dog_app

August 18, 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:

Note: if you are using the Udacity workspace, you *DO NOT* need to re-download these - they can be found in the `/data` folder as noted in the cell below.

- Download the dog dataset. Unzip the folder and place it in this project's home directory, at the location /dog_images.
- Download the human dataset. Unzip the folder and place it in the home directory, at location /lfw.

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

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

In [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

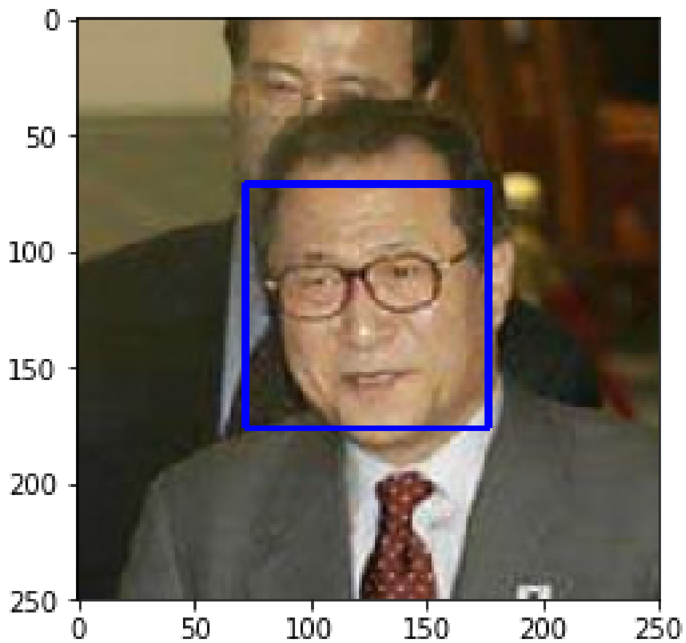
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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[10])
# 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_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
            faces = face_cascade.detectMultiScale(gray_img)
            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: Human face detected in 98.0% images of the first 100 human_files. 17.0% images of the first 100 dog_files detected as human face.

```
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.
        human_count = 0
        dog_count = 0

        for file in human_files_short:
            if face_detector(file):
                human_count += 1

        for file in dog_files_short:
            if face_detector(file):
                dog_count += 1

        print('Human face detected in %.1f%% images of the first 100 human_files.' % human_count
        print('%.1f%% images of the first 100 dog_files detected as human face.' % dog_count)
```

```
Human face detected in 98.0% images of the first 100 human_files.
17.0% images of the first 100 dog_files detected as human face.
```

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 anotherface detection algorithm.
### Feel free to use as many code cells as needed.
!pip install cmake
!pip install dlib
!pip install face-recognition
```

```
Collecting cmake
  Downloading https://files.pythonhosted.org/packages/98/db/b77b44af9a217be1352e9f6e79ade771a7f9
    100% || 18.2MB 2.5MB/s eta 0:00:01
Installing collected packages: cmake
Successfully installed cmake-3.18.0
Collecting dlib
  Downloading https://files.pythonhosted.org/packages/a4/7b/2f7f29f460629a8143b2deea1911e2fb1d9d
    100% || 3.2MB 8.7MB/s eta 0:00:01
Building wheels for collected packages: dlib
  Running setup.py bdist_wheel for dlib ... done
  Stored in directory: /root/.cache/pip/wheels/e3/fd/51/22af51f198c3d1adde947c1189c6dfc70923d70c
Successfully built dlib
Installing collected packages: dlib
Successfully installed dlib-19.21.0
Collecting face-recognition
  Downloading https://files.pythonhosted.org/packages/1e/95/f6c9330f54ab07bfa032bf3715c12455a381
Requirement already satisfied: numpy in /opt/conda/lib/python3.6/site-packages (from face-recogn
Requirement already satisfied: dlib>=19.7 in /opt/conda/lib/python3.6/site-packages (from face-r
Requirement already satisfied: Click>=6.0 in /opt/conda/lib/python3.6/site-packages (from face-r
Requirement already satisfied: Pillow in /opt/conda/lib/python3.6/site-packages (from face-recog
Collecting face-recognition-models>=0.3.0 (from face-recognition)
  Downloading https://files.pythonhosted.org/packages/cf/3b/4fd8c534f6c0d1b80ce0973d01331525538O
    100% || 100.2MB 439kB/s eta 0:00:01 5% |                              | 5.1MB 38.9MB/s eta 0
Building wheels for collected packages: face-recognition-models
  Running setup.py bdist_wheel for face-recognition-models ... done
  Stored in directory: /root/.cache/pip/wheels/d2/99/18/59c6c8f01e39810415c0e63f5bede7d83dfb0ffc
Successfully built face-recognition-models
Installing collected packages: face-recognition-models, face-recognition
Successfully installed face-recognition-1.3.0 face-recognition-models-0.3.0
```

In [7]:
```
import face_recognition
```

```
In [8]: def face_locations(img_path):
            image = face_recognition.load_image_file(img_path)
            face_locations = face_recognition.face_locations(image)
            return len(face_locations)
```

```
In [9]: human_files_short = human_files[:100]
        dog_files_short = dog_files[:100]

        human_count = 0
        dog_count = 0

        for file in human_files_short:
            human_count += face_locations(file)

        for file in dog_files_short:
            dog_count += face_locations(file)

        print('{0} Human faces detected in images in the first 100 human_files.'.format(human_co
        print('{0} Dog faces detected as human faces in the first 100 dog_files.'.format(dog_cou
```

```
109 Human faces detected in images in the first 100 human_files.
10 Dog faces detected as human faces in the first 100 dog_files.
```

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()
```

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.

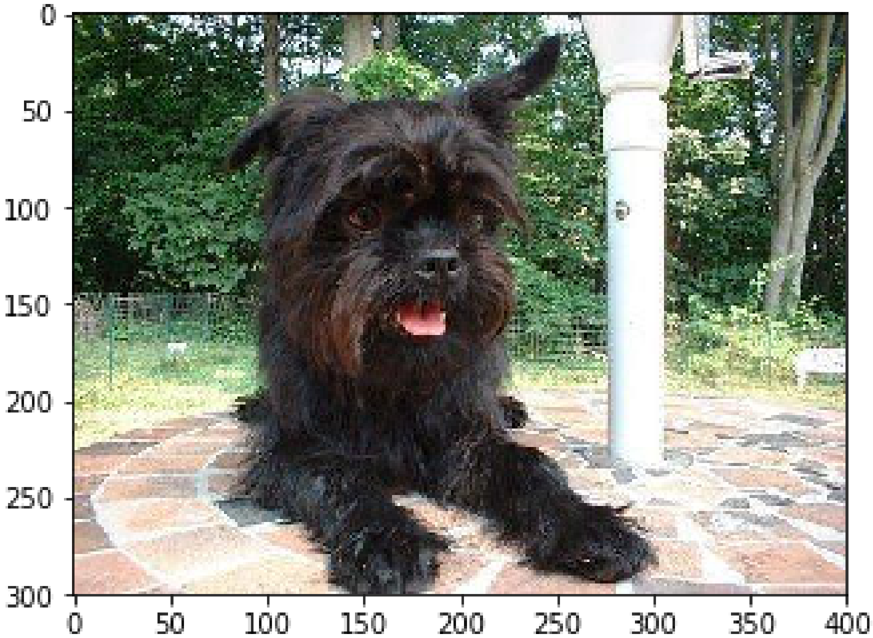
1.1.4 (IMPLEMENTATION) Making Predictions with a Pre-trained Model

