dog_app

February 3, 2019

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

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

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

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

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

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

Step 0: Import Datasets

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

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

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

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

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

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

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

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

Step 1: Detect Humans

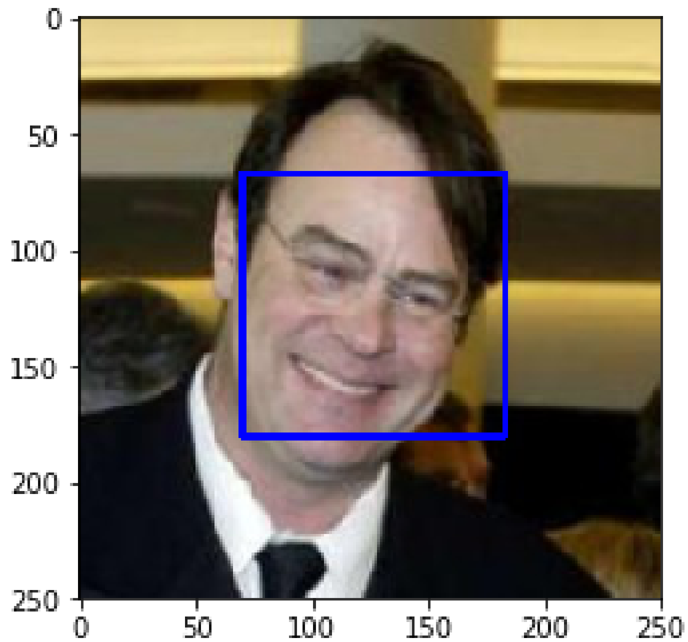
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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

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

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

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

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

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

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

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

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

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



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

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

1.1.1 Write a Human Face Detector

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

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

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

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

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

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

Answer: (You can print out your results and/or write your percentages in this cell)

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

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

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

        ## TODO: Test the performance of the face_detector algorithm
        ## on the images in human_files_short and dog_files_short.
        print(f"Human faces detected in human images: {np.sum([face_detector(fn) for fn in human
        print(f"Human faces detected in dog images: {np.sum([face_detector(fn) for fn in dog_fil
```

```
Human faces detected in human images: 98%
Human faces detected in dog images: 17%
```

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

```
In [5]: ### (Optional)
        ### TODO: Test performance of anotherface detection algorithm.
        ### Feel free to use as many code cells as needed.
```

Step 2: Detect Dogs
In this section, we use a pre-trained model to detect dogs in images.

1.1.3 Obtain Pre-trained VGG-16 Model

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

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

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

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

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

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:05<00:00, 99689972.36it/s]
```

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

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

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

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

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

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

            Args:
                img_path: path to an image

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

```
    '''

    ## TODO: Complete the function.
    ## Load and pre-process an image from the given img_path
    ## Return the *index* of the predicted class for that image

    img = cv2.imread(img_path)

    img_trans  = transforms.Compose(
        [transforms.ToPILImage(),
         transforms.Resize(256),
         transforms.CenterCrop(224),
         transforms.ToTensor()
        ])

    if use_cuda:
        img_tensor = img_trans(img).cuda()

    VGG16.eval()

    pred_class = VGG16(img_tensor.view(1,3,224,224)) # Re-size

    return pred_class.argmax().item()
```

1.1.5 (IMPLEMENTATION) Write a Dog Detector

While looking at the dictionary, you will notice that the categories corresponding to dogs appear in an uninterrupted sequence and correspond to dictionary keys 151-268, inclusive, to include all categories from `'Chihuahua'` to `'Mexican hairless'`. Thus, in order to check to see if an image is predicted to contain a dog by the pre-trained VGG-16 model, we need only check if the pre-trained model predicts an index between 151 and 268 (inclusive).

