dog_app

March 8, 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`.

```
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.

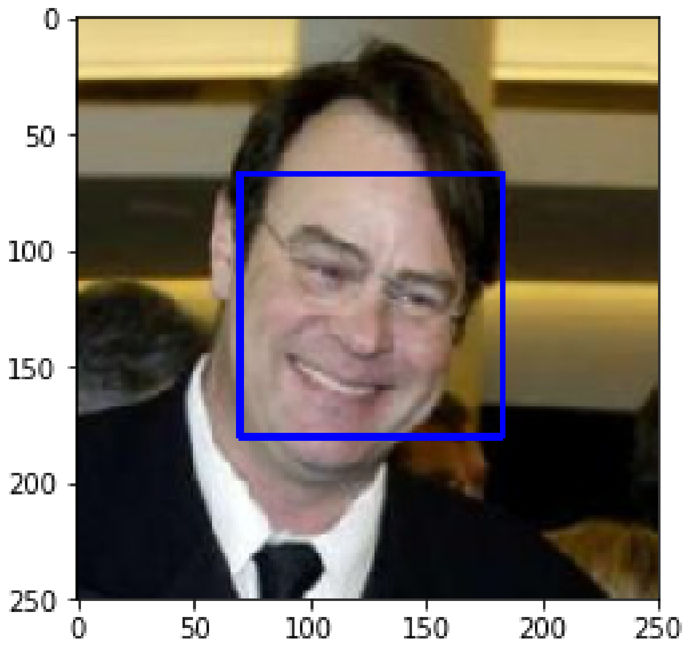
OpenCV provides many pre-trained face detectors, stored as XML files on github. We have downloaded one of these detectors and stored it in the `haarcascades` directory. In the next code cell, we demonstrate how to use this detector to find human faces in a sample image.

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

```
In [3]: # 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 [4]: # 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 [5]: 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.

        detected_human_faces_count=0
        # len(list(map(lambda x: face_detector(x), human_files_short)))
        detected_dog_faces_count=0
        # len(list(map(lambda x: face_detector(x), dog_files_short)))
        for human_face in human_files_short:
            if face_detector(human_face):
                detected_human_faces_count += 1

        for dog_face in dog_files_short:
            if face_detector(dog_face):
                detected_dog_faces_count += 1


        print('Human faces detection percentage: %d%%' % ((detected_human_faces_count/100)*100))
        print('Dog faces detection percentage: %d%%' % ((detected_dog_faces_count/100)*100))
```

```
Human faces detection percentage: 98%
Dog faces detection percentage: 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 [6]: ### (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 [26]: 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.

```
In [30]: 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
             '''

             image=Image.open(img_path)

             image_minimum_size=224
             transform=transforms.Compose([transforms.Resize(image_minimum_size),
                                           transforms.CenterCrop(size=224),
                                           transforms.ToTensor(),
                                           transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                                std=[0.229, 0.224, 0.225])])

             image=transform(image).unsqueeze(0)

             if use_cuda:
                 image=image.cuda()

             prediction=VGG16(image)
             prediction=prediction.cpu().data.numpy().argmax()
             return prediction
```

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 [31]: ### returns "True" if a dog is detected in the image stored at img_path
         def dog_detector(img_path):
             ## TODO: Complete the function.
             return VGG16_predict(img_path) in range(151, 269)
```

```
In [32]: dog_detector(dog_files[0])
```

```
Out[32]: True
```

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 [14]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.
         human_set_detected_dogs_count=0
         dogs_set_detected_dogs_count=0

         for human_face in human_files_short:
             if dog_detector(human_face):
                 human_set_detected_dogs_count += 1

         for dog_face in dog_files_short:
             if dog_detector(dog_face):
                 dogs_set_detected_dogs_count += 1

         print('Dogs detected in human faces percentage: %d%%' % ((human_set_detected_dogs_count
         print('Dogs detected in dog faces percentage: %d%%' % ((dogs_set_detected_dogs_count/le
```

```
Dogs detected in human faces percentage: 0%
Dogs detected in dog faces percentage: 100%
```

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 [ ]: ### (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 [15]: import os
         from torchvision import datasets
         import torchvision.transforms as transforms

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

         image_minimum_size=224
         standard_normalization=transforms.Normalize(
             mean=[0.485, 0.456, 0.406],
             std=[0.229, 0.224, 0.225]
         )

         transofrmers={
             'training': transforms.Compose(
                 [transforms.Resize(256),
         transforms.RandomResizedCrop(image_minimum_size),
         transforms.RandomHorizontalFlip(),
         transforms.RandomRotation(5),
         transforms.ToTensor(),
         standard_normalization]
    ),
    'validation': transforms.Compose(
        [transforms.Resize(256),
         transforms.CenterCrop(image_minimum_size),
         transforms.ToTensor(),
         standard_normalization]
    ),
    'test': transforms.Compose(
        [transforms.Resize(256),
         transforms.CenterCrop(image_minimum_size),
         transforms.ToTensor(),
         standard_normalization]
    )
}

training_dir=os.path.join('/data/dog_images/', 'train')
validation_dir=os.path.join('/data/dog_images/', 'valid')
test_dir=os.path.join('/data/dog_images/', 'test')

training_data=datasets.ImageFolder(training_dir, transform=transofrmers['training'])
validation_data=datasets.ImageFolder(validation_dir, transform=transofrmers['validation
test_data=datasets.ImageFolder(test_dir, transform=transofrmers['test'])

batch_size=16
# Using default num_workers=0

training_loader=torch.utils.data.DataLoader(
    training_data,
    batch_size=batch_size,
    shuffle=True
)
validation_loader=torch.utils.data.DataLoader(
    validation_data,
    batch_size=batch_size
)
test_loader=torch.utils.data.DataLoader(
    test_data,
    batch_size=batch_size
)

loaders_scratch={
    'train': training_loader,
    'valid': validation_loader,
            'test': test_loader
        }
```

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: Images resized to the minimum acceptable size by stretching, the smaller size is used to enhance the performance. Yes, training data was randomly flipped horizontally and rotated randomly up to 5 degrees.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [33]: import torch.nn as nn
         import torch.nn.functional as F

         classes_availbale=133
         linear_flattening_size=(7*7*128)

         # define the CNN architecture
         class Net(nn.Module):

             ### TODO: choose an architecture, and complete the class
             def __init__(self):
                 super(Net, self).__init__()
                 # First conv layer
                 self.conv1 = nn.Conv2d(3, 32, 3, stride=2, padding=1)
                 # Second conv layer
                 self.conv2 = nn.Conv2d(32, 64, 3, stride=2, padding=1)
                 # Third conv layer
                 self.conv3 = nn.Conv2d(64, 128, 3, padding=1)

                 # Max pooling between layers
                 self.pool = nn.MaxPool2d(2, 2)

                 # Drop-out
                 self.dropout = nn.Dropout(0.25)

                 # Fully connected layer
                 self.fc1 = nn.Linear(linear_flattening_size, 500)
                 self.fc2 = nn.Linear(500, classes_availbale)


             def forward(self, x):
                 x=F.relu(self.conv1(x))
                 x=self.pool(x)
```

```
                x=F.relu(self.conv2(x))
                x=self.pool(x)

                x=F.relu(self.conv3(x))
                x=self.pool(x)

                x = x.view(-1, linear_flattening_size)

                x = self.dropout(x)
                x = F.relu(self.fc1(x))

                x = self.dropout(x)
                x = self.fc2(x)

                return x

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

        # instantiate the CNN
        model_scratch = Net()
        print(model_scratch)

        # move tensors to GPU if CUDA is available
        if use_cuda:
            model_scratch.cuda()
```

```
Net(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (dropout): Dropout(p=0.25)
  (fc1): Linear(in_features=6272, out_features=500, bias=True)
  (fc2): Linear(in_features=500, out_features=133, bias=True)
)
```

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

Answer:

1- Applying a convolutional layer with three- 3*3 kernals and summed up, with a stride of 2, this convolutional layer will extract features while downsizing the image by 2

2- For simplicity, a max pooling is applied to downsize the image by 2 again, bringing down the number of features we have to deal with here

3- Steps 1&2 are repteated two more times

4- Finally the output from the last pooling passes two fully connected layers, one with 500 layers and the other with 133 layers predicting the actual dog breed, each FC layer is preceded by a dropout with probability of 0.25 to avoid overfitting

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 [34]: import torch.optim as optim

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

         ### TODO: select optimizer
         optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=0.05)
```

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 [35]: import numpy as np
         from PIL import ImageFile

         ImageFile.LOAD_TRUNCATED_IMAGES = True

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

             for epoch in range(1, n_epochs+1):
                 # initialize variables to monitor training and validation loss
                 train_loss = 0.0
                 valid_loss = 0.0

                 ###################
                 # train the model #
                 ###################
                 model.train()
                 for batch_idx, (data, target) in enumerate(loaders['train']):
                     # move to GPU
                     if use_cuda:
                         data, target = data.cuda(), target.cuda()
                     ## find the loss and update the model parameters accordingly
                     ## record the average training loss, using something like
                     ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_lo
                     optimizer.zero_grad()
                     prediction = model(data)
                     loss = criterion(prediction, target)
                     loss.backward()
                     optimizer.step()
```

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


            ######################
            # validate the model #
            ######################
            model.eval()
            for batch_idx, (data, target) in enumerate(loaders['valid']):
                # move to GPU
                if use_cuda:
                    data, target = data.cuda(), target.cuda()
                ## update the average validation loss
                predection = model(data)
                loss = criterion(predection, target)
                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:
                print("Valid loss dropped below minimum, saving model..")
                torch.save(model.state_dict(), save_path)
                valid_loss_min = valid_loss

        # return trained model
        return model
```

```
In [41]: # train the model
         model_scratch = train(20, loaders_scratch, model_scratch, optimizer_scratch,
                               criterion_scratch, use_cuda, 'model_scratch.pt')
```

```
Epoch: 1         Training Loss: 4.103206         Validation Loss: 3.934501
Valid loss dropped below minimum, saving model..
Epoch: 2         Training Loss: 4.055869         Validation Loss: 4.274770
Epoch: 3         Training Loss: 4.005448         Validation Loss: 3.851963
Valid loss dropped below minimum, saving model..
Epoch: 4         Training Loss: 3.959488         Validation Loss: 4.026988
Epoch: 5         Training Loss: 3.908271         Validation Loss: 3.886464
Epoch: 6         Training Loss: 3.868055         Validation Loss: 3.755533
Valid loss dropped below minimum, saving model..
Epoch: 7         Training Loss: 3.814463         Validation Loss: 3.981435
Epoch: 8         Training Loss: 3.774944         Validation Loss: 3.714422
```

```
Valid loss dropped below minimum, saving model..
Epoch: 9         Training Loss: 3.720410         Validation Loss: 3.852491
Epoch: 10         Training Loss: 3.698553         Validation Loss: 3.546863
Valid loss dropped below minimum, saving model..
Epoch: 11         Training Loss: 3.658459         Validation Loss: 3.636077
Epoch: 12         Training Loss: 3.592643         Validation Loss: 3.533298
Valid loss dropped below minimum, saving model..
Epoch: 13         Training Loss: 3.571358         Validation Loss: 3.523408
Valid loss dropped below minimum, saving model..
Epoch: 14         Training Loss: 3.564193         Validation Loss: 3.612917
Epoch: 15         Training Loss: 3.531997         Validation Loss: 3.668572
Epoch: 16         Training Loss: 3.462643         Validation Loss: 3.616429
Epoch: 17         Training Loss: 3.475028         Validation Loss: 3.421948
Valid loss dropped below minimum, saving model..
Epoch: 18         Training Loss: 3.445777         Validation Loss: 3.372897
Valid loss dropped below minimum, saving model..
Epoch: 19         Training Loss: 3.392254         Validation Loss: 3.530689
Epoch: 20         Training Loss: 3.369791         Validation Loss: 3.668778
```

```
In [42]: # 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 [43]: 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: 3.395980


Test Accuracy: 19% (160/836)
```

## 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 [44]: ## TODO: Specify data loaders
         transfer_loaders_scratch=loaders_scratch.copy()
```

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

         ## TODO: Specify model architecture
         model_transfer = models.resnet50(pretrained=True)

         # Freezing params
         for param in model_transfer.parameters():
             param.requires_grad = False
```

```
        # Replacing last FC layer
        model_transfer.fc = nn.Linear(2048, 1000)
        model_transfer.drop = nn.Dropout(0.25)
        model_transfer.fc1 = nn.Linear(1000, 133)

        if use_cuda:
            model_transfer = model_transfer.cuda()
```

```
Downloading: "https://download.pytorch.org/models/resnet50-19c8e357.pth" to /root/.torch/models/
100%|| 102502400/102502400 [00:01<00:00, 99852245.88it/s]
```

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:

ResNet-50 is a convolutional neural network that is reputable for its performance in image classification. That's why I chose it to form this CNN with transfer learning.

Of course we don't need to retrain the older convolutional weights, that's why we freezed them here. Since the data we're working with in the new model is similar to the data the older model was traindend on, we only need to replace the last fully connected layer. To make it interesting, I've replaced the last FC layer with two FC layers, and a dropout of 25% in between.

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 [46]: criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.SGD(model_transfer.fc.parameters(), lr=0.05)
```

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 [48]: # train the model
         model_transfer = train(20, transfer_loaders_scratch, model_transfer, optimizer_transfer
```

```
Epoch: 1         Training Loss: 0.943981         Validation Loss: 0.422044
Valid loss dropped below minimum, saving model..
Epoch: 2         Training Loss: 0.972851         Validation Loss: 0.447643
Epoch: 3         Training Loss: 0.927173         Validation Loss: 0.449082
Epoch: 4         Training Loss: 0.918243         Validation Loss: 0.422637
Epoch: 5         Training Loss: 0.876477         Validation Loss: 0.447036
Epoch: 6         Training Loss: 0.901954         Validation Loss: 0.426184
Epoch: 7         Training Loss: 0.908228         Validation Loss: 0.478142
Epoch: 8         Training Loss: 0.879847         Validation Loss: 0.436527
Epoch: 9         Training Loss: 0.873932         Validation Loss: 0.441418
```

```
Epoch: 10         Training Loss: 0.881931        Validation Loss: 0.449376
Epoch: 11         Training Loss: 0.870366        Validation Loss: 0.460229
Epoch: 12         Training Loss: 0.900059        Validation Loss: 0.457874
Epoch: 13         Training Loss: 0.839386        Validation Loss: 0.438158
Epoch: 14         Training Loss: 0.838383        Validation Loss: 0.461921
Epoch: 15         Training Loss: 0.831762        Validation Loss: 0.422775
Epoch: 16         Training Loss: 0.847627        Validation Loss: 0.420039
Valid loss dropped below minimum, saving model..
Epoch: 17         Training Loss: 0.834610        Validation Loss: 0.493184
Epoch: 18         Training Loss: 0.793410        Validation Loss: 0.424482
Epoch: 19         Training Loss: 0.845468        Validation Loss: 0.412485
Valid loss dropped below minimum, saving model..
Epoch: 20         Training Loss: 0.801542        Validation Loss: 0.456226
```

```
In [49]: # load the model that got the best validation accuracy (uncomment the line below)
         model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

1.1.16 (IMPLEMENTATION) Test the Model

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

```
In [50]: test(transfer_loaders_scratch, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.480578


Test Accuracy: 87% (731/836)
```

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 [51]: def prepare_image(image):
             composed_transforms = transforms.Compose(
                 [transforms.Resize(256),
                  transforms.CenterCrop(224),
                  transforms.ToTensor(),
                  transforms.Normalize(
                      mean=[0.485, 0.456, 0.406],
                      std=[0.229, 0.224, 0.225]
                  )]
             )

             return composed_transforms(image).unsqueeze(0)
```

```
In [54]: ### 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 transfer_loaders_scratch['train']
         device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

         def predict_breed_transfer(img_path):
             # load the image and return the predicted breed
             image=Image.open(img_path)

             image=prepare_image(image)

             if use_cuda:
                 image=image.cuda()

             model_transfer.eval()
             predection=model_transfer(image)
             class_index=torch.argmax(predection.cpu())

             return class_names[class_index]

In [55]: # Testing predict_breed_transfer
         import random
         image_name=random.choice(os.listdir('./images'))
         image_path=os.path.join('./images', image_name)
         predicted_class=predict_breed_transfer(image_path)
         print("image_file_name: " + image_path)
         print("predition breed: " + predicted_class)

image_file_name: ./images/Brittany_02625.jpg
predition breed: Brittany


In [ ]:
```

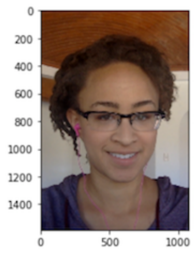
## Step 5: Write your Algorithm

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
```

Sample Human Output

1.1.18 (IMPLEMENTATION) Write your Algorithm

```
In [56]: ### 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
             image = Image.open(img_path)

             predicted_breed = predict_breed_transfer(img_path)
             plt.imshow(image)
             plt.show()

             if dog_detector(img_path):
                 print("Dog detected with a predicted breed: " + predicted_breed)
             elif face_detector(img_path):
                 print("Hoooman detected with the most similar dog breed being: " + predicted_br
             else:
                 print("No face detected. Try smiling more :)")
```

## 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!

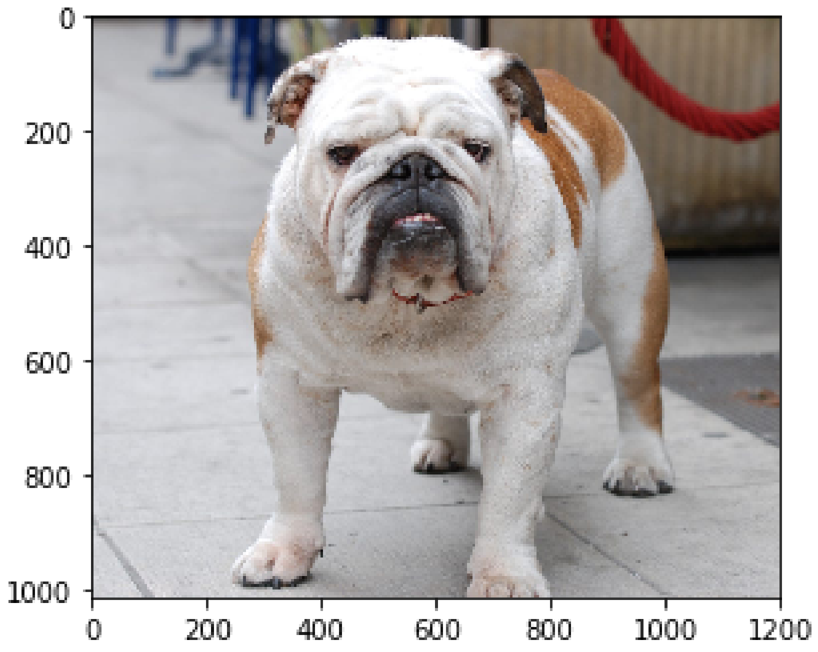
Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

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

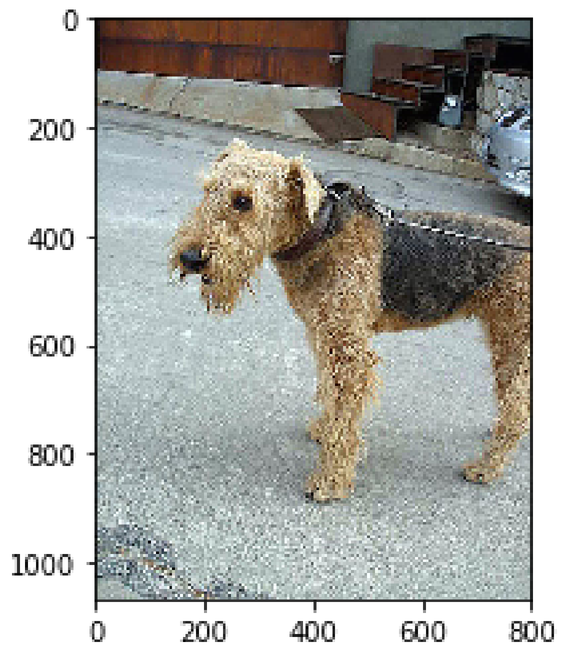
Answer: (Three possible points for improvement)

Output is similar to what's expected. Areas of improvement can be: - Adding more dog images for other dog breeds (current model only recognizes 133, less than half of the known dog breeds currently) - Improving the result of when a human detected by predecting multiple similar dog breeds (maybe showing sample pictures of those breeds) - Improving the performance and accuracy of the breed classifier model by adding more epochs or using a different model maybe, with hyperparameters tuning - Detecting multiple faces in an image (or maybe a human and a dog in the same image and provide similarity score)