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

def image_to_tensor(img_path):
    '''
    As per Pytorch documentations: All pre-trained models expect input images normalized
    i.e. mini-batches of 3-channel RGB images
    of shape (3 x H x W), where H and W are expected to be at least 224.
    The images have to be loaded in to a range of [0, 1] and
    then normalized using mean = [0.485, 0.456, 0.406] and std = [0.229, 0.224, 0.225].
    You can use the following transform to normalize:
    '''
    img = Image.open(img_path).convert('RGB')
    transformations = transforms.Compose([transforms.Resize(size=224),
                                          transforms.CenterCrop((224,224)),
                                         transforms.ToTensor(),
                                         transforms.Normalize(mean=[0.485, 0.456, 0.406]
                                                              std=[0.229, 0.224, 0.225])
    image_tensor = transformations(img)[:3,:,:].unsqueeze(0)
    return image_tensor


# helper function for un-normalizing an image  - from STYLE TRANSFER exercise
# and converting it from a Tensor image to a NumPy image for display
def im_convert(tensor):
    """ Display a tensor as an image. """

    image = tensor.to("cpu").clone().detach()
    image = image.numpy().squeeze()
    image = image.transpose(1,2,0)
    image = image * np.array((0.229, 0.224, 0.225)) + np.array((0.485, 0.456, 0.406))
    image = image.clip(0, 1)

    return image
```

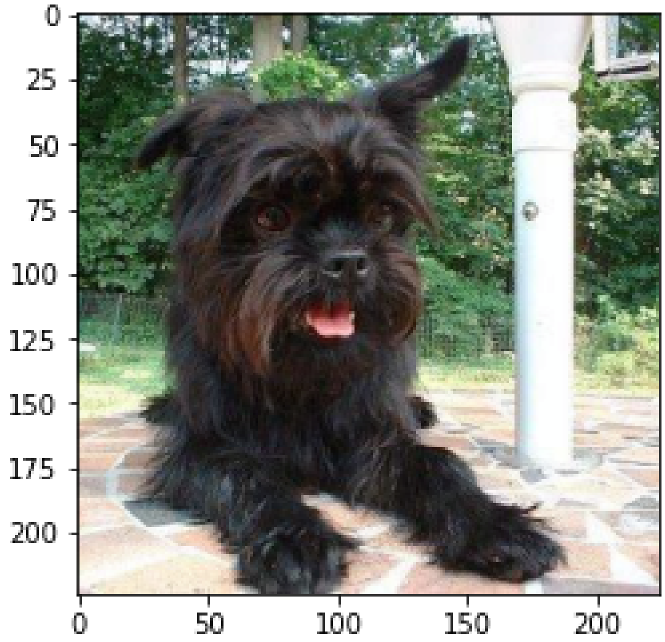
In [7]:
```
dog_image = Image.open('/data/dog_images/train/001.Affenpinscher/Affenpinscher_00001.jpg
plt.imshow(dog_image)
plt.show()
```




```
In [8]: test_tensor = image_to_tensor('/data/dog_images/train/001.Affenpinscher/Affenpinscher_00
        # print(test_tensor)
        print(test_tensor.shape)
        plt.imshow(im_convert(test_tensor))
```

```
torch.Size([1, 3, 224, 224])
```

```
Out[8]: <matplotlib.image.AxesImage at 0x7fc3941af160>
```




```
In [9]: 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
            image_tensor = image_to_tensor(img_path)

            # move model inputs to cuda, if GPU available
            if use_cuda:
                image_tensor = image_tensor.cuda()

            # get sample outputs
            output = VGG16(image_tensor)
            # convert output probabilities to predicted class
            _, preds_tensor = torch.max(output, 1)
            pred = np.squeeze(preds_tensor.numpy()) if not use_cuda else np.squeeze(preds_tensor
```

```
    return int(pred)
```

In [10]:
```
import ast
import requests

LABELS_MAP_URL = "https://gist.githubusercontent.com/yrevar/942d3a0ac09ec9e5eb3a/raw/c2

def get_human_readable_label_for_class_id(class_id):
    labels = ast.literal_eval(requests.get(LABELS_MAP_URL).text)
    print(f"Label:{labels[class_id]}")
    return labels[class_id]


test_prediction = VGG16_predict('/data/dog_images/train/001.Affenpinscher/Affenpinscher
pred_class = int(test_prediction)

print(f"Predicted class id: {pred_class}")
class_description = get_human_readable_label_for_class_id(pred_class)
print(f"Predicted class for image is *** {class_description.upper()} ***")
```

```
Predicted class id: 252
Label:affenpinscher, monkey pinscher, monkey dog
Predicted class for image is *** AFFENPINSCHER, MONKEY PINSCHER, MONKEY DOG ***
```

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 [11]:
```
### returns "True" if a dog is detected in the image stored at img_path

def dog_detector(img_path):
    ## TODO: Complete the function.
    prediction = VGG16_predict(img_path)
    return ((prediction >= 151) & (prediction <=268))
```

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: 1.0% of the images in human_files_short have detected a dog. 100.0% of the images in dog_files_short have detected a dog.

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

         for n in human_files_short:
             if dog_detector(n):
                 human_face_count += 1

         for n in dog_files_short:
             if dog_detector(n):
                 dog_face_count += 1

         print('detected dog in human_files: {0}%'.format((human_face_count/len(human_files_shor
         print('detected dog in dog_files: {0}%'.format((dog_face_count/len(dog_files_short)*100
```

```
detected dog in human_files: 0.0%
detected dog in 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`.

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

Curly-Coated Retriever	American Water Spaniel

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

Yellow Labrador	Chocolate Labrador

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

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

1.1.7 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

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

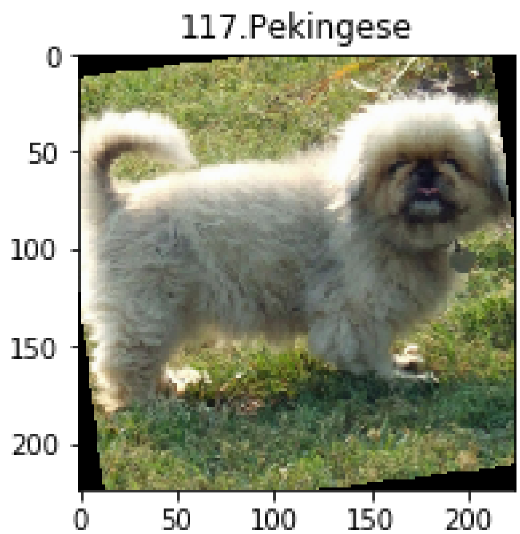
In [13]:
```
import os
import random
import requests
import time
import ast
import numpy as np
from glob import glob
import cv2
from tqdm import tqdm
from PIL import Image, ImageFile

import torch
import torchvision
from torchvision import datasets
import torchvision.transforms as transforms
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision.models as models
```

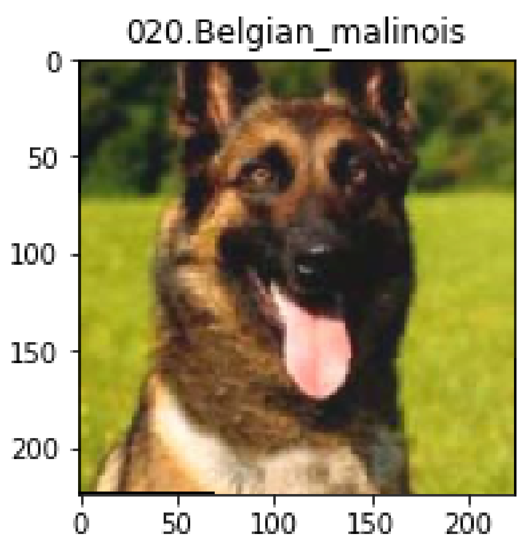
```
import matplotlib.pyplot as plt
%matplotlib inline

ImageFile.LOAD_TRUNCATED_IMAGES = True

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


### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes
 #how many samples per batch to load
batch_size = 16

# number of subprocesses to use for data loading
num_workers = 2

# convert data to a normalized torch.FloatTensor
transform = transforms.Compose([transforms.Resize(size=224),
                                transforms.CenterCrop((224,224)),
                                transforms.RandomHorizontalFlip(),
                                transforms.RandomRotation(10),
                                transforms.ToTensor(),
                                transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0

# define training, test and validation data directories
data_dir = '/data/dog_images/'

image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x), transform)
                  for x in ['train', 'valid', 'test']}
loaders_scratch = {
    x: torch.utils.data.DataLoader(image_datasets[x], shuffle=True, batch_size=batch_si
    for x in ['train', 'valid', 'test']}

class_names = image_datasets['train'].classes
nb_classes = len(class_names)

print("Number of classes:", nb_classes)
print("\nClass names: \n\n", class_names)
```

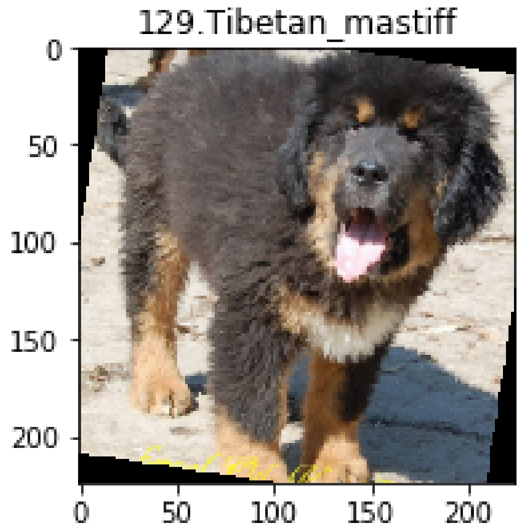
```
Number of classes: 133

Class names:

 ['001.Affenpinscher', '002.Afghan_hound', '003.Airedale_terrier', '004.Akita', '005.Alaskan_mal
```

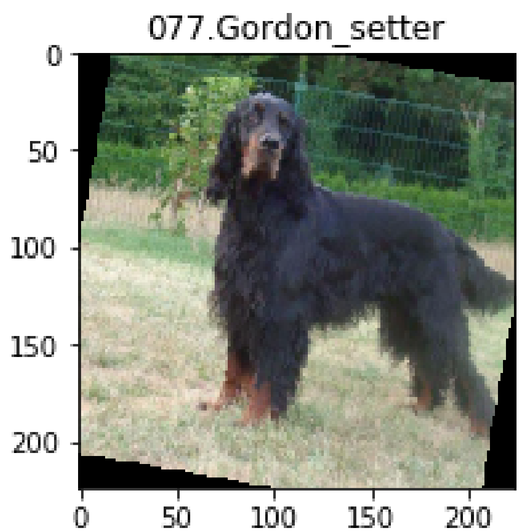
```
In [14]: inputs, classes = next(iter(loaders_scratch['train']))
```

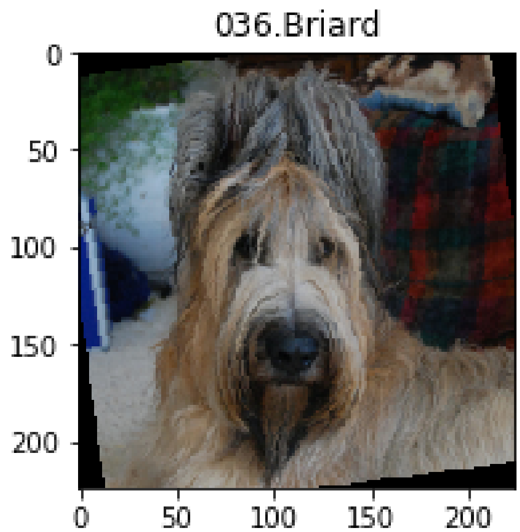
```
for image, label in zip(inputs, classes):
    image = image.to("cpu").clone().detach()
    image = image.numpy().squeeze()
    image = image.transpose(1,2,0)
    image = image * np.array((0.229, 0.224, 0.225)) + np.array((0.485, 0.456, 0.406))
    image = image.clip(0, 1)

    fig = plt.figure(figsize=(12,3))
    plt.imshow(image)
    plt.title(class_names[label])
```

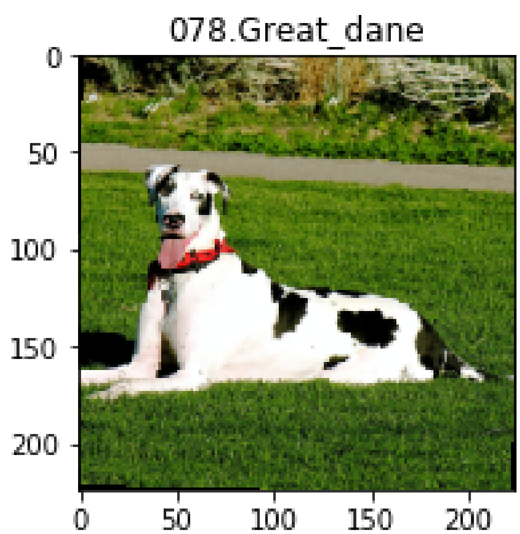


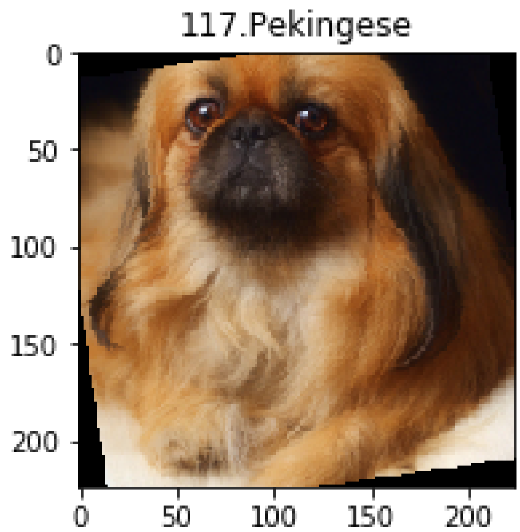


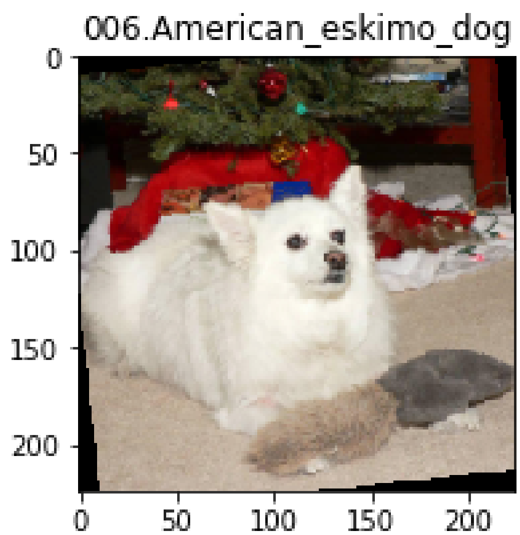

129.Tibetan_mastiff
0
50
100
150
200
0
50
100
150
200

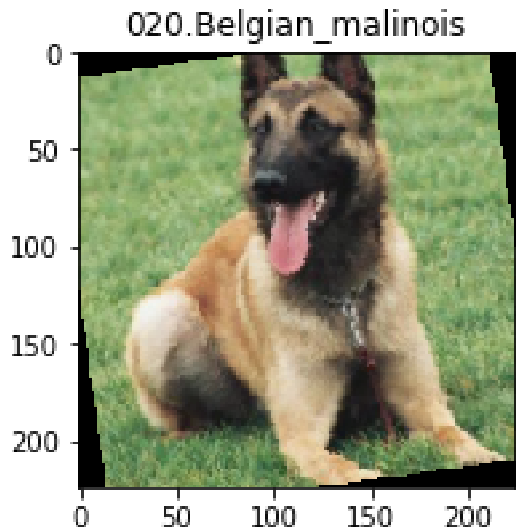

077.Gordon_setter
0
50
100
150
200
0
50
100
150
200

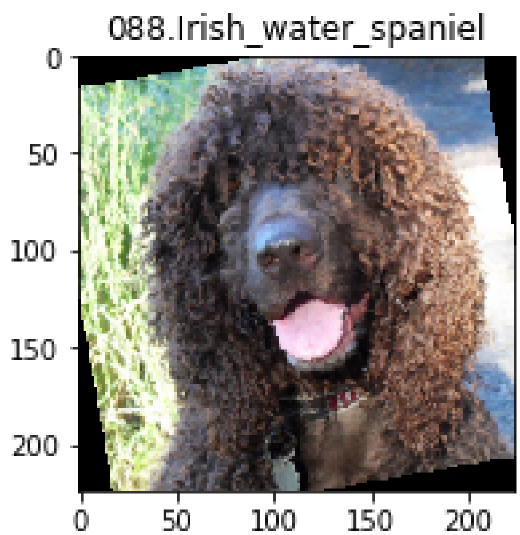

036.Briard
0
50
100
150
200
0
50
100
150
200

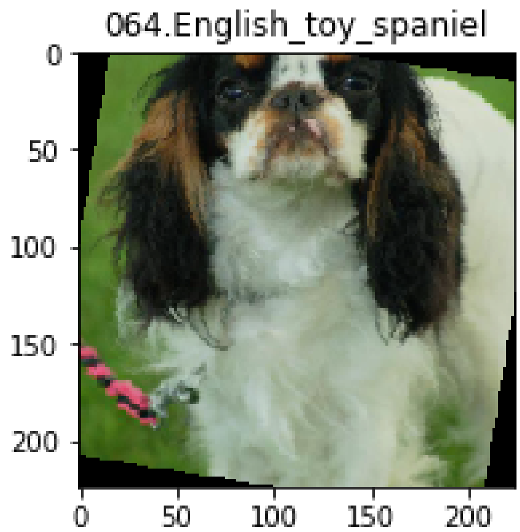

078.Great_dane
0
50
100
150
200
0
50
100
150
200

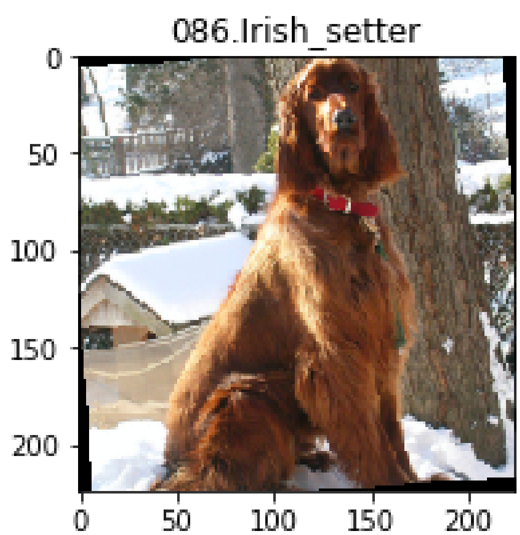

117.Pekingese
0
50
100
150
200
0
50
100
150
200

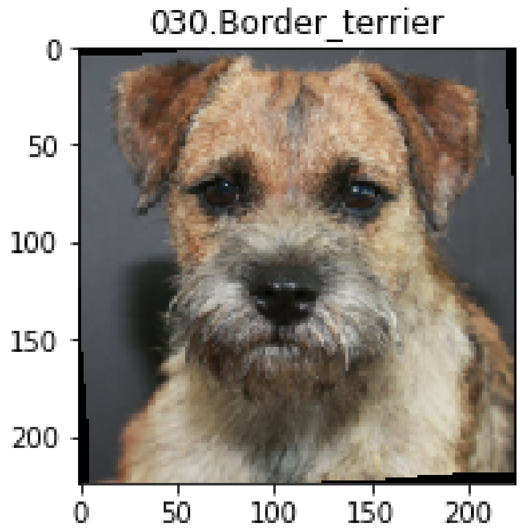

006.American_eskimo_dog
0
50
100
150
200
0
50
100
150
200

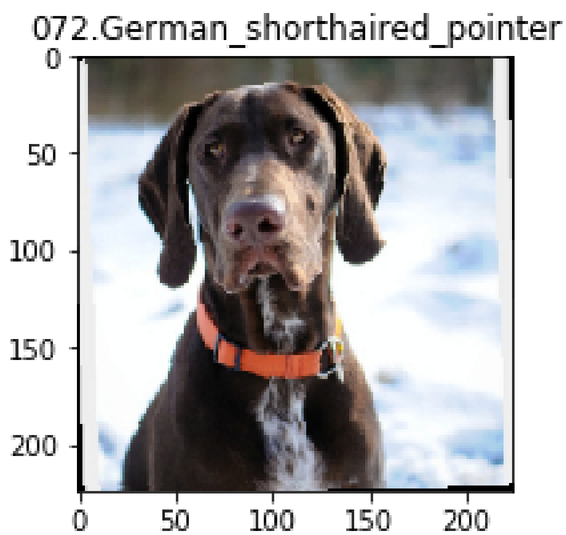

020.Belgian_malinois
0
50
100
150
200
0
50
100
150
200

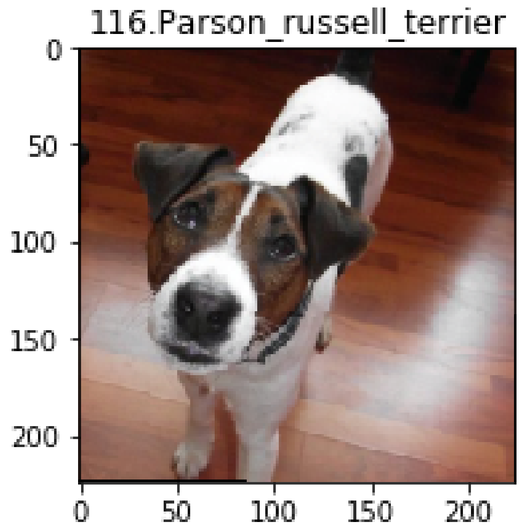

088.Irish_water_spaniel
0
50
100
150
200
0
50
100
150
200

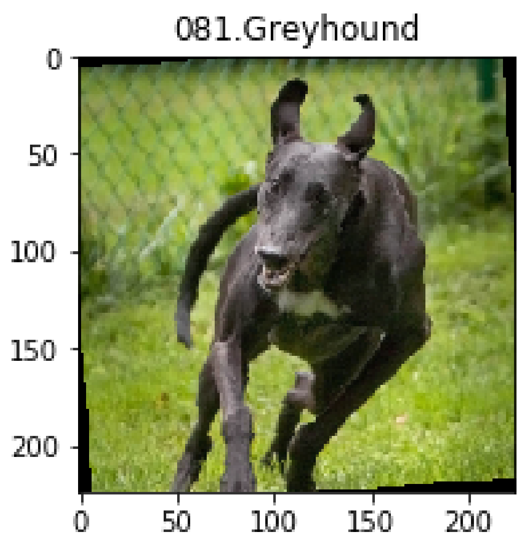

064.English_toy_spaniel
0
50
100
150
200
0
50
100
150
200


086.Irish_setter
0
50
100
150
200
0
50
100
150
200


030.Border_terrier
0
50
100
150
200
0
50
100
150
200


072.German_shorthaired_pointer
0
50
100
150
200
0
50
100
150
200





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

Answer: According to the Pytorch documentations, all pre-trained models expect input images normalized in the same way, i.e. mini-batches of 3-channel RGB images of shape (3 x H x W), where H and W are expected to be at least 224. The images have to be loaded in to a range of [0, 1] and then normalized using mean = [0.485, 0.456, 0.406] and std = [0.229, 0.224, 0.225]. The dataset is augmented using transforms such as, randomRotation which rotates the image at a given degrees enabling the date to become much more diverse and real-time and randomHorizontalFlip to randomly flip the images horizontally.

1.1.8 (IMPLEMENTATION) Model Architecture

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

In [15]:
```
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, padding=1)

        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)

        self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)

        self.fc1 = nn.Linear(64 * 28 * 28, 500)
        self.fc2 = nn.Linear(500, 133)
        self.dropout = nn.Dropout(0.25)
        self.batch_norm = nn.BatchNorm1d(num_features=500)

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

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

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

        x = x.view(x.size(0), -1)

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

        x = self.fc2(x)

        return x

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

# instantiate the CNN
model_scratch = Net()
```

```
        # 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: According the transformations applied to data, the input shape to model is (224, 224, 3). Since, the total number of classes is 133, the output layer should result in 133 classes.

The model architecture contains a stack of convolutional layers initially, followed by a Max pooling layer. The max pooling layer is used to reduce the size of the input, hence retain only active feature pixels from the previous layer. Linear and Dropout layers are used in the architecture to avoid overfitting and finally produce a 133 dimension output.

The forward pass of the neural network would give sizes with 16 filters at every layer: (starting from first layer) [16, 3, 224, 224] [16, 26, 112, 112] [16, 32, 56, 56] [16, 64, 28, 28] [16, 50176] [16, 500] [16, 133]

1.1.9 (IMPLEMENTATION) Specify Loss Function and Optimizer

Use the next code cell to specify a loss function and optimizer. Save the chosen loss function as `criterion_scratch`, and the optimizer as `optimizer_scratch` below.

```
In [16]: import torch.optim as optim

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

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

1.1.10 (IMPLEMENTATION) Train and Validate the Model

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

```
In [17]: def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
             """returns trained model"""
             # initialize tracker for minimum validation loss
             valid_loss_min = np.Inf

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

                 ###################
                 # train the model #
                 ###################
                 model.train()
```

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

        ######################
        # validate the model #
        ######################
        model.eval()
        for batch_idx, (data, target) in enumerate(loaders['valid']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## update the average validation loss
            output = model(data)
            loss = criterion(output, target)
            valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_loss)

        train_loss = train_loss/len(loaders['train'].dataset)
        valid_loss = valid_loss/len(loaders['valid'].dataset)
        # print training/validation statistics
        print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
            epoch,
            train_loss,
            valid_loss
            ))

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

    # return trained model
    return model
```

```
In [23]: # train the model
```

```
        model_scratch = train(25, loaders_scratch, model_scratch, optimizer_scratch,
                              criterion_scratch, use_cuda, 'model_scratch.pt')
```

```
Epoch: 1         Training Loss: 0.000708        Validation Loss: 0.005768
Validation loss decreased (inf --> 0.005768).  Saving model ...
Epoch: 2         Training Loss: 0.000674        Validation Loss: 0.005750
Validation loss decreased (0.005768 --> 0.005750).  Saving model ...
Epoch: 3         Training Loss: 0.000656        Validation Loss: 0.005867
Epoch: 4         Training Loss: 0.000641        Validation Loss: 0.005875
Epoch: 5         Training Loss: 0.000628        Validation Loss: 0.005742
Validation loss decreased (0.005750 --> 0.005742).  Saving model ...
Epoch: 6         Training Loss: 0.000614        Validation Loss: 0.005503
Validation loss decreased (0.005742 --> 0.005503).  Saving model ...
Epoch: 7         Training Loss: 0.000599        Validation Loss: 0.005582
Epoch: 8         Training Loss: 0.000585        Validation Loss: 0.005320
Validation loss decreased (0.005503 --> 0.005320).  Saving model ...
Epoch: 9         Training Loss: 0.000571        Validation Loss: 0.005314
Validation loss decreased (0.005320 --> 0.005314).  Saving model ...
Epoch: 10         Training Loss: 0.000558        Validation Loss: 0.005141
Validation loss decreased (0.005314 --> 0.005141).  Saving model ...
Epoch: 11         Training Loss: 0.000541        Validation Loss: 0.005136
Validation loss decreased (0.005141 --> 0.005136).  Saving model ...
Epoch: 12         Training Loss: 0.000526        Validation Loss: 0.005035
Validation loss decreased (0.005136 --> 0.005035).  Saving model ...
Epoch: 13         Training Loss: 0.000510        Validation Loss: 0.004849
Validation loss decreased (0.005035 --> 0.004849).  Saving model ...
Epoch: 14         Training Loss: 0.000495        Validation Loss: 0.004880
Epoch: 15         Training Loss: 0.000479        Validation Loss: 0.004850
Epoch: 16         Training Loss: 0.000461        Validation Loss: 0.004787
Validation loss decreased (0.004849 --> 0.004787).  Saving model ...
Epoch: 17         Training Loss: 0.000442        Validation Loss: 0.004737
Validation loss decreased (0.004787 --> 0.004737).  Saving model ...
Epoch: 18         Training Loss: 0.000428        Validation Loss: 0.004653
Validation loss decreased (0.004737 --> 0.004653).  Saving model ...
Epoch: 19         Training Loss: 0.000404        Validation Loss: 0.004680
Epoch: 20         Training Loss: 0.000387        Validation Loss: 0.004707
Epoch: 21         Training Loss: 0.000363        Validation Loss: 0.004724
Epoch: 22         Training Loss: 0.000344        Validation Loss: 0.004578
Validation loss decreased (0.004653 --> 0.004578).  Saving model ...
Epoch: 23         Training Loss: 0.000323        Validation Loss: 0.004529
Validation loss decreased (0.004578 --> 0.004529).  Saving model ...
Epoch: 24         Training Loss: 0.000302        Validation Loss: 0.004560
Epoch: 25         Training Loss: 0.000284        Validation Loss: 0.004612
```

```
In [18]: # load the model that got the best validation accuracy
         model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

1.1.11 (IMPLEMENTATION) Test the Model

Try out your model on the test dataset of dog images. Use the code cell below to calculate and print the test loss and accuracy. Ensure that your test accuracy is greater than 10%.

```
In [19]: def test(loaders, model, criterion, use_cuda):

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

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

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

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

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

```
Test Loss: 3.823627


Test Accuracy: 12% (106/836)
```

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

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

1.1.12 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

Use the code cell below to write three separate data loaders for the training, validation, and test datasets of dog images (located at `dogImages/train`, `dogImages/valid`, and `dogImages/test`, respectively).

If you like, **you are welcome to use the same data loaders from the previous step**, when you created a CNN from scratch.

In [20]:
```
import os
import random
import requests
import time
import ast
import numpy as np
from glob import glob
import cv2
from tqdm import tqdm
from PIL import Image, ImageFile

import torch
import torchvision
from torchvision import datasets
import torchvision.transforms as transforms
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision.models as models


## TODO: Specify data loaders
#how many samples per batch to load
batch_size = 20

# number of subprocesses to use for data loading
num_workers = 0

# convert data to a normalized torch.FloatTensor
transform = transforms.Compose([transforms.Resize(size=256),
                                transforms.RandomResizedCrop(224),
                                transforms.RandomHorizontalFlip(),
                                transforms.ToTensor(),
                                transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0
data_dir = '/data/dog_images/'
image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x), transform)
                  for x in ['train', 'valid', 'test']}
loaders_transfer = {
    x: torch.utils.data.DataLoader(image_datasets[x], shuffle=True, batch_size=batch_si
    for x in ['train', 'valid', 'test']}
```

```
In [21]: print('Num training images: ', len(image_datasets['train']))
         print('Num test images: ', len(image_datasets['test']))
```

```
Num training images:  6680
Num test images:  836
```

1.1.13 (IMPLEMENTATION) Model Architecture

Use transfer learning to create a CNN to classify dog breed. Use the code cell below, and save your initialized model as the variable `model_transfer`.

```
In [22]: import torchvision.models as models
         import torch.nn as nn

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

```
In [23]: # Freezing the pre-trained parameters
         for param in model_transfer.features.parameters():
             param.requires_grad = False

         # Last layer input size
         input_size = model_transfer.classifier[6].in_features

         # Change last layer to predict correct number of classes
         model_transfer.classifier[6] = nn.Linear(input_size, 133)
```

```
In [24]: model_transfer
```

```
Out[24]: 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)
      )
    )
```

```
In [25]: # check if CUDA is available
         use_cuda = torch.cuda.is_available()

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

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

Answer: The VGG network architecture is known for its simple yet efficient network using only 3x3 convolutional layers all stacked on top of eachother increasing the depth hence the effective feature extraction. The VGG16 is a model that is pre-trained on the ImageNet dataset. In the process of transfer learning, the VGG16 model is loaded from PyTorch, pretrained, with all the convolutional layers. The last fully connected layer of the classifier is changed where the the output classes predicted equals the number of dog breeds in the dataset in use which is 133.

1.1.14 (IMPLEMENTATION) Specify Loss Function and Optimizer

Use the next code cell to specify a loss function and optimizer. Save the chosen loss function as `criterion_transfer`, and the optimizer as `optimizer_transfer` below.

```
In [26]: import torch.optim as optim
```

```
criterion_transfer = nn.CrossEntropyLoss()
optimizer_transfer = optim.SGD(model_transfer.classifier.parameters(), lr=0.01)
```

1.1.15 (IMPLEMENTATION) Train and Validate the Model

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

```
In [27]: from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True

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

             for epoch in range(1, n_epochs+1):

                 # initialize variables to monitor training and validation loss
                 train_loss = 0.0
                 valid_loss = 0.0

                 ###################
                 # train the model #
                 ###################
                 model.train()

                 for batch_idx, (data, target) in enumerate(loaders['train']):

                     # move to GPU
                     if use_cuda:
                         data, target = data.cuda(), target.cuda()
                     ## find the loss and update the model parameters accordingly
                     ## record the average training loss, using something like
                     ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_lo
                     optimizer.zero_grad()
                     output = model(data)
                     loss = criterion(output, target)
                     loss.backward()
                     optimizer.step()
                     train_loss += loss.item() * data.size(0)

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

```
                # move to GPU
                if use_cuda:
                    data, target = data.cuda(), target.cuda()
                ## update the average validation loss
                output = model(data)
                loss = criterion(output, target)
                valid_loss += loss.item() * data.size(0)


            train_loss = train_loss/len(loaders['train'].dataset)
            valid_loss = valid_loss/len(loaders['valid'].dataset)
            # print training/validation statistics
            print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
                epoch,
                train_loss,
                valid_loss
                ))

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

        # return trained model
        return model
```

```
In [10]: # train the model
         n_epochs = 10

         model_transfer = train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer,
```

```
Epoch: 1 	Training Loss: 2.068655 	Validation Loss: 1.171541
Validation loss decreased (inf --> 1.171541).  Saving model ...
Epoch: 2 	Training Loss: 1.158547 	Validation Loss: 0.989301
Validation loss decreased (1.171541 --> 0.989301).  Saving model ...
Epoch: 3 	Training Loss: 1.060748 	Validation Loss: 0.949551
Validation loss decreased (0.989301 --> 0.949551).  Saving model ...
Epoch: 4 	Training Loss: 0.992093 	Validation Loss: 0.904690
Validation loss decreased (0.949551 --> 0.904690).  Saving model ...
Epoch: 5 	Training Loss: 0.916072 	Validation Loss: 0.907708
Epoch: 6 	Training Loss: 0.869755 	Validation Loss: 0.952865
Epoch: 7 	Training Loss: 0.847218 	Validation Loss: 0.828071
Validation loss decreased (0.904690 --> 0.828071).  Saving model ...
Epoch: 8 	Training Loss: 0.849244 	Validation Loss: 0.854152
Epoch: 9 	Training Loss: 0.817955 	Validation Loss: 0.858325
```

```
Epoch: 10 	Training Loss: 0.782844 	Validation Loss: 0.857221
```

```
In [28]: # load the model that got the best validation accuracy (uncomment the line below)
         model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

1.1.16 (IMPLEMENTATION) Test the Model

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

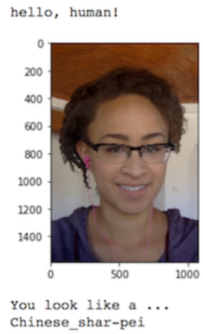
```
In [29]: test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.918351


Test Accuracy: 74% (620/836)
```

1.1.17 (IMPLEMENTATION) Predict Dog Breed with the Model

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

```
In [30]: ### TODO: Write a function that takes a path to an image as input
         ### and returns the dog breed that is predicted by the model.

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

         def image_to_tensor(img_path):
             '''
             As per Pytorch documentations: All pre-trained models expect input images normalize
             i.e. mini-batches of 3-channel RGB images
             of shape (3 x H x W), where H and W are expected to be at least 224.
             The images have to be loaded in to a range of [0, 1] and
             then normalized using mean = [0.485, 0.456, 0.406] and std = [0.229, 0.224, 0.225].
             You can use the following transform to normalize:
             '''
             img = Image.open(img_path).convert('RGB')
             transformations = transforms.Compose([transforms.Resize(size=224),
                                                    transforms.CenterCrop((224,224)),
                                                   transforms.ToTensor(),
                                                   transforms.Normalize(mean=[0.485, 0.456, 0.406
                                                                        std=[0.229, 0.224, 0.225]
             image_tensor = transformations(img)[:3,:,:].unsqueeze(0)
             return image_tensor
```



Sample Human Output

```
def predict_breed_transfer(img_path, model, class_names):
    # load the image and return the predicted breed
    image_tensor = image_to_tensor(img_path)

    if use_cuda:
        image_tensor = image_tensor.cuda()
        model = model.cuda()

    model.eval()
    prediction = model(image_tensor)

    index = torch.max(prediction,1)[1].item()

    return class_names[index]
```

In [31]:
```
# Test
predict_breed_transfer('./images/Welsh_springer_spaniel_08203.jpg', model_transfer, cla
```

Out[31]: 'Welsh springer spaniel'

Step 5: Write your Algorithm

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

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

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

1.1.18 (IMPLEMENTATION) Write your Algorithm

In [36]:
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.
```

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

    if face_detector(img_path):
        print("\n\nHuman Detected!")
        plt.show()
        predict = predict_breed_transfer(img_path, model_transfer, class_names)
        print("You look like a ", predict)
    elif dog_detector(img_path):
        print("\n\nDog Detected!")
        plt.show()
        predict = predict_breed_transfer(img_path, model_transfer, class_names)
        print("You look like a ", predict)
    else:
        print("No dog or human detected!")
```

Step 6: Test Your Algorithm

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

1.1.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

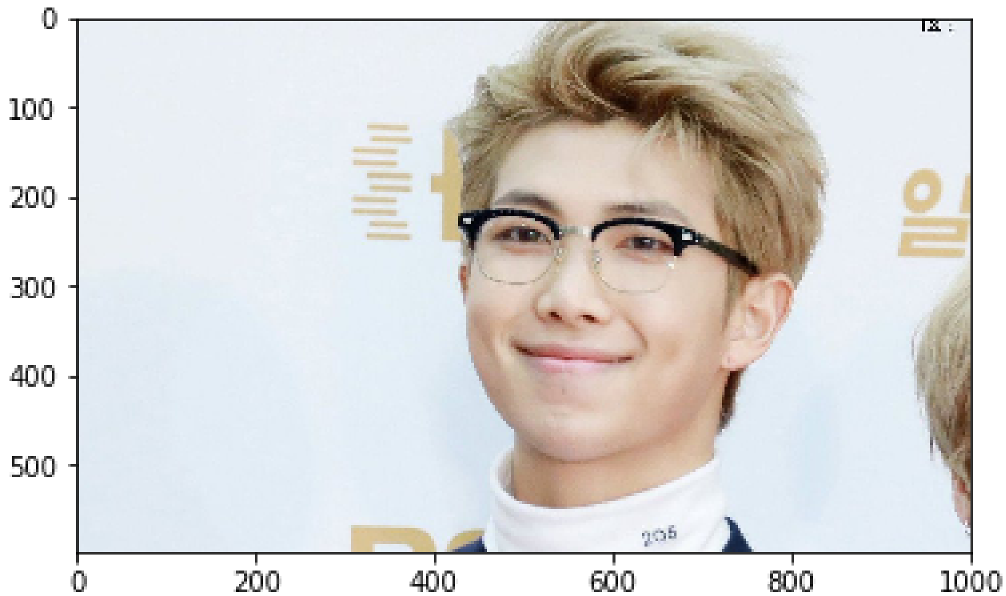
Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

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

Answer: The output is as expected from the model for the given images. - There is scope for improving the model performance by experimenting with different combinations of hyper-parameters. - The classifier component of the architecture can also be designed customized to the problem at hand. - Training the model on larger dataset with performing different data transformations to the images can also improve the accuracy.

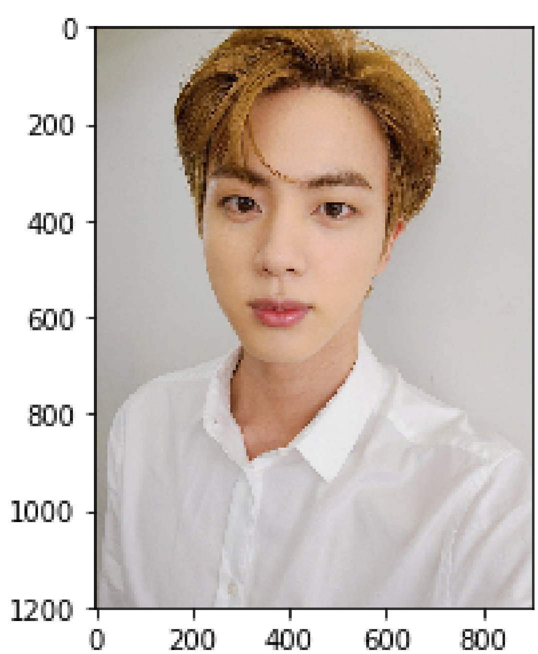
In [37]:
```python
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.
human_files = ['./my_images/RM.jpg', './my_images/jin.jpg', './my_images/v.jpg']
dog_files = ['./my_images/tan.jpg', './my_images/mickey.jpg', './my_images/moni.jpg']
## suggested code, below
for file in np.hstack((human_files[:3], dog_files[:3])):
    run_app(file)
```

Human Detected!



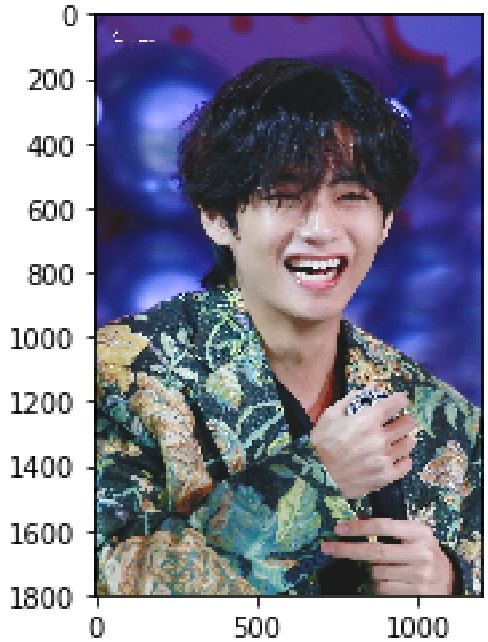
You look like a Lowchen

Human Detected!



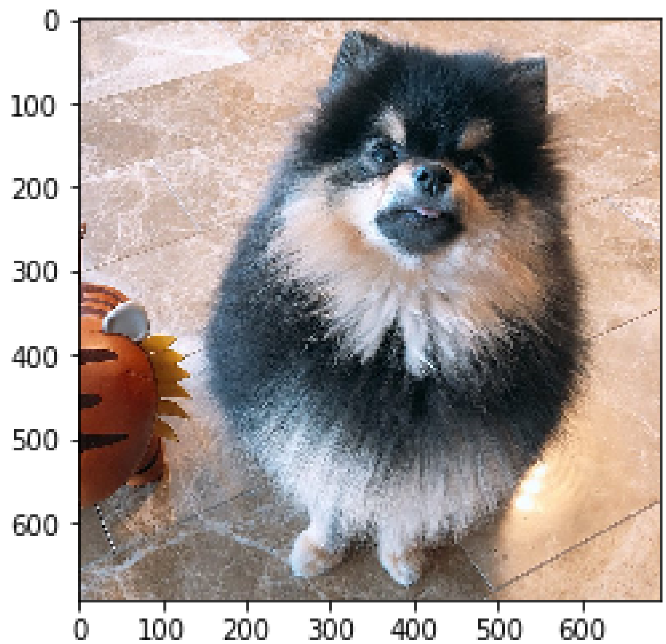
```
You look like a  American eskimo dog
```

```
Human Detected!
```



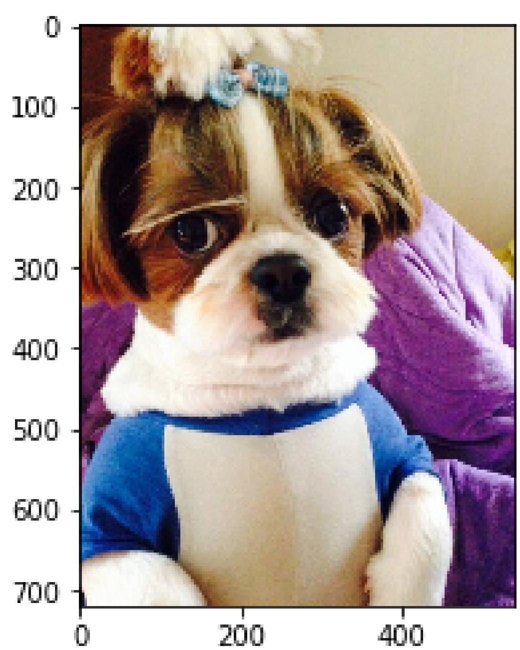

```
You look like a  Italian greyhound
```

```
Dog Detected!
```



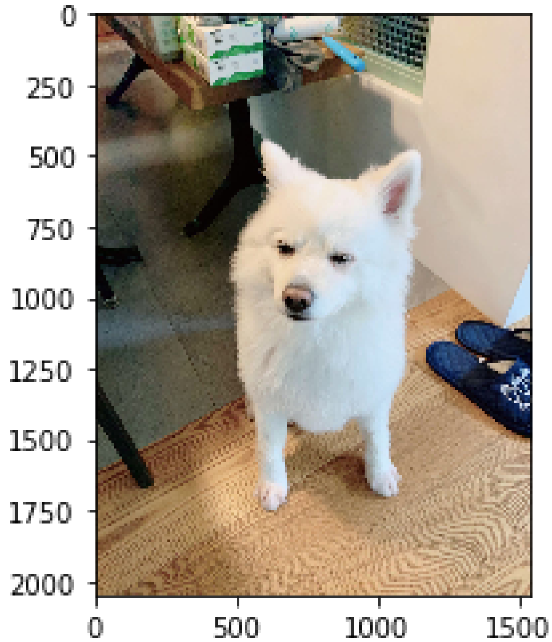

```
You look like a  Pomeranian
```

```
Dog Detected!
```



```
You look like a  English toy spaniel
```

```
Dog Detected!
```




```
You look like a  American eskimo dog
```

In [ ]: