dog_app

February 22, 2019

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

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

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

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

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

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

Step 0: Import Datasets

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

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

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

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

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

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

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

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

Step 1: Detect Humans

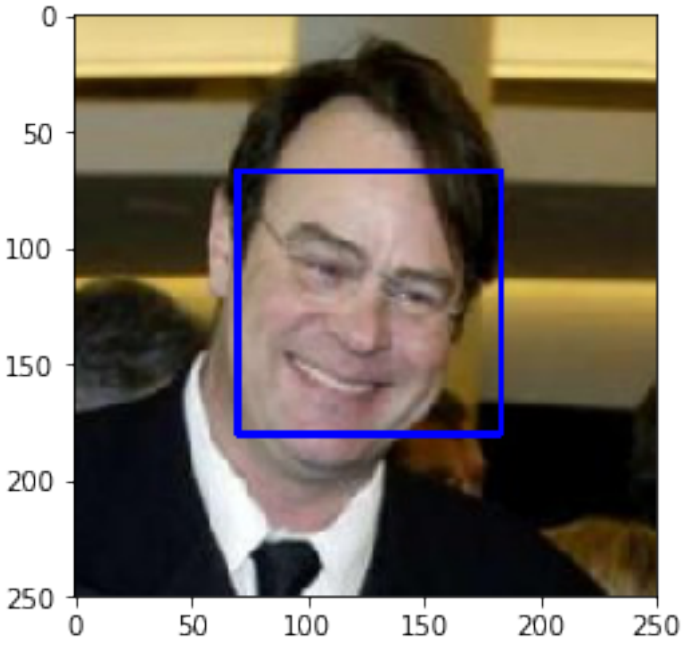
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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

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

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

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

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

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

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

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

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

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



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

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

1.1.1 Write a Human Face Detector

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

```
In [3]: # returns "True" if face is detected in image stored at img_path
        def face_detector(img_path):
```

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

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

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

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

Answer: - 98/100 of human files have a detected human face, which is great - 17/100 of dog files have a detected human face, which is weird, this number should be really low

In [4]:
```
from tqdm import tqdm

human_files_short = human_files[:100]
dog_files_short = dog_files[:100]

#-#-# Do NOT modify the code above this line. #-#-#

## TODO: Test the performance of the face_detector algorithm
## on the images in human_files_short and dog_files_short.
human_count = 0
for h_file in human_files_short:
    if face_detector(h_file):
        human_count += 1
print("%d/100 human images were marked as having human faces (which should be high)" % h

dog_count = 0
for d_file in dog_files_short:
    if face_detector(d_file):
        dog_count += 1
print("%d/100 dog images were marked as having human faces (which shouldn't be high)" %
```

```
98/100 human images were marked as having human faces (which should be high)
17/100 dog images were marked as having human faces (which shouldn't be high)
```

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 [4]: import torch
        import torchvision.models as models

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

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

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

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:05<00:00, 98970073.75it/s]
```

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

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

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

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

```
In [11]: 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
    transform = transforms.Compose((
        transforms.Resize((224, 224)),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
    ))

    in_layer = transform(Image.open(img_path))
    #add batch dimension
    in_layer = in_layer.unsqueeze_(0)
    if use_cuda:
        in_layer = in_layer.cuda()
    out_layer = VGG16(in_layer)[0]
    mx, mx_idx = out_layer.max(0)

    return mx_idx
```

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 [6]:
```
### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    idx = VGG16_predict(img_path)
    if idx >= 151 and idx <= 268:
        return True
    else:
        return 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: - 0/100 of human images were marked as dog, which is fantastic! - 100/100 of dog images were marked as dog, wow!

much better than the opencv detector

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

human_files_short = human_files[:100]
dog_files_short = dog_files[:100]

human_count = 0
for h_file in human_files_short:
    if dog_detector(h_file):
        human_count += 1
print("%d/100 human images were marked as being dog (which should be low)" % human_count

dog_count = 0
for d_file in dog_files_short:
    if dog_detector(d_file):
        dog_count += 1
print("%d/100 dog images were marked as being dog (which should be high)" % dog_count)
```

```
0/100 human images were marked as being dog (which should be low)
100/100 dog images were marked as being dog (which should be high)
```

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

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

Step 3: Create a CNN to Classify Dog Breeds (from Scratch)

Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of at least 10%. In Step 4 of this notebook, you will have the opportunity to use transfer learning to create a CNN that attains greatly improved accuracy.

We mention that the task of assigning breed to dogs from images is considered exceptionally challenging. To see why, consider that *even a human* would have trouble distinguishing between a Brittany and a Welsh Springer Spaniel.

Brittany	Welsh Springer Spaniel

It is not difficult to find other dog breed pairs with minimal inter-class variation (for instance, Curly-Coated Retrievers and American Water Spaniels).

Curly-Coated Retriever	American Water Spaniel

Likewise, recall that labradors come in yellow, chocolate, and black. Your vision-based algorithm will have to conquer this high intra-class variation to determine how to classify all of these different shades as the same breed.

Yellow Labrador	Chocolate Labrador

We also mention that random chance presents an exceptionally low bar: setting aside the fact that the classes are slightly imabalanced, a random guess will provide a correct answer roughly 1 in 133 times, which corresponds to an accuracy of less than 1%.

Remember that the practice is far ahead of the theory in deep learning. Experiment with many different architectures, and trust your intuition. And, of course, have fun!

1.1.7 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

Use the code cell below to write three separate data loaders for the training, validation, and test datasets of dog images (located at `dog_images/train`, `dog_images/valid`, and `dog_images/test`, respectively). You may find this documentation on custom datasets to be a useful resource. If you are interested in augmenting your training and/or validation data, check out the wide variety of transforms!

```
In [7]: import os
        from torchvision import transforms
        from torchvision import datasets
        from torch.utils.data import DataLoader

        ### TODO: Write data loaders for training, validation, and test sets
        ## Specify appropriate transforms, and batch_sizes
        data_trans = {
            "train": 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])
            )),
            "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])
        )),
    }

    data_folders = {
        "train": datasets.ImageFolder("data/dogImages/train",
                                      transform=data_trans["train"]),
        "valid": datasets.ImageFolder("data/dogImages/valid",
                                      transform=data_trans["valid"]),
        "test": datasets.ImageFolder("data/dogImages/test",
                                     transform=data_trans["test"]),
    }

    data_loaders = {
        "train": DataLoader(data_folders["train"], batch_size=64, shuffle=True),
        "valid": DataLoader(data_folders["test"], batch_size=64, shuffle=True),
        "test": DataLoader(data_folders["train"], batch_size=64),
    }
```

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

Answer: - I first make a resize to 224x224, and since this doesn't always give the exact size, I will center crop to 224x224. I used this size because it allows to use more kernels without having a very large vector for the last fully conected layer. - No, since it converges slowly when I use data augmentation. It's true that it will generalize better for new data, but the goal here is to achieve 10% accuracy

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [31]: import torch
         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__()

        self.c11 = nn.Conv2d(3, 8, 3, padding=1)
        self.c12 = nn.Conv2d(8, 8, 3, padding=1)

        self.c21 = nn.Conv2d(8, 16, 3, padding=1)
        self.c22 = nn.Conv2d(16, 16, 3, padding=1)

        self.c31 = nn.Conv2d(16, 32, 3, padding=1)
        self.c32 = nn.Conv2d(32, 32, 3, padding=1)


        self.l1 = nn.Linear(28*28 * 32, 1024)
        self.l2 = nn.Linear(1024, 512)
        self.l3 = nn.Linear(512, 133)

    def forward(self, x):
        # (224, 224, 3)
        x = F.relu(self.c11(x))
        x = F.relu(self.c12(x))
        x = F.max_pool2d(x, kernel_size=2, stride=2)

        # (112, 112, 8)
        x = F.relu(self.c21(x))
        x = F.relu(self.c22(x))
        x = F.max_pool2d(x, kernel_size=2, stride=2)

        # (56, 56, 16)
        x = F.relu(self.c31(x))
        x = F.relu(self.c32(x))
        x = F.max_pool2d(x, kernel_size=2, stride=2)

        # (28, 28, 32)
        batch_size = x.shape[0]
        x = x.reshape(batch_size, -1)
        x = F.relu(self.l1(x))
        x = F.relu(self.l2(x))
        x = self.l3(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
```

```
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 was convinced that 28x28x32 should be the input size of my fully connected layer (more can make it slower to train) - By intuition, I feel like any conv layer should start by increasing the number of filters, than stabilize it to sort of get more knowledge about the picture before increasing the filter another time, this is why I always use the Conv2D(n,n). - I looked for different architecture for classifying objects to get some intuition. - I finally tried different architecture of mine that can get more than 10% accuracy. - I tried to keep it as simple as possible to not suffer from vanishing gradient.

PS: Comments specify input size for each conv layer in the forward function

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 [32]: import torch.optim as optim

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

         ### TODO: select optimizer
         optimizer_scratch = optim.Adadelta(model_scratch.parameters())
```

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'`.

I've faced this error during the training: image file is truncated (150 bytes not processed)

After checking for it, I found a piece of code from stackoverflow that solve the problem - link: https://stackoverflow.com/questions/12984426/python-pil-ioerror-image-file-truncated-with-big-images

```
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True
```

```
In [33]: import numpy as np
         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()
        out = model(data)
        loss = criterion(out, target)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()

    ######################
    # 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
        out = model(data)
        loss = criterion(out, target)
        valid_loss += loss.item()


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

    ## TODO: save the model if validation loss has decreased
    if valid_loss < valid_loss_min:
        valid_loss_min = valid_loss
        torch.save(model.state_dict(), save_path)
```

```
    return model


# train the model
from workspace_utils import active_session

with active_session():
    model_scratch = train(20, data_loaders, 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: 512.783349         Validation Loss: 68.232081
Epoch: 2         Training Loss: 510.175934         Validation Loss: 67.774526
Epoch: 3         Training Loss: 494.404712         Validation Loss: 65.286802
Epoch: 4         Training Loss: 478.012915         Validation Loss: 63.045059
Epoch: 5         Training Loss: 469.169941         Validation Loss: 62.112699
Epoch: 6         Training Loss: 456.815971         Validation Loss: 61.209837
Epoch: 7         Training Loss: 442.617017         Validation Loss: 60.163694
Epoch: 8         Training Loss: 421.042714         Validation Loss: 58.654746
Epoch: 9         Training Loss: 391.574140         Validation Loss: 66.350140
Epoch: 10         Training Loss: 325.200504         Validation Loss: 71.284834
Epoch: 11         Training Loss: 191.869512         Validation Loss: 95.678441
Epoch: 12         Training Loss: 49.658314         Validation Loss: 126.090504
Epoch: 13         Training Loss: 8.842857         Validation Loss: 169.665504
Epoch: 14         Training Loss: 4.224143         Validation Loss: 152.184014
Epoch: 15         Training Loss: 2.312973         Validation Loss: 132.781406
Epoch: 16         Training Loss: 1.461084         Validation Loss: 130.415972
Epoch: 17         Training Loss: 0.610991         Validation Loss: 164.066806
Epoch: 18         Training Loss: 0.803318         Validation Loss: 137.207398
Epoch: 19         Training Loss: 0.769306         Validation Loss: 157.011835
Epoch: 20         Training Loss: 0.781500         Validation Loss: 95.888463
```

We see that it starts overfitting from the 8th epoch, but since we are saving the model that gives the best validation loss, we will use the non overfitting model.

