dog_app

October 28, 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:
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 [4]: import torch
        device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
In [32]: 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

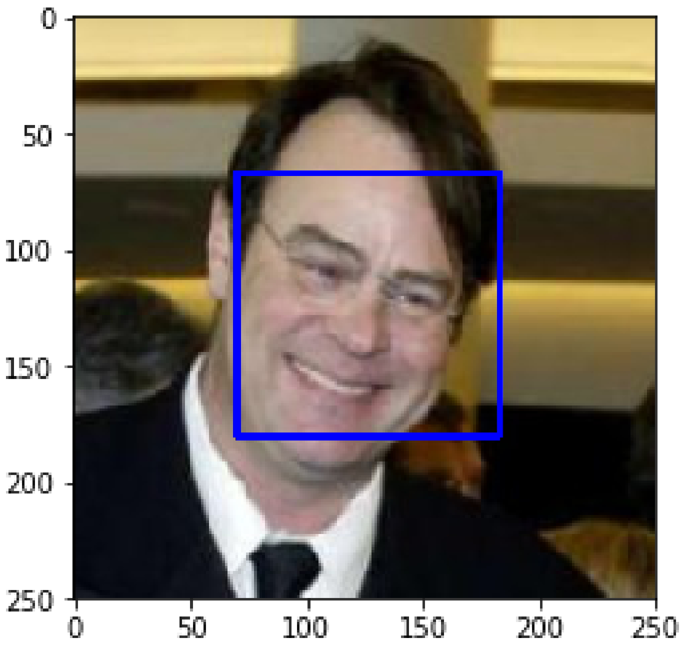
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [33]: 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 [34]: # 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: Using haarcascades face detector:
Detected human faces of first 100 human images: 98.0%
Detected faces of first 100 dogs images: 17.0%

```
In [35]: 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.


         humans = 0.0
         dogs = 0.0

         for i in range(0,len(human_files_short)):
             human_image_path = human_files_short[i]
             dogs_image_path = dog_files_short[i]
             if face_detector(human_image_path) == True:
                 humans += 1
             if face_detector(dogs_image_path) == True:
                 dogs += 1
```

```
        print("Percentage of Humans:",humans/len(human_files_short)*100,"%")
        print("Percentage of Dogs:",dogs/len(human_files_short)*100,"%")
```

```
Percentage of Humans: 98.0 %
Percentage of Dogs: 17.0 %
```

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 [36]: ### (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 [37]: 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()
```

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 [38]: from PIL import Image
         import torchvision.transforms as transforms
         from torch.autograd import Variable

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

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

             Args:
                 img_path: path to an image

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

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

             transform_data = transforms.Compose([transforms.RandomResizedCrop(224),
                                                  transforms.ToTensor(),
                                                  transforms.Normalize([0.485,0.456,0.406],
                                                                       [0.229,0.224,0.225])])
             img = transform_data(img)
             img = img.unsqueeze(0)

             prediction = VGG16(img)
             prediction_class = prediction.data.numpy().argmax()




             return prediction_class # 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 [39]: ### returns "True" if a dog is detected in the image stored at img_path
         def dog_detector(img_path):
             ## TODO: Complete the function.

             class_index = VGG16_predict(img_path)

             if class_index>=151 and class_index<= 268:
                 return True
             else:
                 return 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:

```
In [40]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.

         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.

         humans = 0.0
         dogs = 0.0

         for i in range(0,len(human_files_short)):
             image_path_human = human_files_short[i]
             image_path_dog = dog_files_short[i]
             if dog_detector(image_path_human) == True:
                 humans += 1
             if dog_detector(image_path_dog) == True:
                 dogs += 1


         print("Percentage of Humans:",humans/len(human_files_short)*100,"%")
         print("Percentage of Dogs:",dogs/len(human_files_short)*100,"%")
```

```
Percentage of Humans: 1.0 %
Percentage of Dogs: 97.0 %
```

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 [12]: ### (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 [28]: import os
         from torchvision import datasets
         import torchvision.transforms as transforms
         from torch.utils.data.sampler import SubsetRandomSampler
         ### TODO: Write data loaders for training, validation, and test sets
         ## Specify appropriate transforms, and batch_sizes

         num_workers = 0
         batch_size = 20
         val_size = 0.3

         transform_data_train = transforms.Compose([transforms.Resize(256),
                                             transforms.RandomResizedCrop(224),
                                             transforms.RandomHorizontalFlip(),
                                             transforms.RandomVerticalFlip(),
                                                transforms.ToTensor(),
                                                transforms.Normalize([0.485,0.456,0.406],
                                                                      [0.229,0.224,0.225])])
         transform_data_test_val = transforms.Compose([transforms.Resize(256),
                                             transforms.CenterCrop(224),
                                                transforms.ToTensor(),
                                                transforms.Normalize([0.485,0.456,0.406],
                                                                      [0.229,0.224,0.225])])
         train_data = datasets.ImageFolder('/data/dog_images/train',transform = transform_data_t
         val_data = datasets.ImageFolder('/data/dog_images/valid',transform = transform_data_tes
         test_data = datasets.ImageFolder('/data/dog_images/test',transform = transform_data_tes
         data = {"train" : train_data, "valid" : val_data, "test" : test_data}

         train_loader = torch.utils.data.DataLoader(train_data,batch_size=batch_size,shuffle=Tru
                                                  num_workers = num_workers)
         val_loader = torch.utils.data.DataLoader(val_data,batch_size=batch_size,shuffle=True,
                                                  num_workers = num_workers)
         test_loader = torch.utils.data.DataLoader(test_data,batch_size=batch_size,shuffle=True,
                                                  num_workers = num_workers)

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

```
In [6]: print("Length Of Training Data:",len(train_data))
        print("Length Of validation Data:",len(val_data))
        print("Length Of Testing Data:",len(test_data))

Length Of Training Data: 6680
Length Of validation Data: 835
Length Of Testing Data: 836
```

```
In [15]: class_names = train_data.classes
         class_names

Out[15]: ['001.Affenpinscher',
          '002.Afghan_hound',
          '003.Airedale_terrier',
          '004.Akita',
          '005.Alaskan_malamute',
          '006.American_eskimo_dog',
          '007.American_foxhound',
          '008.American_staffordshire_terrier',
          '009.American_water_spaniel',
          '010.Anatolian_shepherd_dog',
          '011.Australian_cattle_dog',
          '012.Australian_shepherd',
          '013.Australian_terrier',
          '014.Basenji',
          '015.Basset_hound',
          '016.Beagle',
          '017.Bearded_collie',
          '018.Beauceron',
          '019.Bedlington_terrier',
          '020.Belgian_malinois',
          '021.Belgian_sheepdog',
          '022.Belgian_tervuren',
          '023.Bernese_mountain_dog',
          '024.Bichon_frise',
          '025.Black_and_tan_coonhound',
          '026.Black_russian_terrier',
          '027.Bloodhound',
          '028.Bluetick_coonhound',
          '029.Border_collie',
          '030.Border_terrier',
          '031.Borzoi',
          '032.Boston_terrier',
          '033.Bouvier_des_flandres',
          '034.Boxer',
          '035.Boykin_spaniel',
          '036.Briard',
'037.Brittany',
'038.Brussels_griffon',
'039.Bull_terrier',
'040.Bulldog',
'041.Bullmastiff',
'042.Cairn_terrier',
'043.Canaan_dog',
'044.Cane_corso',
'045.Cardigan_welsh_corgi',
'046.Cavalier_king_charles_spaniel',
'047.Chesapeake_bay_retriever',
'048.Chihuahua',
'049.Chinese_crested',
'050.Chinese_shar-pei',
'051.Chow_chow',
'052.Clumber_spaniel',
'053.Cocker_spaniel',
'054.Collie',
'055.Curly-coated_retriever',
'056.Dachshund',
'057.Dalmatian',
'058.Dandie_dinmont_terrier',
'059.Doberman_pinscher',
'060.Dogue_de_bordeaux',
'061.English_cocker_spaniel',
'062.English_setter',
'063.English_springer_spaniel',
'064.English_toy_spaniel',
'065.Entlebucher_mountain_dog',
'066.Field_spaniel',
'067.Finnish_spitz',
'068.Flat-coated_retriever',
'069.French_bulldog',
'070.German_pinscher',
'071.German_shepherd_dog',
'072.German_shorthaired_pointer',
'073.German_wirehaired_pointer',
'074.Giant_schnauzer',
'075.Glen_of_imaal_terrier',
'076.Golden_retriever',
'077.Gordon_setter',
'078.Great_dane',
'079.Great_pyrenees',
'080.Greater_swiss_mountain_dog',
'081.Greyhound',
'082.Havanese',
'083.Ibizan_hound',
'084.Icelandic_sheepdog',
'085.Irish_red_and_white_setter',
'086.Irish_setter',
'087.Irish_terrier',
'088.Irish_water_spaniel',
'089.Irish_wolfhound',
'090.Italian_greyhound',
'091.Japanese_chin',
'092.Keeshond',
'093.Kerry_blue_terrier',
'094.Komondor',
'095.Kuvasz',
'096.Labrador_retriever',
'097.Lakeland_terrier',
'098.Leonberger',
'099.Lhasa_apso',
'100.Lowchen',
'101.Maltese',
'102.Manchester_terrier',
'103.Mastiff',
'104.Miniature_schnauzer',
'105.Neapolitan_mastiff',
'106.Newfoundland',
'107.Norfolk_terrier',
'108.Norwegian_buhund',
'109.Norwegian_elkhound',
'110.Norwegian_lundehund',
'111.Norwich_terrier',
'112.Nova_scotia_duck_tolling_retriever',
'113.Old_english_sheepdog',
'114.Otterhound',
'115.Papillon',
'116.Parson_russell_terrier',
'117.Pekingese',
'118.Pembroke_welsh_corgi',
'119.Petit_basset_griffon_vendeen',
'120.Pharaoh_hound',
'121.Plott',
'122.Pointer',
'123.Pomeranian',
'124.Poodle',
'125.Portuguese_water_dog',
'126.Saint_bernard',
'127.Silky_terrier',
'128.Smooth_fox_terrier',
'129.Tibetan_mastiff',
'130.Welsh_springer_spaniel',
'131.Wirehaired_pointing_griffon',
'132.Xoloitzcuintli',
        '133.Yorkshire_terrier']
```

```
In [16]: print("No. of classes:",len(train_data.classes))
```

```
No. of classes: 133
```

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: 1) Our code resize the image in 256 pixels values and random resized crop of 224 pixel values and center crop of 224 pixel values.

2) I have decided to augment the dataset by flipping the images horizontally and vertically and randomly resizing and cropping the image. Then converting it to tensor values and normalizing them.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [10]: from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True
         num_classes = len(train_data.classes)
         num_classes
```

```
Out[10]: 133
```

```
In [11]: import torch.nn as nn
         import torch.nn.functional as F

         # define the CNN architecture
         class Net(nn.Module):
             def __init__(self, num_classes, depth_1 = 32):
                 super(Net, self).__init__()
                 # Keep track of things
                 depth_2 = depth_1 * 2
                 depth_3 = depth_2 * 2
                 # Max pooling layer
                 self.pool = nn.MaxPool2d(2,2)
                 # Conv set 1
                 self.conv1_1 = nn.Conv2d(3,depth_1,3,stride = 1,padding = 1)
                 self.conv1_2 = nn.Conv2d(depth_1,depth_1,3,stride = 1,padding = 1)
                 self.bn1_1 = nn.BatchNorm2d(depth_1)
                 self.bn1_2 = nn.BatchNorm2d(depth_1)
                 # Conv set 2
                 self.conv2_1 = nn.Conv2d(depth_1,depth_2,3,stride = 1,padding = 1)
                 self.conv2_2 = nn.Conv2d(depth_2,depth_2,3,stride = 1,padding = 1)
```

```
        self.bn2_1 = nn.BatchNorm2d(depth_2)
        self.bn2_2 = nn.BatchNorm2d(depth_2)
        # Conv set 3
        self.conv3_1 = nn.Conv2d(depth_2,depth_3,3,stride = 1,padding = 1)
        self.conv3_2 = nn.Conv2d(depth_3,depth_3,3,stride = 1,padding = 1)
        self.bn3_1 = nn.BatchNorm2d(depth_3)
        self.bn3_2 = nn.BatchNorm2d(depth_3)
        # Output
        self.fc_out = nn.Linear(depth_3,num_classes)
        # Initialize weights
        nn.init.kaiming_normal_(self.conv1_1.weight, nonlinearity='relu')
        nn.init.kaiming_normal_(self.conv1_2.weight, nonlinearity='relu')
        nn.init.kaiming_normal_(self.conv2_1.weight, nonlinearity='relu')
        nn.init.kaiming_normal_(self.conv2_2.weight, nonlinearity='relu')
        nn.init.kaiming_normal_(self.conv3_1.weight, nonlinearity='relu')
        nn.init.kaiming_normal_(self.conv3_2.weight, nonlinearity='relu')

    def forward(self, x):
        # Conv 1
        x = F.relu(self.bn1_1(self.conv1_1(x)))
        x = F.relu(self.bn1_2(self.conv1_2(x)))
        x = self.pool(x)
        # Conv 2
        x = F.relu(self.bn2_1(self.conv2_1(x)))
        x = F.relu(self.bn2_2(self.conv2_2(x)))
        x = self.pool(x)
        # Conv 3
        x = F.relu(self.bn3_1(self.conv3_1(x)))
        x = F.relu(self.bn3_2(self.conv3_2(x)))
        x = self.pool(x)
        # First we fuse the height and width dimensions (2 and 3)
        x = x.view(x.size(0),x.size(1),-1)
        # And now max global pooling
        x = x.max(2)[0]
        # Output
        x = self.fc_out(x)
        return x

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

# instantiate the CNN
model_scratch = Net(num_classes)

# 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: Batch Normalization 2D: It is a technique to provide any laer in a Neural Network with inputs that are zero mean or unit variance

Convolutional Layers -> (Input channels, Output channels):

