Project Report: An Algorithm to Classify Dog Breed

I. Defined Problem and Proposed Solution:

1. Problem

The problem for our project is an image classification problem: "*Can you detect whether it is a human or a dog? If it a dog, can you predict the dog's breed? If it is a human, can you return a resembling dog breed*?"

Classification is a type of supervised machine learning. In classification problem, we focus on identifying which category a new observation belongs, based on a previous training set of labeled data.

We have 3 subproblem: detect human face, detect dog in images and detect dog breed. All 3 subproblems are image classification problem.

2. Potential Solution

I will first create 3 models to detect human face, dog and dog breed. After that I will combine these 3 models into 1 function to solve our problem. The type of model will be CNN model since it works well with image recognition problem

3. Evaluation Metrics

Looking at the dog datasets, we can see that there are 133 dog breeds listed. Each breed data is split in train, test, valid with ratio 80:10:10. There are some slight imbalance in dataset: there are around 3-10 images in test and training set and around 30-90 images in training set. However, I think it is reasonable to choose accuracy score as evaluation metrics since the imbalance is not extreme.

For all models, the loss function will be cross entropy loss since it is default loss function for classification case.

II. Data Exploration

The dataset used for training are provided by Udacity. Data can be found in this repository. The data is in .jpg format

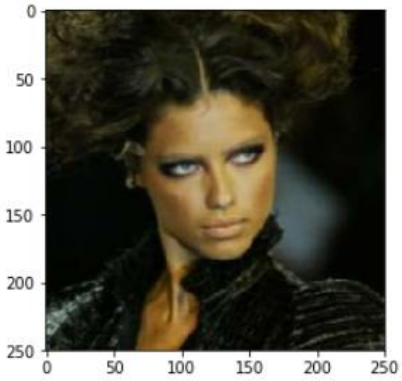
```
import numpy as np
from glob import glob

# load filenames for human and dog images
human_files = np.array(glob("/data/lfw/*/*"))
dog_files = np.array(glob("/data/dog_images/*/*/*"))

# print number of images in each dataset
print('There are %d total human images.' % len(human_files))
print('There are %d total dog images.' % len(dog_files))
```

```
There are 13233 total human images.
There are 8351 total dog images.
```

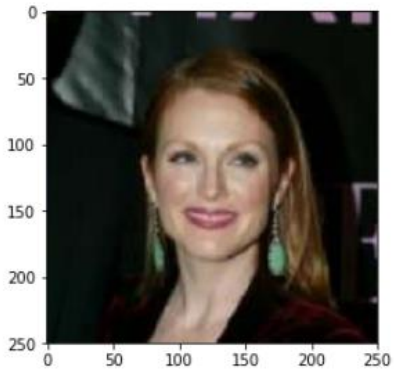
I have 2 datasets used for input, a dog dataset and human dataset. The dog images are categorized based on their breeds and human images are categorized based on names. There are 13233 total human images and 8351 total dog images. The human dataset will be used to train human detection model and the dog dataset will be used to train a dog detection model and dog breed classifier.

Let's look at the images in both dataset

/data/lfw/Adriana_Lima/Adriana_Lima_0001.jpg



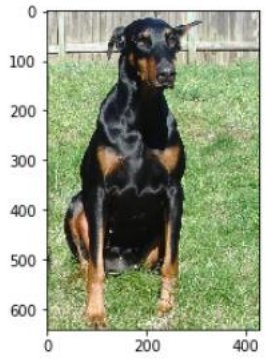
/data/lfw/Julianne_Moore/Julianne_Moore_0003.jpg



You can see that the file path contains the name of person. Pictures have 3 channels: red, green and blue

/data/dog_images/train/103.Mastiff/Mastiff_06846.jpg

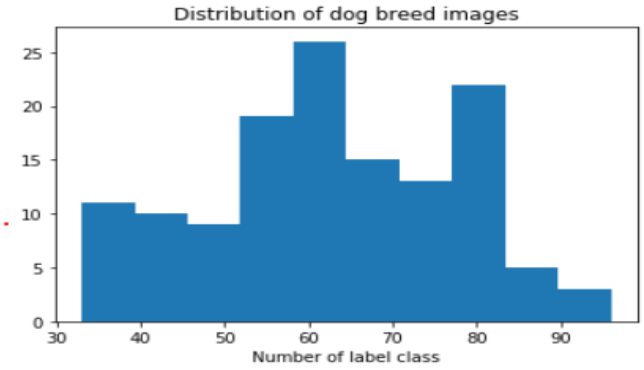
```
/data/dog_images/train/059.Doberman_pinscher/Doberman_pinscher_04199.jpg
```

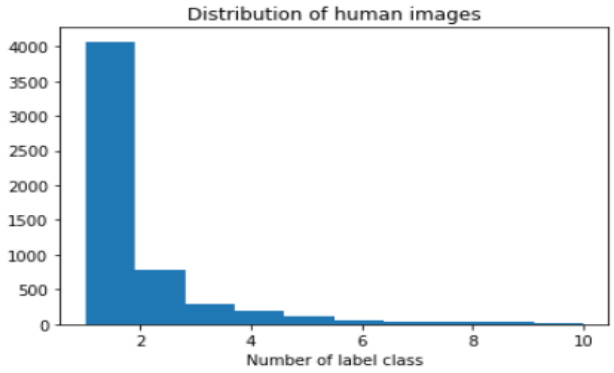


As you can see, the dogs are categorized into breed and split into train, test and validation set.

There are 133 dog breeds in the data file.

Let's see how dog breed images and human images are distributed in the data





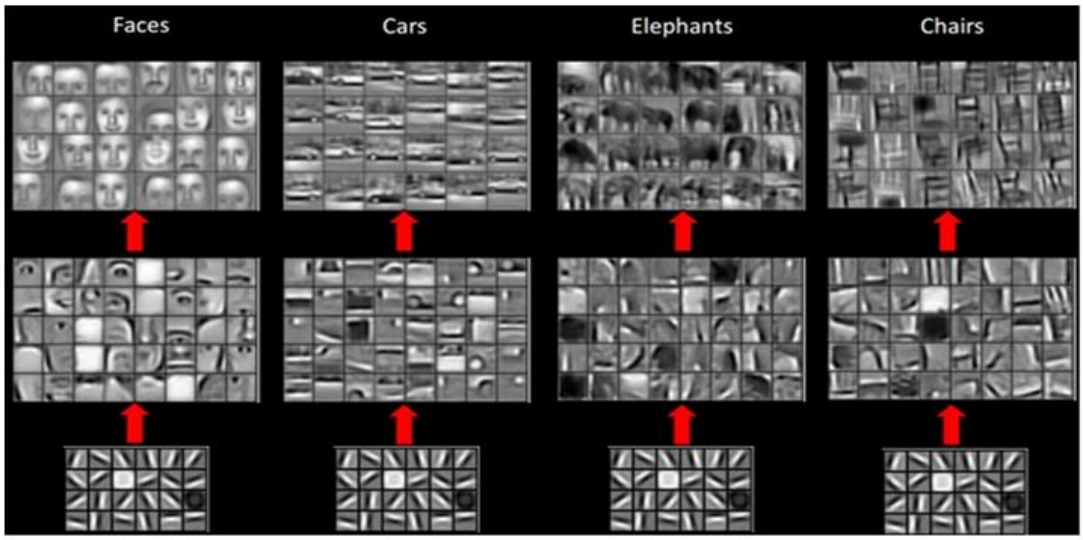
The number of images for each dog breed range from 30-90. This is not really necessary to detect dog but it might really beneficial for detecting dog breed The distribution of images of each human is skewed, dominantly 1 or 2. However, I don't think it's going to be a problem since we will only need to detect human through transfer model.

III. Implementation & Results

1. Algorithms and Techniques:

Before going into details of each model, I will give an overview of the algorithms and some ideas of how it works. Keep in mind, all the concepts below are introduced at a high level. Though I do love explaining it in mathematical ways, I think it is out of scope of this project and also make the report too complicated.

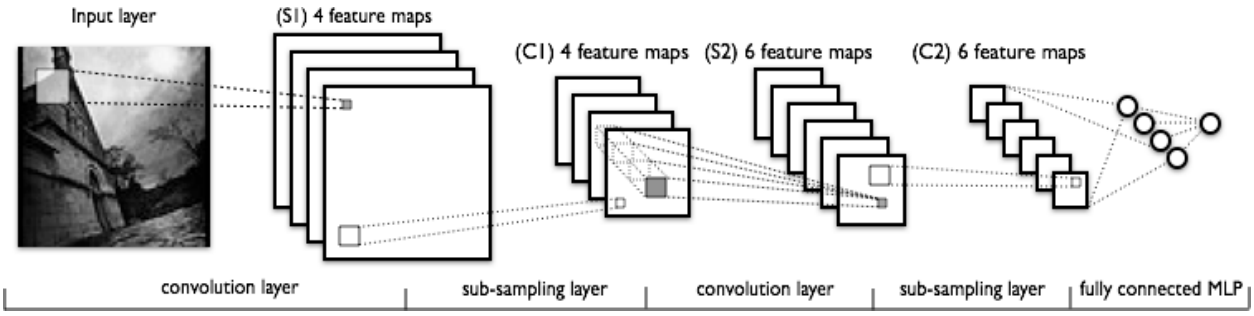
All models I used are **Convolution Neural Network-based** model or CNN. Convolutional Neural Network is a type of Deep Neural Network. It can be used detect and learn the features in the images. It proved to be extremely useful to classify image. An example of successful CNN model is the ResNet model which won the ImageNet Challenge in 2015.



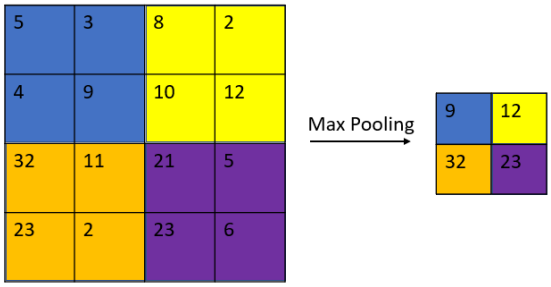
There are some terminologies I would like to discuss before going into details:

- **Convolutions** at a high-level provide a way to transform input image into feature map which makes it easier for the model to learn the features of the image.

- **Fully Connected Layers** is the layer that does discriminative learning in Neural Network. All neurons/nodes in one layer are connected to neurons/nodes in another layers. An illustration for this layer will be seen below with convolutions.



- **Pooling** is a way to pick the domain part of the feature map. It can be used to downsize the image which help improve the model performance and take out noisy data**.** Two common pooling layers are Max Pooling and Average Pooling. In our model, we will use Max Pooling. A simple demonstration of max pooling can be seen below:



There are some other techniques that I use in this model like dropout for regularization, padding to detect edge and maintain image size. However, in our case, the model can still work well without these techniques and parameters so I will not go into details

One more thing I would like to discuss is **Transfer Model.** In my project, I make use of several transfer model. Transfer model is basically a model which have been trained and can be used by other people. One technique I use in this project is **Fine-Tuning** which means I freeze the parameters as well as the layers in the transfer model and add some more layers to fit the output in our problems.

2. Procedures and Outputs

I use OpenCV's implementation of Haar feature-based cascade classifier to detect human face in images. The pretrain face detectors are stored as XML files on github . Expected output: True/False. True means that model detects human face.

I use pretrained model like VGG-16 to detect dogs in images. Expected output: True/False. True means that model detects a dog

I will use Resnet as the transfer learning model and modify the top layers with fully connected layers to get the dog-breed classifier. Expected output is the dog's breed

3. Human Detector Model

I first define the function to detect human face. In our case, we are using OpenCV, implantation of Haar feature-based cascade classifier to detect human in images. I have downloaded one of these detectors and stored it in *haarcascades* directory

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

```
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
```

Let test to see how good our model with the first 100 images from dog and human dataset

```
from tqdm import tqdm

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

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

## TODO: Test the performance of the face_detector algorithm
## on the images in human_files_short and dog_files_short.
human_human = np.sum([face_detector(img) for img in human_files_short])
human_dog = np.sum([face_detector(img) for img in dog_files_short])
# Note: there is 100 pics so the sum is equalto percentage of accuracy)
print('Percentage of the 100 images in human_files have a detected human: {}%'.format(human_human))
print('Percentage of the 100 images in dog_files have a detected human: {}%'.format(human_dog))
```

```
Percentage of the 100 images in human_files have a detected human: 98%
Percentage of the 100 images in dog_files have a detected human: 17%
```

Though not perfect, the model seems to do well to detect human face. Ideally, we would 0% of dog images with detected human face. However, the model's performance is acceptable to use

4. Create a model to detect dogs

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

Let's define the VGG16 model for our use cases

```
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()
```

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

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

    Args:
        img_path: path to an image

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

    ## TODO: Complete the function.
    ## Load and pre-process an image from the given img_path
    ## Return the *index* of the predicted class for that image
    img = Image.open(img_path).convert('RGB')
    transform = transforms.Compose([transforms.Resize(256),
                                    transforms.CenterCrop(224),
                                    transforms.ToTensor()])
    img_tensor = transform(img).unsqueeze(0)
    if use_cuda:
        img_tensor = img_tensor.to('cuda')

    prediction = VGG16(img_tensor)
    if use_cuda:
        prediction = prediction.cpu()

    index = prediction.data.numpy().argmax()

    return index # predicted class index
```

Then, we will create a dog detector function

```python
### 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)
    return (151 <= index and index <= 268) # true/false
```

We will test the performance with the first 100 images from dog and human dataset

```python
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
dog_human =  np.sum([dog_detector(img) for img in human_files_short])
dog_dog = np.sum([dog_detector(img) for img in dog_files_short])
# Note: there is 100 pics so the sum is equalto percentage of accuracy)
print('Percentage of the 100 images in human_files have a detected dog: {}%'.format(dog_human))
print('Percentage of the 100 images in dog_files have a detected dog: {}%'.format(dog_dog))
```

```
Percentage of the 100 images in human_files have a detected dog: 0%
Percentage of the 100 images in dog_files have a detected dog: 89%
```

It's nice that no human is detected as a dog and we can detect dog in dog images for about roughly 90% which is not bad. Thus, I decide to go with this model

5. Create model to detect dog breed

First, I transform image into tensor, and apply some transformation to augment data in train dat a and resize and center crop the test and validation data.

```
import os
from torchvision import datasets
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True
### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes
data_dir = '/data/dog_images'
train = 'train'
valid = 'valid'
test = 'test'
# transform
train_transform = transforms.Compose([transforms.RandomResizedCrop(224),
                                      transforms.RandomHorizontalFlip(),
                                      transforms.ToTensor()])
val_test_transform = transforms.Compose([transforms.Resize(256),
                                       transforms.CenterCrop(224),
                                       transforms.ToTensor()])
data_transform = {train : train_transform,
            valid: val_test_transform,
            test: val_test_transform}
data = {x: datasets.ImageFolder(os.path.join(data_dir, x),
                                transform = data_transform[x])
        for x in [train, valid, test]}

batch_size = 8
num_worker = 0

loaders_scratch = {x: torch.utils.data.DataLoader(data[x],
                                                  batch_size = batch_size,
                                                  shuffle = True, num_workers
              for x in [train, valid, test] }
```

5.a. Build a benchmark model

I created a dog breed classifier from scratch with CNN and the result was terrible.

I define my CNN model as below

```
class Net(nn.Module):
    ### TODO: choose an architecture, and complete the class
    def __init__(self):
        super(Net, self).__init__()
        ## Define layers of a CNN
        # convolutional layers
        self.conv1 = nn.Conv2d(3,32,3, stride = 2, padding = 1)
        self.conv2 = nn.Conv2d(32,64,3, stride = 2, padding = 1)
        self.conv3 = nn.Conv2d(64,128,3, padding = 1)
        # pooling layers
        self.pool = nn.MaxPool2d(2,2)
        # fully-connected layers
        self.fc1 = nn.Linear(7*7*128, 500)
        self.fc2 = nn.Linear(500, num_class)
        # dropout
        self.dropout = nn.Dropout(0.3)

    def forward(self, x):
        ## Define forward behavior
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        # flatten
        x = x.view(-1, 7*7*128)
        x = self.dropout(x)
        x = F.relu(self.fc1(x))

        x = self.dropout(x)
        x = self.fc2(x)
        return x
```

The model architecture is composed of 3 CNN layers and 2 full-connected layers. Each CNN l ayer has a relu activation function and a 2D Max Pooling to reduce the computation and numb er of parameters. Strides will also be used to downsize the input image by 2 in the first 2 CNN layers. A Dropout layer will be applied before each fully-connected layer to prevent overfitting . The 2nd fully-connected layer will provide the final output to predict the breed types.

The validation loss after 25 epochs is 3.34

```
Epoch: 20       Training Loss: 3.437577         Validation Loss: 3.446480
Validation loss has decreased from 3.465461 to 3.446480. Saving Model
Epoch: 21       Training Loss: 3.408978         Validation Loss: 3.400985
Validation loss has decreased from 3.446480 to 3.400985. Saving Model
Epoch: 22       Training Loss: 3.414275         Validation Loss: 3.426483
Epoch: 23       Training Loss: 3.362924         Validation Loss: 3.358168
Validation loss has decreased from 3.400985 to 3.358168. Saving Model
Epoch: 24       Training Loss: 3.387101         Validation Loss: 3.381338
Epoch: 25       Training Loss: 3.351896         Validation Loss: 3.349552
Validation loss has decreased from 3.358168 to 3.349552. Saving Model
```

I applied the model to test set but the result is terrible.

```
Test Loss: 3.391314


Test Accuracy: 18% (155/836)
```

5.b. Final Model using transfer model with some modification on top layers

It seems that the best idea to tackle this problem is to use a transfer model and then modify the top layer to fit our cases. I used a ResNet152 model for our cases and it proved to be efficient

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

## TODO: Specify model architecture
model_transfer = models.resnet152(pretrained = True)

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

model_transfer.fc = nn.Linear(2048, 133)

for param in model_transfer.fc.parameters():
    param.requires_grad= True

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

The model transfer is simple with 1 fully connected layer on top of Resnet152. I train this mod el with just 10 epochs and validation loss seems to be much better than the previous choice. Th e validation loss is 0.9

```
Epoch: 1        Training Loss: 1.372687         Validation Loss: 1.364725
Validation loss has decreased from inf to 1.364725. Saving Model
Epoch: 2        Training Loss: 1.245174         Validation Loss: 1.233666
Validation loss has decreased from 1.364725 to 1.233666. Saving Model
Epoch: 3        Training Loss: 1.128170         Validation Loss: 1.120522
Validation loss has decreased from 1.233666 to 1.120522. Saving Model
Epoch: 4        Training Loss: 1.090664         Validation Loss: 1.081767
Validation loss has decreased from 1.120522 to 1.081767. Saving Model
Epoch: 5        Training Loss: 1.039306         Validation Loss: 1.033855
Validation loss has decreased from 1.081767 to 1.033855. Saving Model
Epoch: 6        Training Loss: 1.001942         Validation Loss: 0.996572
Validation loss has decreased from 1.033855 to 0.996572. Saving Model
Epoch: 7        Training Loss: 0.977961         Validation Loss: 0.970603
Validation loss has decreased from 0.996572 to 0.970603. Saving Model
Epoch: 8        Training Loss: 0.964995         Validation Loss: 0.960203
Validation loss has decreased from 0.970603 to 0.960203. Saving Model
Epoch: 9        Training Loss: 0.923204         Validation Loss: 0.915937
Validation loss has decreased from 0.960203 to 0.915937. Saving Model
Epoch: 10       Training Loss: 0.909716         Validation Loss: 0.902235
Validation loss has decreased from 0.915937 to 0.902235. Saving Model
```

I immediately try it with our test set to see the loss result and it is much better. Thus, I decide t o use this model to classify dog breed.

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

```
Test Loss: 0.388892


Test Accuracy: 89% (749/836)
```

5.c. Compare results

Compared to the result from **benchmark model** I built from section 3.a, we can see that the tra nsfer model works extremely well (The accuracy increase from 18% to 89% and cross entropy loss reduce from 3.4 to 0.39). Thus, we will pick this model as our final model for dog-breed cl assifier

6. Combine 3 models to solve our case

With all 3 models above, we can finally combine these 3 models to solve our problem.

We will create a function to do the following things

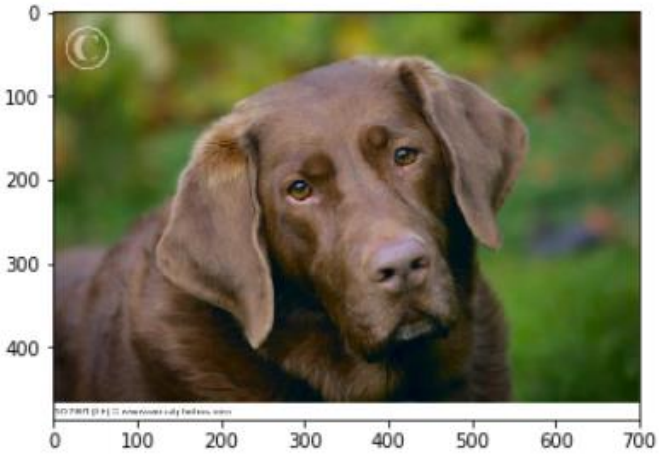
The function shoudl accept a file path to an image and first determines whether the image contains a human, dog, or neither. Then,

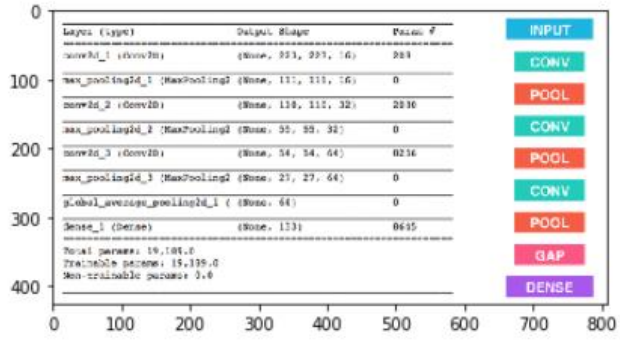
- if a **dog** is detected in the image, return the predicted breed.
- if a **human** is detected in the image, return the resembling dog breed.
- if **neither** is detected in the image, provide output that indicates an error.

```
def run_app(img_path):
    ## handle cases for a human face, dog, and neither
    img = Image.open(img_path).convert('RGB')
    plt.imshow(img)
    plt.show()
    breed = predict_breed_transfer(img_path)
    if face_detector(img_path):
        print('Detect a human')
        print('This human resemble {} dog breed'.format(breed))
    elif dog_detector(img_path):
        print('Detect a dog')
        print('This dog looks like {}'.format(breed))
    else:
        print('It is neither human or dog')
```

My function will be test on the images provided by Udacity in the same repository. Some of the results are below

```
for img in os.listdir('./images'):
    img_path = os.path.join('./images', img)
    run_app(img_path)
```






```
It is neither human or dog
```

IV. Conclusion:

I do see that there are possible improvements in my algorithm:

- I can definitely apply some hyperparameter tunings for my models
- More data augmentation in the training data can improve the performance significantly
- We can see that in our dog breed classification model, the validation loss is still decreasing so we can increase number of epochs to improve performance

Overall, the implementation seems to work well and adequately solves the problem I state with high accuracy

I can definitely expand this problem into detecting multiple dog and human faces in one image