dog_app

February 27, 2020

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 `/dogImages`.

- 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`.

[2]:
```
import numpy as np
from glob import glob

# load filenames for human and dog images
human_files = np.array(glob("lfw/*/*"))
dog_files = np.array(glob("dogImages/*/*/*"))

# 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

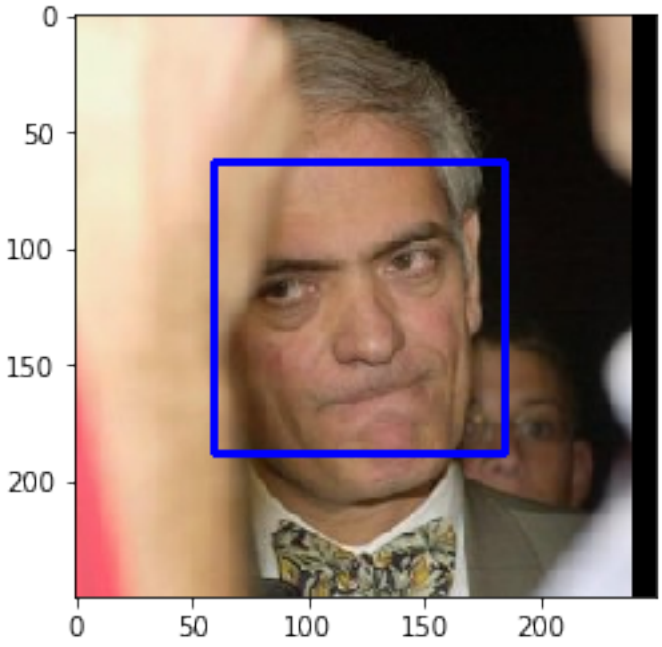
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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.

[3]:
```
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.

[4]:
```
# returns "True" if face is detected in image stored at img_path
def face_detector(img_path):
    img = cv2.imread(img_path)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    faces = face_cascade.detectMultiScale(gray)
    return len(faces) > 0
```

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

Question 1: Use the code cell below to test the performance of the `face_detector` function.
- What percentage of the first 100 images in `human_files` have a detected human face?
- What percentage of the first 100 images in `dog_files` have a detected human face?

Ideally, we would like 100% of human images with a detected face and 0% of dog images with a detected face. You will see that our algorithm falls short of this goal, but still gives acceptable performance. We extract the file paths for the first 100 images from each of the datasets and store them in the numpy arrays `human_files_short` and `dog_files_short`.

Answer: - Correct human detected: 100 % - Wrong human detected: 15 %

[5]:
```
from tqdm import tqdm

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

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

## TODO: Test the performance of the face_detector algorithm
## on the images in human_files_short and dog_files_short.

correct_human_detected = 0
for h in human_files_short:
    if face_detector(h):
        correct_human_detected += 1

wrong_human_detected = 0
for d in dog_files_short:
    if face_detector(d):
        wrong_human_detected += 1

print("Correct human detected: ", correct_human_detected, "%")
print("Wrong human detected: ", wrong_human_detected, "%")
```

```
Correct human detected:  100 %
Wrong human detected:  15 %
```

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`.

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

Step 2: Detect Dogs

In this section, we use a pre-trained model to detect dogs in images.

1.1.3 Obtain Pre-trained VGG-16 Model

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

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

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

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

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

```
Use gpu
```

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.

[5]:
```
VGG16
```

[5]:
```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
```

```
    (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=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (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=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (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=True)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (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=True)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace=True)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

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.

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

# Set PIL to be tolerant of image files that are truncated.
from PIL import ImageFile
from PIL import Image
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
    '''

    ## 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

    transform = transforms.Compose([transforms.Resize((224, 224)),
                                    transforms.ToTensor()])

    im = Image.open(img_path)

    img = transform(im)
    img = img.unsqueeze(0)
    if use_cuda:
        img = img.cuda()

    #print(img.shape)

    preds = VGG16(img)
    indx = torch.argmax(preds, dim=1)
```

```
    #print(indx)

    return indx # predicted class index
```

[45]:
```
indx = VGG16_predict('dogImages/train/001.Affenpinscher/Affenpinscher_00001.
↪jpg')
```

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).

[8]:
```
### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    ## TODO: Complete the function.

    indx = VGG16_predict(img_path)

    return True if indx >= 151 and indx <= 268 else False # true/false
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

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

Answer: - Correct dog detected: 94 % - Wrong dog detected: 0 %

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

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

correct_dog_detected = 0
for d in dog_files_short:
    if dog_detector(d):
        correct_dog_detected += 1
```

```
wrong_dog_detected = 0
for h in human_files_short:
    if dog_detector(h):
        wrong_dog_detected += 1


print("Correct dog detected: ", correct_dog_detected, "%")
print("Wrong dog detected: ", wrong_dog_detected, "%")
```

```
Correct dog detected:  94 %
Wrong dog detected:  0 %
```

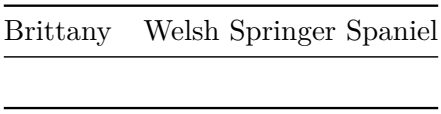
We suggest VGG-16 as a potential network to detect dog images in your algorithm, but you are free to explore other pre-trained networks (such as Inception-v3, ResNet-50, etc). Please use the code cell below to test other pre-trained PyTorch models. If you decide to pursue this *optional* task, report performance on `human_files_short` and `dog_files_short`.

