# 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!).

Sample Dog Output

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
- Step 1: Detect Humans
- Step 2: Detect Dogs

- Step 3: Create a CNN to Classify Dog Breeds (from Scratch)
- Step 4: Create a CNN to Classify Dog Breeds (using Transfer Learning)
- Step 5: Write your Algorithm
- Step 6: Test Your Algorithm

---

## Step 0: Import Datasets

Make sure that you've downloaded the required human and dog datasets:

- Download the dog dataset (https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/dogImages.zip). Unzip the folder and place it in this project's home directory, at the location `/dogImages` .
- Download the human dataset (https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/lfw.zip). 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 (http://www.7-zip.org/) 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

In [1]:
```python
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.
```

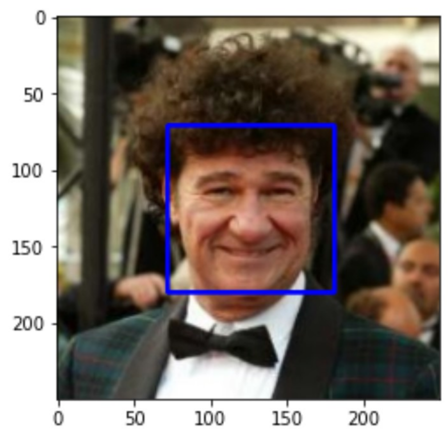
## Step 1: Detect Humans

In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers (http://docs.opencv.org/trunk/d7/d8b/tutorial_py_face_detection.html) to detect human faces in images.

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

In [2]:
```python
import cv2
import matplotlib.pyplot as plt
%matplotlib inline

# extract pre-trained face detector
face_cascade = cv2.CascadeClassifier('haarcascades/haarcascade_frontalface_alt.xm
l')

# 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 [3]:
```
# returns "True" if face is detected in image stored at img_path
def face_detector(img_path):
    img = cv2.imread(img_path)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    faces = face_cascade.detectMultiScale(gray)
    return len(faces) > 0
```

## (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:** Results are printed out following each cell and summarized in a table below optional detector using `MTCNN`

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

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

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

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

        for key, cohort in {'human_files_short': human_files_short, 'dog_files_short': do
        g_files_short}.items():
            face_detected = 0
            for face in cohort:
                if face_detector(face):
                    face_detected += 1

            print(f'% human faces detected in {key} = {round(face_detected/len(human_file
        s_short) * 100, 2)}')
```

```
% human faces detected in human_files_short = 97.0
% human faces detected in dog_files_short = 15.0
```

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

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

# Try a pretrained classifier - MTCNN
# (See https://machinelearningmastery.com/how-to-perform-face-detection-with-clas
sical-and-deep-learning-methods-in-python-with-keras/)

# face detection with mtcnn on a photograph
from matplotlib import pyplot
from matplotlib.patches import Rectangle
from mtcnn.mtcnn import MTCNN
import tensorflow as tf

tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)

def deep_face_detector(image):
    # load image from file
    pixels = pyplot.imread(image)
    # create the detector, using default weights
    detector = MTCNN()
    # detect faces in the image
    faces = detector.detect_faces(pixels)
    # display faces on the original image
    return 1 if faces else 0

for key, cohort in {'human_files_short': human_files_short, 'dog_files_short': do
g_files_short}.items():
    face_detected = 0
    for face in cohort:
        if deep_face_detector(face):
            face_detected += 1

    print(f'% human faces detected in {key} = {round(face_detected/len(human_file
s_short) * 100, 2)}')
```

```
% human faces detected in human_files_short = 100.0
% human faces detected in dog_files_short = 33.0
```

*Interesting that this ML face detector still detects 17% of dog images as having human faces, compared to 18% using* `face_cascade` *in* `cv2`. When I actually look through the 100 dog files, a number of them do have humans in the frame that are being detected. However, some of the dog faces *are* being detected as human, so the false positive rate should be less than 17%, but still greater than zero. Looking through the images, five of the 17 mis-identified images actually have humans in them, so the real false positive value is 12%.

Also, on Linux the percentages are different:

| Type of Image | % Det Windows - CV2 | % Det Windows - MTCNN | % Det Linux - CV2 | % Det Linux - MTCNN |
|---|---|---|---|---|
| Human | 96.0 | 100.0 | 97.0 | 100.0 |
| Dog | 18.0 | 17.0 | 15.0 | 33.0 |

The performance on the dog dataset under Linux for MTCNN method was very bad. I'm using the latest Anaconda distro for both my Linux and Windows machines, so I don't know what might account for the difference.

> NOTE: I did not perform any image tweaking prior to feeding images into MTCNN. May come back to this and try converting to grayscale before passing image into detector.

## Step 2: Detect Dogs

In this section, we use a [pre-trained model (http://pytorch.org/docs/master/torchvision/models.html)](http://pytorch.org/docs/master/torchvision/models.html) 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 (http://www.image-net.org/)](http://www.image-net.org/), 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 (https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a)](https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a).

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

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

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

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

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /home/t
homas/.cache/torch/hub/checkpoints/vgg16-397923af.pth
```

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 (http://pytorch.org/docs/stable/torchvision/models.html)](http://pytorch.org/docs/stable/torchvision/models.html).

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

# Set PIL to be tolerant of image files that are truncated.
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

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

    Args:
        img_path: path to an image

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

    ## 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
    input_image = Image.open(img_path)
    preprocess = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.22
5]),
        ])

    input_image = Image.open(img_path)
    input_image = preprocess(input_image)
    input_image = Variable(input_image)
    input_image = input_image.unsqueeze(0)

    if torch.cuda.is_available():
        input_image = input_image.to('cuda')
        VGG16.to('cuda')

    VGG16.eval()
#     with torch.no_grad():
    output = VGG16(input_image)
    # Tensor of shape 1000, with confidence scores over Imagenet's 1000 classes
#     print(output[0])
    _, index = output[0].max(0)
#     print(index)

    return int(index.numpy()) # predicted class index
```

In [9]:
```
with open("imagenet1000_clsidx_to_labels.txt") as f:
    idx2label = eval(f.read())
```

In [10]:
```
pred = VGG16_predict('dogImages/train/016.Beagle/Beagle_01132.jpg')

print(idx2label[pred])
```

```
beagle
```

## (IMPLEMENTATION) Write a Dog Detector

While looking at the dictionary (https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a), 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 [11]:
```
### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    ## TODO: Complete the function.
    pred = VGG16_predict(img_path)

    dog_detected = False
    if pred in range(151, 269):
        dog_detected = True

    return dog_detected # true/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:** The dog detector correctly identified 94% of the dogs in `dog_files_short` , with no false positives (0%) in the cohort of human images in `human_files_short` detected as dogs.

