dog_app

March 18, 2021

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

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

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

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

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

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

Step 0: Import Datasets

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

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

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

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

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

```
In [3]: 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

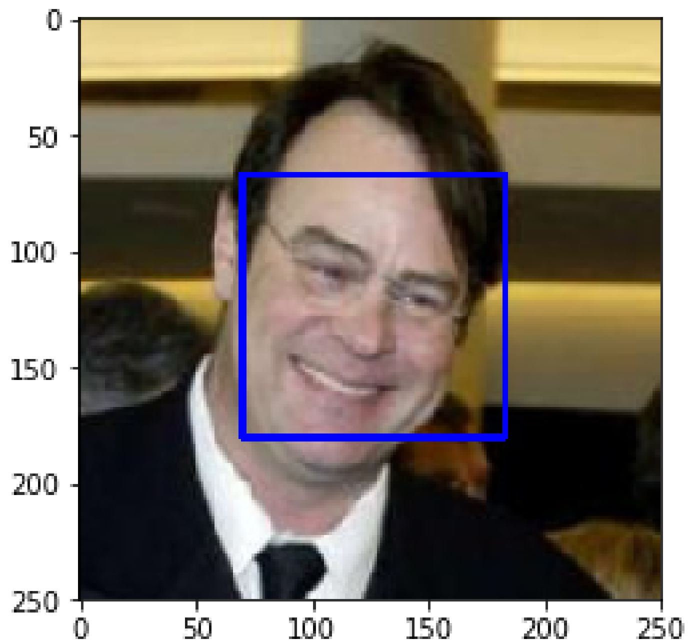
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [4]: # returns "True" if face is detected in image stored at img_path
        def face_detector(img_path):
            img = cv2.imread(img_path)
            gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
            faces = face_cascade.detectMultiScale(gray)
            return len(faces) > 0
```

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

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

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

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

```
In [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.
        has_human = 0
        for image_file in human_files_short:
            if face_detector(image_file): has_human += 1

        has_dog = 0
        for image_file in dog_files_short:
            if face_detector(image_file): has_dog += 1

        print("cv2.CascadeClassifier found a human face in {} human_files.".format(has_human))
        print("cv2.CascadeClassifier found a human face in {} dog_files.".format(has_dog))
```

```
cv2.CascadeClassifier found a human face in 98 human_files.
cv2.CascadeClassifier found a human face in 17 dog_files.
```

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 [ ]: ### (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 [5]: import torch
        import torchvision.models as models

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

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

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:13<00:00, 40382471.31it/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.

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 [100]:
```
### returns "True" if a dog is detected in the image stored at img_path
import cv2
from PIL import Image
import torchvision.transforms as transforms

def dog_detector(img_path):
    ## TODO: Complete the function.
    my_image = Image.open(img_path).convert('RGB')
    image_transform = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485,0.456,0.406], std=[0.229,0.224,0.225])
    ])
    image_tensor = image_transform(my_image)
    batch_of_one = image_tensor.unsqueeze(0)
    with torch.no_grad():
        if torch.cuda.is_available():
            # GPU is 25 times faster.
            # Need to copy (batch_of_one) into the GPU.
            batch_of_one = batch_of_one.cuda()
            output = VGG16(batch_of_one)
            probabilities = torch.nn.functional.softmax(output[0], dim=0 )
            # Need to copy the result of argmax(probabilities) to the CPU.
            # If it isn't in the CPU I can't evaluate  (151 <= indx <= 268) below.
            indx = torch.argmax(probabilities).cpu().numpy()
        else:
            output = VGG16(batch_of_one)
            probabilities = torch.nn.functional.softmax(output[0], dim=0 )
            indx = torch.argmax(probabilities).numpy()
    return 151 <= indx <= 268
```

In [101]:
```
dog_detector('images/Curly-coated_retriever_03896.jpg')
```

Out[101]: True

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

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

- What percentage of the images in `dog_files_short` have a detected dog?
Answer:

```
In [11]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.
         dogs_detected_in_human_files = 0
         for image_file in human_files_short:
             if dog_detector(image_file): dogs_detected_in_human_files += 1

         dogs_detected_in_dog_files = 0
         for image_file in dog_files_short:
             if dog_detector(image_file): dogs_detected_in_dog_files += 1

         print("Found a dog in {} human_files.".format(dogs_detected_in_human_files))
         print("Found a dog in {} dog_files.".format(dogs_detected_in_dog_files))
```

```
Found a dog in 0 human_files.
Found a dog in 100 dog_files.
```

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 [ ]: ### (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 [6]:
```
import os
import torch
from torchvision import datasets
import torchvision.transforms as transforms

train_transforms = transforms.Compose([
    transforms.RandomRotation(30),
    transforms.RandomResizedCrop(224),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize([0.485,0.456,0.406],[0.229,0.224,0.225])
])

# test_transforms used for validation and testing
test_transforms = transforms.Compose([
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize([0.485,0.456,0.406],[0.229,0.224,0.225])
])
```