[ ]:
```
### (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)

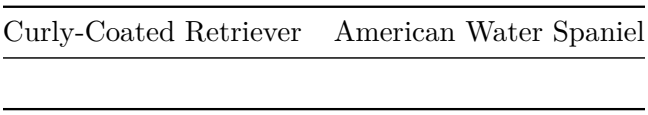
Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of 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

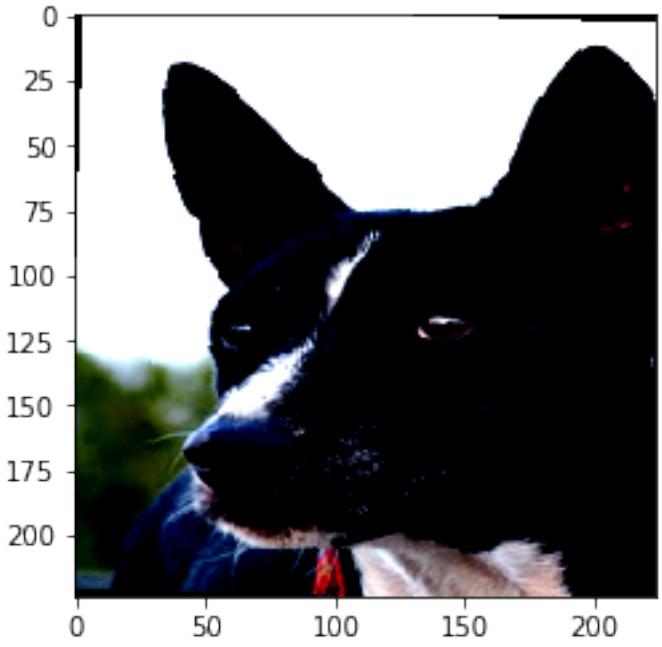
Use the code cell below to write three separate data loaders for the training, validation, and test datasets of dog images (located at `dogImages/train`, `dogImages/valid`, and `dogImages/test`, respectively). You may find this documentation on custom datasets to be a useful resource. If you are interested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

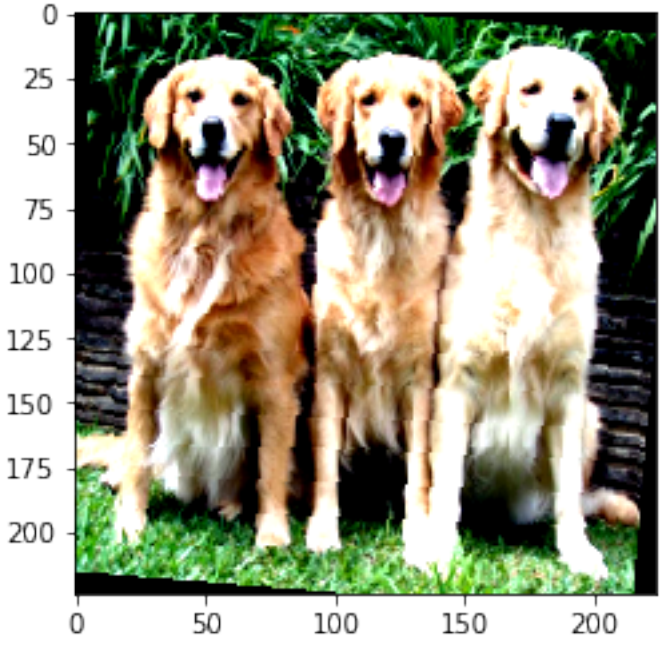
[9]:
```
import os
from torchvision import datasets, transforms


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

transform = transforms.Compose([
                                transforms.Resize((250, 250)),
                                transforms.CenterCrop(224),
                                transforms.RandomHorizontalFlip(),
                                transforms.RandomRotation(5),
                                transforms.ToTensor(),
                                transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                     std=[0.229, 0.224, 0.225])
                                ])

test_transform = transforms.Compose([
                                     transforms.Resize((250, 250)),
                                     transforms.CenterCrop(224),
                                     transforms.ToTensor(),
                                     transforms.Normalize(mean=[0.485, 0.456, 0.
 ↪406],
                                                          std=[0.229, 0.224, 0.
 ↪225])])

train_dataset = datasets.ImageFolder('dogImages/train', transform=transform)
```

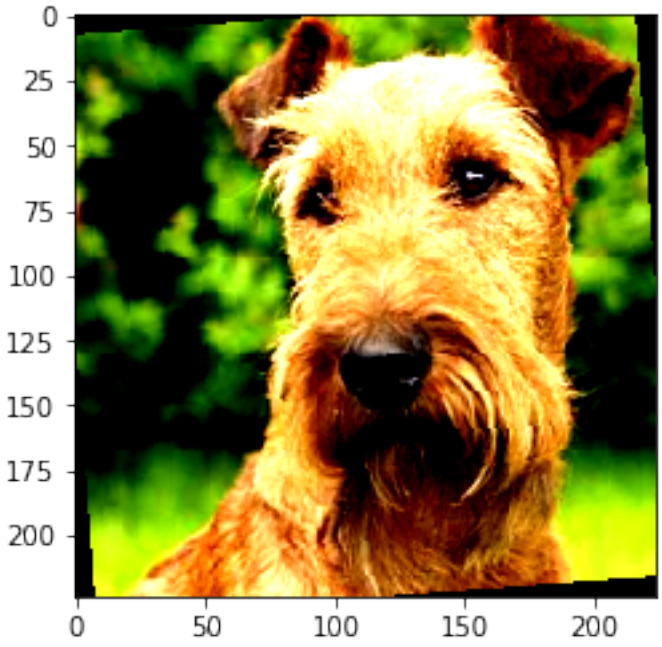
```
train_dataloader = torch.utils.data.DataLoader(train_dataset, batch_size=64,␣
 ↪shuffle=True)

valid_dataset = datasets.ImageFolder('dogImages/valid',␣
 ↪transform=test_transform)
valid_dataloader = torch.utils.data.DataLoader(valid_dataset, batch_size=64,␣
 ↪shuffle=True)

test_dataset = datasets.ImageFolder('dogImages/test', transform=test_transform)
test_dataloader = torch.utils.data.DataLoader(test_dataset, batch_size=64,␣
 ↪shuffle=True)

loaders_scratch = dict()
loaders_scratch['train'] = train_dataloader
loaders_scratch['valid'] = valid_dataloader
loaders_scratch['test'] = test_dataloader
```

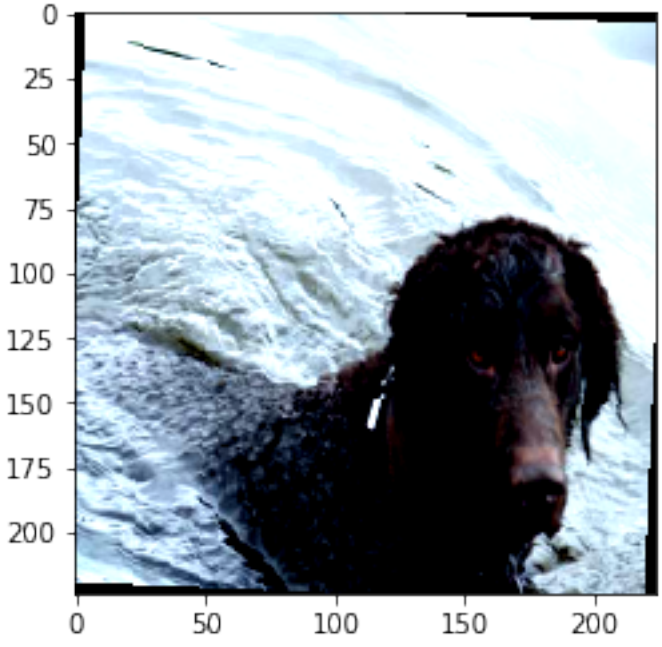
[115]:
```
import matplotlib.pyplot as plt
%matplotlib inline

def imshow(img):
    img = img / 2 + 0.5
    plt.imshow(np.transpose(img, (1, 2, 0)))

dataiter = iter(train_dataloader)
images, labels = dataiter.next()
images = images.numpy()

for idx in np.arange(5):
    fig = plt.figure()
    imshow(images[idx])
```

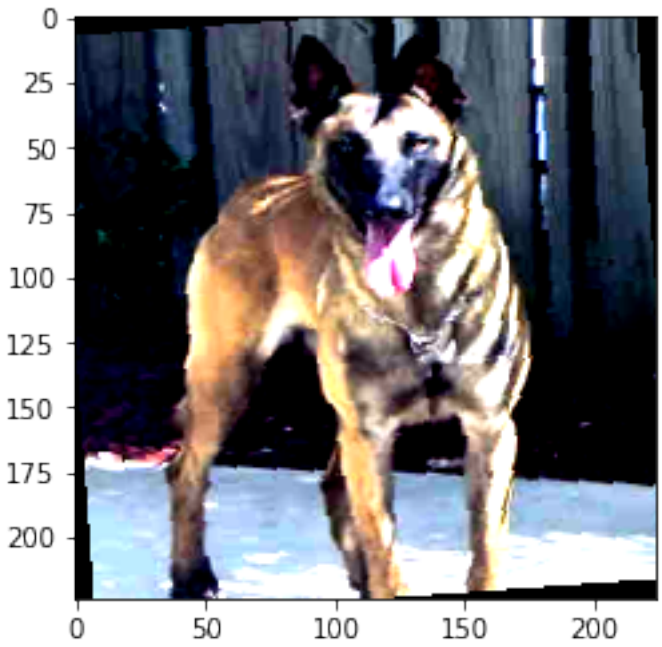
```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```


0
25
50
75
100
125
150
175
200
0
50
100
150
200


0
25
50
75
100
125
150
175
200
0
50
100
150
200


0
25
50
75
100
125
150
175
200
0
50
100
150
200


0
25
50
75
100
125
150
175
200
0
50
100
150
200



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 input size to tesor is 224x224 case pytorch model zoo has this size. Small size increase train speed but there is risk to lose some feature a CNN have to learn. Train data has different size so first I resize it to 250 and then crop to 224. Usually object we are interesting in somewhere next to center of image, so cropping help to focus at main object. To augment data I use random horizontal flipping and rotation. Last step is normalization, parameters I also took from pytorch documentation.

