dog_app

January 20, 2019

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

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

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

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

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

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

Step 0: Import Datasets

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

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

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

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

```
In [2]: 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.
```

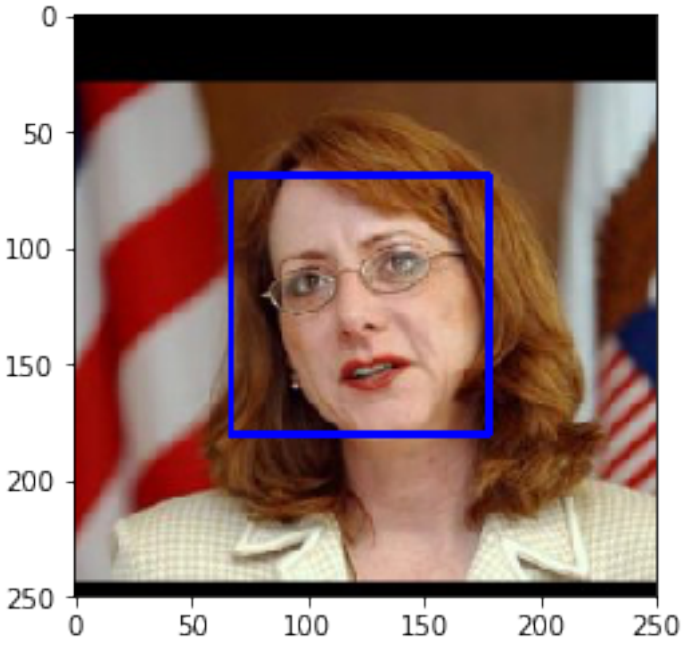
Step 1: Detect Humans

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 [3]: import cv2
        import matplotlib.pyplot as plt
        %matplotlib inline


        face_cascade = cv2.CascadeClassifier('haarcascades/haarcascade_frontalface_alt.xml')

        img = cv2.imread(human_files[15])

        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

        faces = face_cascade.detectMultiScale(gray)

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

        for (x,y,w,h) in faces:
            cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)

        cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

        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]: 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 detection in human images: 98%
- Human detection in dog images: 17%

```
In [10]: from tqdm import tqdm

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

         humans_in_dog_files = 0
         humans_in_human_files = 0

         for i in range(100):
             if(face_detector(human_files_short[i])):
                 humans_in_human_files += 1
             if(face_detector(dog_files_short[i])):
                 humans_in_dog_files += 1

         print('Human detection in human images: {}%'.format(humans_in_human_files))
         print('Human detection in dog images: {}%'.format(humans_in_dog_files))
```

```
Human detection in human images: 98%
Human detection in dog images: 17%
```

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

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 [12]: 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 [00:04<00:00, 118561044.73it/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 [13]: from PIL import Image
         import torchvision.transforms as transforms
         from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True

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

             Args:
                 img_path: path to an image

             Returns:
                 Index corresponding to VGG-16 model's prediction
             '''
             img = Image.open(img_path)

             transform = transforms.Compose([transforms.Resize(256),
                                             transforms.CenterCrop(254),
```

```
                                        transforms.ToTensor(),
                                        transforms.Normalize([0.485, 0.456, 0.406], [0.229,
                                       ])

    tensor = transform(img).unsqueeze_(0)
    if use_cuda:
        tensor = tensor.cuda()

    ## 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
    predictions = VGG16(tensor)

    prediction = torch.argmax(predictions)

    return prediction.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 [14]: def dog_detector(img_path):
             prediction = VGG16_predict(img_path)

             return prediction >= 151 and prediction <= 268
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

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

Answer:

- Dogs detection in human images accuracy: 99%
- Dogs detection in dog images accuracy: 100%

```
In [15]: dogs_in_human_files = 0
         dogs_in_dog_files = 0

         for i in range(100):
             if(dog_detector(human_files_short[i])):
                 dogs_in_human_files += 1
             if(dog_detector(dog_files_short[i])):
```

```
            dogs_in_dog_files += 1

    print('Dogs detection in human images: {}%'.format(dogs_in_human_files))
    print('Dogs detection in dog images: {}%'.format(dogs_in_dog_files))
```

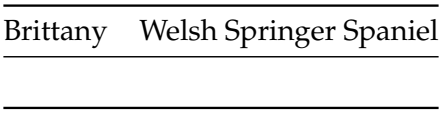
```
Dogs detection in human images: 0%
Dogs detection in dog images: 100%
```

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`.

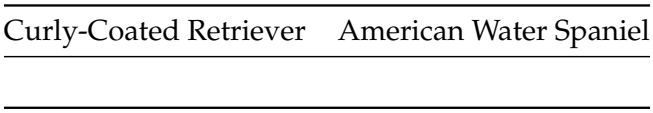
Step 3: Create a CNN to Classify Dog Breeds (from Scratch)

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.

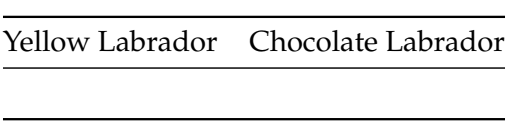
We mention that the task of assigning breed to dogs from images is considered exceptionally challenging. To see why, consider that *even a human* would have trouble distinguishing between a Brittany and a Welsh Springer Spaniel.



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



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.



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 [16]: import torch
         from torchvision import datasets
         import torchvision.transforms as transforms

         print("loading image data ... ")

         train_transforms = transforms.Compose([transforms.Resize(244),
                                                transforms.CenterCrop(244),
                                                transforms.RandomHorizontalFlip(),
                                                transforms.RandomVerticalFlip(),
                                                transforms.RandomRotation(20),
                                                transforms.ToTensor(),
                                                transforms.Normalize([0.485, 0.456, 0.406],
                                                                     [0.229, 0.224, 0.225])])

         test_transforms = transforms.Compose([transforms.Resize(244),
                                               transforms.CenterCrop(244),
                                               transforms.ToTensor(),
                                               transforms.Normalize([0.485, 0.456, 0.406],
                                                                    [0.229, 0.224, 0.225])])


         path = "/data/dog_images"

         train_dataset = datasets.ImageFolder(path + "/train", transform=train_transforms)
         valid_dataset = datasets.ImageFolder(path + "/valid", transform=test_transforms)
         test_dataset = datasets.ImageFolder(path + "/test", transform=test_transforms)

         # ---- print out some data stats ----
         print('  Number of train images: ', len(train_dataset))
         print('  Number of test images:  ', len(test_dataset))
         print('  Number of valid images: ', len(valid_dataset))
         # -----------------------------------


         batch_size = 32

         trainloader = torch.utils.data.DataLoader( train_dataset, batch_size=batch_size, shuffl
         testloader = torch.utils.data.DataLoader( test_dataset, batch_size=batch_size )
```

```
        validloader = torch.utils.data.DataLoader( valid_dataset, batch_size=batch_size, shuffl

        loaders_scratch = {
            'train': trainloader,
            'test': validloader,
            'valid': testloader
        }

        print('done.')
```

```
loading image data ...
  Number of train images:  6680
  Number of test images:   836
  Number of valid images:  835
done.
```

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: The images get resized to 244x244 by rescalating them. I pick 244x244 because the images looks good and also there were a lot of example where people used a similar size.

I decided to augment the training dataset for two reason, first to avoid overfitting and second because there were only 6680 training images. That means that each breed had only like 50 images.

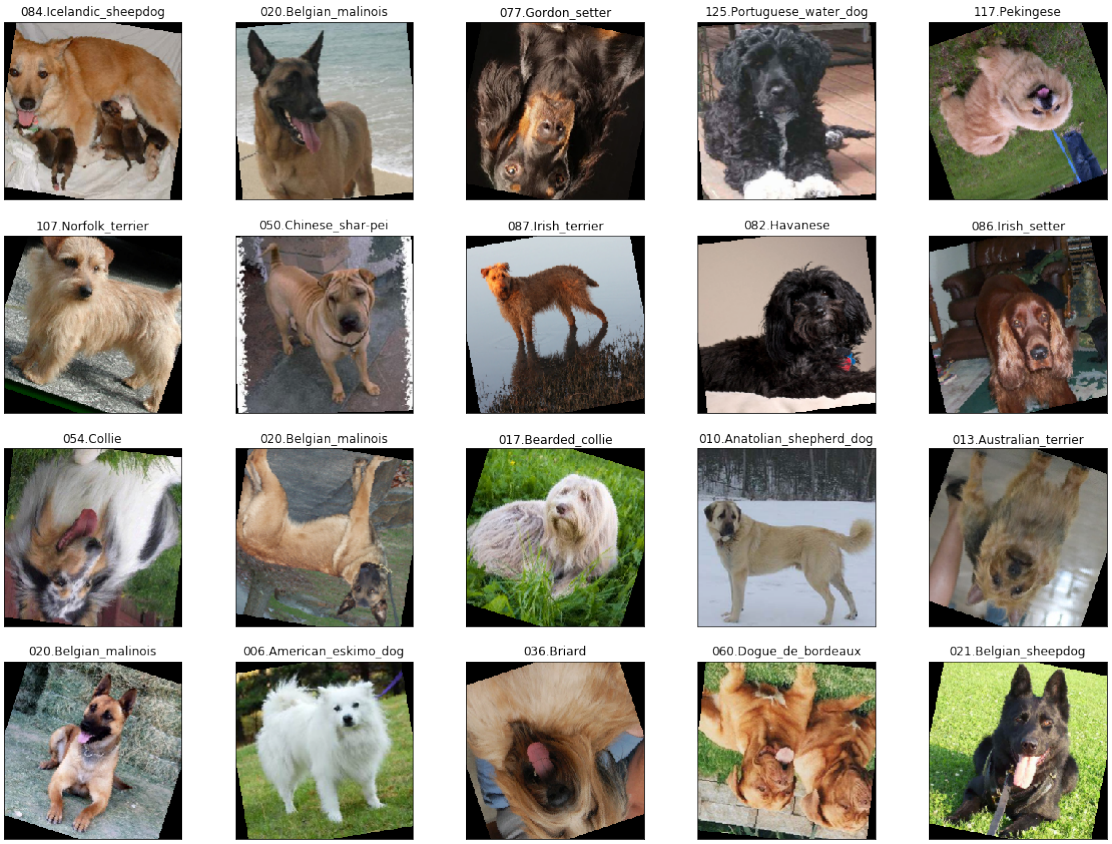
The augmentation consists of a random virtical flipm an random horizontal flip and a random rotation.

```
In [17]: # get classes of training datas
         class_names = train_dataset.classes
         number_classes = len(class_names)

         # correct output-size of the CNN
         param_output_size = len(class_names)

         print("number of classes:", number_classes)
         print("")
         print("class names: \n", class_names)
```

```
number of classes: 133

class names:
 ['001.Affenpinscher', '002.Afghan_hound', '003.Airedale_terrier', '004.Akita', '005.Alaskan_mal
```

```
In [18]: import numpy as np
         import matplotlib.pyplot as plt
         %matplotlib inline
```

```
inputs, classes = next( iter(loaders_scratch['train']) )

fig = plt.figure(figsize=(20, 15))

for idx, (image, label) in enumerate(zip(inputs[:20], classes[:20])):
    image = image.to("cpu").clone().detach()
    image = image.numpy().squeeze()
    image = image.transpose(1,2,0)
    # normalize image
    image = image * np.array((0.229, 0.224, 0.225)) + np.array((0.485, 0.456, 0.406))
    image = image.clip(0, 1)
    ax = fig.add_subplot(4, 5, idx+1, xticks=[], yticks=[])
    plt.imshow(image)
    plt.title(class_names[label])
```



1.1.8 (IMPLEMENTATION) Model Architecture

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

In [19]:
```
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

        self.conv1 = nn.Conv2d(3, 18, 3, padding=1)
        self.conv2 = nn.Conv2d(18, 18, 3, padding=2)
        self.conv3 = nn.Conv2d(18, 36, 3, padding=2)
        self.conv4 = nn.Conv2d(36, 126, 3, padding=1)

        self.pool = nn.MaxPool2d(2, 2)

        self.fc1 = nn.Linear(16*16*126, 500)
        self.fc2 = nn.Linear(500, 250)
        self.fc3 = nn.Linear(250, 133)

        self.dropout = nn.Dropout(p=0.25)



    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 = F.relu(self.conv4(x))
        x = self.pool(x)

        x = x.view(-1, 16*16*126)

        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.relu(self.fc2(x))
        x = self.dropout(x)
        x = self.fc3(x)

        return x



# instantiate the CNN
```

```
        model_scratch = Net()
        print(model_scratch)

        if use_cuda:
            model_scratch.cuda()
```

```
Net(
  (conv1): Conv2d(3, 18, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(18, 18, kernel_size=(3, 3), stride=(1, 1), padding=(2, 2))
  (conv3): Conv2d(18, 36, kernel_size=(3, 3), stride=(1, 1), padding=(2, 2))
  (conv4): Conv2d(36, 126, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=32256, out_features=500, bias=True)
  (fc2): Linear(in_features=500, out_features=250, bias=True)
  (fc3): Linear(in_features=250, out_features=133, bias=True)
  (dropout): Dropout(p=0.25)
)
```

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

Answer: First i began with a very simple CNN, 1 Conv2d layer and 1 Linear layer, after training the model I realised that more layers where needed. I added 3 more Conv2d layers and 2 more Linear layers. I could not achive the 10% so I modify the number of in and out layers of the Conv2d layers.

Then after training the last CNN I achived a 10% of accuracy.

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 [24]: import torch.optim as optim

        criterion_scratch = torch.nn.CrossEntropyLoss()

        optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=0.01)
```

1.1.10 (IMPLEMENTATION) Train and Validate the Model

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

```
In [25]: def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
            """returns trained model"""
            print("start training for {} epochs ...".format(n_epochs))
            # initialize tracker for minimum validation loss
            # valid_loss_min = 4.015238
            valid_loss_min = 0.512025
```

```
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()    # --- set model to train mode
    for batch_idx, (data, target) in enumerate(loaders['train']):

        if use_cuda:
            data, target = data.cuda(), target.cuda()

        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the batch loss
        loss = criterion(output, target)
        # backward pass: compute gradient of the loss with respect to model paramet
        loss.backward()
        # perform a single optimization step (parameter update)
        optimizer.step()
        # update training loss
        #train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss
        train_loss += loss.item()*data.size(0)
        # ----------------------------

    ######################
    # validate the model #
    ######################
    model.eval()        # ---- set model to evaluation mode
    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
        # ----------------------------
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the batch loss
        loss = criterion(output, target)
        # update average validation loss
        valid_loss += loss.item() * data.size(0)
        # ----------------------------

    # ----------------------------
```

```
            # calculate average losses
            train_loss = train_loss / len(loaders['train'].dataset)
            valid_loss = valid_loss / len(loaders['valid'].dataset)
            # -----------------------------

            # print training/validation statistics
            print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format( epo

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

        print("done")
        # return trained model
        return model
```

```
In [26]: load_model = True

        if load_model:
            model_scratch.load_state_dict(torch.load('model_scratch_2.pt'))

        train_model = False
        if train_model:
            train(55, loaders_scratch, model_scratch, optimizer_scratch, criterion_scratch, use
```

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%.

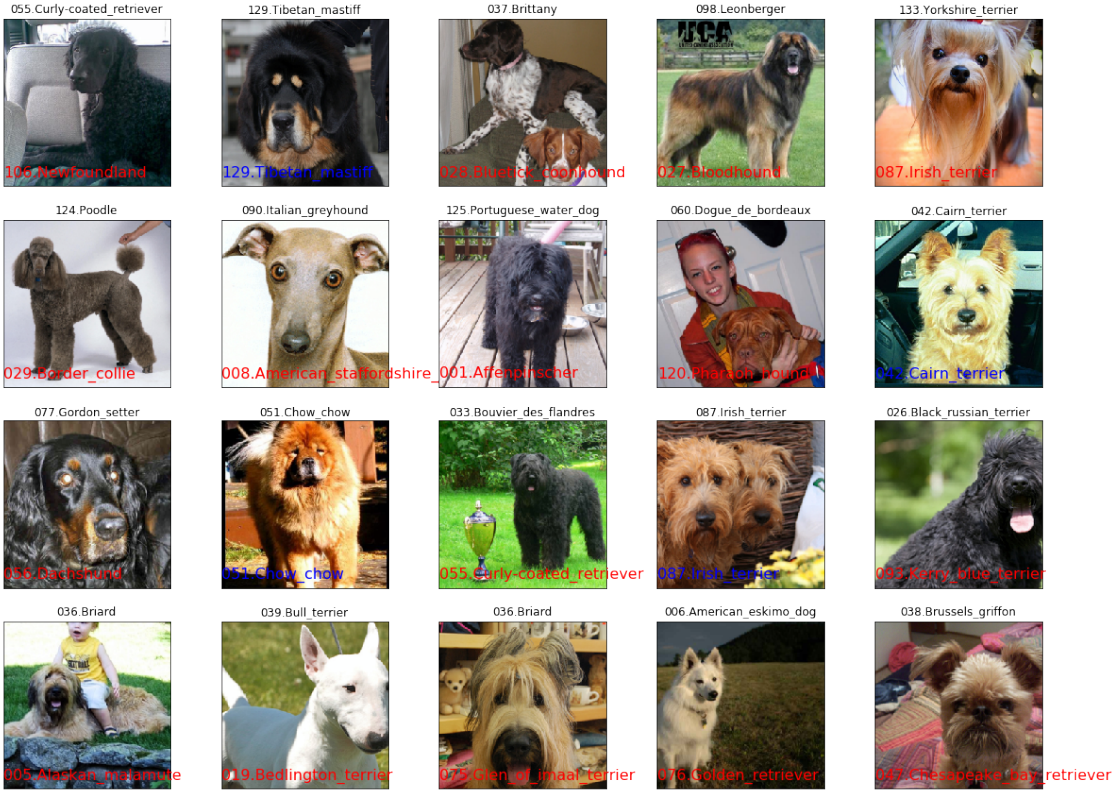
```
In [27]: def test(loaders, model, criterion, use_cuda):

            # 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 += ((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: 3.967154


Test Accuracy: 10% (86/835)
```

In [28]:
```
    model_scratch.eval()

    images, labels = next( iter(loaders_scratch['test']) )

    if use_cuda:
        images, labels = images.cuda(), labels.cuda()

    output = model_scratch(images)
    predictions = output.data.max(1, keepdim=True)[1]

    fig = plt.figure(figsize=(20, 15))

    for idx, (image, label, pred) in enumerate(zip(images[:20], labels[:20], predictions[:2
        image = image.to("cpu").clone().detach()
        image = image.numpy().squeeze()
        image = image.transpose(1,2,0)
        # normalize image
        image = image * np.array((0.229, 0.224, 0.225)) + np.array((0.485, 0.456, 0.406))
        image = image.clip(0, 1)
        ax = fig.add_subplot(4, 5, idx+1, xticks=[], yticks=[])
```

```
plt.imshow(image)
plt.title(class_names[label])
color= "red"
if label == pred:
    color="blue"
plt.text(0, 230, class_names[pred], fontsize=16, color=color)
```



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.

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 [29]:
```
import torchvision.models as models
import torch.nn as nn

## TODO: Specify model architecture
# Load the pretrained model from pytorch
model_transfer = models.vgg16(pretrained=True)



if use_cuda:
    model_transfer = model_transfer.cuda()

# print out the model structure
print(model_transfer)
```

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

```
    (27): ReLU(inplace)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace)
    (2): Dropout(p=0.5)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace)
    (5): Dropout(p=0.5)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

```
In [30]: for param in model_transfer.features.parameters():
             param.requires_grad = False
```

```
In [31]: import torch.nn as nn

         n_inputs = model_transfer.classifier[6].in_features

         last_layer =  nn.Linear(n_inputs, 133)

         model_transfer.classifier[6] = last_layer

         if use_cuda:
             model_transfer.cuda()

         print(model_transfer.classifier[6].out_features)
```

```
133
```

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 use the VGG16 pretrained model, just modify the last layer to produce 133 outputs. I think this architecture is suitable because VGG16 has a lot of Conv2d layers and also I was able to get a 85% of accuracy.

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 [32]: criterion_transfer = torch.nn.CrossEntropyLoss()
         optimizer_transfer = optim.SGD(model_transfer.classifier.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 [33]: # train the model
         train_model = False

         if train_model:
             model_transfer =  train(10, loaders_scratch, model_transfer, optimizer_transfer, cr

         # 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 [34]: test(loaders_scratch, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.447604


Test Accuracy: 85% (712/835)
```

```
In [35]: model_scratch.eval()

         images, labels = next( iter(loaders_scratch['test']) )

         if use_cuda:
             images, labels = images.cuda(), labels.cuda()

         output = model_transfer(images)
         predictions = output.data.max(1, keepdim=True)[1]

         fig = plt.figure(figsize=(20, 15))

         for idx, (image, label, pred) in enumerate(zip(images[:20], labels[:20], predictions[:2
             image = image.to("cpu").clone().detach()
             image = image.numpy().squeeze()
             image = image.transpose(1,2,0)
             # normalize image
             image = image * np.array((0.229, 0.224, 0.225)) + np.array((0.485, 0.456, 0.406))
             image = image.clip(0, 1)
             ax = fig.add_subplot(4, 5, idx+1, xticks=[], yticks=[])
             plt.imshow(image)
             plt.title(class_names[label])
```

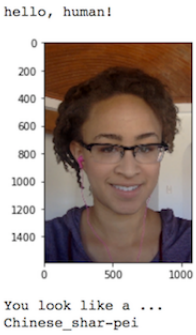
```
        color= "red"
        if label == pred:
            color="blue"
        plt.text(0, 230, class_names[pred], fontsize=16, color=color)
```

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 [36]: class_names = train_dataset.classes
```

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

            transform = transforms.Compose([transforms.Resize(256),
                                            transforms.CenterCrop(254),
                                            transforms.ToTensor(),
                                            transforms.Normalize([0.485, 0.456, 0.406], [0.229,
```



Sample Human Output

```
                                        ])

    tensor = transform(img).unsqueeze_(0)
    if use_cuda:
        tensor = tensor.cuda()

    ## 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
    predictions = model_transfer(tensor)

    prediction = torch.argmax(predictions)


    return class_names[prediction]


predict_breed_transfer("/data/dog_images/test/056.Dachshund/Dachshund_03953.jpg")
```

```
Out[36]: '056.Dachshund'
```

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

         def run_app(img_path):
             if dog_detector(img_path):
                 prediction = predict_breed_transfer(img_path)
                 return 'This dog looks like a ... {}'.format(prediction)
             if face_detector(img_path):
                 prediction = predict_breed_transfer(img_path)
                 return 'You look like a ... {}'.format(prediction)
             return 'No humans or dogs detected'
```

Step 6: Test Your Algorithm

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

1.1.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

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

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

Answer: Much better, after having a difficult time training my own CNN i could not believe that this CNN got a 85%.

My algorithm could be better in this ways.

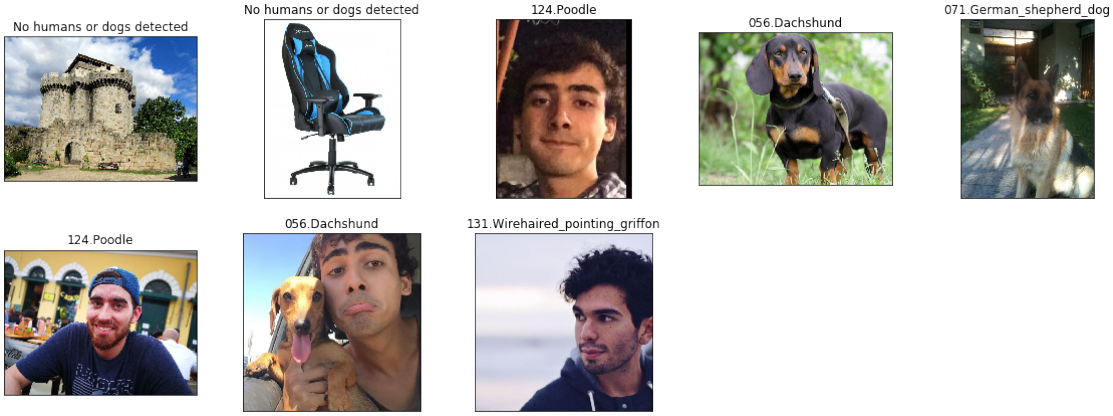
- Pre processing PNG files
- Trying with different pretrained CNN
- Improve Performance With Data: Getting more data could help a lot.
- Experiment with very large and very small learning rates.

PD: Thanks for the first feedback, very nice article recomended!

```
In [38]: import matplotlib.pyplot as plt
         import numpy as np
         from glob import glob
         %matplotlib inline

         my_files = np.array(glob("./my_files/*"))


         fig = plt.figure(figsize=(20, 15))
```

```
        for idx, (file) in enumerate(my_files):
            pred  = run_app(file)
            ax = fig.add_subplot(4, 5, idx+1, xticks=[], yticks=[])
            img = Image.open(file)
            plt.imshow(img)
            plt.title(pred)
```

```
You look like a ... 124.Poodle
This dog looks like a ... 056.Dachshund
This dog looks like a ... 071.German_shepherd_dog
You look like a ... 124.Poodle
This dog looks like a ... 056.Dachshund
You look like a ... 131.Wirehaired_pointing_griffon
```



In [ ]: