

dog_app

February 19, 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/**/*.jpg"))
        dog_files = np.array(glob("/data/dog_images/**/*.jpg"))

        # 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)

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

```

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

```

```

## Test the performance of the face_detector algorithm
## on the images in human_files_short and dog_files_short.

```

```

human_face_detected_from_human = sum([1 for file in tqdm(human_files_short) if face_detector(file) > 0])
print(f"Percentage of human face detected from human files: {(human_face_detected_from_human/len(human_files_short))*100}%")

```

```

human_face_detected_from_dog = sum([1 for file in tqdm(dog_files_short) if face_detector(file) > 0])
print(f"Percentage of human face detected from dog files: {(human_face_detected_from_dog/len(dog_files_short))*100}%")

```

```
HBox(children=(IntProgress(value=0), HTML(value='')))
```

Percentage of human face detected from human files: 98.0%

```
HBox(children=(IntProgress(value=0), HTML(value='')))
```

Percentage of human face detected from dog files: 17.0%

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 [00:04<00:00, 125152557.61it/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
'''

## Load and pre-process an image from the given img_path
image = Image.open(img_path)
data_transforms = transforms.Compose([
    transforms.RandomRotation(45),
    transforms.RandomResizedCrop(224),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(mean = [0.485, 0.456, 0.406],
                          std = [0.229, 0.224, 0.225])
])
image_tensor = data_transforms(image)
image_tensor.unsqueeze_(0)

if use_cuda:
    image_tensor = image_tensor.cuda()

## Return the *index* of the predicted class for that image
class_indexes = VGG16(image_tensor)

if use_cuda:
    class_indexes = class_indexes.cpu()

predicted_class_index = class_indexes.data.numpy().argmax()
return 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):

```

```
class_index = VGG16_predict(img_path)
return class_index >= 151 and class_index <=268
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

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

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

Answer:

```
In [9]: from tqdm import tqdm_notebook as tqdm
```

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

```
### Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
dog_detected_from_human = sum([1 for file in tqdm(human_files_short) if dog_detector(file)])
print(f"Percentage of dog detected from human files: {(dog_detected_from_human/100)*100}%")

dog_detected_from_dog = sum([1 for file in tqdm(dog_files_short) if dog_detector(file)])
print(f"Percentage of dog detected from dog files: {(dog_detected_from_dog/100)*100}%")
```

```
HBox(children=(IntProgress(value=0), HTML(value='')))
```

Percentage of dog detected from human files: 7.000000000000001%

```
HBox(children=(IntProgress(value=0), HTML(value='')))
```

Percentage of dog detected from dog files: 98.0%

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 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

         # To solve problem of PIL reading in blocks of the file
         # and that it expects that the blocks are going to be of a certain size
         # Source: https://stackoverflow.com/a/23575424/1741325
```



```

from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

data_dir = '/data/dog_images'

train_dir = os.path.join(data_dir, 'train')
valid_dir = os.path.join(data_dir, 'valid')
test_dir = os.path.join(data_dir, 'test')

dirs = {'train': train_dir,
        'valid': valid_dir,
        'test': test_dir}

size = 224
data_transforms = {
    'train': transforms.Compose([
        transforms.RandomRotation(45),
        transforms.RandomResizedCrop(size),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406],
                             [0.229, 0.224, 0.225])
    ]),
    'valid': transforms.Compose([
        transforms.Resize(size + 32),
        transforms.CenterCrop(size),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406],
                             [0.229, 0.224, 0.225])
    ]),
    'test': transforms.Compose([
        transforms.Resize(size + 32),
        transforms.CenterCrop(size),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406],
                             [0.229, 0.224, 0.225])
    ])
}

num_workers = 0
batch_size = 20

image_datasets = {x: datasets.ImageFolder(dirs[x], transform=data_transforms[x]) for x in dirs}
loaders_scratch = {x: torch.utils.data.DataLoader(image_datasets[x], batch_size=batch_size, num_workers=num_workers)}

```

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:

- My code resize the images by cropping. Input data is resized to 224x224 pixels as required by the networks.
- I decide to augment the dataset through random rotating, scaling, cropping and flipping.

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):
    def __init__(self):
        super(Net, self).__init__()
        # convolutional layer (sees 32x32x3 image tensor)
        self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
        # convolutional layer (sees 16x16x16 tensor)
        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
        # convolutional layer (sees 8x8x32 tensor)
        self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
        # convolutional layer (sees 4x4x64 tensor)
        self.conv4 = nn.Conv2d(64, 128, 3, padding=1)
        # max pooling layer
        self.pool = nn.MaxPool2d(2, 2)
        # linear layer (128 * 14 * 14 -> 500)
        self.fc1 = nn.Linear(128 * 14 * 14, 500)
        # linear layer (500 -> 133)
        self.fc2 = nn.Linear(500, 133)
        # dropout layer (p=0.25)
        self.dropout = nn.Dropout(0.25)

    def forward(self, x):
        # add sequence of convolutional and max pooling layers
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = self.pool(F.relu(self.conv4(x)))
        # flatten image input
        x = x.view(-1, 128 * 14 * 14)
        # add dropout layer
        x = self.dropout(x)
        # add 1st hidden layer, with relu activation function
        x = F.relu(self.fc1(x))
        # add dropout layer
```

```

        x = self.dropout(x)
        # add 2nd hidden layer, with relu activation function
        x = self.fc2(x)
        return x

# 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:

- Network initialization steps:
 - Add convolutional layer named conv1, which load 224x224x3 tensor and output a stack of 16 feature maps
 - Add convolutional layer named conv2 to increase the depth of conv1 output to 16
 - Add convolutional layer named conv3 to double the depth of conv2 output to 32
 - Add convolutional layer named conv4 to double the depth of conv3 output to 64
 - Downsample the size by 2 using the max pooling layer
 - Add linear fully connected layer fc1, go from $(224 / 2 / 2 / 2 / 2) ** 2$ to 500 features
 - Add second fully connected layer fc2, go from 500 to 133 features (the number of dog breeds needs to identify)
 - Add a dropout layer using 0.25 probability to prevent overfitting
- Forward pass steps:
 - Apply max pooling and the ReLu activation function to conv1
 - Apply max pooling and the ReLu activation function to conv2
 - Apply max pooling and the ReLu activation function to conv3
 - Apply max pooling and the ReLu activation function to conv4
 - Flatten the image input
 - Add a dropout layer
 - Add first hidden layer with applying relu activation function
 - Add dropout layer
 - Return the second hidden layer with applying relu 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
```

```
### select loss function
```

```

# CrossEntropyLoss is useful for classification tasks
criterion_scratch = nn.CrossEntropyLoss()

### 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]: 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
            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()
            ## record the average training 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))

    train_loss = train_loss/len(loaders['train'].dataset)
    valid_loss = valid_loss/len(loaders['valid'].dataset)

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

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

    # return trained model
    return model

# train the model
model_scratch = train(30, loaders_scratch, model_scratch, optimizer_scratch,
                      criterion_scratch, use_cuda, 'model_scratch.pt')

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

```

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: %2d%% (%2d/%2d)' % (
        100. * correct / total, correct, total))

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

```

Test Loss: 3.854630

Test Accuracy: 11% (99/836)

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]: import os
         import torch
         import torchvision.transforms as transforms
         from torchvision import datasets

         # To solve problem of PIL reading in blocks of the file
         # and that it expects that the blocks are going to be of a certain size

```

```

# Source: https://stackoverflow.com/a/23575424/1741325
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

## Specify data loaders
data_dir = '/data/dog_images'

train_dir = os.path.join(data_dir, 'train')
valid_dir = os.path.join(data_dir, 'valid')
test_dir = os.path.join(data_dir, 'test')

dirs = {'train': train_dir,
        'valid': valid_dir,
        'test': test_dir}

size = 224
data_transforms = {
    'train': transforms.Compose([
        transforms.RandomRotation(45),
        transforms.RandomResizedCrop(size),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406],
                             [0.229, 0.224, 0.225])
    ]),
    'valid': transforms.Compose([
        transforms.Resize(size + 32),
        transforms.CenterCrop(size),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406],
                             [0.229, 0.224, 0.225])
    ]),
    'test': transforms.Compose([
        transforms.Resize(size + 32),
        transforms.CenterCrop(size),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406],
                             [0.229, 0.224, 0.225])
    ])
}

num_workers = 0
batch_size = 20

image_datasets = {x: datasets.ImageFolder(dirs[x], transform=data_transforms[x]) for x
loaders_transfer = {x: torch.utils.data.DataLoader(image_datasets[x], batch_size=batch_

```

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

## Specify model architecture

# Load the pretrained model from pytorch
model_transfer = models.vgg16(pretrained=True)

# Freeze training for all "features" layers
for param in model_transfer.features.parameters():
    param.requires_grad = False

n_inputs = model_transfer.classifier[6].in_features

## add a last linear layer that maps n_inputs -> 133 dog breeds
last_layer = nn.Linear(n_inputs, 133)
model_transfer.classifier[6] = last_layer

for param in model_transfer.classifier[6].parameters():
    param.requires_grad = True

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

To define a model for training I follow these steps: 1. Load in a pre-trained VGG16 model 2. "Freeze" all the parameters, so the net acts as a fixed feature extractor 3. Remove the last layer 4. Replace the last layer with a linear classifier of our own

VGGNet is great because it's simple and has great performance, coming in second in the ImageNet competition.

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]: import torch.optim as optim

criterion_transfer = nn.CrossEntropyLoss()
optimizer_transfer = optim.SGD(model_transfer.classifier[6].parameters(), lr=0.01)
```


1.1.15 (IMPLEMENTATION) Train and Validate the Model

Train and validate your model in the code cell below. [Save the final model parameters](#) at filepath 'model_transfer.pt'.

```
In [19]: import numpy as np
         from workspace_utils import active_session

         # train the model
         n_epochs = 20
         model_transfer = train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer,

         # load the model that got the best validation accuracy (uncomment the line below)
         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 [20]: test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

Test Loss: 0.454721

Test Accuracy: 86% (722/836)

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

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

         def predict_breed_transfer(img_path):
             ## Load and pre-process an image from the given img_path
             image = Image.open(img_path)
             data_transforms = transforms.Compose([
                 transforms.RandomRotation(45),
                 transforms.RandomResizedCrop(224),
                 transforms.RandomHorizontalFlip(),
                 transforms.ToTensor(),
                 transforms.Normalize(mean = [0.485, 0.456, 0.406],
                                     std = [0.229, 0.224, 0.225])
```



Sample Human Output

```

])
image_tensor = data_transforms(image)
image_tensor.unsqueeze_(0)

if use_cuda:
    image_tensor = image_tensor.cuda()

## Return the *index* of the predicted class for that image
model_transfer.eval()
output = model_transfer(image_tensor)
_, prediction = torch.max(output.data, 1)
breed = class_names[prediction]

return breed

```

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

def run_app(img_path):
    ## handle cases for a human face, dog, and neither
    dog_breed_detected = dog_detector(img_path)

```

```

human_face_detected = face_detector(img_path)
print(f"-----")
plt.figure(figsize = (3,3))
image = Image.open(file)
imshow(image)
plt.show()
detection_type = ''
message = 'You look like a..'
if dog_breed_detected or human_face_detected:
    dog_breed = predict_breed_transfer(img_path)
    if dog_breed_detected:
        detection_type = 'dog'
    elif human_face_detected:
        detection_type = 'human'
    print(f"Hello, {detection_type}")
    print(f"{message}")
    print(f"{dog_breed}")
else:
    print(f"Error, neither dog or human was detected!")

```

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 was worse than I expected. The prediction is changing and sometimes it's not right. It cannot recognize the faces of people or dogs under certain conditions.

There is possible points of improvement:

- Use different transformation for training images.
- Use different transfer model.
- Add more layers to model.

```

In [68]: import numpy as np
         from glob import glob
         from pylab import *

```

```

## Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.

```

```

human_files = np.array(glob("test_images/human/*"))
dog_files = np.array(glob("test_images/dog/*"))

## suggested code, below
for file in np.hstack((human_files[:3], dog_files[:3])):
    run_app(file)

```



Error, neither dog or human was detected!



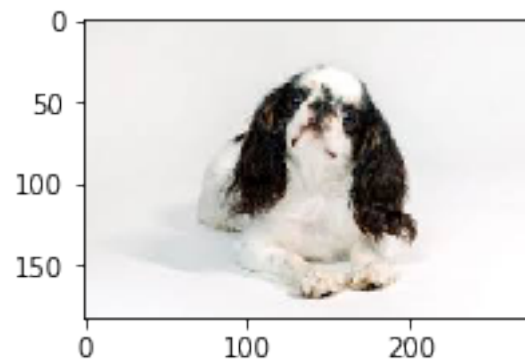
```

Hello, human
You look like a..
English springer spaniel

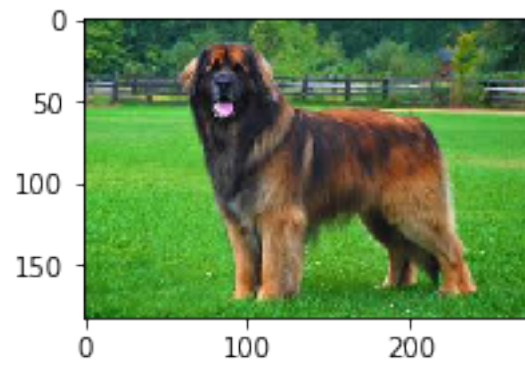
```



Hello, human
You look like a..
Bernese mountain dog



Hello, dog
You look like a..
English toy spaniel



Hello, dog
You look like a..
Leonberger



Hello, dog
You look like a..
Doberman pinscher