1.1.8 (IMPLEMENTATION) Model Architecture

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

[62]:
```
models.resnet18()
```

[62]:
```
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=True)
```

```
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
```

```
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
```

```
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=1000, bias=True)
)
```

[10]:
```python
import torch.nn as nn
import torch.nn.functional as F

# define the CNN architecture
class Net(nn.Module):
    ### TODO: choose an architecture, and complete the class
    def __init__(self):
        super(Net, self).__init__()
        ## Define layers of a CNN

        self.conv_0 = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3)
        self.bn_0 = nn.BatchNorm2d(64)
        self.rl_0 = nn.LeakyReLU(inplace=True)
        self.pool_0 = nn.MaxPool2d(3, stride=2, padding=1)

# layer 1
        self.conv_1_1_1 = nn.Conv2d(64, 64, kernel_size=3, padding=1)
        self.bn_1_1_1 = nn.BatchNorm2d(64)
        self.rl_1_1_1 = nn.LeakyReLU(inplace=True)
        self.conv_1_1_2 = nn.Conv2d(64, 64, kernel_size=3, padding=1)
        self.bn_1_1_2 = nn.BatchNorm2d(64)

        self.conv_1_2_1 = nn.Conv2d(64, 64, kernel_size=3, padding=1)
        self.bn_1_2_1 = nn.BatchNorm2d(64)
```

```
        self.rl_1_2_1 = nn.LeakyReLU(inplace=True)
        self.conv_1_2_2 = nn.Conv2d(64, 64, kernel_size=3, padding=1)
        self.bn_1_2_2 = nn.BatchNorm2d(64)

# layer 2
        #downsample
        self.conv_2_d = nn.Conv2d(64, 128, kernel_size=1, stride=2, bias=False)
        self.bn_2_d = nn.BatchNorm2d(128)

        self.conv_2_1_1 = nn.Conv2d(128, 128, kernel_size=3, padding=1)
        self.bn_2_1_1 = nn.BatchNorm2d(128)
        self.rl_2_1_1 = nn.LeakyReLU(inplace=True)
        self.conv_2_1_2 = nn.Conv2d(128, 128, kernel_size=3, padding=1)
        self.bn_2_1_2 = nn.BatchNorm2d(128)


        self.conv_2_2_1 = nn.Conv2d(128, 128, kernel_size=3, padding=1)
        self.bn_2_2_1 = nn.BatchNorm2d(128)
        self.rl_2_2_1 = nn.LeakyReLU(inplace=True)
        self.conv_2_2_2 = nn.Conv2d(128, 128, kernel_size=3, padding=1)
        self.bn_2_2_2 = nn.BatchNorm2d(128)

# layer 3
        #downsample
        self.conv_3_d = nn.Conv2d(128, 256, kernel_size=1, stride=2, bias=False)
        self.bn_3_d = nn.BatchNorm2d(256)

        self.conv_3_1_1 = nn.Conv2d(256, 256, kernel_size=3, padding=1)
        self.bn_3_1_1 = nn.BatchNorm2d(256)
        self.rl_3_1_1 = nn.LeakyReLU(inplace=True)
        self.conv_3_1_2 = nn.Conv2d(256, 256, kernel_size=3, padding=1)
        self.bn_3_1_2 = nn.BatchNorm2d(256)


        self.conv_3_2_1 = nn.Conv2d(256, 256, kernel_size=3, padding=1)
        self.bn_3_2_1 = nn.BatchNorm2d(256)
        self.rl_3_2_1 = nn.LeakyReLU(inplace=True)
        self.conv_3_2_2 = nn.Conv2d(256, 256, kernel_size=3, padding=1)
        self.bn_3_2_2 = nn.BatchNorm2d(256)

# layer 4
        self.conv_4_d = nn.Conv2d(256, 512, kernel_size=1, bias=False)
        self.bn_4_d = nn.BatchNorm2d(512)

        self.conv_4_1_1 = nn.Conv2d(512, 512, kernel_size=3, padding=1)
        self.bn_4_1_1 = nn.BatchNorm2d(512)
        self.rl_4_1_1 = nn.LeakyReLU(inplace=True)
```

```
        self.conv_4_1_2 = nn.Conv2d(512, 512, kernel_size=3, padding=1)
        self.bn_4_1_2 = nn.BatchNorm2d(512)


        self.conv_4_2_1 = nn.Conv2d(512, 512, kernel_size=3, padding=1)
        self.bn_4_2_1 = nn.BatchNorm2d(512)
        self.rl_4_2_1 = nn.LeakyReLU(inplace=True)
        self.conv_4_2_2 = nn.Conv2d(512, 512, kernel_size=3, padding=1)
        self.bn_4_2_2 = nn.BatchNorm2d(512)

        self.avg = nn.AdaptiveAvgPool2d(output_size=(1, 1))
        self.fc = nn.Linear(512, 133)


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

        x = self.conv_0(x)
        x = self.bn_0(x)
        x = self.rl_0(x)
        x = self.pool_0(x)

# layer 1
        cache = x

        x = self.conv_1_1_1(x)
        x = self.bn_1_1_1(x)
        x = self.rl_1_1_1(x)
        x = self.conv_1_1_2(x)
        x = self.bn_1_1_2(x)

        x = self.conv_1_2_1(x)
        x = self.bn_1_2_1(x)
        x = self.rl_1_2_1(x)
        x = self.conv_1_2_2(x)
        x = self.bn_1_2_2(x)

        x += cache
        x = F.leaky_relu(x)

# layer 2
        x = self.conv_2_d(x)
        x = self.bn_2_d(x)

        cache = x

        x = self.conv_2_1_1(x)
```

```
        x = self.bn_2_1_1(x)
        x = self.rl_2_1_1(x)
        x = self.conv_2_1_2(x)
        x = self.bn_2_1_2(x)

        x = self.conv_2_2_1(x)
        x = self.bn_2_2_1(x)
        x = self.rl_2_2_1(x)
        x = self.conv_2_2_2(x)
        x = self.bn_2_2_2(x)

        x += cache
        x = F.leaky_relu(x)

# layer 3
        x = self.conv_3_d(x)
        x = self.bn_3_d(x)

        cache = x

        x = self.conv_3_1_1(x)
        x = self.bn_3_1_1(x)
        x = self.rl_3_1_1(x)
        x = self.conv_3_1_2(x)
        x = self.bn_3_1_2(x)

        x = self.conv_3_2_1(x)
        x = self.bn_3_2_1(x)
        x = self.rl_3_2_1(x)
        x = self.conv_3_2_2(x)
        x = self.bn_3_2_2(x)

        x += cache
        x = F.leaky_relu(x)

# layer 4
        x = self.conv_4_d(x)
        x = self.bn_4_d(x)

        cache = x

        x = self.conv_4_1_1(x)
        x = self.bn_4_1_1(x)
        x = self.rl_4_1_1(x)
        x = self.conv_4_1_2(x)
        x = self.bn_4_1_2(x)
```

```
        x = self.conv_4_2_1(x)
        x = self.bn_4_2_1(x)
        x = self.rl_4_2_1(x)
        x = self.conv_4_2_2(x)
        x = self.bn_4_2_2(x)

        x += cache
        x = F.leaky_relu(x)

        x = self.avg(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)

        return x


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

# instantiate the CNN
model_scratch = Net()

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

[11]:
```
model_scratch
```

[11]:
```
Net(
  (conv_0): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3))
  (bn_0): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (rl_0): LeakyReLU(negative_slope=0.01, inplace=True)
  (pool_0): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
  (conv_1_1_1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
  (bn_1_1_1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (rl_1_1_1): LeakyReLU(negative_slope=0.01, inplace=True)
  (conv_1_1_2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
  (bn_1_1_2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (conv_1_2_1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
  (bn_1_2_1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
```

```
  (rl_1_2_1): LeakyReLU(negative_slope=0.01, inplace=True)
  (conv_1_2_2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
  (bn_1_2_2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (conv_2_d): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
  (bn_2_d): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (conv_2_1_1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
  (bn_2_1_1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (rl_2_1_1): LeakyReLU(negative_slope=0.01, inplace=True)
  (conv_2_1_2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
  (bn_2_1_2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (conv_2_2_1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
  (bn_2_2_1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (rl_2_2_1): LeakyReLU(negative_slope=0.01, inplace=True)
  (conv_2_2_2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
  (bn_2_2_2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (conv_3_d): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
  (bn_3_d): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (conv_3_1_1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
  (bn_3_1_1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (rl_3_1_1): LeakyReLU(negative_slope=0.01, inplace=True)
  (conv_3_1_2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
  (bn_3_1_2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (conv_3_2_1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
  (bn_3_2_1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (rl_3_2_1): LeakyReLU(negative_slope=0.01, inplace=True)
  (conv_3_2_2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
  (bn_3_2_2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
```

```
  (conv_4_d): Conv2d(256, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn_4_d): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (conv_4_1_1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
  (bn_4_1_1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (rl_4_1_1): LeakyReLU(negative_slope=0.01, inplace=True)
  (conv_4_1_2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
  (bn_4_1_2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (conv_4_2_1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
  (bn_4_2_1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (rl_4_2_1): LeakyReLU(negative_slope=0.01, inplace=True)
  (conv_4_2_2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
  (bn_4_2_2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (avg): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=133, bias=True)
)
```

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

Answer: I tried to implement rest18 architecture case, as I red, it was a breakthrough. The main idea is to simplify gradient flow (and traning process) having splitted layer at two parts: f(x) = h(x) + x. I used pytorch resnet18 as an example. After several experiments I replaced relu with leaky_relu, it increase accuracy to 10% just for 10 epochs.

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.

[12]:
```
import torch.optim as optim

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

### TODO: select optimizer
optimizer_scratch = optim.AdamW(model_scratch.parameters(), amsgrad=True)
```

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'`.

[11]:
```
# the following import is required for training to be robust to truncated images
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

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

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

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

            optimizer.zero_grad()

            output = model(data)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()

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

        ######################
        # validate the model #
        ######################
        model.eval()
        for batch_idx, (data, target) in enumerate(loaders['valid']):
```

```
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## update the average validation loss

            output = model(data)
            loss = criterion(output, target)

            valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data -␣
↪valid_loss))

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

        if valid_loss_min > valid_loss:
            print('Valid loss dicreased, save model')
            valid_loss_min = valid_loss
            torch.save(model.state_dict(), save_path)

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

    # return trained model
    return model
```

[14]:
```
# train the model
model_scratch = train(10, 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: 4.730289         Validation Loss: 7.162275
Valid loss dicreased, save model
Epoch: 2        Training Loss: 4.354377         Validation Loss: 6.189287
Valid loss dicreased, save model
Epoch: 3        Training Loss: 4.159519         Validation Loss: 5.771660
Valid loss dicreased, save model
Epoch: 4        Training Loss: 4.020030         Validation Loss: 4.082870
Valid loss dicreased, save model
Epoch: 5        Training Loss: 3.877778         Validation Loss: 4.031964
Valid loss dicreased, save model
Epoch: 6        Training Loss: 3.774415         Validation Loss: 4.053367
```

```
Epoch: 7        Training Loss: 3.628806         Validation Loss: 4.082312
Epoch: 8        Training Loss: 3.525453         Validation Loss: 4.191248
Epoch: 9        Training Loss: 3.412873         Validation Loss: 4.034206
Epoch: 10       Training Loss: 3.312596         Validation Loss: 3.582655
Valid loss dicreased, save model
```

[14]:
```
<All keys matched successfully>
```

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%.

[4]:
```
model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

[4]:
```
<All keys matched successfully>
```

[10]:
```
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))
```

[16]:
```
# call test function
test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.595380


Test Accuracy: 12% (104/836)
```

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.

[12]:
```
## TODO: Specify data loaders
import os
from torchvision import datasets, transforms


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

transform = transforms.Compose([
                                transforms.RandomResizedCrop(224),
                                transforms.RandomHorizontalFlip(),
                                transforms.RandomRotation(5),
                                transforms.ToTensor(),
                                transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                     std=[0.229, 0.224, 0.225])
                                ])

test_transform = transforms.Compose([
                                     transforms.RandomResizedCrop(224),
                                     transforms.ToTensor(),
                                     transforms.Normalize(mean=[0.485, 0.456, 0.
 ↪406],
```

```
                                                    std=[0.229, 0.224, 0.
↪225])
                                    ])

train_dataset = datasets.ImageFolder('dogImages/train', transform=transform)
train_dataloader = torch.utils.data.DataLoader(train_dataset, batch_size=64,␣
↪shuffle=True)

valid_dataset = datasets.ImageFolder('dogImages/valid',␣
↪transform=test_transform)
valid_dataloader = torch.utils.data.DataLoader(valid_dataset, batch_size=64,␣
↪shuffle=True)

test_dataset = datasets.ImageFolder('dogImages/test', transform=test_transform)
test_dataloader = torch.utils.data.DataLoader(test_dataset, batch_size=64,␣
↪shuffle=True)

loaders_transfer = dict()
loaders_transfer['train'] = train_dataloader
loaders_transfer['valid'] = valid_dataloader
loaders_transfer['test'] = test_dataloader
```

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`.

[15]:
```
import torchvision.models as models
import torch.nn as nn

resnet34 = models.resnet34(pretrained=True)

for param in resnet34.parameters():
    param.requires_grad = False

n_inputs = resnet34.fc.in_features
last_layer = nn.Linear(n_inputs, 133)
resnet34.fc = last_layer

model_transfer = resnet34

if use_cuda:
    print('Use cuda')
    model_transfer = model_transfer.cuda()
```

```
Use cuda
```

[19]:
```
model_transfer
```

[19]:
```
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=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (2): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
```

```
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (2): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (3): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
```

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

```
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (4): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (5): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
```

```
1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (2): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=133, bias=True)
)
```

Question 5: Outline the steps you took to get to your final CNN architecture and your reasoning at each step. Describe why you think the architecture is suitable for the current problem.

Answer: I tested vgg16, resnet18 but resnet34 shows the best result. Then I freeze layers and replaced last Liniar layer with 133 output size. New created layer is unfreez.

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.

[16]:
```
import torch.optim as optim
from torch import nn

criterion_transfer = nn.CrossEntropyLoss()
optimizer_transfer = optim.AdamW(model_transfer.parameters(), amsgrad=True)
```

[17]:
```
import torch
use_cuda = torch.cuda.is_available()
```

[16]:
```
torch.cuda.empty_cache()
```

[18]:
```
model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

[18]:
```
<All keys matched successfully>
```

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'`.

[16]:
```
# train the model
#model_transfer = # train(n_epochs, loaders_transfer, model_transfer,␣
↪optimizer_transfer, criterion_transfer, use_cuda, 'model_transfer.pt')

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

# train the model
model_transfer = train(10, loaders_transfer, model_transfer, optimizer_transfer,
                       criterion_transfer, use_cuda, 'model_transfer.pt')

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

```
Epoch: 1        Training Loss: 3.440468         Validation Loss: 2.055872
Valid loss dicreased, save model
Epoch: 2        Training Loss: 1.751199         Validation Loss: 1.404731
Valid loss dicreased, save model
Epoch: 3        Training Loss: 1.334293         Validation Loss: 1.303654
Valid loss dicreased, save model
Epoch: 4        Training Loss: 1.153965         Validation Loss: 1.126354
Valid loss dicreased, save model
Epoch: 5        Training Loss: 1.092142         Validation Loss: 1.051830
Valid loss dicreased, save model
Epoch: 6        Training Loss: 0.987448         Validation Loss: 1.072479
Epoch: 7        Training Loss: 0.944206         Validation Loss: 0.928595
Valid loss dicreased, save model
Epoch: 8        Training Loss: 0.945624         Validation Loss: 0.936913
Epoch: 9        Training Loss: 0.847765         Validation Loss: 1.003409
Epoch: 10       Training Loss: 0.870708         Validation Loss: 0.948801
```

[16]:
```
<All keys matched successfully>
```

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%.

[19]:
```
test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 1.006529


Test Accuracy: 73% (617/836)
```

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.

[25]:
```
### 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 train_dataset.classes]

print(len(class_names))
print(class_names)


def predict_breed_transfer(img_path):
    # load the image and return the predicted breed

    transform = transforms.Compose([transforms.Resize((224, 224)),
                                    transforms.ToTensor()])

    im = Image.open(img_path)

    img = transform(im)
    img = img.unsqueeze(0)

    if use_cuda:
        img = img.cuda()

    preds = model_transfer(img)
    indx = torch.argmax(preds, dim=1)

    return class_names[indx]
```

```
133
['Affenpinscher', 'Afghan hound', 'Airedale terrier', 'Akita', 'Alaskan
malamute', 'American eskimo dog', 'American foxhound', 'American staffordshire
terrier', 'American water spaniel', 'Anatolian shepherd dog', 'Australian cattle
dog', 'Australian shepherd', 'Australian terrier', 'Basenji', 'Basset hound',
'Beagle', 'Bearded collie', 'Beauceron', 'Bedlington terrier', 'Belgian
malinois', 'Belgian sheepdog', 'Belgian tervuren', 'Bernese mountain dog',
'Bichon frise', 'Black and tan coonhound', 'Black russian terrier',
'Bloodhound', 'Bluetick coonhound', 'Border collie', 'Border terrier', 'Borzoi',
'Boston terrier', 'Bouvier des flandres', 'Boxer', 'Boykin spaniel', 'Briard',
'Brittany', 'Brussels griffon', 'Bull terrier', 'Bulldog', 'Bullmastiff', 'Cairn
terrier', 'Canaan dog', 'Cane corso', 'Cardigan welsh corgi', 'Cavalier king
charles spaniel', 'Chesapeake bay retriever', 'Chihuahua', 'Chinese crested',
'Chinese shar-pei', 'Chow chow', 'Clumber spaniel', 'Cocker spaniel', 'Collie',
'Curly-coated retriever', 'Dachshund', 'Dalmatian', 'Dandie dinmont terrier',
'Doberman pinscher', 'Dogue de bordeaux', 'English cocker spaniel', 'English
setter', 'English springer spaniel', 'English toy spaniel', 'Entlebucher
mountain dog', 'Field spaniel', 'Finnish spitz', 'Flat-coated retriever',
'French bulldog', 'German pinscher', 'German shepherd dog', 'German shorthaired
pointer', 'German wirehaired pointer', 'Giant schnauzer', 'Glen of imaal
terrier', 'Golden retriever', 'Gordon setter', 'Great dane', 'Great pyrenees',
'Greater swiss mountain dog', 'Greyhound', 'Havanese', 'Ibizan hound',
'Icelandic sheepdog', 'Irish red and white setter', 'Irish setter', 'Irish
terrier', 'Irish water spaniel', 'Irish wolfhound', 'Italian greyhound',
'Japanese chin', 'Keeshond', 'Kerry blue terrier', 'Komondor', 'Kuvasz',
'Labrador retriever', 'Lakeland terrier', 'Leonberger', 'Lhasa apso', 'Lowchen',
'Maltese', 'Manchester terrier', 'Mastiff', 'Miniature schnauzer', 'Neapolitan
mastiff', 'Newfoundland', 'Norfolk terrier', 'Norwegian buhund', 'Norwegian
elkhound', 'Norwegian lundehund', 'Norwich terrier', 'Nova scotia duck tolling
retriever', 'Old english sheepdog', 'Otterhound', 'Papillon', 'Parson russell
terrier', 'Pekingese', 'Pembroke welsh corgi', 'Petit basset griffon vendeen',
'Pharaoh hound', 'Plott', 'Pointer', 'Pomeranian', 'Poodle', 'Portuguese water
dog', 'Saint bernard', 'Silky terrier', 'Smooth fox terrier', 'Tibetan mastiff',
'Welsh springer spaniel', 'Wirehaired pointing griffon', 'Xoloitzcuintli',
'Yorkshire terrier']
```

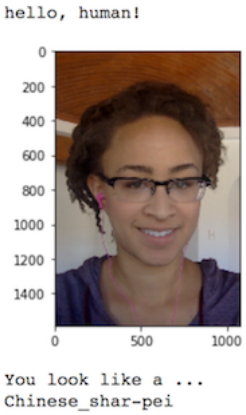
Step 5: Write your Algorithm

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 `dog_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

[26]:
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.

import matplotlib.pyplot as plt
%matplotlib inline

def imshow(title, img_path):
    print(title)

    img = Image.open(img_path)
    plt.imshow(img)
    plt.show()

def run_app(img_path):
    ## handle cases for a human face, dog, and neither

    is_dog = dog_detector(img_path)
    is_human = face_detector(img_path)

    breed = predict_breed_transfer(img_path)

    if is_dog:
        imshow('Hello ' + breed, img_path)
    elif is_human:
        imshow('Hello humman, you look like ' + breed, img_path)
    else:
        imshow('Error, not dog neither human was detected!', img_path)
```

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:

1. Image could have both dog and human it would be nice to recognize them both simultaneously.
2. Using bigger network resnet150 and more epoch to lern.
3. Traning process has constant learning rate, we can use some Lerning rate finder method.
4. Use more agumentation: color, zooming, image shifting etc.

[30]:
```
## 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
my_test_files = np.array(glob("my_test_set/*"))
for file in my_test_files:
    run_app(file)
```

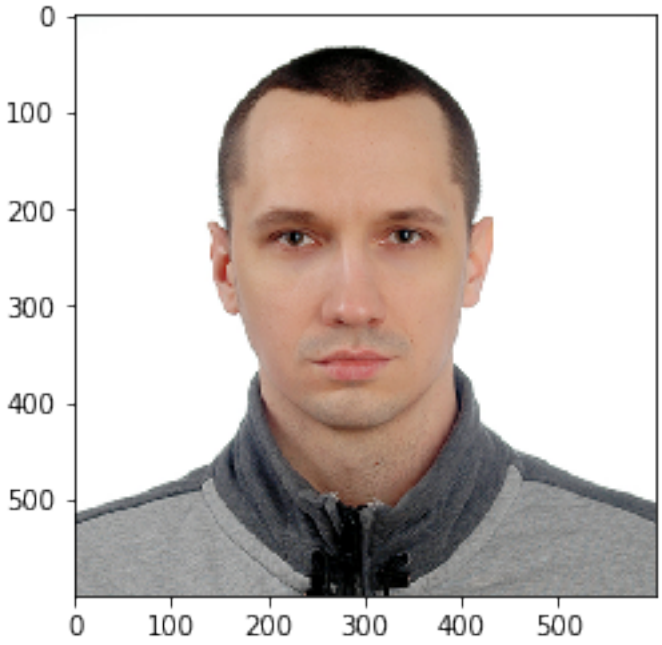
```
Hello humman, you look like Chihuahua
```

```
Hello humman, you look like English cocker spaniel
```

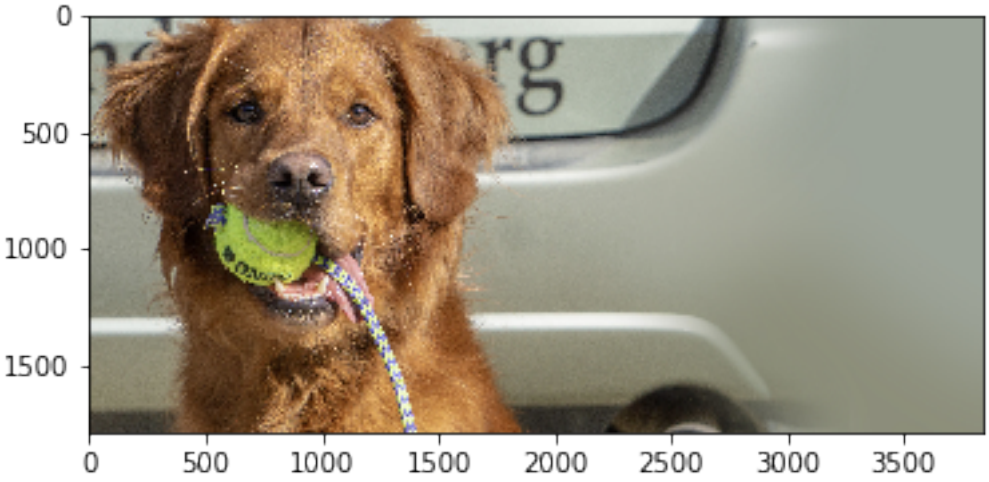

```
Hello Alaskan malamute
```

```
Hello Greyhound
```


```
Hello humman, you look like Chihuahua
```



```
Hello Chinese crested
```



[ ]: