dog_app

February 8, 2020

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

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

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

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

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

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

Step 0: Import Datasets

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

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

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

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

```
In [1]: 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

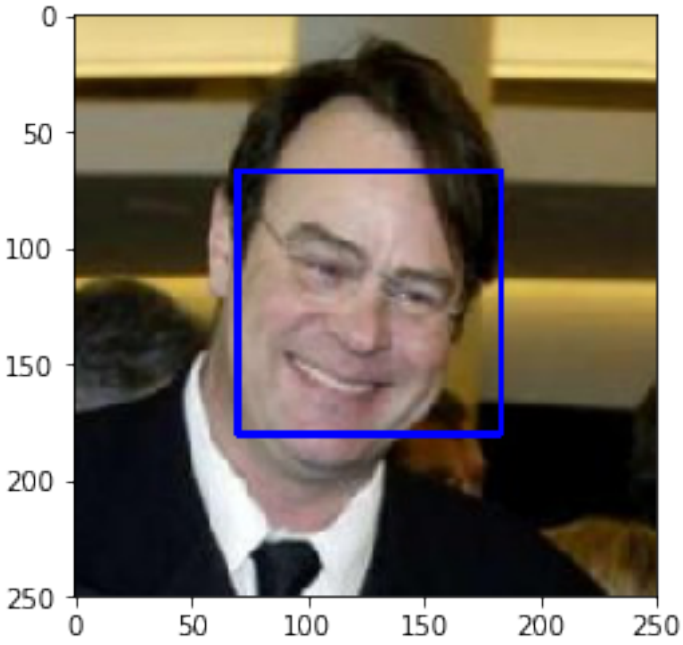
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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

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

        # load color (BGR) image
        img = cv2.imread(human_files[0])
        # convert BGR image to grayscale
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

        # find faces in image
        faces = face_cascade.detectMultiScale(gray)

        # print number of faces detected in the image
        print('Number of faces detected:', len(faces))

        # get bounding box for each detected face
        for (x,y,w,h) in faces:
            # add bounding box to color image
            cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)
```

```
        # convert BGR image to RGB for plotting
        cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

        # display the image, along with bounding box
        plt.imshow(cv_rgb)
        plt.show()
```

```
Number of faces detected: 1
```



Before using any of the face detectors, it is standard procedure to convert the images to grayscale. The `detectMultiScale` function executes the classifier stored in `face_cascade` and takes the grayscale image as a parameter.

In the above code, `faces` is a numpy array of detected faces, where each row corresponds to a detected face. Each detected face is a 1D array with four entries that specifies the bounding box of the detected face. The first two entries in the array (extracted in the above code as `x` and `y`) specify the horizontal and vertical positions of the top left corner of the bounding box. The last two entries in the array (extracted here as `w` and `h`) specify the width and height of the box.

1.1.1 Write a Human Face Detector

We can use this procedure to write a function that returns `True` if a human face is detected in an image and `False` otherwise. This function, aptly named `face_detector`, takes a string-valued file path to an image as input and appears in the code block below.

```
In [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
```

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

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

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

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

```
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.
        humanFiles_faceCount = 0
        dogFiles_faceCount = 0

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

        humanFiles_detectHumanFacePercent = humanFiles_faceCount / len(human_files_short) * 100
        print('Detected Human face from human_files: {}%'.format(humanFiles_detectHumanFacePerce

        dogFiles_detectHumanFacePercent = dogFiles_faceCount / len(dog_files_short) * 100
        print('Detected Human face from dog_files: {}%'.format(dogFiles_detectHumanFacePercent)

Detected Human face from human_files: 98.0%
Detected Human face from dog_files: 17.0%
```

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

```
In [6]: ### (Optional)
        ### TODO: Test performance of another face detection algorithm.
        ### Feel free to use as many code cells as needed.
```

Step 2: Detect Dogs

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

1.1.3 Obtain Pre-trained VGG-16 Model

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

```
In [5]: import torch
        import torchvision.models as models

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

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

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

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:06<00:00, 88175941.97it/s]
```

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

1.1.4 (IMPLEMENTATION) Making Predictions with a Pre-trained Model

In the next code cell, you will write a function that accepts a path to an image (such as `'dogImages/train/001.Affenpinscher/Affenpinscher_00001.jpg'`) as input and returns the index corresponding to the ImageNet class that is predicted by the pre-trained VGG-16 model. The output should always be an integer between 0 and 999, inclusive.

Before writing the function, make sure that you take the time to learn how to appropriately pre-process tensors for pre-trained models in the PyTorch documentation.

```
In [6]: from PIL import Image
        import torchvision.transforms as transforms

        # 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

    image = Image.open(img_path)
    transform = transforms.Compose([transforms.Resize((224,224)),
                                    transforms.ToTensor(),
                                    transforms.Normalize(mean=[0.485,0.456,0.406],
                                                         std=[0.229,0.224,0.225])])
    transformedImg = transform(image)
    newImg = transformedImg.unsqueeze(0)

    if use_cuda:
        newImg = newImg.cuda()

    scoreTensor = VGG16(newImg)
    return torch.argmax(scoreTensor).item() # predicted class index
```

1.1.5 (IMPLEMENTATION) Write a Dog Detector

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

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

```
In [7]: ### returns "True" if a dog is detected in the image stored at img_path
        def dog_detector(img_path):
            ## TODO: Complete the function.
            index = VGG16_predict(img_path)
            if index >= 151 and index <=268:
```

```
        return True
    else:
        return False
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

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

Answer:

```
In [12]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.
         humanFiles_dogCount = 0
         dogFiles_dogCount = 0

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

         humanFiles_detectDogPercent = humanFiles_dogCount / len(human_files_short) * 100
         print('Detected Dog from human_files: {}%'.format(humanFiles_detectDogPercent) )

         dogFiles_detectDogPercent = dogFiles_dogCount / len(dog_files_short) * 100
         print('Detected Dog from dog_files: {}%'.format(dogFiles_detectDogPercent) )

Detected Dog from human_files: 1.0%
Detected Dog from dog_files: 100.0%
```

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

```
In [11]: ### (Optional)
         ### TODO: Report the performance of another pre-trained network.
         ### Feel free to use as many code cells as needed.
```

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

Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of at least 10%. In Step 4 of this notebook, you will have the opportunity to use transfer learning to create a CNN that attains greatly improved accuracy.

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

Brittany	Welsh Springer Spaniel

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

Curly-Coated Retriever	American Water Spaniel

Likewise, recall that labradors come in yellow, chocolate, and black. Your vision-based algorithm will have to conquer this high intra-class variation to determine how to classify all of these different shades as the same breed.

Yellow Labrador	Chocolate Labrador

We also mention that random chance presents an exceptionally low bar: setting aside the fact that the classes are slightly imabalanced, a random guess will provide a correct answer roughly 1 in 133 times, which corresponds to an accuracy of less than 1%.

Remember that the practice is far ahead of the theory in deep learning. Experiment with many different architectures, and trust your intuition. And, of course, have fun!

1.1.7 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

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

```
In [8]: import os
        from torchvision import datasets
        import torchvision.transforms as transforms
        from torch.utils.data.sampler import SubsetRandomSampler

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

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

```
data_dir = '/data/dog_images/'
train_dir = os.path.join(data_dir, 'train/')
test_dir = os.path.join(data_dir, 'test/')
validation_dir = os.path.join(data_dir, 'valid/')

# convert data to a normalized torch.FloatTensor
data_transform = transforms.Compose([
    transforms.Resize(224),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)) #normalize RGB values to betu
])

train_data = datasets.ImageFolder(train_dir, transform=data_transform)
test_data = datasets.ImageFolder(test_dir, transform=data_transform)
validation_data = datasets.ImageFolder(validation_dir, transform=data_transform)

#ensure the number of training images are divisible by batch_size (20 images per batch)
num_train = len(train_data)
training_indices = list(range(num_train))
np.random.shuffle(training_indices)
split = int(np.floor(num_train / batch_size)) * batch_size
train_idx = training_indices[:split]

#ensure the number of validation images are divisible bybatch_size (20 images per batch)
num_validation = len(validation_data)
validation_indices = list(range(num_validation))
np.random.shuffle(validation_indices)
split = int(np.floor(num_validation / batch_size)) * batch_size
valid_idx = validation_indices[:split]

# define samplers for obtaining training and validation batches
train_sampler = SubsetRandomSampler(train_idx)
valid_sampler = SubsetRandomSampler(valid_idx)

loaders_scratch = {}

# prepare data loaders (combine dataset and sampler)
loaders_scratch['train'] = torch.utils.data.DataLoader(train_data, batch_size=batch_size
    sampler=train_sampler, num_workers=num_workers)

classes =  os.listdir(train_dir)
classes.sort()
loaders_scratch['train'].classes = classes

loaders_scratch['valid'] = torch.utils.data.DataLoader(train_data, batch_size=batch_size
    sampler=valid_sampler, num_workers=num_workers)
```

```
        loaders_scratch['test'] = torch.utils.data.DataLoader(test_data, batch_size=batch_size,
            num_workers=num_workers)

        # print out some data stats
        print('No. of training images: ', len( loaders_scratch['train'].sampler) )
        print('No. of validation images: ', len( loaders_scratch['valid'].sampler ) )
        print('No. of test images: ', len(loaders_scratch['test'].sampler))
```

