

dog_app

May 12, 2020

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

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

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

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

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

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

Step 0: Import Datasets

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

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

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

Note: If you are using a Windows machine, you are encouraged to use [7zip](#) to extract the folder.

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

```
In [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))
```

```

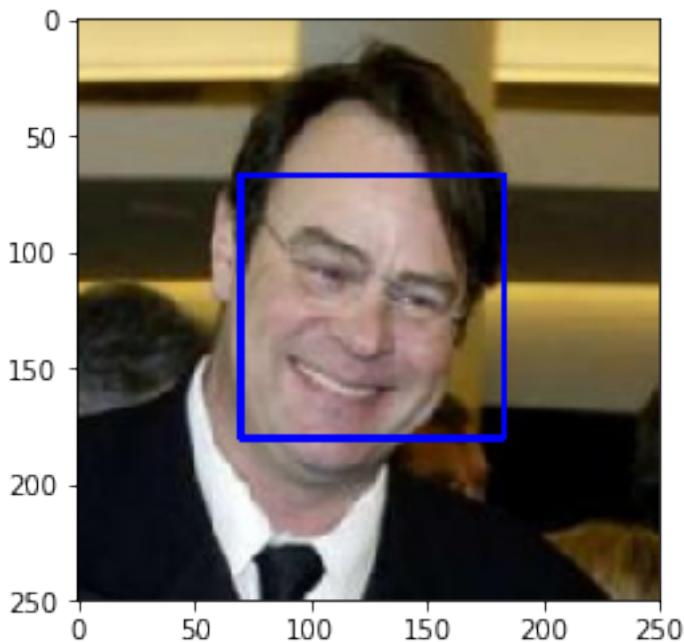
# get bounding box for each detected face
for (x,y,w,h) in faces:
    # add bounding box to color image
    cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)

# 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: - 98.0 % of the first 100 images in `human_files` have a detected human face. - 17.0 % of the first 100 images in `dog_files` have a detected human face.

```
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.
faces_human_files = 0
faces_dog_files = 0

for i in tqdm(human_files_short, position=0):
    faces_human_files += face_detector(i)

for i in tqdm(dog_files_short, position=0):
    faces_dog_files += face_detector(i)

result_human_files = float(faces_human_files) / len(human_files_short)
result_dog_files = float(faces_dog_files) / len(dog_files_short)

print(result_human_files*100, '% of the first 100 images in human_files have detected human face')
print(result_dog_files*100, '% of the first 100 images in dog_files have detected human face')
```

```
100%|| 100/100 [00:02<00:00, 34.67it/s]
100%|| 100/100 [00:30<00:00,  7.21it/s]

98.0 % of the first 100 images in human_files have detected human faces.
17.0 % of the first 100 images in dog_files have detected human faces.
```

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 another face detection algorithm.
### Feel free to use as many code cells as needed.
```

Step 2: Detect Dogs

In this section, we use a [pre-trained model](#) to detect dogs in images.

1.1.3 Obtain Pre-trained VGG-16 Model

The code cell below downloads the VGG-16 model, along with weights that have been trained on [ImageNet](#), a very large, very popular dataset used for image classification and other vision tasks. ImageNet contains over 10 million URLs, each linking to an image containing an object from one of 1000 categories.

```
In [6]: import torch
import torchvision.models as models

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

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

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

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg16-397923af.pth
100%|| 553433881/553433881 [00:05<00:00, 96239346.53it/s]
```

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

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

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

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

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

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

    Args:
        img_path: path to an image

    Returns:
        Index corresponding to VGG-16 model's prediction
    """

    ## TODO: Complete the function.

    ## Load and pre-process an image from the given img_path
    ## Return the *index* of the predicted class for that image
    image = Image.open(img_path)
    transform = transforms.Compose([transforms.CenterCrop(224),
                                    transforms.ToTensor(),
                                    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=
    ])
    image_tensor = transform(image).unsqueeze(0)

    if use_cuda:
        image_tensor = image_tensor.cuda()
    predict=VGG16.eval()(image_tensor)
    predict=predict.data.cpu().numpy()
    return np.argmax(predict) # 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.
    prediction = VGG16_predict(img_path)

    return (prediction in range(151, 268))# true/false
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

Question 2: Use the code cell below to test the performance of your dog_detector function.

