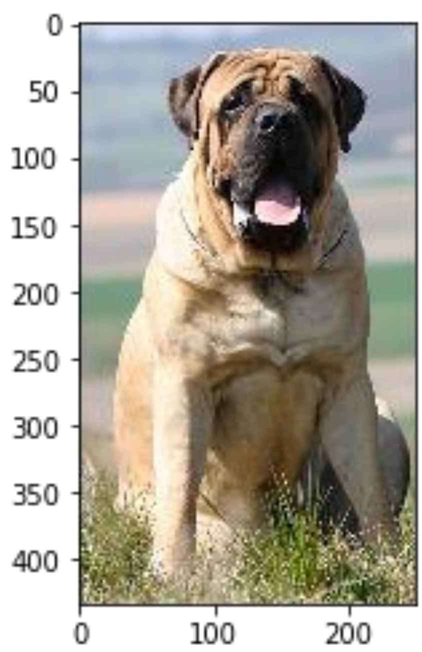
Hello Dog! Predicted breed:
Bullmastiff



Hello Dog!
Predicted breed: Mastiff



Hello Dog! Predicted breed:
Mastiff



References:

1. Original repo for Project - GitHub: https://github.com/udacity/deep-learning-v2pytorch/blob/master/project-dog-classification/
2. Resnet101: https://pytorch.org/docs/stable/_modules/torchvision/models/resnet.html#resnet101
3. Imagenet training in Pytorch: https://github.com/pytorch/examples/blob/97304e232807082c2e7b54c5976 L198
4. Pytorch Documentation: https://pytorch.org/docs/master/