Use these ideas to complete the `dog_detector` function below, which returns `True` if a dog is detected in an image (and `False` if not).

```
In [8]: ### returns "True" if a dog is detected in the image stored at img_path
        def dog_detector(img_path):
            ## TODO: Complete the function.

            pred_idx = VGG16_predict(img_path)

            return 151 <= pred_idx <= 268
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

Question 2: Use the code cell below to test the performance of your `dog_detector` function.
- What percentage of the images in `human_files_short` have a detected dog?
- What percentage of the images in `dog_files_short` have a detected dog?

Answer:

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

        dogs_tp = np.sum([dog_detector(fn) for fn in dog_files_short])
        dogs_fn = np.sum([dog_detector(fn) for fn in human_files_short])
```

```
In [10]: print(f"{dogs_tp}% accuracy classifying dogs")
         print(f"{100-dogs_fn}% accuracy classifying humans as not dogs")
```

```
73% accuracy classifying dogs
100% accuracy classifying humans as not dogs
```

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

```
In [11]: ### (Optional)
         ### TODO: Report the performance of another pre-trained network.
         ### Feel free to use as many code cells as needed.
```

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

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

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

Brittany	Welsh Springer Spaniel

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

Curly-Coated Retriever	American Water Spaniel

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

Yellow Labrador	Chocolate Labrador

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

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

1.1.7 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

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

```
In [12]: import os
         from torchvision import datasets

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

```
In [13]: # Dataloader parameters
         BATCH_SIZE = 30
         NUM_WORKERS=0


         ### TODO: Write data loaders for training, validation, and test sets
         ## Specify appropriate transforms, and batch_sizes
         data_transform_train = transforms.Compose([transforms.RandomResizedCrop(260, scale=(0.7
                                                     transforms.RandomHorizontalFlip(p=.5),
                                                     transforms.RandomRotation(15),
                                                     transforms.CenterCrop(224),
                                                     transforms.ToTensor(),
                                                     transforms.Normalize([0.5, 0.5, 0.5], [0.5,
                                                    ])

         data_transform_test = transforms.Compose([transforms.Resize(260),
                                                    transforms.CenterCrop(224),
                                                     transforms.ToTensor(),
                                                     transforms.Normalize([0.5, 0.5, 0.5], [0.5,
                                                    ])

         data_categories = ['train', 'valid', 'test']
         loaders_scratch = {}
```

```
    for cat in data_categories:

        data = datasets.ImageFolder(f"/data/dog_images/{cat}",
                                    transform = (data_transform_train if cat == 'train' els

        print(f"{len(data)} {cat} images")

        loaders_scratch[cat] = torch.utils.data.DataLoader(data, batch_size=BATCH_SIZE,
                                              num_workers=NUM_WORKERS, shuffle=True)
```

```
6680 train images
835 valid images
836 test images
```

```
In [14]: print(iter(loaders_scratch['train']).next()[0].shape)
         iter(loaders_scratch['valid']).next()[0].shape
```