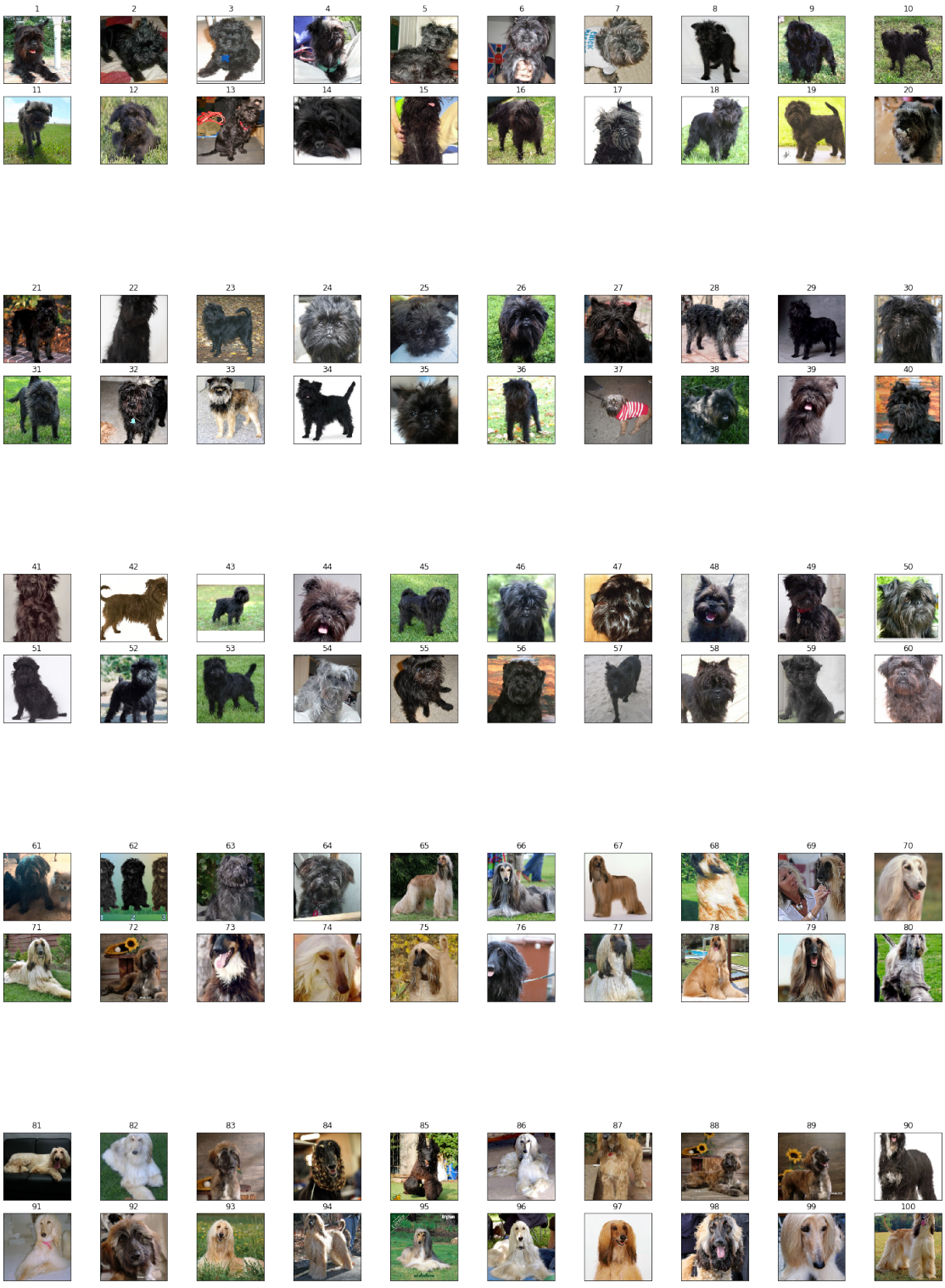
```
No. of training images:  6680
No. of validation images:  820
No. of test images:  836
```

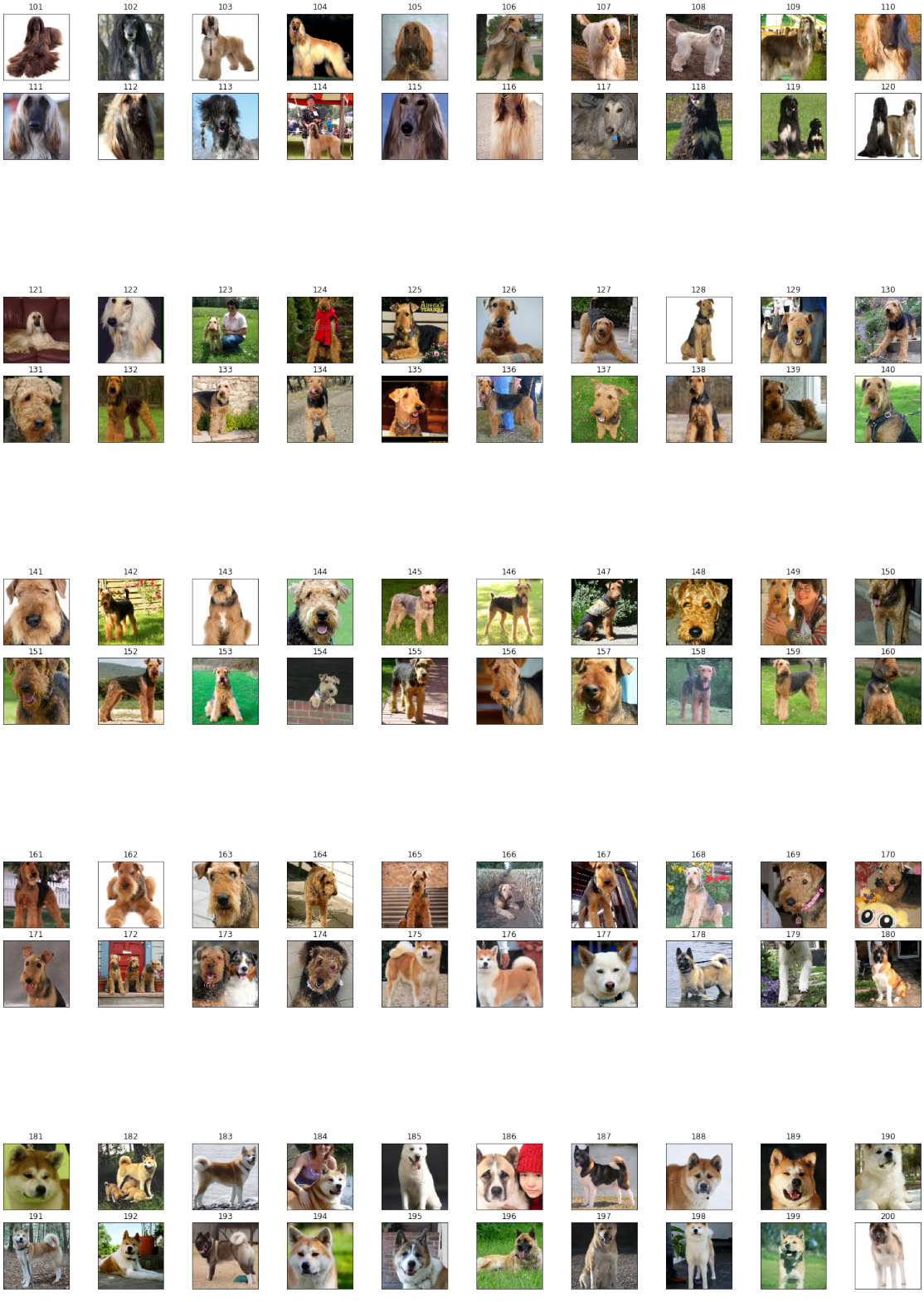
1.1.8 Preview the image transform by visualizing first 10 batches of training data after the transform

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

# helper function to un-normalize and display an image
def imshow(img):
    img = img / 2 + 0.5  # unnormalize
    plt.imshow(np.transpose(img, (1, 2, 0)))  # convert from Tensor image
```

In [9]:
```
# convert data to a normalized torch.FloatTensor
# same transform as above
data_transform_preview = transforms.Compose([
    transforms.Resize(224),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)) #normalize RGB values to betu
])
train_data_preview = datasets.ImageFolder(train_dir, transform=data_transform_preview)

train_loader_preview = torch.utils.data.DataLoader(train_data_preview, batch_size=batch_

# obtain first 10 batch of training images
dataiter = iter(train_loader_preview)
i = 0
index = 0
for i in range(10):
    images, labels = dataiter.next()
    images = images.numpy() # convert images to numpy for display

    # plot the images in the batch, along with the corresponding labels
    fig = plt.figure(figsize=(25, 4))
    # display 20 images (1 batch has 20 images)
```

```
for idx in np.arange(20):
    ax = fig.add_subplot(2, 20/2, idx+1, xticks=[], yticks=[])
    imshow(images[idx])
    index += 1
    ax.set_title(index) #index of the image as listed in the "train" directory
```




101 102 103 104 105 106 107 108 109 110
111 112 113 114 115 116 117 118 119 120
121 122 123 124 125 126 127 128 129 130
131 132 133 134 135 136 137 138 139 140
141 142 143 144 145 146 147 148 149 150
151 152 153 154 155 156 157 158 159 160
161 162 163 164 165 166 167 168 169 170
171 172 173 174 175 176 177 178 179 180
181 182 183 184 185 186 187 188 189 190
191 192 193 194 195 196 197 198 199 200

Question 3: Describe your chosen procedure for preprocessing the data. - How does your code resize the images (by cropping, stretching, etc)? What size did you pick for the input tensor, and why? - Did you decide to augment the dataset? If so, how (through translations, flips, rotations, etc)? If not, why not?

Answer: - Because the training images have different aspect ratio and width/height dimensions varied greatly (from 300 pixels to 1000 pixels), the code resize the image by using transforms.Resize and transforms.CenterCrop to obtain a 224x224 square image without losing too much information. This size is chosen because our CNN is going to take an input tensor of 224x224x3 in order to achieve the desire accuracy without having a complex architecture or very slow performance in training our model.

- I did not augment the data set (e.g. using transforms.RandomHorizontalFlip(), transforms.RandomRotation(10)) because most of the training data has the entire dog (or face of the dog) fill up the image. If using rotations, it will need to be cropped to get rid of the unncessary black edge after the rotation. However, this also means some characteristics of the dog could be cropped out, especially in some cases where the dog may occupied the entire image with little background. This applies to translations as well. As for not doing flips, it is not useful for the dog classifier because in the real test cases, the posture of the dog will be different if the dog has an unusual upside down posture.

1.1.9 (IMPLEMENTATION) Model Architecture

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

```
In [9]: import torch.nn as nn
        import torch.nn.functional as F

        # defining the CNN architecture
        class Net(nn.Module):
            def __init__(self):
                super(Net, self).__init__()
                # sees 224x224x3 image tensor
                self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
                # sees 112x112x16 image tensor
                self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
                # sees 56x56x32 image tensor
                self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
                # sees 28x28x64 image tensor
                self.conv4 = nn.Conv2d(64, 128, 3, padding=1)
                # sees 14x14x128 image tensor
                self.conv5 = nn.Conv2d(128, 256, 3, padding=1)
                # sees 7x7x256 image tensor
                self.fc1 = nn.Linear(256 * 7 * 7, 133)

                self.pool = nn.MaxPool2d(2, 2)
                self.dropout = nn.Dropout(0.2)
```

```
        self.bn2 = nn.BatchNorm2d(16)
        self.bn3 = nn.BatchNorm2d(32)
        self.bn4 = nn.BatchNorm2d(64)
        self.bn5 = nn.BatchNorm2d(128)
        self.bn6 = nn.BatchNorm2d(256)

    def forward(self, x):

        x = self.pool(F.relu(self.conv1(x)))
        x = self.bn2(x)

        x = self.pool(F.relu(self.conv2(x)))
        x = self.bn3(x)

        x = self.pool(F.relu(self.conv3(x)))
        x = self.bn4(x)

        x = self.pool(F.relu(self.conv4(x)))
        x = self.bn5(x)

        x = self.pool(F.relu(self.conv5(x)))
        x = self.bn6(x)

        x = x.view(-1, 256 * 7 * 7)
        x = self.dropout(x)
        x = self.fc1(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()
```

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

Answer: Our CNN architecture consists of 5 convolutional layers, with each convolutional layer decreases the width/height dimensions of the input tensor while extracts certain features that will eventually help to reach the end goal of identifying the breed of the dog.

A 2x2 max pooling follows by a 2d batch normalization is applied at the output of each convolutional layer.

The first convolutional layer is going to take a 224x224x3 image tensor with 3 input channels, 16 output channels (filters), a kernel of 3, and a stride of 1. Since we apply 2x2 max pooling, the dimensions of the output from the first convolutional layer is reduce by a factor of 2, as a 112x112x16 output tensor which is feed into the second convolutional layer.

The second convolutional layer takes in a 112x112x16 tensor, and double the output (filters) to 32 using the same kernel size of 3 and a stride of 1. After we apply 2x2 max pooling, the output is a scale down to a 56x56x32 image tensor which is feed into the third convolutional layer.

The same is repeated for the third, fouth and fifth convolutional layers, with each layer scales down the width and height dimensions by a factor of 2 while double the number of output channels (filters), until it reaches the fully connected layer.

At the end of the fifth convoluational, the output is a 7x7x256 tensor. A dropout of 0.2 is applied to prevent overfitting before the tensor is feed into the fully connected layer by flattening the image using x.view(-1, 256 * 7 * 7). The number of outputs of the fully connected layer is 133, which corresponds to the number of dog breeds (classes) that the model try to identify.

Please note the following:

- A Relu activation function is used for each convolutional layer.
- Max pooling is applied at the output of each convoluation layer, since it is better at noticing the most important details about edges and other features in an image.
- Finally, a 2d batch normalization is used for each convoluation layer to ensure the inputs to the next layer are normalized considering that the squared image (initial input) is a transformation of the original image which has different width and height dimensions (different aspect ratio).
- All convolutional layers used a stride of 1, and a padding of 1 is used for the missing pixels when the kernel moves to the edge of the image, to ensure the convolution layer has the same width and height as the input tensor.

1.1.10 (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 [10]:
```
import torch.optim as optim

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

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

1.1.11 (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 [12]:
```
# the following import is required for training to be robust to truncated images
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
```

```
"""returns trained model"""
# initialize tracker for minimum validation loss
valid_loss_min = np.Inf

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

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

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

        # clear the gradients of all optimized variables
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)

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

    ######################
    # 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
        # calculate the batch loss
```

```
                    loss = criterion(output, target)
                    # update validation loss
                    valid_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss)

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

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

In [13]: # train the model
         model_scratch = train(20, loaders_scratch, model_scratch, optimizer_scratch,
                               criterion_scratch, use_cuda, 'model_scratch.pt')

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

```
Epoch: 1         Training Loss: 4.426201         Validation Loss: 4.465521
Validation loss decreased (inf --> 4.465521).  Saving model ...
Epoch: 2         Training Loss: 3.347016         Validation Loss: 3.415445
Validation loss decreased (4.465521 --> 3.415445).  Saving model ...
Epoch: 3         Training Loss: 2.390483         Validation Loss: 2.499279
Validation loss decreased (3.415445 --> 2.499279).  Saving model ...
Epoch: 4         Training Loss: 1.524407         Validation Loss: 1.644492
Validation loss decreased (2.499279 --> 1.644492).  Saving model ...
Epoch: 5         Training Loss: 0.857257         Validation Loss: 1.029499
Validation loss decreased (1.644492 --> 1.029499).  Saving model ...
Epoch: 6         Training Loss: 0.425895         Validation Loss: 0.606818
Validation loss decreased (1.029499 --> 0.606818).  Saving model ...
Epoch: 7         Training Loss: 0.216882         Validation Loss: 0.433346
Validation loss decreased (0.606818 --> 0.433346).  Saving model ...
Epoch: 8         Training Loss: 0.128684         Validation Loss: 0.347986
Validation loss decreased (0.433346 --> 0.347986).  Saving model ...
Epoch: 9         Training Loss: 0.087963         Validation Loss: 0.338929
Validation loss decreased (0.347986 --> 0.338929).  Saving model ...
Epoch: 10         Training Loss: 0.070049         Validation Loss: 0.329903
```

```
Validation loss decreased (0.338929 --> 0.329903).  Saving model ...
Epoch: 11         Training Loss: 0.054367         Validation Loss: 0.306536
Validation loss decreased (0.329903 --> 0.306536).  Saving model ...
Epoch: 12         Training Loss: 0.048694         Validation Loss: 0.301004
Validation loss decreased (0.306536 --> 0.301004).  Saving model ...
Epoch: 13         Training Loss: 0.045188         Validation Loss: 0.314401
Epoch: 14         Training Loss: 0.038422         Validation Loss: 0.295525
Validation loss decreased (0.301004 --> 0.295525).  Saving model ...
Epoch: 15         Training Loss: 0.034464         Validation Loss: 0.340246
Epoch: 16         Training Loss: 0.033438         Validation Loss: 0.310372
Epoch: 17         Training Loss: 0.030552         Validation Loss: 0.312406
Epoch: 18         Training Loss: 0.029807         Validation Loss: 0.299971
Epoch: 19         Training Loss: 0.027727         Validation Loss: 0.311058
Epoch: 20         Training Loss: 0.024907         Validation Loss: 0.308574
```

1.1.12 (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 [29]:
```
def test(loaders, model, criterion, use_cuda):

    # monitor test loss and accuracy
    test_loss = 0.
    correct = 0.
    total = 0.

    model.eval()
    for batch_idx, (data, target) in enumerate(loaders['test']):
        # move to GPU
        if use_cuda:
            data, target = data.cuda(), target.cuda()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the loss
        loss = criterion(output, target)
        # update average test loss
        test_loss = test_loss + ((1 / (batch_idx + 1)) * (loss.data - test_loss))
        # convert output probabilities to predicted class
        pred = output.data.max(1, keepdim=True)[1]
        # compare predictions to true label
        correct += np.sum(np.squeeze(pred.eq(target.data.view_as(pred))).cpu().numpy())
        total += data.size(0)

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

    print('\nTest Accuracy: %2d%% (%2d/%2d)' % (
```

```
                100. * correct / total, correct, total))
```

```
In [15]: # call test function
         test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.863496


Test Accuracy: 19% (161/836)
```

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

You will now use transfer learning to create a CNN that can identify dog breed from images. Your CNN must attain at least 60% accuracy on the test set.

1.1.13 (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 [16]: ## TODO: Specify data loaders
         ## VGG-16 takes 224x224 images as input
         ## so we use the same data loaders from the previous step
```

1.1.14 (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 [11]: import torchvision.models as models
         import torch.nn as nn

         ## TODO: Specify model architecture
         model_transfer = models.vgg16(pretrained=True)

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

         # Freeze training for all "features" layers
         for param in model_transfer.features.parameters():
             param.requires_grad = False

         n_inputs = model_transfer.classifier[6].in_features
```

```
        # add last linear layer (n_inputs -> 133 dog breed classes)
        # new layers automatically have requires_grad = True
        last_layer = nn.Linear(n_inputs, 133)

        model_transfer.classifier[6] = last_layer

        print(model_transfer)

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

```
4096
1000
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
```

```
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace)
    (2): Dropout(p=0.5)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace)
    (5): Dropout(p=0.5)
    (6): Linear(in_features=4096, out_features=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: Used the VGG-16 Architecture because it is simple and has great performance, coming in second in the ImageNet competition. The pre-trained weights will remain frozen during the training of our network. The last connected layer will be modified to have an output of 133 classes, and only weights in the last layer will be changed during training: - Loaded VGG16 Model with pre-trained weights. - Freeze the weights in the feature layers such that all the pre-trained convoluational weights will not change during training by iterate the parameters of the featured layers: For every pre-trained weights, set requires_grad = false such that the gradient calculations will not change these weights. - Replace the last linear layer with a new linear that output scores for the 133 classes (no. of dog breeds). This new layer will have requires_grad = True as default.

1.1.15 (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 [12]: criterion_transfer = nn.CrossEntropyLoss()

         optimizer_transfer = optim.SGD(model_transfer.classifier.parameters(), lr=0.001)
```

1.1.16 (IMPLEMENTATION) Train and Validate the Model

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

```
In [21]: # train the model
         model_transfer = train(30, loaders_scratch, model_transfer, optimizer_transfer, criteri

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

```
Epoch: 1         Training Loss: 3.944960         Validation Loss: 3.981703
Validation loss decreased (inf --> 3.981703).  Saving model ...
Epoch: 2         Training Loss: 1.940960         Validation Loss: 2.089596
Validation loss decreased (3.981703 --> 2.089596).  Saving model ...
Epoch: 3         Training Loss: 1.165261         Validation Loss: 1.372830
Validation loss decreased (2.089596 --> 1.372830).  Saving model ...
```

```
Epoch: 4         Training Loss: 0.907735         Validation Loss: 1.156684
Validation loss decreased (1.372830 --> 1.156684).  Saving model ...
Epoch: 5         Training Loss: 0.753355         Validation Loss: 0.943218
Validation loss decreased (1.156684 --> 0.943218).  Saving model ...
Epoch: 6         Training Loss: 0.660275         Validation Loss: 0.958510
Epoch: 7         Training Loss: 0.597880         Validation Loss: 0.887118
Validation loss decreased (0.943218 --> 0.887118).  Saving model ...
Epoch: 8         Training Loss: 0.537252         Validation Loss: 0.849710
Validation loss decreased (0.887118 --> 0.849710).  Saving model ...
Epoch: 9         Training Loss: 0.492626         Validation Loss: 0.816302
Validation loss decreased (0.849710 --> 0.816302).  Saving model ...
Epoch: 10         Training Loss: 0.465609         Validation Loss: 0.780330
Validation loss decreased (0.816302 --> 0.780330).  Saving model ...
Epoch: 11         Training Loss: 0.420214         Validation Loss: 0.708141
Validation loss decreased (0.780330 --> 0.708141).  Saving model ...
Epoch: 12         Training Loss: 0.403709         Validation Loss: 0.734486
Epoch: 13         Training Loss: 0.360942         Validation Loss: 0.719565
Epoch: 14         Training Loss: 0.357130         Validation Loss: 0.732009
Epoch: 15         Training Loss: 0.334472         Validation Loss: 0.661307
Validation loss decreased (0.708141 --> 0.661307).  Saving model ...
Epoch: 16         Training Loss: 0.312368         Validation Loss: 0.727609
Epoch: 17         Training Loss: 0.302468         Validation Loss: 0.613018
Validation loss decreased (0.661307 --> 0.613018).  Saving model ...
Epoch: 18         Training Loss: 0.276242         Validation Loss: 0.619628
Epoch: 19         Training Loss: 0.266855         Validation Loss: 0.643085
Epoch: 20         Training Loss: 0.261518         Validation Loss: 0.608168
Validation loss decreased (0.613018 --> 0.608168).  Saving model ...
Epoch: 21         Training Loss: 0.255537         Validation Loss: 0.607064
Validation loss decreased (0.608168 --> 0.607064).  Saving model ...
Epoch: 22         Training Loss: 0.236610         Validation Loss: 0.674507
Epoch: 23         Training Loss: 0.222655         Validation Loss: 0.614735
Epoch: 24         Training Loss: 0.211498         Validation Loss: 0.587244
Validation loss decreased (0.607064 --> 0.587244).  Saving model ...
Epoch: 25         Training Loss: 0.201728         Validation Loss: 0.584033
Validation loss decreased (0.587244 --> 0.584033).  Saving model ...
Epoch: 26         Training Loss: 0.195507         Validation Loss: 0.589573
Epoch: 27         Training Loss: 0.185443         Validation Loss: 0.626466
Epoch: 28         Training Loss: 0.178967         Validation Loss: 0.610650
Epoch: 29         Training Loss: 0.172956         Validation Loss: 0.554667
Validation loss decreased (0.584033 --> 0.554667).  Saving model ...
Epoch: 30         Training Loss: 0.164959         Validation Loss: 0.594550
```

1.1.17 (IMPLEMENTATION) Test the Model

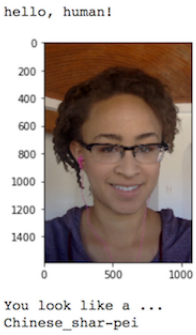
Try out your model on the test dataset of dog images. Use the code cell below to calculate and print the test loss and accuracy. Ensure that your test accuracy is greater than 60%.

```
In [30]: test(loaders_scratch, model_transfer, criterion_transfer, use_cuda)

Test Loss: 0.468358


Test Accuracy: 86% (722/836)
```

1.1.18 (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 [13]: from torch.autograd import Variable

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

         def image_loader(image_name):
             """load image, returns cuda tensor"""
             image = Image.open(image_name)
             image = data_transform(image).float()
             image = Variable(image, requires_grad=True)
             image = image.unsqueeze(0)
             return image.cuda()

In [14]: ### 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_scratch['train'].classes]

         def predict_breed_transfer(img_path):
             # load the image and return the predicted breed
             image = image_loader(img_path)
             output = model_transfer(image)
             pred = output.data.max(1, keepdim=True)[1]
             classIndex = pred.item()
             if classIndex >= 0 and classIndex <= len(class_names):
                 return class_names[classIndex]
             else:
                 return None

In [14]: predict_breed_transfer('/data/dog_images/test/021.Belgian_sheepdog/Belgian_sheepdog_014

Out[14]: 'Belgian sheepdog'
```



Sample Human Output

Step 5: Write your Algorithm

Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then, - if a **dog** is detected in the image, return the predicted breed. - if a **human** is detected in the image, return the resembling dog breed. - if **neither** is detected in the image, provide output that indicates an error.

You are welcome to write your own functions for detecting humans and dogs in images, but feel free to use the `face_detector` and `dog_detector` functions developed above. You are **required** to use your CNN from Step 4 to predict dog breed.

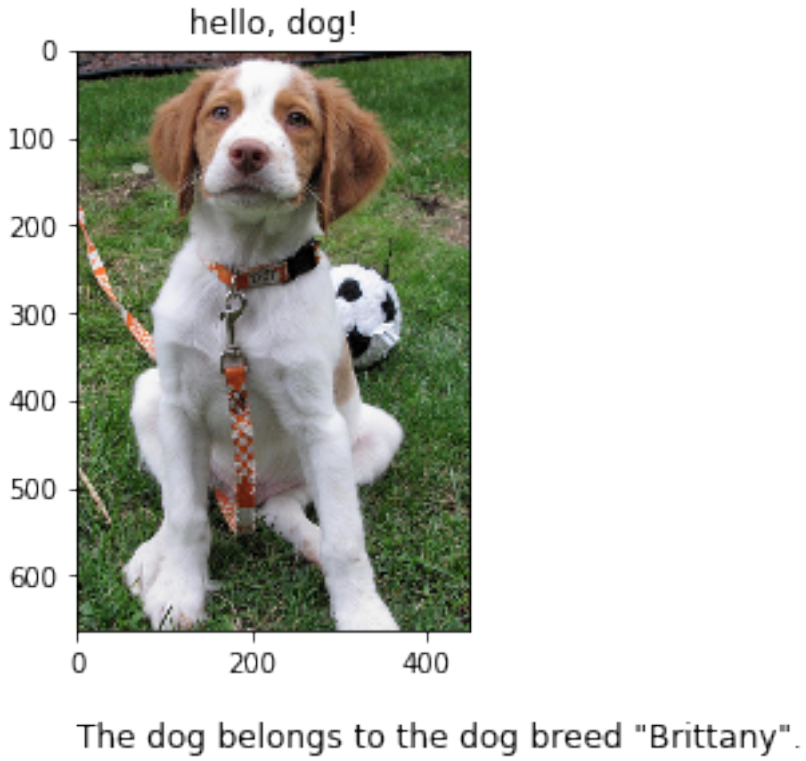
Some sample output for our algorithm is provided below, but feel free to design your own user experience!

1.1.19 (IMPLEMENTATION) Write your Algorithm

In [15]:
```
### 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
    case = 'neither'
    message = ''
    image = Image.open(img_path)

    if ( face_detector( img_path ) ):
        case = 'human'
        dogBreed = predict_breed_transfer(img_path);
        message = 'You look like a ... {}.'.format( dogBreed )

    if ( dog_detector( img_path ) ):
        case = 'dog'
        dogBreed = predict_breed_transfer(img_path);
        message = 'The dog belongs to the dog breed "{}".'.format( dogBreed )

    plt.figure()
    plt.imshow(image)  # convert from Tensor image
```

```
            if ( case == 'neither'):
                plt.title("Error: Cannot detect either human or dog!")
            else:

                plt.title( 'hello, {}!'.format(case) )
                width, height = image.size
                plt.text(0, height*1.2, message, fontsize=12)
```

```
In [70]: run_app('images/Brittany_02625.jpg')
```



Step 6: Test Your Algorithm

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

1.1.20 (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:

The output is better than expected, because my transfer learning CNN is able to identify the correct dog breed such as the French Bulldog with the cigar, the Akita in a sleeping pose (although the built-in VGG16 first identifies it as a human!) and the Golden retriever standing beside a human with a leash that has the same image focus. Also, it is able to identify the "Collie" with a much longer hair style with a sitting posture. If the image is a face, it will correctly identifies the image as a human or if it is a cat, it will return an error to indicate the image is neither a human or a dog.

The following are some possible ways to improve the algorithm:

(1): During transfer learning, the training data should consists of images with dogs on the right or left side of the image, or consider augmenting the data using random translation in the image transform when builting up the training data loaders. This will improve our CNN which may not be able to identify the Akita standing on the left side of the image.

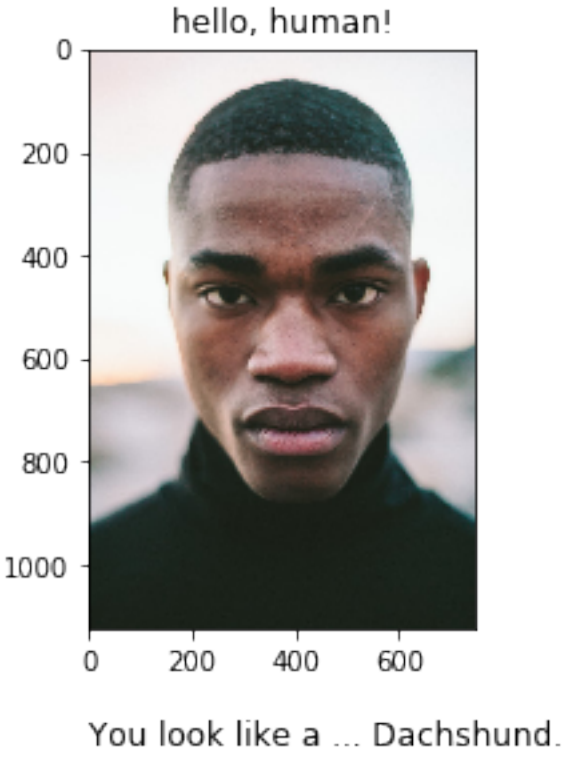
(2): Expand the training data for each dog breed to include their common behaviors, such as dog rinsing themselves. This will broaden what kind of dog images that the CNN can accept to correctly identify the dog breed. In our case, the Bulldog rinsing themselves is incorrectly identify as a Bullmastiff.

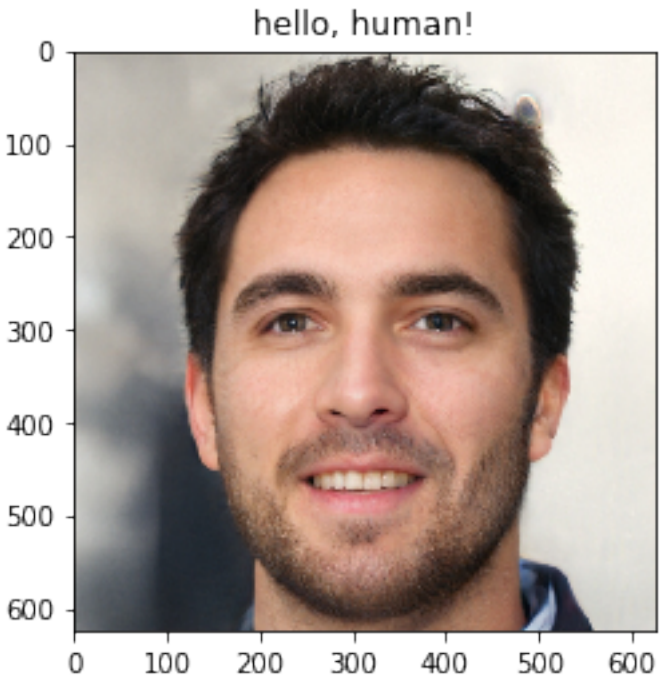
(3): Our CNN requires the minmum width/height dimensions of the image is 224x224. If the image is smaller, our algorithm will result in an error during data_transform (in image_loader function). To solve this problem, our algorithm should consider upsample the image if it falls below the minimum dimensions, and use transpose convolution before feeding the image to our CNN.

(4): Increases the number of Epochs (currently set at 30). As seen in step 4 during Train and Validate the model, the validation lose is able to decrease, even up to the 30th Epoch. In this case, increasing the number of Epochs may further decrease the validation lose which will increase the accuracy of our model.

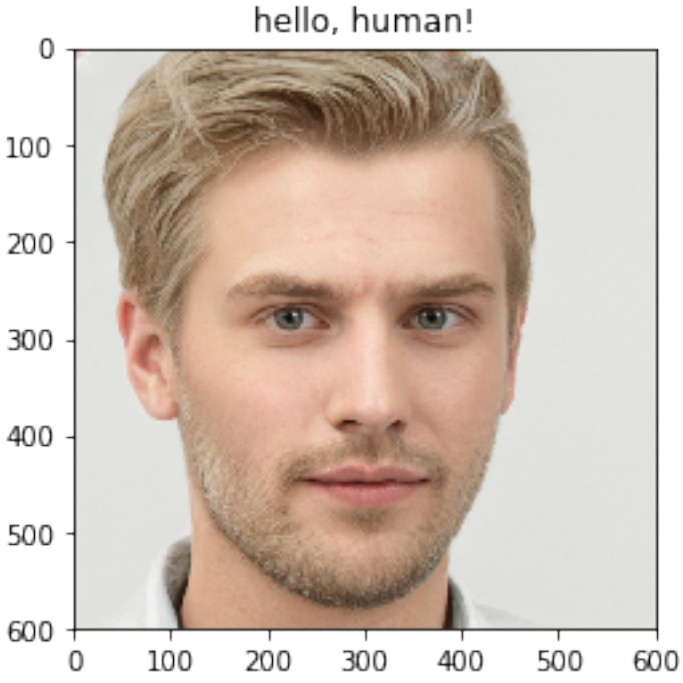
```
In [26]: ## TODO: Execute your algorithm from Step 6 on
         ## at least 6 images on your computer.
         ## Feel free to use as many code cells as needed.
         import os
         import numpy as np

         def get_image_file_path(folder_path):
             files = os.listdir(folder_path)
             try:
                 # remove .ipynb_checkpoints file which is auto created in the workspace folder
                 files.remove('.ipynb_checkpoints')
             except ValueError:
                 pass

             for i in range(len(files)):
                 files[i] = os.path.join(folder_path, files[i])

             return files

         dog_files = get_image_file_path('step6_test_images/dog_files')
         human_files = get_image_file_path('step6_test_images/human_files')
```

```
cat_files =get_image_file_path('step6_test_images/cat_files')

for file in np.hstack(( human_files, dog_files, cat_files )):
    run_app(file)
```

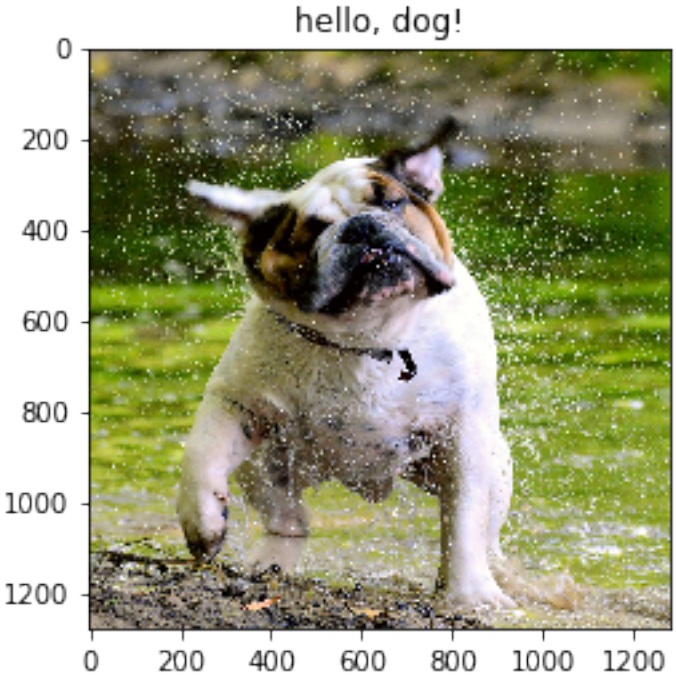




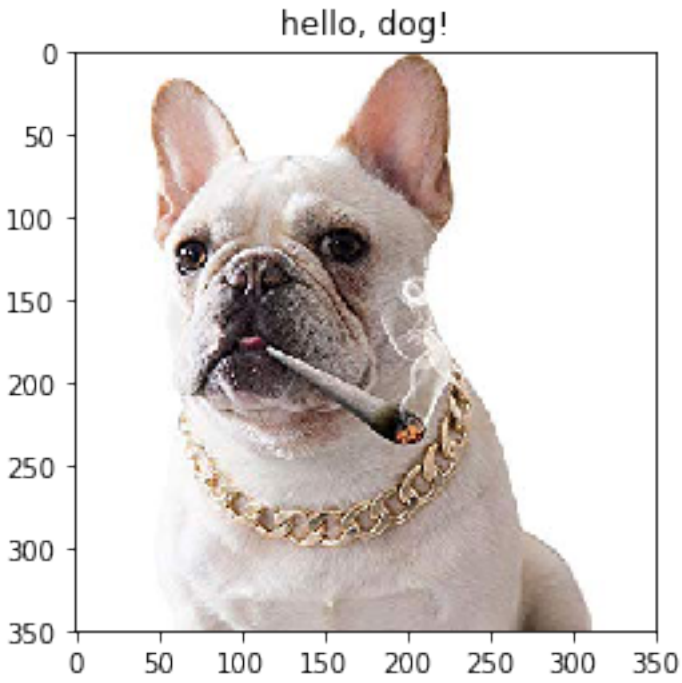
You look like a ... Nova scotia duck tolling retriever.



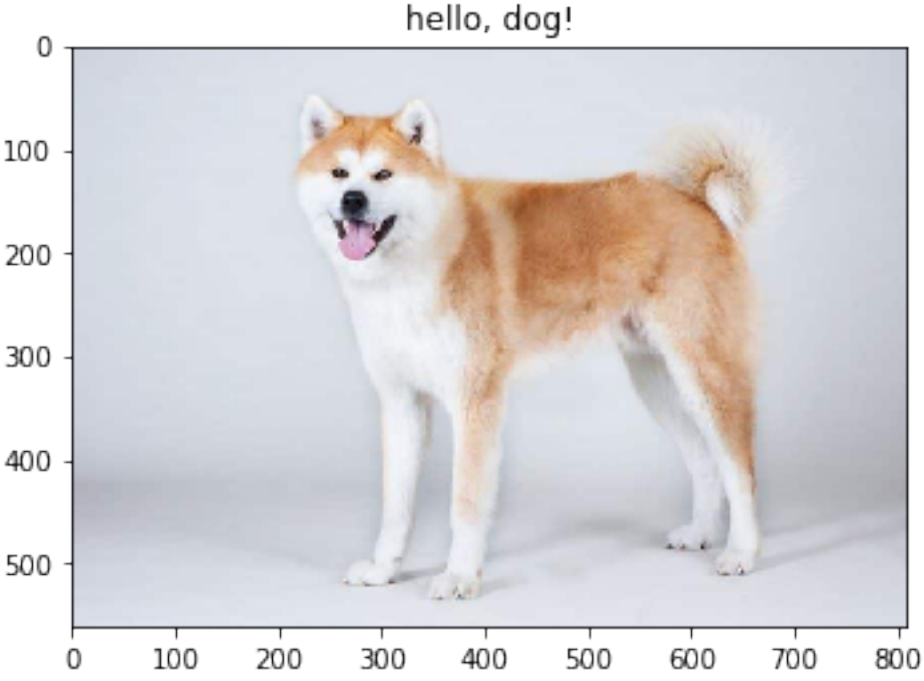
You look like a ... Poodle.



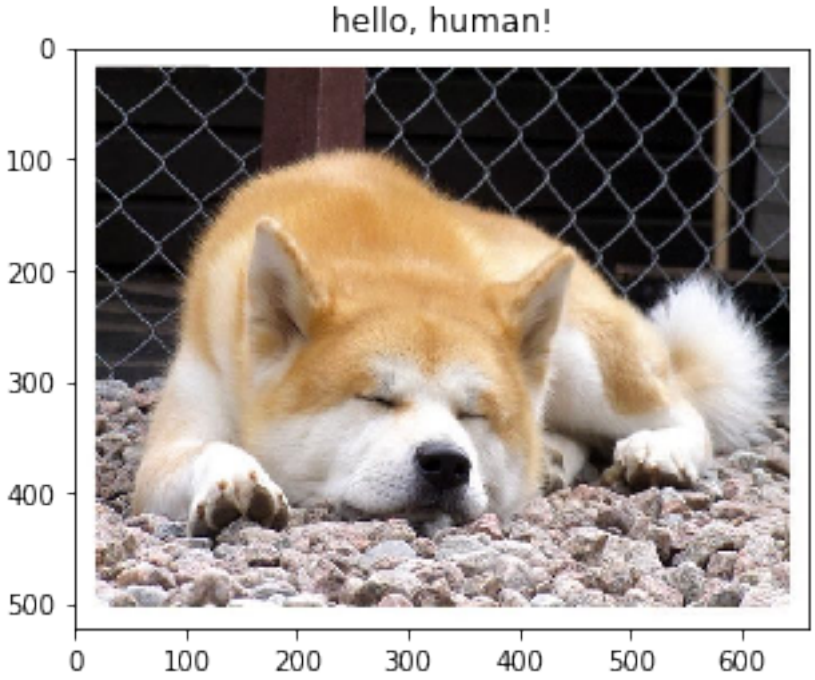
The dog belongs to the dog breed "Bullmastiff".



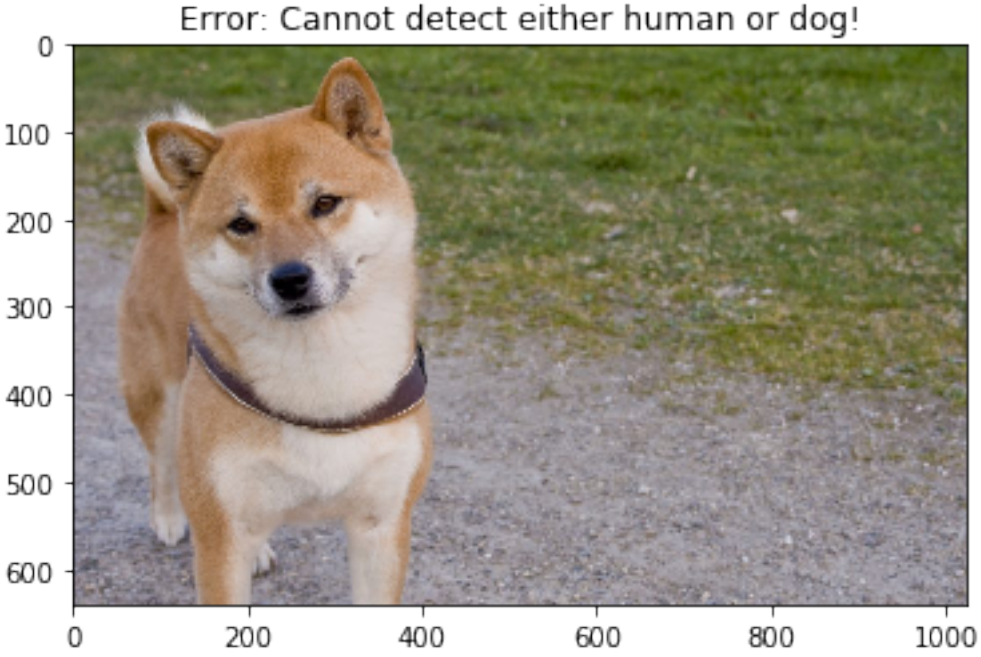
The dog belongs to the dog breed "French bulldog".

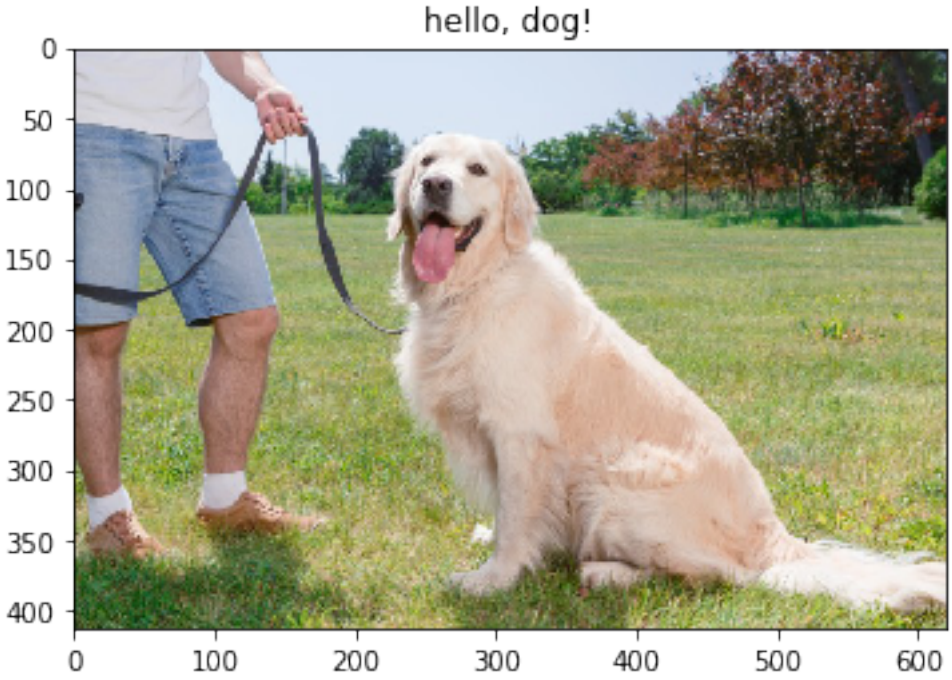


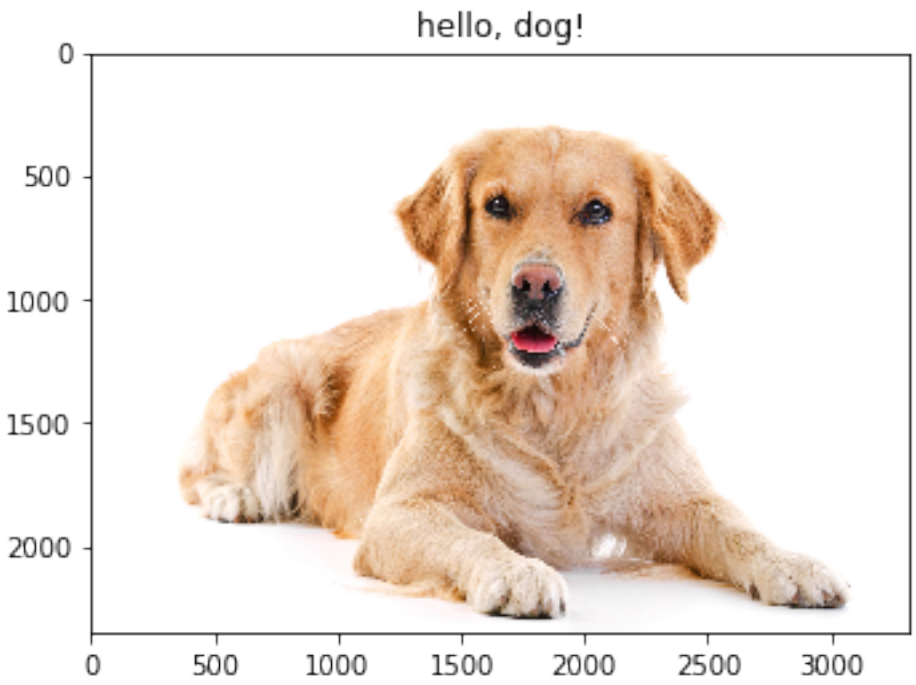
The dog belongs to the dog breed "Akita".



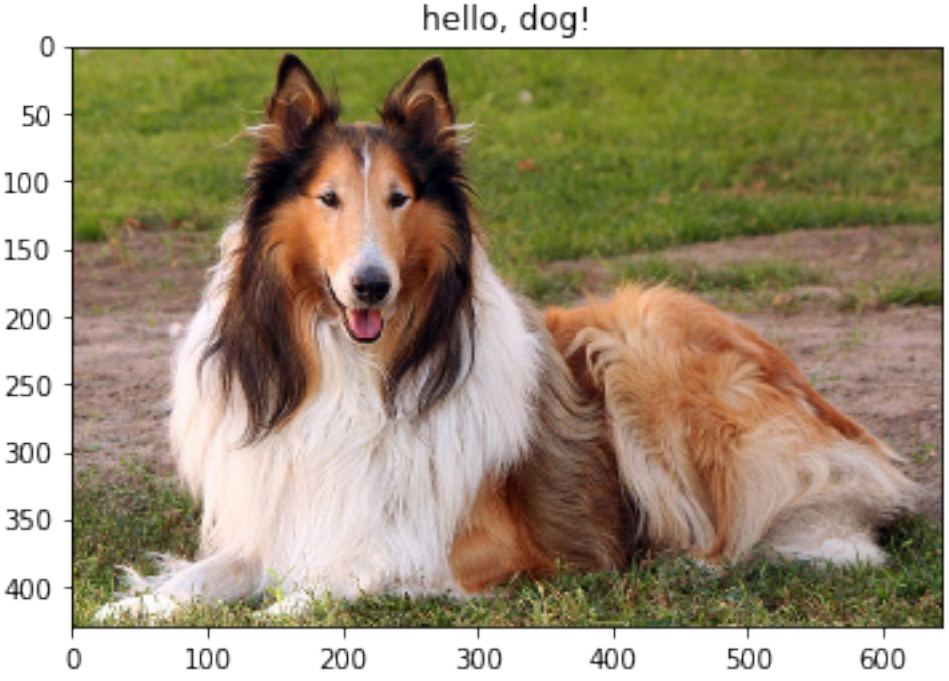
You look like a ... Akita.



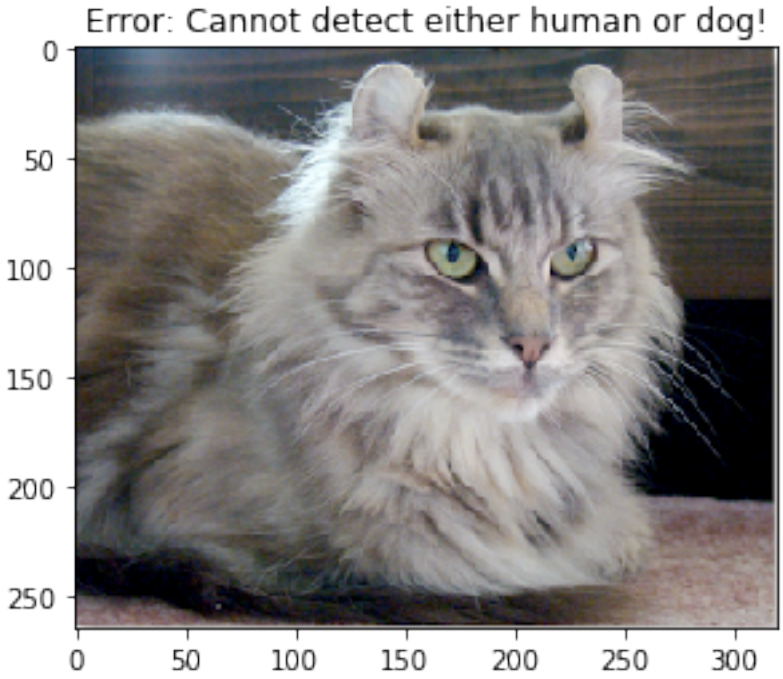

hello, dog!
0
50
100
150
200
250
300
350
400
0
100
200
300
400
500
600
The dog belongs to the dog breed "Golden retriever".

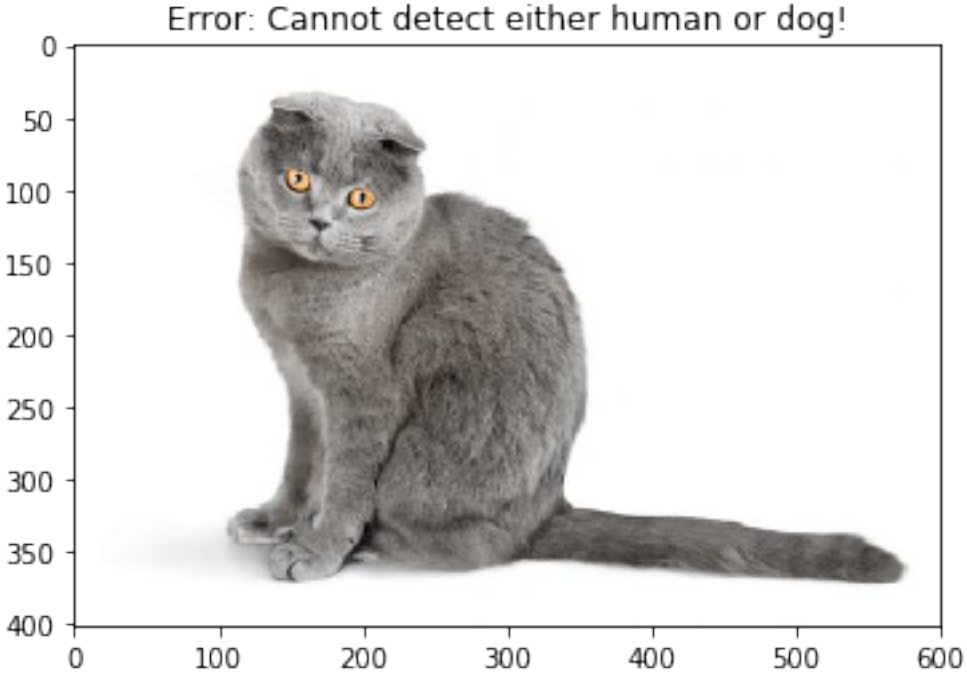


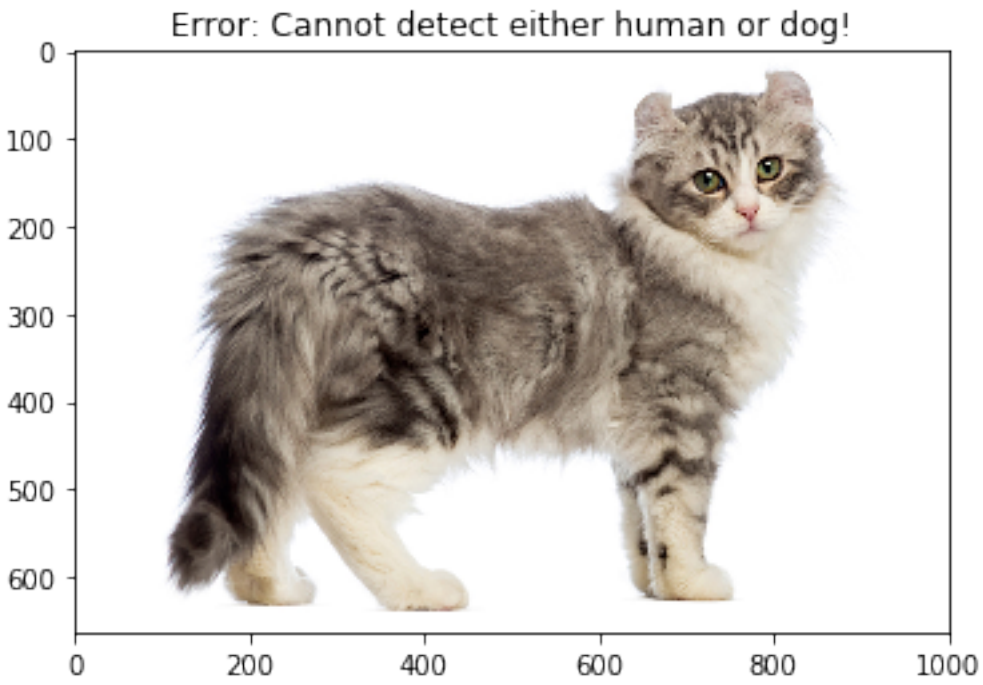
The dog belongs to the dog breed "Golden retriever".



The dog belongs to the dog breed "Collie".


Error: Cannot detect either human or dog!
0
50
100
150
200
250
0
50
100
150
200
250
300


Error: Cannot detect either human or dog!
0
50
100
150
200
250
300
350
400
0
100
200
300
400
500
600


Error: Cannot detect either human or dog!
0
100
200
300
400
500
600
0
200
400
600
800
1000