dog_app

July 27, 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

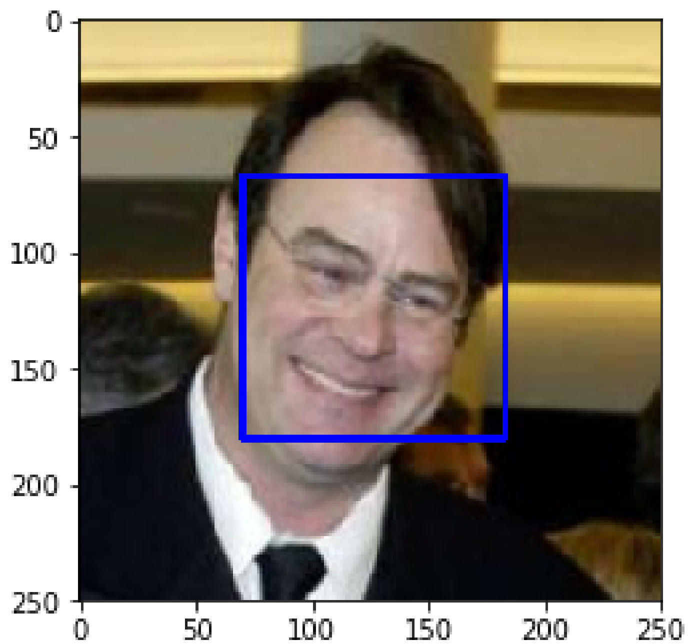
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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))
```

```python
# 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: * human_files have 98% detected human face * dog_files have 17% detected human face

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

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

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

        ## TODO: Test the performance of the face_detector algorithm
        ## on the images in human_files_short and dog_files_short.
        deteced_human_face_human_files = 0
        for human in human_files_short:
            if face_detector(human) == True:
                deteced_human_face_human_files += 1
        print ("human_files have {}% detected human face".format(deteced_human_face_human_files)


        deteced_human_face_dog_files = 0
        for dog in dog_files_short:
            if face_detector(dog) == True:
                deteced_human_face_dog_files += 1
        print ("dog_files have {}% detected human face".format(deteced_human_face_dog_files))

human_files have 98% detected human face
dog_files have 17% detected human face
```

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

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

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

1.1.3 Obtain Pre-trained VGG-16 Model

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

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

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

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

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

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:21<00:00, 25320095.79it/s]
```

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

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

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

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

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

def path_to_tensor(img_path):
    img = Image.open(img_path)
    normalize = transforms.Normalize(mean=(0.485, 0.456, 0.406),
                                     std=(0.229, 0.224, 0.225))
    preprocess = transforms.Compose([transforms.Resize(256),
                                     transforms.CenterCrop(224),
                                     transforms.ToTensor(),
                                     normalize])
    return preprocess(img)[:3,:,:].unsqueeze(0)


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

    # load image
    img = path_to_tensor(img_path)

    # use GPU if available
    if use_cuda:
        img = img.cuda()

    ret = VGG16(img)
    return torch.max(ret,1)[1].item() # predicted class index
```

1.1.5 (IMPLEMENTATION) Write a Dog Detector

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

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

```
In [8]: ### returns "True" if a dog is detected in the image stored at img_path
        def dog_detector(img_path):
            ## TODO: Complete the function.
            idx = VGG16_predict(img_path)
            return idx >= 151 and idx <= 268 # true/false
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

Question 2: Use the code cell below to test the performance of your `dog_detector` function.
- What percentage of the images in `human_files_short` have a detected dog?
- What percentage of the images in `dog_files_short` have a detected dog?
Answer: * human_files have 0% detected dog * dog_files have 100% detected dog

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

        deteced_dog_human_files = 0
        for human in human_files_short:
            if dog_detector(human) == True:
                deteced_dog_human_files += 1
        print ("human_files have {}% detected dog".format(deteced_dog_human_files))


        deteced_dog_dog_files = 0
        for dog in dog_files_short:
            if dog_detector(dog) == True:
                deteced_dog_dog_files += 1
        print ("dog_files have {}% detected dog".format(deteced_dog_dog_files))
```

```
human_files have 0% detected dog
dog_files have 100% detected dog
```

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

Step 3: Create a CNN to Classify Dog Breeds (from Scratch)
Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You

must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of at least 10%. In Step 4 of this notebook, you will have the opportunity to use transfer learning to create a CNN that attains greatly improved accuracy.

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

Brittany	Welsh Springer Spaniel

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

Curly-Coated Retriever	American Water Spaniel

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

Yellow Labrador	Chocolate Labrador

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

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

1.1.7 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

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

```
In [30]: import os
         from torchvision import datasets

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

         # based on code discussed in lectures

         import numpy as np
```

```
import torch

from torchvision import models, transforms
from torch.utils.data.sampler import SubsetRandomSampler
import matplotlib.pyplot as plt

# define dataloader parameters
batch_size  = 20
num_workers = 0

# define training and test data directories
data_dir  = '/data/dog_images/'
train_dir = os.path.join(data_dir, 'train')
valid_dir = os.path.join(data_dir, 'valid')
test_dir  = os.path.join(data_dir, 'test')

# load and transform data using ImageFolder
data_transform = transforms.Compose([transforms.Resize(224),
                                     transforms.CenterCrop(224),
                                     transforms.RandomHorizontalFlip(),
                                     transforms.RandomRotation(10),
                                     transforms.ToTensor(),
                                     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0
train_data = datasets.ImageFolder(train_dir, transform=data_transform)
valid_data = datasets.ImageFolder(valid_dir, transform=data_transform)
test_data  = datasets.ImageFolder(test_dir, transform=data_transform)

## print out some data stats
print('Num training images: ',   len(train_data))
print('Num validation images: ', len(valid_data))
print('Num test images: ',       len(test_data))

# prepare data loaders
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=T
valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=batch_size, shuffle=T
test_loader  = torch.utils.data.DataLoader(test_data,  batch_size=batch_size, shuffle=T

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

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

Question 3: Describe your chosen procedure for preprocessing the data. - How does your code resize the images (by cropping, stretching, etc)? What size did you pick for the input tensor, and why? - Did you decide to augment the dataset? If so, how (through translations, flips, rotations, etc)? If not, why not?

Answer:
I actually just resized and cropped them to 224 pixles and rotatet them randomly.
I also normalized the images, this has to be considered for printing the images and the later tests.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [16]: import torch.nn as nn
         import torch.nn.functional as F

         # define the CNN architecture
         class Net(nn.Module):
             ### TODO: choose an architecture, and complete the class
             def __init__(self):
                 super(Net, self).__init__()
                 ## Define layers of a CNN
                 self.conv1 = nn.Conv2d(3, 32, 3, stride=2, padding=1)
                 self.conv2 = nn.Conv2d(32, 64, 3,stride=2,  padding=1)
                 self.conv3 = nn.Conv2d(64, 128, 3, padding=1)
                 self.pool = nn.MaxPool2d(2, 2)
                 self.fc1 = nn.Linear(128*7*7, 512)
                 self.fc2 = nn.Linear(512, 133)
                 self.dropout = nn.Dropout(0.3)

             def forward(self, x):
                 ## Define forward behavior
                 x = self.pool(F.relu(self.conv1(x)))
                 x = self.pool(F.relu(self.conv2(x)))
                 x = self.pool(F.relu(self.conv3(x)))
                 x = x.view(-1, 128 * 7 * 7)
                 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()

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

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

Answer:

As a starting point I decided to use the network from the lectures and it turned out good enough.

To better understand the network I followed the provided guide: https://cs231n.github.io/convolutional-networks/#layers

The most common form of a ConvNet is:

- INPUT -> [[CONV -> RELU] N -> POOL?] M -> [FC -> RELU] * K -> FC For the one used the paramters are the following:
- N=1 (N>=0 and N <=3)
- M=3 (M>= 0)
- K=1 (K>=0 and K<3)

So the final Network looks like this:

- INPUT -> [[CONV -> RELU] 1 -> POOL] 3 -> [FC -> RELU] * 1 -> FC Additionally a dropout layer is used to prevent overfitting.

Of course the output of the final layer matches the classes, while the input matches the image size and color channels.

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 [17]: 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 [18]: # workaround "OSError: image file is truncated"
         from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True
```

```
In [31]: 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
```

```python
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

        # based on code discussed in lectures

        optimizer.zero_grad()             # clear the gradients of all optimized var
        output = model(data)              # forward pass: compute predicted outputs
        loss = criterion(output, target) # calculate the batch loss
        loss.backward()                   # backward pass: compute gradient of the l
        optimizer.step()                  # perform a single optimization step (para
        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

        # based on code discussed in lectures

        output = model(data)              # forward pass: compute predicted outputs
        loss = criterion(output, target) # calculate the batch loss
        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
```

```python
                ))

            ## TODO: save the model if validation loss has decreased

            # based on code discussed in lectures

            # save model if validation loss has decreased
            if valid_loss <= valid_loss_min:
                print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.fo
                valid_loss_min,
                valid_loss))
                torch.save(model.state_dict(), save_path)
                valid_loss_min = valid_loss

        # return trained model
        return model


    # train the model
    model_scratch = train(20, loaders_scratch, model_scratch, optimizer_scratch,
                          criterion_scratch, use_cuda, 'model_scratch.pt')
```

```
Epoch: 1 	Training Loss: 4.547053 	Validation Loss: 4.555163
Validation loss decreased (inf --> 4.555163).  Saving model ...
Epoch: 2 	Training Loss: 4.361560 	Validation Loss: 4.321399
Validation loss decreased (4.555163 --> 4.321399).  Saving model ...
Epoch: 3 	Training Loss: 4.228007 	Validation Loss: 4.227525
Validation loss decreased (4.321399 --> 4.227525).  Saving model ...
Epoch: 4 	Training Loss: 4.082554 	Validation Loss: 4.182530
Validation loss decreased (4.227525 --> 4.182530).  Saving model ...
Epoch: 5 	Training Loss: 3.989081 	Validation Loss: 4.124466
Validation loss decreased (4.182530 --> 4.124466).  Saving model ...
Epoch: 6 	Training Loss: 3.881308 	Validation Loss: 4.030378
Validation loss decreased (4.124466 --> 4.030378).  Saving model ...
Epoch: 7 	Training Loss: 3.740377 	Validation Loss: 3.992798
Validation loss decreased (4.030378 --> 3.992798).  Saving model ...
Epoch: 8 	Training Loss: 3.649130 	Validation Loss: 4.066396
Epoch: 9 	Training Loss: 3.505269 	Validation Loss: 4.039149
Epoch: 10 	Training Loss: 3.387669 	Validation Loss: 3.856383
Validation loss decreased (3.992798 --> 3.856383).  Saving model ...
Epoch: 11 	Training Loss: 3.235660 	Validation Loss: 3.855532
Validation loss decreased (3.856383 --> 3.855532).  Saving model ...
Epoch: 12 	Training Loss: 3.080685 	Validation Loss: 3.961958
Epoch: 13 	Training Loss: 2.886461 	Validation Loss: 3.924463
Epoch: 14 	Training Loss: 2.755649 	Validation Loss: 3.964626
Epoch: 15 	Training Loss: 2.541966 	Validation Loss: 3.983024
Epoch: 16 	Training Loss: 2.428187 	Validation Loss: 4.088160
Epoch: 17 	Training Loss: 2.220111 	Validation Loss: 4.064488
```

```
Epoch: 18       Training Loss: 2.090311       Validation Loss: 4.242959
Epoch: 19       Training Loss: 1.915143       Validation Loss: 4.247606
Epoch: 20       Training Loss: 1.770078       Validation Loss: 4.071773
```

```
In [32]: # 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 [33]: 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: 3.910885
```

```
Test Accuracy: 10% (89/836)
```

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

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

1.1.12 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

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

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

In [22]:
```
## TODO: Specify data loaders
loaders_transfer = 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 [23]:
```
import torchvision.models as models
import torch.nn as nn

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

# freeze training for all "features" layers
for param in model_transfer.parameters():
    param.requires_grad = False

# modify last layer to match it our classes
model_transfer.fc = nn.Linear(2048, 133, bias=True)
fc_parameters = model_transfer.fc.parameters()
for param in fc_parameters:
    param.requires_grad = True

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, 87552433.83it/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:

I decided to use the pretrained resnet5SS network as shown in the lectures.

In a first step I froze the features paramters as we dont want to change the net itself. We only want to optimice the classifier to match our classes. This is important for the optimizer too.

The last layer of the network is modified. We want to keep the number of inputs, but want the output to match our classes

Again to better understand the network I followed the provided guide: https://cs231n.github.io/convolutional-networks/#layers

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 [24]: criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.Adam(model_transfer.fc.parameters(), lr=0.01)
```

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 [25]: # train the model
         model_transfer = train(5, loaders_transfer, model_transfer, optimizer_transfer, criteri

         # 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: 5.407888         Validation Loss: 1.440149
Validation loss decreased (inf --> 1.440149).  Saving model ...
Epoch: 2         Training Loss: 0.907148         Validation Loss: 1.144908
Validation loss decreased (1.440149 --> 1.144908).  Saving model ...
Epoch: 3         Training Loss: 0.626343         Validation Loss: 0.933565
Validation loss decreased (1.144908 --> 0.933565).  Saving model ...
Epoch: 4         Training Loss: 0.635127         Validation Loss: 1.136321
Epoch: 5         Training Loss: 0.693377         Validation Loss: 1.063679
```

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

```
Test Loss: 1.038646
```

```
Test Accuracy: 75% (635/836)
```

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 [27]: ### 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("_", " ").title() for item in train_data.classes]
         #dog_names = [item[35:-1] for item in sorted(glob("../../../data/dog_images/train/*/"))

         def predict_breed_transfer(img_path):
             # extract bottleneck features
             image_tensor = path_to_tensor(img_path)
             if use_cuda:
                 image_tensor = image_tensor.cuda()

             # obtain predicted vector
             prediction = model_transfer(image_tensor)
             # return dog breed that is predicted by the model
             prediction = prediction.cpu()
             prediction = prediction.data.numpy().argmax()
             return class_names[prediction]
```

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.

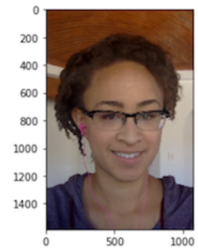
You are welcome to write your own functions for detecting humans and dogs in images, but feel free to use the `face_detector` and `human_detector` functions developed above. You are **required** to use your CNN from Step 4 to predict dog breed.

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 [28]: ### 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
```

```
hello, human!
```




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

Sample Human Output

```python
img = Image.open(img_path)
plt.imshow(img)
plt.axis('off')
plt.show()
if dog_detector(img_path) > 0:
    prediction = predict_breed_transfer(img_path)
    print("This is a {0}".format(prediction))
elif face_detector(img_path) > 0:
    prediction = predict_breed_transfer(img_path)
    print("This photo looks like a {0}".format(prediction))
else:
    print("Error")
```

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:

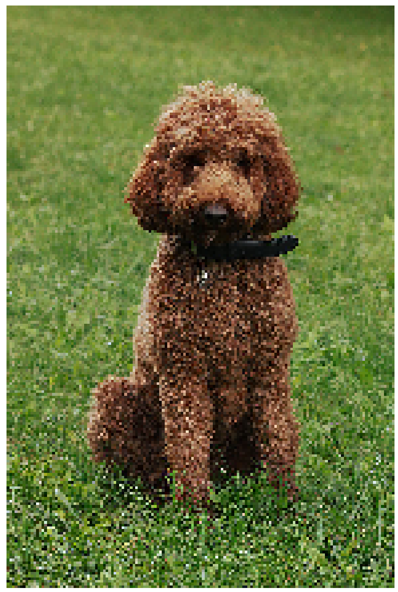
I am really impressed by the result!

Possible steps for improvement I can think of:

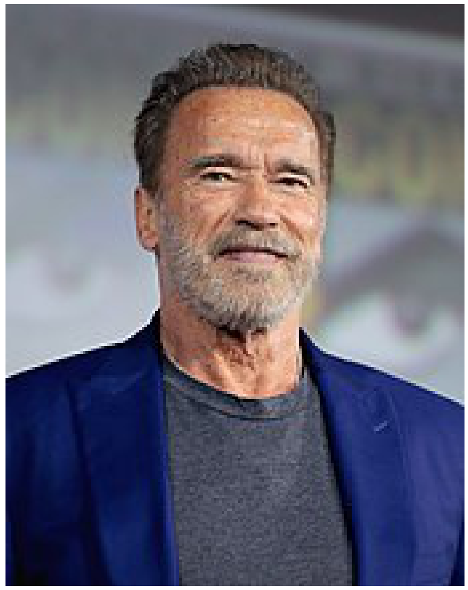
- More Training, maybe even with different optimizer
- Using more Images for training
- Resizing and only using fractions of the original images

In [29]:
```python
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
```

```python
## Feel free to use as many code cells as needed.

import os

for filename in os.listdir('samples'):
    if filename.endswith('.jpg'):
        run_app(os.path.join("samples", filename))
```



```
This is a Irish Water Spaniel
```



This photo looks like a Dogue De Bordeaux

This is a Australian Cattle Dog

This photo looks like a Entlebucher Mountain Dog

This photo looks like a Basset Hound

This is a Cardigan Welsh Corgi

In [ ]:

In [ ]: