dog_app

March 23, 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
```

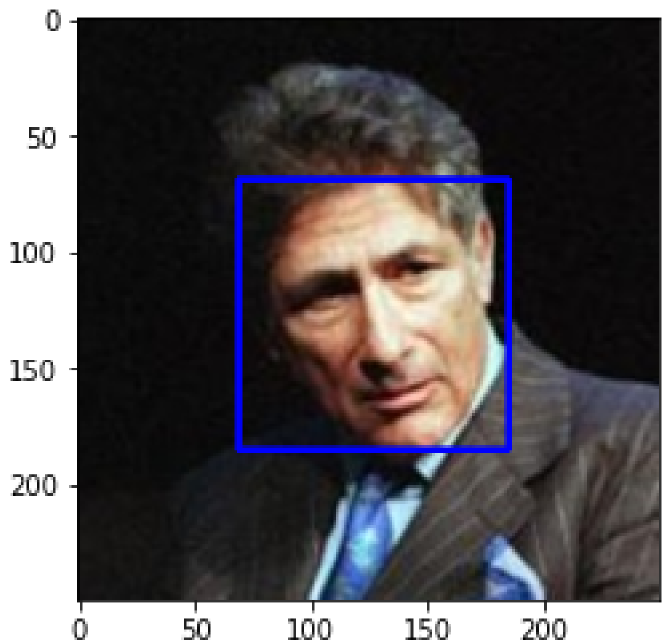
```python
print('Number of faces detected:', len(faces))

# 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}␣
 ↪files')

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

Step 2: Detect Dogs

In this section, we use a pre-trained model to detect dogs in images.

1.1.3 Obtain Pre-trained VGG-16 Model

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

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

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

1.1.7 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

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

[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]:
```python
import torch.nn as nn
import torch.nn.functional as F

# define the CNN architecture
class Net(nn.Module):
    ### TODO: choose an architecture, and complete the class
    def __init__(self):
        super(Net, self).__init__()
        ## Define layers of a CNN
        self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
        self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(64 * 28 * 28, 512)
        self.fc2 = nn.Linear(512, 133)
        self.dropout = nn.Dropout(p=0.4)

    def forward(self, x):
        ## Define forward behavior
        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.dropout(x)
        x = x.view(-1, 64 * 28 * 28)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.log_softmax(self.fc2(x), dim=1)
        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 deliefer 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 tweek 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
         MaxPool2d-2         [-1, 16, 112, 112]               0
            Conv2d-3         [-1, 32, 112, 112]           4,640
         MaxPool2d-4           [-1, 32, 56, 56]               0
            Conv2d-5           [-1, 64, 56, 56]          18,496
         MaxPool2d-6           [-1, 64, 28, 28]               0
           Dropout-7           [-1, 64, 28, 28]               0
            Linear-8                  [-1, 512]      25,690,624
           Dropout-9                  [-1, 512]               0
           Linear-10                  [-1, 133]          68,229
================================================================
Total params: 25,782,437
Trainable params: 25,782,437
Non-trainable params: 0
```

```
----------------------------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 13.79
Params size (MB): 98.35
Estimated Total Size (MB): 112.72
----------------------------------------------------------------
```

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]:
```python
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]:
```python
# 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=50, gamma=0.9)

    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: {} \tTraining Loss: {:.6f} \tValidation␣
↪Loss: {:.6f}'.format(
            time.strftime('%H:%M:%S %Z'),
            epoch,
            scheduler.get_lr(),
            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
[11:43:28 UTC] Epoch: 1         LR: [0.001]     Training Loss: 32651.051121
Validation Loss: 3990.083986
Validation loss decreased (inf --> 3990.083986).  Saving model …
[11:44:55 UTC] Epoch: 2         LR: [0.001]     Training Loss: 31759.180145
Validation Loss: 3796.561300
Validation loss decreased (3990.083986 --> 3796.561300).  Saving model …
[11:46:22 UTC] Epoch: 3         LR: [0.001]     Training Loss: 30807.028530
Validation Loss: 3736.954005
Validation loss decreased (3796.561300 --> 3736.954005).  Saving model …
[11:47:50 UTC] Epoch: 4         LR: [0.001]     Training Loss: 30194.929073
Validation Loss: 3612.514490
Validation loss decreased (3736.954005 --> 3612.514490).  Saving model …
[11:49:18 UTC] Epoch: 5         LR: [0.001]     Training Loss: 29725.682728
Validation Loss: 3591.163340
Validation loss decreased (3612.514490 --> 3591.163340).  Saving model …
[11:50:46 UTC] Epoch: 6         LR: [0.001]     Training Loss: 29483.640526
Validation Loss: 3508.736771
Validation loss decreased (3591.163340 --> 3508.736771).  Saving model …
[11:52:13 UTC] Epoch: 7         LR: [0.001]     Training Loss: 29067.598763
Validation Loss: 3466.810801
Validation loss decreased (3508.736771 --> 3466.810801).  Saving model …
[11:53:42 UTC] Epoch: 8         LR: [0.001]     Training Loss: 28674.673962
Validation Loss: 3410.765670
Validation loss decreased (3466.810801 --> 3410.765670).  Saving model …
[11:55:10 UTC] Epoch: 9         LR: [0.001]     Training Loss: 28344.774681
Validation Loss: 3390.386928
Validation loss decreased (3410.765670 --> 3390.386928).  Saving model …
[11:56:35 UTC] Epoch: 10        LR: [0.001]     Training Loss: 28042.293398
Validation Loss: 3369.328226
```

```
Validation loss decreased (3390.386928 --> 3369.328226).  Saving model …
[11:58:00 UTC] Epoch: 11        LR: [0.001]     Training Loss: 27836.902418
Validation Loss: 3303.778893
Validation loss decreased (3369.328226 --> 3303.778893).  Saving model …
[11:59:24 UTC] Epoch: 12        LR: [0.001]     Training Loss: 27740.873119
Validation Loss: 3288.386326
Validation loss decreased (3303.778893 --> 3288.386326).  Saving model …
[12:00:49 UTC] Epoch: 13        LR: [0.001]     Training Loss: 27391.068645
Validation Loss: 3244.398023
Validation loss decreased (3288.386326 --> 3244.398023).  Saving model …
[12:02:14 UTC] Epoch: 14        LR: [0.001]     Training Loss: 26976.111368
Validation Loss: 3187.968235
Validation loss decreased (3244.398023 --> 3187.968235).  Saving model …
[12:03:41 UTC] Epoch: 15        LR: [0.001]     Training Loss: 26970.650343
Validation Loss: 3186.178970
Validation loss decreased (3187.968235 --> 3186.178970).  Saving model …
[12:05:09 UTC] Epoch: 16        LR: [0.001]     Training Loss: 26651.029133
Validation Loss: 3130.615744
Validation loss decreased (3186.178970 --> 3130.615744).  Saving model …
[12:06:37 UTC] Epoch: 17        LR: [0.001]     Training Loss: 26406.955172
Validation Loss: 3201.583053
[12:08:04 UTC] Epoch: 18        LR: [0.001]     Training Loss: 26090.685707
Validation Loss: 3080.737947
Validation loss decreased (3130.615744 --> 3080.737947).  Saving model …
[12:09:32 UTC] Epoch: 19        LR: [0.001]     Training Loss: 26184.944487
Validation Loss: 3162.566780
[12:10:59 UTC] Epoch: 20        LR: [0.001]     Training Loss: 25766.637211
Validation Loss: 3030.500700
Validation loss decreased (3080.737947 --> 3030.500700).  Saving model …
[12:12:26 UTC] Epoch: 21        LR: [0.001]     Training Loss: 25451.264618
Validation Loss: 3012.295437
Validation loss decreased (3030.500700 --> 3012.295437).  Saving model …
[12:13:53 UTC] Epoch: 22        LR: [0.001]     Training Loss: 25560.384367
Validation Loss: 2964.018837
Validation loss decreased (3012.295437 --> 2964.018837).  Saving model …
[12:15:21 UTC] Epoch: 23        LR: [0.001]     Training Loss: 25232.043829
Validation Loss: 2959.508121
Validation loss decreased (2964.018837 --> 2959.508121).  Saving model …
[12:16:48 UTC] Epoch: 24        LR: [0.001]     Training Loss: 25382.398142
Validation Loss: 2986.288307
[12:18:15 UTC] Epoch: 25        LR: [0.001]     Training Loss: 25038.292519
Validation Loss: 2947.426337
Validation loss decreased (2959.508121 --> 2947.426337).  Saving model …
[12:19:42 UTC] Epoch: 26        LR: [0.001]     Training Loss: 24800.835779
Validation Loss: 2891.445107
Validation loss decreased (2947.426337 --> 2891.445107).  Saving model …
[12:21:10 UTC] Epoch: 27        LR: [0.001]     Training Loss: 24624.372452
Validation Loss: 2920.019126
```

