UDACITY Machine Learning Engineer Nanodegree

Capstone Project: Dog Breed Classifier using CNN

Chee Chung Lee

September 5th, 2020

1. Definition

Project Overview

The project was part of Udacity's Machine Learning nanodegree and is popular also in deep learning nanodegree.

The aim of this project is to create an algorithm that could be part of a web application that is able to identify dog breeds if given an image as input. If the image contains a human face, then the algorithm will return the breed of dog that most resembles this person. If the image contains a dog, then the algorithm will return the breed of the dog. If the image contains neither, it will return an error to try again.

Our concern is to identify dog breeds from any image regardless if it contains human or dog. The data set imported from Udacity provided in the workspace consists of 8351 dog images in 133 different categories. The project has been successfully executed in Udacity workspace (GPU-enabled) with PyTorch and several convolution neural network (CNN) models. Besides predicting dog breeds with this technology, this machine learning algorithm can be applied to identifying other objects, animals, plants, behaviours and expression of people, diagnostics of equipment or even cancer detection. Modern CNN used in computer vision is founded by Yann LeCun which he proposed early form of back propagation learning algorithm in 1987.

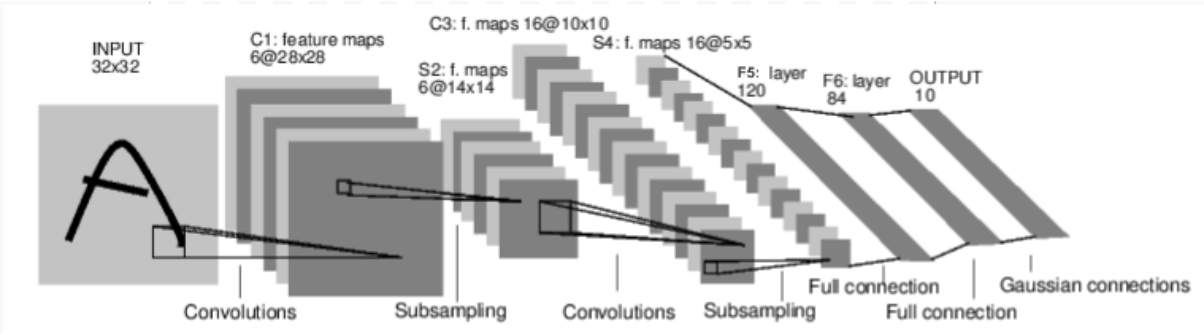


Figure 1: How convolution neural network takes input and produce output

Problem Statement

The aim of this project is to create an algorithm that is able to identify a dog breed if given an image as input. For this problem to be solved, the data requires to be processed and classified with a high accuracy of above 70%. The algorithm has 3 main tasks to be performed and solved:

1. If the images supplied to the application has a human, then the application will return the breed of dog that most resembles the person in the image.

2. If the images supplied to the application has a dog, then the application will return the breed of dog.

3. If the image supplied to the application is neither a human or a dog, the application will return an error.

For us to solve these 3 tasks, Convolutional Neural Network with Transfer learning to classify the dog breeds is used. The benefits of Transfer Learning is that it can speed up the time it takes to develop and train a model by reusing the modules of already developed models. The feature extraction part of the model is reused whereas the classification part is re-trained. Since the feature extraction process is the most complex modelling challenge, reusing it allows you to train a new model with less computational resources and training time.

First a Convolution Neural Network from scratch (without using Transfer Learning) is built and the accuracy obtained was nearly 6% which is reasonable because of simple architecture implementation. By using transfer learning approach the accuracy increased to ~80%. This is because image recognition requires more complex feature detection and VGG16 possess these and able to achieve extremely high accuracy.

Metrics

In order for us to solve the problems for the project, performance metrics need to be present to verify that the models we use are working well and the models we produce will give us a correct prediction. The first performance metrics we will use is accuracy. Accuracy is defined with the calculation as the following:

Accuracy=Number of correctly classified items/ All classified items

Accuracy is used when evaluating the performance of the model prediction for dog detector, human detector and the testing phase for CNN dog breed classification model. The human and dog detectors were tested on 100 images of each.

The human and dog detectors were tested on 100 images of each. The human face detector relied on OpenCV's Haar Feature-based Cascade Classifiers and was trained on 13233 human faces which has been imported from sklearn. Whereas the dog detector was built on a pre-trained VGG16 model with ImageNet of 1000 classes. Its purpose was to confirm if an image was a dog or not. The accuracy of these two detectors were evaluated and their prediction were highly accurate at 96% for human detector and 96% for dog detector.

The other performance metrics we look at is the log loss which is used during the training and validation phase when building and developing the CNN dog breed classification model. Log Loss of the training and validation set can be evaluated after each epoch cycles to check if the model is improving and not overfitting. We do this because the problem we try to solve is a classification problem and the data set provided is unbalanced with uneven number of samples in different classes.

This log loss takes into account the uncertainty of the prediction based on how much it varies between actual label and the prediction. Thus, when working with log loss, the sample image would have probabilities to match with all the different classes. Log loss is defined using the PyTorch nn.CrossentropyLoss with the calculation of the following:

$$\text{loss}(x, \text{class}) = -\log\left(\frac{\exp(x[\text{class}])}{\sum_j \exp(x[j])}\right) = -x[\text{class}] + \log\left(\sum_j \exp(x[j])\right)$$

Loss is calculated based on the summation of all the difference between the probability of the classes by the prediction and the hot labels of one 1 and others being 0 through the cross entropy function. Training loss and validation loss is evaluated to be at a reducing trend after each epoch cycle.

2. Analysis

Data Exploration

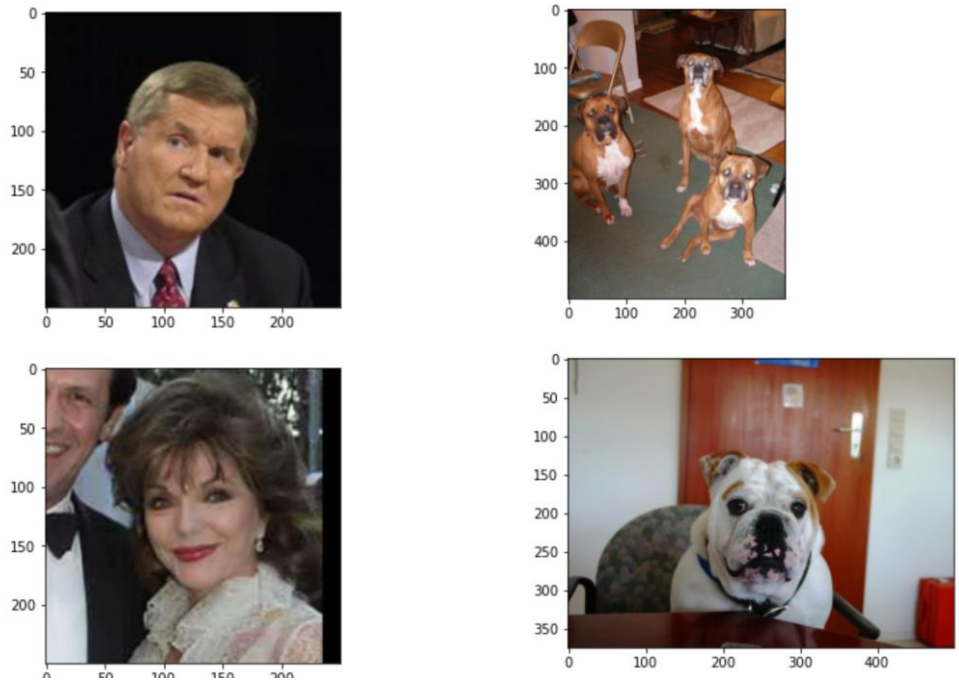
Two datasets were downloaded i.e. Dog Dataset and Human Dataset are used which are provided by Udacity. Dog dataset contains images of 133 classes of dogs in 133 folders each. All the images are not of the same dimensions, orientation of object of the dog or person are not the same. Number of dog or people in the image are sometimes more than 1. The links and insights to both the datasets are given below:

Human Dataset:

There are in total 13233 human images. Example of the images of the human dataset can be viewed in figure below:

```
#View the sample human photos.

for img in random.sample(list(human_files), 2):
#open and display input image
    image = Image.open(img) #img_path
    plt.imshow(image)
    plt.show()
```

```
#View the sample dog photos.
for img in random.sample(list(dog_files), 2):
#open and display input image
    image = Image.open(img) #img_path
    plt.imshow(image)
    plt.show()
```



Dog Dataset:

There are 133 total dog categories.

There are 8351 total dog images.

There are 6680 training dog images.

There are 835 validation dog images.

There are 836 test dog images.

Based on the sample above it can be seen that images for dog images are not of the same size. The number of dogs in the image can be more than 1 as well.

As for human images, they shape are the same. They are consistently the same for every image. They can have more than one human face in the image as well. There is no blurred image as running monochrome filter, it can be seen that the number of images which has a monochrome>0.5 is only 85 which is ~1% of the dog images has 50% of the image being a single colour. I would say the images are high quality and ideal for training.

In [71]:
```
def filtered_images(image_path):
    mono_images=[]
    non_mono_images=[]
    for filename in image_path:
        try:
            img=Image.open(filename)
            #check pixel distribution
            v=img.histogram()
            h,w=img.size
            percentage_monochrome=max(v)/float(h*w)
            # filter bad and small images
            if percentage_monochrome>0.5 :#or h<300 or w<300:
                mono_images.append(filename)
            else:
                non_mono_images.append(filename)
        except:
            pass
    print("Number of monochrome images: {}\n".format(len(mono_images)))
    print("Number of non-monochrome images: {}\n".format(len(non_mono_images)))
    return mono_images, non_mono_images
```

In [72]:
```
filtered_images(dog_files)
```

```
Number of monochrome images: 85

Number of non-monochrome images: 8265
```

Exploratory Visualization

Dog images dataset have 8351 images which are separated in 3 file directories which are train (6680 images), test (836 images) and valid (835 images). Each of these 3 file directories have 133 folders which corresponds to the 133 different classes of dog breeds.

Bar plots to show frequencies of human images in each class of human and dog. The histogram with all the different class shows that most of the classes are in the 4 sample space. Standard deviation, of dog and human classes are different. The spread of image frequencies are different.

Human images dataset have 5,750 folders sorted by the name of the human and contain 13,233 total images. Our data is not balanced because we have one image of some people and several for others. The same is for dog images. (the difference is from 1 to 9 images in most cases)

Dog images have different image sizes, different colour, different backgrounds, different angle and posture and orientation of the dog, some dogs are in full sizes and some just a head. The variation is good so that we can train the model to identify a breed regardless of these factors and minimise overfitting.

Human images are all of the same size 250×250. Images are with different backgrounds, light, from different angles, sometimes with few faces on the image. Human classification of the 5,749 folders was not performed, thus not a huge concern as long as a human can be identified in this project.

Algorithms and Techniques

The use of a filter to a 2D array of input that produce an output is a convolution. The repeated use of the same filter which is smaller than the 2D array of inputs by moving the filter across the whole 2D array produces a map of outputs called feature map. This map indicates the locations and strength of a detected feature in the 2D array input, such as an image. What this means is the result is highly specific features that can be detected anywhere on input images.

Multiple use of convolution to create multiple feature map can be performed in parallel to create a convolution layer. These convolutional layers are the major building blocks used in convolutional neural network. Each output of each convolution of the layer will be linked to be an input for all the convolution for the next convolutional layer. The links between inputs and output for multiple convolution in multiple layers is a Deep Learning algorithm called the convolution neural network.

The convolution neural network learn and improve its prediction by automatically adjusting the filter bias and weight of the filters in parallel through the use of training dataset which has a set input and actual outcome. The neural network knows how much to learn and improve through back propogation of the loss between the actual and predicted outcome. With this loss, the neural network adjusts the weights and biases accordingly as it learns what type of features to extract through every training cycle.

Our input data are a bunch of dog and human images and our output required is a multiclass dog breed classification. Images are highly complex as it possess a lot of parameters and features to determine its classification. Besides producing feature matrix, CNN is able to carryout dimensionality reduction which suits the huge number of parameters in an image which needs to be reduced.

Thus, to solve this problem, we need to perform multiclass classification utilising CNN.

For a CNN to work affectively there are numerous parameter/technique in CNN architecture to improve the learning of the algorithm. They are:

- Dropouts. Based on dropout ratio it will randomly deactivate some of the connection between convolution layers for each feed to reduce overfitting of the algorithm.
- Learning rates is altered to ensure the best prediction can be obtained within the number of epochs
- Epochs are just the number of times the data is used to train the algorithm. Overtraining a model with too high a Epoch number can result in model being overfitting and validation loss will increase instead of reduce.
- Batch size are also critical. Smaller batch size allows the model to be trained more times and faster compared to higher batch size. However the downside of batch size being too small is the gradient estimate for the training of the model will be less accurate.
- Optimizer is used for back propagation. The ones tested are Adam and Stochastic gradient descent. Adam has a gradual reduction in learning rate to ensure the minimum loss can be achieved through learning. Stochastic gradient descent learns faster, but it has higher chance it does not achieve the best model prediction accuracy as the minimum loss is not achieved.

- Transfer learning is a technique by using pretrained algorithms which are highly accurate for CNN. It is incorporated by altering the output layers of the pretrained model.

Benchmark

Benchmark model will be the CNN built from scratch which had approximately 10% accuracy rate. 10% is accuracy signifies that the model built is better than a random guess because if it was a guess it would have been 1 out of 133 (breeds).

The CNN model created using transfer learning must have an accuracy level of 60% or above to show that the model is working well and trained well for the problem at hand.

3. Methodology

Data Preprocessing

Before any analysis can be performed, data pre-processing needs to take place to ensure the data can be fed correctly to the model in the correct format.

```
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()
```

```
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()
```

Implementation

To solve this problem, we would deploy a multiclass classification utilising convolution neural network. A convolutional neural network (CNN) is a Deep Learning algorithm which can take in an

input image, assign importance through learnable weights and biases to various patterns/objects in the image. The distinct patterns in the images are able to be differentiated from each other using the algorithm.

In this section, the workflow for approaching a solution given the problem is as follow.

Step 1: Detect Humans

Explore different face detectors from OpenCV to be used. By creating a face detector we would be able to tell if a human is present in the image. To do this we will implement Haar feature-based cascade classifiers. The workflow required to detect faces are:

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

-initialize pre-trained face detector

```
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.
# run the function face_detector to check number of True values.

#define first row
k=0
number_human_face=0
number_dog_face=0
#create loop to run all rows, since human and dog are using 100 images
for k in tqdm(range(len(human_files_short))):
    if face_detector(human_files_short[k]) == True:
        number_human_face+=1
    if face_detector(dog_files_short[k]) == True:
        number_dog_face+=1
    k+=1
percent_human=(number_human_face/len(human_files_short))*100
print('human faces ',percent_human, "%")
percent_dog=(number_dog_face/len(dog_files_short))*100
print('dog faces ',percent_dog, "%")
```

```
100%|
| 100/100 [00:28<00:00,  3.54it/s]
```

```
human faces  96.0 %
dog faces  18.0 %
```

- load image of humans imported
- convert image to grayscale
- test the human detector by loading human and dog images
- verify that the human detector can detect human faces accurately and cant detect dog in the image.

Step 2: Detect Dogs

We will use the pre-trained model VGG16 to detect dogs in images. This model has been pretrained for 1000 different classes through ImageNet.

- Define VGG16 model

```
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
    '''
    #open image file
    input_image = Image.open(img_path)
    #preprocess function defined which transform the image so that images are converted to tensors,
    #all consistent size and normalised accordingly
    preprocess = transforms.Compose([transforms.Resize(256),transforms.CenterCrop(224),transforms.ToTensor(),
                                     transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])])

    input_tensor = preprocess(input_image) #create input tensor by feeding input image with the preprocess function.
    input_batch = input_tensor.unsqueeze(0) # create a mini-batch as expected by the model

    with torch.no_grad(): #no back propogation altering the gradients. Keeping the model attributes the same.
        output = VGG16(input_batch) #Tensor of shape 1000, with confidence scores over Imagenet's 1000 classes
    # print(output)
    # The output has unnormalized scores. To get probabilities, you can run a softmax on it.
    # print(torch.nn.functional.softmax(output[0], dim=0))

    #extract the index with the maximum value in the second dimension which has 0:999 classes of dogs
    xx,predicted = output.max(dim=1)    #xx is the value, predicted is the class index

    #print('Pre-trained model predicts an index of', predicted.item())
    return predicted.item()
```

– Use GPU for better performance if available

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

#     #check if output index is between 151 and 268. If yes, True, False for No.

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

– load and pre-process the image

– send an image to the VGG16 model

– verify that the dog detector can detect dog in images accurantely and cant detect dog in the human images.

