Convolutional Neural Networks

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.

Why We're Here

In this notebook, you will make the first steps towards developing an algorithm that could be used as part of a mobile or web app. At the end of this project, your code will accept any user-supplied image as input. If a dog is detected in the image, it will provide an estimate of the dog's breed. If a human is detected, it will provide an estimate of the dog breed that is most resembling. The image below displays potential sample output of your finished project (... but we expect that each student's algorithm will behave differently!).

In this real-world setting, you will need to piece together a series of models to perform different tasks; for instance, the algorithm that detects humans in an image will be different from the CNN that infers dog breed. There are many points of possible failure, and no perfect algorithm exists. Your imperfect solution will nonetheless create a fun user experience!

The Road Ahead

We break the notebook into separate steps. Feel free to use the links below to navigate the 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

- 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 [3]:

```
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

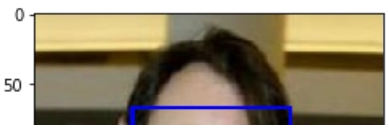
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

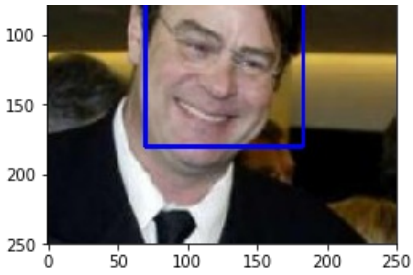
OpenCV provides many pre-trained face detectors, stored as XML files on github. We have downloaded one of these detectors and stored it in the `haarcascades` directory. In the next code cell, we demonstrate how to use this detector to find human faces in a sample image.

In [4]:

```
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.

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 [5]:

```
# 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
```

(IMPLEMENTATION) Assess the Human Face Detector

Question 1: Use the code cell below to test the performance of the `face_detector` function.

- What percentage of the first 100 images in `human_files` have a detected human face?
- What percentage of the first 100 images in `dog_files` have a detected human face?

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

Answer: (You can print out your results and/or write your percentages in this cell)

In [6]:

```
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.
human_files_hf_detected = 0
dog_files_hf_detected = 0

num_human_files = len(human_files_short)
num_dog_files = len(dog_files_short)

for i in tqdm(range(0, num_human_files), desc='Processing images', unit='Image'):

    human_path = human_files_short[i]
    if face_detector(human_path) == True:
        human_files_hf_detected += 1
```

```
        human_files_hf_detected += 1

for i in tqdm(range(0, num_dog_files), desc='Processing images', unit='Image'):
    dog_path = dog_files_short[i]
    if face_detector(dog_path) == True:
        dog_files_hf_detected += 1

print('The percentage of the first 100 images in human_files have a ' + \
      'detected human face is {0:.0%}'.format(human_files_hf_detected / num_human_files))
print('The percentage of the first 100 images in dog_files have a ' + \
      'detected human face is {0:.0%}'.format(dog_files_hf_detected / num_dog_files))
```

```
Processing images: 100%|██████████| 100/100 [00:02<00:00, 34.90Image/s]
Processing images: 100%|██████████| 100/100 [00:30<00:00,  3.32Image/s]
```

```
The percentage of the first 100 images in human_files have a detected human face is 98%
The percentage of the first 100 images in dog_files have a detected human face is 17%
```

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

In [7]:

```
### (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.

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 [8]:

```
import torch
import torchvision.models as models

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

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

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

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to
/root/.torch/models/vgg16-397923af.pth
100%|██████████| 553433881/553433881 [00:04<00:00, 112248463.61it/s]
```

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

(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 [9]:

```
from PIL import Image
import torchvision.transforms as transforms
from torch.autograd import Variable

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
    img = Image.open(img_path).convert('RGB')

    # Resize PIL image to 224x224 tensors as VGG-16 input and
    # Normailize the image tensors to 0-1 value.
    data_transform = transforms.Compose([transforms.RandomResizedCrop(224),
                                         transforms.ToTensor(),
                                         transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                              std=[0.229, 0.224, 0.225])])

    # transform the image with the data_transform function
    img = data_transform(img)

    # Insert a new dimension at index 0 - in front of the other the current dims of (num color cha
nnels,
    # height, width)
    img = img.unsqueeze(0)

    # Converting img tensor to Pytorch Variable - s wrapper around a PyTorch Tensor.
    if use_cuda:
        img = img.cuda()

    # Returns a Tensor of shape (batch, num class labels)
    prediction = VGG16(img)

    ## Return the *index* of the predicted class for that image

    return torch.max(prediction,1)[1].item() # predicted class index
```

(IMPLEMENTATION) Write a Dog Detector

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

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

In [10]:

```
### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    ## TODO: Complete the function.
    predicted_class_index = VGG16_predict(img_path)
```

```
    if predicted_class_index >= 151 and predicted_class_index <= 268:
        return True
    else:
        return False
```

(IMPLEMENTATION) Assess the Dog Detector

Question 2: Use the code cell below to test the performance of your `dog_detector` function.

- What percentage of the images in `human_files_short` have a detected dog?
- What percentage of the images in `dog_files_short` have a detected dog?

Answer:

In [11]:

```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
human_files_dog_detected = 0
dog_files_dog_detected = 0

num_human_files = len(human_files_short)
num_dog_files = len(dog_files_short)

for i in tqdm(range(0, num_human_files), desc='Processing images', unit='Image'):

    human_path = human_files_short[i]
    if dog_detector(human_path) == True:
        human_files_dog_detected += 1

for i in tqdm(range(0, num_dog_files), desc='Processing images', unit='Image'):
    dog_path = dog_files_short[i]
    if dog_detector(dog_path) == True:
        dog_files_dog_detected += 1

print('{0:.0%} of the images in human_files_short have a detected
dog.'.format(human_files_dog_detected / num_human_files))
print('{0:.0%} of the images in dog_files_short have a detected
dog.'.format(dog_files_dog_detected / num_dog_files))
```

```
Processing images: 100%|██████████| 100/100 [00:03<00:00, 30.60Image/s]
Processing images: 100%|██████████| 100/100 [00:04<00:00, 23.04Image/s]
```

```
1% of the images in human_files_short have a detected dog.
98% of the images in dog_files_short have a detected dog.
```

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

In [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	Black 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!

(IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

Use the code cell below to write three separate [data loaders](http://pytorch.org/docs/master/data.html#torch.utils.data.DataLoader) 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](http://pytorch.org/docs/master/torchvision/datasets.html#imagefolder) to be a useful resource. If you are interested in augmenting your training and/or validation data, check out the wide variety of [transforms](http://pytorch.org/docs/master/torchvision/transforms.html)!

In [13]:

```
import os
from torchvision import datasets
from PIL import ImageFile

ImageFile.LOAD_TRUNCATED_IMAGES = True

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

# number of subprocesses to use for data loading
num_workers = 0
# how many samples per batch to load
batch_size = 20

# convert data to a normalized torch.FloatTensor
data_transform_train = transforms.Compose([transforms.Resize(256),
                                      transforms.RandomResizedCrop(224),
                                      transforms.RandomHorizontalFlip(),
                                      transforms.RandomRotation(10),
                                      transforms.ToTensor(),
                                      transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                      std=[0.229, 0.224, 0.225])])

data_transform_val_test = transforms.Compose([transforms.Resize(256),
                                      transforms.RandomResizedCrop(224),
                                      transforms.ToTensor(),
                                      transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                      std=[0.229, 0.224, 0.225])])

# choose and transform the training, validation, and test datasets
train_data = datasets.ImageFolder('/data/dog_images/train', transform=data_transform_train)
valid_data = datasets.ImageFolder('/data/dog_images/valid', transform=data_transform_val_test)
test_data = datasets.ImageFolder('/data/dog_images/test', transform=data_transform_val_test)

# prepare data loaders with random shuffling included
data_loaders = {}
data_loaders['train'] = torch.utils.data.DataLoader(train_data,
                                          batch_size=batch_size,
                                          shuffle=True,
```

```
                                          num_workers=num_workers)

data_loaders['valid']  = torch.utils.data.DataLoader(valid_data,
                                          batch_size=batch_size,
                                          shuffle=True,
                                          num_workers=num_workers)

data_loaders['test']  = torch.utils.data.DataLoader(test_data,
                                          batch_size=batch_size,
                                          shuffle=True,
                                          num_workers=num_workers)

print('Total training images: {}'.format(len(train_data)))
print('Total validation images: {}'.format(len(valid_data)))
print('Total test images: {}'.format(len(test_data)))

# Get the number of classes
n_classes = len(train_data.classes)
print('Number of classes in train_data: {}'.format(n_classes))
```

```
Total training images: 6680
Total validation images: 835
Total test images: 836
Number of classes in train_data: 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:

> Reply: I have incorporated random resize by cropping to size of 224x224. The size was 224*224 as this size seems to work pretty well in human face and dog recognition as shown in previous cells.
>
> Reply: I have incorporated random horizontal flip and random rotation of 10 degrees as the dog faces can be taken from the opposite angle and the dog faces can tilt in about 10 degrees.

In [14]:

```
from torchvision import utils

def visualize_sample_images(inp):
    inp = inp.numpy().transpose((1, 2, 0))
    inp = np.clip(inp, 0, 1)

    fig = plt.figure(figsize=(50, 25))
    plt.axis('off')
    plt.imshow(inp)
    plt.pause(0.001)

# Get a batch of training data.
inputs, classes = next(iter(data_loaders['train']))

# Convert the batch to a grid.
grid = utils.make_grid(inputs)

# Display!
visualize_sample_images(grid)
```

(IMPLEMENTATION) Model Architecture

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

In [35]:

```
import torch.nn as nn
import torch.nn.functional as F

# define the CNN architecture
class Net(nn.Module):
    ### TODO: choose an architecture, and complete the class
    def __init__(self):
        super(Net, self).__init__()
        ## Define layers of a CNN
        self.conv1 = nn.Conv2d(3, 32, 3, stride=2, padding=1)
        self.conv2 = nn.Conv2d(32, 64, 3, stride=2, padding=1)
        self.conv3 = nn.Conv2d(64, 128, 3, padding=1)

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

        # fully-connected
        self.fc1 = nn.Linear(7*7*128, 500)
        self.fc2 = nn.Linear(500, n_classes)

        # drop-out
        self.dropout = nn.Dropout(0.3)

    def forward(self, x):
        ## Define forward behavior
        x = F.relu(self.conv1(x))
        x = self.pool(x)
        x = F.relu(self.conv2(x))
        x = self.pool(x)
        x = F.relu(self.conv3(x))
        x = self.pool(x)

        # flatten
        x = x.view(-1, 7*7*128)

        # applied dropout for inputs of fully connected layer
        x = self.dropout(x)
        x = F.relu(self.fc1(x))

        x = self.dropout(x)
        x = self.fc2(x)

        return x

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

# instantiate the CNN
model_scratch = Net()
print(model_scratch)
# move tensors to GPU if CUDA is available
if use_cuda:
    model_scratch.cuda()
```

```
Net(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=6272, out_features=500, bias=True)
```

```
  (fc2): Linear(in_features=500, out_features=133, bias=True)
  (dropout): Dropout(p=0.3)
)
```

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

Answer:

I have used a basic convulational network model to achieve minimum of 10% accuracy.

There are 2 conv layers with kernel size of 3, stride of 2, padding of 1 and 1 conv layers with kernel size of 3, stride of 2, padding of 1. There were incremental filter sizes of 32, 64, and **128** in the 3 layers, respectively. Here the XY-dimension in each filter gets to reduce by **4** (due to stride of 2 in each of the layers 1 and 2). The values of the XY-dimension of each filter were relu-activated for each conv layer.

To reduce the dimension further, maxpooling with stride of 2 were applied after each of the 3 relu-activated conv layers. The XY-dimension gets to reduce further by a factor of 2 at each maxpooling steps - hence make up a total of **8** in total.

The flatten output of the final conv layers is fllaten as input for the fully-connected layer (fc1) - hence **224/(4*8*) 128** (6272). The out_features of fc1 was set as 500. The out_features of the final fc2 were set as 133 - the number of trainning classes. A dropout of 0.3 was applied for each inputs (in_features) of each of the 2 relu-activated fc's.

(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 [36]:

```
import torch.optim as optim

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

### TODO: select optimizer
optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=0.001, momentum=0.9)
```

(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 [21]:

```
# import EarlyStopping
from pytorchtools import EarlyStopping

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

    # to track the average training loss per epoch as the model trains
    avg_train_losses = []
    # to track the average validation loss per epoch as the model trains
    avg_valid_losses = []

    # initialize the early_stopping object
    early_stopping = EarlyStopping(patience=patience, verbose=True)

    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()
            # clear the gradients of all optimized variables
            optimizer.zero_grad()
            ## find the loss and update the model parameters accordingly
            # forward pass: compute predicted outputs by passing inputs to the model
            output = model(data)
            # calculate the batch loss
            loss = criterion(output, target)
            # backward pass: compute gradient of the loss with respect to model parameters
            loss.backward()
            # perform a single optimization step (parameter update)
            optimizer.step()
            ## record the average training loss, using something like
            ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss))
            # update training loss
            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
            # forward pass: compute predicted outputs by passing inputs to the model
            output = model(data)
            # calculate the batch loss
            loss = criterion(output, target)
            # update average validation loss
            valid_loss += loss.item()*data.size(0)

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


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

        # print training/validation statistics
        # calculate average loss over an epoch
        avg_train_losses.append(train_loss)
        avg_valid_losses.append(valid_loss)


        # early_stopping needs the validation loss to check if it has decresed,
        # and if it has, it will make a checkpoint of the current model
        early_stopping(valid_loss, model)

        if early_stopping.early_stop:
            print("Early stopping")
            break

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

    # return trained model
    return model, avg_train_losses, avg_valid_losses

# train the model
model_scratch, train_loss, valid_loss  = train(100, data_loaders, 20, model_scratch, optimizer_scra
tch,
                      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 / 100  Training Loss: 90.350648  Validation Loss: 90.417217
Validation loss decreased (inf --> 90.417217).  Saving model ...
Validation loss decreased (inf --> 90.417217).  Saving model ...
Epoch: 2 / 100  Training Loss: 90.316117  Validation Loss: 89.675355
Validation loss decreased (90.417217 --> 89.675355).  Saving model ...
Validation loss decreased (90.417217 --> 89.675355).  Saving model ...
Epoch: 3 / 100  Training Loss: 89.621195  Validation Loss: 88.935678
Validation loss decreased (89.675355 --> 88.935678).  Saving model ...
Validation loss decreased (89.675355 --> 88.935678).  Saving model ...
Epoch: 4 / 100  Training Loss: 89.371411  Validation Loss: 89.233538
EarlyStopping counter: 1 out of 20
Epoch: 5 / 100  Training Loss: 88.709588  Validation Loss: 88.525445
Validation loss decreased (88.935678 --> 88.525445).  Saving model ...
Validation loss decreased (88.935678 --> 88.525445).  Saving model ...
Epoch: 6 / 100  Training Loss: 88.143291  Validation Loss: 88.184009
Validation loss decreased (88.525445 --> 88.184009).  Saving model ...
Validation loss decreased (88.525445 --> 88.184009).  Saving model ...
Epoch: 7 / 100  Training Loss: 87.834446  Validation Loss: 87.372496
Validation loss decreased (88.184009 --> 87.372496).  Saving model ...
Validation loss decreased (88.184009 --> 87.372496).  Saving model ...
Epoch: 8 / 100  Training Loss: 87.368399  Validation Loss: 87.934211
EarlyStopping counter: 1 out of 20
Epoch: 9 / 100  Training Loss: 86.748650  Validation Loss: 86.491724
Validation loss decreased (87.372496 --> 86.491724).  Saving model ...
Validation loss decreased (87.372496 --> 86.491724).  Saving model ...
Epoch: 10 / 100  Training Loss: 86.393943  Validation Loss: 86.923097
EarlyStopping counter: 1 out of 20
Epoch: 11 / 100  Training Loss: 85.802853  Validation Loss: 85.746852
Validation loss decreased (86.491724 --> 85.746852).  Saving model ...
Validation loss decreased (86.491724 --> 85.746852).  Saving model ...
Epoch: 12 / 100  Training Loss: 85.185788  Validation Loss: 86.680553
EarlyStopping counter: 1 out of 20
Epoch: 13 / 100  Training Loss: 85.165219  Validation Loss: 86.218624
EarlyStopping counter: 2 out of 20
Epoch: 14 / 100  Training Loss: 84.256929  Validation Loss: 85.635996
Validation loss decreased (85.746852 --> 85.635996).  Saving model ...
Validation loss decreased (85.746852 --> 85.635996).  Saving model ...
Epoch: 15 / 100  Training Loss: 83.865294  Validation Loss: 84.791502
Validation loss decreased (85.635996 --> 84.791502).  Saving model ...
Validation loss decreased (85.635996 --> 84.791502).  Saving model ...
Epoch: 16 / 100  Training Loss: 83.858987  Validation Loss: 85.811993
EarlyStopping counter: 1 out of 20
Epoch: 17 / 100  Training Loss: 83.119790  Validation Loss: 84.051903
Validation loss decreased (84.791502 --> 84.051903).  Saving model ...
Validation loss decreased (84.791502 --> 84.051903).  Saving model ...
Epoch: 18 / 100  Training Loss: 82.593848  Validation Loss: 83.695048
Validation loss decreased (84.051903 --> 83.695048).  Saving model ...
Validation loss decreased (84.051903 --> 83.695048).  Saving model ...
Epoch: 19 / 100  Training Loss: 81.699981  Validation Loss: 82.852800
Validation loss decreased (83.695048 --> 82.852800).  Saving model ...
Validation loss decreased (83.695048 --> 82.852800).  Saving model ...
Epoch: 20 / 100  Training Loss: 81.618933  Validation Loss: 82.965652
EarlyStopping counter: 1 out of 20
Epoch: 21 / 100  Training Loss: 80.963535  Validation Loss: 83.435257
EarlyStopping counter: 2 out of 20
Epoch: 22 / 100  Training Loss: 80.240317  Validation Loss: 82.075425
Validation loss decreased (82.852800 --> 82.075425).  Saving model ...
Validation loss decreased (82.852800 --> 82.075425).  Saving model ...
Epoch: 23 / 100  Training Loss: 79.499053  Validation Loss: 81.897827
Validation loss decreased (82.075425 --> 81.897827).  Saving model ...
Validation loss decreased (82.075425 --> 81.897827).  Saving model ...
Epoch: 24 / 100  Training Loss: 79.579097  Validation Loss: 80.869419
Validation loss decreased (81.897827 --> 80.869419).  Saving model ...
Validation loss decreased (81.897827 --> 80.869419).  Saving model ...
Epoch: 25 / 100  Training Loss: 78.775978  Validation Loss: 81.124105
EarlyStopping counter: 1 out of 20
Epoch: 26 / 100  Training Loss: 78.335807  Validation Loss: 80.761124
Validation loss decreased (80.869419 --> 80.761124).  Saving model ...
Validation loss decreased (80.869419 --> 80.761124).  Saving model ...
Epoch: 27 / 100  Training Loss: 78.031258  Validation Loss: 81.166850
EarlyStopping counter: 1 out of 20
Epoch: 28 / 100  Training Loss: 77.629041  Validation Loss: 81.209465
```

```
EarlyStopping counter: 2 out of 20
Epoch: 29 / 100  Training Loss: 77.016107  Validation Loss: 80.813079
EarlyStopping counter: 3 out of 20
Epoch: 30 / 100  Training Loss: 76.583289  Validation Loss: 79.704759
Validation loss decreased (80.761124 --> 79.704759).  Saving model ...
Validation loss decreased (80.761124 --> 79.704759).  Saving model ...
Epoch: 31 / 100  Training Loss: 75.776464  Validation Loss: 79.311938
Validation loss decreased (79.704759 --> 79.311938).  Saving model ...
Validation loss decreased (79.704759 --> 79.311938).  Saving model ...
Epoch: 32 / 100  Training Loss: 75.828116  Validation Loss: 79.119414
Validation loss decreased (79.311938 --> 79.119414).  Saving model ...
Validation loss decreased (79.311938 --> 79.119414).  Saving model ...
Epoch: 33 / 100  Training Loss: 75.348971  Validation Loss: 80.510631
EarlyStopping counter: 1 out of 20
Epoch: 34 / 100  Training Loss: 74.737722  Validation Loss: 78.992535
Validation loss decreased (79.119414 --> 78.992535).  Saving model ...
Validation loss decreased (79.119414 --> 78.992535).  Saving model ...
Epoch: 35 / 100  Training Loss: 74.346798  Validation Loss: 78.199842
Validation loss decreased (78.992535 --> 78.199842).  Saving model ...
Validation loss decreased (78.992535 --> 78.199842).  Saving model ...
Epoch: 36 / 100  Training Loss: 74.142069  Validation Loss: 79.699306
EarlyStopping counter: 1 out of 20
Epoch: 37 / 100  Training Loss: 73.853793  Validation Loss: 79.177121
EarlyStopping counter: 2 out of 20
Epoch: 38 / 100  Training Loss: 73.537467  Validation Loss: 78.528689
EarlyStopping counter: 3 out of 20
Epoch: 39 / 100  Training Loss: 73.127239  Validation Loss: 77.340458
Validation loss decreased (78.199842 --> 77.340458).  Saving model ...
Validation loss decreased (78.199842 --> 77.340458).  Saving model ...
Epoch: 40 / 100  Training Loss: 72.609992  Validation Loss: 77.510021
EarlyStopping counter: 1 out of 20
Epoch: 41 / 100  Training Loss: 71.675200  Validation Loss: 77.883229
EarlyStopping counter: 2 out of 20
Epoch: 42 / 100  Training Loss: 71.808082  Validation Loss: 76.845066
Validation loss decreased (77.340458 --> 76.845066).  Saving model ...
Validation loss decreased (77.340458 --> 76.845066).  Saving model ...
Epoch: 43 / 100  Training Loss: 71.473153  Validation Loss: 76.308007
Validation loss decreased (76.845066 --> 76.308007).  Saving model ...
Validation loss decreased (76.845066 --> 76.308007).  Saving model ...
Epoch: 44 / 100  Training Loss: 71.056301  Validation Loss: 76.571269
EarlyStopping counter: 1 out of 20
Epoch: 45 / 100  Training Loss: 70.762823  Validation Loss: 77.850425
EarlyStopping counter: 2 out of 20
Epoch: 46 / 100  Training Loss: 70.231497  Validation Loss: 76.531737
EarlyStopping counter: 3 out of 20
Epoch: 47 / 100  Training Loss: 70.067234  Validation Loss: 75.996168
Validation loss decreased (76.308007 --> 75.996168).  Saving model ...
Validation loss decreased (76.308007 --> 75.996168).  Saving model ...
Epoch: 48 / 100  Training Loss: 69.688415  Validation Loss: 76.619075
EarlyStopping counter: 1 out of 20
Epoch: 49 / 100  Training Loss: 68.956668  Validation Loss: 77.396712
EarlyStopping counter: 2 out of 20
Epoch: 50 / 100  Training Loss: 68.477762  Validation Loss: 76.451242
EarlyStopping counter: 3 out of 20
Epoch: 51 / 100  Training Loss: 68.653350  Validation Loss: 75.806820
Validation loss decreased (75.996168 --> 75.806820).  Saving model ...
Validation loss decreased (75.996168 --> 75.806820).  Saving model ...
Epoch: 52 / 100  Training Loss: 68.145242  Validation Loss: 77.393060
EarlyStopping counter: 1 out of 20
Epoch: 53 / 100  Training Loss: 67.385896  Validation Loss: 75.509542
Validation loss decreased (75.806820 --> 75.509542).  Saving model ...
Validation loss decreased (75.806820 --> 75.509542).  Saving model ...
Epoch: 54 / 100  Training Loss: 67.599582  Validation Loss: 76.868359
EarlyStopping counter: 1 out of 20
Epoch: 55 / 100  Training Loss: 66.942385  Validation Loss: 76.216064
EarlyStopping counter: 2 out of 20
Epoch: 56 / 100  Training Loss: 66.562348  Validation Loss: 75.339821
Validation loss decreased (75.509542 --> 75.339821).  Saving model ...
Validation loss decreased (75.509542 --> 75.339821).  Saving model ...
Epoch: 57 / 100  Training Loss: 66.658216  Validation Loss: 73.790168
Validation loss decreased (75.339821 --> 73.790168).  Saving model ...
Validation loss decreased (75.339821 --> 73.790168).  Saving model ...
Epoch: 58 / 100  Training Loss: 66.276973  Validation Loss: 76.006407
EarlyStopping counter: 1 out of 20
Epoch: 59 / 100  Training Loss: 65.711250  Validation Loss: 76.058851
EarlyStopping counter: 2 out of 20
Epoch: 60 / 100  Training Loss: 65.739342  Validation Loss: 75.577038
```

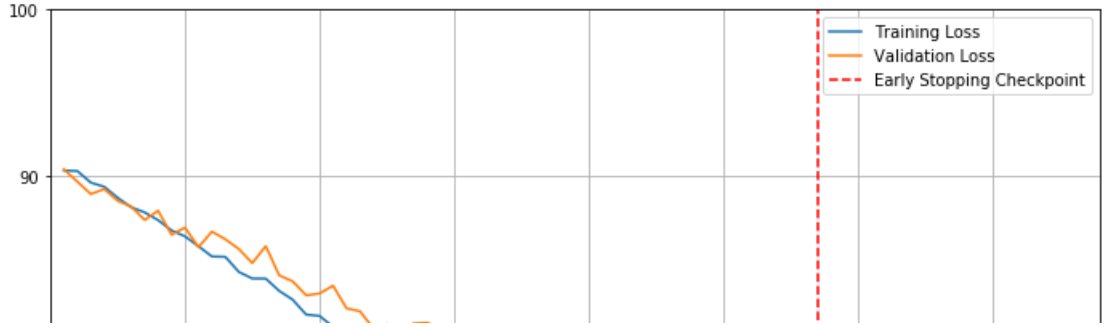
```
EarlyStopping counter: 3 out of 20
Epoch: 61 / 100  Training Loss: 65.077031  Validation Loss: 75.949653
EarlyStopping counter: 4 out of 20
Epoch: 62 / 100  Training Loss: 64.914573  Validation Loss: 75.561111
EarlyStopping counter: 5 out of 20
Epoch: 63 / 100  Training Loss: 64.278085  Validation Loss: 76.034991
EarlyStopping counter: 6 out of 20
Epoch: 64 / 100  Training Loss: 63.965625  Validation Loss: 74.361879
EarlyStopping counter: 7 out of 20
Epoch: 65 / 100  Training Loss: 63.639503  Validation Loss: 75.704378
EarlyStopping counter: 8 out of 20
Epoch: 66 / 100  Training Loss: 62.675564  Validation Loss: 74.888009
EarlyStopping counter: 9 out of 20
Epoch: 67 / 100  Training Loss: 63.521172  Validation Loss: 74.861048
EarlyStopping counter: 10 out of 20
Epoch: 68 / 100  Training Loss: 63.141471  Validation Loss: 75.369044
EarlyStopping counter: 11 out of 20
Epoch: 69 / 100  Training Loss: 62.416203  Validation Loss: 75.062136
EarlyStopping counter: 12 out of 20
Epoch: 70 / 100  Training Loss: 61.768213  Validation Loss: 73.809586
EarlyStopping counter: 13 out of 20
Epoch: 71 / 100  Training Loss: 62.375057  Validation Loss: 74.858974
EarlyStopping counter: 14 out of 20
Epoch: 72 / 100  Training Loss: 61.609354  Validation Loss: 74.906863
EarlyStopping counter: 15 out of 20
Epoch: 73 / 100  Training Loss: 61.573459  Validation Loss: 75.576906
EarlyStopping counter: 16 out of 20
Epoch: 74 / 100  Training Loss: 61.202809  Validation Loss: 74.894888
EarlyStopping counter: 17 out of 20
Epoch: 75 / 100  Training Loss: 61.112160  Validation Loss: 76.555054
EarlyStopping counter: 18 out of 20
Epoch: 76 / 100  Training Loss: 60.248291  Validation Loss: 75.210728
EarlyStopping counter: 19 out of 20
Epoch: 77 / 100  Training Loss: 60.289575  Validation Loss: 74.905981
EarlyStopping counter: 20 out of 20
Early stopping
```

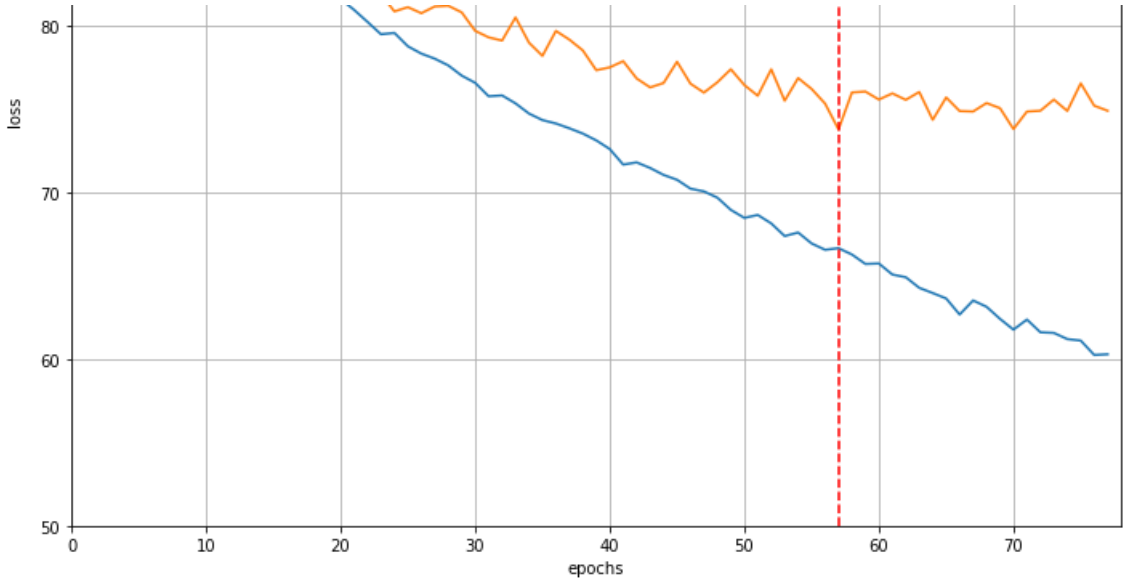
Visualizing the Loss and the Early Stopping Checkpoint

From the plot we can see that the last Early Stopping Checkpoint was saved right before the model started to overfit.

In [32]:

```
# visualize the loss as the network trained
fig = plt.figure(figsize=(10,8))
plt.plot(range(1,len(train_loss)+1),train_loss, label='Training Loss')
plt.plot(range(1,len(valid_loss)+1),valid_loss,label='Validation Loss')

# find position of lowest validation loss
minposs = valid_loss.index(min(valid_loss))+1
plt.axvline(minposs, linestyle='--', color='r',label='Early Stopping Checkpoint')

plt.xlabel('epochs')
plt.ylabel('loss')
plt.ylim(50, 100) # consistent scale
plt.xlim(0, len(train_loss)+1) # consistent scale
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
# fig.savefig('loss_plot.png', bbox_inches='tight')
```





(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 [26]:

```
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))

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

# # call test function
test(data_loaders, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.837710


Test Accuracy: 14% (121/836)
```

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

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.

(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 [17]:

```
## TODO: Specify data loaders
loaders_transfer = data_loaders.copy()
```

(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 [20]:

```
import torchvision.models as models
import torch.nn as nn

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

model_transfer
```

```
Downloading: "https://download.pytorch.org/models/resnet50-19c8e357.pth" to
/root/.torch/models/resnet50-19c8e357.pth
100%|██████████| 102502400/102502400 [00:04<00:00, 22726175.82it/s]
```

Out[20]:

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

```
    (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
)
(layer2): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
    (downsample): Sequential(
      (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
  (2): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
  (3): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
)
(layer3): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
    (downsample): Sequential(
      (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
  (2): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
```

```
    )
    (3): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
    (4): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
    (5): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
  )
  (layer4): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (downsample): Sequential(
        (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
    (2): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
  )
  (avgpool): AvgPool2d(kernel_size=7, stride=1, padding=0)
  (fc): Linear(in_features=2048, out_features=1000, bias=True)
)
```

In [22]:

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

model_transfer.fc = nn.Linear(in_features=2048, out_features=133, bias=True)
fc_parameters = model_transfer.fc.parameters()

for param in fc_parameters:
    param.requires_grad = True
```

```
if use_cuda:
    model_transfer = model_transfer.cuda()
```

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:

The resnet50 model pre-trained to classify 1000 classes were downloaded for model transfer. As the new training data size is quite small (6680 images) and possibly similar to the original training data, the fully connected layer at the end of the neural network were replaced with a new fully connected layer (in_features of 2048 and out_features of 133 matching with the number of new training classes). The weights of the new fully connected layer were randomized and subjected to weight training; all other weights from the pre-trained network were frozen.

The dog_app with dataset consisted of dogs' images that share similar features with many animals with 4 legs and the 1000 classes in the ImageNet Databases used for training ResNets and VGGs should have many animals with 4 legs, including dog, as shown in VGG16 above. The transfer learning is useful in this context at least in getting the edges and shapes for a dog, before further classfying into "species" that can be trained with the new dog_app datasets.

The ResNet has a structure with connections to deep CNNs can skip between layers and that help to reduce Vanishing Gradient problems present in Deep CNN. I picked ResNet50 as i believe 50 layers on training ImageNet Database with 1000 classes has sufficient features extracted yet well-trained to differentiate animals with 4 legs (including edges and shapes).

(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 [25]:

```
import torch.optim as optim

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

(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 [23]:

```
# train the model
n_epochs = 60
# model_transfer, train_loss, valid_loss = train(n_epochs, loaders_transfer, 10, model_transfer, o
ptimizer_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'))
```

(IMPLEMENTATION) Test the Model

Try out your model on the test dataset of dog images. Use the code cell below to calculate and print the test loss and accuracy. Ensure that your test accuracy is greater than 60%.

In [26]:

```
test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 1.346886


Test Accuracy: 71% (597/836)
```

(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 [28]:

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

# First 10 class_names
class_names[:10]
```

Out[28]:

```
['Affenpinscher',
 'Afghan hound',
 'Airedale terrier',
 'Akita',
 'Alaskan malamute',
 'American eskimo dog',
 'American foxhound',
 'American staffordshire terrier',
 'American water spaniel',
 'Anatolian shepherd dog']
```

In [29]:

```
# First 10 loaders_transfer class names
loaders_transfer['train'].dataset.classes[:10]
```

Out[29]:

```
['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']
```

In [30]:

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

from PIL import Image
import torchvision.transforms as transforms

def load_input_image(img_path):
    ## Load and pre-process an image from the given img_path
    img = Image.open(img_path).convert('RGB')

    # Resize PIL image to 224x224 tensors as VGG-16 input and
    # Normailize the image tensors to 0-1 value.
    prediction_transform = transforms.Compose([transforms.Resize(size=(224, 224)),
                                    transforms.ToTensor(),
                                    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                         std=[0.229, 0.224, 0.225])])

    # discard the transparent, alpha channel (that's the :3) and add the batch dimension
    img = prediction_transform(img)[:3,:,:].unsqueeze(0)

    if use_cuda:
        img = img.cuda()
    else:
        img = img.cpu()

    return img
```

```
def predict_breed_transfer(model_transfer, class_names, img_path):
    # load the image and return the predicted breed
    img = load_input_image(img_path)

    model_transfer.eval()

    if use_cuda:
        model_transfer = model_transfer.cuda()
    else:
        model_transfer = model_transfer.cpu()

    idx = torch.argmax(model_transfer(img))

    return class_names[idx]
```

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!

(IMPLEMENTATION) Write your Algorithm

In [31]:

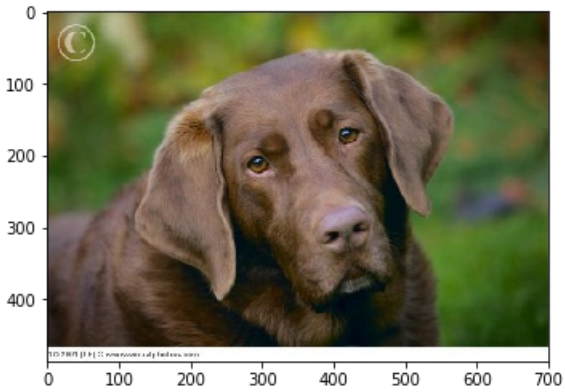
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.


def run_app(img_path):
    ## handle cases for a human face, dog, and neither
    img = Image.open(img_path)

    # Show the image
    plt.imshow(img)
    plt.show()

    if dog_detector(img_path):
        prediction = predict_breed_transfer(model_transfer, class_names, img_path)
        print("A dog detected - it looks like a {0}".format(prediction))
    elif face_detector(img_path) > 0:
        prediction = predict_breed_transfer(model_transfer, class_names, img_path)
        print("Hello human! You look like a ...\n{0}".format(prediction))
    else:
        print("No dog or human is detect.")
```

In [32]:

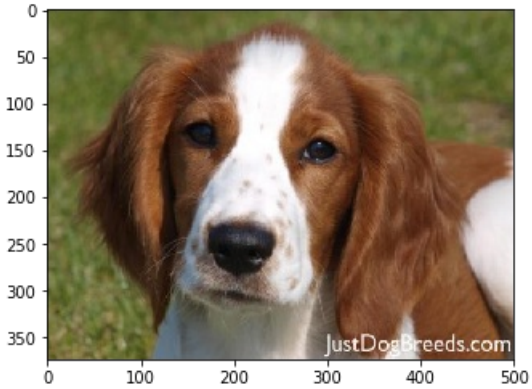
```
# Testing the run_app
for img_file in os.listdir('./images'):
    img_path = os.path.join('./images', img_file)
    run_app(img_path)
```

A dog detected - it looks like a Curly-coated retriever

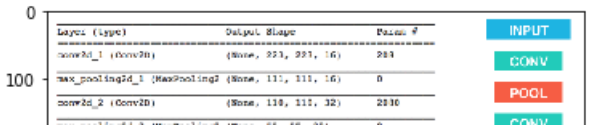
A dog detected - it looks like a Flat-coated retriever

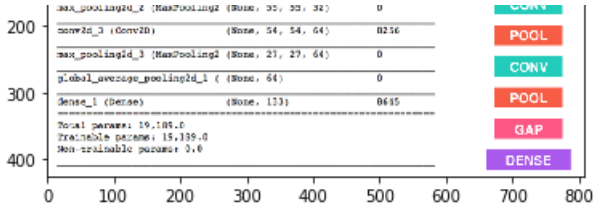


A dog detected - it looks like a Labrador retriever

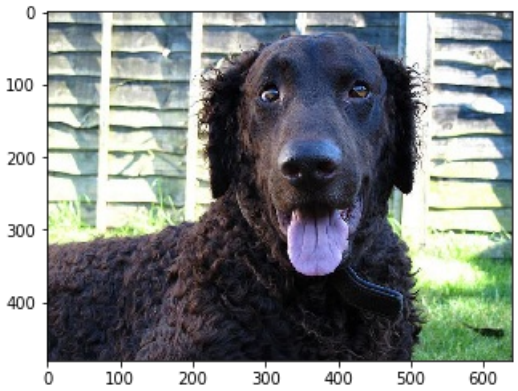


A dog detected - it looks like a Welsh springer spaniel

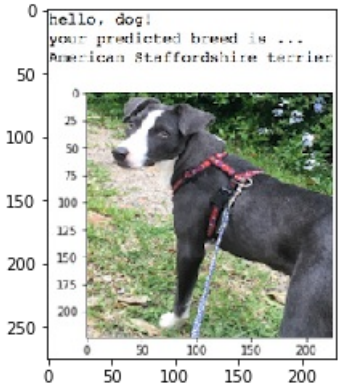




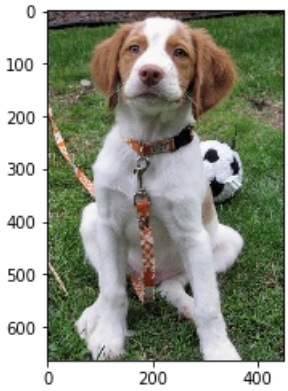
No dog or human is detect.



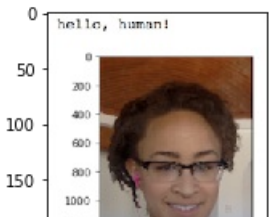
A dog detected - it looks like a Curly-coated retriever

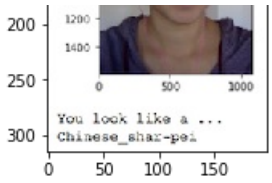


A dog detected - it looks like a Entlebucher mountain dog

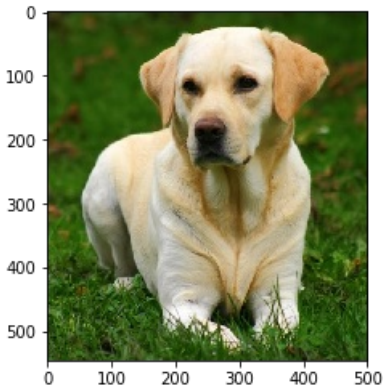


A dog detected - it looks like a Brittany






```
Hello human! You look like a ...
Dogue de bordeaux
```



```
A dog detected - it looks like a Labrador retriever
```

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?

(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.

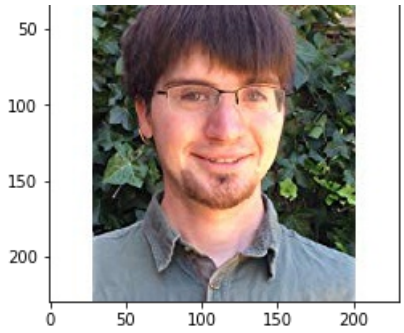
The algorithm is able to differentiate human, dog, and none images accurately - better than expected.

1. More training epoch may increase the model accuracy. Currently, the validation loss is still decreasing.
2. More image datasets of dogs and more image augmentations for training will improve the accuracy of the models.
3. Adding 1 more fully connected layer.
4. Ensembling of a few other pre-trained models may improve the accuracy further

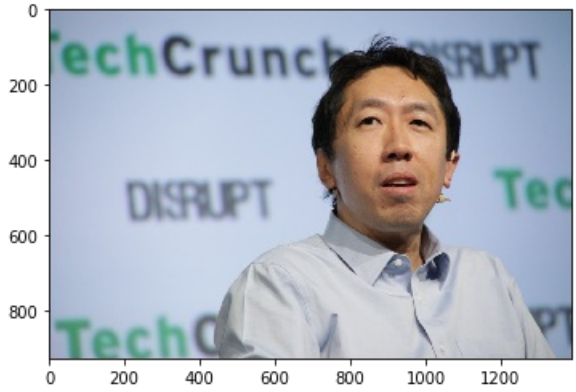
Answer: (Three possible points for improvement)

In [33]:

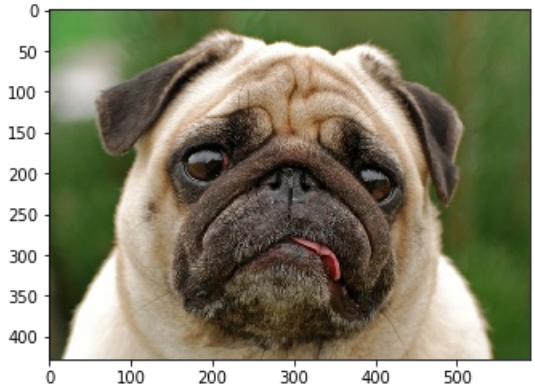
```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.
import glob

human_files = list(glob.glob('./My_uploaded_images/human_*'))
dog_files = list(glob.glob('./My_uploaded_images/dog_*'))

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



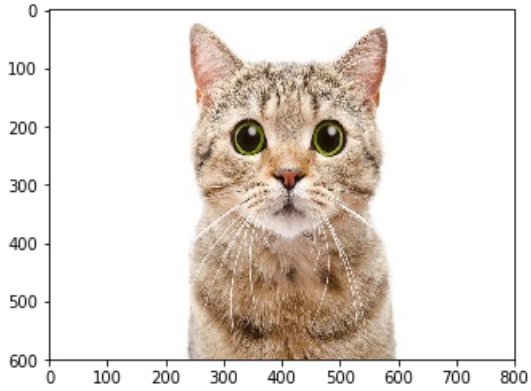
```
Hello human! You look like a ...
Cane corso
```




```
Hello human! You look like a ...
Australian shepherd
```



```
A dog detected - it looks like a Bullmastiff
```



```
No dog or human is detect.
```

A dog detected - it looks like a Tibetan mastiff

A dog detected - it looks like a Basenji