```
self.conv1_1 = nn.Conv2d(3,depth_1,3,stride = 1,padding = 1)
self.conv1_2 = nn.Conv2d(depth_1,depth_1,3,stride = 1,padding = 1)

self.conv2_1 = nn.Conv2d(depth_1,depth_2,3,stride = 1,padding = 1)
self.conv2_2 = nn.Conv2d(depth_2,depth_2,3,stride = 1,padding = 1)

self.conv3_1 = nn.Conv2d(depth_2,depth_3,3,stride = 1,padding = 1)
self.conv3_2 = nn.Conv2d(depth_3,depth_3,3,stride = 1,padding = 1)
```

1 Max Pool Layer

```
self.max_pool = nn.MaxPool2d(2,2)
```

I also used kaiming intialization

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 [20]: import torch.optim as optim

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

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

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

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

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

    optimizer.zero_grad()
    output = model(data)
    loss = criterion_scratch(output,target)
    loss.backward()
    optimizer.step()
    train_loss += loss.item()*data.size(0)

######################
# 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_scratch(output,target)
    valid_loss += loss.item()*data.size(0)

train_loss = train_loss/len(train_loader.dataset)
valid_loss = valid_loss/len(val_loader.dataset)


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

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

```
            # return trained model
            return model


        # train the model
        model_scratch = train(100, loaders_scratch, model_scratch, optimizer_scratch,
                              criterion_scratch, use_cuda, 'model_scratch.pt')

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

```
Epoch: 1         Training Loss: 4.943218         Validation Loss: 4.869157
Validation loss decreased (inf --> 4.869157).    Saving model...
Epoch: 2         Training Loss: 4.839848         Validation Loss: 4.811384
Validation loss decreased (4.869157 --> 4.811384).    Saving model...
Epoch: 3         Training Loss: 4.763176         Validation Loss: 4.732584
Validation loss decreased (4.811384 --> 4.732584).    Saving model...
Epoch: 4         Training Loss: 4.701967         Validation Loss: 4.678071
Validation loss decreased (4.732584 --> 4.678071).    Saving model...
Epoch: 5         Training Loss: 4.640558         Validation Loss: 4.622224
Validation loss decreased (4.678071 --> 4.622224).    Saving model...
Epoch: 6         Training Loss: 4.596980         Validation Loss: 4.566811
Validation loss decreased (4.622224 --> 4.566811).    Saving model...
Epoch: 7         Training Loss: 4.557859         Validation Loss: 4.522739
Validation loss decreased (4.566811 --> 4.522739).    Saving model...
Epoch: 8         Training Loss: 4.517121         Validation Loss: 4.482687
Validation loss decreased (4.522739 --> 4.482687).    Saving model...
Epoch: 9         Training Loss: 4.473934         Validation Loss: 4.457240
Validation loss decreased (4.482687 --> 4.457240).    Saving model...
Epoch: 10         Training Loss: 4.433447         Validation Loss: 4.371790
Validation loss decreased (4.457240 --> 4.371790).    Saving model...
Epoch: 11         Training Loss: 4.397140         Validation Loss: 4.346061
Validation loss decreased (4.371790 --> 4.346061).    Saving model...
Epoch: 12         Training Loss: 4.349878         Validation Loss: 4.329629
Validation loss decreased (4.346061 --> 4.329629).    Saving model...
Epoch: 13         Training Loss: 4.316680         Validation Loss: 4.285714
Validation loss decreased (4.329629 --> 4.285714).    Saving model...
Epoch: 14         Training Loss: 4.256686         Validation Loss: 4.237934
Validation loss decreased (4.285714 --> 4.237934).    Saving model...
Epoch: 15         Training Loss: 4.223526         Validation Loss: 4.257216
Epoch: 16         Training Loss: 4.188722         Validation Loss: 4.157389
Validation loss decreased (4.237934 --> 4.157389).    Saving model...
Epoch: 17         Training Loss: 4.133028         Validation Loss: 4.154134
Validation loss decreased (4.157389 --> 4.154134).    Saving model...
Epoch: 18         Training Loss: 4.105645         Validation Loss: 4.025022
Validation loss decreased (4.154134 --> 4.025022).    Saving model...
Epoch: 19         Training Loss: 4.051988         Validation Loss: 4.086281
```

```
Epoch: 20         Training Loss: 4.035077         Validation Loss: 3.970492
Validation loss decreased (4.025022 --> 3.970492).    Saving model...
Epoch: 21         Training Loss: 3.988084         Validation Loss: 3.986096
Epoch: 22         Training Loss: 3.978823         Validation Loss: 3.906524
Validation loss decreased (3.970492 --> 3.906524).    Saving model...
Epoch: 23         Training Loss: 3.932647         Validation Loss: 3.949496
Epoch: 24         Training Loss: 3.900060         Validation Loss: 3.841688
Validation loss decreased (3.906524 --> 3.841688).    Saving model...
Epoch: 25         Training Loss: 3.870670         Validation Loss: 3.835120
Validation loss decreased (3.841688 --> 3.835120).    Saving model...
Epoch: 26         Training Loss: 3.836970         Validation Loss: 3.825278
Validation loss decreased (3.835120 --> 3.825278).    Saving model...
Epoch: 27         Training Loss: 3.833037         Validation Loss: 3.905061
Epoch: 28         Training Loss: 3.820469         Validation Loss: 3.779954
Validation loss decreased (3.825278 --> 3.779954).    Saving model...
Epoch: 29         Training Loss: 3.778704         Validation Loss: 3.692588
Validation loss decreased (3.779954 --> 3.692588).    Saving model...
Epoch: 30         Training Loss: 3.760223         Validation Loss: 3.708528
Epoch: 31         Training Loss: 3.732917         Validation Loss: 3.698874
Epoch: 32         Training Loss: 3.723491         Validation Loss: 3.729954
Epoch: 33         Training Loss: 3.684220         Validation Loss: 3.637979
Validation loss decreased (3.692588 --> 3.637979).    Saving model...
Epoch: 34         Training Loss: 3.677728         Validation Loss: 3.673351
Epoch: 35         Training Loss: 3.666886         Validation Loss: 3.636609
Validation loss decreased (3.637979 --> 3.636609).    Saving model...
Epoch: 36         Training Loss: 3.640089         Validation Loss: 3.615474
Validation loss decreased (3.636609 --> 3.615474).    Saving model...
Epoch: 37         Training Loss: 3.611876         Validation Loss: 3.554966
Validation loss decreased (3.615474 --> 3.554966).    Saving model...
Epoch: 38         Training Loss: 3.594562         Validation Loss: 3.494895
Validation loss decreased (3.554966 --> 3.494895).    Saving model...
Epoch: 39         Training Loss: 3.580281         Validation Loss: 3.540882
Epoch: 40         Training Loss: 3.539222         Validation Loss: 3.498809
Epoch: 41         Training Loss: 3.533095         Validation Loss: 3.523072
Epoch: 42         Training Loss: 3.533053         Validation Loss: 3.582003
Epoch: 43         Training Loss: 3.489877         Validation Loss: 3.533021
Epoch: 44         Training Loss: 3.463930         Validation Loss: 3.546593
Epoch: 45         Training Loss: 3.471195         Validation Loss: 3.468438
Validation loss decreased (3.494895 --> 3.468438).    Saving model...
Epoch: 46         Training Loss: 3.445878         Validation Loss: 3.366381
Validation loss decreased (3.468438 --> 3.366381).    Saving model...
Epoch: 47         Training Loss: 3.420483         Validation Loss: 3.424762
Epoch: 48         Training Loss: 3.399203         Validation Loss: 3.385672
Epoch: 49         Training Loss: 3.400352         Validation Loss: 3.452099
Epoch: 50         Training Loss: 3.378854         Validation Loss: 3.388642
Epoch: 51         Training Loss: 3.357808         Validation Loss: 3.303061
Validation loss decreased (3.366381 --> 3.303061).    Saving model...
Epoch: 52         Training Loss: 3.304674         Validation Loss: 3.312835
```

```
Epoch: 53        Training Loss: 3.309182        Validation Loss: 3.289970
Validation loss decreased (3.303061 --> 3.289970).   Saving model...
Epoch: 54        Training Loss: 3.301121        Validation Loss: 3.246204
Validation loss decreased (3.289970 --> 3.246204).   Saving model...
Epoch: 55        Training Loss: 3.266731        Validation Loss: 3.251697
Epoch: 56        Training Loss: 3.264378        Validation Loss: 3.321580
Epoch: 57        Training Loss: 3.260893        Validation Loss: 3.351787
Epoch: 58        Training Loss: 3.229588        Validation Loss: 3.305263
Epoch: 59        Training Loss: 3.225798        Validation Loss: 3.270953
Epoch: 60        Training Loss: 3.207409        Validation Loss: 3.226351
Validation loss decreased (3.246204 --> 3.226351).   Saving model...
Epoch: 61        Training Loss: 3.184018        Validation Loss: 3.197380
Validation loss decreased (3.226351 --> 3.197380).   Saving model...
Epoch: 62        Training Loss: 3.157944        Validation Loss: 3.306705
Epoch: 63        Training Loss: 3.138006        Validation Loss: 3.248719
Epoch: 64        Training Loss: 3.147142        Validation Loss: 3.150096
Validation loss decreased (3.197380 --> 3.150096).   Saving model...
Epoch: 65        Training Loss: 3.114355        Validation Loss: 3.297082
Epoch: 66        Training Loss: 3.123967        Validation Loss: 3.118178
Validation loss decreased (3.150096 --> 3.118178).   Saving model...
Epoch: 67        Training Loss: 3.080215        Validation Loss: 3.055447
Validation loss decreased (3.118178 --> 3.055447).   Saving model...
Epoch: 68        Training Loss: 3.058915        Validation Loss: 3.073874
Epoch: 69        Training Loss: 3.046066        Validation Loss: 3.077153
Epoch: 70        Training Loss: 3.061198        Validation Loss: 3.128662
Epoch: 71        Training Loss: 3.032474        Validation Loss: 3.094101
Epoch: 72        Training Loss: 3.013408        Validation Loss: 3.111464
Epoch: 73        Training Loss: 3.017143        Validation Loss: 3.000041
Validation loss decreased (3.055447 --> 3.000041).   Saving model...
Epoch: 74        Training Loss: 2.997061        Validation Loss: 2.961633
Validation loss decreased (3.000041 --> 2.961633).   Saving model...
Epoch: 75        Training Loss: 2.975124        Validation Loss: 2.982455
Epoch: 76        Training Loss: 2.965183        Validation Loss: 3.003348
Epoch: 77        Training Loss: 2.934093        Validation Loss: 2.987402
Epoch: 78        Training Loss: 2.950453        Validation Loss: 2.973889
Epoch: 79        Training Loss: 2.904710        Validation Loss: 2.907919
Validation loss decreased (2.961633 --> 2.907919).   Saving model...
Epoch: 80        Training Loss: 2.925716        Validation Loss: 2.958429
Epoch: 81        Training Loss: 2.893911        Validation Loss: 2.981297
Epoch: 82        Training Loss: 2.898709        Validation Loss: 2.909172
Epoch: 83        Training Loss: 2.896990        Validation Loss: 2.926434
Epoch: 84        Training Loss: 2.859048        Validation Loss: 2.899319
Validation loss decreased (2.907919 --> 2.899319).   Saving model...
Epoch: 85        Training Loss: 2.825108        Validation Loss: 2.911480
Epoch: 86        Training Loss: 2.832081        Validation Loss: 2.899457
Epoch: 87        Training Loss: 2.817196        Validation Loss: 2.878673
Validation loss decreased (2.899319 --> 2.878673).   Saving model...
Epoch: 88        Training Loss: 2.842327        Validation Loss: 2.917044
```

```
Epoch: 89         Training Loss: 2.827874         Validation Loss: 2.855609
Validation loss decreased (2.878673 --> 2.855609).    Saving model...
Epoch: 90         Training Loss: 2.810314         Validation Loss: 2.866723
Epoch: 91         Training Loss: 2.776347         Validation Loss: 2.957351
Epoch: 92         Training Loss: 2.762933         Validation Loss: 2.800294
Validation loss decreased (2.855609 --> 2.800294).    Saving model...
Epoch: 93         Training Loss: 2.775220         Validation Loss: 2.787567
Validation loss decreased (2.800294 --> 2.787567).    Saving model...
Epoch: 94         Training Loss: 2.765478         Validation Loss: 2.809963
Epoch: 95         Training Loss: 2.742064         Validation Loss: 2.816903
Epoch: 96         Training Loss: 2.740799         Validation Loss: 2.818954
Epoch: 97         Training Loss: 2.757551         Validation Loss: 2.737043
Validation loss decreased (2.787567 --> 2.737043).    Saving model...
Epoch: 98         Training Loss: 2.691305         Validation Loss: 2.729866
Validation loss decreased (2.737043 --> 2.729866).    Saving model...
Epoch: 99         Training Loss: 2.693714         Validation Loss: 2.749711
Epoch: 100         Training Loss: 2.696266         Validation Loss: 2.771648
```

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 [21]: 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: 2.710234


Test Accuracy: 31% (261/836)
```

## 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 [12]: ## TODO: Specify data loaders
         loaders_transfer = loaders_scratch
         print(loaders_transfer)
```

```
{'train': <torch.utils.data.dataloader.DataLoader object at 0x7faf88bc5320>, 'valid': <torch.uti
```

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

         ## TODO: Specify model architecture
         model_transfer = models.resnet50(pretrained=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, 101372256.19it/s]
```

```
In [14]: model_transfer
```

```
Out[14]: 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=
               (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
               (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
               (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
               (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
               )
             )
             (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=
               (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
               (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
               (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
               (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=
               (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
               (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
               (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
               (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
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
        (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
        (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
        )
      )
      (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
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
        (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
        (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
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
        (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
        (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
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
        (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
        (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
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
        (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_stat
        (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_stat
    )
  )
  (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
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (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_stat
    (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
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (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_stat
    (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
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (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_stat
    (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
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (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_stat
    (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
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (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_stat
    (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
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
        (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_stat
        (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_stat
        )
      )
      (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
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
        (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_stat
        (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
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
        (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_stat
        (relu): ReLU(inplace)
      )
    )
    (avgpool): AvgPool2d(kernel_size=7, stride=1, padding=0)
    (fc): Linear(in_features=2048, out_features=1000, 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:

It is very efficient to use pre-trained networks to solve challenging problems in computer vision.

Once trained, these models work very well as feature detectors for images they weren't trained on. Here we'll use transfer learning to train a network that can classify our dog photos.

For this task speciffically, I'll use resnet50 trained on ImageNet available from torchvision.

The classifier part of the model is a single fully-connected layer:

(fc): Linear(in_features=2048, out_features=1000, bias=True)

This layer was trained on the ImageNet dataset, so it won't work for the dog classification specific problem.

That means we need to replace the classifier (133 classes), but the features will work perfectly on their own.

Choice of criterion: nn.CrossEntropyLoss() This criterion combines :func:nn.LogSoftmax and :func:nn.NLLLoss in one single class. It is useful when training a classification problem with C classes.

```
In [15]: # Freeze parameters so we don't backprop through them
         for param in model_transfer.parameters():
             param.requires_grad = False
         # Replace the last fully connected layer with a Linnear layer with 133 out features
         model_transfer.fc = nn.Linear(2048, 133)
         nn.init.kaiming_normal_(model_transfer.fc.weight, nonlinearity='relu')
         if use_cuda:
             model_transfer = model_transfer.cuda()
```

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 [17]: import torch.optim as optim

         criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.Adam(model_transfer.fc.parameters(), lr=0.001)
```

1.1.15 (IMPLEMENTATION) Train and Validate the Model

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

```
In [21]: def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
             """returns trained model"""
             # initialize tracker for minimum validation loss
             valid_loss_min = np.Inf

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

                 ###################
                 # train the model #
                 ###################
                 model.train()
                 for batch_idx, (data, target) in enumerate(loaders['train']):
                     # move to GPU
```

```
                if use_cuda:
                    data, target = data.cuda(), target.cuda()
                ## find the loss and update the model parameters accordingly
                ## record the average training loss, using something like
                ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_lo

                optimizer.zero_grad()
                output = model(data)
                loss = criterion_scratch(output,target)
                loss.backward()
                optimizer.step()
                train_loss += loss.item()*data.size(0)

            ######################
            # 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_scratch(output,target)
                valid_loss += loss.item()*data.size(0)

            train_loss = train_loss/len(train_loader.dataset)
            valid_loss = valid_loss/len(val_loader.dataset)


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

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

        # return trained model
        return model
```

In [22]:
```
# train the model
model_transfer = train(20, loaders_transfer, model_transfer, optimizer_transfer, criter
    # 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: 3.384301         Validation Loss: 1.123161
Validation loss decreased (inf --> 1.123161).    Saving model...
Epoch: 2         Training Loss: 2.017018         Validation Loss: 0.900304
Validation loss decreased (1.123161 --> 0.900304).    Saving model...
Epoch: 3         Training Loss: 1.785830         Validation Loss: 0.672333
Validation loss decreased (0.900304 --> 0.672333).    Saving model...
Epoch: 4         Training Loss: 1.681943         Validation Loss: 0.642487
Validation loss decreased (0.672333 --> 0.642487).    Saving model...
Epoch: 5         Training Loss: 1.533681         Validation Loss: 0.712035
Epoch: 6         Training Loss: 1.548654         Validation Loss: 0.653631
Epoch: 7         Training Loss: 1.505691         Validation Loss: 0.622015
Validation loss decreased (0.642487 --> 0.622015).    Saving model...
Epoch: 8         Training Loss: 1.443167         Validation Loss: 0.588621
Validation loss decreased (0.622015 --> 0.588621).    Saving model...
Epoch: 9         Training Loss: 1.424833         Validation Loss: 0.574699
Validation loss decreased (0.588621 --> 0.574699).    Saving model...
Epoch: 10         Training Loss: 1.392046         Validation Loss: 0.641413
Epoch: 11         Training Loss: 1.407750         Validation Loss: 0.673414
Epoch: 12         Training Loss: 1.396201         Validation Loss: 0.600620
Epoch: 13         Training Loss: 1.337806         Validation Loss: 0.663416
Epoch: 14         Training Loss: 1.332503         Validation Loss: 0.594167
Epoch: 15         Training Loss: 1.334946         Validation Loss: 0.674096
Epoch: 16         Training Loss: 1.315096         Validation Loss: 0.604902
Epoch: 17         Training Loss: 1.348075         Validation Loss: 0.626031
Epoch: 18         Training Loss: 1.291242         Validation Loss: 0.638766
Epoch: 19         Training Loss: 1.285940         Validation Loss: 0.675343
Epoch: 20         Training Loss: 1.292737         Validation Loss: 0.648361
```

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 [24]: 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 [25]: test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.570940


Test Accuracy: 83% (698/836)
```

1.1.17 (IMPLEMENTATION) Predict Dog Breed with the Model

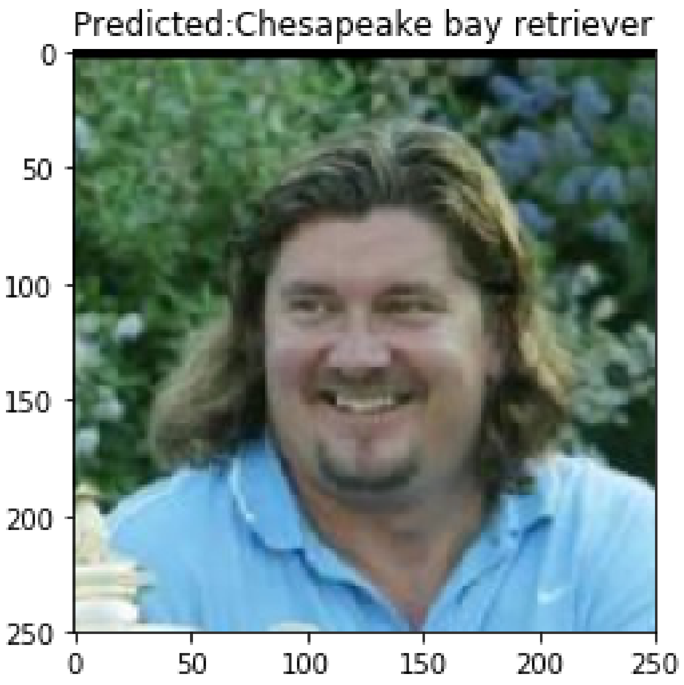
Write a function that takes an image path as input and returns the dog breed (`Affenpinscher`, `Afghan hound`, etc) that is predicted by your model.

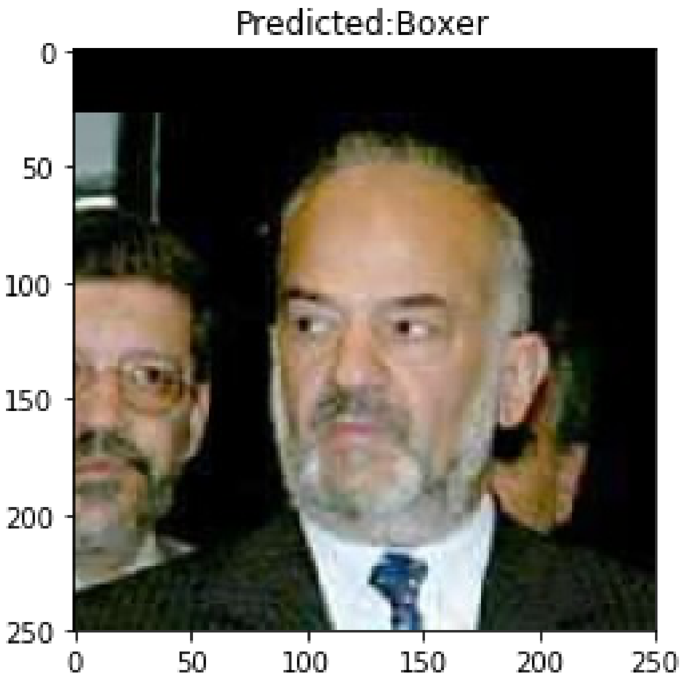
```
In [63]: ### TODO: Write a function that takes a path to an image as input
         ### and returns the dog breed that is predicted by the model.


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


         model_transfer.load_state_dict(torch.load('model_transfer.pt'))

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

             # Define normalization step for image
             normalize = transforms.Normalize(mean=(0.485, 0.456, 0.406),
                                              std=(0.229, 0.224, 0.225))
```

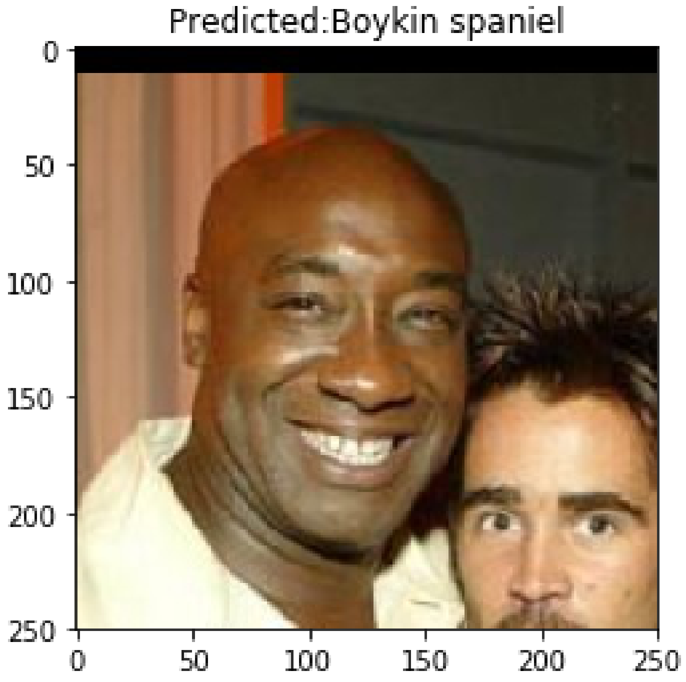
```
        # Define transformations of image
        preprocess = transforms.Compose([transforms.Resize(258),
                                         transforms.CenterCrop(224),
                                         transforms.ToTensor(),
                                         normalize])

        # Preprocess image to 4D Tensor (.unsqueeze(0) adds a dimension)
        img_tensor = preprocess(img).unsqueeze_(0)

        # Move tensor to GPU if available
        if use_cuda:
            img_tensor = img_tensor.cuda()

        ## Inference
        # Turn on evaluation mode
        model_transfer.eval()

        # Get predicted category for image
        with torch.no_grad():
            output = model_transfer(img_tensor)
            prediction = torch.argmax(output).item()

        # Turn off evaluation mode
        model_transfer.train()

        # Use prediction to get dog breed
        breed = class_names[prediction]

        return breed
```

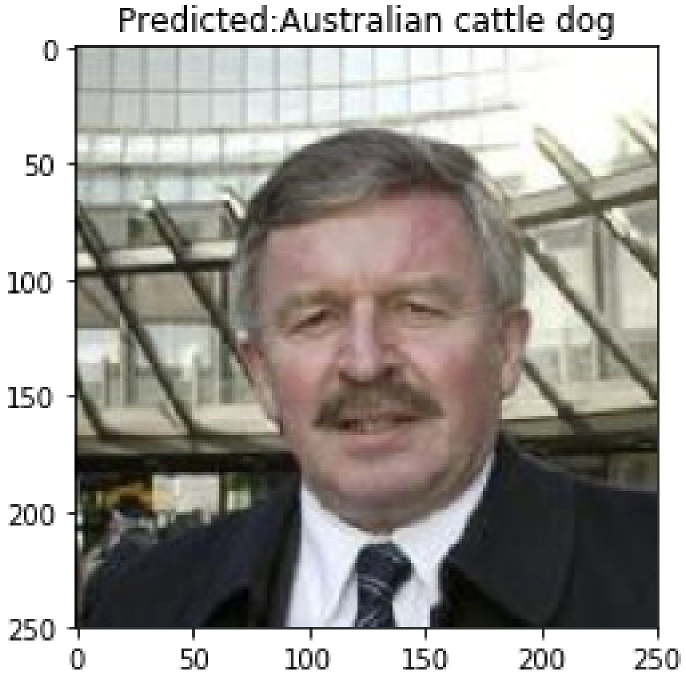
```
In [64]: def display_image(img_path, title="Title"):
             image = Image.open(img_path)
             plt.title(title)
             plt.imshow(image)
             plt.show()
```

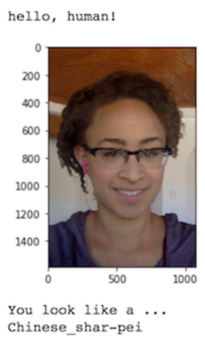
```
In [65]: import random

         # Try out the function
         for image in random.sample(list(human_files_short), 4):
             predicted_breed = predict_breed_transfer(image)
             display_image(image, title=f"Predicted:{predicted_breed}")
```


Predicted:Chesapeake bay retriever
0
50
100
150
200
250
0
50
100
150
200
250


Predicted:Boxer
0
50
100
150
200
250
0
50
100
150
200
250


Predicted:Boykin spaniel
0
50
100
150
200
250
0
50
100
150
200
250


Predicted:Australian cattle dog
0
50
100
150
200
250
0
50
100
150
200
250



```
hello, human!
```

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

Sample Human Output

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

         def run_app(img_path):
             # check if image has human faces:
             if (face_detector(img_path)):
                 print("Hello Human!")
                 predicted_breed = predict_breed_transfer(img_path)
                 display_image(img_path, title=f"Predicted:{predicted_breed}")

                 print("You look like a ...")
                 print(predicted_breed.upper())
             # check if image has dogs:
             elif dog_detector(img_path):
                 print("Hello Doggie!")
                 predicted_breed = predict_breed_transfer(img_path)
                 display_image(img_path, title=f"Predicted:{predicted_breed}")

                 print("Your breed is ...")
                 print(predicted_breed.upper())
             else:
                 print("We couldn't detect any dog or human face")
```

```
            display_image(img_path, title="...")
            print("Try another!")
        print("\n")
```

## Step 6: Test Your Algorithm

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

1.1.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

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

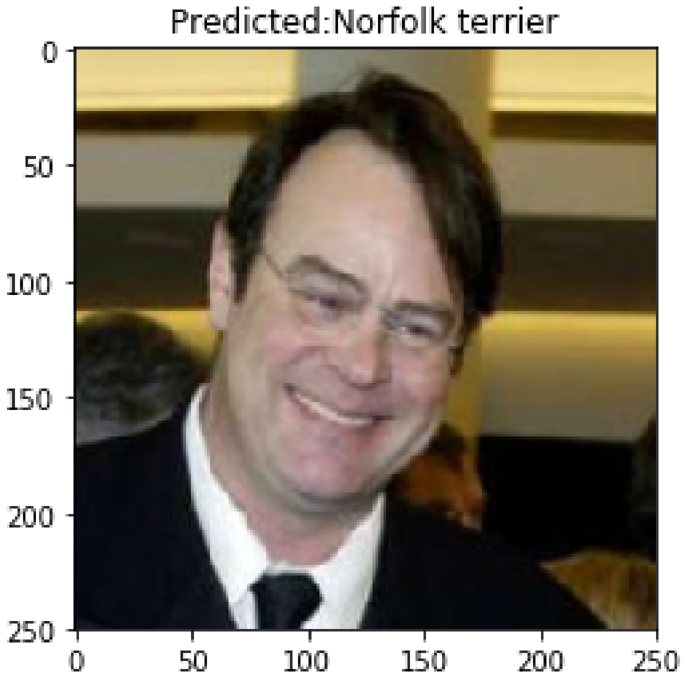
Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer: (Three possible points for improvement)

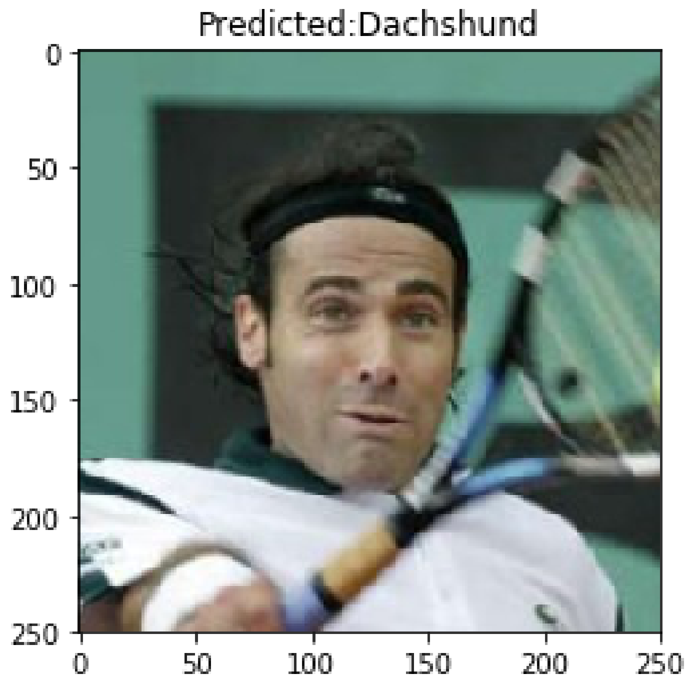
- The major point of improvement is in the two detectors. I think that for a flawless user experience the app absolutely cannot fail when it comes to detecting whether a dog is even in the picture or not. If it can't even do that, the user would think, how can it possibly distinguish dog breeds? But the true positive rate is not the only relevant matter here. If the app would confuse a human or a landscape image with a dog it would also be a pretty bad sign. These points probably don't apply as much to the human detector since that's not the main goal but rather a fun quirk. What is then needed is a large dataset with classes "dog", "human", and "other". The easiest way to go would still be to train two detectors, one for dogs and one for humans, to avoid dealing with situations when both a dog and a human are present in the same photo. Since we have both dog and human images already available, all we'd need would be the "other" class, which should contain images of as many things as possible, in addition to dogs and humans. Lots of images of mammals other than dogs would certainly help.
- The transfer learning model we use to predict breeds can probably be substantially improved. The learning rate I used might not be optimal, the model could be trained for more epochs, maybe with a smarter learning rate scheduler. We could unfreeze the conv layers and fine-tune them a bit, maybe with fastai's approach of differential learning rates where we use larger learning rates the closer we are to the network's output. I don't think the model itself will matter too much but we could also try other, possibly deeper architectures. Since there is no sign of overfitting reducing some of the augmentation could also be an option, for example by increasing the first value of the "scale" parameter of RandomResizedCrop.
- Maybe a generative adversarial network could be used as a dog detector? Just a thought though, I have yet to really learn about them.

```
In [69]: ## 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)
```

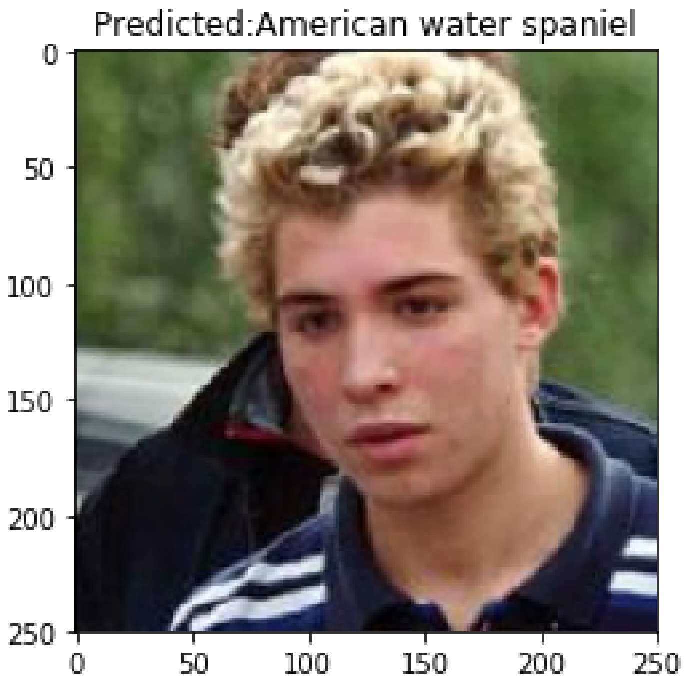
```
Hello Human!
```



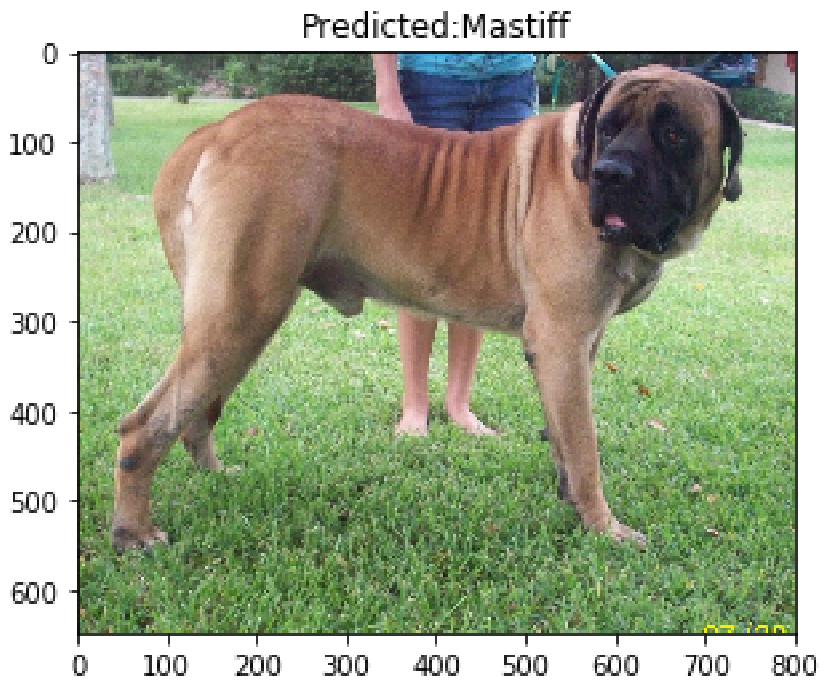
```
You look like a ...
NORFOLK TERRIER


Hello Human!
```



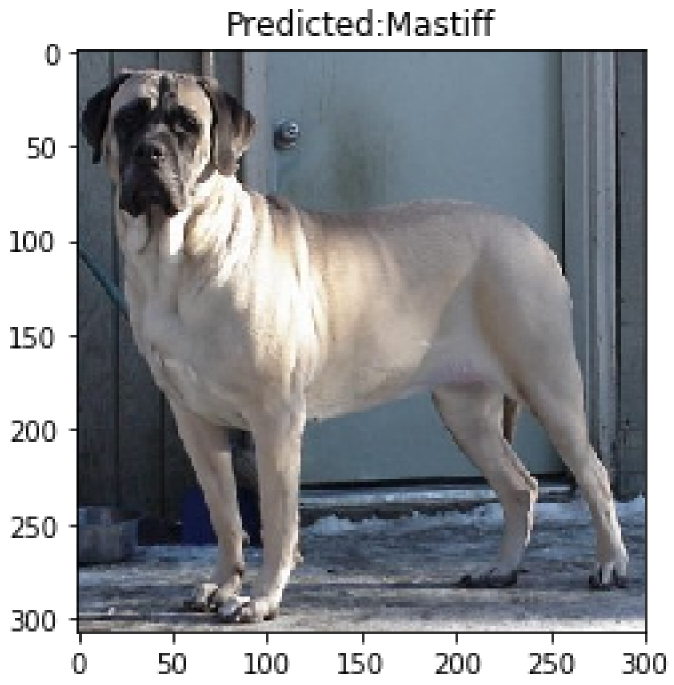
```
You look like a ...
DACHSHUND


Hello Human!
```



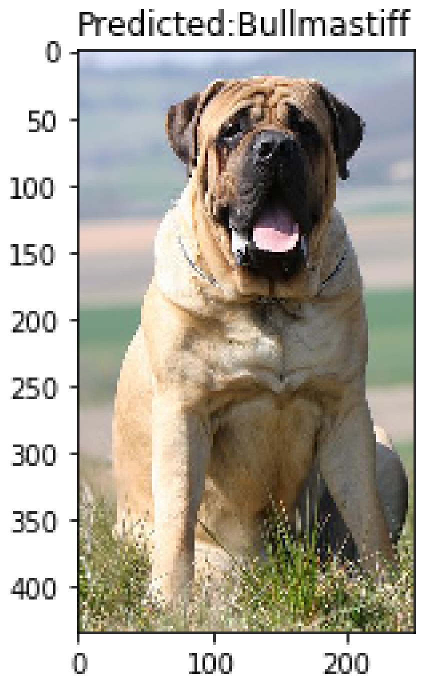
```
You look like a ...
AMERICAN WATER SPANIEL


Hello Doggie!
```



```
Your breed is ...
MASTIFF


Hello Doggie!
```



```
Your breed is ...
MASTIFF


Hello Doggie!
```



```
Your breed is ...
BULLMASTIFF
```

```
In [ ]:
```

```
In [ ]:
```