- Model return true if the index is 151 to 268 as the classes between 151 and 268 is for dogs.

```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
nbr_human=0
nbr_dog=0
i=0
for i in tqdm(range(len(human_files_short))):
    if dog_detector(human_files_short[i]):
        nbr_human+=1
    if dog_detector(dog_files_short[i]):
        nbr_dog+=1
    i+=1

percent_human=(nbr_human/len(human_files_short))*100
print('detect a dog in human_files',percent_human, "%")
percent_dog=(nbr_dog/len(dog_files_short))*100
print('detect a dog in dog_files',percent_dog, "%")
```

```
100%|
| 100/100 [01:46<00:00,  1.07s/it]
detect a dog in human_files 0.0 %
detect a dog in dog_files 95.0 %
```

Step 3: Create a benchmark CNN model to classify dog breeds (from scratch)

The dog data is already divided into training, validation and test data. The data in images needs to be converted to batch data in the form of tensors for the model. Training and validation is used concurrently to verify that the validation loss of the model created is actually reducing and improving and not just training loss getting reduced. Once the model is sufficiently trained through a few epochs, it's accuracy is being evaluated using the test data. If it doesnt hit 10% accuracy, the model architecture and training parameters can be fine tuned. Once this is achieved, it can considered as the benchmark model.

```
import os
from random import shuffle
from __future__ import print_function, division
import torch
from torch.utils.data import Dataset, DataLoader
from torchvision import datasets
from torchvision import transforms, utils

### TODO: Write data loaders for training, validation, and test sets
# define training, test and validation data directories, if on udacity: use /data/....
train_dir = 'dogImages/train'
valid_dir = 'dogImages/valid'
test_dir = 'dogImages/test'

standard_normalization = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                              std=[0.229, 0.224, 0.225])
#define transform criterias
train_transforms = transforms.Compose([transforms.Resize(size=256),
                                       transforms.RandomHorizontalFlip(),
                                       transforms.RandomRotation(10),
                                       transforms.CenterCrop(224),
                                       transforms.ToTensor(),
                                       standard_normalization])
validTest_transforms = transforms.Compose([transforms.Resize(size=256),
                                           transforms.CenterCrop(224),
                                           transforms.ToTensor(),
                                           standard_normalization])

#create the transformed images into tensors datasets
datatrain = datasets.ImageFolder(train_dir, train_transforms)
datavalid = datasets.ImageFolder(valid_dir, validTest_transforms)
datatest = datasets.ImageFolder(test_dir, validTest_transforms)

#prepare data loaders for each dataset which will be used to feed into the model for batch training, test and validation.
#training loader
train_loader = torch.utils.data.DataLoader(datatrain,batch_size=32,shuffle=True,num_workers=0)
#test loader
valid_loader = torch.utils.data.DataLoader(datavalid,batch_size=32,shuffle=True,num_workers=0)
# test loader
test_loader = torch.utils.data.DataLoader(datatest,batch_size=32,shuffle=True,num_workers=0)
```

Step 5: Create a CNN model using transfer learning using Resnet or VGG to clasify dog breeds. With transfer learning, we can maintain the model architect and just integrate it with the output layer classification we require. We train, validate and test the model just as in step 4. This time we expect the accuracy to be over 60%.

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

## TODO: Specify model architecture
#import the VGG16 model
model_transfer=models.vgg16(pretrained=True)
```

```
from collections import OrderedDict

# replace the fully connected layer classifier to match the number of dog classifications required.
classifier = nn.Sequential(OrderedDict([
                          ('fc1', nn.Linear(25088, 4096)),
                          ('relu', nn.ReLU()),
                          ('fc2', nn.Linear(4096, 133)),
                          ('output', nn.LogSoftmax(dim=1))
                          ]))

# ensures we don't update the weights from the pre-trained model.
for param in model_transfer.parameters():
    param.requires_grad = False

# Unfreeze training for fully connected (fc) layers
# for param in model_transfer.fc.parameters():
#     param.requires_grad = True

#map new classifier to the model
model_transfer.classifier=classifier

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

```
Out[96]: VGG(
           (features): Sequential(
             (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (1): ReLU(inplace=True)
             (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (3): ReLU(inplace=True)
             (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=True)
             (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (8): ReLU(inplace=True)
             (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=True)
             (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (13): ReLU(inplace=True)
             (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (15): ReLU(inplace=True)
             (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=True)
             (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (20): ReLU(inplace=True)
             (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (22): ReLU(inplace=True)
             (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=True)
             (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (27): ReLU(inplace=True)
             (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (29): ReLU(inplace=True)
             (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
           )
           (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
           (classifier): Sequential(
             (fc1): Linear(in_features=25088, out_features=4096, bias=True)
             (relu): ReLU()
             (fc2): Linear(in_features=4096, out_features=133, bias=True)
             (output): LogSoftmax()
           )
```

```
criterion_transfer = nn.CrossEntropyLoss()
optimizer_transfer = optim.Adam(model_transfer.classifier.parameters(), lr=0.0001)
```

```
# train the model
#def train(n_epochs, loaders1, loaders2, model, optimizer, criterion, use_cuda, save_path)
# train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer, criterion_transfer, use_cuda, 'model_transfer.pt')
model_transfer =  train(10,train_loader ,valid_loader ,
                        model_transfer, optimizer_transfer, criterion_transfer, use_cuda, 'model_transfer.pt')
```

```
Epoch: 3        Training Loss: 7.364786         Validation Loss: 23.654617

0it [00:00, ?it/s]

Validation loss has decreased (26.922888 ======> 23.654617). Saving model ....

209it [1:10:45, 20.31s/it]
27it [07:58, 17.71s/it]
0it [00:00, ?it/s]

Epoch: 4        Training Loss: 6.006021         Validation Loss: 26.807202

209it [3:58:54, 68.59s/it]
27it [04:02,  8.97s/it]
0it [00:00, ?it/s]

Epoch: 5        Training Loss: 3.992521         Validation Loss: 26.117661
```

```
In [128]: test(test_loader, model_transfer, criterion_transfer, use_cuda)

Test Loss: 0.971181


Test Accuracy: 78% (659/836)
```

Step 6: Create and test the algorithm.

Create the algorithm by combining the human detector, dog detector and the transfer learning dog breed classifier model. The response the algorithm should produce are the folowing:

- If the image input contain a human, confirm that human is present and what dog breed the human resembles.

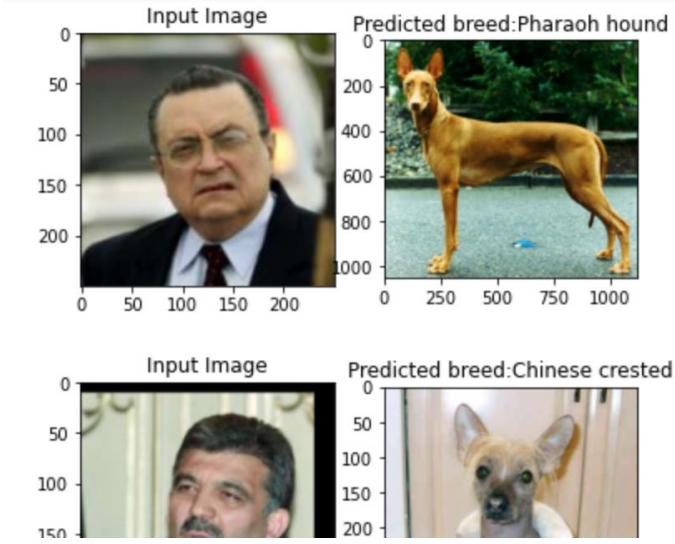
- If the image input contains a dog, confirm that a dog is present and what dog breed is the dog.

- If the input image contains neither, provide output which indicates error.

```python
### TODO: Write a function that takes a path to an image as input
### and returns the dog breed that is predicted by the model.
import glob
# 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 datatrain.classes]
#raw_class= [item[4:].replace("_", " ") for item in datatrain.classes]
def predict_breed_transfer(img_path):


    #open image file
    input_image = Image.open(img_path)
    #preprocess function defined which transform the image so that images are converted to tensors,
    #all consistent size and normalised accordingly
    preprocess = transforms.Compose([transforms.Resize(256),transforms.CenterCrop(224),transforms.ToTensor(),
                                     transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])])

    input_tensor = preprocess(input_image)
    input_batch = input_tensor.unsqueeze(0) # create a mini-batch as expected by the model

    with torch.no_grad(): #no back propogation altering the gradients. Keeping the model attributes the same.
        output = model_transfer(input_batch)

    #The max value in all outcome probability for all output channels is the predicted class.
    #extract the index with the maximum value in the second dimension which has 0:999 classes of dogs
    xx,predict = output.max(dim=1)

    return datatrain.classes[predict]
```

```python
#if dog or human detected
def display_image(img_path):
    my_list=[]
    #predictedimage=datatrain.classes(raw_class)
    raw_class=predict_breed_transfer(img_path)
    pred_dir=train_dir+"/"+raw_class+"/*.*"

    class_names =raw_class.replace("_"," ")[4:]

    #prepare random sample data from the predicted breed
    predpath=glob.glob(pred_dir)
    for file in random.sample(list(predpath),1):
      #  print(file)
        a=Image.open(file)
        my_list.append(a)

    #open and display input image
    image = Image.open(img_path) #img_path
    plt.subplot(1,2,1)
    plt.imshow(image)
    plt.title("Input Image")

    # display the random sample image from the predicted breed
    plt.subplot(1,2,2)
    plt.imshow(my_list[0])
    plt.title(label=f"Predicted breed:{class_names}")

    plt.show()
```

```
#to testthe function
import random

# Try out the function
for image in random.sample(list(human_files_short), 4):
 #   predicted_breed = predict_breed_transfer(image)

    display_image(image)
```



Refinement

The process of improving upon the algorithms and techniques used is clearly documented. Both the initial and final solutions are reported, along with intermediate solutions, if necessary.

The benchmark model which was built from scratch was fine-tuned with various parameters such as the following:

- Learning rate variation between 0.01 to 0.003
- Number of convolution layers
- Number of input and output each layers
- Number of linear output.
- Varied the dropout layers.
- Varied the type between SGD or Adam.

4. Results

Model Evaluation and Validation

The final model's qualities—such as parameters—are evaluated in detail. Some type of analysis is used to validate the robustness of the model's solution.

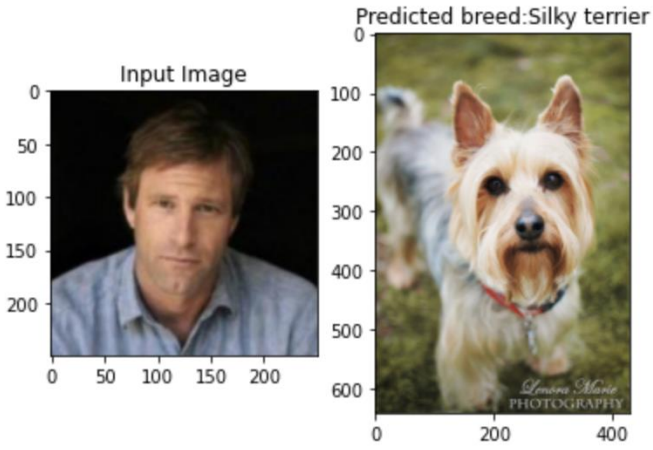
In your `Justification` section, please be sure to compare your models' results to the benchmark. How did the models perform and did they exceed the benchmark results for the project? How would you explain or justify that you've adequately solved the problem?

The transfer learning model produced using VGG16 was impressively accurate with an Adam.

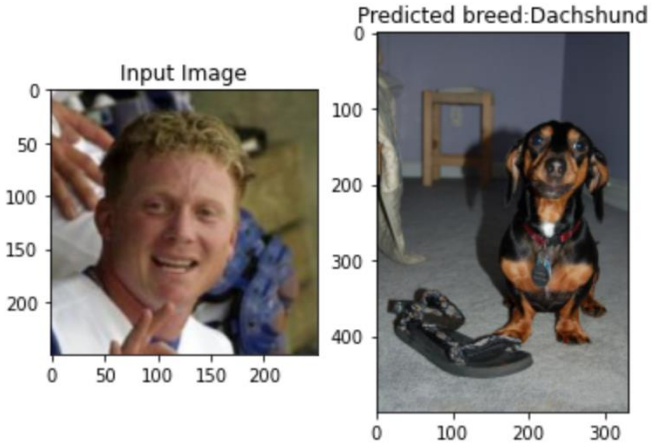
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.

def run_app(img_path):
    ## handle cases for a human face, dog, and neither
# if a human face is present, print( "human face present")
    if face_detector(img_path) == True:
        print(" human face present!....This human looks like a ....")
# and then predict what dog breed does the human face looks like
        display_image(img_path)

# check if a dog is present
# predict dog breed it look like
    elif dog_detector2(img_path) == True:
        print(" dog is present!...This dog looks like a...")
        display_image(img_path)

 #   we dont find a human or dog in the photo. please try again to reinput the image. No output prediction avaialble.
    else:
        print("cant find a human or a dog in the photo, please try again")
        image = Image.open(img_path) #img_path
        plt.title("Input Image")
        plt.imshow(image)
        plt.show()
```

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

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

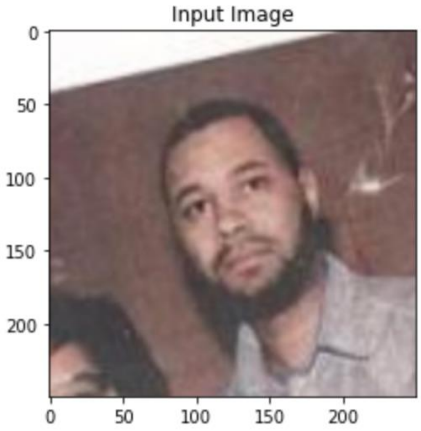
```
human face present!....This human looks like a ....
```



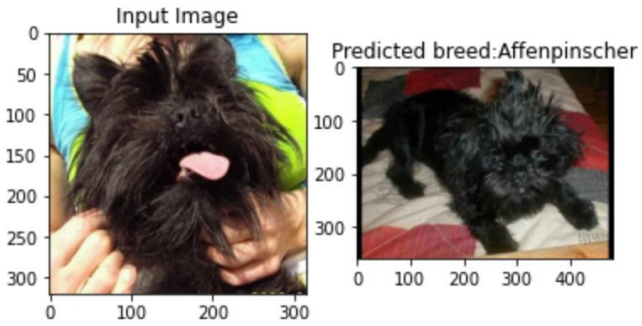

```
human face present!....This human looks like a ....
```



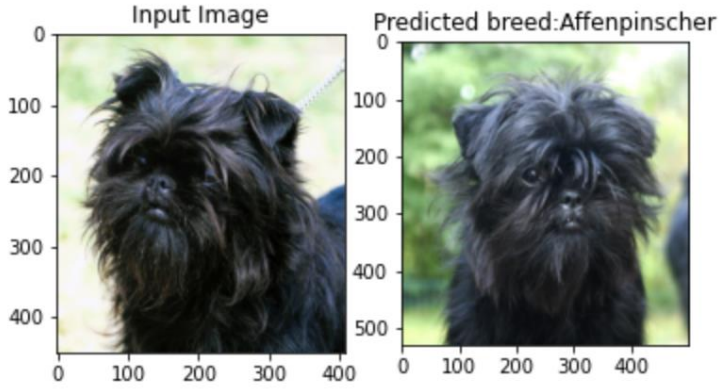
cant find a human or a dog in the photo, please try again



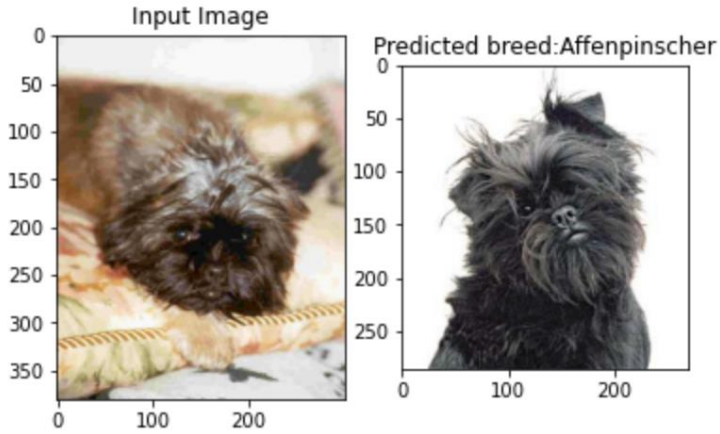
dog is present!...This dog looks like a...



```
dog is present!...This dog looks like a...
```




```
120.Pharaoh_hound
 dog is present!...This dog looks like a...
```



Justification

The final results are compared to the benchmark result or threshold with some type of statistical analysis. Justification is made as to whether the final model and solution is significant enough to have adequately solved the problem.

In your `Model Evaluation and Validation` section, please be sure to provide some discussion about the final parameters or characteristics of the model. How do these align with the characteristics of the dataset? Why would these be a robust solution to the problem?

Another approach that you could take would be to demonstrate that your tuned model is robust would be to perform a k-fold cross validation. In this case, you'd document how the model performs across each individual validation fold. If the validation performance is stable and doesn't fluctuate much, then you can argue that the model is robust against small perturbations in the training data.

References

https://en.wikipedia.org/wiki/Yann_LeCun