- What percentage of the images in human_files_short have a detected dog?
- What percentage of the images in dog_files_short have a detected dog?

Answer:

- 1.0 % of the first 100 images in human_files_short have a detected dog.
- 91.0 % of the first 100 images in dog_files_short have a detected dog.

```
In [9]: ### TODO: Test the performance of the dog_detector function
        ### on the images in human_files_short and dog_files_short.
breed_human_files = 0
breed_dog_files = 0

for i in tqdm(human_files_short, position=0):
    breed_human_files += dog_detector(i)

for i in tqdm(dog_files_short, position=0):
    breed_dog_files += dog_detector(i)

result_human_files = float(breed_human_files) / len(human_files_short)
result_dog_files = float(breed_dog_files) / len(dog_files_short)

print(result_human_files*100, '% of the first 100 images in human_files have detected dogs.')
print(result_dog_files*100, '% of the first 100 images in dog_files have detected dogs.')

100%| 100/100 [00:03<00:00, 31.38it/s]
100%| 100/100 [00:04<00:00, 24.80it/s]

1.0 % of the first 100 images in human_files have detected dogs.
91.0 % of the first 100 images in dog_files have detected dogs.
```

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.

inceptionv3= models.inception_v3(pretrained=True)

if use_cuda:
    inceptionv3 = inceptionv3.cuda()

def dog_detector_inceptionv3(img_path):
    img = Image.open(img_path)
    transform = transforms.Compose([transforms.CenterCrop(299),
                                    transforms.Resize(299),
                                    transforms.ToTensor(),
                                    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])])
    img_tensor = transform(img).unsqueeze(0)
    if use_cuda:
        img_tensor = img_tensor.cuda()
    predict=inceptionv3.eval()(img_tensor)
    predict=predict.data.cpu().numpy()
    return (np.argmax(predict) in range(151, 269))

breed_human_files = 0
breed_dog_files = 0

for i in tqdm(human_files_short, position=0):
    breed_human_files += dog_detector_inceptionv3(i)

for i in tqdm(dog_files_short, position=0):
    breed_dog_files += dog_detector_inceptionv3(i)

result_human_files = float(breed_human_files) / len(human_files_short)
result_dog_files = float(breed_dog_files) / len(dog_files_short)

print(result_human_files*100, '% of the first 100 images in human_files have detected dogs.')
print(result_dog_files*100, '% of the first 100 images in dog_files have detected dogs.')

Downloading: "https://download.pytorch.org/models/inception_v3_google-1a9a5a14.pth" to /root/.torch_cache/inception_v3_google-1a9a5a14.pth
100%|| 108857766/108857766 [00:01<00:00, 89756667.45it/s]
100%|| 100/100 [00:03<00:00, 28.64it/s]
100%|| 100/100 [00:04<00:00, 25.88it/s]

1.0 % of the first 100 images in human_files have detected dogs.
98.0 % of the first 100 images in dog_files have detected dogs.

```

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 imbalanced, a random guess will provide a correct answer roughly 1 in 133 times, which corresponds to an accuracy of less than 1%.

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

1.1.7 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

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

```
In [11]: import os
        from torchvision import datasets

        ### TODO: Write data loaders for training, validation, and test sets
```

```

## Specify appropriate transforms, and batch_sizes
path = "/data/dog_images/"
batch_size = 32
num_workers = 0

train_transform = transforms.Compose([transforms.Resize(224),
                                    transforms.CenterCrop(224),
                                    transforms.RandomRotation(10),
                                    transforms.RandomVerticalFlip(p=0.20),
                                    transforms.RandomHorizontalFlip(),
                                    transforms.RandomGrayscale(p=0.20),
                                    transforms.ToTensor(),
                                    transforms.Normalize([0.485, 0.456, 0.406],
                                                       [0.229, 0.224, 0.225])
                                   ])

