

dog_app

April 9, 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

        # 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]: def check_face_detector_files(func_detector,files):
    detected_face_count = 0

    for file in files:
        if func_detector(file): detected_face_count += 1

    return detected_face_count
```

```
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.
```

```
print("Checking files...")
```

```
human_files_detected_face_count = check_face_detector_files(face_detector,human_files_short)
dog_files_detected_face_count = check_face_detector_files(face_detector,dog_files_short)
```

```
print (f"Face detected in { human_files_detected_face_count }% of the first 100 images in human_files_short")
print (f"Face detected in { dog_files_detected_face_count }% of the first 100 images in dog_files_short")
```

Checking files...

Face detected in 98% of the first 100 images in human_files.

Face detected in 17% of the first 100 images in 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.

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

Step 2: Detect Dogs

In this section, we use a [pre-trained model](#) to detect dogs in images.

1.1.3 Obtain Pre-trained VGG-16 Model

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

```
In [6]: import torch
        import torchvision.models as models
        import torchvision.transforms as transforms
        from PIL import Image, ImageFile

        normalizer = transforms.Normalize( mean = [0.485, 0.456, 0.406] , std = [0.229, 0.224, 0.225] )
        use_cuda = torch.cuda.is_available()
        device = torch.device("cuda" if use_cuda else "cpu")
        ImageFile.LOAD_TRUNCATED_IMAGES = True

In [7]: # define VGG16 model
        VGG16 = models.vgg16(pretrained=True)

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

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg16-397923af.pth
100%|| 553433881/553433881 [00:11<00:00, 47386749.92it/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 [8]: def load_image(img_path):
        image = Image.open(img_path)
        image_transform = transforms.Compose([
            transforms.Resize(224),
            transforms.CenterCrop(224),
            transforms.ToTensor(),
            normalizer])
        image_tensor = image_transform(image).unsqueeze(0)
        result = image_tensor.to(device)
        return result

In [9]: 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

        image = load_image(img_path)

        with torch.no_grad():
            vgg16_model_output = VGG16(image)

        out, index = vgg16_model_output.max(1)
        result = index.item()

        return result
```

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

    prediction = VGG16_predict(img_path)
    prediction_result = bool(prediction >= 151 and prediction <= 268)

    return prediction_result
```

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

print("Checking files...")

human_files_detected_face_count = check_face_detector_files(dog_detector, human_files_short)
dog_files_detected_face_count = check_face_detector_files(dog_detector, dog_files_short)

print(f"Face detected in { human_files_detected_face_count }% of the first 100 images")
print(f"Face detected in { dog_files_detected_face_count }% of the first 100 images in dog_files")
```

Checking files...

Face detected in 1% of the first 100 images in human_files.

Face detected in 100% of the first 100 images in 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`.

```
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 | Black Labrador

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

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

1.1.7 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

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

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

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

         num_workers = 4
         batch_size = 8
         data_root = '/data/dog_images/'

         train_dataloader = torch.utils.data.DataLoader(
             datasets.ImageFolder(
                 os.path.join(data_root, 'train/'),
                 transform = transforms.Compose(
                     [transforms.RandomResizedCrop(224),
```



```

        transforms.RandomHorizontalFlip(),
        transforms.RandomRotation(10),
        transforms.ToTensor(),
        normalizer])
    ),
    batch_size = batch_size,
    num_workers = num_workers,
    shuffle = True
)

validation_dataloader = torch.utils.data.DataLoader(
    datasets.ImageFolder(
        os.path.join(data_root, 'valid/'),
        transform = transforms.Compose(
            [transforms.Resize(224),
             transforms.CenterCrop(224),
             transforms.ToTensor(),
             normalizer]
        )
    ),
    batch_size = batch_size,
    num_workers = num_workers,
    shuffle = True
)