```
torch.Size([30, 3, 224, 224])
```

```
Out[14]: torch.Size([30, 3, 224, 224])
```

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: > - *RandomResizedCrop*: This randomly resizes & crops images to given size. I chose 260 to allow the transforms to have a larger canvas to work with before rotating. I also changed the default scale as I felt it went too low (no real basis other than my immature gut feel) > - *RandomHorizontalFlip*: Randomly flips image with p=0.5 - Standard way to augment > - *RandomRotation*: Up to 15 deg - Also standard > - *CenterCrop(224)*: This ultimate output size was chosen to match the VGG paper

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [15]: import torch.nn as nn
         import torch.nn.functional as F
         import torch
         import torchvision.models as models

         # define the CNN architecture
         class Net(nn.Module):
             ### TODO: choose an architecture, and complete the class
             def __init__(self, N):
                 super(Net, self).__init__()
```

```python
        # Input is an NxN image
        self.N = N

        ## Define layers of a CNN
        self.cv1 = nn.Conv2d(3, 64, 3, padding=1)
        self.bn1 = nn.BatchNorm2d(64)

        self.cv2 = nn.Conv2d(64, 128, 3, padding=1)
        self.bn2 = nn.BatchNorm2d(128)

        self.cv3 = nn.Conv2d(128, 256, 3, padding=1)
        self.bn3 = nn.BatchNorm2d(256)
        self.cv4 = nn.Conv2d(256, 256, 3, padding=1)
        self.bn4 = nn.BatchNorm2d(256)

        #self.cv5 = nn.Conv2d(256, 512, 3, padding=1)
        #self.cv6 = nn.Conv2d(512, 512, 3, padding=1)

        #self.cv7 = nn.Conv2d(512, 512, 3, padding=1)
        #self.cv8 = nn.Conv2d(512, 512, 3, padding=1)

        # Max pooling layer
        self.mp = nn.MaxPool2d(2)

        # Fully connected - N/4 * 128
        self.fc1 = nn.Linear(self.N*self.N*4, 1024)
        #self.fc2 = nn.Linear(4096, 4096)
        self.fc3 = nn.Linear(1024, 133)

        # Dropout
        self.do = nn.Dropout(0.25)


    def forward(self, x):
        ## Define forward behavior
        #print(f"Input {x.shape}")

        # N x N x 3 -> N/2 x N/2 x 64
        x = self.cv1(x)
        x = self.bn1(x)
        x = F.relu(x)
        x = self.mp(x)

        # N/2 x N/2 x 64 -> N/4x N/4 x 128
        x = self.cv2(x)
        x = self.bn2(x)
        x = F.relu(x)
        x = self.mp(x)
```

```python
        # N/4 x N/4 x 128 -> N/8 x N/8 x 256
        x = self.cv3(x)
        x = self.bn3(x)
        x = F.relu(x)
        x = self.cv4(x)
        x = self.bn4(x)
        x = F.relu(x)
        x = self.mp(x)

        ## N/8 x N/8 x 256 -> N/16 x N/16 x 512 = 2*N^2
        #x = self.cv5(x)
        #x = self.bn3(x)
        #x = F.relu(x)
        #x = self.cv6(x)
        #x = self.bn4(x)
        #x = F.relu(x)
        #x = self.mp()

        ## N/16 x N/16 x 512 = 2*N^2
        #x = self.cv7(x)
        #x = self.bn3(x)
        #x = F.relu(x)
        #x = self.cv8(x)
        #x = self.bn4(x)
        #x = F.relu(x)
        #x = self.mp()

        # Flatten
        x = x.view(-1, self.N*self.N*4)

        x = self.do(x)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.do(x)
        #x = self.fc2(x)
        #x = F.relu(x)
        #x = self.do(x)
        x = self.fc3(x)

        return x

#-#-# You do NOT have to modify the code below this line. #-#-#

# instantiate the CNN
model_scratch = Net(224)
print(model_scratch)
# move tensors to GPU if CUDA is available
```

```
    if use_cuda:
        model_scratch.cuda()
```

```
Net(
  (cv1): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (cv2): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (cv3): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (cv4): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (mp): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=200704, out_features=1024, bias=True)
  (fc3): Linear(in_features=1024, out_features=133, bias=True)
  (do): Dropout(p=0.25)
)
```

Question 4: Outline the steps you took to get to your final CNN architecture and your reasoning at each step.

Answer:

Starting point is the VGG paper's *A* architecture: https://www.kaggle.com/keras/vgg16/home

.

Following decisions/ compromises were made: 1. Excluded last 4 conv layers & middle FC layer due to memory constraints 2. BatchNorm2d layers added - Suggestion to speed up from forums 3. The model started overfitting quite quickly so added dropout layers between fully connected layers

---

1.1.9 (IMPLEMENTATION) Specify Loss Function and Optimizer

Use the next code cell to specify a loss function and optimizer. Save the chosen loss function as `criterion_scratch`, and the optimizer as `optimizer_scratch` below.

```
In [16]: import torch.optim as optim
```

```
In [17]: ### TODO: select loss function
         criterion_scratch = nn.CrossEntropyLoss()

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

1.1.10 (IMPLEMENTATION) Train and Validate the Model

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

In [18]:
```
def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
    """returns trained model"""
    # initialize tracker for minimum validation loss
    valid_loss_min = np.Inf

    for epoch in range(1, n_epochs+1):
        print(f"Epoch: {epoch}")
        # 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']):

            optimizer.zero_grad()

            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## find the loss and update the model parameters accordingly
            prediction = model(data)
            loss = criterion(prediction,target)
            # Backward pass
            loss.backward()
            # Optimizer step
            optimizer.step()
            ## record the average training loss, using something like
            train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss)

