

dog_app

March 24, 2021

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

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

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

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

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

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

Step 0: Import Datasets

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

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

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

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

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

```
[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.

```
[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))

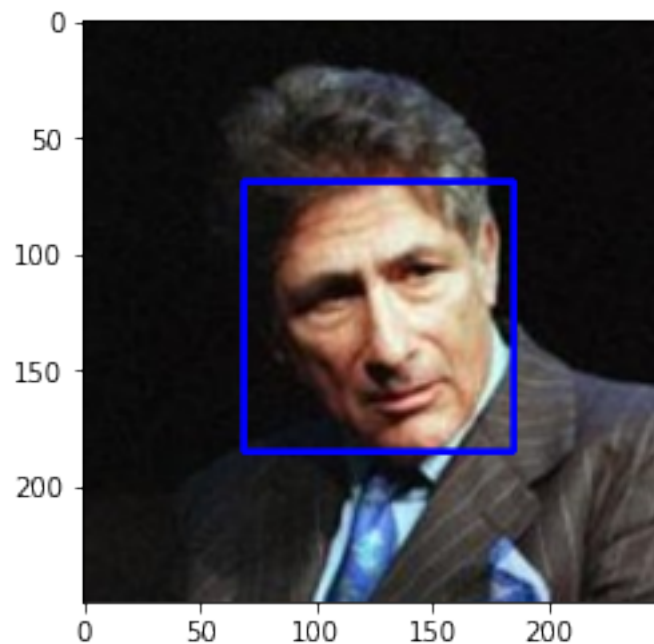
# get bounding box for each detected face
for (x,y,w,h) in faces:
    # add bounding box to color image
    cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)

# 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.

```
[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)

```
[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.
def face_detector_performance(file_paths, file_type):
    file_cnt = len(file_paths)
    face_detection_cnt=0
    for img_path in file_paths:
        if face_detector(img_path):
            face_detection_cnt+=1
    print(f'{face_detection_cnt} faces detected on {file_cnt} {file_type} \u2192files')

face_detector_performance(human_files_short, 'human')
face_detector_performance(dog_files_short, 'dog')
```

99 faces detected on 100 human files
7 faces detected on 100 dog files

We suggest the face detector from OpenCV as a potential way to detect human images in your algorithm, but you are free to explore other approaches, especially approaches that make use of deep learning :). Please use the code cell below to design and test your own face detection algorithm. If you decide to pursue this *optional* task, report performance on `human_files_short` and `dog_files_short`.

```
[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.

```
[6]: import torch  
import torchvision.models as models  
  
# define VGG16 model  
VGG16 = models.vgg16(pretrained=True)  
  
# check if CUDA is available  
use_cuda = torch.cuda.is_available()  
  
# move model to GPU if CUDA is available  
if use_cuda:  
    VGG16 = VGG16.cuda()
```

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

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

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

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

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

# Set PIL to be tolerant of image files that are truncated.
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

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
    global use_cuda
    img_transforms = transforms.Compose([transforms.CenterCrop(224),
                                         transforms.ToTensor()])
    img = img_transforms(Image.open(img_path))[None, :, :]
    if use_cuda:
        img = img.cuda()
    VGG16.eval()
    output = VGG16(img)
    _, predictions = torch.max(output, 1)
    if use_cuda:
        predictions = predictions.cpu()
    return predictions.numpy()[0] # 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).

```
[8]: ### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    ## TODO: Complete the function.
    predicted_class = VGG16_predict(img_path)
    return predicted_class>=151 and predicted_class<=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:

```
[9]: ### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
def dog_detector_performance(file_paths, file_type):
    file_cnt = len(file_paths)
    dog_detection_cnt=0
    for img_path in file_paths:
        if dog_detector(img_path):
            dog_detection_cnt+=1
    print(f'{dog_detection_cnt} dogs detected on {file_cnt} {file_type} files')

dog_detector_performance(human_files_short, 'human')
dog_detector_performance(dog_files_short, 'dog')
```

```
1 dogs detected on 100 human files
79 dogs detected on 100 dog files
```

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

```
[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](#)!

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

      ### TODO: Write data loaders for training, validation, and test sets
      ## Specify appropriate transforms, and batch_sizes
      batch_size = 64
```



```

num_workers=0

data_dir = '/data/dog_images/'
train_dir = os.path.join(data_dir, 'train/')
val_dir = os.path.join(data_dir, 'valid/')
test_dir = os.path.join(data_dir, 'test/')

train_transforms = transforms.Compose([transforms.RandomRotation(30),
                                       transforms.RandomResizedCrop(224),
                                       transforms.RandomHorizontalFlip(),
                                       transforms.ToTensor(),
                                       transforms.Normalize([0.485, 0.456, 0.
↪406],
                                                         [0.229, 0.224, 0.
↪225]]))

val_transforms = transforms.Compose([transforms.Resize(255),
                                     transforms.CenterCrop(224),
                                     transforms.ToTensor(),
                                     transforms.Normalize([0.485, 0.456, 0.406],
                                                         [0.229, 0.224, 0.
↪225]]))

test_transforms = transforms.Compose([transforms.Resize(255),
                                       transforms.CenterCrop(224),
                                       transforms.ToTensor(),
                                       transforms.Normalize([0.485, 0.456, 0.
↪406],
                                                         [0.229, 0.224, 0.
↪225]]))

train_data = datasets.ImageFolder(train_dir, transform=train_transforms)
val_data = datasets.ImageFolder(val_dir, transform=val_transforms)
test_data = datasets.ImageFolder(test_dir, transform=test_transforms)

print('Num training images: ', len(train_data))
print('Num validation images: ', len(val_data))
print('Num test images: ', len(test_data))

train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,
                                           num_workers=num_workers,
↪shuffle=True)
val_loader = torch.utils.data.DataLoader(val_data, batch_size=batch_size,
                                          num_workers=num_workers, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size,

```

```

num_workers=num_workers,
shuffle=False)

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

```

Num training images: 6680

Num validation images: 835

Num test images: 836

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: Based on the "Detect Dogs" and "Transfer Learning" tasks in this project, I used the VGG-16 input size of 224x224 as a baseline. I added some crop for all transformations, as well as some image augmentations for the train transformations. The used image augmentation transformations should improve the quality of the model and are based on the "Transfer Learning" examples of this course (Getting started with PyTorch)

1.1.8 (IMPLEMENTATION) Model Architecture

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

```

[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.layer1 = nn.Sequential(
            nn.Conv2d(3, 16, 3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2)
        )
        self.layer2 = nn.Sequential(
            nn.Conv2d(16, 32, 3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2)
        )
        self.layer3 = nn.Sequential(

```

```

        nn.Conv2d(32, 64, 3, padding=1),
        nn.ReLU(),
        nn.MaxPool2d(2, 2)
    )
    self.fc1 = nn.Sequential(
        nn.Linear(64 * 28 * 28, 512),
        nn.ReLU(),
    )
    self.fc2 = nn.Sequential(
        nn.Linear(512, 133),
        nn.LogSoftmax(dim=1)
    )
    self.dropout = nn.Dropout(p=0.4)

    def forward(self, x):
        ## Define forward behavior
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.dropout(x)
        x = x.view(-1, 64 * 28 * 28)
        x = self.fc1(x)
        x = self.dropout(x)
        x = self.fc2(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: A research on the Internet, about this problem domain, didn't deliver any useful information. That's why I decided to take the CIFAR10 example (provided in the CNN section of this course) as a base. This uses the standard "Image Detection" pattern (Conv-Pool, Conv-Pool, ... Flatten, Dense, Dense (Softmax). After running a "test" training of the model, I added another hidden layer and re-run the training process. Looking at the results, I decided to tweak the hyperparameters (e.g. dropout layer, num of neurons, etc.) to my needs.

```

[13]: from torchsummary import summary
      summary(model_scratch, (3, 224, 224))

```

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 16, 224, 224]	448
ReLU-2	[-1, 16, 224, 224]	0
MaxPool2d-3	[-1, 16, 112, 112]	0
Conv2d-4	[-1, 32, 112, 112]	4,640
ReLU-5	[-1, 32, 112, 112]	0
MaxPool2d-6	[-1, 32, 56, 56]	0
Conv2d-7	[-1, 64, 56, 56]	18,496
ReLU-8	[-1, 64, 56, 56]	0
MaxPool2d-9	[-1, 64, 28, 28]	0
Dropout-10	[-1, 64, 28, 28]	0
Linear-11	[-1, 512]	25,690,624
ReLU-12	[-1, 512]	0
Dropout-13	[-1, 512]	0
Linear-14	[-1, 133]	68,229
LogSoftmax-15	[-1, 133]	0
Total params: 25,782,437		
Trainable params: 25,782,437		
Non-trainable params: 0		
Input size (MB): 0.57		
Forward/backward pass size (MB): 24.51		
Params size (MB): 98.35		
Estimated Total Size (MB): 123.44		

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.

```
[14]: import torch.optim as optim

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

      ### TODO: select optimizer
      optimizer_scratch = optim.Adam(params=model_scratch.parameters(), lr=0.001)
```

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'`.

```

[15]: # the following import is required for training to be robust to truncated images
import time
import torch.cuda
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
    if use_cuda:
        print("CUDA Device:", torch.cuda.get_device_name(0))
        print("Memory Allocated:", round(torch.cuda.memory_allocated(0)/
↪1024**3,1), "GB")
        scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=25, gamma=0.8)

    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))
            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()
            train_loss += loss.item()*data.size(0)

        #####
        # validate the model #
        #####
        model.eval()
        for batch_idx, (data, target) in enumerate(loaders['valid']):
            # move to GPU
            if use_cuda:

```

```

        data, target = data.cuda(), target.cuda()
        ## update the average validation loss
        output = model(data)
        loss = criterion(output, target)
        valid_loss += loss.item()*data.size(0)

        # print training/validation statistics
        print('[{}] Epoch: {} \tLR: {:.6f} \tTraining Loss: {:.6f} \tValidation_
↪Loss: {:.6f}'.format(
            time.strftime('%H:%M:%S %Z'),
            epoch,
            scheduler.get_lr()[0],
            train_loss,
            valid_loss
        ))
        scheduler.step()

        ## TODO: 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(100, 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'))

```

CUDA Device: Tesla V100-PCIE-16GB

Memory Allocated: 0.6 GB

[17:03:16 UTC] Epoch: 1 LR: 0.001000 Training Loss: 32607.044933

Validation Loss: 3957.573839

Validation loss decreased (inf --> 3957.573839). Saving model ...

[17:04:41 UTC] Epoch: 2 LR: 0.001000 Training Loss: 31537.199303

Validation Loss: 3814.445919

Validation loss decreased (3957.573839 --> 3814.445919). Saving model ...

[17:06:06 UTC] Epoch: 3 LR: 0.001000 Training Loss: 30713.704098

Validation Loss: 3713.925223

Validation loss decreased (3814.445919 --> 3713.925223). Saving model ...
[17:07:30 UTC] Epoch: 4 LR: 0.001000 Training Loss: 30211.198147
Validation Loss: 3681.189517
Validation loss decreased (3713.925223 --> 3681.189517). Saving model ...
[17:08:53 UTC] Epoch: 5 LR: 0.001000 Training Loss: 29731.583904
Validation Loss: 3576.255570
Validation loss decreased (3681.189517 --> 3576.255570). Saving model ...
[17:10:17 UTC] Epoch: 6 LR: 0.001000 Training Loss: 29387.712658
Validation Loss: 3512.044501
Validation loss decreased (3576.255570 --> 3512.044501). Saving model ...
[17:11:41 UTC] Epoch: 7 LR: 0.001000 Training Loss: 29094.651173
Validation Loss: 3493.312143
Validation loss decreased (3512.044501 --> 3493.312143). Saving model ...
[17:13:04 UTC] Epoch: 8 LR: 0.001000 Training Loss: 28676.177555
Validation Loss: 3426.358253
Validation loss decreased (3493.312143 --> 3426.358253). Saving model ...
[17:14:28 UTC] Epoch: 9 LR: 0.001000 Training Loss: 28376.763124
Validation Loss: 3392.587538
Validation loss decreased (3426.358253 --> 3392.587538). Saving model ...
[17:15:52 UTC] Epoch: 10 LR: 0.001000 Training Loss: 28157.645329
Validation Loss: 3345.360082
Validation loss decreased (3392.587538 --> 3345.360082). Saving model ...
[17:17:17 UTC] Epoch: 11 LR: 0.001000 Training Loss: 27773.021027
Validation Loss: 3370.885903
[17:18:40 UTC] Epoch: 12 LR: 0.001000 Training Loss: 27570.004763
Validation Loss: 3257.591638
Validation loss decreased (3345.360082 --> 3257.591638). Saving model ...
[17:20:06 UTC] Epoch: 13 LR: 0.001000 Training Loss: 27419.248432
Validation Loss: 3240.312006
Validation loss decreased (3257.591638 --> 3240.312006). Saving model ...
[17:21:33 UTC] Epoch: 14 LR: 0.001000 Training Loss: 27246.686251
Validation Loss: 3229.112357
Validation loss decreased (3240.312006 --> 3229.112357). Saving model ...
[17:22:56 UTC] Epoch: 15 LR: 0.001000 Training Loss: 26797.270283
Validation Loss: 3181.470847
Validation loss decreased (3229.112357 --> 3181.470847). Saving model ...
[17:24:20 UTC] Epoch: 16 LR: 0.001000 Training Loss: 26660.990223
Validation Loss: 3188.712860
[17:25:42 UTC] Epoch: 17 LR: 0.001000 Training Loss: 26529.126408
Validation Loss: 3166.789121
Validation loss decreased (3181.470847 --> 3166.789121). Saving model ...
[17:27:06 UTC] Epoch: 18 LR: 0.001000 Training Loss: 26227.042545
Validation Loss: 3105.232056
Validation loss decreased (3166.789121 --> 3105.232056). Saving model ...
[17:28:28 UTC] Epoch: 19 LR: 0.001000 Training Loss: 26127.484974
Validation Loss: 3136.826362
[17:29:51 UTC] Epoch: 20 LR: 0.001000 Training Loss: 25901.967623
Validation Loss: 3076.867345

Validation loss decreased (3105.232056 --> 3076.867345). Saving model ...
[17:31:14 UTC] Epoch: 21 LR: 0.001000 Training Loss: 25696.886181
Validation Loss: 3058.728885
Validation loss decreased (3076.867345 --> 3058.728885). Saving model ...
[17:32:37 UTC] Epoch: 22 LR: 0.001000 Training Loss: 25681.408653
Validation Loss: 3049.662723
Validation loss decreased (3058.728885 --> 3049.662723). Saving model ...
[17:34:00 UTC] Epoch: 23 LR: 0.001000 Training Loss: 25259.856474
Validation Loss: 3040.417213
Validation loss decreased (3049.662723 --> 3040.417213). Saving model ...
[17:35:23 UTC] Epoch: 24 LR: 0.001000 Training Loss: 25149.101212
Validation Loss: 3083.362658
[17:36:45 UTC] Epoch: 25 LR: 0.001000 Training Loss: 25104.599684
Validation Loss: 2959.345135
Validation loss decreased (3040.417213 --> 2959.345135). Saving model ...
[17:38:09 UTC] Epoch: 26 LR: 0.000900 Training Loss: 24791.108759
Validation Loss: 3004.460589
[17:39:31 UTC] Epoch: 27 LR: 0.000900 Training Loss: 24640.773575
Validation Loss: 2928.616585
Validation loss decreased (2959.345135 --> 2928.616585). Saving model ...
[17:40:57 UTC] Epoch: 28 LR: 0.000900 Training Loss: 24582.252817
Validation Loss: 2947.239994
[17:42:21 UTC] Epoch: 29 LR: 0.000900 Training Loss: 24598.803339
Validation Loss: 2888.971970
Validation loss decreased (2928.616585 --> 2888.971970). Saving model ...
[17:43:46 UTC] Epoch: 30 LR: 0.000900 Training Loss: 24415.452774
Validation Loss: 2926.236439
[17:45:08 UTC] Epoch: 31 LR: 0.000900 Training Loss: 24296.468885
Validation Loss: 2927.588938
[17:46:30 UTC] Epoch: 32 LR: 0.000900 Training Loss: 24368.390015
Validation Loss: 2884.620264
Validation loss decreased (2888.971970 --> 2884.620264). Saving model ...
[17:47:53 UTC] Epoch: 33 LR: 0.000900 Training Loss: 24216.139824
Validation Loss: 2858.248840
Validation loss decreased (2884.620264 --> 2858.248840). Saving model ...
[17:49:17 UTC] Epoch: 34 LR: 0.000900 Training Loss: 23914.749577
Validation Loss: 2875.658385
[17:50:39 UTC] Epoch: 35 LR: 0.000900 Training Loss: 23830.955925
Validation Loss: 2849.318174
Validation loss decreased (2858.248840 --> 2849.318174). Saving model ...
[17:52:08 UTC] Epoch: 36 LR: 0.000900 Training Loss: 23867.591600
Validation Loss: 2853.705539
[17:53:31 UTC] Epoch: 37 LR: 0.000900 Training Loss: 23725.542704
Validation Loss: 2889.762198
[17:54:56 UTC] Epoch: 38 LR: 0.000900 Training Loss: 23589.577509
Validation Loss: 2867.210135
[17:56:22 UTC] Epoch: 39 LR: 0.000900 Training Loss: 23568.120802
Validation Loss: 2839.600791

Validation loss decreased (2849.318174 --> 2839.600791). Saving model ...
[17:57:46 UTC] Epoch: 40 LR: 0.000900 Training Loss: 23277.352148
Validation Loss: 2841.020954
[17:59:09 UTC] Epoch: 41 LR: 0.000900 Training Loss: 23389.242073
Validation Loss: 2869.574959
[18:00:32 UTC] Epoch: 42 LR: 0.000900 Training Loss: 23231.137106
Validation Loss: 2828.072725
Validation loss decreased (2839.600791 --> 2828.072725). Saving model ...
[18:01:55 UTC] Epoch: 43 LR: 0.000900 Training Loss: 23176.265123
Validation Loss: 2806.788370
Validation loss decreased (2828.072725 --> 2806.788370). Saving model ...
[18:03:19 UTC] Epoch: 44 LR: 0.000900 Training Loss: 23115.738432
Validation Loss: 2843.376087
[18:04:43 UTC] Epoch: 45 LR: 0.000900 Training Loss: 23171.045151
Validation Loss: 2780.578101
Validation loss decreased (2806.788370 --> 2780.578101). Saving model ...
[18:06:07 UTC] Epoch: 46 LR: 0.000900 Training Loss: 22973.082796
Validation Loss: 2790.419773
[18:07:30 UTC] Epoch: 47 LR: 0.000900 Training Loss: 22795.824669
Validation Loss: 2781.658633
[18:08:53 UTC] Epoch: 48 LR: 0.000900 Training Loss: 22708.290642
Validation Loss: 2804.786034
[18:10:16 UTC] Epoch: 49 LR: 0.000900 Training Loss: 22824.173035
Validation Loss: 2802.604506
[18:11:39 UTC] Epoch: 50 LR: 0.000900 Training Loss: 22748.091881
Validation Loss: 2728.935154
Validation loss decreased (2780.578101 --> 2728.935154). Saving model ...
[18:13:04 UTC] Epoch: 51 LR: 0.000810 Training Loss: 22539.182886
Validation Loss: 2740.757390
[18:14:27 UTC] Epoch: 52 LR: 0.000810 Training Loss: 22418.065231
Validation Loss: 2729.175307
[18:15:50 UTC] Epoch: 53 LR: 0.000810 Training Loss: 22371.935240
Validation Loss: 2746.660300
[18:17:14 UTC] Epoch: 54 LR: 0.000810 Training Loss: 22408.017366
Validation Loss: 2733.898347
[18:18:37 UTC] Epoch: 55 LR: 0.000810 Training Loss: 22315.596827
Validation Loss: 2729.108277
[18:20:00 UTC] Epoch: 56 LR: 0.000810 Training Loss: 22301.286940
Validation Loss: 2701.917021
Validation loss decreased (2728.935154 --> 2701.917021). Saving model ...
[18:21:23 UTC] Epoch: 57 LR: 0.000810 Training Loss: 22175.119226
Validation Loss: 2768.139924
[18:22:47 UTC] Epoch: 58 LR: 0.000810 Training Loss: 21817.300180
Validation Loss: 2683.369607
Validation loss decreased (2701.917021 --> 2683.369607). Saving model ...
[18:24:10 UTC] Epoch: 59 LR: 0.000810 Training Loss: 22048.517891
Validation Loss: 2707.904028
[18:25:33 UTC] Epoch: 60 LR: 0.000810 Training Loss: 22058.173355

Validation Loss: 2662.575852
 Validation loss decreased (2683.369607 --> 2662.575852). Saving model ...
 [18:26:59 UTC] Epoch: 61 LR: 0.000810 Training Loss: 21672.149782
 Validation Loss: 2674.969828
 [18:28:23 UTC] Epoch: 62 LR: 0.000810 Training Loss: 21965.111137
 Validation Loss: 2654.551626
 Validation loss decreased (2662.575852 --> 2654.551626). Saving model ...
 [18:29:47 UTC] Epoch: 63 LR: 0.000810 Training Loss: 21717.858765
 Validation Loss: 2734.038833
 [18:31:10 UTC] Epoch: 64 LR: 0.000810 Training Loss: 21821.021933
 Validation Loss: 2654.529372
 Validation loss decreased (2654.551626 --> 2654.529372). Saving model ...
 [18:32:34 UTC] Epoch: 65 LR: 0.000810 Training Loss: 21600.449759
 Validation Loss: 2678.224232
 [18:33:57 UTC] Epoch: 66 LR: 0.000810 Training Loss: 21720.920355
 Validation Loss: 2710.011400
 [18:35:22 UTC] Epoch: 67 LR: 0.000810 Training Loss: 21714.251944
 Validation Loss: 2645.487927
 Validation loss decreased (2654.529372 --> 2645.487927). Saving model ...
 [18:36:47 UTC] Epoch: 68 LR: 0.000810 Training Loss: 21445.336029
 Validation Loss: 2650.221228
 [18:38:11 UTC] Epoch: 69 LR: 0.000810 Training Loss: 21238.579561
 Validation Loss: 2638.913090
 Validation loss decreased (2645.487927 --> 2638.913090). Saving model ...
 [18:39:37 UTC] Epoch: 70 LR: 0.000810 Training Loss: 21502.468124
 Validation Loss: 2712.274915
 [18:41:02 UTC] Epoch: 71 LR: 0.000810 Training Loss: 21389.553406
 Validation Loss: 2664.504452
 [18:42:26 UTC] Epoch: 72 LR: 0.000810 Training Loss: 21102.417265
 Validation Loss: 2655.164025
 [18:43:50 UTC] Epoch: 73 LR: 0.000810 Training Loss: 21311.446703
 Validation Loss: 2630.317467
 Validation loss decreased (2638.913090 --> 2630.317467). Saving model ...
 [18:45:15 UTC] Epoch: 74 LR: 0.000810 Training Loss: 21062.947075
 Validation Loss: 2692.471210
 [18:46:37 UTC] Epoch: 75 LR: 0.000810 Training Loss: 21133.964476
 Validation Loss: 2678.888234
 [18:48:00 UTC] Epoch: 76 LR: 0.000729 Training Loss: 20919.637873
 Validation Loss: 2663.366065
 [18:49:22 UTC] Epoch: 77 LR: 0.000729 Training Loss: 21085.275072
 Validation Loss: 2653.682890
 [18:50:44 UTC] Epoch: 78 LR: 0.000729 Training Loss: 20966.737005
 Validation Loss: 2661.213080
 [18:52:06 UTC] Epoch: 79 LR: 0.000729 Training Loss: 20789.035772
 Validation Loss: 2693.929796
 [18:53:28 UTC] Epoch: 80 LR: 0.000729 Training Loss: 20577.011316
 Validation Loss: 2629.493285
 Validation loss decreased (2630.317467 --> 2629.493285). Saving model ...

```

[18:54:51 UTC] Epoch: 81      LR: 0.000729      Training Loss: 21080.557922
Validation Loss: 2587.176175
Validation loss decreased (2629.493285 --> 2587.176175). Saving model ...
[18:56:14 UTC] Epoch: 82      LR: 0.000729      Training Loss: 20968.385176
Validation Loss: 2624.752258
[18:57:40 UTC] Epoch: 83      LR: 0.000729      Training Loss: 20564.677666
Validation Loss: 2648.587368
[18:59:06 UTC] Epoch: 84      LR: 0.000729      Training Loss: 20862.881592
Validation Loss: 2630.990173
[19:00:31 UTC] Epoch: 85      LR: 0.000729      Training Loss: 20909.284798
Validation Loss: 2656.557203
[19:01:54 UTC] Epoch: 86      LR: 0.000729      Training Loss: 20551.945585
Validation Loss: 2650.954657
[19:03:16 UTC] Epoch: 87      LR: 0.000729      Training Loss: 20432.176710
Validation Loss: 2612.331162
[19:04:39 UTC] Epoch: 88      LR: 0.000729      Training Loss: 20567.389801
Validation Loss: 2622.816123
[19:06:01 UTC] Epoch: 89      LR: 0.000729      Training Loss: 20704.525118
Validation Loss: 2620.964123
[19:07:23 UTC] Epoch: 90      LR: 0.000729      Training Loss: 20536.378088
Validation Loss: 2663.576511
[19:08:46 UTC] Epoch: 91      LR: 0.000729      Training Loss: 20452.958563
Validation Loss: 2618.985712
[19:10:08 UTC] Epoch: 92      LR: 0.000729      Training Loss: 20662.580475
Validation Loss: 2708.769854
[19:11:30 UTC] Epoch: 93      LR: 0.000729      Training Loss: 20341.142874
Validation Loss: 2651.643086
[19:12:53 UTC] Epoch: 94      LR: 0.000729      Training Loss: 20512.333860
Validation Loss: 2675.254523
[19:14:15 UTC] Epoch: 95      LR: 0.000729      Training Loss: 20175.657980
Validation Loss: 2643.210196
[19:15:38 UTC] Epoch: 96      LR: 0.000729      Training Loss: 20380.079050
Validation Loss: 2654.943565
[19:17:01 UTC] Epoch: 97      LR: 0.000729      Training Loss: 20012.667730
Validation Loss: 2677.203529
[19:18:24 UTC] Epoch: 98      LR: 0.000729      Training Loss: 20237.932493
Validation Loss: 2646.095762
[19:19:47 UTC] Epoch: 99      LR: 0.000729      Training Loss: 20307.271173
Validation Loss: 2626.256720
[19:21:10 UTC] Epoch: 100     LR: 0.000729      Training Loss: 20080.685913
Validation Loss: 2689.109642

```

[15]: <All keys matched successfully>

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%.

```
[16]: 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: 3.048686

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.

```
[17]: ## TODO: Specify data loaders
batch_size = 64
num_workers=0

data_dir = '/data/dog_images/'
train_dir = os.path.join(data_dir, 'train/')
val_dir = os.path.join(data_dir, 'valid/')
test_dir = os.path.join(data_dir, 'test/')

train_transforms = transforms.Compose([transforms.RandomRotation(30),
                                       transforms.RandomResizedCrop(224),
                                       transforms.RandomHorizontalFlip(),
                                       transforms.ToTensor(),
                                       transforms.Normalize([0.485, 0.456, 0.
↪406],
                                                         [0.229, 0.224, 0.
↪225]]))

val_transforms = transforms.Compose([transforms.Resize(255),
                                     transforms.CenterCrop(224),
                                     transforms.ToTensor(),
                                     transforms.Normalize([0.485, 0.456, 0.406],
                                                         [0.229, 0.224, 0.
↪225]]))

test_transforms = transforms.Compose([transforms.Resize(255),
                                       transforms.CenterCrop(224),
                                       transforms.ToTensor(),
                                       transforms.Normalize([0.485, 0.456, 0.
↪406],
                                                         [0.229, 0.224, 0.
↪225]]))

train_data = datasets.ImageFolder(train_dir, transform=train_transforms)
val_data = datasets.ImageFolder(val_dir, transform=val_transforms)
test_data = datasets.ImageFolder(test_dir, transform=test_transforms)
```

```

print('Num training images: ', len(train_data))
print('Num validation images: ', len(val_data))
print('Num test images: ', len(test_data))

train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,
                                           num_workers=num_workers,
                                           ↪shuffle=True)
val_loader = torch.utils.data.DataLoader(val_data, batch_size=batch_size,
                                          num_workers=num_workers, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size,
                                           num_workers=num_workers,
                                           ↪shuffle=False)

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

```

```

Num training images: 6680
Num validation images: 835
Num test images: 836

```

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`.

```

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

## TODO: Specify model architecture
model_transfer = models.vgg16_bn(pretrained=True)
for param in model_transfer.features.parameters():
    param.requires_grad = False

model_transfer.classifier = nn.Sequential(
    nn.Linear(25088, 512),
    nn.ReLU(),
    nn.Linear(512, 133),
    nn.LogSoftmax(dim=1)
)

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 checked the documentation/structure of the VGG16_BN pre-trained model and what the input of the classifier portion was. Then I fixed the feature layers and added 2 layers for the classifier. The first of these two layer had to meet the input shape base the the feature output, and I used the same output shape and tail structure like I used in the CNN from scratch. After running through a model training the test accuracy was good enough. That's why I kept this architecture.

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.

```
[19]: criterion_transfer = nn.NLLLoss()
      optimizer_transfer = optim.Adam(model_transfer.classifier.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'`.

```
[20]: # train the model
      n_epochs=30
      model_transfer = train(n_epochs, loaders_transfer, model_transfer,
      ↪optimizer_transfer, criterion_transfer, use_cuda, 'model_transfer.pt')

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

CUDA Device: Tesla V100-PCIE-16GB

Memory Allocated: 1.0 GB

```
[19:23:23 UTC] Epoch: 1          LR: 0.001000    Training Loss: 19975.271615
Validation Loss: 958.041075
Validation loss decreased (inf --> 958.041075). Saving model ...
[19:24:55 UTC] Epoch: 2          LR: 0.001000    Training Loss: 10231.230290
Validation Loss: 635.754512
Validation loss decreased (958.041075 --> 635.754512). Saving model ...
[19:26:28 UTC] Epoch: 3          LR: 0.001000    Training Loss: 8558.977414
Validation Loss: 629.210720
Validation loss decreased (635.754512 --> 629.210720). Saving model ...
[19:28:00 UTC] Epoch: 4          LR: 0.001000    Training Loss: 7818.850636
Validation Loss: 546.431923
Validation loss decreased (629.210720 --> 546.431923). Saving model ...
[19:29:33 UTC] Epoch: 5          LR: 0.001000    Training Loss: 7299.551068
```

Validation Loss: 537.067657
 Validation loss decreased (546.431923 --> 537.067657). Saving model ...
 [19:31:04 UTC] Epoch: 6 LR: 0.001000 Training Loss: 6898.366635
 Validation Loss: 521.547827
 Validation loss decreased (537.067657 --> 521.547827). Saving model ...
 [19:32:36 UTC] Epoch: 7 LR: 0.001000 Training Loss: 6756.054753
 Validation Loss: 535.805120
 [19:34:07 UTC] Epoch: 8 LR: 0.001000 Training Loss: 6655.768623
 Validation Loss: 526.460833
 [19:35:37 UTC] Epoch: 9 LR: 0.001000 Training Loss: 6548.543907
 Validation Loss: 517.682921
 Validation loss decreased (521.547827 --> 517.682921). Saving model ...
 [19:37:08 UTC] Epoch: 10 LR: 0.001000 Training Loss: 6352.941887
 Validation Loss: 526.983126
 [19:38:39 UTC] Epoch: 11 LR: 0.001000 Training Loss: 6097.283121
 Validation Loss: 528.137788
 [19:40:09 UTC] Epoch: 12 LR: 0.001000 Training Loss: 6018.770069
 Validation Loss: 494.548126
 Validation loss decreased (517.682921 --> 494.548126). Saving model ...
 [19:41:40 UTC] Epoch: 13 LR: 0.001000 Training Loss: 5832.717118
 Validation Loss: 490.587965
 Validation loss decreased (494.548126 --> 490.587965). Saving model ...
 [19:43:13 UTC] Epoch: 14 LR: 0.001000 Training Loss: 5676.168815
 Validation Loss: 516.311228
 [19:44:45 UTC] Epoch: 15 LR: 0.001000 Training Loss: 5890.073278
 Validation Loss: 552.322058
 [19:46:17 UTC] Epoch: 16 LR: 0.001000 Training Loss: 5703.868167
 Validation Loss: 606.009134
 [19:47:48 UTC] Epoch: 17 LR: 0.001000 Training Loss: 5741.728276
 Validation Loss: 554.242997
 [19:49:19 UTC] Epoch: 18 LR: 0.001000 Training Loss: 5776.942283
 Validation Loss: 530.228691
 [19:50:50 UTC] Epoch: 19 LR: 0.001000 Training Loss: 5501.571335
 Validation Loss: 561.112966
 [19:52:20 UTC] Epoch: 20 LR: 0.001000 Training Loss: 5555.368585
 Validation Loss: 582.715979
 [19:53:49 UTC] Epoch: 21 LR: 0.001000 Training Loss: 5429.841600
 Validation Loss: 593.751277
 [19:55:19 UTC] Epoch: 22 LR: 0.001000 Training Loss: 5374.940749
 Validation Loss: 541.943728
 [19:56:49 UTC] Epoch: 23 LR: 0.001000 Training Loss: 5334.453332
 Validation Loss: 563.468523
 [19:58:19 UTC] Epoch: 24 LR: 0.001000 Training Loss: 5336.116144
 Validation Loss: 578.669789
 [19:59:48 UTC] Epoch: 25 LR: 0.001000 Training Loss: 5499.230329
 Validation Loss: 578.899636
 [20:01:18 UTC] Epoch: 26 LR: 0.000900 Training Loss: 5022.569094
 Validation Loss: 545.072046


```

[20:02:48 UTC] Epoch: 27      LR: 0.000900      Training Loss: 5110.152073
Validation Loss: 522.708822
[20:04:18 UTC] Epoch: 28      LR: 0.000900      Training Loss: 5143.219263
Validation Loss: 588.629686
[20:05:48 UTC] Epoch: 29      LR: 0.000900      Training Loss: 5085.175494
Validation Loss: 577.765621
[20:07:17 UTC] Epoch: 30      LR: 0.000900      Training Loss: 4916.845466
Validation Loss: 539.073174

```

[20]: <All keys matched successfully>

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%.

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

Test Loss: 0.640943

Test Accuracy: 82% (689/836)

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.

```

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

data_transfer = {
    'train' : datasets.ImageFolder(train_dir),
    'valid' : datasets.ImageFolder(val_dir),
    'test'  : datasets.ImageFolder(test_dir)
}

# 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
    global use_cuda
    img_transforms = transforms.Compose([transforms.CenterCrop(224),
                                         transforms.ToTensor(),

```

```

                                transforms.Normalize([0.485, 0.456, 0.
↪406],
                                [0.229, 0.224, 0.
↪225]]))
    img = img_transforms(Image.open(img_path))[None , :]
    if use_cuda:
        img = img.cuda()
    model_transfer.eval()
    output = model_transfer(img)
    _, predictions = torch.max(output, 1)
    if use_cuda:
        predictions = predictions.cpu()
    return class_names[predictions.numpy()[0]]

```

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!

```

hello, human!
0
200
400
600
800
1000
1200
1400
0 500 1000
You look like a ...
Chinese_shar-pei

```



1.1.18 (IMPLEMENTATION) Write your Algorithm

```

[23]: ### TODO: 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
img = Image.open(img_path)
if dog_detector(img_path):
    print(f"hello, dog!")
    imgplot = plt.imshow(img)
    plt.show()
    print("You look like a ...")
    print(predict_breed_transfer(img_path))
elif face_detector(img_path):
    print(f"hello, human!")
    imgplot = plt.imshow(img)
    plt.show()
    print("You look like a ...")
    print(predict_breed_transfer(img_path))
else:
    print("An error has occurred, please contact support")

```

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: Possible ways to improve my ML model and code 1. Add more images (with different lighting, angles, etc.), for each class (to also have the same count of images for each class 1. Add more ways to do image augmentation, e.g. shifting images, adding noise, blurring, changing brightness, etc. to produce more images 1. Hyperparameter tuning by using cloud services like AWS Sagemaker or Azure ML Services 1. Test/Use other pre-trained models 1. Provide better ways to handle exceptions if images contain neither a human or dog.

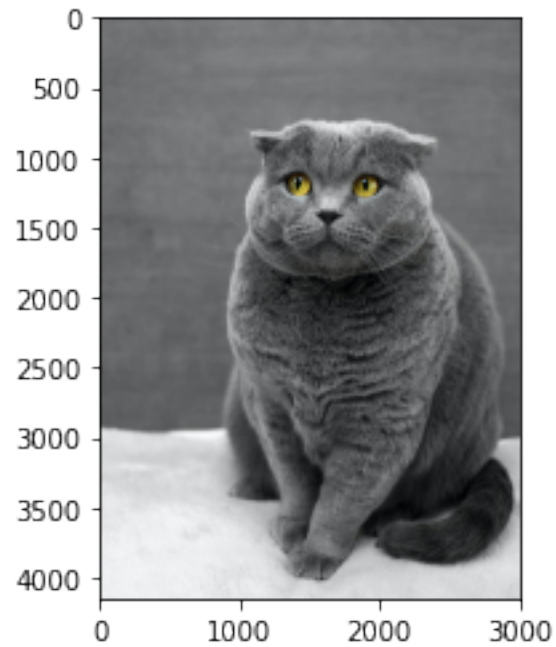
```

[24]: ## 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.
      local_files = np.array(glob("local/*"))

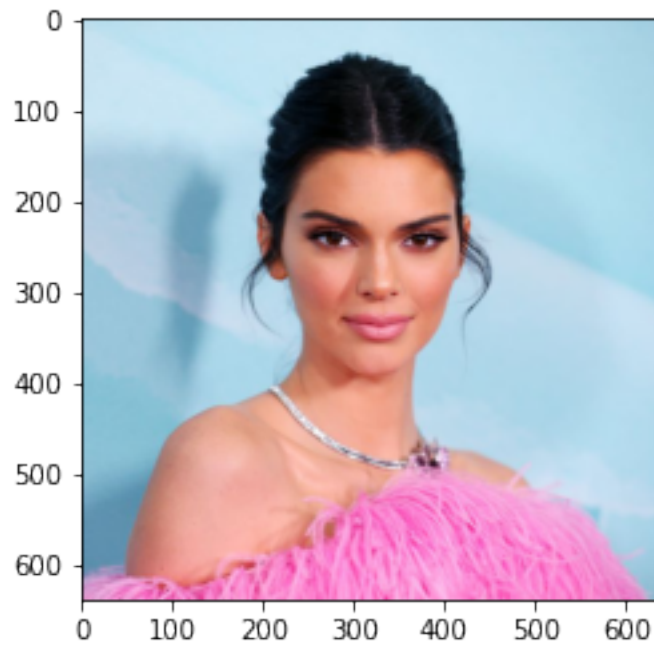
      ## suggested code, below
      for file in np.hstack((local_files, human_files[:3], dog_files[:3])):
          print("file -> ", file)
          run_app(file)

```

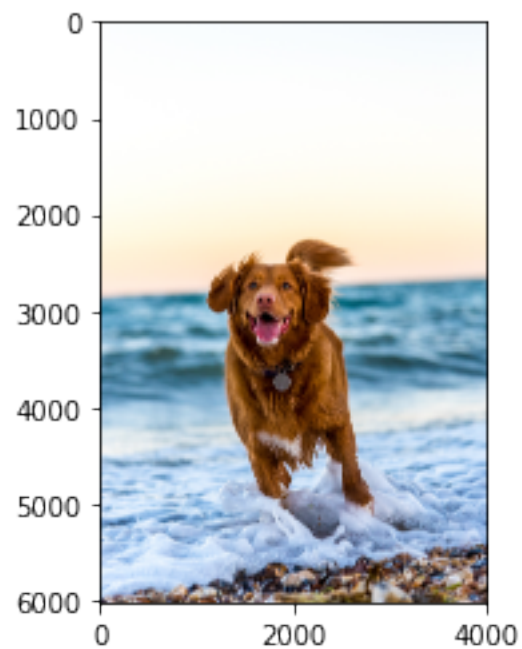
```
file -> local/misc1.jpg
An error has occurred, please contact support
file -> local/cat2.jpg
hello, human!
```



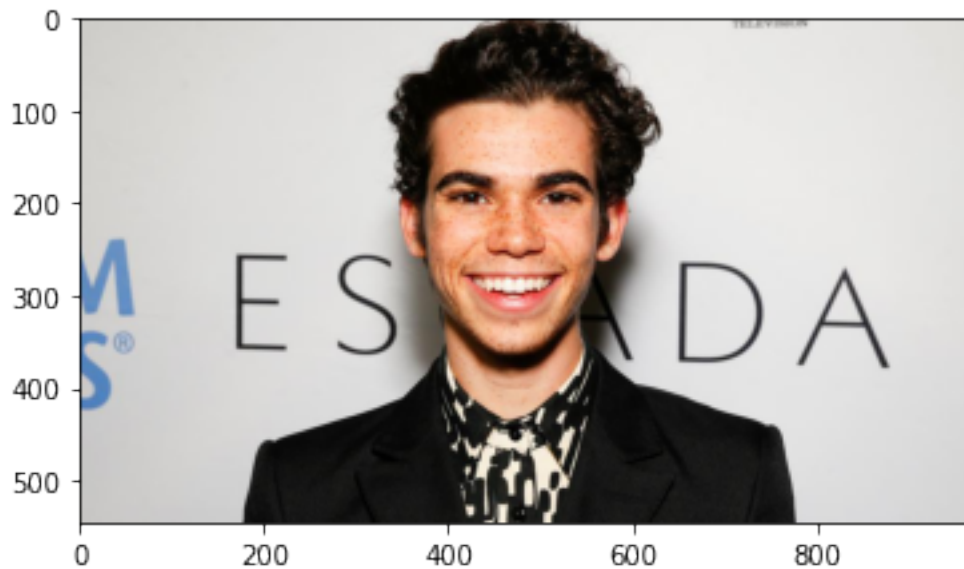
```
You look like a ...
Affenpinscher
file -> local/cat1.jpg
An error has occurred, please contact support
file -> local/human1.jpeg
hello, human!
```



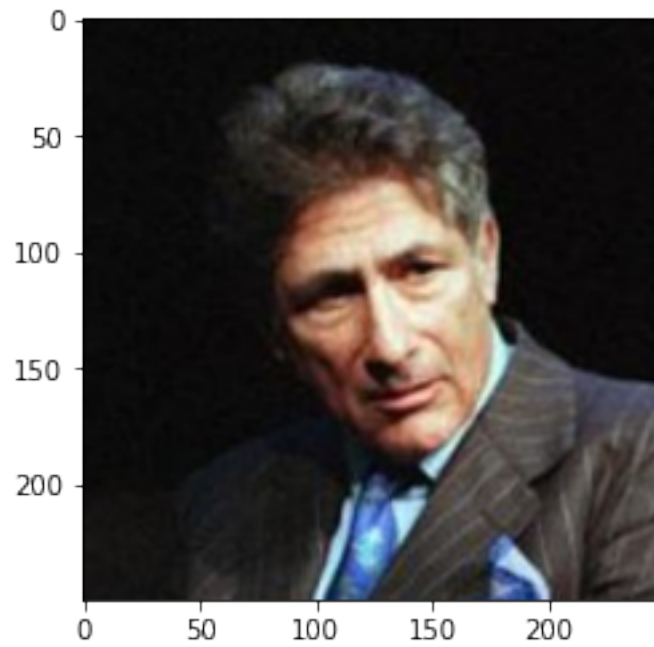
You look like a ...
Labrador retriever
file -> local/misc2.jpg
An error has occurred, please contact support
file -> local/dog2.jpg
hello, human!



```
You look like a ...  
German wirehaired pointer  
file -> local/dog1.jpg  
An error has occurred, please contact support  
file -> local/human2.jpeg  
hello, human!
```



```
You look like a ...  
American staffordshire terrier  
file -> /data/lfw/Edward_Said/Edward_Said_0001.jpg  
hello, human!
```

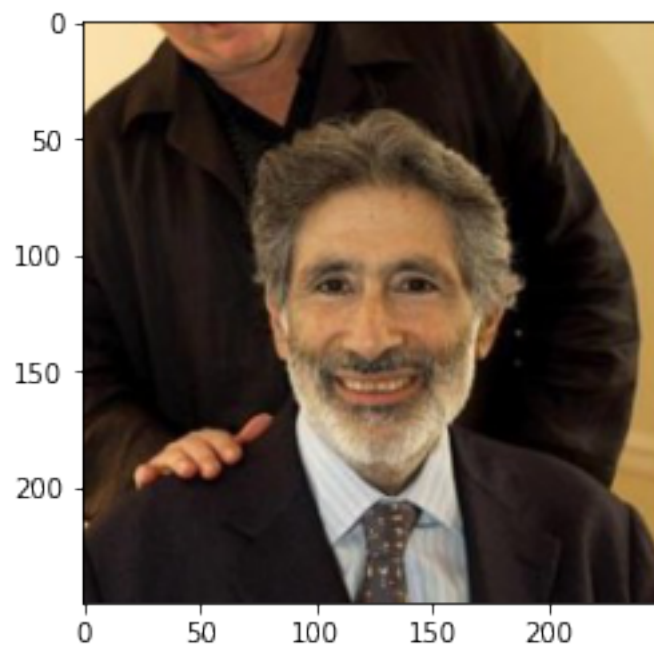


You look like a ...

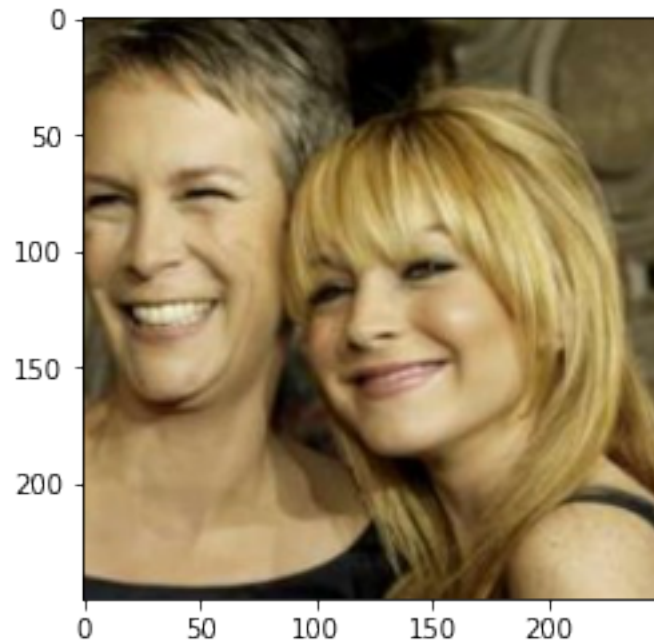
Collie

file -> /data/lfw/Edward_Said/Edward_Said_0002.jpg

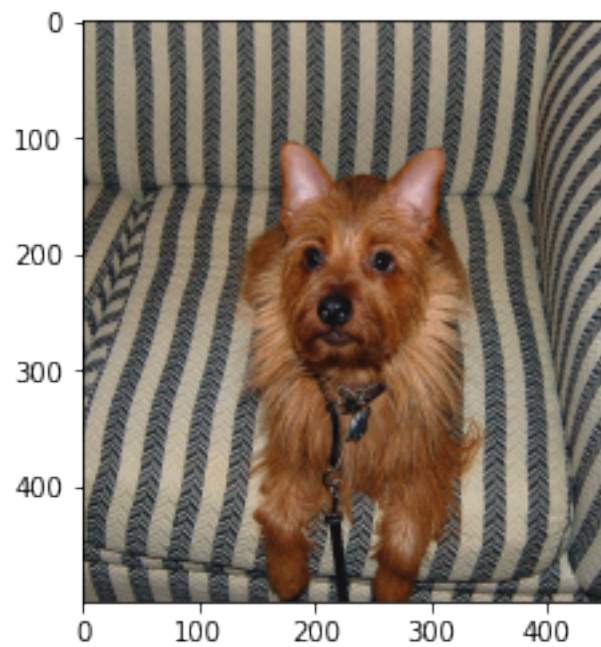
hello, human!



```
You look like a ...  
Doberman pinscher  
file -> /data/lfw/Lindsay_Lohan/Lindsay_Lohan_0001.jpg  
hello, human!
```



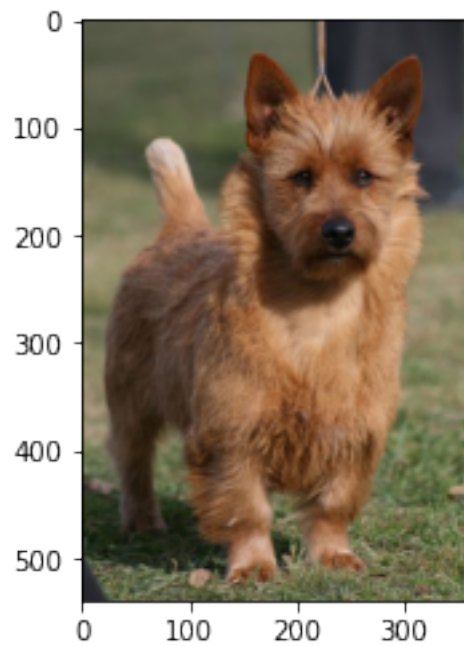
```
You look like a ...  
Bichon frise  
file ->  
/data/dog_images/test/013.Australian_terrier/Australian_terrier_00897.jpg  
hello, dog!
```

You look like a ...
Australian terrier
file ->
/data/dog_images/test/013.Australian_terrier/Australian_terrier_00918.jpg
hello, dog!



```
You look like a ...  
Irish terrier  
file ->  
/data/dog_images/test/013.Australian_terrier/Australian_terrier_00930.jpg  
hello, dog!
```



```
You look like a ...  
Irish terrier
```

```
[ ]:
```