In [12]:
```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.

for key, cohort in {'human_files_short': human_files_short, 'dog_files_short': do
g_files_short}.items():
    dog_detected = 0
    for image in cohort:
        if dog_detector(image):
            dog_detected += 1

    print(f'% dogs detected in {key} = {round(dog_detected/len(human_files_short)
* 100, 2)}')
```

```
% dogs detected in human_files_short = 0.0
% dogs detected in dog_files_short = 99.0
```

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 (http://pytorch.org/docs/master/torchvision/models.html#inception-v3), ResNet-50 (http://pytorch.org/docs/master/torchvision/models.html#id3), 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 [13]:
```
dir(models)
```

```
Out[13]: ['AlexNet',
          'DenseNet',
          'GoogLeNet',
          'GoogLeNetOutputs',
          'Inception3',
          'InceptionOutputs',
          'MNASNet',
          'MobileNetV2',
          'ResNet',
          'ShuffleNetV2',
          'SqueezeNet',
          'VGG',
          '_GoogLeNetOutputs',
          '_InceptionOutputs',
          '__builtins__',
          '__cached__',
          '__doc__',
          '__file__',
          '__loader__',
          '__name__',
          '__package__',
          '__path__',
          '__spec__',
          '_utils',
          'alexnet',
          'densenet',
          'densenet121',
          'densenet161',
          'densenet169',
          'densenet201',
          'detection',
          'googlenet',
          'inception',
          'inception_v3',
          'mnasnet',
          'mnasnet0_5',
          'mnasnet0_75',
          'mnasnet1_0',
          'mnasnet1_3',
          'mobilenet',
          'mobilenet_v2',
          'quantization',
          'resnet',
          'resnet101',
          'resnet152',
          'resnet18',
          'resnet34',
          'resnet50',
          'resnext101_32x8d',
          'resnext50_32x4d',
          'segmentation',
          'shufflenet_v2_x0_5',
          'shufflenet_v2_x1_0',
          'shufflenet_v2_x1_5',
          'shufflenet_v2_x2_0',
          'shufflenetv2',
          'squeezenet',
          'squeezenet1_0',
          'squeezenet1_1',
          'utils',
          'vgg',
          'vgg11',
          'vgg11_bn',
          'vgg13',
          'vgg13_bn',
          'vgg16',
          'vgg16_bn',
          'vgg19',
          'vgg19_bn',
          'video',
          'wide_resnet101_2',
          'wide_resnet50_2']
```

In [14]:

```python
### (Optional)
### TODO: Report the performance of another pre-trained network.
### Feel free to use as many code cells as needed.
# define VGG16 model
model = models.wide_resnet101_2(pretrained=True)

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

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

```
Downloading: "https://download.pytorch.org/models/wide_resnet101_2-32ee1156.pth"
to /home/thomas/.cache/torch/hub/checkpoints/wide_resnet101_2-32ee1156.pth
```

In [21]:
```
from PIL import Image
import torchvision.transforms as transforms
from torch.autograd import Variable

# Set PIL to be tolerant of image files that are truncated.
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

def model_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
    input_image = Image.open(img_path)
    preprocess = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.22
5]),
        ])

    input_image = Image.open(img_path)
    input_image = preprocess(input_image)
    input_image = Variable(input_image)
    input_image = input_image.unsqueeze(0)

    if torch.cuda.is_available():
        input_image = input_image.to('cuda')
        model.to('cuda')

    model.eval()
    output = model(input_image)

    _, index = output[0].max(0)

    return int(index.numpy()) # predicted class index
```

In [22]:
```
### returns "True" if a dog is detected in the image stored at img_path
def model_dog_detector(img_path):
    ## TODO: Complete the function.
    pred = model_predict(img_path)

    dog_detected = False
    if pred in range(151, 269):
        dog_detected = True

    return dog_detected # true/false
```

```
In [23]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.

         for key, cohort in {'dog_files_short': dog_files_short, 'human_files_short': huma
         n_files_short}.items():
             dog_detected = 0
             for image in cohort:
                 if model_dog_detector(image):
                     dog_detected += 1

             print(f'% dogs detected in {key} = {round(dog_detected/len(human_files_short)
         * 100, 2)}')
```

```
% dogs detected in dog_files_short = 99.0
% dogs detected in human_files_short = 0.0
```

### Results Table

Acceptable results taken to be ≥ VGG16 results. I could not get the `inception` model to run even after adjusting transforms to input 299x299 pixel images as specified, so I ran a bunch of others while doing other things.

| Model | % Dogs Detected | % Humans Detected | Acceptable? |
|---|---|---|---|
| **VGG16** | **94.0** | **0.0** | ✓ |
| squeezenet1_1 | 96.0 | 1.0 | ✓ |
| googlenet | 93.0 | 0.0 | X |
| densenet201 | 97.0 | 0.0 | ✓ |
| resnet50 | 95.0 | 0.0 | ✓ |
| resnet101 | 99.0 | 0.0 | ✓ |
| resnet152 | 92.0 | 1.0 | X |
| alexnet | 96.0 | 1.0 | ✓ |
| mnasnet1_0 | 93.0 | 0.0 | X |
| mobilenet_v2 | 94.0 | 0.0 | ✓ |
| shufflenet_v2_x1_0 | 95.0 | 1.0 | ✓ |
| wide_resnet101_2 | 95.0 | 0.0 | ✓ |

## 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.

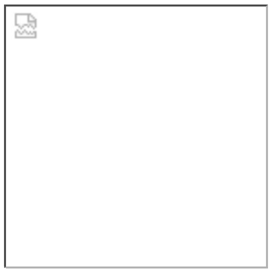
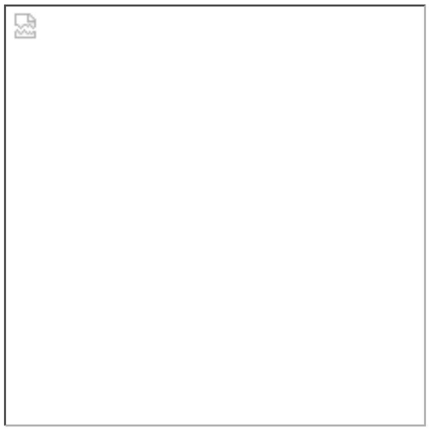
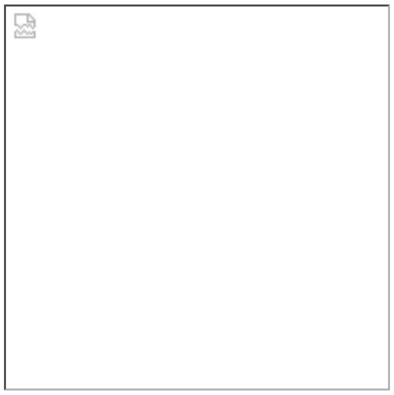
| Brittany | Welsh Springer Spaniel |
|---|---|
| | |

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 |
|---|---|---|

| 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%.

In [24]:
```
import os
from torchvision import datasets
import torchvision.transforms as transforms
import torch

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

# Much of this drawn from video lessons

batch_size = 32
num_workers = 0

data_dir = 'dogImages'

train_dir = os.path.join(data_dir, 'train')
test_dir = os.path.join(data_dir, 'test')
valid_dir = os.path.join(data_dir, 'valid')

# Include minor augmentation in train transforms
train_transform = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.RandomRotation(30),
        transforms.RandomAffine(0, translate=(0.2, 0.2)),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.22
5])
])

# To be used for test & validation - no augmentation
test_transform = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.22
5])
])

train_data = datasets.ImageFolder(train_dir, transform=train_transform)
test_data = datasets.ImageFolder(test_dir, transform=test_transform)
valid_data = datasets.ImageFolder(valid_dir, transform=test_transform)

# Had to refactor this into a dict...
loaders_scratch = {'train': torch.utils.data.DataLoader(train_data, batch_size=ba
tch_size, num_workers=num_workers, shuffle=True),
                   'valid': torch.utils.data.DataLoader(train_data, batch_size=ba
tch_size, num_workers=num_workers, shuffle=False),
                   'test': torch.utils.data.DataLoader(test_data, batch_size=batc
h_size, num_workers=num_workers, shuffle=False)}
```

**Question 3:** Describe your chosen procedure for preprocessing the data.

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

**Answer**:

I used the standard resize and crop dimensions. The final height and width dimension of 224 matches what is used in the majority of the models trained on ImageNet, with the exception of `inception` . If my model happens to perform fantastically well (ha!), I would like other people to be able to use it without modification. Resizing down to 256x256 before the final crop ensures that minimal information is lost, while focusing on the subject of the image, which is assumed to be centered. I used the means and standard deviations for the ImageNet database for normalization.

For the training data I specified a random rotation of 30 degrees, as well as a random horizontal flip. These transforms were added to augment the data and aid in preventing overfitting. I also added random horizontal and vertical translation to the training data transforms, again to avoid overfitting.

The test and validation data is not augmented since that data is not used to modify the weights, and it needs to remain the same to accurately track loss and accuracy.

## (IMPLEMENTATION) Model Architecture

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

In [25]:
```
# Re-adding stuff as necessary so I don't have to re-run everything above on kern
el restarts
import numpy as np
# check if CUDA is available
use_cuda = torch.cuda.is_available()
```

In [26]:
```
classes = len(next(os.walk(train_dir))[1])
```

```
In [27]: 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, 16, 3, stride=2, padding=1)
                 self.conv2 = nn.Conv2d(16, 32, 3, stride=2, padding=1)
                 self.conv3 = nn.Conv2d(32, 64, 3, stride = 2, padding=1)
                 self.conv4 = nn.Conv2d(64, 128, 3, padding=1)
                 self.pool = nn.MaxPool2d(2, 2)
                 self.fc1 = nn.Linear(1 * 1 * 128, 500) # I *think* this will be 1 * 1 * 1
         28 w/ 4th layer
                 self.fc2 = nn.Linear(500, classes)
                 self.dropout = nn.Dropout(0.2)

             def forward(self, x):
                 ## Define forward behavior
                 x = self.pool(F.relu(self.conv1(x)))
                 x = self.pool(F.relu(self.conv2(x)))
                 x = self.pool(F.relu(self.conv3(x)))
                 x = self.pool(F.relu(self.conv4(x)))

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

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

         model_scratch
```

```
Out[27]: Net(
           (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
           (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
           (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
           (conv4): 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=Fa
         lse)
           (fc1): Linear(in_features=128, out_features=500, bias=True)
           (fc2): Linear(in_features=500, out_features=133, bias=True)
           (dropout): Dropout(p=0.2, inplace=False)
         )
```

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

**Answer:** I started with a similar architecture to the one used in the `cifar10` exercise, but I wanted to add one more convolutional layer for better performance since the input images are in color and categorizing breeds requires more features than just detecting whether an image is of a dog.

I stuck with a standard kernel of 3 and padding of 1, with a stride of 2 for the first three `Conv2d` layers. The last layer has a default stride of 1 since the output of the convolutional and pooling layers had already gotten down to a size of 1x1x128. If I had run into issues with training I planned to go back and change the stride on one or more previous layers, but the model trained well.

Some dropout as added before each fully connected layer to mitigate overfitting.

The final fully connected layer has an output of `classes`, which in this case is 133, the number of dog breeds represented in the training data.

## (IMPLEMENTATION) Specify Loss Function and Optimizer

Use the next code cell to specify a [loss function (http://pytorch.org/docs/stable/nn.html#loss-functions)](http://pytorch.org/docs/stable/nn.html#loss-functions) and [optimizer (http://pytorch.org/docs/stable/optim.html)](http://pytorch.org/docs/stable/optim.html). Save the chosen loss function as `criterion_scratch`, and the optimizer as `optimizer_scratch` below.

In [28]:
```
import torch.optim as optim

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

### TODO: select optimizer
optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=0.1)
# Tried lr = 0.01, but training not going well
```

## (IMPLEMENTATION) Train and Validate the Model

Train and validate your model in the code cell below. [Save the final model parameters (http://pytorch.org/docs/master/notes/serialization.html)](http://pytorch.org/docs/master/notes/serialization.html) at filepath `'model_scratch.pt'`.

```
In [29]: # the following import is required for training to be robust to truncated images
         import time
         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):
                 epoch_start = time.time()

                 # 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']):
                     batch_start = time.time()
                     # move to GPU
                     if use_cuda:
                         data, target = data.cuda(), target.cuda()

                     optimizer.zero_grad()

                     ## find the loss and update the model parameters accordingly
                     output = model(data)
                     loss = criterion(output, target)
                     loss.backward()
                     optimizer.step()

                     ## record the average training loss, using something like
                     train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train
         _loss))

                     # Occasionally print progress
                     if (batch_idx+1) % 100 == 0:
                         print(f'Epoch: {epoch}, Batch: {batch_idx + 1}, Training loss: {t
         rain_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}, \tEpo
         ch Time: {:.2f}'.format(
            epoch,
            train_loss,
            valid_loss,
            time.time() - epoch_start
            ))

        ## 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


# train the model
model_scratch = train(30, 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, Batch: 100, Training loss: 4.887012481689453
Epoch: 1, Batch: 200, Training loss: 4.881507396697998
Epoch: 1        Training Loss: 4.881681         Validation Loss: 4.860235,
Epoch Time: 343.91
Validation loss decreased (inf --> 4.860235).  Saving model ...
Epoch: 2, Batch: 100, Training loss: 4.8586225509643555
Epoch: 2, Batch: 200, Training loss: 4.849359512329102
Epoch: 2        Training Loss: 4.849445         Validation Loss: 4.800694,
Epoch Time: 267.83
Validation loss decreased (4.860235 --> 4.800694).  Saving model ...
Epoch: 3, Batch: 100, Training loss: 4.7903008460998535
Epoch: 3, Batch: 200, Training loss: 4.785928249359131
Epoch: 3        Training Loss: 4.785944         Validation Loss: 4.715376,
Epoch Time: 267.19
Validation loss decreased (4.800694 --> 4.715376).  Saving model ...
Epoch: 4, Batch: 100, Training loss: 4.741987705230713
Epoch: 4, Batch: 200, Training loss: 4.728640079498291
Epoch: 4        Training Loss: 4.723383         Validation Loss: 4.710718,
Epoch Time: 266.60
Validation loss decreased (4.715376 --> 4.710718).  Saving model ...
Epoch: 5, Batch: 100, Training loss: 4.681036949157715
Epoch: 5, Batch: 200, Training loss: 4.659914970397949
Epoch: 5        Training Loss: 4.658728         Validation Loss: 4.565597,
Epoch Time: 266.72
Validation loss decreased (4.710718 --> 4.565597).  Saving model ...
Epoch: 6, Batch: 100, Training loss: 4.572766304016113
Epoch: 6, Batch: 200, Training loss: 4.554416656494141
Epoch: 6        Training Loss: 4.551991         Validation Loss: 4.518350,
Epoch Time: 267.59
Validation loss decreased (4.565597 --> 4.518350).  Saving model ...
Epoch: 7, Batch: 100, Training loss: 4.490131855010986
Epoch: 7, Batch: 200, Training loss: 4.491408348083496
Epoch: 7        Training Loss: 4.491200         Validation Loss: 4.385848,
Epoch Time: 266.98
Validation loss decreased (4.518350 --> 4.385848).  Saving model ...
Epoch: 8, Batch: 100, Training loss: 4.4277262687683105
Epoch: 8, Batch: 200, Training loss: 4.437042236328125
Epoch: 8        Training Loss: 4.435461         Validation Loss: 4.423955,
Epoch Time: 268.03
Epoch: 9, Batch: 100, Training loss: 4.399413108825684
Epoch: 9, Batch: 200, Training loss: 4.401973724365234
Epoch: 9        Training Loss: 4.402404         Validation Loss: 4.266252,
Epoch Time: 267.67
Validation loss decreased (4.385848 --> 4.266252).  Saving model ...
Epoch: 10, Batch: 100, Training loss: 4.350475788116455
Epoch: 10, Batch: 200, Training loss: 4.341382026672363
Epoch: 10       Training Loss: 4.346916         Validation Loss: 4.270985,
Epoch Time: 267.78
Epoch: 11, Batch: 100, Training loss: 4.325940132141113
Epoch: 11, Batch: 200, Training loss: 4.313450813293457
Epoch: 11       Training Loss: 4.310138         Validation Loss: 4.334950,
Epoch Time: 266.42
Epoch: 12, Batch: 100, Training loss: 4.296782970428467
Epoch: 12, Batch: 200, Training loss: 4.293903827667236
Epoch: 12       Training Loss: 4.295736         Validation Loss: 4.412406,
Epoch Time: 266.66
Epoch: 13, Batch: 100, Training loss: 4.216949939727783
Epoch: 13, Batch: 200, Training loss: 4.229761600494385
Epoch: 13       Training Loss: 4.232775         Validation Loss: 4.119966,
Epoch Time: 266.81
Validation loss decreased (4.266252 --> 4.119966).  Saving model ...
Epoch: 14, Batch: 100, Training loss: 4.1996331214904785
Epoch: 14, Batch: 200, Training loss: 4.200648307800293
```

```
Epoch: 14       Training Loss: 4.200630         Validation Loss: 4.344113,
Epoch Time: 266.98
Epoch: 15, Batch: 100, Training loss: 4.178210258483887
Epoch: 15, Batch: 200, Training loss: 4.164714813232422
Epoch: 15       Training Loss: 4.170273         Validation Loss: 4.093867,
Epoch Time: 266.73
Validation loss decreased (4.119966 --> 4.093867).  Saving model ...
Epoch: 16, Batch: 100, Training loss: 4.1077399253845215
Epoch: 16, Batch: 200, Training loss: 4.101799488067627
Epoch: 16       Training Loss: 4.105224         Validation Loss: 3.986394,
Epoch Time: 266.31
Validation loss decreased (4.093867 --> 3.986394).  Saving model ...
Epoch: 17, Batch: 100, Training loss: 4.0987162590026855
Epoch: 17, Batch: 200, Training loss: 4.078978061676025
Epoch: 17       Training Loss: 4.086902         Validation Loss: 3.961097,
Epoch Time: 267.45
Validation loss decreased (3.986394 --> 3.961097).  Saving model ...
Epoch: 18, Batch: 100, Training loss: 4.0458083152771
Epoch: 18, Batch: 200, Training loss: 4.03560209274292
Epoch: 18       Training Loss: 4.041360         Validation Loss: 3.923707,
Epoch Time: 266.99
Validation loss decreased (3.961097 --> 3.923707).  Saving model ...
Epoch: 19, Batch: 100, Training loss: 4.002011299133301
Epoch: 19, Batch: 200, Training loss: 4.0165886878967285
Epoch: 19       Training Loss: 4.016042         Validation Loss: 4.138726,
Epoch Time: 267.61
Epoch: 20, Batch: 100, Training loss: 3.9724245071411133
Epoch: 20, Batch: 200, Training loss: 3.984074354171753
Epoch: 20       Training Loss: 3.978470         Validation Loss: 3.810425,
Epoch Time: 266.30
Validation loss decreased (3.923707 --> 3.810425).  Saving model ...
Epoch: 21, Batch: 100, Training loss: 3.920644760131836
Epoch: 21, Batch: 200, Training loss: 3.9318044185638428
Epoch: 21       Training Loss: 3.931187         Validation Loss: 3.982528,
Epoch Time: 266.49
Epoch: 22, Batch: 100, Training loss: 3.8853700160980225
Epoch: 22, Batch: 200, Training loss: 3.9099483489990234
Epoch: 22       Training Loss: 3.908142         Validation Loss: 3.834654,
Epoch Time: 267.30
Epoch: 23, Batch: 100, Training loss: 3.846562147140503
Epoch: 23, Batch: 200, Training loss: 3.86418080329895
Epoch: 23       Training Loss: 3.864014         Validation Loss: 3.876613,
Epoch Time: 266.87
Epoch: 24, Batch: 100, Training loss: 3.817936897277832
Epoch: 24, Batch: 200, Training loss: 3.8380484580993652
Epoch: 24       Training Loss: 3.844681         Validation Loss: 3.637212,
Epoch Time: 266.62
Validation loss decreased (3.810425 --> 3.637212).  Saving model ...
Epoch: 25, Batch: 100, Training loss: 3.851989269256592
Epoch: 25, Batch: 200, Training loss: 3.8395800590515137
Epoch: 25       Training Loss: 3.833626         Validation Loss: 3.609095,
Epoch Time: 266.55
Validation loss decreased (3.637212 --> 3.609095).  Saving model ...
Epoch: 26, Batch: 100, Training loss: 3.781928300857544
Epoch: 26, Batch: 200, Training loss: 3.788914203643799
Epoch: 26       Training Loss: 3.786834         Validation Loss: 3.537222,
Epoch Time: 266.00
Validation loss decreased (3.609095 --> 3.537222).  Saving model ...
Epoch: 27, Batch: 100, Training loss: 3.7894515991210938
Epoch: 27, Batch: 200, Training loss: 3.781296491622925
Epoch: 27       Training Loss: 3.778372         Validation Loss: 3.660208,
Epoch Time: 266.79
Epoch: 28, Batch: 100, Training loss: 3.743389368057251
Epoch: 28, Batch: 200, Training loss: 3.727191686630249
```

```
Epoch: 28       Training Loss: 3.728818         Validation Loss: 3.663486,
Epoch Time: 266.46
Epoch: 29, Batch: 100, Training loss: 3.7178826332092285
Epoch: 29, Batch: 200, Training loss: 3.722106456756592
Epoch: 29       Training Loss: 3.723890         Validation Loss: 3.529714,
Epoch Time: 266.64
Validation loss decreased (3.537222 --> 3.529714).  Saving model ...
Epoch: 30, Batch: 100, Training loss: 3.667236089706421
Epoch: 30, Batch: 200, Training loss: 3.6861391067504883
Epoch: 30       Training Loss: 3.685707         Validation Loss: 3.441438,
Epoch Time: 267.41
```

Out[29]:
```
<All keys matched successfully>
```

### (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 [30]:
```
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().nu
mpy())
        total += data.size(0)

    print('Test Loss: {:.6f}\n'.format(test_loss))

    print('\nTest Accuracy: %2d%% (%2d/%2d)' % (
        100. * correct / total, correct, total))

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

```
Test Loss: 3.801882


Test Accuracy: 12% (103/836)
```

## 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.

### (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 `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 [31]:
```
## TODO: Specify data loaders
loaders_transfer = loaders_scratch # No need to make a copy as no mods are made
```

### (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 [32]:
```
import torchvision.models as models
import torch.nn as nn

## TODO: Specify model architecture

# densenet201 performed the best on dog detection, so I'll use that as a starter
model_transfer = models.densenet201(pretrained=True)

# Not training the feature detection parameters
for param in model_transfer.features.parameters():
    param.requires_grad = False

if use_cuda:
    model_transfer = model_transfer.cuda()

# Change the classifier to give the approprite number of outputs
model_transfer.classifier = nn.Linear(1920, classes)
```

```
Downloading: "https://download.pytorch.org/models/densenet201-c1103571.pth" to /
home/thomas/.cache/torch/hub/checkpoints/densenet201-c1103571.pth
```

In [33]:
```
model_transfer.classifier
```

Out[33]:
```
Linear(in_features=1920, 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 used `densenet201` as a starting point as it was the top performer in my survey of models when implementing the optional pre-trained networks.

The feature parameters were frozen since those were not to be trained.

The classifier was replaced with a linear fully connected layer with an input size of 1920 to match the output of the `densenet201` features section, and an output size of `classes`, which in this case is 133, the number of dog breeds represented in the training data.

## (IMPLEMENTATION) Specify Loss Function and Optimizer

Use the next code cell to specify a [loss function (http://pytorch.org/docs/master/nn.html#loss-functions)](http://pytorch.org/docs/master/nn.html#loss-functions) and [optimizer (http://pytorch.org/docs/master/optim.html)](http://pytorch.org/docs/master/optim.html). Save the chosen loss function as `criterion_transfer`, and the optimizer as `optimizer_transfer` below.

In [34]:
```
# Use CrossEntropyLoss since this is a classification task
criterion_transfer = nn.CrossEntropyLoss()
optimizer_transfer = optim.SGD(model_transfer.classifier.parameters(), lr=0.01)
```

## (IMPLEMENTATION) Train and Validate the Model

Train and validate your model in the code cell below. [Save the final model parameters (http://pytorch.org/docs/master/notes/serialization.html)](http://pytorch.org/docs/master/notes/serialization.html) at filepath `'model_transfer.pt'`.

In [35]:
```
# check if CUDA is available
use_cuda = torch.cuda.is_available()

if use_cuda:
    model_transfer = model_transfer.cuda()

# train the model
model_transfer = train(8, loaders_transfer, model_transfer, optimizer_transfer, c
riterion_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'))
```

```
Epoch: 1, Batch: 100, Training loss: 4.6187543869018555
Epoch: 1, Batch: 200, Training loss: 4.315049648284912
Epoch: 1        Training Loss: 4.291208         Validation Loss: 3.550349,
Epoch Time: 2804.14
Validation loss decreased (inf --> 3.550349).  Saving model ...
Epoch: 2, Batch: 100, Training loss: 3.4026267528533936
Epoch: 2, Batch: 200, Training loss: 3.205131769180298
Epoch: 2        Training Loss: 3.186797         Validation Loss: 2.606934,
Epoch Time: 2790.12
Validation loss decreased (3.550349 --> 2.606934).  Saving model ...
Epoch: 3, Batch: 100, Training loss: 2.567464828491211
Epoch: 3, Batch: 200, Training loss: 2.4435410499572754
Epoch: 3        Training Loss: 2.432548         Validation Loss: 1.939937,
Epoch Time: 2796.81
Validation loss decreased (2.606934 --> 1.939937).  Saving model ...
Epoch: 4, Batch: 100, Training loss: 2.0151662826538086
Epoch: 4, Batch: 200, Training loss: 1.9437682628631592
Epoch: 4        Training Loss: 1.939593         Validation Loss: 1.614605,
Epoch Time: 2795.55
Validation loss decreased (1.939937 --> 1.614605).  Saving model ...
Epoch: 5, Batch: 100, Training loss: 1.6745928525924683
Epoch: 5, Batch: 200, Training loss: 1.6217467784881592
Epoch: 5        Training Loss: 1.613315         Validation Loss: 1.303564,
Epoch Time: 2791.58
Validation loss decreased (1.614605 --> 1.303564).  Saving model ...
Epoch: 6, Batch: 100, Training loss: 1.4386097192764282
Epoch: 6, Batch: 200, Training loss: 1.3955378532409668
Epoch: 6        Training Loss: 1.391597         Validation Loss: 1.137358,
Epoch Time: 2788.80
Validation loss decreased (1.303564 --> 1.137358).  Saving model ...
Epoch: 7, Batch: 100, Training loss: 1.295832633972168
Epoch: 7, Batch: 200, Training loss: 1.2459064722061157
Epoch: 7        Training Loss: 1.244691         Validation Loss: 1.010389,
Epoch Time: 2761.49
Validation loss decreased (1.137358 --> 1.010389).  Saving model ...
Epoch: 8, Batch: 100, Training loss: 1.128366470336914
Epoch: 8, Batch: 200, Training loss: 1.1170756816864014
Epoch: 8        Training Loss: 1.118789         Validation Loss: 0.931129,
Epoch Time: 2789.11
Validation loss decreased (1.010389 --> 0.931129).  Saving model ...
```

Out[35]:
```
<All keys matched successfully>
```

## (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 [36]:
```
test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 1.205510


Test Accuracy: 82% (686/836)
```

## (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 [37]:
```
plt.rcParams["figure.figsize"] = (8, 6)
```

In [38]:
```
from PIL import Image
from torch.autograd import Variable

def preprocess_image(img_path):
    preprocess_transform = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.22
5]),
    ])

    input_image = Image.open(img_path)
    input_image = preprocess_transform(input_image)
    input_image = Variable(input_image)
    input_image = input_image.unsqueeze(0)

    return input_image
```

In [39]:
```
### 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 loaders_transfer['train'].d
ataset.classes]

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

    if torch.cuda.is_available():
        image = image.to('cuda')
        model_transfer.to('cuda')

    model_transfer.eval()
    output = model_transfer(image)

    _, index = output[0].max(0)
    index = int(index.cpu().numpy())

    return class_names[index]
```

## 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!

Sample Human Output

### (IMPLEMENTATION) Write your Algorithm

In [40]:
```
### 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
    if dog_detector(img_path):
        title = 'Dog detected!'
        xlabel = f'I think the breed is {predict_breed_transfer(img_path).lower
()}'

    elif face_detector(img_path):
        title = 'Hooman detected!'
        xlabel = f'That hooman looks like a[n] {predict_breed_transfer(img_path).
lower()}'

    else:
        title = 'I don\'t know what that is!'
        xlabel = f'I don\'t know what that is.\nWhat is that thing?\nI don\'t kno
w, but I want my picture taken with it. (Steve Martin)'

    fig, ax = plt.subplots()
    ax.imshow(Image.open(img_path))
    ax.set_xlabel(xlabel)
    ax.set_title(title);
```

## 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.

**Answer:** (Three possible points for improvement) The output is better than I expected in terms of detecting dogs vs humans vs other things. It does reasonably well on determining the breed of dog, though as noted in the instructions the differences between many breeds is subtle.

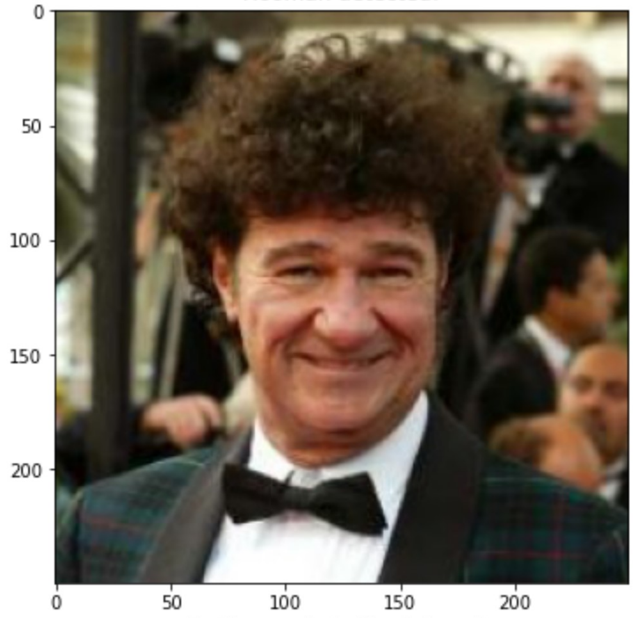
Some points for improvement:

- The face detector does not do well on images that are not front-facing and centered. More effort could be put into preprocessing images to center the faces.
- Train `model_transfer` for more epochs. I only trained for five full epochs before stopping the training in the interest of time. The training loss was still decreasing, so performance should improve with more training. I will train it for eight epochs before submitting the final project.
- `densenet201` is capable of detecting all 1000 classes in the ImageNet database. It would be interesting to use an unmodified model to attempt to identify the objects that are not human faces or dogs, rather than using a generic "I don't know" output.
- [Extra] I had planned to have a `%reset -f` below Step 5 and re-import all of the necessary modules, but there was a lot of infrastructure that would need to be repeated or refactored, such as the train loaders and so forth, so I did not do that. I may come back to this and use the `model_transfer` checkpoint to create a standalone "app".

In [41]:

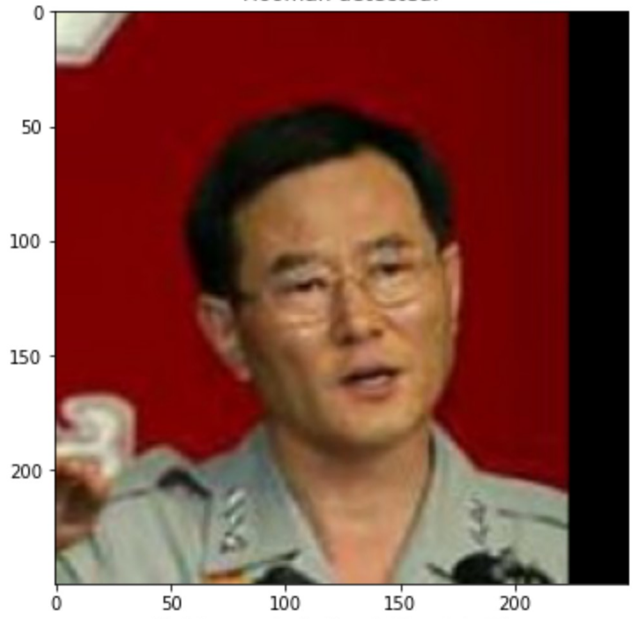
```python
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.

## suggested code, below
for file in np.hstack((human_files[:3], dog_files[:3])):
    run_app(file)
```

Hooman detected!



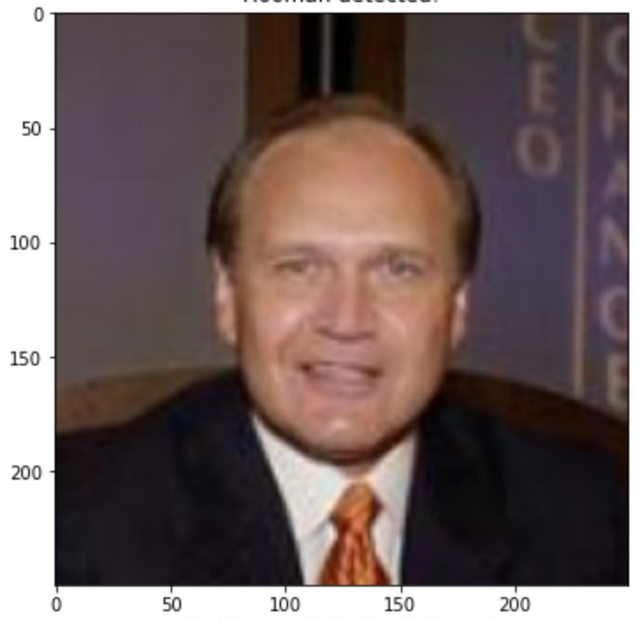
That hooman looks like a[n] poodle

Hooman detected!



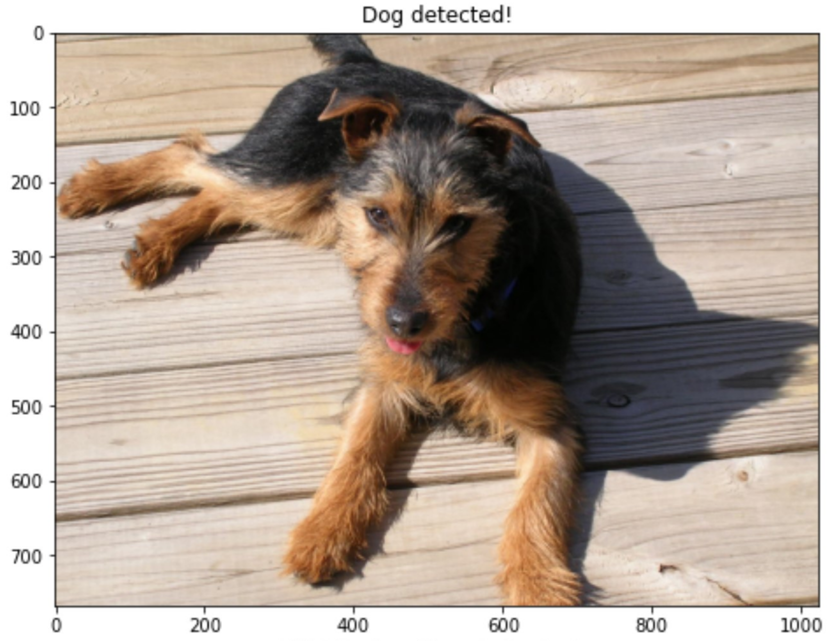
That hooman looks like a[n] french bulldog

Hooman detected!



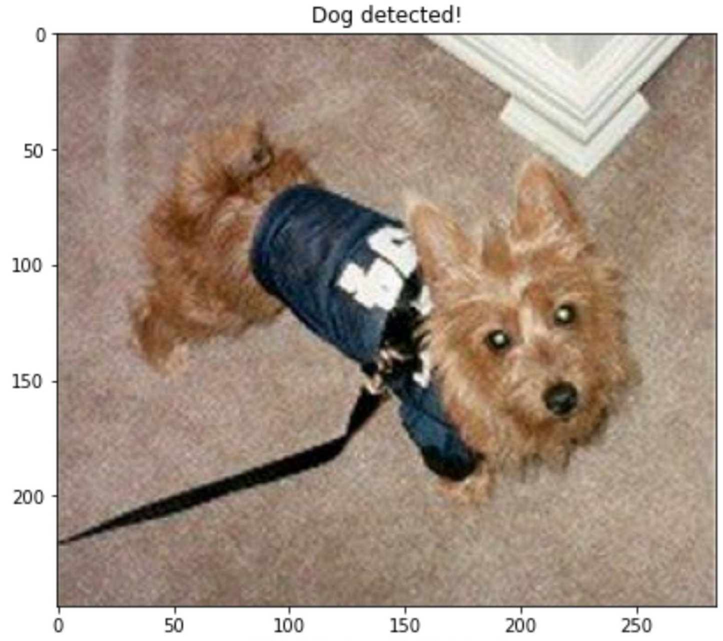
That hooman looks like a[n] basenji

Dog detected!



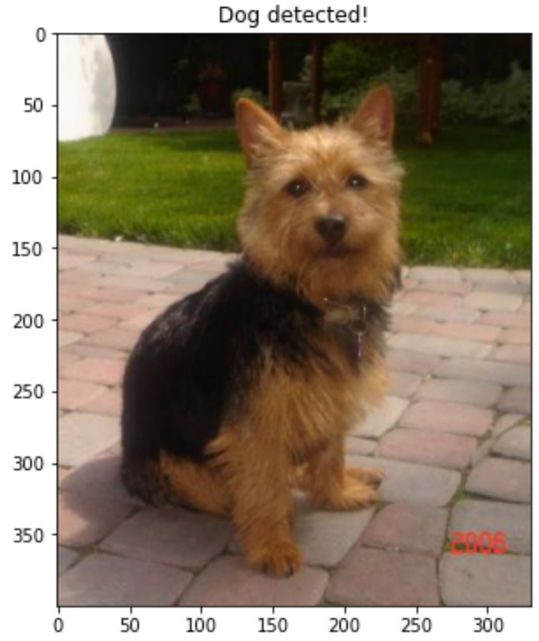
I think the breed is australian terrier

Dog detected!



I think the breed is irish terrier

Dog detected!

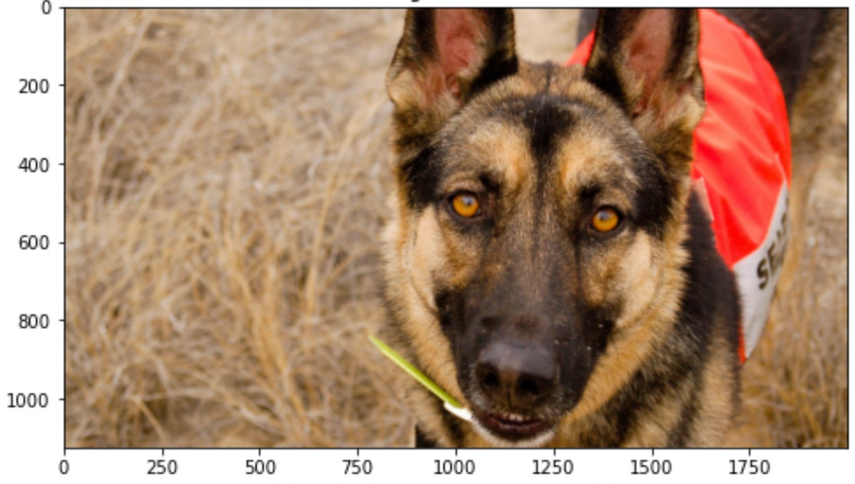


I think the breed is australian terrier

In [42]:

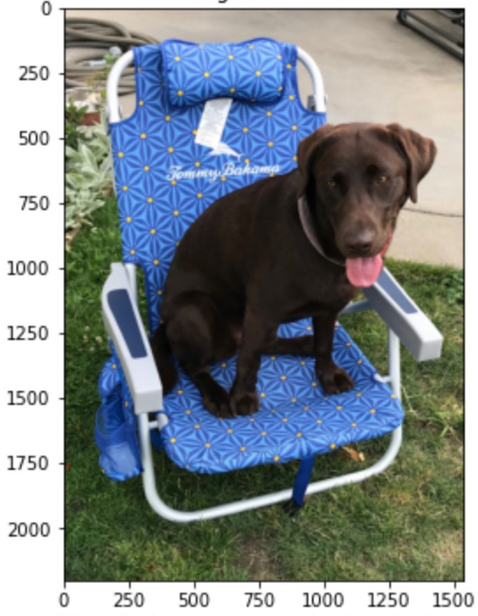
```
# My own set of images, mainly taken from the web
# Close-up faces from https://www.thispersondoesnotexist.com/

# load filenames for human and dog images
test_files = np.array(glob("appTestImages/*"))
# print (test_files)

for file in test_files:
    run_app(file)
```

Dog detected!



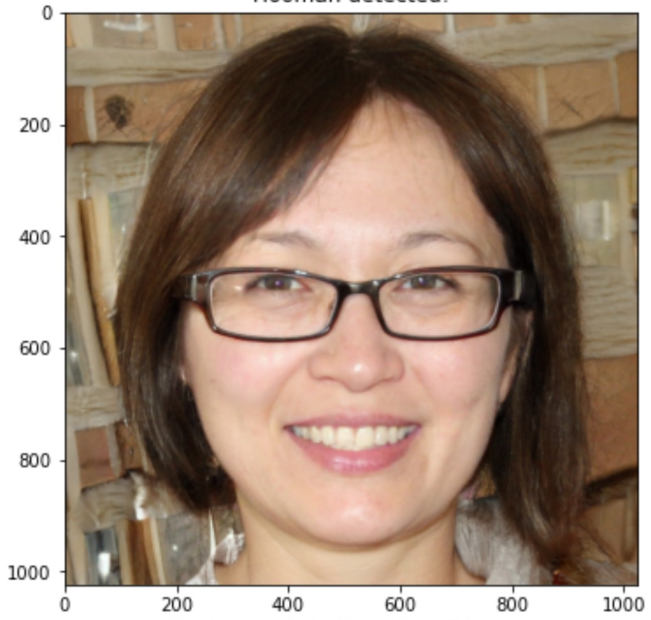
I think the breed is german shepherd dog

Dog detected!



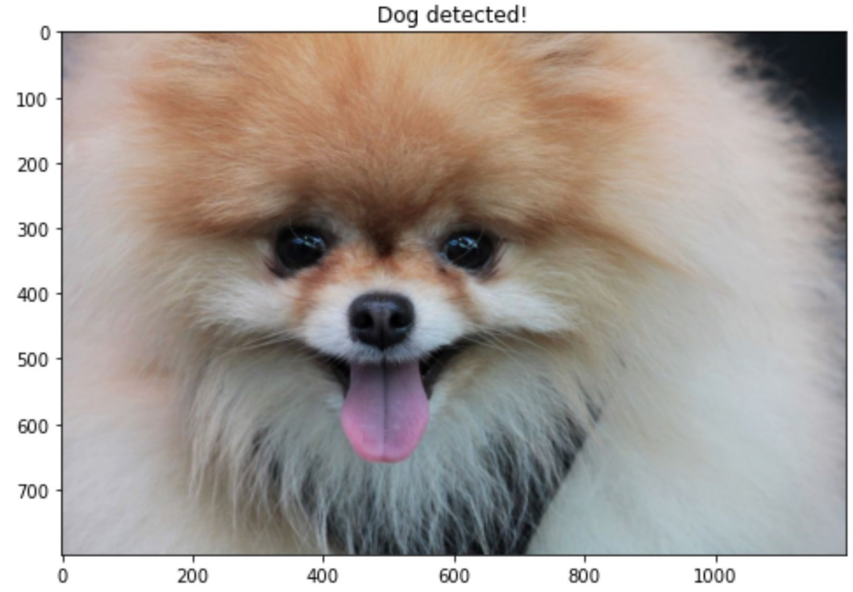
I think the breed is german shorthaired pointer

Hooman detected!

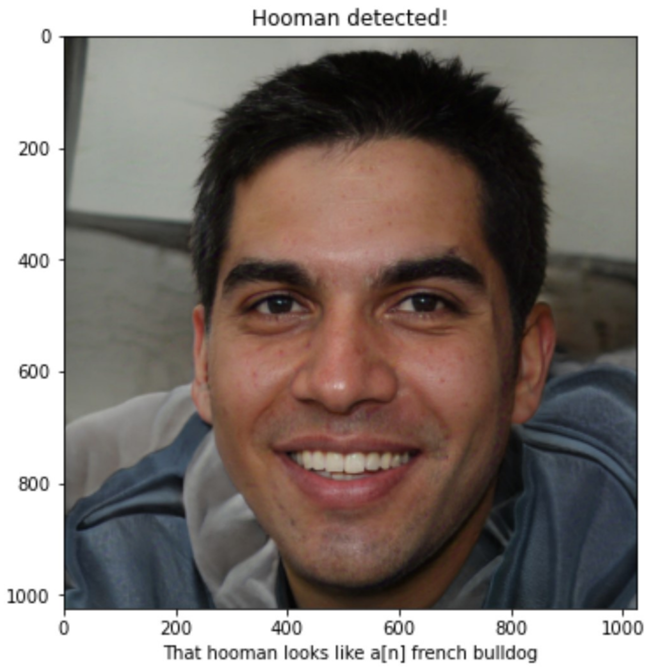


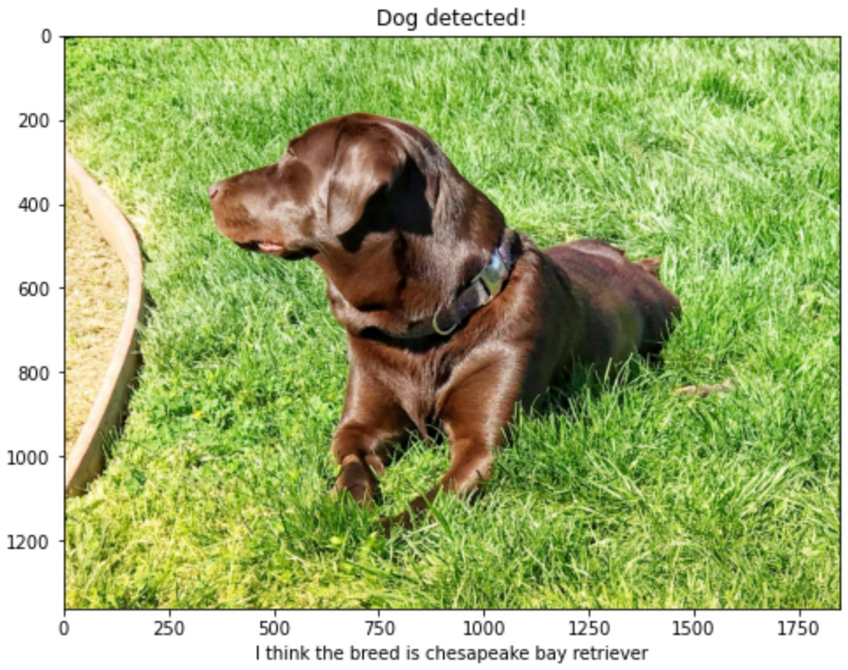
That hooman looks like a[n] english setter

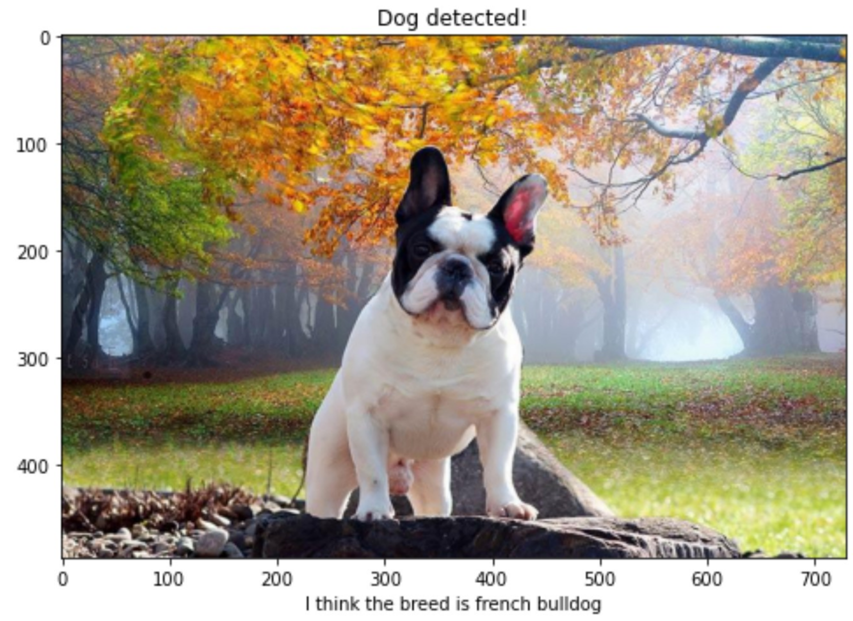
Dog detected!

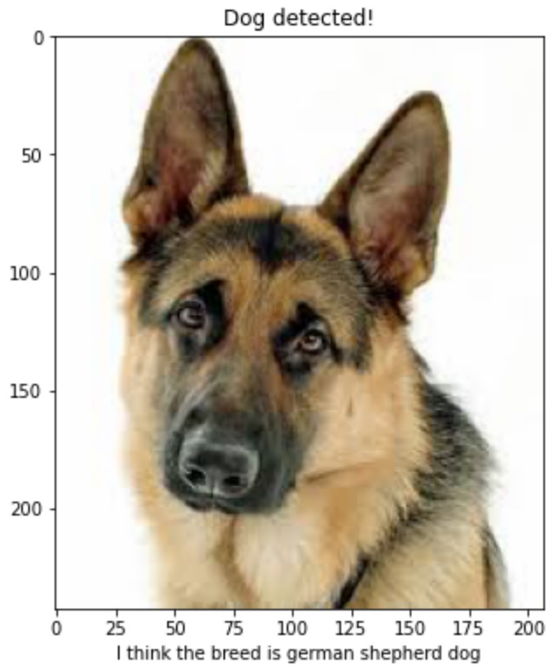


I think the breed is pomeranian

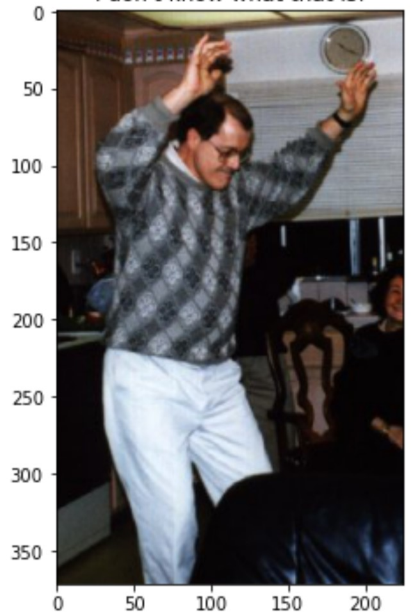

Hooman detected!
0
200
400
600
800
1000
0
200
400
600
800
1000
That hooman looks like a[n] french bulldog


Dog detected!
0
200
400
600
800
1000
1200
0
250
500
750
1000
1250
1500
1750
I think the breed is chesapeake bay retriever


Dog detected!
0
100
200
300
400
0
100
200
300
400
500
600
700
I think the breed is french bulldog

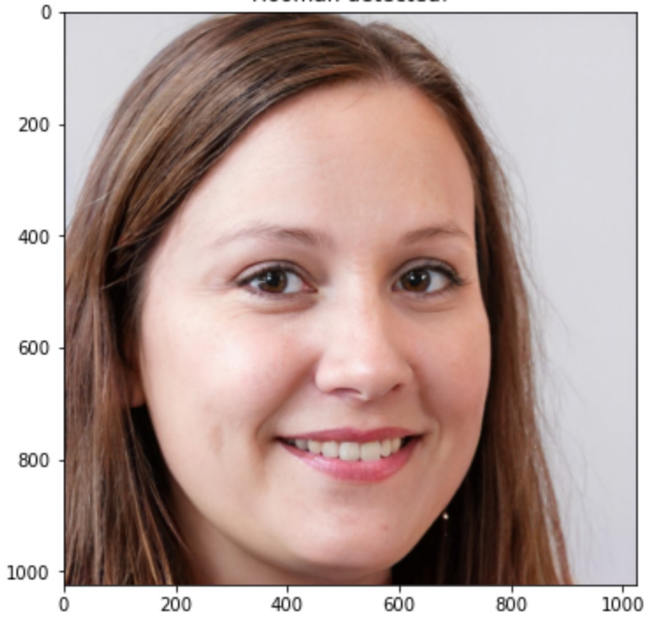

Dog detected!
0
50
100
150
200
0
25
50
75
100
125
150
175
200
I think the breed is german shepherd dog

I don't know what that is!



I don't know what that is.
What is that thing?
I don't know, but I want my picture taken with it. (Steve Martin)

Hooman detected!



That hooman looks like a[n] irish setter

I don't know what that is!

I don't know what that is.
What is that thing?
I don't know, but I want my picture taken with it. (Steve Martin)