dog_app

May 11, 2020

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

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

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

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

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

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

Step 0: Import Datasets

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

Note: if you are using the Udacity workspace, you *DO NOT* need to re-download these - they can be found in the `/data` folder as noted in the cell below.

- Download the dog dataset. Unzip the folder and place it in this project's home directory, at the location `/dog_images`.
- Download the human dataset. Unzip the folder and place it in the home directory, at location `/lfw`.

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

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

```
In [36]: 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

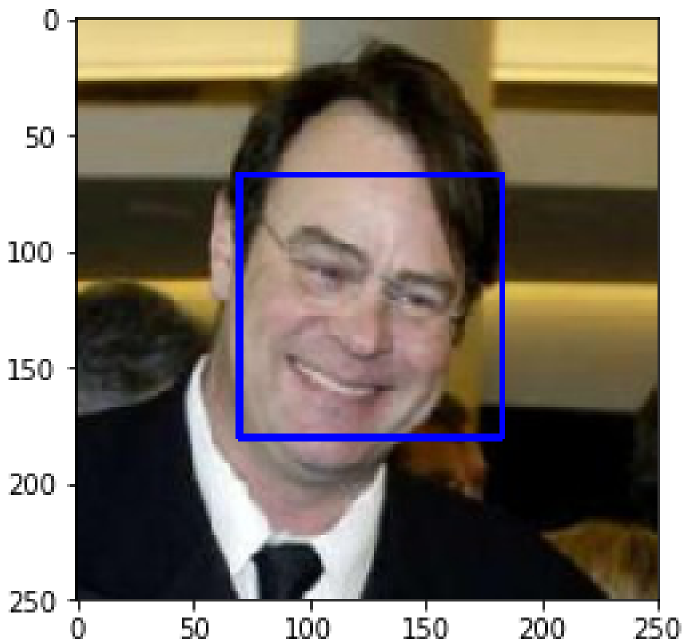
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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:
Percentage of faces in Human Images: 98.00%
Percentage of faces in Dog Images: 17.00%

```
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. #-#-#
        def face_detection(images):
            count = 0
            for image in tqdm(images):
                if face_detector(image):
                    count += 1
            return count
        ## TODO: Test the performance of the face_detector algorithm
        ## on the images in human_files_short and dog_files_short.
        print("Percentage of faces in Human Images: {}%".format(face_detection(human_files_short
        print("Percentage of faces in Dog Images: {}%".format(face_detection(dog_files_short)))
```

```
100%|| 100/100 [00:06<00:00, 15.47it/s]
  0%|          | 0/100 [00:00<?, ?it/s]

Percentage of faces in Human Images: 98%


100%|| 100/100 [00:47<00:00,  2.12it/s]
```

```
Percentage of faces in Dog Images: 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

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

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

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

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:05<00:00, 93014219.90it/s]
```

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

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

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

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

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

#function to load the image
def load_image(img_path):
    image = Image.open(img_path).convert('RGB') #converting to RGB
    transform = transforms.Compose([ #converting to normarilzed torch.tensor
    transforms.Scale(224),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(
    mean = [0.485, 0.456, 0.406],
    std = [0.229, 0.224, 0.225])
    ])
    return transform(image)[:3,:,:].unsqueeze(0)


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 = load_image(img_path)
    if use_cuda:
        image = image.cuda()

    outputs = VGG16(image) #getting predictions
    return torch.max(outputs, 1)[1].item() # predicted class index by argmax
```

1.1.5 (IMPLEMENTATION) Write a Dog Detector

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

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

```
In [8]: ### returns "True" if a dog is detected in the image stored at img_path
        def dog_detector(img_path):
            ## TODO: Complete the function.
            index = VGG16_predict(img_path)
            # Checking index number between 151 and 268
            return (index >= 151 and index <= 268) # 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:
Percentage of dogs in Human Images: 0%
Percentage of dogs in Dog Images: 100%

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

        def dog_detection_count(images):
            count = 0
            for image in tqdm(images):
                if dog_detector(image):
                    count += 1
            return count

        print('Percentage of dogs in Human Images: {}%'.format(dog_detection_count(human_files_s
        print("Percentage of dogs in Dog Images: {}%".format(dog_detection_count(dog_files_short
```

```
  0%|          | 0/100 [00:00<?, ?it/s]/opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-
100%|| 100/100 [00:03<00:00, 29.06it/s]
  3%|         | 3/100 [00:00<00:03, 25.95it/s]

Percentage of dogs in Human Images: 0%


100%|| 100/100 [00:04<00:00, 25.20it/s]

Percentage of dogs in Dog Images: 100%
```

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

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

import torch
import torchvision.models as models

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

if use_cuda:
    ResNet50 = ResNet50.cuda()

def Resnet(img_path):
    image = load_image(img_path)
    ResNet50.eval()
    if use_cuda:
        image = image.cuda()

    output = ResNet50(image) #getting predictions
    _, index = torch.max(output, 1)
    return index

def resnet_detector(img_path):
    index = Resnet(img_path)
    return (index>= 151 and index <= 268)


def resnet_detection_count(images):
    count = 0
    for image in tqdm(images):
        if resnet_detector(image):
            count += 1
    return count

print('Percentage of dogs in Human Images: {}%'.format(resnet_detection_count(human_fil
print("Percentage of dogs in Dog Images: {}%".format(resnet_detection_count(dog_files_s
```

```
Downloading: "https://download.pytorch.org/models/resnet50-19c8e357.pth" to /root/.torch/models/
100%|| 102502400/102502400 [00:01<00:00, 88151841.22it/s]
```

```
  0%|          | 0/100 [00:00<?, ?it/s]/opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-
100%|| 100/100 [00:02<00:00, 47.60it/s]
  0%|          | 0/100 [00:00<?, ?it/s]

Percentage of dogs in Human Images: 0%


100%|| 100/100 [00:03<00:00, 27.71it/s]

Percentage of dogs in Dog Images: 100%
```

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, transforms
         from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True
         use_cuda = torch.cuda.is_available()

         ### TODO: Write data loaders for training, validation, and test sets
         ## Specify appropriate transforms, and batch_sizes
         batch_size = 16
         num_workers = 2

         #loading data
         dog_file='/data/dog_images/'

         train=os.path.join(dog_file,'train')
         valid=os.path.join(dog_file,'valid')
         test=os.path.join(dog_file,'test')

         Image_transformation = {
             'train' : transforms.Compose([
             transforms.Resize([256, 256]),
             transforms.RandomCrop(224),
             transforms.ColorJitter(hue=.05, saturation=.05),
             transforms.RandomHorizontalFlip(),
             transforms.RandomRotation(20, resample=Image.BILINEAR),
             transforms.ToTensor(),
             transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                  std=[0.229, 0.224, 0.225])]),
             'test': transforms.Compose([
             transforms.Resize((224,224)),
             transforms.ToTensor(),
             transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                  std=[0.229, 0.224, 0.225])])}

         train_data=datasets.ImageFolder(train,transform=Image_transformation['train'])
         valid_data=datasets.ImageFolder(valid,transform=Image_transformation['test'])
         test_data=datasets.ImageFolder(test,transform=Image_transformation['test'])
```

```
loaders_scratch={
    'train':torch.utils.data.DataLoader(train_data, batch_size=batch_size, num_workers=
    'valid':torch.utils.data.DataLoader(valid_data,batch_size=batch_size, num_workers=n
    'test': torch.utils.data.DataLoader(test_data,batch_size=batch_size, num_workers=nu
}
```

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

Answer: - I resized the image first to 256 to maintain more information before cropping it to 224 and Randomcrop provides more randomness. - I decided to augument the dataset using RandomHorizontalFLip and RandomRotation for this model

