Capstone Project **Seongjae Ryu**

Machine Learning Engineer Nanodegree **August 20, 2021**

CNN Project: Dog Breed Classifier

Project Report

I. Definition

Project Overview

Dog breed classifier is one of the popular CNN(convolutional neural network) projects [1]. The main problem is to identify a dog breed from an input image. It also needs to be identified whether it is an image of a dog or not. In case when a given input is identified as an image with a human face, a similar dog breed will be provided.

This idea is based on supervised machine learning with a multi-class classification problem. The intention of this model is to deploy APIs and build an application with the APIs.

Problem Statement

The problem is to build a dog breed classification model and deploy this model for an application.

- **Dog-image identifier**: Inferring whether it is a dog image or not from an input image.
- **Dog-breed classifier**: Inferring a dog breed from an input image.
- **Human-face-image identifier**: Inferring whether it is a human face image or not from an input image.
- **Human-face-image Resemble-dog classifier**: Inferring a dog breed, resembling a human face input image.

Metrics

Only the dataset for the main problem of building a dog breed classification is balanced. 'Accuracy' is an optimal means of performance measurement only when the dataset is balanced. Thus 'accuracy' is used only for measuring 'dog breed classification' performance.

1. Human-face-image identifier
    - The percentage of human face images inferred as a human face:
        - $\frac{Number\ of\ predictions\ as\ human-face}{Total\ number\ of\ prediction\ with\ human\ face\ dataset}$

- The percentage of dog images inferred as not a human face:
  - $\frac{Number\ of\ predictions\ as\ not-human-face}{Total\ number\ of\ prediction\ with\ dog\ dataset}$

2. Dog-image identifier
   - The percentage of dog images inferred as a dog:
     - $\frac{Number\ of\ predictions\ as\ dog}{Total\ number\ of\ prediction\ with\ dog\ dataset}$
   - The percentage of human face images inferred as not a dog:
     - $\frac{Number\ of\ predictions\ as\ not-dog}{Total\ number\ of\ prediction\ with\ human\ face\ dataset}$

3. Dog-breed classifier
   - $Accuracy = \frac{Number\ of\ correct\ predictions}{Total\ number\ of\ predictions}$

II. Analysis

Data Exploration

Dog images and human-face images are included in the dataset, which is labeled, organized, and provided by Udacity [2]. The dog image dataset is balanced, but the human-face dataset is imbalanced.

1. **Dog image dataset [3]**
   a. It contains 8,351 RGB images of dogs with each labeled file name. The images are organized in each directory; train, test, and valid. There are 6,680 train images, 836 test images, and 835 valid images. Each breed of 133 is grouped in each folder with label number and breed name; i.e. '/dog_images/train/103.Mastiff/Mastiff_06826.jpg'. Image sizes are varied.
   b. Balanced dataset, which is optimal to use 'accuracy'.

```
pd.Series(dog_count_dict).describe()
```

```
count    133.000000
mean      62.789474
std       14.852330
min       33.000000
25%       53.000000
50%       62.000000
75%       76.000000
max       96.000000
```

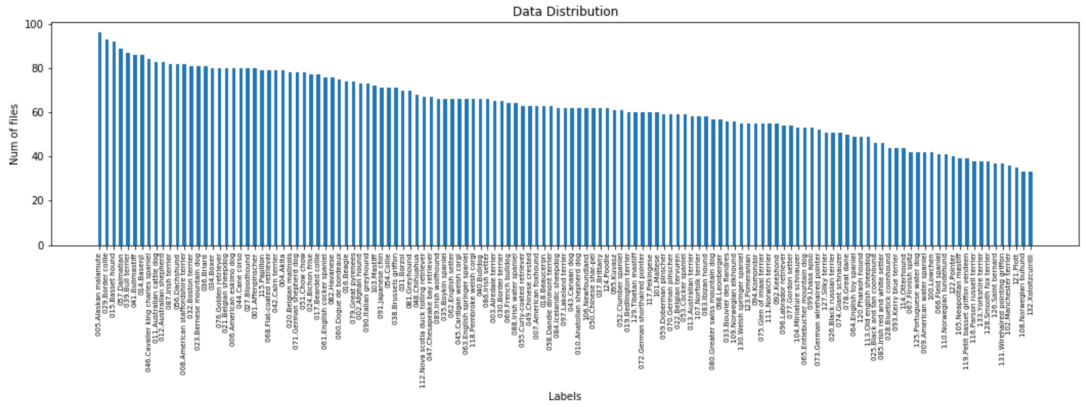
2. **Human-face dataset [4]**
   a. It contains 13,233 RGB human-face images with each labeled file name. Images are grouped by each person in each folder with label number and person's name; i.e. '/Daniele_Bergamin/Daniele_Bergamin_0001.jpg'. There are 5,750 folders and the data is imbalanced, such as one has few images and the other has many images. The image size is 250 by 250 px.
   b. Imbalanced dataset, which is not optimal to use 'accuracy'.

```
pd.Series(human_count_dict).describe()
```

```
count    5749.000000
mean        2.301792
std         9.016410
min         1.000000
25%         1.000000
50%         1.000000
75%         2.000000
max       530.000000
```

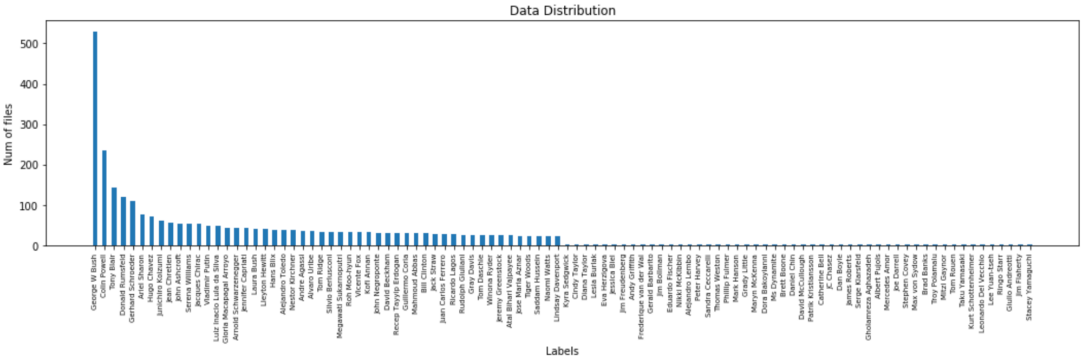
Exploratory Visualization

1. **Dog image dataset**: It is suitable for modeling. Each label has enough data for modeling and the data is properly divided into each set of training, validation, and test.



2. **Human-face dataset**: It is suitable for the purpose of testing whether it is a human face or not. But it is not suitable for creating a new model. Some labels have only 1 image. This is the reason for using a pre-trained face detection model.

```
5749: Num of labels are too large. So it will display only Top 50 and Bottom 50 labels.
```



Algorithms and Techniques

A convolutional neural network is used for multi-class classification, dog breed classifier. VGG16 is chosen for a pre-trained CNN model because it achieves 92.7% top-5 test accuracy in ImageNet, which is a dataset of over 14 million images belonging to 1000 classes [5]. One of the cons of using pre-trained CNN type models is that it only fits on specific situations. If the condition changes, such as parameter modification, the modified model needs to be trained a lot again to fit the very large amount of weights.

Haar Cascade Classifier is used for a face detection model. One of the primary benefits of using the classifier is the fastness and the lightness, which fit for the situation that the face detection is not a primary problem [6].

1. **Dog-image identifier**: Pre-trained VGG-16 torchvision model [7].
    a. Load pre-trained VGG-16 model via torchvision.
    b. Load image and convert into tensor image.
    c. Infer the image and check whether the output is in 133 breeds or not.

2. **Dog-breed classifier**: Fitted pre-trained VGG-16 torchvision model with custom classifier for 133 breeds.
    a. Load pre-trained VGG-16 model via torchvision.
    b. Define a new custom classifier to fit the model into the dataset with 133 breed categories.
    c. Load image and convert into tensor image with transforms.
    d. Train the model with train dataset and valid dataset.
    e. Test the model with a test dataset.
    f. Deploy the model and infer the most likely breed.

3. **Human-face-image identifier**: OpenCV with pre-trained face detector, haarcascade_frontalface_alt [8].
    a. Load pre-trained OpenCV model with haarcascade_frontalface_alt.xml.
    b. Load and convert RGB image into grayscale.
    c. Infer the image.

4. **Human-face-image Resemble-dog classifier**: Combination of human-face-image identifier and dog-breed classifier.
    a. Infer the image via human-face-image identifier and if the inference tells it is a human face image, infer the breed via dog-breed classifier.

Benchmark

1. Human-face-image identifier must have 70% or above true inference and vice versa.
    - The percentage of human face images inferred as a human face:
        - $\frac{Number\ of\ predictions\ as\ human-face}{Total\ number\ of\ prediction\ with\ human\ face\ dataset}$
    - The percentage of dog images inferred as not a human face:
        - $\frac{Number\ of\ predictions\ as\ not-human-face}{Total\ number\ of\ prediction\ with\ dog\ dataset}$

2. Dog-image identifier must have 70% or above true inference and vice versa.
    - The percentage of dog images inferred as a dog:
        - $\frac{Number\ of\ predictions\ as\ dog}{Total\ number\ of\ prediction\ with\ dog\ dataset}$
    - The percentage of human face images inferred as not a dog:
        - $\frac{Number\ of\ predictions\ as\ not-dog}{Total\ number\ of\ prediction\ with\ human\ face\ dataset}$

3. Dog-breed classifier must have 60% or above accuracy.
    - $Accuracy = \frac{Number\ of\ correct\ predictions}{Total\ number\ of\ predictions}$

III. Methodology

Data Processing

Dog images and human-face images are included in the dataset, which is labeled, organized, and provided by Udacity [2]. Each data path is stored in a 'numpy array' and it is sliced for model tests. But further transformations are still needed for each model after this data processing.

1. **Data path list creation of dataset:** The list of each dataset into each numpy array by using 'glob' and 'numpy' library.

```
# import numpy as np
# from glob import glob

# Setup data dir for human and dog images
# format: '/*/**'
human_data_dir = '/data/lfw'
dog_data_dir = '/data/dog_images'

# Load filenames for human and dog images
human_files = np.array(glob(f'{human_data_dir}/*/*'))
dog_files = np.array(glob(f'{dog_data_dir}/*/*/*'))
```

2. **Data path list division for test:** Slicing each numpy array

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

Implementation

1. **Import library:** Python(3.6.3), PIL(5.2.0), cv2(3.3.1), matplotlib(2.1.0), numpy(1.12.1), pandas(0.23.3), seaborn(0.8.1), torch(0.4.0), torchvision(0.2.1), tqdm(4.11.2)

```
import numpy as np
from glob import glob

import os

import sys
from itertools import groupby
import itertools
import pprint

import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
%matplotlib inline
import seaborn as sb

import cv2

import torch
import torchvision.models as models

from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

from PIL import Image
import torchvision.transforms as transforms
```

```
import json
import ast
import urllib

import time
from datetime import datetime

import os
from torchvision import datasets
```

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

```
import torch.optim as optim
```

2. Import dataset

2.1. Load dataset

2.1.1. Load dataset by ‘data directory’ and check each file and label number is correct.

```
# import numpy as np
# from glob import glob

# Setup data dir for human and dog images
# format: '/*/**'
human_data_dir = '/data/lfw'
dog_data_dir = '/data/dog_images'

# Load filenames for human and dog images
human_files = np.array(glob(f'{human_data_dir}/*/*'))
dog_files = np.array(glob(f'{dog_data_dir}/*/*/*'))

# Count files
print(f'human_files: {len(human_files)} file(s)')
print(f'dog_files  : {len(dog_files)} file(s)')
```

```
human_files: 13233 file(s)
dog_files  : 8351 file(s)
```

2.1.2. Check the number of dog data labels is correct.

```
def all_equal(iterable):
    g = groupby(iterable)
    return next(g, True) and not next(g, False)
```

```
def craete_sub_dir_list(path):
    sub_dir_list = []
    for sub_path in glob(f'{path}/*', recursive=True):
        sub_dir_list.append(sub_path[len(path)+1:])
    return sub_dir_list
```

```
def create_label(sub_dir):
    label_list = craete_sub_dir_list(sub_dir)
    return label_list
```

```
def count_label(sub_dir, is_msg_on=True):
    label_list = create_label(sub_dir)
    label_count = len(label_list)
    if is_msg_on:
            print(f'[_] Data Path: {sub_dir}\n',
                  f'   Num. of labels: {label_count}')
    return label_count
```

```
def find_intersection(dictionary):
    inte_set = set(list(dictionary.values())[0])
    for l in [v for v in dictionary.values()]:
        inte_set = inte_set.intersection(l)
    return inte_set
```

```
def check_is_same_label_w_sub(path, is_msg_on=True):
    sub_dir_list = craete_sub_dir_list(path)
    if is_msg_on:
        print(f'Sub directory list of the path: {sub_dir_list}')
    label_dict = {}
    for i, sub_dir in enumerate(sub_dir_list):
        label_dict[i] = craete_sub_dir_list(f'{path}/{sub_dir}')
        label_dict[i] = sorted(label_dict[i])

    # Error: if label sizes are inequal, return false with error msg
    label_size_list = [len(x) for x in label_dict.values()]
    if not all_equal(iter(label_size_list)):
        if is_msg_on:
            print(f'[!] Alert: inequal label size\n',
                  f'   Data Path: {path}\n',
                  f'   Labels: {list(zip((label_dict.keys()),label_size_list))}')
        return False

    # Good: if all labels are same
    elif all_equal(iter(label_dict.values())):
        if is_msg_on:
            print(f'[v] Good to go. All the labels are same!\n',
                  f'   Data Path: {path}\n',
                  f'   Num. of sub dirs: {len(sub_dir_list)}\n',
                  f'   Num. of labels: {label_size_list[0]}')
        return True, label_size_list[0]

    # Error: if label sizes are equal but they are not same, return false
    else:
        if is_msg_on:
            inte_set = find_intersection(label_dict)
            print(f'[!] Alert: Label sizes are equal but not same.\n',
                  f'   Data Path: {path}\n',
                  f'   Different Labels below.____\n')
            for i, labels in label_dict.items():
                # sub dir : e.g. train, validation, test
                _sub_dir = sub_dir_list[i]
                for label in labels:
                    print(f'{_sub_dir}: {label}')
            print('END_____________________________\n')
        return False
```

```
# Check data: Dog
check_is_same_label_w_sub(dog_data_dir, is_msg_on=True)
```

```
Sub directory list of the path: ['train', 'test', 'valid']
[v] Good to go. All the labels are same!
    Data Path: /data/dog_images
    Num. of sub dirs: 3
    Num. of labels: 133
(True, 133)
```

2.2. Explore the data

2.2.1. **Check the distribution of human images**. If the number of labels is above 150, plot_human_data_distribution function will show a chart of top 50 and bottom 50. The number of the human labels is 5749, so the function shows the result of top 50 and bottom 50.

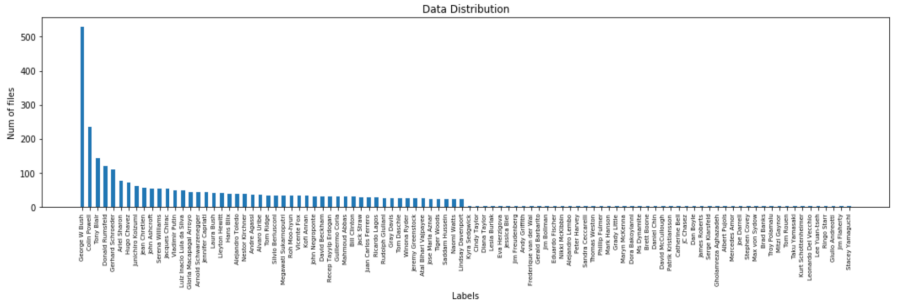
```
def plot_human_data_distribution(dictionary, show_title = False):
    x = range(len(dictionary))
    y = dictionary.values()

    if not show_title:
        # Figure Size
        fig, ax = plt.subplots(figsize =(12, 4))
        # Bar Plot
        ax.bar(x, y)
        # Remove x Ticks
        ax.xaxis.set_ticks([])

    else:
        mode = False
        if len(x) > 150:
            print(f'{len(x)}: Num of labels are too large. So it will display only Top 50 and Bottom 50
labels.')
            mode = True
            x = range(100)
            y = list(dictionary.values())[:50] + list(dictionary.values())[-50:]
        # Figure Size
        fig, ax = plt.subplots(figsize =(18, 4))
        ax.bar(x, y, 0.5, align='center')
        if mode:
            xlabels = list(dictionary.keys())[:50] + list(dictionary.keys())[-50:]
        else:
            xlabels = dictionary.keys()
        ax.set_xticks(x)
        ax.set_xticklabels(xlabels, rotation=90, fontsize=7)

    # Show Plot
    plt.title('Data Distribution')
    plt.ylabel('Num of files')
    plt.xlabel('Labels')
    plt.show()
```

```
human_count_dict = creat_count_dict(human_files, spliter='/', spliter_idx=3)
human_count_dict_sorted = sort_dict(human_count_dict, sorted_by='value' ,reverse=True, replace_key=Tru
e)

_check = human_count_dict_sorted
print(dict(itertools.islice(_check.items(), 5)))
print(dict(itertools.islice(_check.items(), len(_check)-5, len(_check))))
del _check
```

```
{'George W Bush': 530, 'Colin Powell': 236, 'Tony Blair': 144, 'Donald Rumsfeld': 121, 'Gerhard Schroed
er': 109}
{'Lee Yuan-tseh': 1, 'Ringo Starr': 1, 'Giulio Andreotti': 1, 'Jim Flaherty': 1, 'Stacey Yamaguchi': 1}
```

```
plot_human_data_distribution(human_count_dict_sorted, show_title = True)
```

```
5749: Num of labels are too large. So it will display only Top 50 and Bottom 50 labels.
```



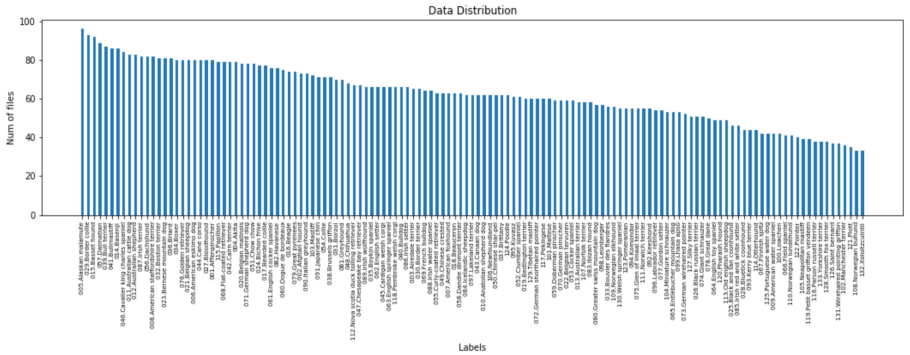
2.2.2. Check the distribution of dog images.

```
def plot_dog_data_distribution(dictionary, show_title = False):
    x = range(len(dictionary))
    y = dictionary.values()

    if not show_title:
        # Figure Size
        fig, ax = plt.subplots(figsize =(12, 4))
        # Bar Plot
        ax.bar(x, y)
        # Remove x Ticks
        ax.xaxis.set_ticks([])

    else:
        # Figure Size
        fig, ax = plt.subplots(figsize =(18, 4))
        ax.bar(x, y, 0.5, align='center')
        xlabels = dictionary.keys()
        ax.set_xticks(x)
        ax.set_xticklabels(xlabels, rotation=90, fontsize=7)

    # Show Plot
    plt.title('Data Distribution')
    plt.ylabel('Num of files')
    plt.xlabel('Labels')
    plt.show()
```

```
dog_count_dict = creat_count_dict(dog_files, spliter='/', spliter_idx=4)
dog_count_dict_sorted = sort_dict(dog_count_dict, sorted_by='value' ,reverse=True, replace_key=True)

_check = dog_count_dict_sorted
print(dict(itertools.islice(_check.items(), 5)))
print(dict(itertools.islice(_check.items(), len(_check)-5, len(_check))))
del _check
```

```
{'005.Alaskan malamute': 96, '029.Border collie': 93, '015.Basset hound': 92, '057.Dalmatian': 89, '03
9.Bull terrier': 87}
{'131.Wirehaired pointing griffon': 37, '102.Manchester terrier': 36, '121.Plott': 35, '108.Norwegian b
uhund': 33, '132.Xoloitzcuintli': 33}
```

```
plot_dog_data_distribution(dog_count_dict_sorted, show_title = True)
```



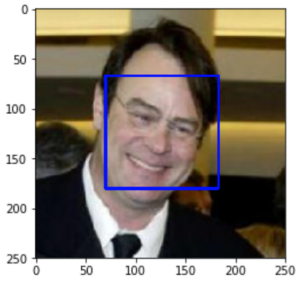
3. Detect humans

3.1. **Implement a face detector**: With the implementation, the model will infer face from a human face image. Then, the detected area will be shown.

```
def implement_face_cascade_w_test(human_files, xml='haarcascades/haarcascade_frontalface_alt.xml'):
    # extract pre-trained face detector
    face_cascade = cv2.CascadeClassifier(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()

    return face_cascade
```

```
face_cascade = implement_face_cascade_w_test(human_files, xml='haarcascades/haarcascade_frontalface_alt.xml')
```

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



3.2. **Test the face detector with part of the whole dataset:** to get the percentage of human face images inferred as a human face and the percentage of dog images inferred as not a human face.

```
def face_detector_test(img_files):
    '''Test the performance of the face_detector algorithm on the images in human_files_short and dog_files_short.
    '''

    num_files = len(img_files)

    print(f'\nStart face detector testing {namestr(img_files, globals())}...')
    print(f'\nNum. of files: {num_files}')

    count = 0
    for f in img_files:
        if face_detector(f):
            count += 1
    ratio = count / num_files

    print(f'\nNum. of face detected: {count}')
    print(f'\nDetected percent: {round(ratio, 4)*100}%')
    print(f'\nEnd testing {namestr(img_files, globals())}. ^-^')

    return round(ratio, 4)
```

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

```
face_detector_test(human_files_short)
```

```
Start face detector testing human_files_short...

Num. of files: 100

Num. of face detected: 98

Detected percent: 98.0%

End testing human_files_short. ^-^
0.98
```

```
face_detector_test(dog_files_short)
```

```
Start face detector testing dog_files_short...

Num. of files: 100

Num. of face detected: 17

Detected percent: 17.0%

End testing dog_files_short. ^-^
0.17
```

4. Dog Detector

4.1. **Implement a pre-trained classifier model through 'torchvision' library**:

VGG16 is used. The input data is transformed into a randomly cropped 224 by 224 tensor. If CUDA is available, the tensor object and the model object will work with GPU. The model returns predicted classes. An index of the maximum value among the classes will be used for the final output of VGG16_predict function.

```
# 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()
```

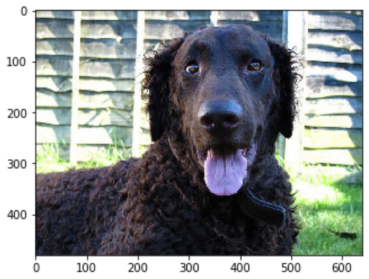
```
def VGG16_predict(img_path):
    '''
    Use pre-trained VGG-16 model to obtain index corresponding to
    predicted ImageNet class for image at specified path

    Args:
        img_path: path to an image

    Returns:
        Index corresponding to VGG-16 model's prediction
    '''
    ## Load and pre-process an image from the given img_path
    ## Return the *index* of the predicted class for that image

    # PIL img
    img = Image.open(img_path)

    # Compose Set
    compose_set = [transforms.RandomResizedCrop(224),
                   transforms.ToTensor()]

    # Apply compose setting
    transform = transforms.Compose(compose_set)

    # Apply transform including ToTensor and unsqueeze
    img_tensor = transform(img).unsqueeze(0)

    if torch.cuda.is_available():
        img_tensor = img_tensor.cuda()

    # Apply model(VGG16)
    predict = VGG16(img_tensor)

    # Tensor.cpu() to copy the tensor to host memory
    if torch.cuda.is_available():
        predict = predict.cpu()

    # Argmax to get index of the maximum value(among predicted classes)
    index = predict.data.numpy().argmax()

    return index # predicted class index
```

4.2. **Use pre-trained classifier model as a dog detector**: The labels of the dog dataset correspond to dictionary keys 151-268, inclusive, to include all categories from 'Chihuahua' to 'Mexican hairless' of model label text file(imagenet 1000 class idx to human readable labels) [9]. So if an input image is a dog, the index will be in the dictionary keys.

```
### 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)

    is_inclusive = (151 <= index) & (index <= 268) # true/false
    return is_inclusive
```

```
plt.imshow(Image.open('images/Curly-coated_retriever_03896.jpg'))
```

```
<matplotlib.image.AxesImage at 0x7fa7a819bd68>
```




```
dog_detector('images/Curly-coated_retriever_03896.jpg')
```

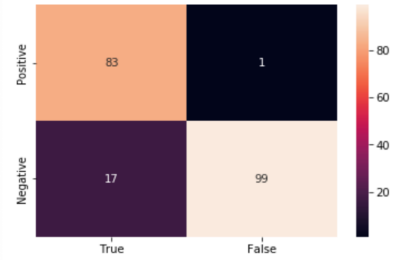
```
True
```

4.3. **Test dog detector**: Test the performance of the dog_detector function on the images in human_files_short and dog_files_short.

```
def dog_detector_test_heatmap(p_img_files, n_img_files):
    p_num_files = len(p_img_files)
    n_num_files = len(n_img_files)

    print(f'Start dog detector testing ...')
    print(f'Num. of positive files: {p_num_files}')
    print(f'Num. of negative files: {n_num_files}')

    tp = 0
    tn = 0
    fp = 0
    fn = 0

    for f in p_img_files:
        if dog_detector(f):
            tp += 1
        else:
            fn += 1

    for f in n_img_files:
        if not dog_detector(f):
            tn += 1
        else:
            fp += 1

    df = pd.DataFrame([[tp, fp], [fn, tn]], index = ['Positive', 'Negative'], columns = ['True', 'Fals
e'])
    plt.figure(figsize = (6,4))
    g = sb.heatmap(df, annot=True)
    g.set_yticklabels(labels=g.get_yticklabels(), va='center')
    print(f'Dog detector test finished.')
```

```
dog_detector_test_heatmap(dog_files_short, human_files_short)
```

```
Start dog detector testing ...
Num. of positive files: 100
Num. of negative files: 100
Dog detector test finished.
```



4.4. **Compare torchvision models**: Compare the performance of ‘torchvision’ models to decide whether to use VGG16 or not. For the comparison, define functions for multiple-model creation and tests [10].

```
def load_torchvision_model(model_name = None):
    ref_url = 'https://pytorch.org/vision/stable/models.html'
    print(f'Loading model.. and reference url below.\n{ref_url}\n')

    if type(model_name) == str:
        model_name = model_name.lower()

    # Default model
    if model_name == 0 or model_name == None:
        model = models.vgg16(pretrained=True)

    elif model_name == 1 or model_name == 'resnet18':
        model = models.resnet18(pretrained=True)
    elif model_name == 2 or model_name == 'alexnet':
        model = models.alexnet(pretrained=True)
    elif model_name == 3 or model_name == 'squeezenet1_0':
        model = models.squeezenet1_0(pretrained=True)
    elif model_name == 4 or model_name == 'vgg16':
        model = models.vgg16(pretrained=True)
    elif model_name == 5 or model_name == 'densenet161':
        model = models.densenet161(pretrained=True)
    elif model_name == 6 or model_name == 'inception_v3':
        model = models.inception_v3(pretrained=True)
    elif model_name == 7 or model_name == 'googlenet':
        model = models.googlenet(pretrained=True)
    elif model_name == 8 or model_name == 'shufflenet_v2_x1_0':
        model = models.shufflenet_v2_x1_0(pretrained=True)
    elif model_name == 9 or model_name == 'mobilenet_v2':
        model = models.mobilenet_v2(pretrained=True)
    elif model_name == 10 or model_name == 'mobilenet_v3_large':
        model = models.mobilenet_v3_large(pretrained=True)
    elif model_name == 11 or model_name == 'mobilenet_v3_small':
        model = models.mobilenet_v3_small(pretrained=True)
    elif model_name == 12 or model_name == 'resnext50_32x4d':
        model = models.resnext50_32x4d(pretrained=True)
    elif model_name == 13 or model_name == 'resnet50_2':
        model = models.wide_resnet50_2(pretrained=True)
    elif model_name == 14 or model_name == 'mnasnet1_0':
        model = models.mnasnet1_0(pretrained=True)

    else:
        print('Error: Please check the model name.')
        return

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

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

    torchvision_model = model
    return torchvision_model
```

```
def torchvision_model_predict(img_path, torchvision_model):
    # PIL img
    img = Image.open(img_path)

    # Compose Set
    compose_set = [transforms.RandomResizedCrop(224),
                   transforms.ToTensor()]

    # Apply compose setting
    transform = transforms.Compose(compose_set)

    # Apply transform including ToTensor and unsqueeze
    img_tensor = transform(img).unsqueeze(0)

    if torch.cuda.is_available():
        img_tensor = img_tensor.cuda()

    # Apply model(VGG16)
    predict = torchvision_model(img_tensor)

    # Tensor.cpu() to copy the tensor to host memory
    if torch.cuda.is_available():
        predict = predict.cpu()

    # Argmax to get index of the maximum value(among predicted classes)
    index = predict.data.numpy().argmax()

    return index # predicted class index


def torchvision_dog_detector(img_path, torchvision_model):
    index = torchvision_model_predict(img_path, torchvision_model)
    is_inclusive = (151 <= index) & (index <= 268)
    return is_inclusive


def test_torchvision_dog_detector(p_img_files, n_img_files, torchvision_model):
    start_time = time.time()
    print(f'{datetime.now().strftime("%H:%M:%S")}: Start testing')
    tp = 0
    tn = 0
    fp = 0
    fn = 0

    for img_path in p_img_files:
        if torchvision_dog_detector(img_path, torchvision_model):
            tp += 1
        else:
            fn += 1

    for img_path in n_img_files:
        if not torchvision_dog_detector(img_path, torchvision_model):
            tn += 1
        else:
            fp += 1

    print(f'{datetime.now().strftime("%H:%M:%S")}: End testing')
    end_time = time.time()
    seconds_elapsed = end_time - start_time

    return [[[tp, fp], [fn, tn]], seconds_elapsed]
```

```
def test_torchvision_dog_detectors(p_img_files, n_img_files, torchvision_model_list):
    result_dict = {}
    for model_name in torchvision_model_list:
        print(f'Load {model_name}')

        # Load Model
        torchvision_model = load_torchvision_model(model_name = model_name)
        # Save Prediction Result
        result_dict[model_name] = test_torchvision_dog_detector(p_img_files, n_img_files, torchvision_model)
        print(f'{model_name} prediction result saved.\n')

        # Clear GPU memody
        del torchvision_model
        torch.cuda.empty_cache()
    return result_dict
```