        ######################
        # validate the model #
        ######################

        print('Moving to Eval')

        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
                prediction = model(data)
                loss = criterion(prediction,target)

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

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

            ## TODO: save the model if validation loss has decreased
            if valid_loss <= valid_loss_min:

                # REWRITE:
                print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.fo

                torch.save(model.state_dict(), save_path)

                valid_loss_min = valid_loss

        # return trained model
        return model
```

In [30]:
```
%%time
# train the model
model_scratch = train(25, loaders_scratch, model_scratch, optimizer_scratch,
                      criterion_scratch, use_cuda, 'model_scratch.pt')
```

```
Epoch: 1
Moving to Eval
Epoch: 1          Training Loss: 4.738313         Validation Loss: 4.470997
Validation loss decreased (inf --> 4.470997).  Saving model ...
Epoch: 2
Moving to Eval
Epoch: 2          Training Loss: 4.371082         Validation Loss: 4.271564
Validation loss decreased (4.470997 --> 4.271564).  Saving model ...
Epoch: 3
Moving to Eval
Epoch: 3          Training Loss: 4.176295         Validation Loss: 4.159624
Validation loss decreased (4.271564 --> 4.159624).  Saving model ...
Epoch: 4
Moving to Eval
Epoch: 4          Training Loss: 4.024529         Validation Loss: 4.087957
Validation loss decreased (4.159624 --> 4.087957).  Saving model ...
Epoch: 5
```

```
Moving to Eval
Epoch: 5        Training Loss: 3.905927        Validation Loss: 4.020880
Validation loss decreased (4.087957 --> 4.020880).  Saving model ...
Epoch: 6
Moving to Eval
Epoch: 6        Training Loss: 3.792347        Validation Loss: 3.870909
Validation loss decreased (4.020880 --> 3.870909).  Saving model ...
Epoch: 7
Moving to Eval
Epoch: 7        Training Loss: 3.653920        Validation Loss: 3.969525
Epoch: 8
Moving to Eval
Epoch: 8        Training Loss: 3.593251        Validation Loss: 3.808012
Validation loss decreased (3.870909 --> 3.808012).  Saving model ...
Epoch: 9
Moving to Eval
Epoch: 9        Training Loss: 3.502166        Validation Loss: 3.935073
Epoch: 10
Moving to Eval
Epoch: 10        Training Loss: 3.418550        Validation Loss: 3.679215
Validation loss decreased (3.808012 --> 3.679215).  Saving model ...
Epoch: 11
Moving to Eval
Epoch: 11        Training Loss: 3.337316        Validation Loss: 3.660758
Validation loss decreased (3.679215 --> 3.660758).  Saving model ...
Epoch: 12
Moving to Eval
Epoch: 12        Training Loss: 3.241210        Validation Loss: 3.710969
Epoch: 13
Moving to Eval
Epoch: 13        Training Loss: 3.146957        Validation Loss: 3.505728
Validation loss decreased (3.660758 --> 3.505728).  Saving model ...
Epoch: 14
Moving to Eval
Epoch: 14        Training Loss: 3.069679        Validation Loss: 3.563920
Epoch: 15
Moving to Eval
Epoch: 15        Training Loss: 2.999975        Validation Loss: 4.005156
Epoch: 16
Moving to Eval
Epoch: 16        Training Loss: 2.916625        Validation Loss: 3.694385
Epoch: 17
Moving to Eval
Epoch: 17        Training Loss: 2.841942        Validation Loss: 3.502520
Validation loss decreased (3.505728 --> 3.502520).  Saving model ...
Epoch: 18
Moving to Eval
Epoch: 18        Training Loss: 2.750790        Validation Loss: 3.476121
```