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
                 # convolutional layer
                 self.conv1 = nn.Conv2d(3, 32, 3, padding = 1)
                 self.conv2 = nn.Conv2d(32, 32, 3, padding= 1)
                 self.conv3 = nn.Conv2d(32, 64, 3, padding= 1)

                 self.dropout = nn.Dropout(p= 0.25)
                 self.pool = nn.MaxPool2d(2,2)

                 self.fc1 = nn.Linear(64*28*28, 64*28)
                 self.fc2 = nn.Linear(64*28, 64*14)
                 self.fc3 = nn.Linear(64*14, 64*7)
                 self.fc4 = nn.Linear(64*7, 133)


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

```
        x = x.view(x.shape[0],-1)

        x = self.dropout(F.relu(self.fc1(x)))
        x = self.dropout(F.relu(self.fc2(x)))
        x = F.relu(self.fc3(x))
        x = self.fc4(x)

        return x

#-#-# You so 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()
print(model_scratch)
```

```
Net(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (dropout): Dropout(p=0.25)
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=50176, out_features=1792, bias=True)
  (fc2): Linear(in_features=1792, out_features=896, bias=True)
  (fc3): Linear(in_features=896, out_features=448, bias=True)
  (fc4): Linear(in_features=448, out_features=133, bias=True)
)
```

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

Answer: - I used three 2Dconv layers with kernel_size = 3x3, stride = 1 and padding = 1 - Applied Maxpool layer for conv1, conv2 and conv3 to reduce the size and flaten the images to get fully connected layers - Created 3 fully connected layers along with droupout rate of 0.25 for first fully connected layer

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=0.01, momentum=0.9)
```

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()
                 #print(use_cuda)
                 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_scratch.zero_grad()
                     output = model(data)
                     loss = criterion_scratch(output, target)
                     loss.backward()
                     optimizer_scratch.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_scratch(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 model ...'.fo
                torch.save(model.state_dict(), 'model_scratch.pt')
                valid_loss_min = valid_loss
        # return trained model
        return model
```

In [15]:
```
# train the model
model_scratch = train(30, 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: 4.857310         Validation Loss: 4.754003
Validation loss decreased (inf --> 4.754003).  Saving model ...
Epoch: 2         Training Loss: 4.693100         Validation Loss: 4.600338
Validation loss decreased (4.754003 --> 4.600338).  Saving model ...
Epoch: 3         Training Loss: 4.566752         Validation Loss: 4.522989
Validation loss decreased (4.600338 --> 4.522989).  Saving model ...
Epoch: 4         Training Loss: 4.423067         Validation Loss: 4.432785
Validation loss decreased (4.522989 --> 4.432785).  Saving model ...
Epoch: 5         Training Loss: 4.334473         Validation Loss: 4.297378
Validation loss decreased (4.432785 --> 4.297378).  Saving model ...
Epoch: 6         Training Loss: 4.251721         Validation Loss: 4.206086
Validation loss decreased (4.297378 --> 4.206086).  Saving model ...
Epoch: 7         Training Loss: 4.189883         Validation Loss: 4.098696
Validation loss decreased (4.206086 --> 4.098696).  Saving model ...
Epoch: 8         Training Loss: 4.122212         Validation Loss: 4.119152
Epoch: 9         Training Loss: 4.081985         Validation Loss: 4.185503
Epoch: 10         Training Loss: 4.000071         Validation Loss: 4.045099
Validation loss decreased (4.098696 --> 4.045099).  Saving model ...
Epoch: 11         Training Loss: 3.958002         Validation Loss: 3.998131
Validation loss decreased (4.045099 --> 3.998131).  Saving model ...
Epoch: 12         Training Loss: 3.903251         Validation Loss: 4.035226
Epoch: 13         Training Loss: 3.844988         Validation Loss: 3.903144
Validation loss decreased (3.998131 --> 3.903144).  Saving model ...
Epoch: 14         Training Loss: 3.794073         Validation Loss: 3.903510
```

```
Epoch: 15         Training Loss: 3.730035        Validation Loss: 3.832015
Validation loss decreased (3.903144 --> 3.832015).  Saving model ...
Epoch: 16         Training Loss: 3.664619        Validation Loss: 3.856458
Epoch: 17         Training Loss: 3.633093        Validation Loss: 3.815004
Validation loss decreased (3.832015 --> 3.815004).  Saving model ...
Epoch: 18         Training Loss: 3.582412        Validation Loss: 4.229582
Epoch: 19         Training Loss: 3.583673        Validation Loss: 3.696789
Validation loss decreased (3.815004 --> 3.696789).  Saving model ...
Epoch: 20         Training Loss: 3.534974        Validation Loss: 3.853058
Epoch: 21         Training Loss: 3.463275        Validation Loss: 3.696544
Validation loss decreased (3.696789 --> 3.696544).  Saving model ...
Epoch: 22         Training Loss: 3.419765        Validation Loss: 3.663097
Validation loss decreased (3.696544 --> 3.663097).  Saving model ...
Epoch: 23         Training Loss: 3.402473        Validation Loss: 3.741365
Epoch: 24         Training Loss: 3.392871        Validation Loss: 3.655990
Validation loss decreased (3.663097 --> 3.655990).  Saving model ...
Epoch: 25         Training Loss: 3.332924        Validation Loss: 3.820809
Epoch: 26         Training Loss: 3.274928        Validation Loss: 3.986592
Epoch: 27         Training Loss: 3.269442        Validation Loss: 3.684378
Epoch: 28         Training Loss: 3.228619        Validation Loss: 3.586563
Validation loss decreased (3.655990 --> 3.586563).  Saving model ...
Epoch: 29         Training Loss: 3.213064        Validation Loss: 3.599514
Epoch: 30         Training Loss: 3.190086        Validation Loss: 3.799192
```

1.1.11 (IMPLEMENTATION) Test the Model

Try out your model on the test dataset of dog images. Use the code cell below to calculate and print the test loss and accuracy. Ensure that your test accuracy is greater than 10%.

In [16]:
```python
def test(loaders, model, criterion, use_cuda):

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

    model.eval()
    for batch_idx, (data, target) in enumerate(loaders['test']):
        # move to GPU
        if use_cuda:
            data, target = data.cuda(), target.cuda()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the loss
        loss = criterion(output, target)
        # update average test loss
        test_loss = 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: 3.576264


Test Accuracy: 15% (130/836)
```

Step 4: Create a CNN to Classify Dog Breeds (using Transfer Learning)

You will now use transfer learning to create a CNN that can identify dog breed from images. Your CNN must attain at least 60% accuracy on the test set.

1.1.12 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

Use the code cell below to write three separate data loaders for the training, validation, and test datasets of dog images (located at `dogImages/train`, `dogImages/valid`, and `dogImages/test`, respectively).

If you like, **you are welcome to use the same data loaders from the previous step**, when you created a CNN from scratch.

```
In [17]: ## 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 [18]: import torchvision.models as models
         import torch.nn as nn

         ## TODO: Specify model architecture



         model_transfer = models.resnet50(pretrained= True)
```

```
for param in model_transfer.parameters():
    param.requires_grad = False
```

In [19]:
```
inputs = model_transfer.fc.in_features

model_transfer.fc = nn.Sequential(
    nn.Linear(inputs, 1024),
    nn.ReLU(),
    nn.Dropout(0.4),
    nn.Linear(1024, 133),
)

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

Out[19]:
```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
  (relu): ReLU(inplace)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (relu): ReLU(inplace)
      (downsample): Sequential(
        (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (relu): ReLU(inplace)
    )
    (2): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
```

```
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (relu): ReLU(inplace)
    )
  )
  (layer2): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (relu): ReLU(inplace)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (relu): ReLU(inplace)
    )
    (2): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (relu): ReLU(inplace)
    )
    (3): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (relu): ReLU(inplace)
    )
  )
```

```
(layer3): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
    (relu): ReLU(inplace)
    (downsample): Sequential(
      (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
    )
  )
  (1): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
    (relu): ReLU(inplace)
  )
  (2): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
    (relu): ReLU(inplace)
  )
  (3): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
    (relu): ReLU(inplace)
  )
  (4): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
```

```
      (relu): ReLU(inplace)
    )
    (5): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      (relu): ReLU(inplace)
    )
  )
  (layer4): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      (relu): ReLU(inplace)
      (downsample): Sequential(
        (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      (relu): ReLU(inplace)
    )
    (2): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      (relu): ReLU(inplace)
    )
  )
  (avgpool): AvgPool2d(kernel_size=7, stride=1, padding=0)
  (fc): Sequential(
    (0): Linear(in_features=2048, out_features=1024, bias=True)
```

```
        (1): ReLU()
        (2): Dropout(p=0.4)
        (3): Linear(in_features=1024, out_features=133, bias=True)
      )
    )
```

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

Answer: - i'm using Resnet50 model as transfer learning model as it is so good at image classification and set all traning parameters to flase. I only changed the last fully connected layer to our need and dropout as Resnet is so huge

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]: import torch.optim as optim

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

         ### TODO: select optimizer
         optimizer_transfer = optim.SGD(model_transfer.fc.parameters(),lr=0.001, momentum=0.9)
```

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]: 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()
                 #print(use_cuda)
                 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_transfer.zero_grad()
            output = model(data)
            loss = criterion_transfer(output, target)
            loss.backward()
            optimizer_transfer.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_transfer(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 model ...'.fo
            torch.save(model.state_dict(), save_path)
            valid_loss_min = valid_loss
    # return trained model
    return model


# train the model
model_transfer = train(20, loaders_transfer, model_transfer, optimizer_transfer,
                      criterion_transfer, use_cuda, 'model_tranfser.pt')
# 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: 4.561423         Validation Loss: 3.926649
Validation loss decreased (inf --> 3.926649).  Saving model ...
```

```
Epoch: 2 	Training Loss: 3.397151 	Validation Loss: 2.369460
Validation loss decreased (3.926649 --> 2.369460).  Saving model ...
Epoch: 3 	Training Loss: 2.212339 	Validation Loss: 1.428512
Validation loss decreased (2.369460 --> 1.428512).  Saving model ...
Epoch: 4 	Training Loss: 1.585846 	Validation Loss: 1.028878
Validation loss decreased (1.428512 --> 1.028878).  Saving model ...
Epoch: 5 	Training Loss: 1.259397 	Validation Loss: 0.854684
Validation loss decreased (1.028878 --> 0.854684).  Saving model ...
Epoch: 6 	Training Loss: 1.110003 	Validation Loss: 0.752263
Validation loss decreased (0.854684 --> 0.752263).  Saving model ...
Epoch: 7 	Training Loss: 0.995168 	Validation Loss: 0.684699
Validation loss decreased (0.752263 --> 0.684699).  Saving model ...
Epoch: 8 	Training Loss: 0.914801 	Validation Loss: 0.664689
Validation loss decreased (0.684699 --> 0.664689).  Saving model ...
Epoch: 9 	Training Loss: 0.861524 	Validation Loss: 0.612512
Validation loss decreased (0.664689 --> 0.612512).  Saving model ...
Epoch: 10 	Training Loss: 0.833505 	Validation Loss: 0.576190
Validation loss decreased (0.612512 --> 0.576190).  Saving model ...
Epoch: 11 	Training Loss: 0.769140 	Validation Loss: 0.578085
Epoch: 12 	Training Loss: 0.737211 	Validation Loss: 0.537752
Validation loss decreased (0.576190 --> 0.537752).  Saving model ...
Epoch: 13 	Training Loss: 0.722796 	Validation Loss: 0.522449
Validation loss decreased (0.537752 --> 0.522449).  Saving model ...
Epoch: 14 	Training Loss: 0.722222 	Validation Loss: 0.533125
Epoch: 15 	Training Loss: 0.672579 	Validation Loss: 0.498869
Validation loss decreased (0.522449 --> 0.498869).  Saving model ...
Epoch: 16 	Training Loss: 0.644036 	Validation Loss: 0.510983
Epoch: 17 	Training Loss: 0.661668 	Validation Loss: 0.489747
Validation loss decreased (0.498869 --> 0.489747).  Saving model ...
Epoch: 18 	Training Loss: 0.626894 	Validation Loss: 0.501621
Epoch: 19 	Training Loss: 0.612229 	Validation Loss: 0.498952
Epoch: 20 	Training Loss: 0.621403 	Validation Loss: 0.495707
```

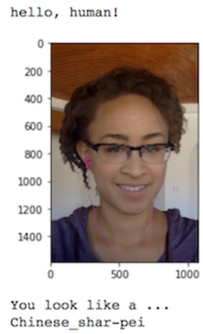
```
In [23]: model_transfer.load_state_dict(torch.load('model_tranfser.pt'))
```

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

```
Test Loss: 0.491510


Test Accuracy: 85% (713/836)
```



Sample Human Output

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

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

         def predict_breed_transfer(img_path):
             # load the image and return the predicted breed
             image = load_image(img_path)
             model = model_transfer.cpu()
             model.eval()
             idx = torch.argmax(model(image))
             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.

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

```
def run_app(img_path):
    ## handle cases for a human face, dog, and neither
    img = Image.open(img_path)
    plt.imshow(img)
    plt.show()
    if dog_detector(img_path):
        print('Dog detected!, Breed is ', end=' ')
        breed = predict_breed_transfer(img_path)
        print(breed)
    elif face_detector(img_path):
        print('Human detected!\n Looks like: ', end=' ')
        breed = predict_breed_transfer(img_path)
        print(breed)
    else:
        print('Neither Dog nor Human')
```

Step 6: Test Your Algorithm

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

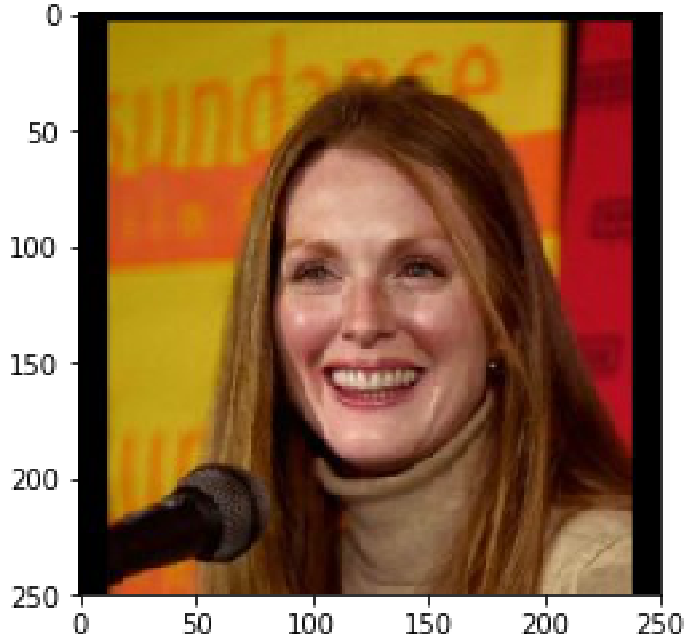
1.1.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

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

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

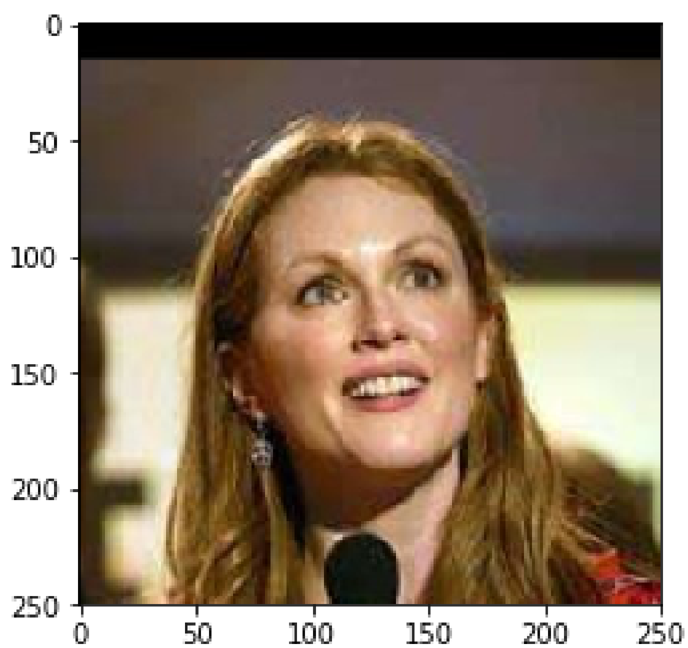
Answer: (Three possible points for improvement) - More balanced data can improve the results - Better augmentaion can also improve its performance - Hyper parameters tunning and Ensembles of models could do wonders

```
In [45]: ## 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[57:60], dog_files[57:60])):
             run_app(file)
```

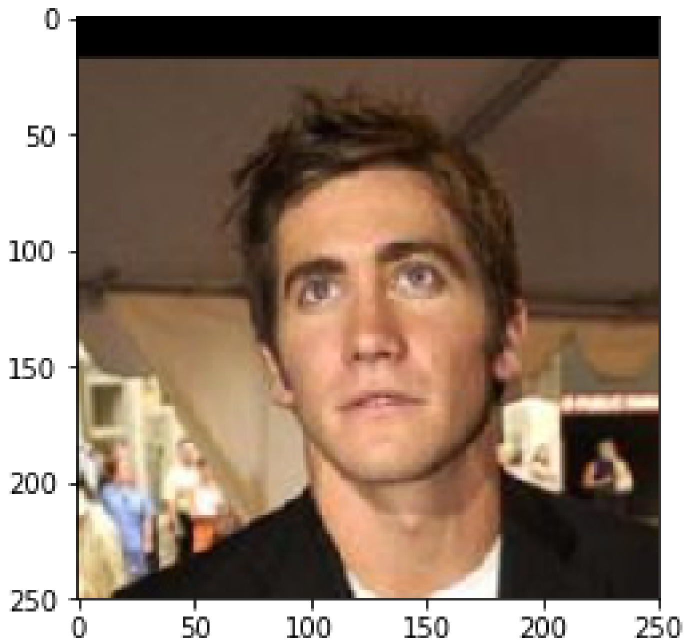



```
/opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-py3.6.egg/torchvision/transforms/transf
```

```
Human detected!
 Looks like:  Dachshund
```

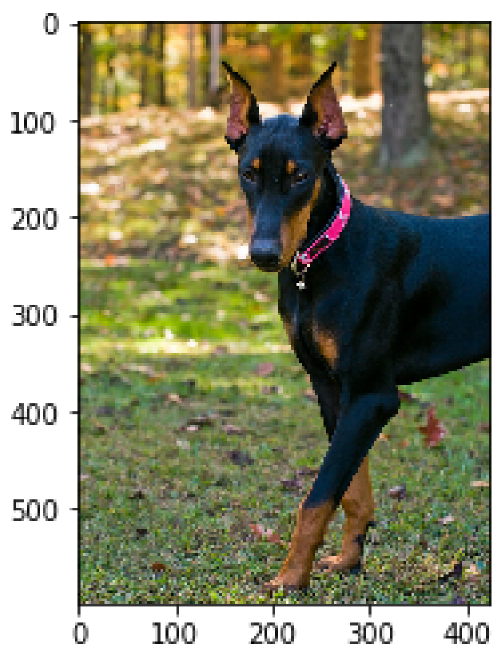


```
Human detected!
 Looks like:  Dachshund
```




```
Human detected!
 Looks like:  American water spaniel
```


```
Dog detected!, Breed is  Bullmastiff
```



```
Dog detected!, Breed is  Doberman pinscher
```


```
Dog detected!, Breed is  Doberman pinscher
```