Do test 5 times to compare the performance of each model: 'resnet18', 'alexnet', 'squeezenet1_0', 'vgg16', 'densenet161' [10].

```
torchvision_model_list = ['resnet18', 'alexnet', 'squeezenet1_0',
                          'vgg16', 'densenet161', 'inception_v3',
                          'googlenet', 'shufflenet_v2_x1_0', 'mobilenet_v2',
                          'mobilenet_v3_large', 'mobilenet_v3_small', 'resnext50_32x4d',
                          'resnet50_2', 'mnasnet1_0']
```

```
torchvision_dog_test_result_dict = test_torchvision_dog_detectors(dog_files_short, human_files_short,
                                                                  torchvision_model_list[:5])
torchvision_dog_test_result_dict_1 = test_torchvision_dog_detectors(dog_files_short, human_files_short,
                                                                  torchvision_model_list[:5])
torchvision_dog_test_result_dict_2 = test_torchvision_dog_detectors(dog_files_short, human_files_short,
                                                                  torchvision_model_list[:5])
torchvision_dog_test_result_dict_3 = test_torchvision_dog_detectors(dog_files_short, human_files_short,
                                                                  torchvision_model_list[:5])
torchvision_dog_test_result_dict_4 = test_torchvision_dog_detectors(dog_files_short, human_files_short,
                                                                  torchvision_model_list[:5])
torchvision_dog_test_result_dict_avg = {}
for model_name in torchvision_model_list[:5]:
    tmp = np.array([torchvision_dog_test_result_dict[model_name][0],
                    torchvision_dog_test_result_dict_1[model_name][0],
                    torchvision_dog_test_result_dict_2[model_name][0],
                    torchvision_dog_test_result_dict_3[model_name][0],
                    torchvision_dog_test_result_dict_4[model_name][0]])
    sec = np.array([torchvision_dog_test_result_dict[model_name][1],
                    torchvision_dog_test_result_dict_1[model_name][1],
                    torchvision_dog_test_result_dict_2[model_name][1],
                    torchvision_dog_test_result_dict_3[model_name][1],
                    torchvision_dog_test_result_dict_4[model_name][1]])
    torchvision_dog_test_result_dict_avg[model_name] = np.mean(tmp, axis=0).tolist(), int(np.mean(sec, axis=0))
```

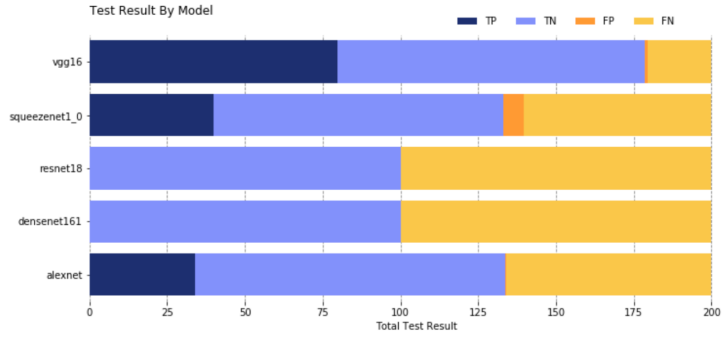
Compare the result and select a model with high true result and low false result. Among the five models, VGG16 fits best for the dataset.

```
def plot_df_n_test_t_d_d(df):
    fields = ['True_Positivie','True_Negative','False_Positive','False_Negative']
    colors = ['#1D2F6F', '#8390FA', '#FF9933', '#FAC748']
    labels = ['TP', 'TN', 'FP', 'FN']

    df[fields] = df[fields].astype(float)
    df_grouped = df.groupby('Model_Name').mean()

    # figure and axis
    fig, ax = plt.subplots(1, figsize=(12, 5))
    # plot bars
    left = len(df_grouped) * [0]
    for idx, name in enumerate(fields):
        plt.barh(df_grouped.index, df_grouped[name], left = left, color=colors[idx])
        left = left + df_grouped[name]
    # title, legend, labels
    plt.title('Test Result By Model\n', loc='left')
    plt.legend(labels, bbox_to_anchor=([0.55, 1, 0, 0]), ncol=4, frameon=False)
    plt.xlabel('Total Test Result')
    # remove spines
    ax.spines['right'].set_visible(False)
    ax.spines['left'].set_visible(False)
    ax.spines['top'].set_visible(False)
    ax.spines['bottom'].set_visible(False)
    # adjust limits and draw grid lines
    plt.ylim(-0.5, ax.get_yticks()[-1] + 0.5)
    ax.set_axisbelow(True)
    ax.xaxis.grid(color='gray', linestyle='dashed')
    plt.show()
```

```
df_columns = ['Model_Name','True_Positivie','True_Negative','False_Positive','False_Negative', 'Seconds']
df = pd.DataFrame(data=None, columns=df_columns)
for model_name in torchvision_model_list[:5]:
    arr = np.array([model_name,
                    torchvision_dog_test_result_dict_avg[model_name][0][0][0],
                    torchvision_dog_test_result_dict_avg[model_name][0][1][1],
                    torchvision_dog_test_result_dict_avg[model_name][0][0][1],
                    torchvision_dog_test_result_dict_avg[model_name][0][1][0],
                    torchvision_dog_test_result_dict_avg[model_name][1]])
    df = df.append(pd.DataFrame([arr], columns=df.columns), ignore_index=True)
df
```

	Model_Name	True_Positivie	True_Negative	False_Positive	False_Negative	Seconds
0	resnet18	0.0	100.0	0.0	100.0	3
1	alexnet	34.0	99.8	0.2	66.0	2
2	squeezenet1_0	39.8	93.4	6.6	60.2	2
3	vgg16	79.6	99.0	1.0	20.4	7
4	densenet161	0.0	100.0	0.0	100.0	14

```
plot_df_n_test_t_d_d(df)
```



5. Create a CNN to classify dog breeds (from scratch)

5.1. **Specify data loaders for the dog dataset**: For the train set, an image is randomly 30 degree rotated and randomly resized into 224 by 224 tensor. It is also randomly horizontally flipped. For validation set and test set, an image is resized into 255 by 255 tensor and centerly cropped into 224 by 224 tensor. Both input tensors are 224 by 224. Reason of choosing the input size is that it is a common size choice for pre-trained CNN models.

```
def load_data(data_dir, batch_size=[64, 16, 16]):
    """Load data and create sets and loaders for data.

    Argument(1):
        data_dir -- (str) directory path of data folders.

    Return(4):
        dir_sets -- (dic)
            Value: (str) directory path
            Keys : 'train', 'valid', 'test'
        compose_sets -- (dic)
            Value: (list) of transforms method(s)
            Keys : 'train', 'valid', 'test'
        image_datasets -- (dic)
            Value: ImageFolder method
            Keys : 'train', 'valid', 'test'
        dataloaders -- (dic)
            Value: DataLoader method
            Keys : 'train', 'valid', 'test'
    """
    # Define dir_set
    dir_sets = {}
    dir_sets['train'] = f'{data_dir}/train'
    dir_sets['valid'] = f'{data_dir}/valid'
    dir_sets['test'] = f'{data_dir}/test'
    # Define compose_set
    compose_sets = {}
    compose_sets['train'] = [transforms.RandomRotation(30),
                             transforms.RandomResizedCrop(224),
                             transforms.RandomHorizontalFlip(),
                             transforms.ToTensor(),
                             transforms.Normalize(mean = [0.485, 0.456, 0.406],
                                                  std = [0.229, 0.224, 0.225])]
    compose_sets['valid'] = [transforms.Resize(255),
                             transforms.CenterCrop(224),
                             transforms.ToTensor(),
                             transforms.Normalize(mean = [0.485, 0.456, 0.406],
                                                  std = [0.229, 0.224, 0.225])]
    compose_sets['test'] = [transforms.Resize(255),
                            transforms.CenterCrop(224),
                            transforms.ToTensor(),
                            transforms.Normalize(mean = [0.485, 0.456, 0.406],
                                                 std = [0.229, 0.224, 0.225])]
    # Define image_datasets
    image_datasets = {}
    image_datasets['train'] = datasets.ImageFolder(dir_sets['train'],
                                                   transforms.Compose(compose_sets['train']))
    image_datasets['valid'] = datasets.ImageFolder(dir_sets['valid'],
                                                   transforms.Compose(compose_sets['valid']))
    image_datasets['test'] = datasets.ImageFolder(dir_sets['test'],
                                                  transforms.Compose(compose_sets['test']))
    # Define dataloaders
    dataloaders = {}
    dataloaders['train'] = torch.utils.data.DataLoader(image_datasets['train'],
                                                       batch_size = batch_size[0], shuffle = True)
    dataloaders['valid'] = torch.utils.data.DataLoader(image_datasets['valid'],
                                                       batch_size = batch_size[1], shuffle = True)
    dataloaders['test'] = torch.utils.data.DataLoader(image_datasets['test'],
                                                      batch_size = batch_size[2], shuffle = True)

    return dir_sets, compose_sets, image_datasets, dataloaders
```

```
dir_sets, compose_sets, image_datasets, dataloaders = load_data('/data/dog_images')
```

```
loaders_scratch = dataloaders
```

5.2. **Implement Model Architecture**: The initial input of the convolutional 2d layer is ‘3’ which is each channel of RGB. LeakyReLU with 0.2 angle negative slope is used for the activation function for each convolutional layer. MaxPool2d(2,2) is used for dimension reduction in half. After 4 sequential layer sets of Conv2d, LeakyReLU, and MaxPool2d, AvgPool2d(14) is used to apply a 2D average

pooling over an input signal composed of several input planes, and a method, ‘view’, is used to allow a tensor to be a view of an existing tensor. Then, a classifier gets 128 inputs which is equal to the output number of AvgPool2d, and returns 133 outputs which is equal to the number of dog labels.

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

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

        self.conv = nn.Sequential(nn.Conv2d(3, 16, 3, padding=1), nn.LeakyReLU(0.2),
                                  nn.MaxPool2d(2, 2),
                                  nn.Conv2d(16, 32, 3, padding=1), nn.LeakyReLU(0.2),
                                  nn.MaxPool2d(2, 2),
                                  nn.Conv2d(32, 64, 3, padding=1), nn.LeakyReLU(0.2),
                                  nn.MaxPool2d(2, 2),
                                  nn.Conv2d(64, 128, 3, padding=1), nn.LeakyReLU(0.2),
                                  nn.MaxPool2d(2, 2))

        self.avg_pool = nn.AvgPool2d(14)
        self.classifier = nn.Linear(128, 133)

    def forward(self, x):
        features = self.conv(x)
        x = self.avg_pool(features)
        x = x.view(features.size(0), -1)
        x = self.classifier(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()
```

5.3. **Implement a loss function and an optimizer**.

```
import torch.optim as optim

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

if use_cuda:
    criterion_scratch.cuda()

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

5.4. **Implement a model training function with validation.** Then train the model and save and use the model with the best validation accuracy.

```
def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path, load_pt = True, valid_loss_min = np.Inf):
    """returns trained model"""
    if load_pt & os.path.exists(save_path):
        print('Loading previous model...')
        model.load_state_dict(torch.load(save_path))

    print('Start training...')

    # initialize tracker for minimum validation loss
    # valid_loss_min = np.Inf

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

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

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

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

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

        ## TODO: save the model if validation loss has decreased
        if valid_loss <= valid_loss_min:
            print(f'\nSaving model ...')
            torch.save(model.state_dict(), save_path)
            valid_loss_min = valid_loss
            print(f'\nModel has been saved. Minimum vaild loss has been updated.')
    # return trained model
    return model
```

```
# train the model
model_scratch = train(10, loaders_scratch, model_scratch, optimizer_scratch,
                      criterion_scratch, use_cuda, 'model_scratch.pt')

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

```
Loading previous model...
Start training...
Epoch: 1        Training Loss: 4.224779         Validation Loss: 4.106660

Saving model ...

Model has been saved. Minimum vaild loss has been updated.
Epoch: 2        Training Loss: 4.138141         Validation Loss: 3.910244

Saving model ...

Model has been saved. Minimum vaild loss has been updated.
Epoch: 3        Training Loss: 4.180687         Validation Loss: 3.994664
Epoch: 4        Training Loss: 4.060096         Validation Loss: 4.038754
Epoch: 5        Training Loss: 4.059994         Validation Loss: 3.909028

Saving model ...

Model has been saved. Minimum vaild loss has been updated.
Epoch: 6        Training Loss: 4.024179         Validation Loss: 4.016210
Epoch: 7        Training Loss: 3.968403         Validation Loss: 3.732280

Saving model ...

Model has been saved. Minimum vaild loss has been updated.
Epoch: 8        Training Loss: 3.911215         Validation Loss: 3.826729
Epoch: 9        Training Loss: 3.879438         Validation Loss: 3.758792
Epoch: 10       Training Loss: 3.832906         Validation Loss: 3.700521

Saving model ...

Model has been saved. Minimum vaild loss has been updated.
```

5.5. **Test the model**: Keep training until the accuracy of the test result gets at least above 10%. If the accuracy is below 60%, the model will not be used.

```
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: 3.686383


Test Accuracy: 13% (114/836)
```

6. Create a CNN to classify dog breeds (using transfer learning)

6.1. **Specify Data Loaders for the Dog Dataset**: Also delete previous data loaders to avoid memory issues.

```
if 'loaders_transfer' in globals():
    del loaders_transfer

torch.cuda.empty_cache()

## TODO: Specify data loaders

# Decrease batch size for memory
dir_sets, compose_sets, image_datasets, loaders_transfer = load_data('/data/dog_images', [32,16,16])
```

6.2. **Model Architecture**: In the comparison of torchvision models, VGG16 fits best for the dataset so it is used as the base model. After turning off the gradient option of the loaded base model, the last linear layer of the classifier is newly defined. Its output parameter is the only difference. This is to maintain the original performance as much as possible by minimizing the deformation from the original. It is changed from 1000 to 133, which is equal to the number of dog labels.

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

if 'model_transfer' in globals():
    del model_transfer

torch.cuda.empty_cache()

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

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

model_transfer.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=1000, bias=True)
)
```

```
output_size = 133

model_transfer.classifier[-1] = nn.Linear(4096, output_size, bias=True)

# model_transfer.classifier[0] = nn.Linear(25088, 2048, bias=True)
# model_transfer.classifier[3] = nn.Linear(2048, 1024, bias=True)
# model_transfer.classifier[6] = nn.Linear(1024, output_size, bias=True)

if use_cuda:
    model_transfer = model_transfer.cuda()

model_transfer.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)
)
```

6.3. **Implement a loss function and an optimizer**: Same loss function and optimizer is used. This is to maintain the original performance as much as possible by minimizing the deformation from the original.

```
import torch.optim as optim

if 'criterion_transfer' in globals():
    del criterion_transfer

if 'optimizer_transfer' in globals():
    del optimizer_transfer

torch.cuda.empty_cache()

criterion_transfer = nn.CrossEntropyLoss()

# Remaining only x.requires_grad is true.
optimizer_transfer = optim.Adam(filter(lambda x: x.requires_grad, model_transfer.parameters()),
                                lr=0.005)
```

6.4. **Implement a model training function with validation**. Then train the model and save and use the model with the best validation accuracy.

```
n_epochs = 10

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

# prev_training_loss = 3.2
# load_model_pt = True
# model_transfer = train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer,
#                        criterion_transfer, use_cuda, 'model_transfer.pt', load_model_pt, prev_training_loss)
```

```
Start training...
Epoch: 1        Training Loss: 3.239886         Validation Loss: 1.340621

Saving model ...

Model has been saved. Minimum vaild loss has been updated.
Epoch: 2        Training Loss: 3.273698         Validation Loss: 1.517020
Epoch: 3        Training Loss: 3.468552         Validation Loss: 1.507740
Epoch: 4        Training Loss: 3.568731         Validation Loss: 1.701078
Epoch: 5        Training Loss: 3.691881         Validation Loss: 1.793137
Epoch: 6        Training Loss: 4.104842         Validation Loss: 1.509692
Epoch: 7        Training Loss: 3.953236         Validation Loss: 1.689692
Epoch: 8        Training Loss: 3.975469         Validation Loss: 1.806905
Epoch: 9        Training Loss: 4.237475         Validation Loss: 1.630043
Epoch: 10       Training Loss: 4.404488         Validation Loss: 2.076963
```

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

6.5. **Test the model**: Keep training until the accuracy of the test result gets at least above 10%. If the accuracy is below 60%, the model will not be used.

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

```
Test Loss: 1.391676


Test Accuracy: 74% (622/836)
```

6.6. **Implement a dog breed predictor with the best accuracy model**.

```
### 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 image_datasets['train'].classes]

def predict_breed_transfer(img_path):
    # load the image and return the predicted breed
    img = Image.open(img_path).convert('RGB')

    compose_set = [transforms.Resize(255),
                   transforms.CenterCrop(224),
                   transforms.ToTensor(),
                   transforms.Normalize(mean = [0.485, 0.456, 0.406],
                                        std = [0.229, 0.224, 0.225])]

    transform = transforms.Compose(compose_set)

    img_tensor = transform(img).unsqueeze(0)

    if use_cuda:
        img_tensor = img_tensor.cuda()

    # Apply model
    predict = model_transfer(img_tensor)

    # Tensor.cpu() to copy the tensor to host memory
    if torch.cuda.is_available():
        predict = predict.cpu()

    # Argmax to get index of the maximum value(among predicted classes)
    index = predict.data.numpy().argmax()

    return class_names[index]
```

7. **Define an app function:** It will display input file name and graphic. Then it will display a prediction output result. There are 3 cases for the result, which is human-face-case, dog-case and neither-case.

```
def run_app(img_path):
    ## handle cases for a human face, dog, and neither
    print('----------------------------------------')
    print(f'{img_path}')

    plt.imshow(Image.open(img_path))
    plt.show()
    is_human_face = face_detector(img_path)
    is_dog = dog_detector(img_path)
    pred_breed = predict_breed_transfer(img_path)

    # Case(1): Human face
    if is_human_face and (not is_dog):
        print('^-^ It looks like a human face.')
        print(f'The face is similar to {pred_breed}.')

    # Case(2): Dog
    elif is_dog and (not is_human_face):
        print('^-^ It looks like a dog.')
        print(f'The breed looks like {pred_breed}.')

    # Case(3): Neither
    else:
        if is_human_face and is_dog:
            print(f'Opss ...\nIt looks like a dog or a human face similar to {pred_breed}.')

        else:
            print(f"Opss ...\nIt doesn't looks like neighter a dog nor a human face, but it's similar to {pred_breed}.")
```

Refinement

Minimizing the deformation from the original results in the best performance. The minimization is changing only the output size of the last linear layer of the classifier. The better performance can be expected through more learning. But the difference in performance between the minimization case and the others is definitely too big. One example for ‘the others’ is changing 3 linear layers of the classifier. The total learning time is above 20 minutes, which is 5 times longer than the minimization case. But the accuracy of the example is only 1% by comparison with 74% accuracy of the minimization case.

```
model_transfer.classifier[0] = nn.Linear(25088, 8192, bias=True)
model_transfer.classifier[3] = nn.Linear(8192, 4096, bias=True)
model_transfer.classifier[6] = nn.Linear(4096, output_size, bias=True)

if use_cuda:
    model_transfer = model_transfer.cuda()

model_transfer.classifier
```

```
Sequential(
  (0): Linear(in_features=25088, out_features=8192, bias=True)
  (1): ReLU(inplace)
  (2): Dropout(p=0.5)
  (3): Linear(in_features=8192, out_features=4096, bias=True)
  (4): ReLU(inplace)
  (5): Dropout(p=0.5)
  (6): Linear(in_features=4096, out_features=133, bias=True)
)

Start training...

Start Time: 17:00:31
Epoch: 1        Training Loss: 11.484295        Validation Loss: 4.883412

Saving model ...

Model has been saved. Minimum vaild loss has been updated.
Seconds Elapsed:143.53500819206238
Epoch: 2        Training Loss: 6.139649         Validation Loss: 4.873174

Saving model ...

Model has been saved. Minimum vaild loss has been updated.
Seconds Elapsed:144.08326911926270
Epoch: 3        Training Loss: 5.372767         Validation Loss: 4.873069

Saving model ...

Model has been saved. Minimum vaild loss has been updated.
Seconds Elapsed:144.45049381256104
Epoch: 4        Training Loss: 5.823307         Validation Loss: 4.880303
Seconds Elapsed:142.40441513061523
Epoch: 5        Training Loss: 5.618485         Validation Loss: 4.868700

Saving model ...

Model has been saved. Minimum vaild loss has been updated.
Seconds Elapsed:144.70687484741211
Epoch: 6        Training Loss: 6.115527         Validation Loss: 4.871411
Seconds Elapsed:142.25667595863342
Epoch: 7        Training Loss: 5.796935         Validation Loss: 4.867333

Saving model ...

Model has been saved. Minimum vaild loss has been updated.
Seconds Elapsed:144.2760763168335
Epoch: 8        Training Loss: 5.166148         Validation Loss: 4.867727
Seconds Elapsed:142.51736760139465
Epoch: 9        Training Loss: 5.495259         Validation Loss: 4.868310
Seconds Elapsed:142.6917426586151
Epoch: 10       Training Loss: 5.390154         Validation Loss: 4.868669
Seconds Elapsed:142.74094915390015
Log file created: training_log_08221700.csv

End Time: 17:24:25
```

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

```
Test Loss: 4.853214


Test Accuracy:  1% (10/836)
```

IV. Result

Model Evaluation and Validation

1. **Human-face-image identifier must have 70% or above true inference and vice versa.**

```
face_detector_test(human_files_short)
```

```
Start face detector testing human_files_short...

Num. of files: 100

Num. of face detected: 98

Detected percent: 98.0%

End testing human_files_short. ^-^
0.98
```

```
face_detector_test(dog_files_short)
```

```
Start face detector testing dog_files_short...

Num. of files: 100

Num. of face detected: 17

Detected percent: 17.0%

End testing dog_files_short. ^-^
0.17
```

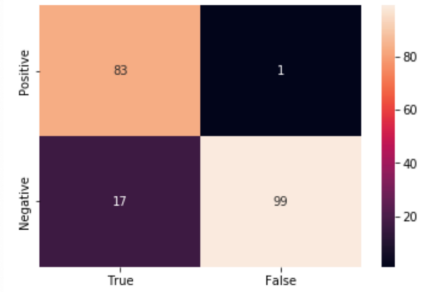
   - The percentage of human face images inferred as a human face:

        - 98%, which is above 70% and fits the criterion.
    - The percentage of dog images inferred as not a human face:
        - 17%, which is below 30% and fits the criterion.

2. **Dog-image identifier must have 70% or above true inference and vice versa.**

```
dog_detector_test_heatmap(dog_files_short, human_files_short)
```

```
Start dog detector testing ...
Num. of positive files: 100
Num. of negative files: 100
Dog detector test finished.
```



    - The percentage of dog images inferred as a dog:
        - 83%, which is above 70% and fits the criterion.
    - The percentage of human face images inferred as not a dog:
        - 17%, which is below 30% and fits the criterion.

3. **Dog-breed classifier must have 60% or above accuracy.**

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

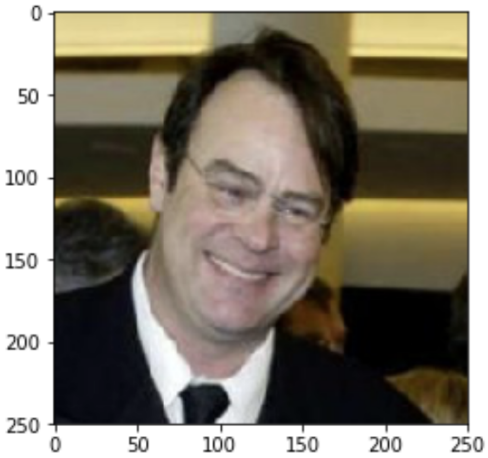
```
Test Loss: 1.391676


Test Accuracy: 74% (622/836)
```

    - The accuracy is 74%, which is above 60% and fits the criterion.

4. **Human-face-image Resemble-dog classifier**: 6 true result from 6 tests

- **True**: Case 2. Human

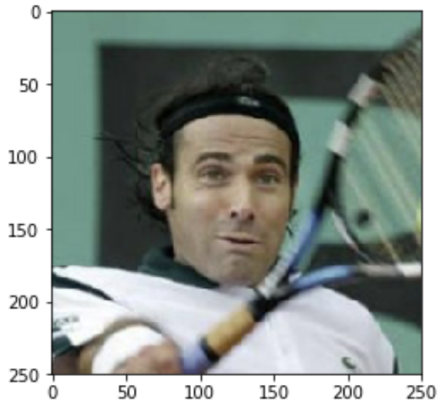
```
---------------------------------------
/data/lfw/Dan_Ackroyd/Dan_Ackroyd_0001.jpg
```




```
^-^ It looks like a human face.
The face is similar to Basenji.
```

- **True**: Case 2. Human

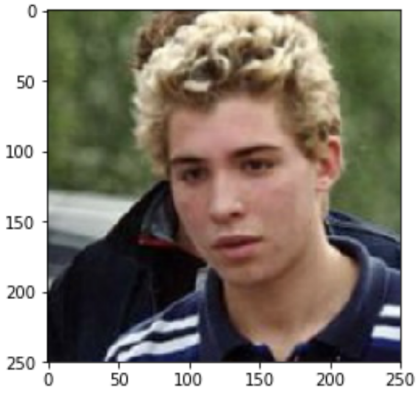
```
---------------------------------------
/data/lfw/Alex_Corretja/Alex_Corretja_0001.jpg
```



```
^-^ It looks like a human face.
The face is similar to Lowchen.
```

- **True**: Case 2. Human

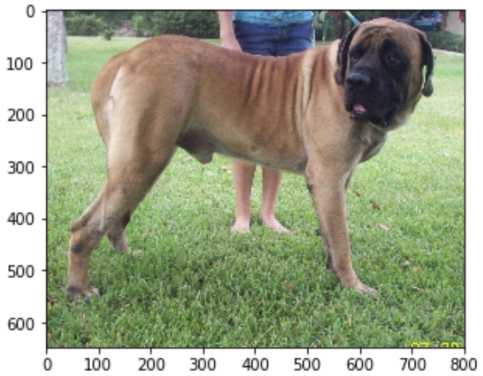
```
---------------------------------------
/data/lfw/Daniele_Bergamin/Daniele_Bergamin_000
1.jpg
```



```
^-^ It looks like a human face.
The face is similar to Bearded collie.
```

- **True**: Case 1. dog

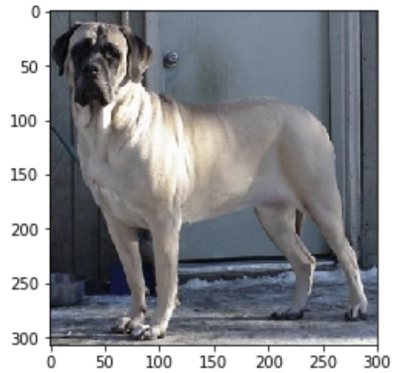
```
---------------------------------------
/data/dog_images/train/103.Mastiff/Mastiff_0683
3.jpg
```



```
^-^ It looks like a dog.
The breed looks like Mastiff.
```

- **True**: Case 1. dog

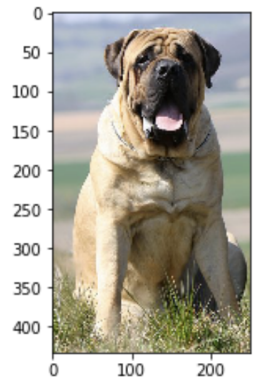
```
---------------------------------------
/data/dog_images/train/103.Mastiff/Mastiff_0682
6.jpg
```




```
^-^ It looks like a dog.
The breed looks like Bullmastiff.
```

- **True**: Case 1. dog

```
---------------------------------------
/data/dog_images/train/103.Mastiff/Mastiff_0687
1.jpg
```




```
^-^ It looks like a dog.
The breed looks like Bullmastiff.
```

Justification

All test results meet each criterion. But it is still possible to get better prediction results through the options below.

1. Feed more new training images.
2. Adjust learning rate and train for more epoches.
3. Variate the transformation of training images.
4. Try various pre-trained models and optimize hyper parameters appropriately.

IV. Reference

Reference

[1] Dog Breed Identification, Kaggle - Kaggle: https://www.kaggle.com/c/dog-breed-identification
[2] Dog Project, Udacity - Github:
https://github.com/udacity/dog-project/blob/master/dog_app.ipynb
[3] Dog image dataset, Dog App - Udacity:
https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/dogImages.zip
[4] Human image dataset, Dog App - Udacity:
https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/lfw.zip
[5] VGG16 – Convolutional Network for Classification and Detection - Neurohive:
https://neurohive.io/en/popular-networks/vgg16
[6] Cascade Classifier, OpenCV - OpenCV:
https://docs.opencv.org/3.4/db/d28/tutorial_cascade_classifier.html
[7] torchvision.models - Pytorch: https://pytorch.org/vision/stable/models.html
[8] haarcascades, OpenCV - Github:
https://github.com/opencv/opencv/tree/master/data/haarcascades
[9] imagenet 1000 class idx to human readable labels, yrevar, Gist - Github:
https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a
[10] torchvision.models, Torchvision - PyTorch: https://pytorch.org/vision/stable/models.html