```
Validation loss decreased (3.502520 --> 3.476121).  Saving model ...
Epoch: 19
Moving to Eval
Epoch: 19 	Training Loss: 2.685582 	Validation Loss: 3.606239
Epoch: 20
Moving to Eval
Epoch: 20 	Training Loss: 2.610869 	Validation Loss: 3.512245
Epoch: 21
Moving to Eval
Epoch: 21 	Training Loss: 2.513334 	Validation Loss: 3.617355
Epoch: 22
Moving to Eval
Epoch: 22 	Training Loss: 2.455744 	Validation Loss: 3.628527
Epoch: 23
Moving to Eval
Epoch: 23 	Training Loss: 2.363952 	Validation Loss: 3.473887
Validation loss decreased (3.476121 --> 3.473887).  Saving model ...
Epoch: 24
Moving to Eval
Epoch: 24 	Training Loss: 2.328944 	Validation Loss: 3.535422
Epoch: 25
Moving to Eval
Epoch: 25 	Training Loss: 2.214098 	Validation Loss: 3.487274
CPU times: user 58min 53s, sys: 5min 5s, total: 1h 3min 59s
Wall time: 57min 45s
```

In [19]:
```
# load the model that got the best validation accuracy
model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

1.1.11 (IMPLEMENTATION) Test the Model

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

In [20]:
```
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))
```

In [21]:
```
# call test function
test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.477499


Test Accuracy: 22% (187/836)
```

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

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

1.1.12 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

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

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

In [22]:
```
# Dataloader parameters
BATCH_SIZE = 30
NUM_WORKERS=0


### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes
data_transform_train = transforms.Compose([transforms.RandomResizedCrop(260, scale=(0.7
                                           transforms.RandomHorizontalFlip(p=.5),
                                           transforms.RandomRotation(15),
```

```
                                                    transforms.CenterCrop(224),
                                                    transforms.ToTensor(),
                                                    transforms.Normalize([0.5, 0.5, 0.5], [0.5,
                                                   ])

        data_transform_test = transforms.Compose([transforms.Resize(260),
                                                   transforms.CenterCrop(224),
                                                    transforms.ToTensor(),
                                                    transforms.Normalize([0.5, 0.5, 0.5], [0.5,
                                                   ])

        data_categories = ['train', 'valid', 'test']
        data_transfer = {}
        loaders_transfer = {}

        for cat in data_categories:

            data_transfer[cat] = datasets.ImageFolder(f"/data/dog_images/{cat}",
                                         transform = (data_transform_train if cat == 'train' els

            print(f"{len(data)} {cat} images")

            loaders_transfer[cat] = torch.utils.data.DataLoader(data_transfer[cat], batch_size=
                                               num_workers=NUM_WORKERS, shuffle=True)
```

```
836 train images
836 valid images
836 test images
```

1.1.13 (IMPLEMENTATION) Model Architecture

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

```
In [23]: import torch.nn as nn
         import torch.nn.functional as F
         import torch
         import torchvision.models as models


         model_transfer = models.vgg16(pretrained=True)

         print(model_transfer)
```

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace)
```

```
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace)
    (2): Dropout(p=0.5)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace)
    (5): Dropout(p=0.5)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

We need to replace the classifier with one out size 133

In [24]:
```
model_transfer.classifier[6] = nn.Linear(in_features=4096, out_features=133, bias=True)
```

Freeze training on features:

In [25]:
```
for param in model_transfer.features.parameters():
    param.requires_grad=False
```

```
In [26]: 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:

Steps taken: 1. Fetch pretrained model: VGG16, as we saw in a previous lesson 2. Change just the output layer to predict 133 dog breeds

1.1.14 (IMPLEMENTATION) Specify Loss Function and Optimizer

Use the next code cell to specify a loss function and optimizer. Save the chosen loss function as `criterion_transfer`, and the optimizer as `optimizer_transfer` below.

```
In [27]: criterion_transfer = nn.CrossEntropyLoss()

         # Optimise just the classifier parameters:
         optimizer_transfer = optim.SGD(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'`.

```
In [ ]: %%time

        # train the model
        model_transfer = train(50, loaders_transfer, model_transfer,
                               optimizer_transfer, criterion_transfer, use_cuda, 'model_transfer
```

```
Epoch: 1
Moving to Eval
Epoch: 1        Training Loss: 4.445627         Validation Loss: 3.628706
Validation loss decreased (inf --> 3.628706).  Saving model ...
Epoch: 2
Moving to Eval
Epoch: 2        Training Loss: 3.139152         Validation Loss: 2.176020
Validation loss decreased (3.628706 --> 2.176020).  Saving model ...
Epoch: 3
Moving to Eval
Epoch: 3        Training Loss: 2.048873         Validation Loss: 1.343856
Validation loss decreased (2.176020 --> 1.343856).  Saving model ...
Epoch: 4
Moving to Eval
Epoch: 4        Training Loss: 1.516557         Validation Loss: 0.999414
Validation loss decreased (1.343856 --> 0.999414).  Saving model ...
Epoch: 5
Moving to Eval
Epoch: 5        Training Loss: 1.252584         Validation Loss: 0.842267
```

```
Validation loss decreased (0.999414 --> 0.842267).  Saving model ...
Epoch: 6
Moving to Eval
Epoch: 6        Training Loss: 1.092260        Validation Loss: 0.756083
Validation loss decreased (0.842267 --> 0.756083).  Saving model ...
Epoch: 7
Moving to Eval
Epoch: 7        Training Loss: 0.987736        Validation Loss: 0.695668
Validation loss decreased (0.756083 --> 0.695668).  Saving model ...
Epoch: 8
Moving to Eval
Epoch: 8        Training Loss: 0.920545        Validation Loss: 0.656052
Validation loss decreased (0.695668 --> 0.656052).  Saving model ...
Epoch: 9
Moving to Eval
Epoch: 9        Training Loss: 0.844562        Validation Loss: 0.620181
Validation loss decreased (0.656052 --> 0.620181).  Saving model ...
Epoch: 10
Moving to Eval
Epoch: 10        Training Loss: 0.799041        Validation Loss: 0.599406
Validation loss decreased (0.620181 --> 0.599406).  Saving model ...
Epoch: 11
Moving to Eval
Epoch: 11        Training Loss: 0.786013        Validation Loss: 0.576641
Validation loss decreased (0.599406 --> 0.576641).  Saving model ...
Epoch: 12
Moving to Eval
Epoch: 12        Training Loss: 0.738657        Validation Loss: 0.572876
Validation loss decreased (0.576641 --> 0.572876).  Saving model ...
Epoch: 13
Moving to Eval
Epoch: 13        Training Loss: 0.706403        Validation Loss: 0.557193
Validation loss decreased (0.572876 --> 0.557193).  Saving model ...
Epoch: 14
Moving to Eval
Epoch: 14        Training Loss: 0.658957        Validation Loss: 0.547270
Validation loss decreased (0.557193 --> 0.547270).  Saving model ...
Epoch: 15
Moving to Eval
Epoch: 15        Training Loss: 0.659556        Validation Loss: 0.535066
Validation loss decreased (0.547270 --> 0.535066).  Saving model ...
Epoch: 16
Moving to Eval
Epoch: 16        Training Loss: 0.649434        Validation Loss: 0.526287
Validation loss decreased (0.535066 --> 0.526287).  Saving model ...
Epoch: 17
Moving to Eval
Epoch: 17        Training Loss: 0.620054        Validation Loss: 0.522992
```

```
Validation loss decreased (0.526287 --> 0.522992).  Saving model ...
Epoch: 18
Moving to Eval
Epoch: 18        Training Loss: 0.613136        Validation Loss: 0.513782
Validation loss decreased (0.522992 --> 0.513782).  Saving model ...
Epoch: 19
Moving to Eval
Epoch: 19        Training Loss: 0.571425        Validation Loss: 0.504678
Validation loss decreased (0.513782 --> 0.504678).  Saving model ...
Epoch: 20
Moving to Eval
Epoch: 20        Training Loss: 0.587055        Validation Loss: 0.504217
Validation loss decreased (0.504678 --> 0.504217).  Saving model ...
Epoch: 21
Moving to Eval
Epoch: 21        Training Loss: 0.562756        Validation Loss: 0.502751
Validation loss decreased (0.504217 --> 0.502751).  Saving model ...
Epoch: 22
Moving to Eval
Epoch: 22        Training Loss: 0.551554        Validation Loss: 0.497697
Validation loss decreased (0.502751 --> 0.497697).  Saving model ...
Epoch: 23
Moving to Eval
Epoch: 23        Training Loss: 0.527800        Validation Loss: 0.501619
Epoch: 24
Moving to Eval
Epoch: 24        Training Loss: 0.515090        Validation Loss: 0.484503
Validation loss decreased (0.497697 --> 0.484503).  Saving model ...
Epoch: 25
Moving to Eval
Epoch: 25        Training Loss: 0.511221        Validation Loss: 0.479480
Validation loss decreased (0.484503 --> 0.479480).  Saving model ...
Epoch: 26
Moving to Eval
Epoch: 26        Training Loss: 0.488935        Validation Loss: 0.484479
Epoch: 27
Moving to Eval
Epoch: 27        Training Loss: 0.474668        Validation Loss: 0.472875
Validation loss decreased (0.479480 --> 0.472875).  Saving model ...
Epoch: 28
Moving to Eval
Epoch: 28        Training Loss: 0.490038        Validation Loss: 0.475958
Epoch: 29
Moving to Eval
Epoch: 29        Training Loss: 0.484024        Validation Loss: 0.469292
Validation loss decreased (0.472875 --> 0.469292).  Saving model ...
Epoch: 30
Moving to Eval
```

```
Epoch: 30        Training Loss: 0.446966        Validation Loss: 0.480300
Epoch: 31
Moving to Eval
Epoch: 31        Training Loss: 0.449989        Validation Loss: 0.467925
Validation loss decreased (0.469292 --> 0.467925).  Saving model ...
Epoch: 32
Moving to Eval
Epoch: 32        Training Loss: 0.450739        Validation Loss: 0.472817
Epoch: 33
Moving to Eval
Epoch: 33        Training Loss: 0.441568        Validation Loss: 0.473826
Epoch: 34
Moving to Eval
Epoch: 34        Training Loss: 0.432885        Validation Loss: 0.468006
Epoch: 35
Moving to Eval
Epoch: 35        Training Loss: 0.417005        Validation Loss: 0.465467
Validation loss decreased (0.467925 --> 0.465467).  Saving model ...
Epoch: 36
Moving to Eval
Epoch: 36        Training Loss: 0.427821        Validation Loss: 0.467103
Epoch: 37
Moving to Eval
Epoch: 37        Training Loss: 0.420244        Validation Loss: 0.466600
Epoch: 38
Moving to Eval
Epoch: 38        Training Loss: 0.403681        Validation Loss: 0.463843
Validation loss decreased (0.465467 --> 0.463843).  Saving model ...
Epoch: 39
Moving to Eval
Epoch: 39        Training Loss: 0.406210        Validation Loss: 0.463697
Validation loss decreased (0.463843 --> 0.463697).  Saving model ...
Epoch: 40
Moving to Eval
Epoch: 40        Training Loss: 0.386933        Validation Loss: 0.473116
Epoch: 41
Moving to Eval
Epoch: 41        Training Loss: 0.393897        Validation Loss: 0.461776
Validation loss decreased (0.463697 --> 0.461776).  Saving model ...
Epoch: 42
Moving to Eval
Epoch: 42        Training Loss: 0.375833        Validation Loss: 0.454575
Validation loss decreased (0.461776 --> 0.454575).  Saving model ...
Epoch: 43
Moving to Eval
Epoch: 43        Training Loss: 0.365521        Validation Loss: 0.466558
Epoch: 44
Moving to Eval
```

1.1.16 (IMPLEMENTATION) Test the Model

Try out your model on the test dataset of dog images. Use the code cell below to calculate and print the test loss and accuracy. Ensure that your test accuracy is greater than 60%.

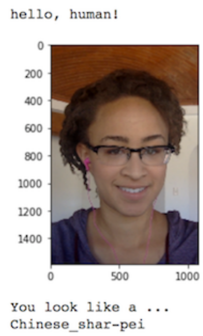
```
In [33]: # load the model that got the best validation accuracy (uncomment the line below)
         model_transfer.load_state_dict(torch.load('model_transfer.pt'))

         test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.509282


Test Accuracy: 83% (701/836)
```

1.1.17 (IMPLEMENTATION) Predict Dog Breed with the Model

Write a function that takes an image path as input and returns the dog breed (`Affenpinscher`, `Afghan hound`, etc) that is predicted by your model.

```
In [29]: ### TODO: Write a function that takes a path to an image as input
         ### and returns the dog breed that is predicted by the model.

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

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

             # Read in img using OpenCV
             img_array = cv2.imread(img_path)

             # Need to convert to PIL before using previous transforms
             img_PIL = transforms.ToPILImage()(img_array)

             img_tensor = data_transform_test(img_PIL)

             # Into required dimensions
             img_tensor = img_tensor.view(1,3,224,224)

             # To GPU
             if use_cuda:
                 img_tensor = img_tensor.cuda()


             model_transfer.eval()
```



Sample Human Output

```
    pred_class = class_names[model_transfer(img_tensor).argmax().item()]

    return pred_class
```

Step 5: Write your Algorithm

Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then, - if a **dog** is detected in the image, return the predicted breed. - if a **human** is detected in the image, return the resembling dog breed. - if **neither** is detected in the image, provide output that indicates an error.

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

Some sample output for our algorithm is provided below, but feel free to design your own user experience!

1.1.18 (IMPLEMENTATION) Write your Algorithm

In [34]:
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.

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

    dog_detected, human_detected = dog_detector(img_path), face_detector(img_path)

    if dog_detected or human_detected:
        breed = predict_breed_transfer(img_path)

        if dog_detected:
            print("Detected a dog")
            print(f"Predicted breed: {breed}")
        else:
            print("Detected a human")
            print(f"Closest dog breed resembled: {breed}")
```

```
            return breed
        else:
            print("Nothing detected")
```

Step 6: Test Your Algorithm

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

1.1.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

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

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

Answer: (Three possible points for improvement) Human & dogs are correctly separated. Only 2/5 dog breeds correctly identified in the sample below- this is worse than the expected %age from the test above.

1. Used the same normalisation of images for the dog breed detector as originally specified (means & s.d's all 0.5). The VGG model expects a different normalisation for each colour channel - this could provide an improvement.
2. In the model training, 50 epochs were allowed, but the operation timed out after 44, for which we saw an improvment in testing accraucy in the epoch previous. This suggests that the model should be run to at least 50 epochs, probably more. Could use early stopping here.
3. The algorithm currently only outputs the best match of dog. It could be altered to also produce the %age likelihood of the top 3

```
In [31]: ## 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.
```

```
In [35]: ## suggested code, below
         import random

         for file in np.hstack((random.choices(human_files, k=5), random.choices(dog_files, k=5)
             print(file)
             run_app(file)
             print(" ---- ")
```

```
/data/lfw/Garry_Trudeau/Garry_Trudeau_0002.jpg
Detected a human
Closest dog breed resembled: Dachshund
 ----
/data/lfw/Martin_Short/Martin_Short_0001.jpg
Detected a human
```

```
Closest dog breed resembled: Dachshund
 ----
/data/lfw/Penelope_Ann_Miller/Penelope_Ann_Miller_0002.jpg
Detected a human
Closest dog breed resembled: Chinese crested
 ----
/data/lfw/Teri_ORourke/Teri_ORourke_0001.jpg
Detected a human
Closest dog breed resembled: Dogue de bordeaux
 ----
/data/lfw/Norah_Jones/Norah_Jones_0015.jpg
Detected a human
Closest dog breed resembled: Chinese crested
 ----
/data/dog_images/train/028.Bluetick_coonhound/Bluetick_coonhound_01985.jpg
Detected a dog
Predicted breed: German shorthaired pointer
 ----
/data/dog_images/valid/015.Basset_hound/Basset_hound_01109.jpg
Detected a dog
Predicted breed: Plott
 ----
/data/dog_images/train/053.Cocker_spaniel/Cocker_spaniel_03777.jpg
Detected a dog
Predicted breed: Cocker spaniel
 ----
/data/dog_images/test/054.Collie/Collie_03835.jpg
Detected a dog
Predicted breed: Alaskan malamute
 ----
/data/dog_images/train/096.Labrador_retriever/Labrador_retriever_06446.jpg
Detected a dog
Predicted breed: Labrador retriever
 ----
```