test_dataloader = torch.utils.data.DataLoader(
    datasets.ImageFolder(
        os.path.join(data_root, 'test/'),
        transform = transforms.Compose(
            [transforms.Resize(224),
             transforms.CenterCrop(224),
             transforms.ToTensor(),
             normalizer]
        )
    ),
    batch_size = batch_size,
    num_workers = num_workers,
    shuffle = True
)

loaders_scratch = {
    'train' : train_dataloader,
    'valid': validation_dataloader,
    'test': test_dataloader
}

```

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: I cropped the training images to random size and aspect ratio, and the images resized to 224x224. I also horizontally flipped the training images with the default probability of 0.5. I resized the validation and test images to 224x224 and cropped the image at the center to 224x224, if image size is smaller than 224x224 along any edge, image is padded with 0 and then center cropped. We resized the images to 224x224 since we are dealing with different image sizes. I augmented the dataset using the RandomResizedCrop method to crop the given image randomly, and the RandomHorizontalFlip method to horizontally flip the given image randomly, and the RandomRotation method to rotate the given image by angle.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [13]: 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

        # Apply a 2D convolutional layer, taking in 3 input channel
        # outputting 16 convolutional features, with a square kernel size of 3
        self.conv1 = nn.Conv2d(3, 16, kernel_size = 3, stride = 1, padding = 1)

        # Apply a 2D convolutional layer, taking in 16 input channel
        # outputting 32 convolutional features, with a square kernel size of 3
        self.conv2 = nn.Conv2d(16, 32, kernel_size = 3, stride = 1, padding = 1)

        # Apply a 2D convolutional layer, taking in 32 input channel
        # outputting 64 convolutional features, with a square kernel size of 3
        self.conv3 = nn.Conv2d(32, 64, kernel_size = 3, stride = 1, padding = 1)

        # Apply zero out entire channels to ensure that adjacent pixels are either
        # all 0s or all active with an input probability
        self.dropout = nn.Dropout2d(0.25)

        # Add first fully connected layer that outputs 512 labels
        self.fc1 = nn.Linear(64 * 28 * 28, 512)

        # Add second fully connected layer that outputs our 133 labels
        self.fc2 = nn.Linear(512, 133)
```

```

        # Apply batch normalization to normalize the activations of the network between
        self.batch_normalization1 = nn.BatchNorm2d(16)
        self.batch_normalization2 = nn.BatchNorm2d(32)
        self.batch_normalization3 = nn.BatchNorm2d(64)
        self.batch_normalization4 = nn.BatchNorm1d(512)

    def forward(self, x):
        ## Define forward behavior

        x = F.max_pool2d(F.relu(self.batch_normalization1(self.conv1(x))), (2, 2))
        x = F.max_pool2d(F.relu(self.batch_normalization2(self.conv2(x))), (2, 2))
        x = F.max_pool2d(F.relu(self.batch_normalization3(self.conv3(x))), (2, 2))

        x = x.view(-1, 64 * 28 * 28)

        x = F.relu(self.batch_normalization4(self.fc1(self.dropout(x))))
        x = self.fc2(self.dropout(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, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (dropout): Dropout2d(p=0.25)
  (fc1): Linear(in_features=50176, out_features=512, bias=True)
  (fc2): Linear(in_features=512, out_features=133, bias=True)
  (batch_normalization1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (batch_normalization2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (batch_normalization3): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (batch_normalization4): BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
)

```

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

Answer:

- Firstly, I added a 2D convolutional layer, taking in 3 input channel outputting 16 convolutional features, with a square kernel size of 3, stride of 1 and padding of 1.
- Then I added a 2D convolutional layer, taking in 16 input channel outputting 32 convolutional features, with a square kernel size of 3, stride of 1 and padding of 1.
- Then I added a 2D convolutional layer, taking in 32 input channel outputting 64 convolutional features, with a square kernel size of 3, stride of 2 and padding of 1.
- After adding three convolutional layers, I applied a 2D max pooling over convolutional layers to reduce dimensions. I also applied the RELU function to make the network non-linear.
- I added two fully connected layers that outputs 512 and 133 labels. I applied a 2D dropout to promote independence between feature maps.
- I added 2D batch normalization to convolutional layers and 1D batch normalization to the first fully connected layer to normalize the activations of the network between 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 [14]: from torch.optim import SGD

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

        ### TODO: select optimizer
        optimizer_scratch = SGD(model_scratch.parameters(), lr = 0.03, momentum = 0.9)
```

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 [15]: 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_loss))

    optimizer.zero_grad()
    output = model(data)
    loss = criterion(output, 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

        output = model(data)
        loss = criterion(output, 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:
        torch.save(model.state_dict(), save_path)
        valid_loss_min = valid_loss

    # return trained model
    return model

# train the model
model_scratch = train(30, loaders_scratch, model_scratch, optimizer_scratch, criterion_scratch)

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

```

Epoch: 1	Training Loss: 4.806272	Validation Loss: 4.598965
Epoch: 2	Training Loss: 4.620492	Validation Loss: 4.442835
Epoch: 3	Training Loss: 4.564020	Validation Loss: 4.360752
Epoch: 4	Training Loss: 4.523397	Validation Loss: 4.359308
Epoch: 5	Training Loss: 4.498525	Validation Loss: 4.225847
Epoch: 6	Training Loss: 4.468140	Validation Loss: 4.239058
Epoch: 7	Training Loss: 4.435859	Validation Loss: 4.227190
Epoch: 8	Training Loss: 4.429532	Validation Loss: 4.175518
Epoch: 9	Training Loss: 4.400808	Validation Loss: 4.135453
Epoch: 10	Training Loss: 4.394260	Validation Loss: 4.210156
Epoch: 11	Training Loss: 4.349110	Validation Loss: 4.074286
Epoch: 12	Training Loss: 4.321904	Validation Loss: 4.060047
Epoch: 14	Training Loss: 4.294847	Validation Loss: 4.129696
Epoch: 15	Training Loss: 4.290298	Validation Loss: 3.995526
Epoch: 16	Training Loss: 4.264571	Validation Loss: 3.994423
Epoch: 17	Training Loss: 4.231234	Validation Loss: 4.032627
Epoch: 18	Training Loss: 4.232930	Validation Loss: 3.901232
Epoch: 19	Training Loss: 4.202550	Validation Loss: 3.905613
Epoch: 20	Training Loss: 4.225564	Validation Loss: 3.844921
Epoch: 21	Training Loss: 4.172461	Validation Loss: 3.951441
Epoch: 22	Training Loss: 4.187317	Validation Loss: 3.877223
Epoch: 23	Training Loss: 4.161955	Validation Loss: 3.760353
Epoch: 24	Training Loss: 4.147604	Validation Loss: 3.824188
Epoch: 25	Training Loss: 4.127481	Validation Loss: 3.768083
Epoch: 26	Training Loss: 4.121225	Validation Loss: 3.719263
Epoch: 27	Training Loss: 4.128139	Validation Loss: 3.754927
Epoch: 28	Training Loss: 4.094692	Validation Loss: 3.682965
Epoch: 29	Training Loss: 4.115124	Validation Loss: 3.714743
Epoch: 30	Training Loss: 4.093284	Validation Loss: 3.677129

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 [16]: def test(loaders, model, criterion, use_cuda):

    # monitor test loss and accuracy
    test_loss = 0.
    correct = 0.
    total = 0.

    model.eval()
    for batch_idx, (data, target) in enumerate(loaders['test']):
        # move to GPU
        if use_cuda:
            data, target = data.cuda(), target.cuda()
```

```

# forward pass: compute predicted outputs by passing inputs to the model
output = model(data)
# calculate the loss
loss = criterion(output, target)
# update average test loss
test_loss = test_loss + ((1 / (batch_idx + 1)) * (loss.data - test_loss))
# convert output probabilities to predicted class
pred = output.data.max(1, keepdim=True)[1]
# compare predictions to true label
correct += np.sum(np.squeeze(pred.eq(target.data.view_as(pred))).cpu().numpy())
total += data.size(0)

print('Test Loss: {:.6f}\n'.format(test_loss))
print('\nTest Accuracy: %2d%% (%2d/%2d)' % (100. * correct / total, correct, total))

```

```
In [17]: # call test function
```

```
test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

Test Loss: 3.669476

Test Accuracy: 13% (112/836)

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 [18]: ## TODO: Specify data loaders
```

```
loaders_transfer = loaders_scratch
```

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 [19]: import torchvision.models as models
import torch.nn as nn
```

```

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

for param in model_transfer.parameters():
    param.requires_grad = False

model_transfer.fc = nn.Sequential(nn.Linear(2048, 133))
model_transfer = model_transfer.to(device)
print(model_transfer)

```

Downloading: "https://download.pytorch.org/models/resnet50-19c8e357.pth" to /root/.torch/models/100%|| 102502400/102502400 [00:00<00:00, 105446750.64it/s]

```

ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (downsample): Sequential(
        (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
  )
  (1): Bottleneck(
    (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
  (2): Bottleneck(
    (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  )
)

```



```

        (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace)
    )
)
(layer2): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
    (downsample): Sequential(
      (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
)
  (1): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
)
  (2): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
)
  (3): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
)
)
(layer3): Sequential(
  (0): Bottleneck(

```

```

(conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
(bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
(bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
(bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(relu): ReLU(inplace)
(downsample): Sequential(
  (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False)
  (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
)
)
(1): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace)
)
(2): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace)
)
(3): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace)
)
(4): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace)
)
)

```

```

(5): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace)
)
)
(layer4): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
    (downsample): Sequential(
      (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): Bottleneck(
    (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
  (2): Bottleneck(
    (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
)
(avgpool): AvgPool2d(kernel_size=7, stride=1, padding=0)
(fc): Sequential(
  (0): Linear(in_features=2048, out_features=133, bias=True)
)
)

```

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 preferred using Residual Networks which is a popular neural network used as a backbone for many computer vision tasks. The fundamental breakthrough with ResNet was it allowed us to train extremely deep neural networks with 150+ layers successfully. I used ResNet50 which is mentioned before as an alternative pre-trained network to VGG-16, ResNet50 is a convolutional neural network that is 50 layers deep.

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 [20]: criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = SGD(model_transfer.fc.parameters(), lr = 0.03, momentum = 0.9)
```

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 [21]: # train the model
         # train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer, criterion_transfer)
         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_loss))

                     optimizer.zero_grad()
                     output = model(data)
```

```

        loss = criterion(output, 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

    output = model(data)
    loss = criterion(output, 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:
    torch.save(model.state_dict(), save_path)
    valid_loss_min = valid_loss

# return trained model
return model

# train the model
model_transfer = train(10, loaders_transfer, model_transfer, optimizer_transfer, criterion_transfer)

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

```

Epoch: 1	Training Loss: 8.218162	Validation Loss: 4.107317
Epoch: 2	Training Loss: 7.613813	Validation Loss: 4.166151
Epoch: 3	Training Loss: 6.832040	Validation Loss: 3.577013
Epoch: 4	Training Loss: 6.979835	Validation Loss: 3.820831
Epoch: 5	Training Loss: 7.007840	Validation Loss: 3.319883
Epoch: 6	Training Loss: 6.788256	Validation Loss: 3.490227
Epoch: 7	Training Loss: 6.746976	Validation Loss: 3.529666

Epoch: 8	Training Loss: 6.589819	Validation Loss: 3.486921
Epoch: 9	Training Loss: 6.615377	Validation Loss: 3.327050
Epoch: 10	Training Loss: 6.207893	Validation Loss: 3.790337

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

```
Test Loss: 3.301214
```

```
Test Accuracy: 75% (629/836)
```

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 [23]: ### 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 loaders_scratch['train'].dataset.  
  
def predict_breed_transfer(img_path):  
    # load the image and return the predicted breed  
  
    image = load_image(img_path)  
  
    with torch.no_grad():  
        model_transfer_output = model_transfer(image)  
  
        out, index = model_transfer_output.max(1)  
        result = index.item()  
  
    return class_names[result]
```

```
In [24]: import matplotlib.image as mpimg
```

```
test_image = human_files_short[0]  
breed_transfer_result = predict_breed_transfer(test_image)  
  
print(f'Image Path: {test_image}')  
print(f'Result: {breed_transfer_result}')
```

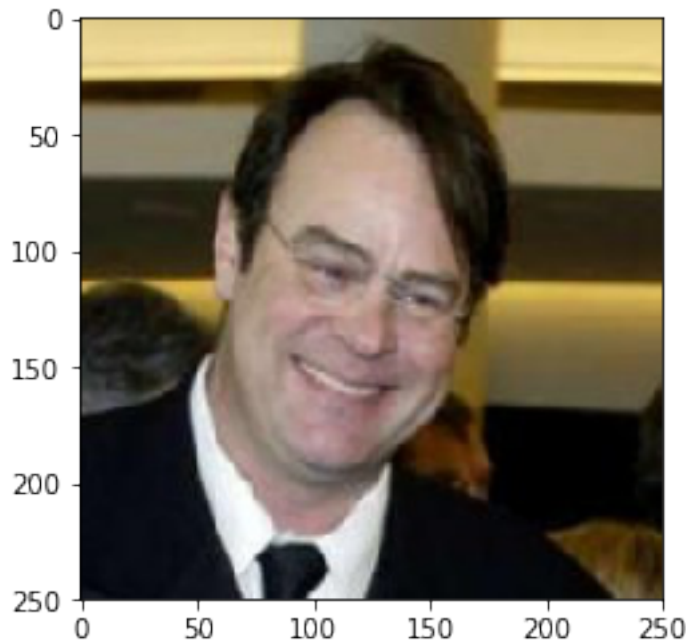
```

image = mpimg.imread(test_image)
imgplot = plt.imshow(image)
plt.show()

```

Image Path: /data/lfw/Dan_Ackroyd/Dan_Ackroyd_0001.jpg

Result: Beagle



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.

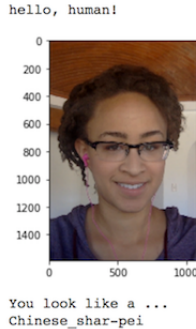
Some sample output for our algorithm is provided below, but feel free to design your own user experience!

1.1.18 (IMPLEMENTATION) Write your Algorithm

```

In [25]: ### TODO: Write your algorithm.
         ### Feel free to use as many code cells as needed.

```



Sample Human Output

```
def run_app(img_path):
    ## handle cases for a human face, dog, and neither
    result = 'It seems the image does not contain a human or dog face.'
    if dog_detector(img_path):
        breed_transfer_result = predict_breed_transfer(img_path)
        result = f'Dog prediction: {breed_transfer_result}'
    elif face_detector(img_path):
        breed_transfer_result = predict_breed_transfer(img_path)
        result = f'Human prediction: {breed_transfer_result}'

    print(result)
    image = mpimg.imread(img_path)
    imgplot = plt.imshow(image)
    plt.show()
```

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.

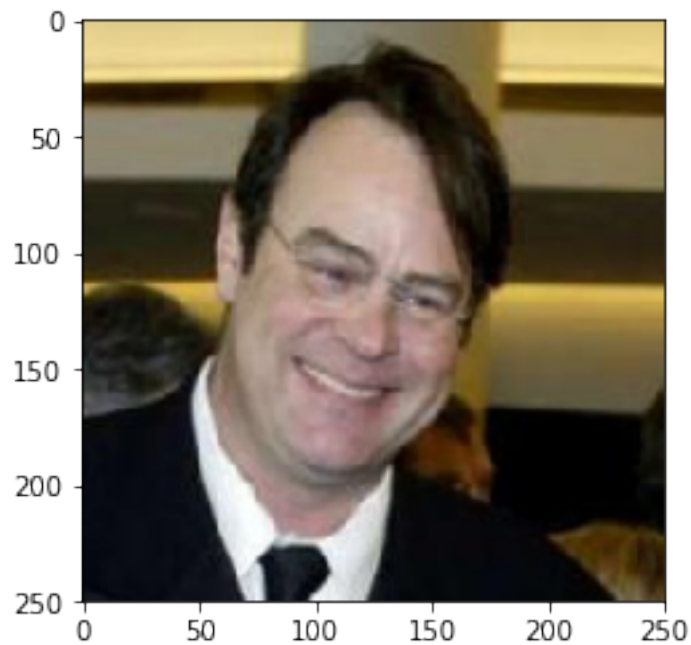
Answer:

The output is better than I expected. I can improve the performance of our CNN model applying the changes below: * I can tune parameters such as number of epochs, learning rate, optimizer parameters etc. * I can try to identify overfitting or underfitting issues. Training with more data, cross validation, early stopping and adding more layers can help us to find a good balance. * I can apply more preprocessing steps like data augmentation.

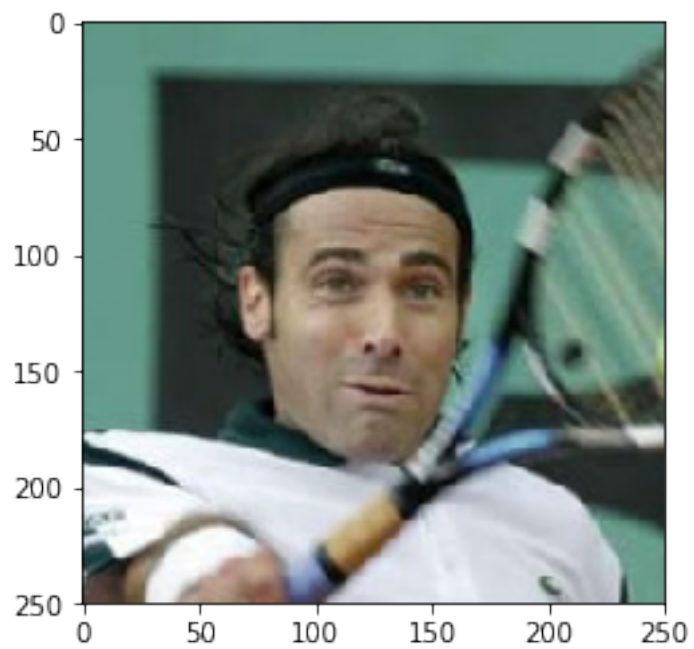

```
In [26]: ## TODO: Execute your algorithm from Step 6 on
        ## at least 6 images on your computer.
        ## Feel free to use as many code cells as needed.

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

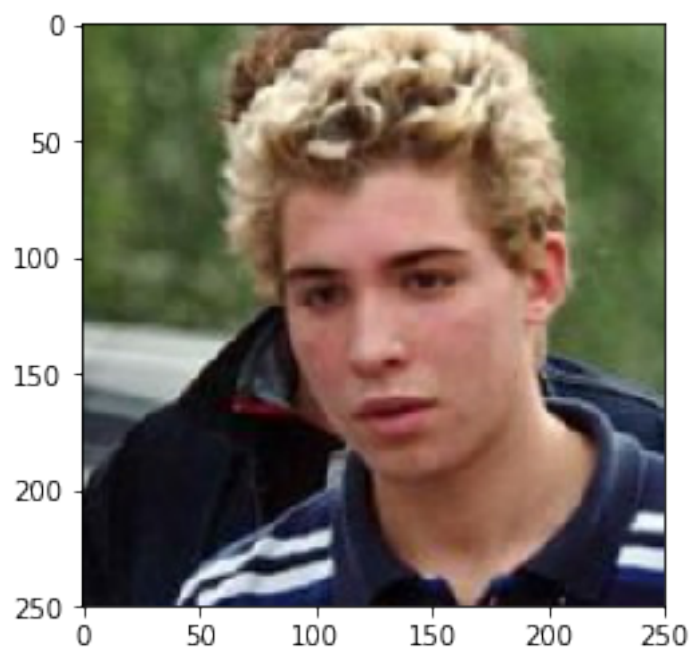
Human prediction: Beagle



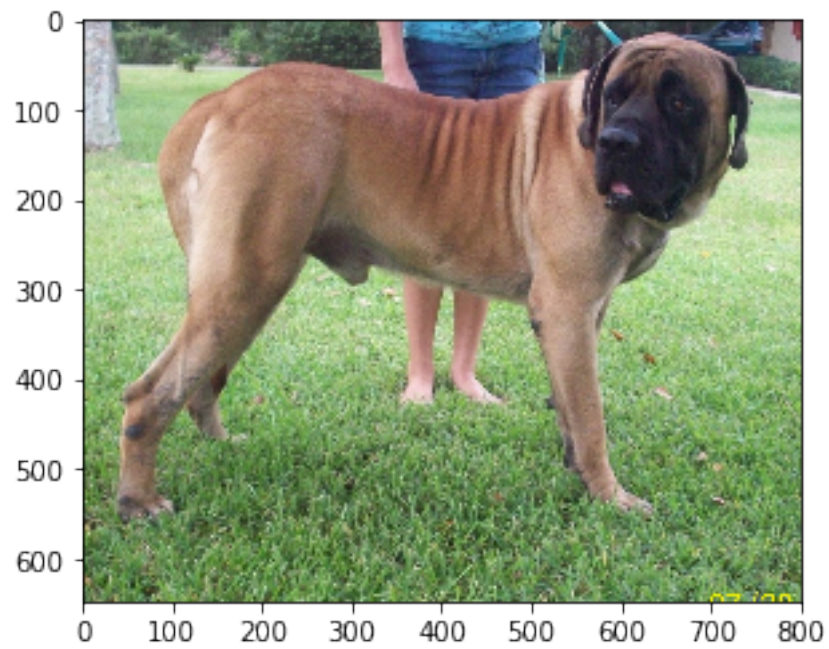
Human prediction: Bull terrier



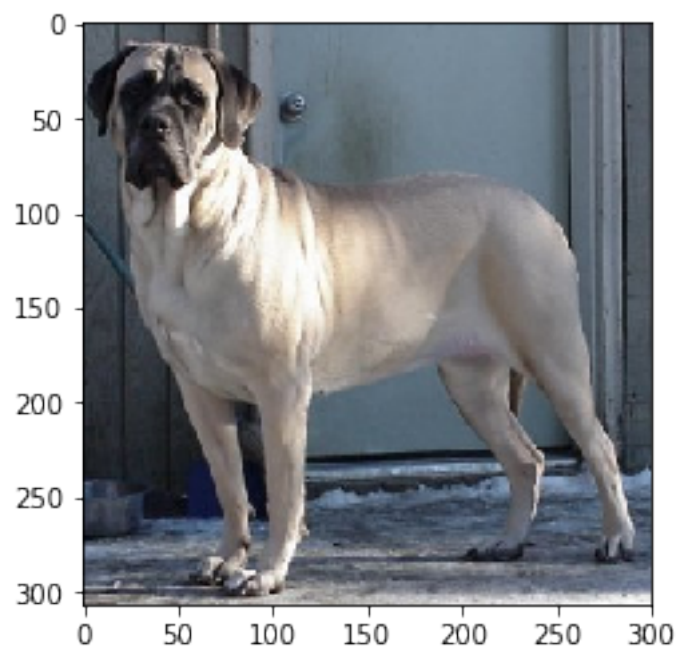
Human prediction: American water spaniel



Dog prediction: Bullmastiff



Dog prediction: Bullmastiff



Dog prediction: Bullmastiff