1.1.11 (IMPLEMENTATION) Test the Model

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

```
In [34]: import numpy as np

         from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True
```

```
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
model_scratch = Net()
# move tensors to GPU if CUDA is available
use_cuda = torch.cuda.is_available()
if use_cuda:
    model_scratch.cuda()
model_scratch.load_state_dict(torch.load('model_scratch.pt'))
test(data_loaders, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.651777


Test Accuracy: 14% (983/6680)
```

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 [3]: ## TODO: Specify data loaders
        loaders_transfer = {
            "train": DataLoader(data_folders["train"], batch_size=16, shuffle=True),
            "valid": DataLoader(data_folders["test"], batch_size=16, shuffle=True),
            "test": DataLoader(data_folders["train"], batch_size=16),
        }
```

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 [5]: import torchvision.models as models
        import torch.nn as nn
        import torch.cuda

        ## TODO: Specify model architecture
        model_transfer = models.densenet161(pretrained=True)
        model_transfer.classifier = nn.Sequential(
            nn.Linear(2208, 1024),
            nn.Linear(1024, 133)
        )

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

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

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: - Densenet161 is a personal preference, since it has proven good results with me, and it's strength against vanishing gradient (which I hate). - Densenet161 uses a final Linear layer of 2208->1000 as a classifier, to adapt it to our problem which requires 133 as size of the output vector, I have added another layer to sort of do the transition from 2208 to 133. The final classifier will go from 2208 to 1024, then from 1024 to 133, which exactly what we expect.

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 [6]: from torch import optim

        criterion_transfer = nn.CrossEntropyLoss()
        optimizer_transfer = optim.Adadelta(model_transfer.parameters())
```

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 [9]: import numpy as np
        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_los
                    optimizer.zero_grad()
                    out = model(data)
                    loss = criterion(out, target)
                    loss.backward()
                    optimizer.step()
                    train_loss += loss.item()

                ######################
                # 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
                    out = model(data)
                    loss = criterion(out, target)
                    valid_loss += loss.item()


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

                ## TODO: save the model if validation loss has decreased
                if valid_loss < valid_loss_min:
                    valid_loss_min = valid_loss
                    torch.save(model.state_dict(), save_path)


            return model
```

In [10]:
```
from workspace_utils import active_session

# train the whole network for 10 epochs
with active_session():
    train(10, loaders_transfer, model_transfer, optimizer_transfer, criterion_transfer,
model_transfer.load_state_dict(torch.load('model_transfer.pt'))
# train the classifier only for another 5 epochs
for param in model_transfer.features.parameters():
    param.requires_grad = False
with active_session():
    train(5, loaders_transfer, model_transfer, optimizer_transfer, criterion_transfer,
model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

```
Epoch: 1         Training Loss: 1337.431220         Validation Loss: 186.135493
Epoch: 2         Training Loss: 1218.305768         Validation Loss: 170.517284
Epoch: 3         Training Loss: 1113.799185         Validation Loss: 253.801962
Epoch: 4         Training Loss: 1015.540295         Validation Loss: 190.111874
Epoch: 5         Training Loss: 918.820572         Validation Loss: 152.274545
Epoch: 6         Training Loss: 841.065305         Validation Loss: 165.975869
Epoch: 7         Training Loss: 757.872044         Validation Loss: 146.477418
Epoch: 8         Training Loss: 683.955789         Validation Loss: 139.245556
```

```
Epoch: 9         Training Loss: 614.056834         Validation Loss: 207.471901
Epoch: 10         Training Loss: 562.759557         Validation Loss: 118.602638
Epoch: 1         Training Loss: 363.740279         Validation Loss: 89.202431
Epoch: 2         Training Loss: 312.606826         Validation Loss: 89.972537
Epoch: 3         Training Loss: 299.806895         Validation Loss: 91.203890
Epoch: 4         Training Loss: 288.317382         Validation Loss: 94.031360
Epoch: 5         Training Loss: 276.630669         Validation Loss: 95.618038
```

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 [11]:
```
import numpy as np

from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

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))
```

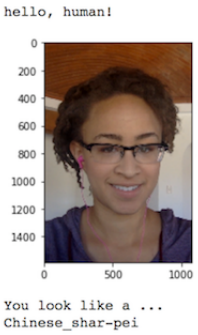
```
model_transfer.load_state_dict(torch.load('model_transfer.pt'))
test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.480887


Test Accuracy: 85% (5683/6680)
```

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 [8]: from PIL import Image
        from torchvision import transforms
        import torchvision.models as models
        import torch.nn as nn
        import torch.cuda


        model = models.densenet161(pretrained=True)
        model.classifier = nn.Sequential(
            nn.Linear(2208, 1024),
            nn.Linear(1024, 133)
        )
        use_cuda = torch.cuda.is_available()
        if use_cuda:
            model = model.cuda()
        model.load_state_dict(torch.load('model_transfer.pt'))

        pred_trans = 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])
            ))

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

        def predict_breed_transfer(img_path):
            img = Image.open(img_path)
            x = pred_trans(img)
            # add batch_size of 1
            x = x.unsqueeze(0)
            if use_cuda:
                x = x.cuda()
```



Sample Human Output

```
    out = model(x)
    _, breed_idx = out.topk(1)
    return class_names[breed_idx]
```

```
/opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-py3.6.egg/torchvision/models/densenet.p
Downloading: "https://download.pytorch.org/models/densenet161-8d451a50.pth" to /root/.torch/mode
100%|| 115730790/115730790 [00:01<00:00, 75345647.62it/s]
```

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!

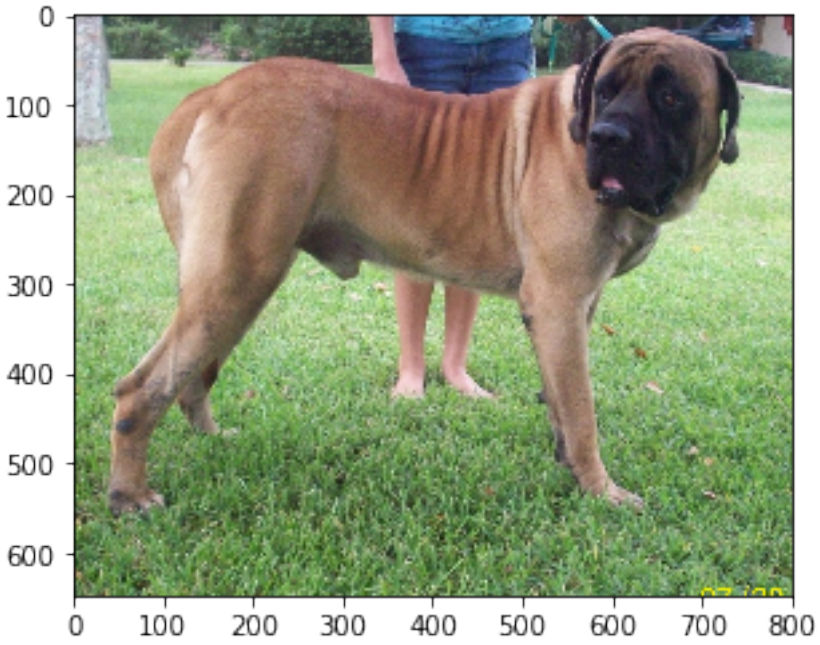
1.1.18 (IMPLEMENTATION) Write your Algorithm

```
In [15]: ### TODO: Write your algorithm.
         ### Feel free to use as many code cells as needed.

         def run_app(img_path):
             img = Image.open(img_path)
             if dog_detector(img_path):
                 print("Hi dog, let me guess your breed...")
                 plt.imshow(img)
                 plt.show()
                 print(predict_breed_transfer(img_path))
             elif face_detector(img_path):
                 print("Hi human, it's better to have a dog that looks like you")
```

```
            print("You will look like a start walking in the street ;)")
            print("You should better get a %s" % predict_breed_transfer(img_path))
            plt.imshow(img)
            plt.show()
        else:
            print("[-] Can't find a dog or a human in this picture")
```

```
In [16]: run_app(np.array(glob("/data/dog_images/*/*/*"))[0])
```

```
Hi dog, let me guess your breed...
```




```
Irish wolfhound
```

Step 6: Test Your Algorithm

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

1.1.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

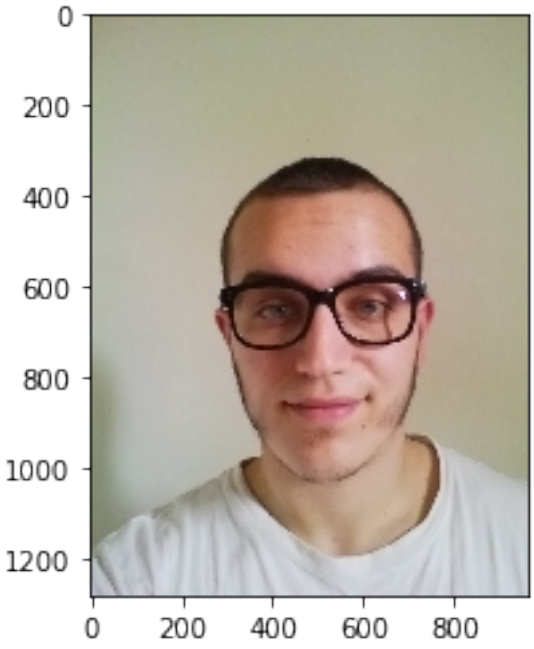
Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

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

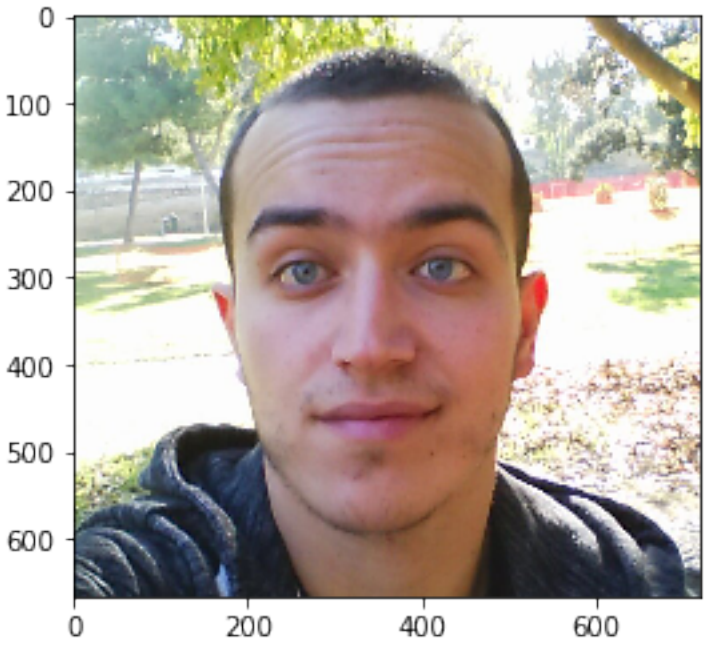
Answer: (Three possible points for improvement) I think my model love Irish wolfhound dogs :p - It doesn't generalize well to data from the web, so maybe it's because data is structured in a specefic manner in our dataset, so I should use data augmetatino techniques or scrap more data from the web - I should use more sophisticated techniques to detect human faces, since a clear image of a human wasn't detected. - What if more than one dog were on the picture, I should make prediction for each one of them separatly

```
In [21]: from glob import glob

         for file in glob("my_imgs/*"):
             run_app(file)
```

```
Hi human, it's better to have a dog that looks like you
You will look like a start walking in the street ;)
You should better get a Irish wolfhound
```




```
Hi human, it's better to have a dog that looks like you
You will look like a start walking in the street ;)
You should better get a Irish wolfhound
```




```
[-] Can't find a dog or a human in this picture
Hi dog, let me guess your breed...
```

```
Lhasa apso
Hi dog, let me guess your breed...
```


```
Irish wolfhound
Hi dog, let me guess your breed...
```

```
Irish wolfhound
```