In [7]:
```
### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes
import numpy as np
```

```
from torch.utils.data.sampler import SubsetRandomSampler

train_data = datasets.ImageFolder("/data/dog_images/train",transform=train_transforms)
train_indices = list(range(len(train_data)))
np.random.shuffle(train_indices)
train_sampler = SubsetRandomSampler(train_indices)
train_loader = torch.utils.data.DataLoader(train_data, batch_size=20,sampler=train_sampl

valid_data = datasets.ImageFolder("/data/dog_images/valid",transform=test_transforms)
valid_indices = list(range(len(valid_data)))
np.random.shuffle(valid_indices)
valid_sampler = SubsetRandomSampler(valid_indices)
valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=20,sampler=valid_sampl

test_data = datasets.ImageFolder("/data/dog_images/test",transform=test_transforms)
test_indices = list(range(len(test_data)))
np.random.shuffle(test_indices)
test_sampler = SubsetRandomSampler(test_indices)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=20,sampler=test_sampler,

loaders_scratch = {'train': train_loader,'valid': valid_loader,'test': test_loader}
```

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

Answer: The images are cropped to 224x224 pixels because they are much higher resolution that the images in MNIST and these images are similar to those in ImageNet which are 224x224. Random Crop, Rotation and HorizonzontalFlip is applied to the training data to add variety.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [5]: #from torchvision import models
        #VGG16 = models.vgg16(pretrained=True)
        #print(VGG16)
```

```
In [11]: 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(3,30,3,padding=1)# input dim = 224x224x3
                 self.conv2 = nn.Conv2d(30,64,3,padding=1)# input dim = 112x112x3
```

```
        self.conv3 = nn.Conv2d(64,64,3,padding=1)# input dim = 56x56x3

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

        self.fc1 = nn.Linear(64*28*28,500)
        self.fc2 = nn.Linear(500,500)
        self.fc3 = nn.Linear(500,133)
        self.drop = nn.Dropout(0.2)

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

        x = x.view(-1, 64*28*28) # Flatten image input
        x = self.drop(x)
        x = F.relu(self.fc1(x))
        x = self.drop(x)
        x = F.relu(self.fc2(x))
        x = self.drop(x)
        x = self.fc3(x)
        return x

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

# instantiate the CNN
model_scratch = Net()

# move tensors to GPU if CUDA is available
# check if CUDA is available
use_cuda = torch.cuda.is_available()

if use_cuda:
    model_scratch.cuda()
```

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

Answer: I choose an architecture similar to VGG16, but smaller. I an assuming networks like VGG16 are created by large comapines with deep pockets. Udacity is probably not making available over $40,000,000 worth of GPUs for me to use, and I have limited GPU hours available. Hence, I assumed my network must be smaller than VGG16.

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 [12]: 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)
```

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 [13]: from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True

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

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

                 ###################
                 # train the model #
                 ###################
                 model.train()
                 for batch_idx, (data, target) in enumerate(loaders['train']):
                     # move to GPU
                     if use_cuda:
                         data, target = data.cuda(), target.cuda()
                     ## find the loss and update the model parameters accordingly
                     ## record the average training loss, using something like
                     ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_lo
                     optimizer.zero_grad()
                     output = model(data)
                     loss = criterion(output, target)
                     loss.backward()
                     optimizer.step()
                     train_loss += loss.item()*data.size(0)
                 ######################
```

```
            # 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 += loss.item()*data.size(0)
            # print training/validation statistics
            train_loss = train_loss/len(train_loader.sampler)
            valid_loss = valid_loss/len(valid_loader.sampler)
            print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
                epoch, train_loss, valid_loss))

            ## TODO: save the model if validation loss has decreased
            if valid_loss <= valid_loss_min:
                torch.save(model.state_dict(), 'model_scratch.pt')
                valid_loss_min = valid_loss
        # return trained model
        return model


    # train the model
    model_scratch = train(14, 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: 4.885322        Validation Loss: 4.871196
Epoch: 2        Training Loss: 4.862533        Validation Loss: 4.821221
Epoch: 3        Training Loss: 4.810674        Validation Loss: 4.747142
Epoch: 4        Training Loss: 4.762031        Validation Loss: 4.696060
Epoch: 5        Training Loss: 4.736365        Validation Loss: 4.683454
Epoch: 6        Training Loss: 4.706437        Validation Loss: 4.638847
Epoch: 7        Training Loss: 4.667200        Validation Loss: 4.537657
Epoch: 8        Training Loss: 4.594837        Validation Loss: 4.475869
Epoch: 9        Training Loss: 4.559815        Validation Loss: 4.469384
Epoch: 10        Training Loss: 4.524506        Validation Loss: 4.441437
Epoch: 11        Training Loss: 4.500592        Validation Loss: 4.406379
Epoch: 12        Training Loss: 4.476878        Validation Loss: 4.396184
Epoch: 13        Training Loss: 4.441495        Validation Loss: 4.347263
Epoch: 14        Training Loss: 4.419624        Validation Loss: 4.310967
```

Accuracy is only 4%, so I train some more.

```
In [15]: model_scratch = train(14, 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: 4.384865         Validation Loss: 4.289352
Epoch: 2         Training Loss: 4.377223         Validation Loss: 4.378320
Epoch: 3         Training Loss: 4.337682         Validation Loss: 4.294084
Epoch: 4         Training Loss: 4.324796         Validation Loss: 4.245477
Epoch: 5         Training Loss: 4.296654         Validation Loss: 4.260554
Epoch: 6         Training Loss: 4.291731         Validation Loss: 4.182183
Epoch: 7         Training Loss: 4.268699         Validation Loss: 4.203884
Epoch: 8         Training Loss: 4.244170         Validation Loss: 4.175646
Epoch: 9         Training Loss: 4.208871         Validation Loss: 4.149079
Epoch: 10         Training Loss: 4.187955         Validation Loss: 4.214932
Epoch: 11         Training Loss: 4.172393         Validation Loss: 4.125490
Epoch: 12         Training Loss: 4.153517         Validation Loss: 4.114551
Epoch: 13         Training Loss: 4.132414         Validation Loss: 4.117334
Epoch: 14         Training Loss: 4.092488         Validation Loss: 4.089554
```

Accuracy is 8%, so I train some more but with lower lr (learning rate).

```
In [17]: optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=0.007)
         model_scratch = train(8, 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: 4.076001         Validation Loss: 4.047868
Epoch: 2         Training Loss: 4.031253         Validation Loss: 4.035524
Epoch: 3         Training Loss: 4.015488         Validation Loss: 4.044472
Epoch: 4         Training Loss: 3.994652         Validation Loss: 3.985336
Epoch: 5         Training Loss: 3.973393         Validation Loss: 3.967837
Epoch: 6         Training Loss: 3.958292         Validation Loss: 3.973751
Epoch: 7         Training Loss: 3.924204         Validation Loss: 3.971107
Epoch: 8         Training Loss: 3.896593         Validation Loss: 3.975693
```

Accuracy is 9% so I train some more.

```
In [19]: model_scratch = train(4, 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: 3.931576        Validation Loss: 4.097849
Epoch: 2         Training Loss: 3.928273        Validation Loss: 3.927834
Epoch: 3         Training Loss: 3.923602        Validation Loss: 3.953747
Epoch: 4         Training Loss: 3.894203        Validation Loss: 3.933117
```

Accuracy is still 9% so I train some more.

```
In [21]: model_scratch = train(3, 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: 3.914709        Validation Loss: 3.941952
Epoch: 2         Training Loss: 3.884913        Validation Loss: 3.965918
Epoch: 3         Training Loss: 3.867754        Validation Loss: 3.915075
```

Now accuracy is 8% which is worse even though both training loss and validation loss decreased! That might be due to randomly picking a batch with too many of outliers. I continue with more training.

```
In [23]: model_scratch = train(3, 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: 3.845110        Validation Loss: 3.906489
Epoch: 2         Training Loss: 3.824629        Validation Loss: 3.906453
Epoch: 3         Training Loss: 3.838432        Validation Loss: 3.972973
```

Accuracy is 83/836 which is not close enough, so I train some more.

```
In [25]: model_scratch = train(3, 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: 3.819453        Validation Loss: 3.914817
Epoch: 2         Training Loss: 3.802491        Validation Loss: 4.076510
Epoch: 3         Training Loss: 3.798865        Validation Loss: 3.884808
```

I finally have 10% accuracy.

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 [26]: model_scratch.load_state_dict(torch.load('model_scratch.pt'))

         def test(loaders, model, criterion, use_cuda):

             # monitor test loss and accuracy
             test_loss = 0.0
             correct = 0.0
             total = 0.0

             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: 3.927720


Test Accuracy: 10% (87/836)
```

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 [30]:
```
import os
import torch
import torchvision.transforms as transforms

train_transforms = transforms.Compose([
    transforms.RandomRotation(30),
    transforms.RandomResizedCrop(224),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize([0.485,0.456,0.406],[0.229,0.224,0.225])
])

# test_transforms used for validation and testing
test_transforms = transforms.Compose([
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize([0.485,0.456,0.406],[0.229,0.224,0.225])
])
```

In [31]:
```
## TODO: Specify data loaders
### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes
import numpy as np
from torchvision import datasets
from torch.utils.data.sampler import SubsetRandomSampler

train_data = datasets.ImageFolder("/data/dog_images/train",transform=train_transforms)
train_indices = list(range(len(train_data)))
np.random.shuffle(train_indices)
train_sampler = SubsetRandomSampler(train_indices)
train_loader = torch.utils.data.DataLoader(train_data, batch_size=20,sampler=train_samp

valid_data = datasets.ImageFolder("/data/dog_images/valid",transform=test_transforms)
valid_indices = list(range(len(valid_data)))
np.random.shuffle(valid_indices)
valid_sampler = SubsetRandomSampler(valid_indices)
valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=20,sampler=valid_samp

test_data = datasets.ImageFolder("/data/dog_images/test",transform=test_transforms)
test_indices = list(range(len(test_data)))
np.random.shuffle(test_indices)
```

```
test_sampler = SubsetRandomSampler(test_indices)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=20,sampler=test_sampler

loaders_transfer = {'train': train_loader,'valid': valid_loader,'test': test_loader}
```

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`.

Instead of transfer learning from VGG16, I wanted to see if I could use my skills with a more sophisticated CNN. Besides that the more sophisticated CNN might give better results than if I used VGG16.

```
In [1]: from torchvision import models
        model_transfer = models.densenet121(pretrained=True)
        print(model_transfer)
```

```
/opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-py3.6.egg/torchvision/models/densenet.p
Downloading: "https://download.pytorch.org/models/densenet121-a639ec97.pth" to /root/.torch/mode
100%|| 32342954/32342954 [00:00<00:00, 84289648.25it/s]


DenseNet(
  (features): Sequential(
    (conv0): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
    (norm0): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu0): ReLU(inplace)
    (pool0): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
    (denseblock1): _DenseBlock(
      (denselayer1): _DenseLayer(
        (norm1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu1): ReLU(inplace)
        (conv1): Conv2d(64, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        (relu2): ReLU(inplace)
        (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      )
      (denselayer2): _DenseLayer(
        (norm1): BatchNorm2d(96, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu1): ReLU(inplace)
        (conv1): Conv2d(96, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        (relu2): ReLU(inplace)
        (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      )
      (denselayer3): _DenseLayer(
        (norm1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        (relu1): ReLU(inplace)
        (conv1): Conv2d(128, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

```
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer4): _DenseLayer(
      (norm1): BatchNorm2d(160, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(160, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer5): _DenseLayer(
      (norm1): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(192, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer6): _DenseLayer(
      (norm1): BatchNorm2d(224, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(224, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
  )
  (transition1): _Transition(
    (norm): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
    (conv): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (pool): AvgPool2d(kernel_size=2, stride=2, padding=0)
  )
  (denseblock2): _DenseBlock(
    (denselayer1): _DenseLayer(
      (norm1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(128, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer2): _DenseLayer(
      (norm1): BatchNorm2d(160, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(160, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

```
    (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
    (relu2): ReLU(inplace)
    (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  )
  (denselayer3): _DenseLayer(
    (norm1): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
    (relu1): ReLU(inplace)
    (conv1): Conv2d(192, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
    (relu2): ReLU(inplace)
    (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  )
  (denselayer4): _DenseLayer(
    (norm1): BatchNorm2d(224, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
    (relu1): ReLU(inplace)
    (conv1): Conv2d(224, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
    (relu2): ReLU(inplace)
    (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  )
  (denselayer5): _DenseLayer(
    (norm1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
    (relu1): ReLU(inplace)
    (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
    (relu2): ReLU(inplace)
    (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  )
  (denselayer6): _DenseLayer(
    (norm1): BatchNorm2d(288, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
    (relu1): ReLU(inplace)
    (conv1): Conv2d(288, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
    (relu2): ReLU(inplace)
    (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  )
  (denselayer7): _DenseLayer(
    (norm1): BatchNorm2d(320, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
    (relu1): ReLU(inplace)
    (conv1): Conv2d(320, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
    (relu2): ReLU(inplace)
    (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  )
  (denselayer8): _DenseLayer(
    (norm1): BatchNorm2d(352, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
    (relu1): ReLU(inplace)
    (conv1): Conv2d(352, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

```
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer9): _DenseLayer(
      (norm1): BatchNorm2d(384, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(384, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer10): _DenseLayer(
      (norm1): BatchNorm2d(416, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(416, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer11): _DenseLayer(
      (norm1): BatchNorm2d(448, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(448, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer12): _DenseLayer(
      (norm1): BatchNorm2d(480, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(480, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
  )
  (transition2): _Transition(
    (norm): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
    (conv): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (pool): AvgPool2d(kernel_size=2, stride=2, padding=0)
  )
  (denseblock3): _DenseBlock(
    (denselayer1): _DenseLayer(
      (norm1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

```
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer2): _DenseLayer(
      (norm1): BatchNorm2d(288, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(288, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer3): _DenseLayer(
      (norm1): BatchNorm2d(320, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(320, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer4): _DenseLayer(
      (norm1): BatchNorm2d(352, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(352, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer5): _DenseLayer(
      (norm1): BatchNorm2d(384, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(384, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer6): _DenseLayer(
      (norm1): BatchNorm2d(416, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(416, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer7): _DenseLayer(
      (norm1): BatchNorm2d(448, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(448, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

```
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer8): _DenseLayer(
      (norm1): BatchNorm2d(480, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(480, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer9): _DenseLayer(
      (norm1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer10): _DenseLayer(
      (norm1): BatchNorm2d(544, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(544, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer11): _DenseLayer(
      (norm1): BatchNorm2d(576, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(576, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer12): _DenseLayer(
      (norm1): BatchNorm2d(608, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(608, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer13): _DenseLayer(
      (norm1): BatchNorm2d(640, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(640, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

```
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer14): _DenseLayer(
      (norm1): BatchNorm2d(672, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(672, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer15): _DenseLayer(
      (norm1): BatchNorm2d(704, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(704, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer16): _DenseLayer(
      (norm1): BatchNorm2d(736, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(736, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer17): _DenseLayer(
      (norm1): BatchNorm2d(768, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(768, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer18): _DenseLayer(
      (norm1): BatchNorm2d(800, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(800, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer19): _DenseLayer(
      (norm1): BatchNorm2d(832, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(832, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

```
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer20): _DenseLayer(
      (norm1): BatchNorm2d(864, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(864, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer21): _DenseLayer(
      (norm1): BatchNorm2d(896, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(896, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer22): _DenseLayer(
      (norm1): BatchNorm2d(928, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(928, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer23): _DenseLayer(
      (norm1): BatchNorm2d(960, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(960, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer24): _DenseLayer(
      (norm1): BatchNorm2d(992, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(992, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
  )
  (transition3): _Transition(
    (norm): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
```

```
    (conv): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (pool): AvgPool2d(kernel_size=2, stride=2, padding=0)
  )
  (denseblock4): _DenseBlock(
    (denselayer1): _DenseLayer(
      (norm1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer2): _DenseLayer(
      (norm1): BatchNorm2d(544, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(544, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer3): _DenseLayer(
      (norm1): BatchNorm2d(576, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(576, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer4): _DenseLayer(
      (norm1): BatchNorm2d(608, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(608, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer5): _DenseLayer(
      (norm1): BatchNorm2d(640, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(640, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer6): _DenseLayer(
      (norm1): BatchNorm2d(672, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(672, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

```
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer7): _DenseLayer(
      (norm1): BatchNorm2d(704, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(704, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer8): _DenseLayer(
      (norm1): BatchNorm2d(736, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(736, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer9): _DenseLayer(
      (norm1): BatchNorm2d(768, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(768, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer10): _DenseLayer(
      (norm1): BatchNorm2d(800, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(800, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer11): _DenseLayer(
      (norm1): BatchNorm2d(832, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(832, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu2): ReLU(inplace)
      (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    )
    (denselayer12): _DenseLayer(
      (norm1): BatchNorm2d(864, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
      (relu1): ReLU(inplace)
      (conv1): Conv2d(864, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

```
        (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        (relu2): ReLU(inplace)
        (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      )
      (denselayer13): _DenseLayer(
        (norm1): BatchNorm2d(896, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        (relu1): ReLU(inplace)
        (conv1): Conv2d(896, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        (relu2): ReLU(inplace)
        (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      )
      (denselayer14): _DenseLayer(
        (norm1): BatchNorm2d(928, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        (relu1): ReLU(inplace)
        (conv1): Conv2d(928, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        (relu2): ReLU(inplace)
        (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      )
      (denselayer15): _DenseLayer(
        (norm1): BatchNorm2d(960, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        (relu1): ReLU(inplace)
        (conv1): Conv2d(960, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        (relu2): ReLU(inplace)
        (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      )
      (denselayer16): _DenseLayer(
        (norm1): BatchNorm2d(992, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        (relu1): ReLU(inplace)
        (conv1): Conv2d(992, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
        (relu2): ReLU(inplace)
        (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      )
    )
    (norm5): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (classifier): Linear(in_features=1024, out_features=1000, bias=True)
)
```

The result of print( ) above shows that the last layer of densenet121 is model_transfer.classifier. This in confirmed with the next line.

```
In [2]: model_transfer.classifier
```

```
Out[2]: Linear(in_features=1024, out_features=1000, bias=True)
```

```
In [33]: # Freeze parameters of the denesenet121 model
         for param in model_transfer.features.parameters():
             param
         # Replace the last_layer with my own linear layer that will classify 133 dog_breeds.
         import torch.nn as nn
         model_transfer.classifier=nn.Linear(1024,133)
         use_cuda = torch.cuda.is_available()
         if use_cuda:
             model_transfer = model_transfer.cuda()
         use_cuda
```

```
Out[33]: False
```

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: We have a relativly small set of images that are similar to images in ImageNet. In that case we should only replace the last layer of an CNN that classifies ImagNet images. The last layer should be replaced with a linear layer having number_of_outputs = number_of_breeds=133.

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 [4]: import torch.optim as optim
        import torch.nn as nn
        criterion_transfer = nn.CrossEntropyLoss()
        optimizer_transfer = optim.SGD(model_transfer.parameters(), lr=0.01)
```

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 [10]: # train the model
         from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True

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

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

                 ###################
```

```
        # train the model #
        ###################
        model.train()
        for batch_idx, (data, target) in enumerate(loaders['train']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## find the loss and update the model parameters accordingly
            ## record the average training loss, using something like
            ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_lo
            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()
            train_loss += loss.item()*data.size(0)
        ######################
        # 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 += loss.item()*data.size(0)
        # print training/validation statistics
        train_loss = train_loss/len(train_loader.sampler)
        valid_loss = valid_loss/len(valid_loader.sampler)
        print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
            epoch, train_loss, valid_loss))

        ## TODO: save the model if validation loss has decreased
        if valid_loss <= valid_loss_min:
            torch.save(model.state_dict(), 'model_transfer.pt')
            valid_loss_min = valid_loss
    # return trained model
    return model


# train the model
model_transfer = train(4, loaders_transfer, model_transfer, optimizer_transfer,
                      criterion_transfer, use_cuda, 'model_transfer.pt')

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

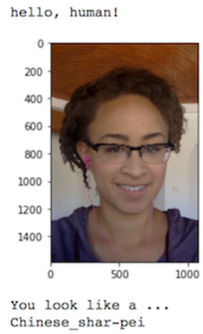
```
Epoch: 1        Training Loss: 3.614847        Validation Loss: 2.082807
Epoch: 2        Training Loss: 1.943640        Validation Loss: 1.319558
Epoch: 3        Training Loss: 1.438528        Validation Loss: 1.093860
Epoch: 4        Training Loss: 1.211910        Validation Loss: 0.951406
```

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 [12]: 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_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.979303


Test Accuracy: 73% (617/836)
```



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.

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 [10]: from glob import glob
         import numpy as np

         human_files = np.array(glob("/data/lfw/*/*"))
         dog_files = np.array(glob("/data/dog_images/*/*/*"))

         import torch
         import torch.nn as nn
         from torchvision import models

         model_transfer = models.densenet121(pretrained=True)
         model_transfer.eval()
         model_transfer.classifier=nn.Linear(1024,133)
         #model_transfer.load_state_dict(torch.load('model_transfer.pt', map_location=device))
```

```
/opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-py3.6.egg/torchvision/models/densenet.p
```

```
In [11]: if torch.cuda.is_available():
             model_transfer = model_transfer.cuda()
             model_transfer.load_state_dict(torch.load('model_transfer.pt'))
         else:
             model_transfer.load_state_dict(torch.load('model_transfer.pt', map_location='cpu'))
```

```
In [12]: import cv2

         def has_human(file_path):
             face_cascade = cv2.CascadeClassifier('haarcascades/haarcascade_frontalface_alt.xml'
             image = cv2.imread(file_path)
             gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
             faces = face_cascade.detectMultiScale(gray)
             return  0<len(faces)
```

```
In [13]: # VGG16 is used in  has_dog(image,transform)
         VGG16 = models.vgg16(pretrained=True)
         # move model to GPU if CUDA is available
         if torch.cuda.is_available():
             VGG16 = VGG16.cuda()

         def has_dog(transformed_image):
             with torch.no_grad():
                 if torch.cuda.is_available():
                     # GPU is 25 times faster, so copy (transformed_image) into the GPU.
                     transformed_image = transformed_image.cuda
                     output = VGG16(transformed_image)
                     probabilities = torch.nn.functional.softmax(output[0], dim=0 )
                     # Need to copy the result of argmax(probabilities) to the CPU.
                     # If it isn't in the CPU I can't evaluate  (151 <= indx <= 268) below.
                     indx = torch.argmax(probabilities).cpu().numpy()
                 else:
                     output = VGG16(transformed_image)
                     probabilities = torch.nn.functional.softmax(output[0], dim=0 )
                     indx = torch.argmax(probabilities).numpy()
             return 151 <= indx <= 268
```

```
In [14]: import torchvision.transforms as transforms
         from torchvision import datasets
         import torch.nn.functional as F

         train_data = datasets.ImageFolder("/data/dog_images/train")
         train_loader = torch.utils.data.DataLoader(train_data)
         class_names = [item[4:].replace("_", " ") for item in train_loader.dataset.classes]

         def predicted_dog_breed(transformed_image):
             if torch.cuda.is_available():
                 transformed_image = transformed_image.cuda()
```

```
        raw_prediction = model_transfer(transformed_image)
        softmax_prediction = F.softmax(raw_prediction, dim=1)
        class_score, class_index = torch.max(softmax_prediction, 1)
        #  probability  = class_score.item()
        return class_names[class_index.item()]
```

```
In [15]: from PIL import Image
         import torchvision.transforms as transforms
         import matplotlib.pyplot as plt
         %matplotlib inline

         def run_app(file_path):
             ## handle cases for a human face, dog, and neither
             humanQ = has_human(file_path)
             transform = transforms.Compose([
                 transforms.Resize(224),
                 transforms.CenterCrop(224),
                 transforms.ToTensor(),
                 transforms.Normalize(mean=[0.485,0.456,0.406], std=[0.229,0.224,0.225])
             ])
             image = Image.open(file_path).convert('RGB')
             # Use unsqueeze(0) to make it into a batch of one image.
             transformed_image = transform(image).unsqueeze(0)
             dogQ = has_dog(transformed_image)
             if (not humanQ) and (not dogQ):
                 image_title = "Error: The picture has neither a dog or a person."
             else:
                 breed = predicted_dog_breed(transformed_image)
                 if dogQ:
                     if humanQ:
                         image_title = "This picture has a person and a dog and one of them look
                     else:
                         image_title = "This dog is a " + breed
                 else:
                     image_title = "This person looks like a " + breed
             # Formatting the plot is based on
             #
             # https://stackoverflow.com/questions/47718341/matplotlib-imshow-issues-title-on-to
             #
             # https://discourse.matplotlib.org/t/removing-ticks-and-frame-imshow/16283/4
             #
             fig, ax = plt.subplots()
             im = ax.imshow(cv2.cvtColor(cv2.imread(file_path), cv2.COLOR_BGR2RGB), cmap=plt.cm.
             ax.set_title(image_title)
             plt.axis('off')
             return plt.show()
```

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.

```
In [ ]: ## 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.
```

I went above and beyond and correctly delt with an image that has a person and a dog. I am not sure if my algorithm classified all dogs correctly. Any mistakes are for breeds very similar to the correct breed. The results are outstanding!

```
In [19]: run_app(dog_files[27])
```

This picture has a person and a dog and one of them looks like a Bullmastiff

My algorithm can tell that an image of a door or a donkey is neither a dog or a person. However, it mistakes a picture of a coyote for a person! so the Haar feature-based cascade classifier has too many false positives when used to detecct people in an image.

```
In [20]: run_app('images/door.jpg')
         run_app('images/Donkey.jpg')
         run_app('images/Coyote.jpg')
```

Error: The picture has neither a dog or a person.

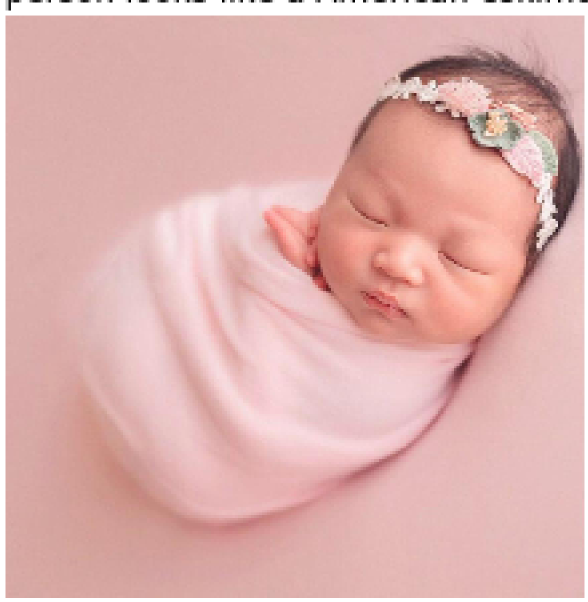
Error: The picture has neither a dog or a person.

This person looks like a Norwegian elkhound

Here only one of two baby pictures is classified as a person. This shows a weakness of using the Haar feature-based cascade classifier for identifiying pictures of people.

```
In [23]: run_app('images/Baby1.jpg')
         run_app('images/Baby2.jpg')
```

This person looks like a American eskimo dog



The images below from Udacity provided are correctly classified as people.

```
In [21]: for file in human_files[:3]:
             run_app(file)
```

This person looks like a Basenji

This person looks like a Basenji

If any dogs below are misclassified, it is for a very similar breed.

```
In [11]: for file in dog_files[:3]:
             run_app(file)
         run_app('images/Brittany_02625.jpg')
         run_app('images/Brittany_1.jpg')
         run_app('images/Brittany_2.jpg')
         run_app('images/American_water_spaniel_00648.jpg')
         run_app('images/America_water_spaniel_1.jpg')
         run_app('images/America_water_spaniel_2.jpg')
         run_app('images/Labrador_retriever_06449.jpg')
         run_app('images/Labrador_retriever_06455.jpg')
         run_app('images/Labrador_retriever_06457.jpg')
         run_app('images/Welsh_springer_spaniel_08203.jpg')
```

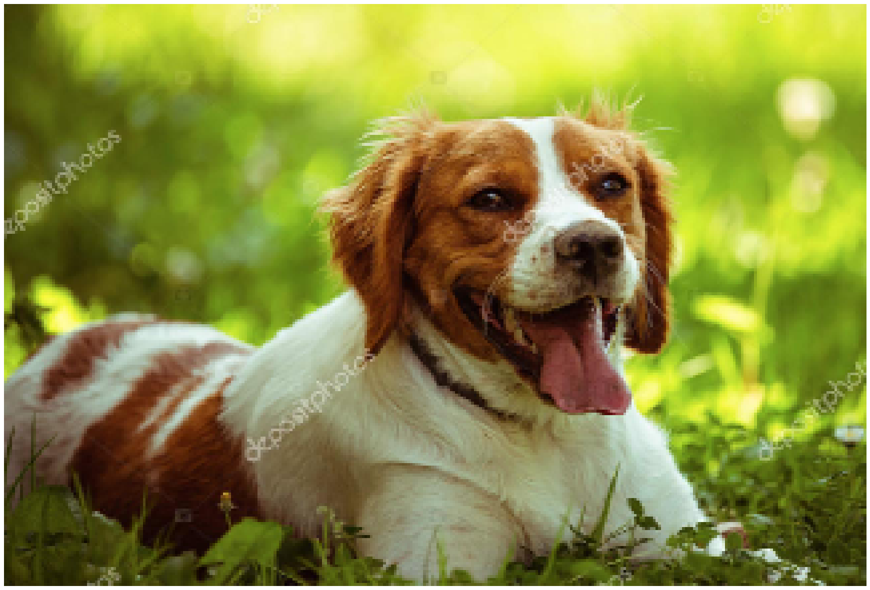
This dog is a Bullmastiff

This dog is a Bullmastiff

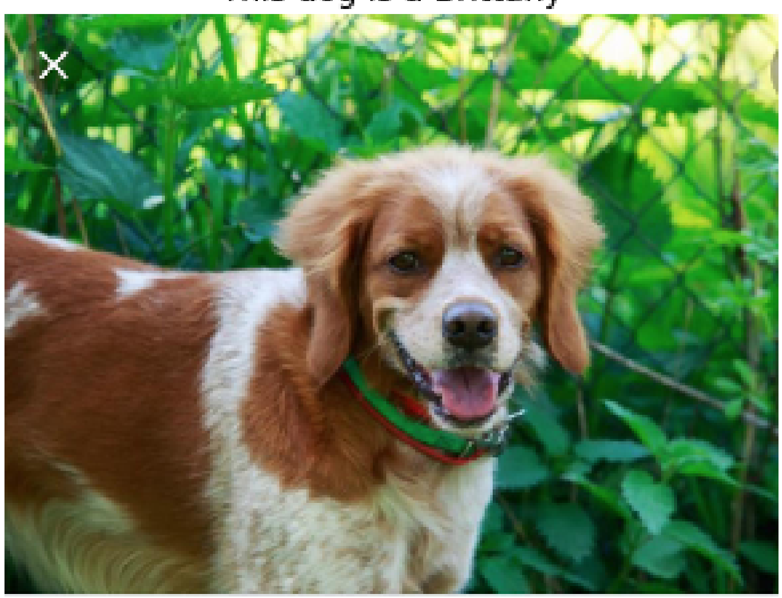
This dog is a Chinese shar-pei

This dog is a Brittany

This dog is a Brittany



This dog is a Brittany

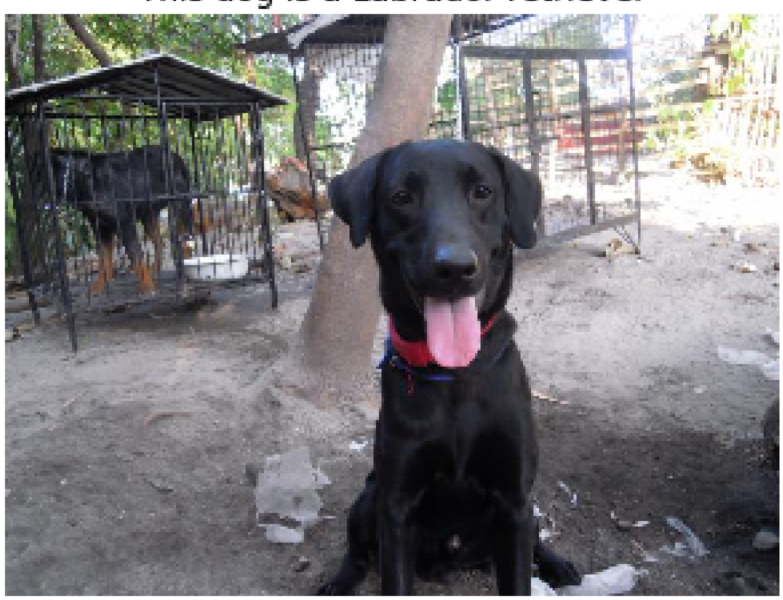


This dog is a American water spaniel

This picture has a person and a dog and one of them looks like a American water spaniel

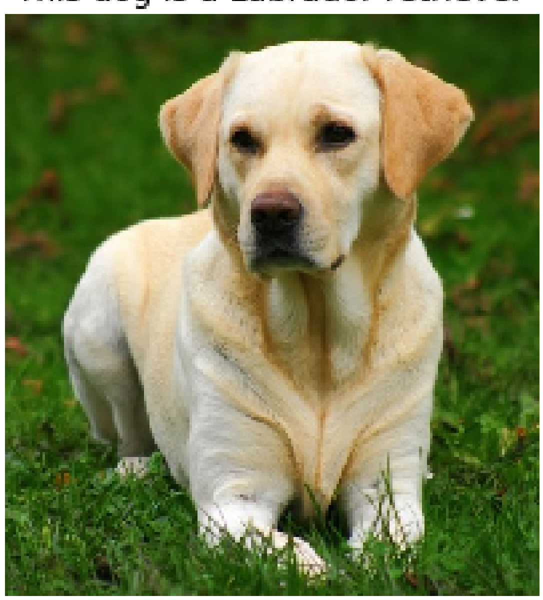
This dog is a Boykin spaniel

This dog is a Labrador retriever



This dog is a Chesapeake bay retriever

This dog is a Labrador retriever



This dog is a Welsh springer spaniel

Improvements could include the following:

1) Ensure the algorithm can to count the number of dogs and the number of people in an image.

2) Allow the user to ensure the people and dogs in an image are ignored if the confidence in the classification is not above a specified level.

3) Estimate the age, gender, and race of people in an image.

4) Be able to recognize a person in an image even when thier face is not visible.

5) Be able to recognize people wearing sunglasses and/or a cloth mask and/or a hat.

6) Be able to tell the difference between a dog, a Wolf and a Coyote.

7) Ensure the algorithm is robust at recognizing images of babies, very old people, and people with very dark skin.

8) Ensure the algorithm works well with images that are slightly out of focus, over exposed, or under exposed.