```
[12:22:37 UTC] Epoch: 28        LR: [0.001]     Training Loss: 24501.282091
Validation Loss: 2914.720865
[12:24:04 UTC] Epoch: 29        LR: [0.001]     Training Loss: 24591.943047
Validation Loss: 2889.036699
Validation loss decreased (2891.445107 --> 2889.036699).  Saving model …
[12:25:31 UTC] Epoch: 30        LR: [0.001]     Training Loss: 24307.702501
Validation Loss: 2861.641175
Validation loss decreased (2889.036699 --> 2861.641175).  Saving model …
[12:26:56 UTC] Epoch: 31        LR: [0.001]     Training Loss: 24197.328051
Validation Loss: 2837.930471
Validation loss decreased (2861.641175 --> 2837.930471).  Saving model …
[12:28:19 UTC] Epoch: 32        LR: [0.001]     Training Loss: 24066.132940
Validation Loss: 2880.680111
[12:29:43 UTC] Epoch: 33        LR: [0.001]     Training Loss: 24230.554695
Validation Loss: 2852.564818
[12:31:06 UTC] Epoch: 34        LR: [0.001]     Training Loss: 24147.640104
Validation Loss: 2826.096497
Validation loss decreased (2837.930471 --> 2826.096497).  Saving model …
[12:32:31 UTC] Epoch: 35        LR: [0.001]     Training Loss: 24079.572260
Validation Loss: 2879.337068
[12:33:57 UTC] Epoch: 36        LR: [0.001]     Training Loss: 24063.486725
Validation Loss: 2848.747950
[12:35:24 UTC] Epoch: 37        LR: [0.001]     Training Loss: 23765.749157
Validation Loss: 2891.347237
[12:36:50 UTC] Epoch: 38        LR: [0.001]     Training Loss: 23722.655701
Validation Loss: 2884.091669
[12:38:18 UTC] Epoch: 39        LR: [0.001]     Training Loss: 23578.820127
Validation Loss: 2788.766747
Validation loss decreased (2826.096497 --> 2788.766747).  Saving model …
[12:39:46 UTC] Epoch: 40        LR: [0.001]     Training Loss: 23548.046364
Validation Loss: 2792.255716
[12:41:13 UTC] Epoch: 41        LR: [0.001]     Training Loss: 23575.519859
Validation Loss: 2790.525541
[12:42:39 UTC] Epoch: 42        LR: [0.001]     Training Loss: 23325.727316
Validation Loss: 2795.919219
[12:44:05 UTC] Epoch: 43        LR: [0.001]     Training Loss: 23268.400726
Validation Loss: 2778.062009
Validation loss decreased (2788.766747 --> 2778.062009).  Saving model …
[12:45:33 UTC] Epoch: 44        LR: [0.001]     Training Loss: 23223.110247
Validation Loss: 2802.697781
[12:46:59 UTC] Epoch: 45        LR: [0.001]     Training Loss: 23179.503014
Validation Loss: 2765.134291
Validation loss decreased (2778.062009 --> 2765.134291).  Saving model …
[12:48:27 UTC] Epoch: 46        LR: [0.001]     Training Loss: 23304.643448
Validation Loss: 2809.515805
[12:49:54 UTC] Epoch: 47        LR: [0.001]     Training Loss: 22927.141367
Validation Loss: 2757.000488
Validation loss decreased (2765.134291 --> 2757.000488).  Saving model …
```

```
[12:51:21 UTC] Epoch: 48        LR: [0.001]     Training Loss: 22954.605228
Validation Loss: 2738.206357
Validation loss decreased (2757.000488 --> 2738.206357).  Saving model …
[12:52:48 UTC] Epoch: 49        LR: [0.001]     Training Loss: 23037.601984
Validation Loss: 2737.940920
Validation loss decreased (2738.206357 --> 2737.940920).  Saving model …
[12:54:16 UTC] Epoch: 50        LR: [0.001]     Training Loss: 22832.444290
Validation Loss: 2760.991824
[12:55:43 UTC] Epoch: 51        LR: [0.0009000000000000001]     Training Loss:
22603.994535     Validation Loss: 2739.849085
[12:57:08 UTC] Epoch: 52        LR: [0.0009000000000000001]     Training Loss:
22516.924507     Validation Loss: 2818.210475
[12:58:30 UTC] Epoch: 53        LR: [0.0009000000000000001]     Training Loss:
22709.108793     Validation Loss: 2739.552328
[12:59:52 UTC] Epoch: 54        LR: [0.0009000000000000001]     Training Loss:
22658.523420     Validation Loss: 2686.867562
Validation loss decreased (2737.940920 --> 2686.867562).  Saving model …
[13:01:15 UTC] Epoch: 55        LR: [0.0009000000000000001]     Training Loss:
22429.985035     Validation Loss: 2714.423144
[13:02:38 UTC] Epoch: 56        LR: [0.0009000000000000001]     Training Loss:
22629.084961     Validation Loss: 2717.783019
[13:04:05 UTC] Epoch: 57        LR: [0.0009000000000000001]     Training Loss:
22470.883135     Validation Loss: 2722.663224
[13:05:32 UTC] Epoch: 58        LR: [0.0009000000000000001]     Training Loss:
22416.123964     Validation Loss: 2714.486282
[13:06:58 UTC] Epoch: 59        LR: [0.0009000000000000001]     Training Loss:
22081.970995     Validation Loss: 2705.896204
[13:08:24 UTC] Epoch: 60        LR: [0.0009000000000000001]     Training Loss:
22427.530668     Validation Loss: 2721.377263
[13:09:51 UTC] Epoch: 61        LR: [0.0009000000000000001]     Training Loss:
22191.711630     Validation Loss: 2665.570457
Validation loss decreased (2686.867562 --> 2665.570457).  Saving model …
[13:11:19 UTC] Epoch: 62        LR: [0.0009000000000000001]     Training Loss:
22182.468782     Validation Loss: 2778.280014
[13:12:46 UTC] Epoch: 63        LR: [0.0009000000000000001]     Training Loss:
22182.258583     Validation Loss: 2674.613253
[13:14:12 UTC] Epoch: 64        LR: [0.0009000000000000001]     Training Loss:
22152.200745     Validation Loss: 2708.091594
[13:15:38 UTC] Epoch: 65        LR: [0.0009000000000000001]     Training Loss:
22142.603645     Validation Loss: 2728.085694
[13:17:05 UTC] Epoch: 66        LR: [0.0009000000000000001]     Training Loss:
22067.592089     Validation Loss: 2690.771099
[13:18:31 UTC] Epoch: 67        LR: [0.0009000000000000001]     Training Loss:
21731.638388     Validation Loss: 2657.268123
Validation loss decreased (2665.570457 --> 2657.268123).  Saving model …
[13:19:58 UTC] Epoch: 68        LR: [0.0009000000000000001]     Training Loss:
22011.602646     Validation Loss: 2659.506314
[13:21:25 UTC] Epoch: 69        LR: [0.0009000000000000001]     Training Loss:
```

```
22102.736502     Validation Loss: 2661.886173
[13:22:51 UTC] Epoch: 70        LR: [0.0009000000000000001]     Training Loss:
21823.727972     Validation Loss: 2679.257533
[13:24:18 UTC] Epoch: 71        LR: [0.0009000000000000001]     Training Loss:
21775.269484     Validation Loss: 2612.934145
Validation loss decreased (2657.268123 --> 2612.934145).  Saving model …
[13:25:44 UTC] Epoch: 72        LR: [0.0009000000000000001]     Training Loss:
21617.593346     Validation Loss: 2669.371229
[13:27:10 UTC] Epoch: 73        LR: [0.0009000000000000001]     Training Loss:
21812.076059     Validation Loss: 2646.236751
[13:28:33 UTC] Epoch: 74        LR: [0.0009000000000000001]     Training Loss:
21742.208277     Validation Loss: 2625.614487
[13:29:56 UTC] Epoch: 75        LR: [0.0009000000000000001]     Training Loss:
21694.450727     Validation Loss: 2658.453210
[13:31:19 UTC] Epoch: 76        LR: [0.0009000000000000001]     Training Loss:
21474.541555     Validation Loss: 2609.426294
Validation loss decreased (2612.934145 --> 2609.426294).  Saving model …
[13:32:42 UTC] Epoch: 77        LR: [0.0009000000000000001]     Training Loss:
21556.678953     Validation Loss: 2637.389127
[13:34:09 UTC] Epoch: 78        LR: [0.0009000000000000001]     Training Loss:
21381.277868     Validation Loss: 2664.582187
[13:35:35 UTC] Epoch: 79        LR: [0.0009000000000000001]     Training Loss:
21356.870470     Validation Loss: 2617.134401
[13:37:02 UTC] Epoch: 80        LR: [0.0009000000000000001]     Training Loss:
21188.112143     Validation Loss: 2696.295825
[13:38:28 UTC] Epoch: 81        LR: [0.0009000000000000001]     Training Loss:
21306.221268     Validation Loss: 2634.417409
[13:39:54 UTC] Epoch: 82        LR: [0.0009000000000000001]     Training Loss:
21329.373159     Validation Loss: 2610.004219
[13:41:21 UTC] Epoch: 83        LR: [0.0009000000000000001]     Training Loss:
21381.589821     Validation Loss: 2600.447167
Validation loss decreased (2609.426294 --> 2600.447167).  Saving model …
[13:42:49 UTC] Epoch: 84        LR: [0.0009000000000000001]     Training Loss:
21363.349876     Validation Loss: 2650.807292
[13:44:16 UTC] Epoch: 85        LR: [0.0009000000000000001]     Training Loss:
21121.845541     Validation Loss: 2657.648361
[13:45:42 UTC] Epoch: 86        LR: [0.0009000000000000001]     Training Loss:
21127.886084     Validation Loss: 2628.706581
[13:47:09 UTC] Epoch: 87        LR: [0.0009000000000000001]     Training Loss:
21108.637110     Validation Loss: 2646.654927
[13:48:35 UTC] Epoch: 88        LR: [0.0009000000000000001]     Training Loss:
21302.624718     Validation Loss: 2560.243587
Validation loss decreased (2600.447167 --> 2560.243587).  Saving model …
[13:50:02 UTC] Epoch: 89        LR: [0.0009000000000000001]     Training Loss:
20959.273384     Validation Loss: 2596.350504
[13:51:28 UTC] Epoch: 90        LR: [0.0009000000000000001]     Training Loss:
21001.760622     Validation Loss: 2581.490982
[13:52:55 UTC] Epoch: 91        LR: [0.0009000000000000001]     Training Loss:
```

```
21185.073969     Validation Loss: 2657.596701
[13:54:22 UTC] Epoch: 92        LR: [0.0009000000000000001]     Training Loss:
20765.165247     Validation Loss: 2626.277271
[13:55:48 UTC] Epoch: 93        LR: [0.0009000000000000001]     Training Loss:
21026.735645     Validation Loss: 2685.548317
[13:57:14 UTC] Epoch: 94        LR: [0.0009000000000000001]     Training Loss:
21095.799812     Validation Loss: 2599.594627
[13:58:37 UTC] Epoch: 95        LR: [0.0009000000000000001]     Training Loss:
20865.499781     Validation Loss: 2582.590023
[14:00:00 UTC] Epoch: 96        LR: [0.0009000000000000001]     Training Loss:
21019.104933     Validation Loss: 2619.780005
[14:01:23 UTC] Epoch: 97        LR: [0.0009000000000000001]     Training Loss:
20683.005648     Validation Loss: 2613.481182
[14:02:46 UTC] Epoch: 98        LR: [0.0009000000000000001]     Training Loss:
20682.813890     Validation Loss: 2622.951355
[14:04:12 UTC] Epoch: 99        LR: [0.0009000000000000001]     Training Loss:
20740.453896     Validation Loss: 2673.947548
[14:05:38 UTC] Epoch: 100       LR: [0.0009000000000000001]     Training Loss:
20605.845522     Validation Loss: 2587.722453
```

[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]:
```python
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: 3.058699


Test Accuracy: 24% (208/836)
```

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]:
```python
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 classifer. 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]:
```python
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
[14:07:49 UTC] Epoch: 1         LR: [0.001]     Training Loss: 19919.870417
Validation Loss: 890.987163
Validation loss decreased (inf --> 890.987163).  Saving model …
[14:09:25 UTC] Epoch: 2         LR: [0.001]     Training Loss: 10101.270415
Validation Loss: 658.844847
Validation loss decreased (890.987163 --> 658.844847).  Saving model …
[14:11:02 UTC] Epoch: 3         LR: [0.001]     Training Loss: 8639.788027
Validation Loss: 657.915735
Validation loss decreased (658.844847 --> 657.915735).  Saving model …
[14:12:37 UTC] Epoch: 4         LR: [0.001]     Training Loss: 7830.346040
Validation Loss: 498.054647
Validation loss decreased (657.915735 --> 498.054647).  Saving model …
[14:14:12 UTC] Epoch: 5         LR: [0.001]     Training Loss: 7475.688619
Validation Loss: 582.812002
[14:15:47 UTC] Epoch: 6         LR: [0.001]     Training Loss: 7087.457613
Validation Loss: 541.271941
[14:17:22 UTC] Epoch: 7         LR: [0.001]     Training Loss: 6889.489479
Validation Loss: 489.343630
Validation loss decreased (498.054647 --> 489.343630).  Saving model …
[14:18:57 UTC] Epoch: 8         LR: [0.001]     Training Loss: 6442.146340
Validation Loss: 497.057448
[14:20:31 UTC] Epoch: 9         LR: [0.001]     Training Loss: 6502.565487
Validation Loss: 517.220642
[14:22:05 UTC] Epoch: 10        LR: [0.001]     Training Loss: 6187.165226
Validation Loss: 514.298538
[14:23:40 UTC] Epoch: 11        LR: [0.001]     Training Loss: 6301.863658
Validation Loss: 437.207703
Validation loss decreased (489.343630 --> 437.207703).  Saving model …
[14:25:15 UTC] Epoch: 12        LR: [0.001]     Training Loss: 5966.055787
Validation Loss: 491.221255
[14:26:50 UTC] Epoch: 13        LR: [0.001]     Training Loss: 6170.382189
Validation Loss: 512.938025
[14:28:23 UTC] Epoch: 14        LR: [0.001]     Training Loss: 5944.227346
Validation Loss: 487.595534
[14:29:54 UTC] Epoch: 15        LR: [0.001]     Training Loss: 5873.598657
Validation Loss: 559.457891
```

```
[14:31:26 UTC] Epoch: 16        LR: [0.001]     Training Loss: 5916.860840
Validation Loss: 569.030219
[14:32:57 UTC] Epoch: 17        LR: [0.001]     Training Loss: 5836.252331
Validation Loss: 549.621708
[14:34:30 UTC] Epoch: 18        LR: [0.001]     Training Loss: 5507.756001
Validation Loss: 531.393442
[14:36:05 UTC] Epoch: 19        LR: [0.001]     Training Loss: 5637.135949
Validation Loss: 534.329271
[14:37:40 UTC] Epoch: 20        LR: [0.001]     Training Loss: 5555.402829
Validation Loss: 568.805254
[14:39:15 UTC] Epoch: 21        LR: [0.001]     Training Loss: 5681.228722
Validation Loss: 566.699930
[14:40:50 UTC] Epoch: 22        LR: [0.001]     Training Loss: 5439.244751
Validation Loss: 610.986230
[14:42:25 UTC] Epoch: 23        LR: [0.001]     Training Loss: 5146.048045
Validation Loss: 531.447920
[14:43:59 UTC] Epoch: 24        LR: [0.001]     Training Loss: 5376.938160
Validation Loss: 547.493917
[14:45:33 UTC] Epoch: 25        LR: [0.001]     Training Loss: 5323.899542
Validation Loss: 539.518768
[14:47:08 UTC] Epoch: 26        LR: [0.001]     Training Loss: 5274.369201
Validation Loss: 533.980906
[14:48:42 UTC] Epoch: 27        LR: [0.001]     Training Loss: 5206.362816
Validation Loss: 565.923264
[14:50:17 UTC] Epoch: 28        LR: [0.001]     Training Loss: 5263.674394
Validation Loss: 617.691031
[14:51:51 UTC] Epoch: 29        LR: [0.001]     Training Loss: 5191.112370
Validation Loss: 675.351403
[14:53:26 UTC] Epoch: 30        LR: [0.001]     Training Loss: 5139.419198
Validation Loss: 605.940219
```

[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: 0.618399


Test Accuracy: 82% (688/836)
```

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.

[36]:
```
### 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]]
```

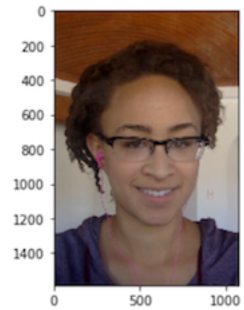
Step 5: Write your Algorithm

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!
```




```
You look like a ...
Chinese_shar-pei
```

1.1.18 (IMPLEMENTATION) Write your Algorithm

[37]:
```python
### 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.

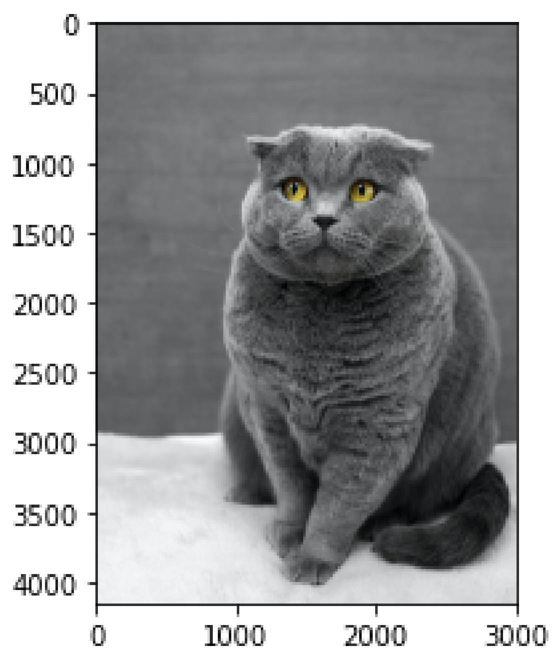
Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

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.

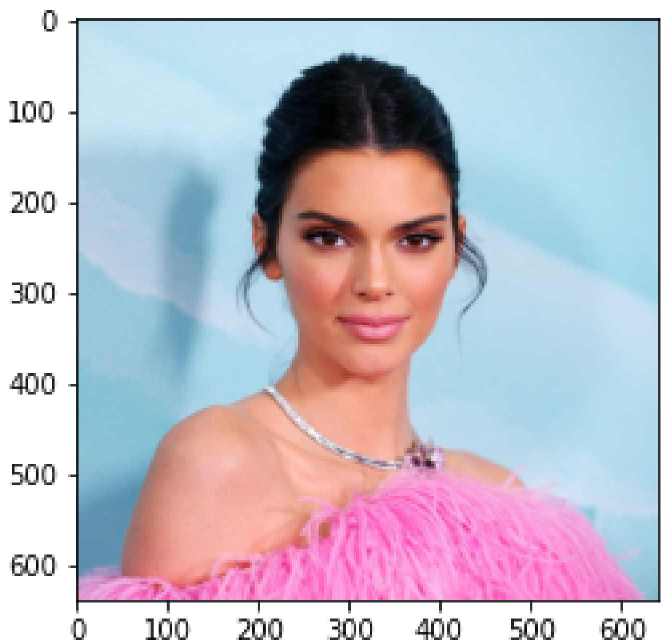
[38]:
```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.
local_files = np.array(glob("local/*"))

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

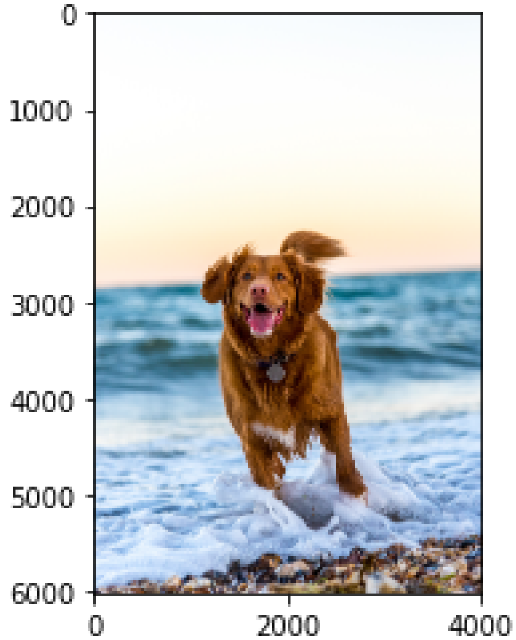
```
file ->  local/misc1.jpg
An error has occurred, please contact support
file ->  local/cat2.jpg
hello, human!
```



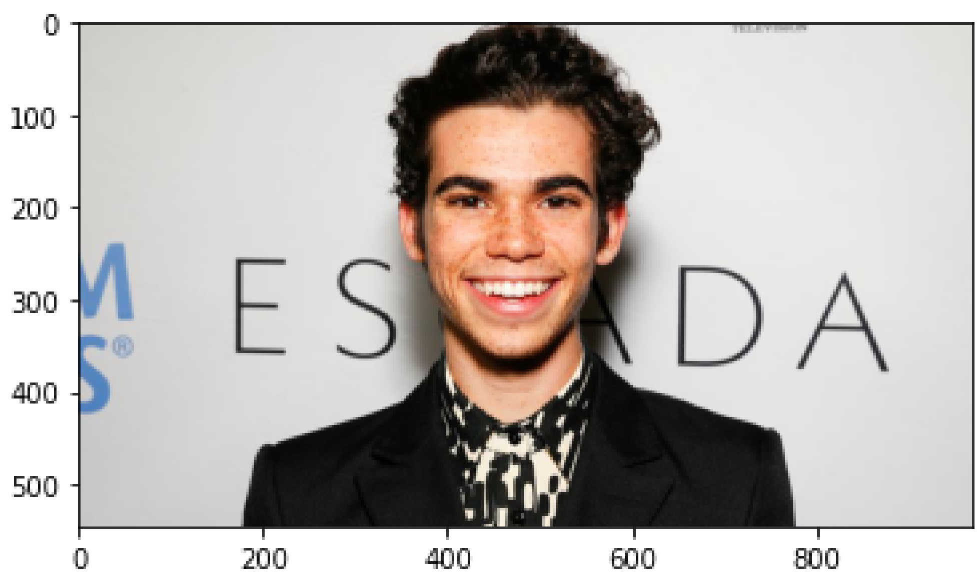
```
You look like a …
Affenpinscher
file ->  local/cat1.jpg
An error has occurred, please contact support
file ->  local/human1.jpeg
hello, human!
```



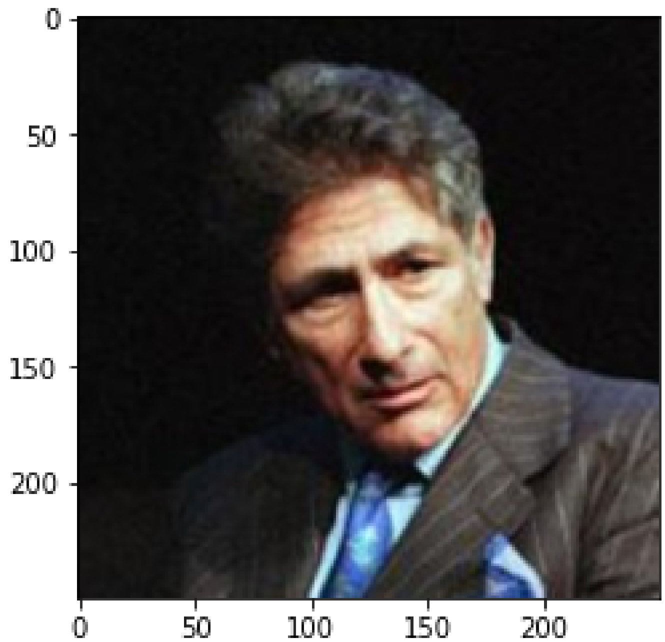

```
You look like a …
American staffordshire terrier
file ->  local/misc2.jpg
An error has occurred, please contact support
file ->  local/dog2.jpg
hello, human!
```



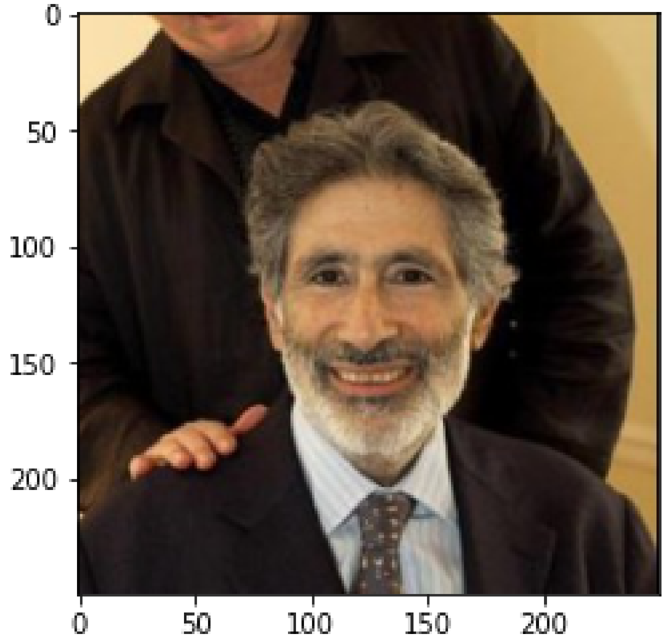

```
You look like a …
Nova scotia duck tolling retriever
file ->  local/dog1.jpg
An error has occurred, please contact support
file ->  local/human2.jpeg
hello, human!
```



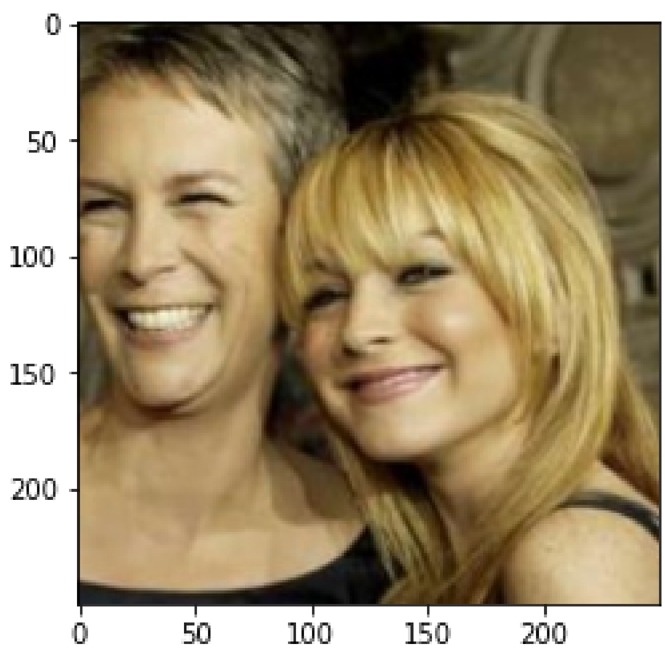
```
You look like a …
American staffordshire terrier
file ->  /data/lfw/Edward_Said/Edward_Said_0001.jpg
hello, human!
```



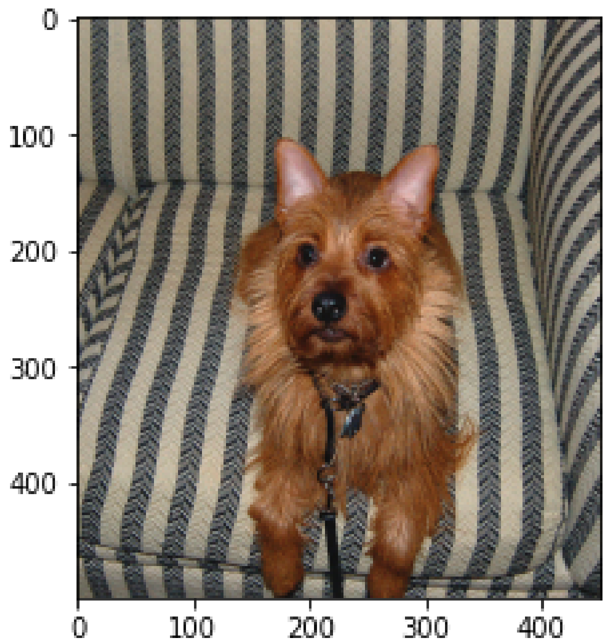
```
You look like a …
Silky terrier
file ->  /data/lfw/Edward_Said/Edward_Said_0002.jpg
hello, human!
```



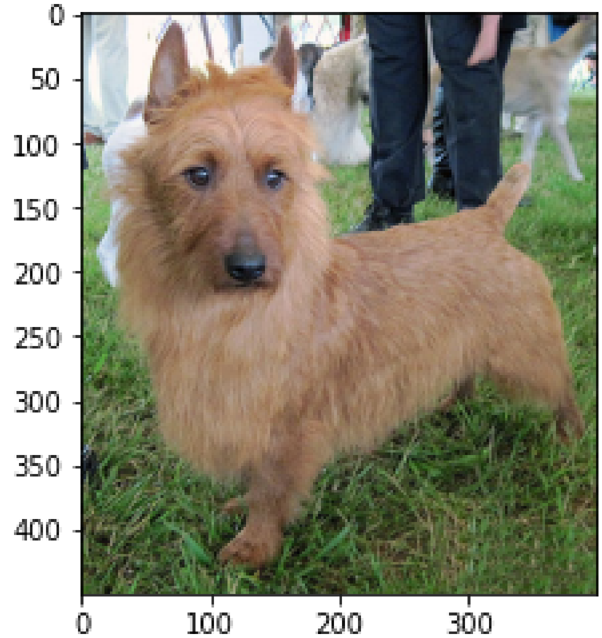

```
You look like a …
Labrador retriever
file ->  /data/lfw/Lindsay_Lohan/Lindsay_Lohan_0001.jpg
hello, human!
```



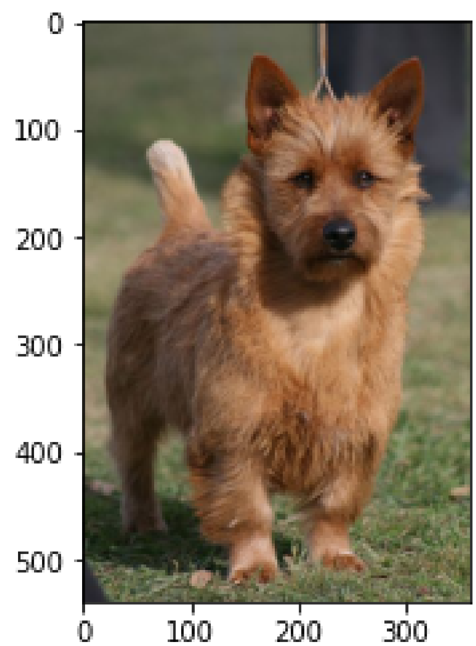
```
You look like a …
American eskimo dog
file ->
/data/dog_images/test/013.Australian_terrier/Australian_terrier_00897.jpg
hello, dog!
```



```
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 …
Australian terrier
```

```
[ ]:
```