dog_app

January 30, 2019

0.0.1 *ATTENTION FOR A REVIEWER: Some cells are executed not consequently because I had problems with CUDA memory. Not enough CUDA memory to run the project on my server with GTX1050 and only 2Gb of RAM. I sincerely appologize for that.*

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

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

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

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

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

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

Step 0: Import Datasets

Make sure that you've downloaded the required human and dog datasets: * Download the dog dataset. Unzip the folder and place it in this project's home directory, at the location `/dogImages`.

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

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

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

In [1]:
```
import numpy as np
from glob import glob

# load filenames for human and dog images
human_files = np.array(glob("lfw/*/*"))
dog_files = np.array(glob("dogImages/*/*/*"))

# print number of images in each dataset
print('There are %d total human images.' % len(human_files))
print('There are %d total dog images.' % len(dog_files))
```

```
There are 13233 total human images.
There are 8351 total dog images.
```

Step 1: Detect Humans

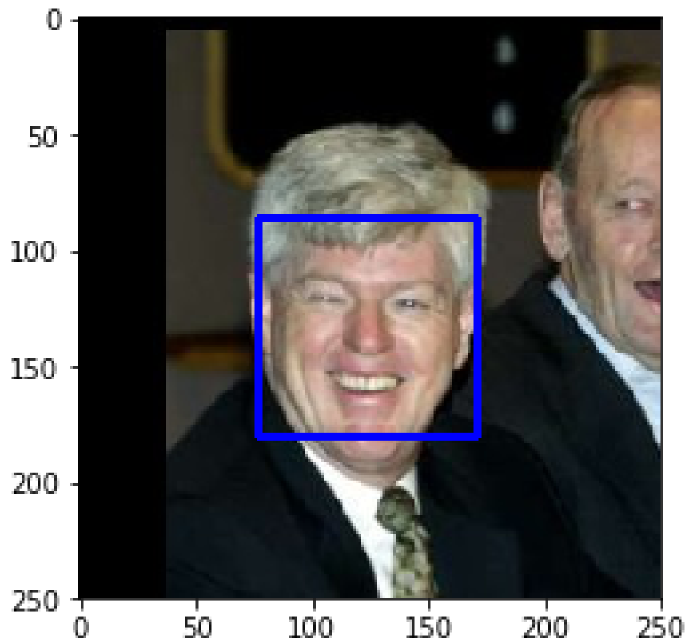
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

OpenCV provides many pre-trained face detectors, stored as XML files on github. We have downloaded one of these detectors and stored it in the `haarcascades` directory. In the next code cell, we demonstrate how to use this detector to find human faces in a sample image.

In [2]:
```
import cv2
import matplotlib.pyplot as plt
%matplotlib inline

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

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

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

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

# get bounding box for each detected face
for (x,y,w,h) in faces:
```

```
        # add bounding box to color image
        cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)

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

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

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



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

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

1.1.1 Write a Human Face Detector

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

```
In [3]: # returns "True" if face is detected in image stored at img_path
        def face_detector(img_path):
            img = cv2.imread(img_path)
            gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
            faces = face_cascade.detectMultiScale(gray)
            return len(faces) > 0
```

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

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

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

Answer: (You can print out your results and/or write your percentages in this cell)
I printed results. Please, see cell below.

```
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.
        print('What percentage of the first 100 images in human_files have a detected human fa
              '\n{:.0%}'.format(sum(map(face_detector, human_files_short)) / 100))
        print('What percentage of the first 100 images in dog_files have a detected human face
              '\n{:.0%}'.format(sum(map(face_detector, dog_files_short)) / 100))
```

```
What percentage of the first 100 images in human_files have a detected human face?
99%
What percentage of the first 100 images in dog_files have a detected human face?
18%
```

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()
```

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

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

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

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

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

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

            Args:
                img_path: path to an image

            Returns:
```

```
        Index corresponding to VGG-16 model's prediction
    '''

    ## TODO: Complete the function.
    ## Load and pre-process an image from the given img_path
    ## Return the *index* of the predicted class for that image
    pil_image = Image.open(img_path)
    transformed_image = transforms.Resize(224)(pil_image)
    transformed_image = transforms.CenterCrop(224)(pil_image)
    transformed_image = transforms.ToTensor()(transformed_image)
    transformed_image = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                             std=[0.229, 0.224, 0.225])(transformed_ima
    transformed_image = transformed_image.cuda() if torch.cuda.is_available() else tran
    output_index = int((VGG16(transformed_image.unsqueeze(0))).argmax())
    return output_index # predicted class index
```

1.1.5 (IMPLEMENTATION) Write a Dog Detector

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

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

```
In [8]: ### returns "True" if a dog is detected in the image stored at img_path
        def dog_detector(img_path):
            ## TODO: Complete the function.
            return True if VGG16_predict(img_path) in range(151, 269) else False # true/false
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

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

Answer: I printed results, please, see output of the cell below.

```
In [10]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.
         print('What percentage of the images in human_files_short have a detected dog?'
               '\n{:.0%}'.format(sum(map(dog_detector, human_files_short)) / 100))
         print('What percentage of the images in dog_files_short have a detected dog?'
               '\n{:.0%}'.format(sum(map(dog_detector, dog_files_short)) / 100))
```

```
What percentage of the images in human_files_short have a detected dog?
1%
What percentage of the images in dog_files_short have a detected dog?
```

96%

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 [11]: ### (Optional)
         ### TODO: Report the performance of another pre-trained network.
         ### Feel free to use as many code cells as needed.
         def INCEPTION101_predict_dog(img_path):
             INCEPTION101 = models.resnet101(pretrained=True).cuda() if torch.cuda.is_availabl
                                                          else models.resnet101(pretrained=Tru
             pil_image = Image.open(img_path)
             transformed_image = transforms.Resize(224)(pil_image)
             transformed_image = transforms.CenterCrop(224)(pil_image)
             transformed_image = transforms.ToTensor()(transformed_image)
             transformed_image = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                      std=[0.229, 0.224, 0.225])(transformed_i

             transformed_image = transformed_image.cuda() if torch.cuda.is_available() else tr
             output_index = int((VGG16(transformed_image.unsqueeze(0))).argmax())
             return True if output_index in range(151, 269) else False

         print('What percentage of the images in human_files_short have a detected dog?'
               '\n{:.0%}'.format(sum(map(INCEPTION101_predict_dog, human_files_short)) / 100))
         print('What percentage of the images in dog_files_short have a detected dog?'
               '\n{:.0%}'.format(sum(map(INCEPTION101_predict_dog, dog_files_short)) / 100))
```

```
What percentage of the images in human_files_short have a detected dog?
0%
What percentage of the images in dog_files_short have a detected dog?
98%
```

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 `dogImages/train`, `dogImages/valid`, and `dogImages/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 [1]:
```
import os
from torchvision import datasets

### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes
import numpy as np
import torch
from PIL import ImageFile
from torchvision import transforms

ImageFile.LOAD_TRUNCATED_IMAGES = True
use_cuda = torch.cuda.is_available()

image_transforms = {'train': transforms.Compose([transforms.RandomRotation(10),
                                     transforms.RandomHorizontalFlip(),
                                     transforms.RandomResizedCrop(224),
```

```
                                   transforms.Resize(224),
                                   transforms.CenterCrop(224),
                                   transforms.ToTensor(),
                                   transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                        std=[0.229, 0.224, 0.225])])
                  'valid': 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])])
                  'test': 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])])
                 }

loaders_scratch = {'train': torch.utils.data.DataLoader(datasets.ImageFolder('dogImages
                                                                  transform=
                                                        batch_size=25, shuffle=True),
                  'valid': torch.utils.data.DataLoader(datasets.ImageFolder('dogImages
                                                                  transform=
                                                        batch_size=25, shuffle=True),
                  'test': torch.utils.data.DataLoader(datasets.ImageFolder('dogImages/
                                                                 transform=
                                                        batch_size=25, shuffle=True)}
```

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:

How does your code resize the images (by cropping, stretching, etc)? What size did you pick for the input tensor, and why?

I used `transforms.Resize(224)` and `transforms.CenterCrop(224)` to extract valuable information from the photos. I pick size 224 by 224 for input tensor. Because this size of widely supported by many model in deep leanrning, especially ImageNet. *Did you decide to augment the dataset? If so, how (through translations, flips, rotations, etc)? If not, why not?*

Augmentation is good in production. Although I don't want to make process more complicated. I used simple horizonatal flip, random rotation and random resized crop. Because I had a look at all data: `train`, `valid` and `test` and decided that these transformations are the best considering photos in a dataset and its parameters.

ADDITION: I experimented with own class inherited from `torch.utils.data.Dataset` and it was fun although performance is not great. PyTorch solution of `ImageFolder` is far more effective. Although I learned how to create custom `Dataset` class which is very usefull to load sound or 3D models not supported by usual built-in loaders. Here is my try:

```
class ImageLoader(torch.utils.data.Dataset):
```

```
    def __init__(self, img_path):
        super(ImageLoader, self).__init__()
        self.img_path = list(glob(img_path + "/*/*"))
        self.cc = transforms.CenterCrop(250)
        self.random = transforms.RandomRotation(5)
        self.tt = transforms.ToTensor()

    def __getitem__(self, index):
        image_item = self.tt(self.random(self.cc(Image.open(self.img_path[index]))))
        dir_num, dog_type = self.img_path[index].split('/')[2].split('.')
        dir_num_minus_one = int(dir_num) - 1
        return image_item, dir_num_minus_one

    def __len__(self):
        return len(self.img_path)
```

1.1.8 (IMPLEMENTATION) Model Architecture

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

In [2]:
```
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.c1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3)
        self.c2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3)
        self.c3 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3)
        self.fc1 = nn.Linear(in_features=6272, out_features=512)
        self.fc2 = nn.Linear(in_features=512, out_features=256)
        self.fc3 = nn.Linear(in_features=256, out_features=412)
        self.fc4 = nn.Linear(in_features=412, out_features=133)

    def forward(self, x):
        ## Define forward behavior
        x = F.relu(F.max_pool2d(self.c1(x), 3))
        x = F.relu(F.max_pool2d(self.c2(x), 3))
        x = F.relu(F.max_pool2d(self.c3(x), 3))
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = 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()
```

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

Answer: First I tried model that I usually use for image classification problems from my experience:

```
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.c1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3)
        self.c2 = nn.Conv2d(in_channels=32, out_channels=32, kernel_size=5)
        self.c3 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=7)
        self.fc1 = nn.Linear(in_features=5184, out_features=128)
        self.fc2 = nn.Linear(in_features=128, out_features=64)
        self.fc3 = nn.Linear(in_features=64, out_features=256)
        self.fc4 = nn.Linear(in_features=256, out_features=133)

    def forward(self, x):
        ## Define forward behavior
        x = F.relu(F.max_pool2d(self.c1(x), 3))
        x = F.relu(F.max_pool2d(self.c2(x), 3))
        x = F.relu(F.max_pool2d(self.c3(x), 3, 2))
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        x = F.softmax(self.fc4(x))

        return x
```

Softmax is really slow and model converges very slow. 100 epochs is not enough as I experienced very slight loss decrease in time. I have chosen my model trying around 30 different architectures. The most important factor is a speed. So I cut my model and it started performing well for 100 epochs. I optimized it more. And now it's fine to have 50 epochs to reach accuracy of 10% on test dataset. Maybe even little more. Maximum accuracy that I've got arount 36% on test set. I recorded each my step, below are some of my attempts:

```
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.c1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3)
        self.c2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=5)
        self.fc1 = nn.Linear(in_features=12544, out_features=128)
        self.fc2 = nn.Linear(in_features=128, out_features=64)
        self.fc3 = nn.Linear(in_features=64, out_features=133)

    def forward(self, x):
        ## Define forward behavior
        x = F.relu(F.max_pool2d(self.c1(x), 3))
        x = F.relu(F.max_pool2d(self.c2(x), 5))
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.softmax(self.fc3(x), dim=1)

        return x
```

```
Epoch: 1    Training Loss: 4.890113     Validation Loss: 4.889642
Saving the model...
Epoch: 2    Training Loss: 4.886004     Validation Loss: 4.880586
Saving the model...
Epoch: 3    Training Loss: 4.874945     Validation Loss: 4.875909
Saving the model...
Epoch: 4    Training Loss: 4.869064     Validation Loss: 4.871696
Saving the model...
Epoch: 5    Training Loss: 4.865634     Validation Loss: 4.868212
Saving the model...
Epoch: 6    Training Loss: 4.863153     Validation Loss: 4.868111
Saving the model...
Epoch: 7    Training Loss: 4.859790     Validation Loss: 4.873127
Epoch: 8    Training Loss: 4.859132     Validation Loss: 4.867167
Saving the model...
Epoch: 9    Training Loss: 4.855651     Validation Loss: 4.872077
Epoch: 10   Training Loss: 4.854802     Validation Loss: 4.862651
Saving the model...
Epoch: 11   Training Loss: 4.852467     Validation Loss: 4.863158
Epoch: 12   Training Loss: 4.851509     Validation Loss: 4.864380
Epoch: 13   Training Loss: 4.848721     Validation Loss: 4.869511
Epoch: 14   Training Loss: 4.847045     Validation Loss: 4.854262
Saving the model...
Epoch: 15   Training Loss: 4.846539     Validation Loss: 4.869208
Epoch: 16   Training Loss: 4.843713     Validation Loss: 4.873142
Epoch: 17   Training Loss: 4.841424     Validation Loss: 4.865548
Epoch: 18   Training Loss: 4.840212     Validation Loss: 4.861228
```

```
Epoch: 19   Training Loss: 4.838126     Validation Loss: 4.870139
Epoch: 20   Training Loss: 4.837753     Validation Loss: 4.864596
Epoch: 21   Training Loss: 4.834942     Validation Loss: 4.857877
Epoch: 22   Training Loss: 4.833254     Validation Loss: 4.853379
Saving the model...
Epoch: 23   Training Loss: 4.835518     Validation Loss: 4.854793
Epoch: 24   Training Loss: 4.831944     Validation Loss: 4.852176
Saving the model...
Epoch: 25   Training Loss: 4.831647     Validation Loss: 4.870401
Epoch: 26   Training Loss: 4.829886     Validation Loss: 4.860188
Epoch: 27   Training Loss: 4.827880     Validation Loss: 4.860277
Epoch: 28   Training Loss: 4.827973     Validation Loss: 4.863347
Epoch: 29   Training Loss: 4.825450     Validation Loss: 4.865826
Epoch: 30   Training Loss: 4.825446     Validation Loss: 4.865802
Epoch: 31   Training Loss: 4.823805     Validation Loss: 4.854404
Epoch: 32   Training Loss: 4.823603     Validation Loss: 4.867640
Epoch: 33   Training Loss: 4.823084     Validation Loss: 4.860510
Epoch: 34   Training Loss: 4.821469     Validation Loss: 4.857656
Epoch: 35   Training Loss: 4.821595     Validation Loss: 4.859674
Epoch: 36   Training Loss: 4.820993     Validation Loss: 4.851964
Saving the model...
Epoch: 37   Training Loss: 4.820076     Validation Loss: 4.857115
Epoch: 38   Training Loss: 4.819050     Validation Loss: 4.865714
Epoch: 39   Training Loss: 4.818829     Validation Loss: 4.852709
Epoch: 40   Training Loss: 4.818889     Validation Loss: 4.859948
Epoch: 41   Training Loss: 4.817372     Validation Loss: 4.854918
Epoch: 42   Training Loss: 4.815913     Validation Loss: 4.868264
Epoch: 43   Training Loss: 4.816840     Validation Loss: 4.868748
Epoch: 44   Training Loss: 4.816405     Validation Loss: 4.862210
Epoch: 45   Training Loss: 4.816432     Validation Loss: 4.859618
Epoch: 46   Training Loss: 4.817708     Validation Loss: 4.873077
Epoch: 47   Training Loss: 4.818389     Validation Loss: 4.861778
Epoch: 48   Training Loss: 4.815666     Validation Loss: 4.863521
Epoch: 49   Training Loss: 4.813986     Validation Loss: 4.858815
Epoch: 50   Training Loss: 4.813973     Validation Loss: 4.843730
Saving the model...
Epoch: 51   Training Loss: 4.813049     Validation Loss: 4.869843
```

```
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.c1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3)
        self.c2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=5)
        self.fc1 = nn.Linear(in_features=33856, out_features=128)
        self.fc2 = nn.Linear(in_features=128, out_features=64)
        self.fc3 = nn.Linear(in_features=64, out_features=133)
```

```
    def forward(self, x):
        ## Define forward behavior
        x = F.relu(F.max_pool2d(self.c1(x), 3))
        x = F.relu(F.max_pool2d(self.c2(x), 3))
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.softmax(self.fc3(x), dim=1)

        return x
Epoch: 1    Training Loss: 4.889660     Validation Loss: 4.882358
Saving the model...
Epoch: 2    Training Loss: 4.880186     Validation Loss: 4.878861
Saving the model...
Epoch: 3    Training Loss: 4.870876     Validation Loss: 4.871905
Saving the model...
Epoch: 4    Training Loss: 4.861131     Validation Loss: 4.868393
Saving the model...
Epoch: 5    Training Loss: 4.854351     Validation Loss: 4.863762
Saving the model...
Epoch: 6    Training Loss: 4.849358     Validation Loss: 4.867866
Epoch: 7    Training Loss: 4.844779     Validation Loss: 4.868028
Epoch: 8    Training Loss: 4.840405     Validation Loss: 4.865107
Epoch: 9    Training Loss: 4.834197     Validation Loss: 4.868524
Epoch: 10   Training Loss: 4.829739     Validation Loss: 4.869371
Epoch: 11   Training Loss: 4.827268     Validation Loss: 4.865402
Epoch: 12   Training Loss: 4.823582     Validation Loss: 4.864699
Epoch: 13   Training Loss: 4.822828     Validation Loss: 4.864300
Epoch: 14   Training Loss: 4.819210     Validation Loss: 4.865251
Epoch: 15   Training Loss: 4.816064     Validation Loss: 4.867522
Epoch: 16   Training Loss: 4.814193     Validation Loss: 4.865754
Epoch: 17   Training Loss: 4.811436     Validation Loss: 4.871307
Epoch: 18   Training Loss: 4.810716     Validation Loss: 4.867208
Epoch: 19   Training Loss: 4.806495     Validation Loss: 4.868084
Epoch: 20   Training Loss: 4.804619     Validation Loss: 4.867496
Epoch: 21   Training Loss: 4.800927     Validation Loss: 4.872746
Epoch: 22   Training Loss: 4.797945     Validation Loss: 4.866517
Epoch: 23   Training Loss: 4.795191     Validation Loss: 4.864305
Epoch: 24   Training Loss: 4.790440     Validation Loss: 4.868612
Epoch: 25   Training Loss: 4.788748     Validation Loss: 4.866583
Epoch: 26   Training Loss: 4.787388     Validation Loss: 4.865874
Epoch: 27   Training Loss: 4.783623     Validation Loss: 4.865449
Epoch: 28   Training Loss: 4.782757     Validation Loss: 4.868507
Epoch: 29   Training Loss: 4.779568     Validation Loss: 4.865638
Epoch: 30   Training Loss: 4.778420     Validation Loss: 4.861831
```

```
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.c1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3)
        self.c2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=5)
        self.c3 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=7)
        self.fc1 = nn.Linear(in_features=1600, out_features=128)
        self.fc2 = nn.Linear(in_features=128, out_features=64)
        self.fc3 = nn.Linear(in_features=64, out_features=133)

    def forward(self, x):
        ## Define forward behavior
        x = F.relu(F.max_pool2d(self.c1(x), 3))
        x = F.relu(F.max_pool2d(self.c2(x), 3))
        x = F.relu(F.max_pool2d(self.c3(x), 3))
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.softmax(self.fc3(x), dim=1)

        return x
```

```
Epoch: 1    Training Loss: 4.889463     Validation Loss: 4.887413
Saving the model...
Epoch: 2    Training Loss: 4.884832     Validation Loss: 4.877435
Saving the model...
Epoch: 3    Training Loss: 4.874662     Validation Loss: 4.875548
Saving the model...
Epoch: 4    Training Loss: 4.869679     Validation Loss: 4.869301
Saving the model...
Epoch: 5    Training Loss: 4.865074     Validation Loss: 4.870189
Epoch: 6    Training Loss: 4.861003     Validation Loss: 4.871571
Epoch: 7    Training Loss: 4.859458     Validation Loss: 4.863881
Saving the model...
Epoch: 8    Training Loss: 4.855563     Validation Loss: 4.864351
Epoch: 9    Training Loss: 4.856219     Validation Loss: 4.871207
Epoch: 10   Training Loss: 4.853308     Validation Loss: 4.862496
Saving the model...
Epoch: 11   Training Loss: 4.850274     Validation Loss: 4.863542
Epoch: 12   Training Loss: 4.846232     Validation Loss: 4.860454
Saving the model...
Epoch: 13   Training Loss: 4.842957     Validation Loss: 4.859951
Saving the model...
Epoch: 14   Training Loss: 4.841687     Validation Loss: 4.861704
Epoch: 15   Training Loss: 4.840243     Validation Loss: 4.859412
Saving the model...
```

```
Epoch: 16   Training Loss: 4.835746     Validation Loss: 4.862660
Epoch: 17   Training Loss: 4.834301     Validation Loss: 4.861983
```

```
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.c1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3)
        self.fc1 = nn.Linear(in_features=175232, out_features=128)
        self.fc2 = nn.Linear(in_features=128, out_features=64)
        self.fc3 = nn.Linear(in_features=64, out_features=133)

    def forward(self, x):
        ## Define forward behavior
        x = F.relu(F.max_pool2d(self.c1(x), 3))
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.softmax(self.fc3(x), dim=1)

        return x
```

```
Epoch: 1    Training Loss: 4.889722     Validation Loss: 4.888547
Saving the model...
Epoch: 2    Training Loss: 4.884030     Validation Loss: 4.882665
Saving the model...
Epoch: 3    Training Loss: 4.875358     Validation Loss: 4.862727
Saving the model...
Epoch: 4    Training Loss: 4.871405     Validation Loss: 4.869135
Epoch: 5    Training Loss: 4.867872     Validation Loss: 4.875798
Epoch: 6    Training Loss: 4.865258     Validation Loss: 4.867260
Epoch: 7    Training Loss: 4.862958     Validation Loss: 4.869361
Epoch: 8    Training Loss: 4.861319     Validation Loss: 4.870008
Epoch: 9    Training Loss: 4.859105     Validation Loss: 4.864526
Epoch: 10   Training Loss: 4.856936     Validation Loss: 4.868347
Epoch: 11   Training Loss: 4.855128     Validation Loss: 4.860038
Saving the model...
```

And learning rate search:

```
optimizer_scratch = optim.SGD(model_scratch.parameters(), lr = 0.05)
```

```
Accuracy test batch: 2%
Epoch: 1    Training Loss: 4.863262     Validation Loss: 4.758816
Saving the model...
Accuracy test batch: 5%
Epoch: 2    Training Loss: 4.577145     Validation Loss: 4.361482
```

```
Saving the model...
Accuracy test batch: 4%
Epoch: 3    Training Loss: 4.287975     Validation Loss: 4.398997
Accuracy test batch: 9%
Epoch: 4    Training Loss: 4.087015     Validation Loss: 4.049116
Saving the model...
Accuracy test batch: 11%
Epoch: 5    Training Loss: 3.850907     Validation Loss: 4.171186
Accuracy test batch: 12%
Epoch: 6    Training Loss: 3.542592     Validation Loss: 4.063121
Accuracy test batch: 24%
Epoch: 7    Training Loss: 3.122781     Validation Loss: 3.104610
Saving the model...
Accuracy test batch: 46%
Epoch: 8    Training Loss: 2.551389     Validation Loss: 1.942181
Saving the model...
Accuracy test batch: 45%
Epoch: 9    Training Loss: 1.775914     Validation Loss: 2.498568
Accuracy test batch: 61%
Epoch: 10   Training Loss: 1.014163     Validation Loss: 2.024113

Test Loss: 8.157884


Test Accuracy:  6% (56/836)
```

```
optimizer_scratch = optim.Adam(model_scratch.parameters(), lr = 0.0001)
```

```
Accuracy test batch: 3%
Epoch: 1    Training Loss: 4.839472     Validation Loss: 4.693197
Saving the model...
Accuracy test batch: 5%
Epoch: 2    Training Loss: 4.502904     Validation Loss: 4.443671
Saving the model...
Accuracy test batch: 7%
Epoch: 3    Training Loss: 4.174366     Validation Loss: 4.257962
Saving the model...
Accuracy test batch: 8%
Epoch: 4    Training Loss: 3.914688     Validation Loss: 4.153160
Saving the model...
Accuracy test batch: 9%
Epoch: 5    Training Loss: 3.634557     Validation Loss: 4.164093
Accuracy test batch: 11%
Epoch: 6    Training Loss: 3.357990     Validation Loss: 4.125675
Saving the model...
Accuracy test batch: 10%
Epoch: 7    Training Loss: 3.045627     Validation Loss: 4.194654
Accuracy test batch: 9%
```

```
Epoch: 8    Training Loss: 2.710299     Validation Loss: 4.307045
Accuracy test batch: 11%
Epoch: 9    Training Loss: 2.326876     Validation Loss: 4.430572
Accuracy test batch: 10%
Epoch: 10   Training Loss: 1.937403     Validation Loss: 4.674092
Accuracy test batch: 11%
Epoch: 11   Training Loss: 1.544526     Validation Loss: 4.992631
Accuracy test batch: 10%
Epoch: 12   Training Loss: 1.179304     Validation Loss: 5.287921
Accuracy test batch: 11%
Epoch: 13   Training Loss: 0.820766     Validation Loss: 5.768975
Accuracy test batch: 10%
Epoch: 14   Training Loss: 0.540662     Validation Loss: 6.483326

optimizer_scratch = optim.Adam(model_scratch.parameters(), lr=0.001)
Accuracy test batch: 4%
Epoch: 1    Training Loss: 4.797839     Validation Loss: 4.590925
Saving the model...
Accuracy test batch: 6%
Epoch: 2    Training Loss: 4.380329     Validation Loss: 4.352962
Saving the model...
Accuracy test batch: 7%
Epoch: 3    Training Loss: 4.043909     Validation Loss: 4.165596
Saving the model...
Accuracy test batch: 10%
Epoch: 4    Training Loss: 3.736174     Validation Loss: 4.043446
Saving the model...
Accuracy test batch: 11%
Epoch: 5    Training Loss: 3.322265     Validation Loss: 4.079185
Accuracy test batch: 12%
Epoch: 6    Training Loss: 2.845153     Validation Loss: 4.243596
Accuracy test batch: 12%
Epoch: 7    Training Loss: 2.276746     Validation Loss: 4.699338
Accuracy test batch: 12%
Epoch: 8    Training Loss: 1.680334     Validation Loss: 5.020378
Accuracy test batch: 12%
Epoch: 9    Training Loss: 1.155823     Validation Loss: 5.998614
Accuracy test batch: 12%
Epoch: 10   Training Loss: 0.737184     Validation Loss: 6.429693
```

Adam optimizer brings best results in terms of speed of loss decrease and accuracy. The most optimal learning rata for Adam is 0.0005. The fastest.

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 [3]: import torch.optim as optim

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

        ### TODO: select optimizer
        optimizer_scratch = optim.Adam(model_scratch.parameters(), lr = 0.0005)
```

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

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

                ###################
                # train the model #
                ###################
                model.train()
                for batch_idx, (data, target) in enumerate(loaders['train']):
                    # move to GPU
                    if use_cuda:
                        data, target = data.cuda(), target.cuda()
                    ## find the loss and update the model parameters accordingly
                    ## record the average training loss, using something like
                    ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_l

                    # ATTENTION: I added this code myself
                    output = model(data)
                    loss = criterion(output, target)
                    # print("Loss: {}".format(loss))
                    train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss
                    optimizer.zero_grad()
                    loss.backward()
                    optimizer.step()


                ######################
                # validate the model #
                ######################
```

```
                model.eval()
                # ATTENTION: Added total_correct and total
                total_correct = 0
                total = 0
                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

                    # ATTENTION: I added this code myself
                    output = model(data)
                    loss = criterion(output, target)
                    valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_loss
                    max_arg_output = torch.argmax(output, dim=1)
                    total_correct += int(torch.sum(max_arg_output == target))
                    total += data.shape[0]
                print('Validation accuracy: {:.0%}'.format(total_correct/total))

                # 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_min > valid_loss:
                    print("Saving the model...")
                    valid_loss_min = valid_loss
                    torch.save(model.state_dict(), save_path)


            # return trained model
            return model
```

In [5]:
```
# train the model
model_scratch = train(50, 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'))
```

```
Validation accuracy: 3%
Epoch: 1 	Training Loss: 4.852280 	Validation Loss: 4.696955
Saving the model...
Validation accuracy: 4%
Epoch: 2 	Training Loss: 4.633039 	Validation Loss: 4.465677
```

```
Saving the model...
Validation accuracy: 5%
Epoch: 3         Training Loss: 4.468325         Validation Loss: 4.326585
Saving the model...
Validation accuracy: 6%
Epoch: 4         Training Loss: 4.365996         Validation Loss: 4.167918
Saving the model...
Validation accuracy: 6%
Epoch: 5         Training Loss: 4.265707         Validation Loss: 4.147124
Saving the model...
Validation accuracy: 7%
Epoch: 6         Training Loss: 4.162678         Validation Loss: 4.005342
Saving the model...
Validation accuracy: 9%
Epoch: 7         Training Loss: 4.091837         Validation Loss: 3.922440
Saving the model...
Validation accuracy: 9%
Epoch: 8         Training Loss: 4.002562         Validation Loss: 3.888850
Saving the model...
Validation accuracy: 9%
Epoch: 9         Training Loss: 3.949916         Validation Loss: 3.796360
Saving the model...
Validation accuracy: 12%
Epoch: 10         Training Loss: 3.885401         Validation Loss: 3.783873
Saving the model...
Validation accuracy: 10%
Epoch: 11         Training Loss: 3.837733         Validation Loss: 3.841457
Validation accuracy: 12%
Epoch: 12         Training Loss: 3.741338         Validation Loss: 3.708586
Saving the model...
Validation accuracy: 13%
Epoch: 13         Training Loss: 3.712761         Validation Loss: 3.674278
Saving the model...
Validation accuracy: 14%
Epoch: 14         Training Loss: 3.640559         Validation Loss: 3.587135
Saving the model...
Validation accuracy: 14%
Epoch: 15         Training Loss: 3.596549         Validation Loss: 3.584131
Saving the model...
Validation accuracy: 16%
Epoch: 16         Training Loss: 3.546422         Validation Loss: 3.532239
Saving the model...
Validation accuracy: 16%
Epoch: 17         Training Loss: 3.484814         Validation Loss: 3.455312
Saving the model...
Validation accuracy: 17%
Epoch: 18         Training Loss: 3.470113         Validation Loss: 3.451111
Saving the model...
```

```
Validation accuracy: 17%
Epoch: 19       Training Loss: 3.406382         Validation Loss: 3.370838
Saving the model...
Validation accuracy: 16%
Epoch: 20       Training Loss: 3.344461         Validation Loss: 3.400913
Validation accuracy: 20%
Epoch: 21       Training Loss: 3.287737         Validation Loss: 3.344915
Saving the model...
Validation accuracy: 17%
Epoch: 22       Training Loss: 3.247657         Validation Loss: 3.310797
Saving the model...
Validation accuracy: 19%
Epoch: 23       Training Loss: 3.229870         Validation Loss: 3.372190
Validation accuracy: 21%
Epoch: 24       Training Loss: 3.189680         Validation Loss: 3.415124
Validation accuracy: 19%
Epoch: 25       Training Loss: 3.139629         Validation Loss: 3.340824
Validation accuracy: 20%
Epoch: 26       Training Loss: 3.100478         Validation Loss: 3.202381
Saving the model...
Validation accuracy: 22%
Epoch: 27       Training Loss: 3.040116         Validation Loss: 3.190710
Saving the model...
Validation accuracy: 20%
Epoch: 28       Training Loss: 3.029283         Validation Loss: 3.349988
Validation accuracy: 22%
Epoch: 29       Training Loss: 2.986070         Validation Loss: 3.276131
Validation accuracy: 23%
Epoch: 30       Training Loss: 2.945365         Validation Loss: 3.161633
Saving the model...
Validation accuracy: 21%
Epoch: 31       Training Loss: 2.887540         Validation Loss: 3.163230
Validation accuracy: 23%
Epoch: 32       Training Loss: 2.875289         Validation Loss: 3.160801
Saving the model...
Validation accuracy: 21%
Epoch: 33       Training Loss: 2.838518         Validation Loss: 3.377404
Validation accuracy: 22%
Epoch: 34       Training Loss: 2.828271         Validation Loss: 3.213231
Validation accuracy: 22%
Epoch: 35       Training Loss: 2.778944         Validation Loss: 3.226667
Validation accuracy: 23%
Epoch: 36       Training Loss: 2.768426         Validation Loss: 3.144235
Saving the model...
Validation accuracy: 24%
Epoch: 37       Training Loss: 2.718857         Validation Loss: 3.147774
Validation accuracy: 25%
Epoch: 38       Training Loss: 2.664525         Validation Loss: 3.108729
```

```
Saving the model...
Validation accuracy: 24%
Epoch: 39         Training Loss: 2.652055         Validation Loss: 3.132187
Validation accuracy: 24%
Epoch: 40         Training Loss: 2.655876         Validation Loss: 3.165745
Validation accuracy: 25%
Epoch: 41         Training Loss: 2.641927         Validation Loss: 3.121022
Validation accuracy: 25%
Epoch: 42         Training Loss: 2.588422         Validation Loss: 3.052789
Saving the model...
Validation accuracy: 28%
Epoch: 43         Training Loss: 2.592636         Validation Loss: 3.136038
Validation accuracy: 26%
Epoch: 44         Training Loss: 2.558873         Validation Loss: 3.209397
Validation accuracy: 25%
Epoch: 45         Training Loss: 2.493666         Validation Loss: 3.184327
Validation accuracy: 24%
Epoch: 46         Training Loss: 2.465608         Validation Loss: 3.325321
Validation accuracy: 25%
Epoch: 47         Training Loss: 2.446159         Validation Loss: 3.191333
Validation accuracy: 26%
Epoch: 48         Training Loss: 2.469759         Validation Loss: 3.158408
Validation accuracy: 26%
Epoch: 49         Training Loss: 2.431810         Validation Loss: 3.189232
Validation accuracy: 26%
Epoch: 50         Training Loss: 2.376969         Validation Loss: 3.208405
```

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 [7]: 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))
```

In [7]:
```
# call test function
test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 2.993139


Test Accuracy: 27% (230/836)
```

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 [1]:
```
## TODO: Specify data loaders
import numpy as np
import torch
from PIL import ImageFile
from torchvision import transforms,datasets

ImageFile.LOAD_TRUNCATED_IMAGES = True
use_cuda = torch.cuda.is_available()

image_transforms = {'train': transforms.Compose([transforms.RandomRotation(10),
                                    transforms.RandomHorizontalFlip(),
                                    transforms.RandomResizedCrop(224),
```

```
                                              transforms.Resize(224),
                                              transforms.CenterCrop(224),
                                              transforms.ToTensor(),
                                              transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                                   std=[0.229, 0.224, 0.225])]),
                                'valid': 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])]),
                                'test': 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])])
                               }

        loaders_transfer = {'train': torch.utils.data.DataLoader(datasets.ImageFolder('dogImage
                                                                                   transform=
                                                                  batch_size=25, shuffle=True),
                            'valid': torch.utils.data.DataLoader(datasets.ImageFolder('dogImages
                                                                                   transform=
                                                                  batch_size=25, shuffle=True),
                            'test': torch.utils.data.DataLoader(datasets.ImageFolder('dogImages/
                                                                                  transform=
                                                                  batch_size=25, shuffle=True)}
```

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 [2]: import torchvision.models as models
        import torch.nn as nn

        ## TODO: Specify model architecture
        import torch.nn.functional as F

        model_transfer = models.resnet18(pretrained=True)

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

        num_ftrs = model_transfer.fc.in_features

        class TransferLearning(nn.Module):
            def __init__(self):
                super(TransferLearning, self).__init__()
```

```
            self.fc1 = nn.Linear(in_features=num_ftrs, out_features=256)
            self.fc2 = nn.Linear(in_features=256, out_features=133)

        def forward(self, x):
            x = F.relu(self.fc1(x))
            x = self.fc2(x)
            return x

    model_transfer.fc = TransferLearning()

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

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

Answer: Having much experience in previous task I decided to stop on shallow last layer. And it had reached test accuracy above 60%. Truly saying I tried only 5 architectures which is unusual for me but experience in previous task helped me to stop on the best and fastest model in shortest time.

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 [3]: import torch.optim as optim

        criterion_transfer = torch.nn.CrossEntropyLoss()
        optimizer_transfer = optim.Adam(model_transfer.parameters(), lr = 0.0005)
```

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 [8]: # load the model that got the best validation accuracy (uncomment the line below)
        # train the model
        model_transfer = train(50, loaders_transfer, model_transfer, optimizer_transfer, crite
        model_transfer.load_state_dict(torch.load('model_transfer.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 [9]: test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)

Test Loss: 0.512415
```

```
Test Accuracy: 84% (703/836)
```

1.1.17 (IMPLEMENTATION) Predict Dog Breed with the Model

Write a function that takes an image path as input and returns the dog breed (`Affenpinscher`, `Afghan hound`, etc) that is predicted by your model.

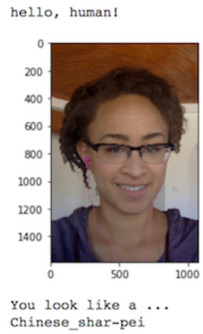
```
In [101]: ### TODO: Write a function that takes a path to an image as input
          ### and returns the dog breed that is predicted by the model.

          # list of class names by index, i.e. a name can be accessed like class_names[0]
          from PIL import Image
          from torchvision  import transforms
          from torch.nn.functional import softmax
          class_names = [item[4:].replace("_", " ") for item in loaders_transfer['train'].datas

          def predict_breed_transfer(img_path):
              # load the image and return the predicted breed
              pil_image = Image.open(img_path)
              transformed_image = transforms.Resize(224)(pil_image)
              transformed_image = transforms.CenterCrop(224)(pil_image)
              transformed_image = transforms.ToTensor()(transformed_image)
              transformed_image = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                       std=[0.229, 0.224, 0.225])(transformed_

              transformed_image = transformed_image.cuda() if torch.cuda.is_available() else t
              output_raw = model_transfer(transformed_image.unsqueeze(0))
              output_softmax = softmax(output_raw, dim=1)
              output = int(output_raw.argmax())

              return class_names[output], float(output_softmax[0][output])
```

Step 5: Write your Algorithm

Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then, - if a **dog** is detected in the image, return the predicted breed. - if a **human** is detected in the image, return the resembling dog breed. - if **neither** is detected in the image, provide output that indicates an error.

You are welcome to write your own functions for detecting humans and dogs in images, but feel free to use the `face_detector` and `human_detector` functions developed above. You are **required** to use your CNN from Step 4 to predict dog breed.

Some sample output for our algorithm is provided below, but feel free to design your own user experience!



Sample Human Output

1.1.18 (IMPLEMENTATION) Write your Algorithm

```
In [138]: ### TODO: Write your algorithm.
          ### Feel free to use as many code cells as needed.
          from PIL import Image
          import matplotlib.pyplot as plt
          from IPython import display


          def run_app(img_path):
              ## handle cases for a human face, dog, and neither
              face = face_detector(img_path)
              breed, confidence = predict_breed_transfer(img_path)
              image_plt = plt.imread(img_path)
              if confidence < 0.3:
                  raise Exception('Niether dog or human are predicted.')
              elif face:
                  plt.imshow(image_plt)
                  plt.title('hello, human! You look like: {}'.format(breed))
              else:
                  plt.imshow(image_plt)
                  plt.title('hello, dog! You are: {}'.format(breed))



              return breed
```

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)

ATTENTION: I took photos from Google Images that are labeled for reuse and modificatation. The output satisfied me. There is only one incorrectly classified picture of Golden retriever that was recognized by `face_detector()` but transfer learning algorithm correctly classified it as Golden retriever. I tried my best to make it in a minmum code.

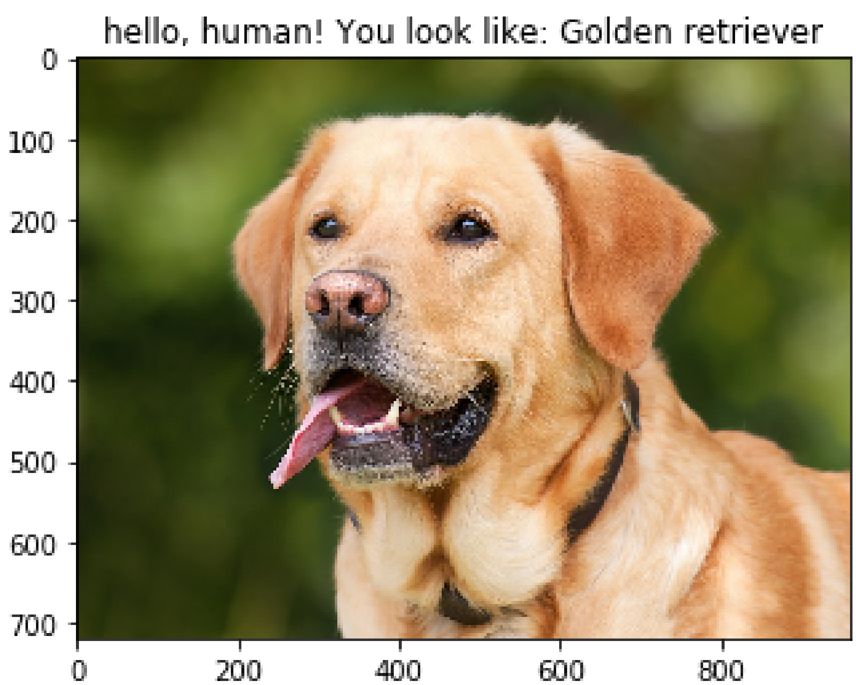
1) Making a class that is responsible for the whole process is a good solution and it's logical and scalable instead of a function.
2) Softmax is slow. I used it. We can try to take raw tensor output but it requires addional research regarding threshold.
3) I might made a function that process images it would be more DRY.
4) List comprehension might be used to process images instead of that I used usual function call.
5) I could have iterate the list and show images. But I encountered an error with an image so I have chosen safest way to debug.
6) The output image might be nicer but I decided to use Agile sprint to complete the project.
7) Speed might be impoved by converting using PyTorch JIT compiler to make C++ code.
8) The model might be improved by making it more deep.
9) I might try another built-in models and maybe search the internet to find a better model than provided by PyTorch.
10) Face detector algorithm might be improved by adding better Haar like features.
11) I could make threshole better by playing more with it.

In [140]:
```
## 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
run_app('add/dog_1.jpg')
```

Out[140]:
```
'Border terrier'
```

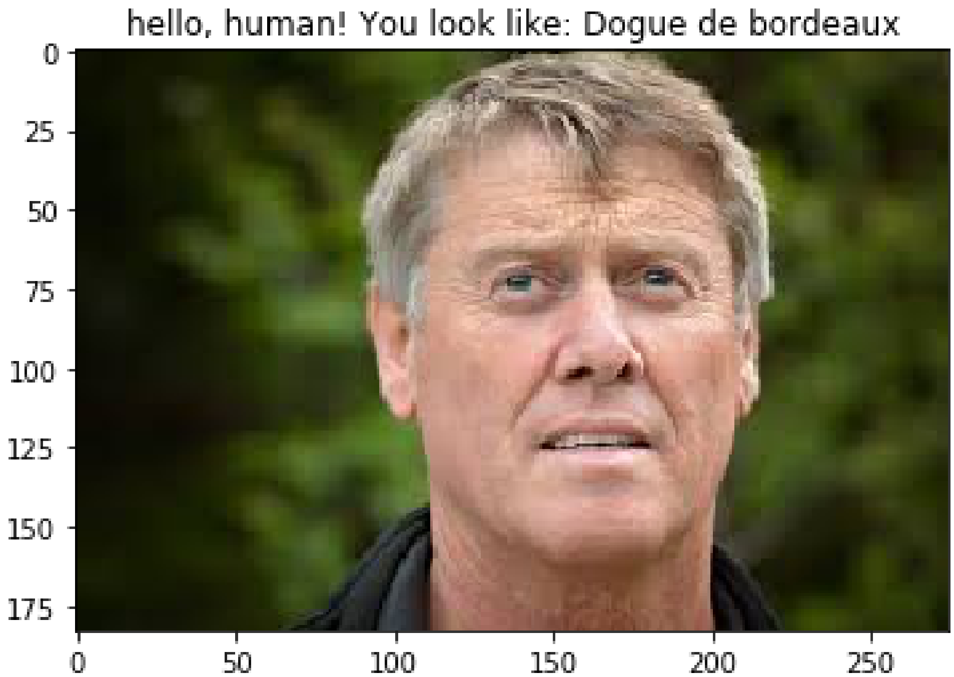

```
In [134]: run_app('add/dog_2.jpg')
```

```
Out[134]: 'Golden retriever'
```



In [135]: run_app('add/face_1.jpg')

Out[135]: 'Dogue de bordeaux'



In [136]: run_app('add/face_2.jpg')

Out[136]: 'Irish wolfhound'


```
In [145]: run_app('add/other_2.jpg')
```

```
---------------------------------------------------------------------------

Exception                                 Traceback (most recent call last)

<ipython-input-145-710674a31b9c> in <module>
----> 1 run_app('add/other_2.jpg')


<ipython-input-138-215fd8daccdf> in run_app(img_path)
     12     image_plt = plt.imread(img_path)
     13     if confidence < 0.3:
---> 14         raise Exception('Niether dog or human are predicted.')
     15     elif face:
     16         plt.imshow(image_plt)


Exception: Niether dog or human are predicted.
```

```
In [147]: run_app('add/other_1.jpg')
```

```
---------------------------------------------------------------------------
```

```
    Exception                                 Traceback (most recent call last)

    <ipython-input-147-ab2150be5e60> in <module>
----> 1 run_app('add/other_1.jpg')


    <ipython-input-138-215fd8daccdf> in run_app(img_path)
     12     image_plt = plt.imread(img_path)
     13     if confidence < 0.3:
---> 14         raise Exception('Niether dog or human are predicted.')
     15     elif face:
     16         plt.imshow(image_plt)


    Exception: Niether dog or human are predicted.
```