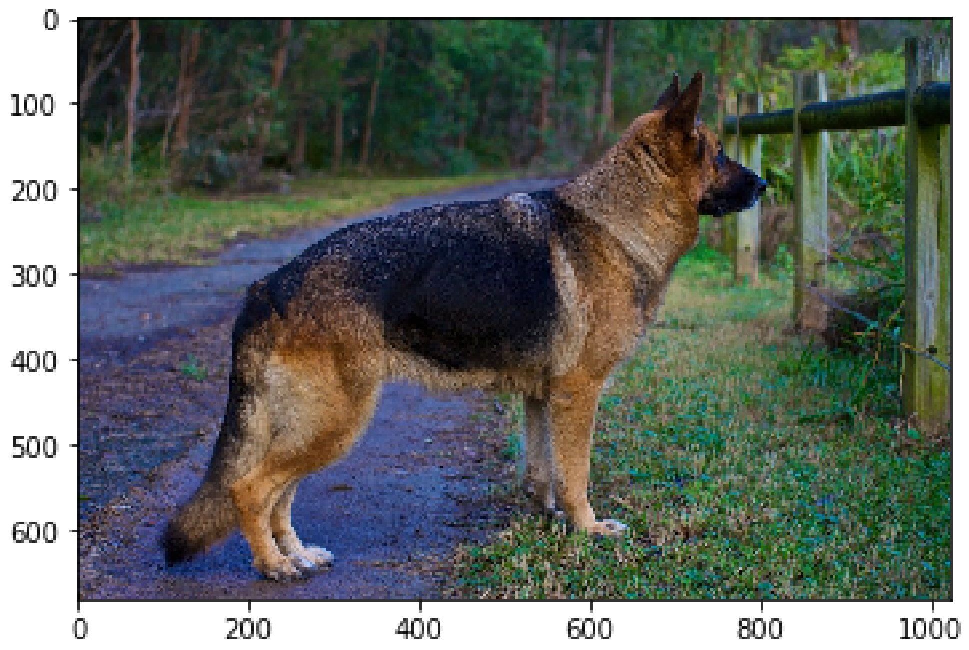
```
In [72]: ## TODO: Execute your algorithm from Step 6 on
         ## at least 6 images on your computer.
         ## Feel free to use as many code cells as needed.

         ## suggested code, below

         ## Reading from 'my_images'
         my_images=os.listdir('./my_images')
         for file in my_images:
             image_path=os.path.join('./my_images', file)
             run_app(image_path)
```



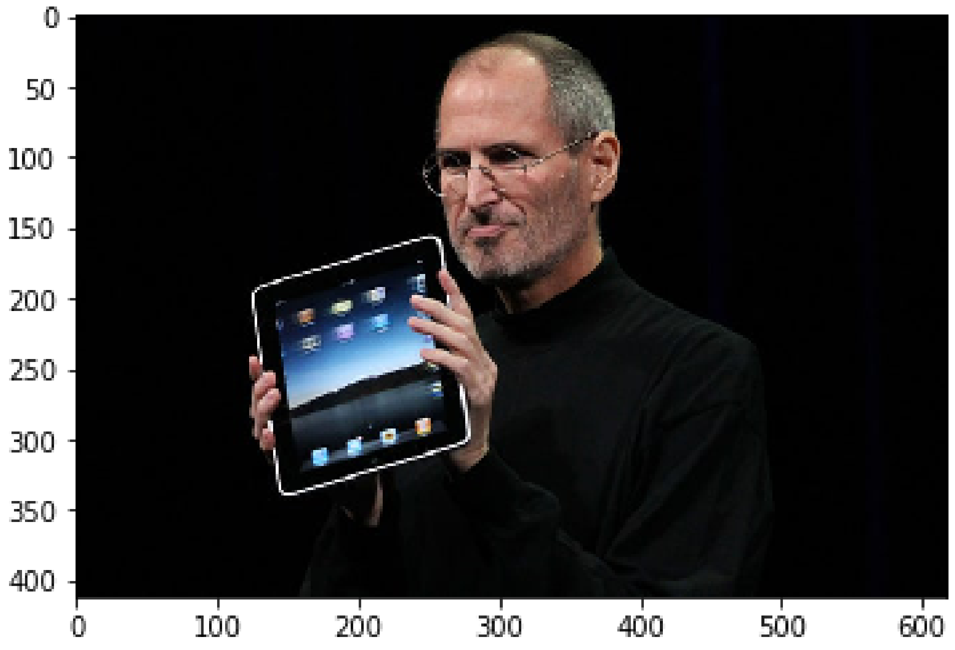
```
Dog detected with a predicted breed: Bulldog
```



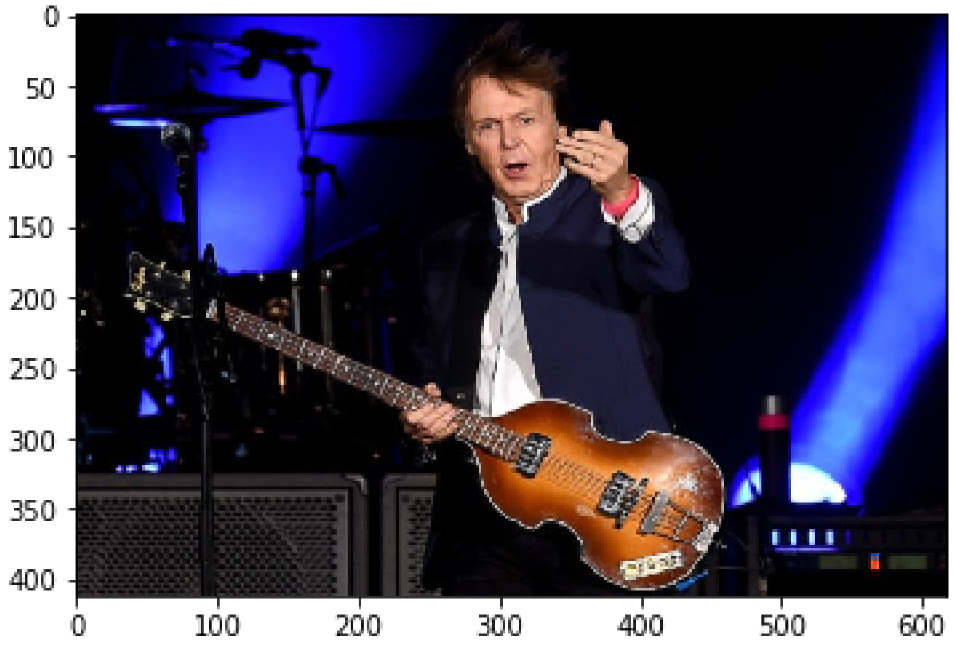
```
Dog detected with a predicted breed: Airedale terrier
```



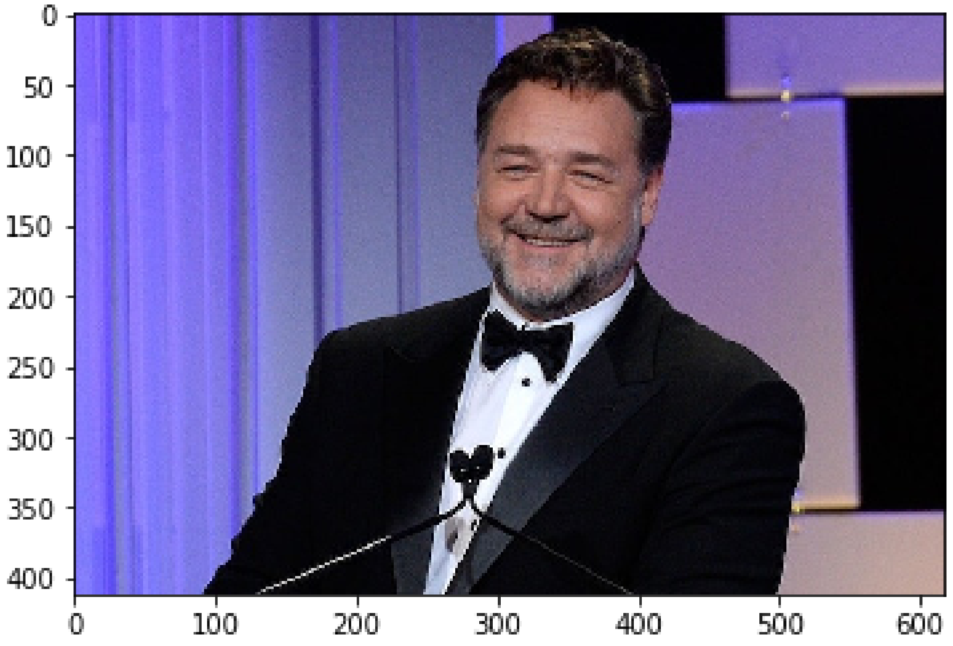
```
Dog detected with a predicted breed: German shepherd dog
```



```
Hoooman detected with the most similar dog breed being: Chihuahua
```



```
Hoooman detected with the most similar dog breed being: Cocker spaniel
```



```
Hoooman detected with the most similar dog breed being: American foxhound
```

```
In [ ]:
```