valid_test_transform = transforms.Compose([transforms.Resize(224),
                                         transforms.CenterCrop(224),
                                         transforms.ToTensor(),
                                         transforms.Normalize([0.485, 0.456, 0.406],
                                                            [0.229, 0.224, 0.225])
                                         ])

train_dataset = datasets.ImageFolder(path + 'train/', transform=train_transform)
valid_dataset = datasets.ImageFolder(path + 'valid/', transform=valid_test_transform)
test_dataset = datasets.ImageFolder(path + 'test/', transform=valid_test_transform)

train_dataloader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
valid_dataloader = torch.utils.data.DataLoader(valid_dataset, batch_size=batch_size, shuffle=True)
test_dataloader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size, shuffle=True)

loaders_scratch = {'train':train_dataloader, 'valid':valid_dataloader, 'test':test_dataloader}

```

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

Answer: - I resized the image and crop them in 224x224 px . The dog images are considerable larger than the human images, many of them seem to be of a high quality and centrally focused. it is therefore important to standardize the images. After some research, the dimensions of 224x224 show a model working well.

- To prevent overfitting, I decided to augment the dataset through randomised flips and rotations. Also, randomised grayscale was added in order to focus on the features of breed rather than color.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [12]: import torch.nn as nn
        import torch.nn.functional as F

        # define the CNN architecture
        class Net(nn.Module):
            ### TODO: choose an architecture, and complete the class
            def __init__(self):
                super(Net, self).__init__()
                ## Define layers of a CNN
                self.conv1 = nn.Conv2d(3, 16, 3)
                self.conv2 = nn.Conv2d(16, 32, 3)
                self.conv3 = nn.Conv2d(32, 64, 3)
                self.conv4 = nn.Conv2d(64, 128, 3)
                self.conv5 = nn.Conv2d(128, 256, 3)

                self.fc1 = nn.Linear(256 * 6 * 6, 133)

                self.max_pool = nn.MaxPool2d(2, 2, ceil_mode=True)
                self.dropout = nn.Dropout(0.2)

                self.conv_bn2 = nn.BatchNorm2d(16)
                self.conv_bn3 = nn.BatchNorm2d(32)
                self.conv_bn4 = nn.BatchNorm2d(64)
                self.conv_bn5 = nn.BatchNorm2d(128)
                self.conv_bn6 = nn.BatchNorm2d(256)

            def forward(self, x):
                ## Define forward behavior
                x = self.conv1(x)
                x = self.conv_bn2(x)
                x = F.relu(x)
                x = self.max_pool(x)

                x = self.conv2(x)
                x = self.conv_bn3(x)
                x = F.relu(x)
                x = self.max_pool(x)

                x = self.conv3(x)
                x = self.conv_bn4(x)
                x = F.relu(x)
                x = self.max_pool(x)

                x = self.conv4(x)
                x = self.conv_bn5(x)
```

```

        x = self.max_pool(x)
        x = F.relu(x)

        x = self.conv5(x)
        x = self.conv_bn6(x)
        x = self.max_pool(x)
        x = F.relu(x)

        x = x.view(-1, 256 * 6 * 6)

# fully connected layers here
x = self.dropout(x)
x = self.fc1(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:

I used 5 convolutional layers (for applying the convolution filters to the image to detect the features) each associated with a batchnormalisation, Relu function(which consists in replacing the negative numbers of the filtered images by zeros) and MaxPooling layer(for gradually reducing the size of the image while keeping only the most important features). Dropout is included to prevent overfitting and a linear fully connected layer is added at the end for producing inputs to our final activation 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 [13]: `import torch.optim as optim`

```

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

### TODO: select optimizer
optimizer_scratch = optim.SGD(model_scratch.parameters(), lr = 0.01)

```

1.1.10 (IMPLEMENTATION) Train and Validate the Model

Train and validate your model in the code cell below. Save the final model parameters at filepath 'model_scratch.pt'.

```
In [14]: 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_loss)
            # clear the accumulated gradients
            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()

            ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss)
            train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss))

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

```

        loss = criterion(output, target)
        valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_loss))

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

## TODO: save the model if validation loss has decreased
if valid_loss < valid_loss_min:
    torch.save(model.state_dict(), save_path)
    print('Validation loss decreased {:.6f} --> {:.6f}). Saving model ...'.format(
        valid_loss_min,
        valid_loss))
    valid_loss_min = valid_loss
else :
    pass

# return trained model
return model

# train the model
model_scratch = train(25, loaders_scratch, model_scratch, optimizer_scratch,
                      criterion_scratch, use_cuda, 'model_scratch.pt')

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


Epoch: 1      Training Loss: 4.726687      Validation Loss: 4.461962
Validation loss decreased (inf --> 4.461962). Saving model ...
Epoch: 2      Training Loss: 4.334765      Validation Loss: 4.252725
Validation loss decreased (4.461962 --> 4.252725). Saving model ...
Epoch: 3      Training Loss: 4.139474      Validation Loss: 4.120038
Validation loss decreased (4.252725 --> 4.120038). Saving model ...
Epoch: 4      Training Loss: 3.980102      Validation Loss: 4.151811
Epoch: 5      Training Loss: 3.819966      Validation Loss: 3.951158
Validation loss decreased (4.120038 --> 3.951158). Saving model ...
Epoch: 6      Training Loss: 3.714760      Validation Loss: 4.210033
Epoch: 7      Training Loss: 3.595324      Validation Loss: 3.930234
Validation loss decreased (3.951158 --> 3.930234). Saving model ...
Epoch: 8      Training Loss: 3.454952      Validation Loss: 3.745796
Validation loss decreased (3.930234 --> 3.745796). Saving model ...
Epoch: 9      Training Loss: 3.328849      Validation Loss: 3.985780
Epoch: 10     Training Loss: 3.226836      Validation Loss: 3.745386
Validation loss decreased (3.745796 --> 3.745386). Saving model ...

```

```

Epoch: 11      Training Loss: 3.083956      Validation Loss: 3.701863
Validation loss decreased (3.745386 --> 3.701863). Saving model ...
Epoch: 12      Training Loss: 3.001704      Validation Loss: 3.603222
Validation loss decreased (3.701863 --> 3.603222). Saving model ...
Epoch: 13      Training Loss: 2.912347      Validation Loss: 3.572169
Validation loss decreased (3.603222 --> 3.572169). Saving model ...
Epoch: 14      Training Loss: 2.829951      Validation Loss: 3.843917
Epoch: 15      Training Loss: 2.739177      Validation Loss: 3.768910
Epoch: 16      Training Loss: 2.657325      Validation Loss: 3.662688
Epoch: 17      Training Loss: 2.581887      Validation Loss: 3.338341
Validation loss decreased (3.572169 --> 3.338341). Saving model ...
Epoch: 18      Training Loss: 2.503821      Validation Loss: 3.671345
Epoch: 19      Training Loss: 2.408663      Validation Loss: 3.523644
Epoch: 20      Training Loss: 2.344889      Validation Loss: 3.472524
Epoch: 21      Training Loss: 2.291073      Validation Loss: 3.208655
Validation loss decreased (3.338341 --> 3.208655). Saving model ...
Epoch: 22      Training Loss: 2.231944      Validation Loss: 3.429370
Epoch: 23      Training Loss: 2.156713      Validation Loss: 3.245465
Epoch: 24      Training Loss: 2.071299      Validation Loss: 3.326765
Epoch: 25      Training Loss: 2.039184      Validation Loss: 3.435233

```

1.1.11 (IMPLEMENTATION) Test the Model

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

```
In [15]: def test(loaders, model, criterion, use_cuda):

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

    model.eval()
    for batch_idx, (data, target) in enumerate(loaders['test']):
        # move to GPU
        if use_cuda:
            data, target = data.cuda(), target.cuda()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the loss
        loss = criterion(output, target)
        # update average test loss
        test_loss = test_loss + ((1 / (batch_idx + 1)) * (loss.data - test_loss))
        # convert output probabilities to predicted class
        pred = output.data.max(1, keepdim=True)[1]
        # compare predictions to true label
```

```

    correct += np.sum(np.squeeze(pred.eq(target.data.view_as(pred))).cpu().numpy())
    total += data.size(0)

    print('Test Loss: {:.6f}\n'.format(test_loss))

    print('\nTest Accuracy: %d%% (%d/%d)' % (
        100. * correct / total, correct, total))

# call test function
test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)

```

Test Loss: 3.267607

Test Accuracy: 24% (201/836)

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

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

1.1.12 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

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

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

```
In [16]: ## TODO: Specify data loaders
train_transform = transforms.Compose([transforms.Resize(299),
                                    transforms.CenterCrop(299),
                                    transforms.RandomRotation(10),
                                    transforms.RandomGrayscale(p=0.25),
                                    transforms.RandomVerticalFlip(p=0.25),
                                    transforms.RandomHorizontalFlip(),
                                    transforms.ToTensor(),
                                    transforms.Normalize([0.485, 0.456, 0.406],
                                                       [0.229, 0.224, 0.225])
                                   ])

valid_test_transform = transforms.Compose([transforms.Resize(299),
                                         transforms.CenterCrop(299),
                                         transforms.ToTensor(),
                                         transforms.Normalize([0.485, 0.456, 0.406],
                                                            [0.229, 0.224, 0.225])
                                         ])
```

```

train_dataset = datasets.ImageFolder(path + 'train/', transform=train_transform)
valid_dataset = datasets.ImageFolder(path + 'valid/', transform=valid_test_transform)
test_dataset = datasets.ImageFolder(path + 'test/', transform=valid_test_transform)

train_dataloader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, sh
valid_dataloader = torch.utils.data.DataLoader(valid_dataset, batch_size=batch_size, sh
test_dataloader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size)

loaders_transfer = {'train':train_dataloader, 'valid':valid_dataloader, 'test':test_datalo

```

1.1.13 (IMPLEMENTATION) Model Architecture

Use transfer learning to create a CNN to classify dog breed. Use the code cell below, and save your initialized model as the variable `model_transfer`.

```

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

        ## TODO: Specify model architecture
        model_transfer = models.inception_v3(pretrained=True)

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

        model_transfer.aux_logits=False

        model_transfer.fc = nn.Linear(model_transfer.fc.in_features, 133)

        if use_cuda:
            model_transfer = model_transfer.cuda()

```

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

Answer:

I used inceptionV3 Architecture because it was trained on ImageNet for classification. It an efficient way to tackle computer vision problems. <https://arxiv.org/abs/1512.00567>

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 [18]: criterion_transfer =nn.CrossEntropyLoss()
        optimizer_transfer = optim.Adam(model_transfer.fc.parameters(), lr = 0.001)

```

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 [19]: # train the model
n_epochs = 20
model_transfer = train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer,
                      lr_scheduler)

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

Epoch: 1 Training Loss: 3.213670 Validation Loss: 1.286887
Validation loss decreased (inf --> 1.286887). Saving model ...
Epoch: 2 Training Loss: 1.759306 Validation Loss: 0.737108
Validation loss decreased (1.286887 --> 0.737108). Saving model ...
Epoch: 3 Training Loss: 1.447441 Validation Loss: 0.591407
Validation loss decreased (0.737108 --> 0.591407). Saving model ...
Epoch: 4 Training Loss: 1.324817 Validation Loss: 0.536811
Validation loss decreased (0.591407 --> 0.536811). Saving model ...
Epoch: 5 Training Loss: 1.267277 Validation Loss: 0.489368
Validation loss decreased (0.536811 --> 0.489368). Saving model ...
Epoch: 6 Training Loss: 1.210142 Validation Loss: 0.481174
Validation loss decreased (0.489368 --> 0.481174). Saving model ...
Epoch: 7 Training Loss: 1.157261 Validation Loss: 0.468939
Validation loss decreased (0.481174 --> 0.468939). Saving model ...
Epoch: 8 Training Loss: 1.142804 Validation Loss: 0.472176
Epoch: 9 Training Loss: 1.092301 Validation Loss: 0.471664
Epoch: 10 Training Loss: 1.055837 Validation Loss: 0.436260
Validation loss decreased (0.468939 --> 0.436260). Saving model ...
Epoch: 11 Training Loss: 1.079079 Validation Loss: 0.449854
Epoch: 12 Training Loss: 1.052173 Validation Loss: 0.438360
Epoch: 13 Training Loss: 1.037590 Validation Loss: 0.445557
Epoch: 14 Training Loss: 1.028342 Validation Loss: 0.458495
Epoch: 15 Training Loss: 1.016245 Validation Loss: 0.423110
Validation loss decreased (0.436260 --> 0.423110). Saving model ...
Epoch: 16 Training Loss: 1.000448 Validation Loss: 0.416614
Validation loss decreased (0.423110 --> 0.416614). Saving model ...
Epoch: 17 Training Loss: 0.958714 Validation Loss: 0.452857
Epoch: 18 Training Loss: 1.007780 Validation Loss: 0.425995
Epoch: 19 Training Loss: 1.001193 Validation Loss: 0.443575
Epoch: 20 Training Loss: 0.980438 Validation Loss: 0.428033

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

```
Test Loss: 0.461953
```

```
Test Accuracy: 86% (720/836)
```

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 [21]: ### TODO: Write a function that takes a path to an image as input
### and returns the dog breed that is predicted by the model.
path_subfolders = glob('/data/dog_images/*/')
data_transfer = {path.split('/')[-3]: datasets.ImageFolder(path, transform = valid_test_}
# list of class names by index, i.e. a name can be accessed like class_names[0]
class_names = [item[4:][replace("_", " ") for item in data_transfer['train'].classes]

def predict_breed_transfer(img_path):
    # load the image and return the predicted breed

    image = Image.open(img_path)
    image_tensor = valid_test_transform(image).unsqueeze(0)
    if use_cuda:
        image_tensor = image_tensor.cuda()
    output = model_transfer(image_tensor)
    return class_names[output.data.argmax().cpu().numpy()]
```

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.

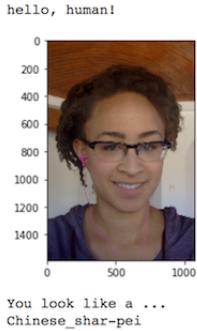
You are welcome to write your own functions for detecting humans and dogs in images, but feel free to use the `face_detector` and `human_detector` functions developed above. You are **required** to use your CNN from Step 4 to predict dog breed.

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

```
def run_app(img_path):
```



Sample Human Output

```
## handle cases for a human face, dog, and neither
detect_dog = dog_detector_inceptionv3(img_path)
detect_human = face_detector(img_path)
if detect_dog:
    breed = predict_breed_transfer(img_path)
    message = ["Dog detected!", "It's a cute : ", breed, '\n']

elif detect_human:
    breed = predict_breed_transfer(img_path)
    message = ['Human detected!', ' but you look like a ', breed, '\n']
else:
    message = ["No dog or human detected!"]

print(message[0])
display(Image.open(img_path))
print(''.join(message[1:]))
```

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)

The output is about as well as I expected. There are a couple of improvements possible for the current algorithm.

1. Adding dropout probability and Fine tuning may improve the performance of the model.

2. Choosing another torchvision.model.
3. Trying a larger dataset or using more image augmentation

```
In [28]: ## 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.
files_path = glob('test_images/*')
## suggested code, below
for file in files_path:
    run_app(file)
    print('-'*100)
```

Dog detected!



It's a cute : Bulldog

Dog detected!



It's a cute : German pinscher

No dog or human detected!



Human detected!



but you look like a Bullmastiff!

Human detected!



but you look like a Alaskan malamute!

Human detected!



but you look like a Affenpinscher!

Dog detected!



It's a cute : Pomeranian

In []: