CNN Project: Dog Breed Classifier
Project Report

Lingchen Zhu

July 23, 2020

1 Project Overview

The goal of this project is to learn how to build a pipeline to process real-world, user-supplied images. Particularly, This project is to classify the breed of dog using Convolutional Neural Network (CNN). Given an image of a dog, your algorithm will identify an estimate of the canine's breed. If supplied an image of a human, the code will identify the resembling dog breed. In order to train a CNN that can successfully classify dog breed, sophisticated CNN model architectures may be used, and transfer learning from some well-known CNN model architectures pre-trained on natural image dataset may also be applied.

2 Problem Statement

.

Describing the content of an image is a typical computer vision problem. In this project, we will build a two level image classifier. The first level, which is a rough classification object, is to tell whether an input image describes a human face or a dog. The second level, which is more detailed, is to tell which exactly the dog breed is if the image has a dog, or to tell which dog breed the human face mostly resembles if there is one in the image.

OpenCV provides human face detector but it does not provide any "dog detector". That's the reason we will need to build one with CNN in this project. In this project, we will first use OpenCV to detect human face using Haar cascades. Then, in order to classify dog breed, we will define our own CNN model architecture from scratch and use a well-known CNN model architecture, e.g., VGG16, ResNet50, etc., pre-trained on the ImageNet dataset and update its top (head) layers for transfer learning.

3 Metrics

Similar to other classification problems, the CrossEntropyLoss function provided by PyTorch will be used to evaluate the train, validation and test losses of the CNN model. We can also use a plain accuracy metric, i.e., the total number of occurrences when the predicted dog breed class label is equivalent to the reference dog breed class label divided by the total number of images, to evaluate the performance of the trained CNN model.

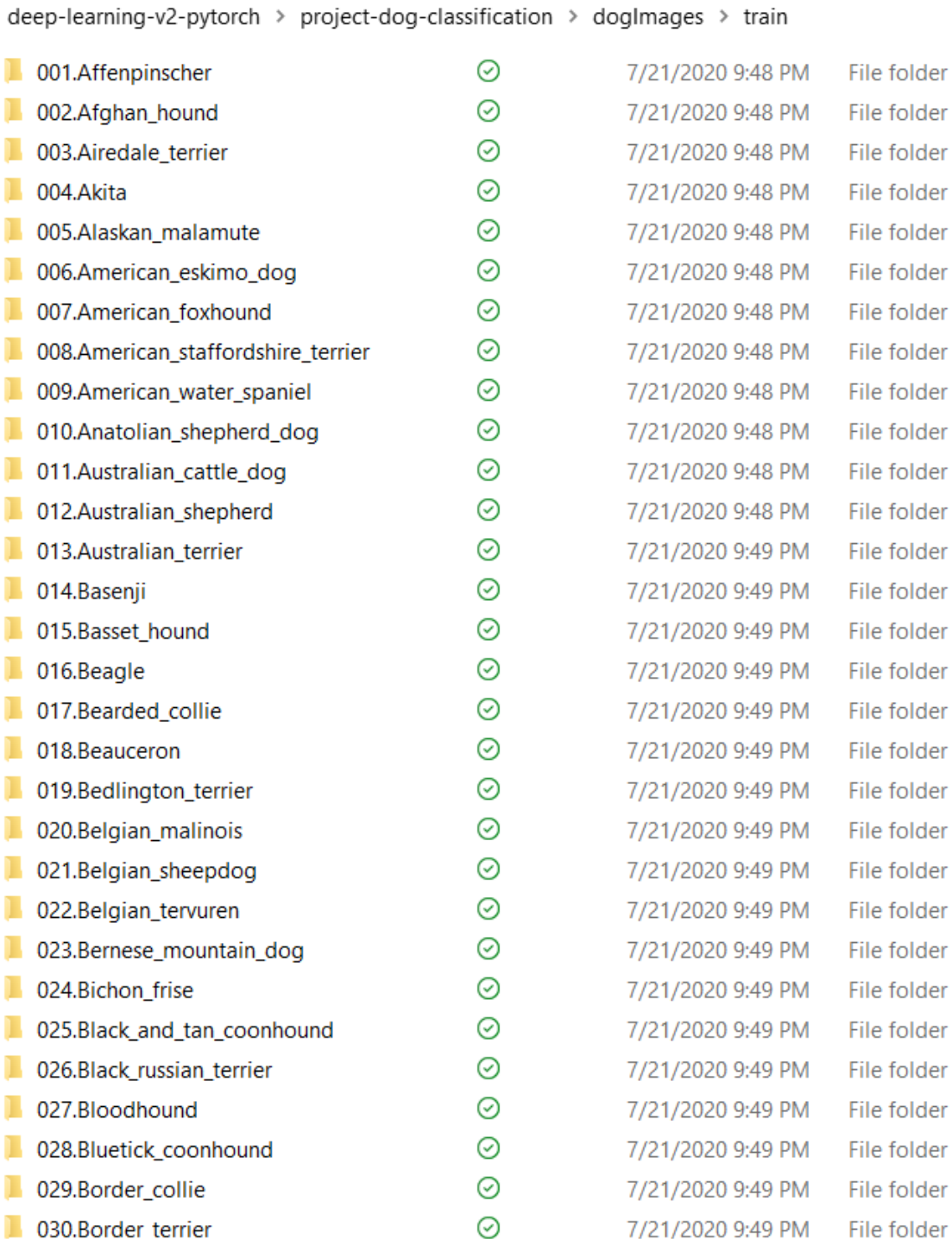


Figure 1: Folders of training images

4 Data Exploration

The dog dataset is provided by Udacity and can be downloaded from here. The human face dataset is also provided by Udacity and can be downloaded from here. Since we will train the CNN model on the dog dataset, we will focus on more it. After unzipping the dog dataset, we can see that the images are well organized in three subfolders: train, valid, and test. Under each of these three sub-folders, we see 133 sub-sub-folders and each one is named by a breed class index along with the its class name. Fig. 1 shows an example of the breed sub-sub-folders in the train sub-folder of the dog dataset.

Each sub-sub-folder contains many images for train, validation, or test. Fig. 2 gives a sub-sub-folder under the train sub-folder that includes many images for golden retrievers for CNN training purposes.

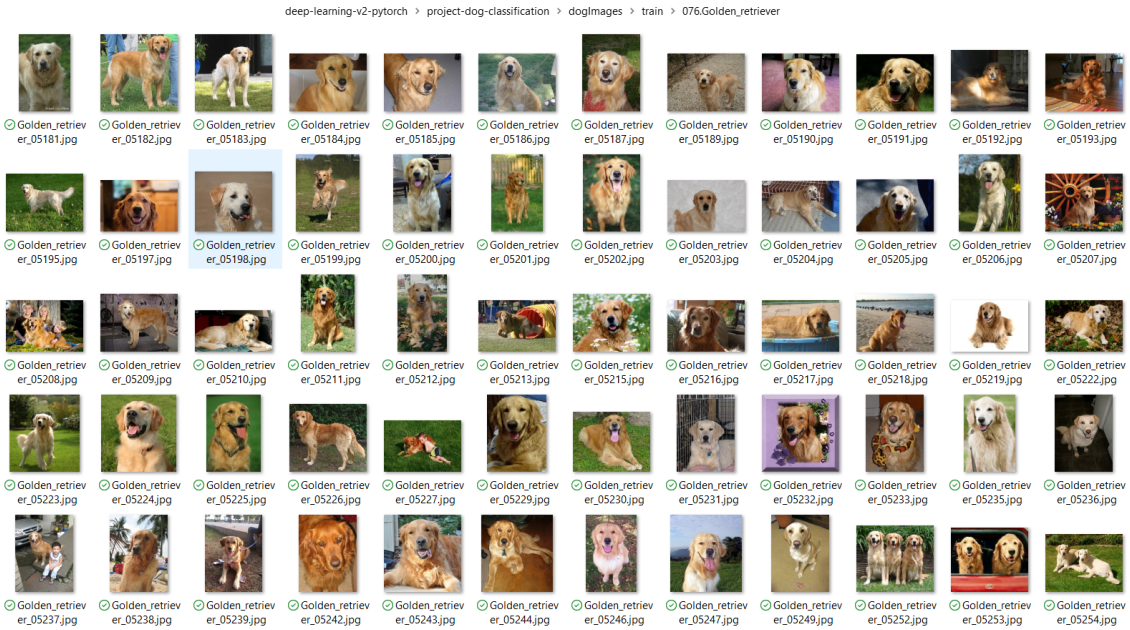


Figure 2: Images of golden retrievers for training

5 Algorithms and Techniques

First, in order to roughly classify whether a human face is contained in an input image, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images. It only accepts grayscale images as input so we need to convert all images to be detected from RGB to grayscale. OpenCV functions can help to do that. After the face detection is finished, we can also draw bounding boxes on the detected human face(s) to highlight them.

Then, in order to roughly classify whether a dog is contained in an input image, we can directly use a out-of-box CNN model, such as VGG16, ResNet50, etc., pre-trained on the ImageNet dataset

who has 1000 different classes including 118 different kinds of dogs. We just simply infer a class from the input image from such pre-trained CNN models and verify if the output class index is between 151 and 268 (inclusive). Note that the pre-trained CNN models only accepts image of size 224×224 with 3 color channels (RGB), meaning that the input tensor shape for each sample should be $3 \times 224 \times 224$ as PyTorch always places channel dimension before size dimensions ("channels_first").

However, in order to detect all possible 133 different kinds of dog breeds, the pre-trained CNN models may not be enough as there are only 118 different kinds of dogs. The number of classes do not match! To solve this problem, we need to design our own CNN models.

The first natural idea is to design a CNN starting from scratch. Fig. 3 shows a summary of my neural network My deep neural network consists of a few combinations of Conv2D layers,

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 16, 220, 220]           1,216
       BatchNorm2d-2         [-1, 16, 220, 220]              32
         MaxPool2d-3         [-1, 16, 110, 110]               0
            Conv2d-4         [-1, 32, 106, 106]          12,832
       BatchNorm2d-5         [-1, 32, 106, 106]              64
         MaxPool2d-6           [-1, 32, 53, 53]               0
            Conv2d-7           [-1, 64, 51, 51]          18,496
         MaxPool2d-8           [-1, 64, 25, 25]               0
            Conv2d-9          [-1, 128, 23, 23]          73,856
        MaxPool2d-10          [-1, 128, 11, 11]               0
           Conv2d-11            [-1, 256, 9, 9]         295,168
        MaxPool2d-12            [-1, 256, 4, 4]               0
           Linear-13                  [-1, 512]         131,584
          Dropout-14                  [-1, 512]               0
           Linear-15                  [-1, 133]          68,229
================================================================
Total params: 601,477
Trainable params: 601,477
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 21.87
Params size (MB): 2.29
Estimated Total Size (MB): 24.74
----------------------------------------------------------------
```

Figure 3: My CNN model from scratch

BatchNorm2d layers, ReLU activation functions, and MaxPooling2D layers for multiscale feature extraction, and then a global average pooling operation to get the averages of all filtered feature maps, whereafter the values are 1D tensors. Then I add two fully-connected layers with a dropout layer between them that prevents overfitting. The first fully-connected layer is a hidden layer with the ReLU activation function followed by the dropout layer with probability of an element to be zeroed as 0.5 and the second one is the output layer that provides the probability results, if a softmax function follows, of dog breed classification.

After training for 20 epochs, the accuracy of such model reaches 17% (150/836), i.e., the breeds of 150 out of 836 testing images are correctly classified. It has reached the expected goal that the test accuracy is greater than 10%.

Still, 17% accuracy is a good starting point but not the end of the game. We would like to further improve the classification accuracy on the test dataset. Transfer learning can greatly empower our dog breed classifier with the pre-trained feature extractors learned for other targets (like ImageNet classification).

I used ResNet50 model as the base model for transfer learning. I freezed the bottom layers and replaced its top (head) as a one-layer fully-connected network. The ResNet50 model pre-trained on ImageNet is good enough to classify 1000 different image classes in the ImageNet dataset. That means its bottom layers are good enough to extract image feature maps from different kinds of natural images, and hence I freezed all of them. Since we are working on a more specific image dataset, i.e., a dataset of dogs with 133 different breeds (classes), we need to fine-tune its top (head) layer(s) only to adapt with such change in the number of classes.

After training for another 20 epochs on such transfer learning model, I noticed that the train and validation losses are much lower than the previous CNN model started from scratch. That implies such a transfer learning model can achieve much better classification accuracy on the test dataset. After performing the test task, we see a 86% (719/836) accuracy – much better than the previous CNN model started from scratch. Of course, it is also much better than the expected 60% accuracy on the test set as defined for the project.

6 Data Preprocessing

Since the input images can be of different shapes of RGB channel values between 0 and 255, we need to normalize them for our CNN models.

For the regular CNN model from scratch, we only need to resize them to the same "channel_first" shape.

All pre-trained CNN models expect input images normalized in the same way, i.e. mini-batches of 3-channel RGB images of shape $(3 \times H \times W)$, where H and W are expected to be at least 224. The images have to be loaded in to a range of [0, 1] and then normalized using mean = [0.485, 0.456, 0.406] and std = [0.229, 0.224, 0.225].

As the last step, since PyTorch models can only accept torch.tensor data type as inputs, all the preprocessed images along with their training labels need to be converted to tensors. Fig. 4 shows the implementation codes I used to build custom PyTorch datasets. We can inspect the properties of these datasets, and double check that

Number of training samples: 6680,

Number of validation samples: 835,

Number of testing samples: 836,

Number of dog classes: 133.

In order to load batches of images (as well as their labels, if needed) from the built PyTorch datasets, I also need to build PyTorch dataloaders, which can be used as iterators in the training process. Fig. 5 shows the implementation codes I used to build the corresponding dataloaders from previously built datasets.

```python
import os
from sklearn.datasets import load_files
# from torchvision import datasets  # datasets.ImageFolder can also be used to generate PyTorch Dataset
from torch.utils.data import Dataset, DataLoader

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

class DogDataset(Dataset):
    def __init__(self, root_dir, transform=None):
        self.root_dir = os.path.normpath(root_dir)
        self.transform = transform
        all_img_data = load_files(root_dir)
        self.all_img_path = all_img_data['filenames']
        self.all_img_target = all_img_data['target']
        self.classes = all_img_data['target_names']

    def __len__(self):
        return len(self.all_img_path)

    def __getitem__(self, idx):
        img_path = self.all_img_path[idx]
        img_target = self.all_img_target[idx]
        img = Image.open(img_path).convert("RGB")
        if self.transform:
            img = self.transform(img)
        img_target = torch.from_numpy(np.array(img_target)).long()
        return img, img_target
```

```python
img_transform = 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])
                                   ])
dog_dataset = {}
dog_dataset['train'] = DogDataset("dogImages/train", img_transform)
dog_dataset['valid'] = DogDataset("dogImages/valid", img_transform)
dog_dataset['test'] = DogDataset("dogImages/test", img_transform)
```

Figure 4: Build custom PyTorch dataset

```python
batch_size = 64

loaders_scratch = {}
loaders_scratch['train'] = DataLoader(dog_dataset['train'], batch_size=batch_size, shuffle=True, num_workers=0)
loaders_scratch['valid'] = DataLoader(dog_dataset['valid'], batch_size=batch_size, shuffle=True, num_workers=0)
loaders_scratch['test'] = DataLoader(dog_dataset['test'], batch_size=batch_size, shuffle=True, num_workers=0)
```

Figure 5: Build PyTorch dataloaders from the built datasets

7 Implementation and Refinement

After discussing the algorithms and techniques in Section 5, I will introduce my implementations here. Fig. 6 shows the PyTorch implementation of the CNN model from scratch, which has a more readable summarization in Fig. 3. Since we utilize GPU(s) to train the model, we can move the entire model, i.e., all the model weights to the GPU(s).

```
import torch.nn as nn

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

        self.conv1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=5, stride=1)
        self.bn1 = nn.BatchNorm2d(num_features=16)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=5, stride=1)
        self.bn2 = nn.BatchNorm2d(num_features=32)

        self.conv3 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=1)
        self.conv4 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, stride=1)
        self.conv5 = nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, stride=1)

        self.linear1 = nn.Linear(in_features=256, out_features=512)
        self.dropout = nn.Dropout(p=0.5)
        self.linear2 = nn.Linear(in_features=512, out_features=n_classes)

    def forward(self, x):
        ## Define forward behavior
        x = F.relu(self.bn1(self.conv1(x)))
        x = self.pool(x)
        x = F.relu(self.bn2(self.conv2(x)))
        x = self.pool(x)
        x = F.relu(self.conv3(x))
        x = self.pool(x)
        x = F.relu(self.conv4(x))
        x = self.pool(x)
        x = F.relu(self.conv5(x))
        x = self.pool(x)
        x = F.adaptive_avg_pool2d(x, (1, 1))  # global average pooling
        x = x.view(x.shape[:2])  # squeeze the shape of (1, 1) at the last two dimensions
        # x = x.view(x.shape[0], -1)  # flatten
        x = self.dropout(F.relu(self.linear1(x)))
        x = self.linear2(x)

        return x

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

# instantiate the CNN
model_scratch = Net(n_classes)

# move tensors to GPU if CUDA is available
if use_cuda:
    model_scratch.cuda()
```

Figure 6: My implementation of the CNN model from scratch

Since it is a multiclass classification problem, cross entropy loss needs to be used. I also chose the Adam optimizer to update the network parameters (weights), as shown in Fig. 7.

The training process is designed in a standard way as a train function: for each training epoch, I load the training and validation data batch by batch from the corresponding dataloaders, move them to GPU if applicable, then calculate the losses and update the model parameters accordingly. For each batch operation, as a PyTorch standard, I need to clear all optimized variables before cal-

```
import torch.optim as optim

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

### TODO: select optimizer
lr = 1e-3
optimizer_scratch = optim.Adam(model_scratch.parameters(), lr)
```

Figure 7: My choices of loss function and optimizer

culating the model prediction. After calculating the training loss by applying the loss function with respect to the prediction and the loss, I perform backward propagation and update the parameters by one step. I save the model if validation loss has decreased and print out log messages including losses and whether the model has been saved to the console output. The train function returns the trained model as its output.

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

## TODO: Specify model architecture

model_transfer = models.resnet50(pretrained=True)

# freeze all model parameters
for param in model_transfer.parameters():
    param.require_grad = False

n_features = model_transfer.fc.in_features
model_transfer.fc = nn.Linear(n_features, n_classes)

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

Figure 8: Transfer learning on the ResNet50 model

Since the train function is quite long I don't place the codes here, please refer to this finished Jupyter Notebook for full details.

The test function is also designed similarly. We load the test dataset batch by batch from the built dataloaders, infer the model outputs and then get the element index that has a largest value in the 133-dimension classifier output vector. We can query the breed class name with such index.

The transfer learning of a pre-trained ResNet50 model is the refinement for our problem. Fig. 8 shows my implementation of transfer learning on the pre-trained ResNet50 model. Note that we need the freeze all ResNet50 layers to avoid them from being re-trained and replace its top (head) layer as a one-layer fully-connected network of 133 outputs. We only need to train such one-layer fully-connected network to re-align the extract features to the output dog breed classes.

8 Model Evaluation, Validation, and Justification

After training the CNN models we can do evaluations of their performances. The accuracy of the model from scratch reaches 17% (150/836). The accuracy of the transfer learning model reaches 86% (719/836). Both accuracy are better than the project targets of 10% and 60%, respectively.

This indeed proves that the pre-trained model like ResNet50 can be used as a good feature extractor to obtain all possible features from natural images, like dogs in this project. Such a feature

extractor are way better than the one we trained from scratch. This justifies the powerfulness of transfer learning.

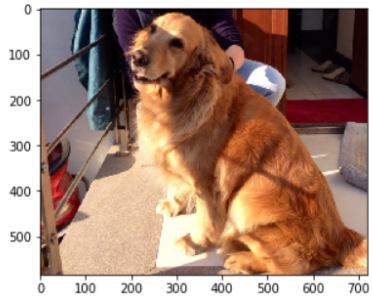
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.

def run_app(img_path):
    ## handle cases for a human face, dog, and neither
    img = cv2.imread(img_path)
    if dog_detector(ResNet50, img_path):
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)  # convert BGR image to RGB for plotting
        plt.imshow(img)
        plt.show()
        breed_name = predict_breed_transfer(img_path)
        print('A dog is detected in the file {}, and its predicted breed is {}.'.format(img_path, breed_name))
    elif face_detector_with_cascade(img_path):
        # face detection and mark bounding box for each detected face
        img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)  # convert BGR image to grayscale
        cascade_path = 'haarcascades/haarcascade_frontalface_alt.xml'
        face_cascade = cv2.CascadeClassifier(cascade_path)
        faces = face_cascade.detectMultiScale(img_gray)  # find faces in image
        for (x, y, w, h) in faces:
            # get bounding box for each detected face
            cv2.rectangle(img, (x, y), (x + w, y + h), (255, 0, 0), 2)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)  # convert BGR image to RGB for plotting
        plt.imshow(img)
        plt.show()
        breed_name = predict_breed_transfer(img_path)
        print('A human face is detected in the file {}, and it resembles a dog of breed {}.'.format(img_path, breed_name))
    else:
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)  # convert BGR image to RGB for plotting
        plt.imshow(img)
        plt.show()
        print('Error: Neither dog nor human is detected in the file {}'.format(img_path))
```

Figure 9: Top-level algorithm

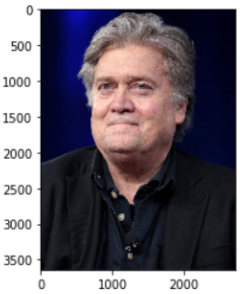
At last, we combines all the components of such project together as a whole. First, we use OpenCV's face detection function and the pre-trained ResNet50 model (without transfer learning at all) to roughly tell whether an input image contains a human face or a dog. If a dog is detected in the image, the ResNet50 model after transfer learning will provide an estimate of the dog's breed. If a human face is detected, the model will provide an estimate of the dog breed that is most resembling. If neither dog nor human face can be detected, it reports a message indicating such information. Fig. 9 summarizes such a top-level algorithm with a function named "run_app" for the project.

I also did some home-made tests by applying the 'run_app" function to some of my own dog images and other images I found online, as shown in Fig. fig:homemade-tests. Thanks to the well-trained transfer learning model, all the classification results of dogs are correct, while some classification results that look for the mostly resembled dog breed of a human face is quite fun! Also, for curiosity, I placed a cat image, and the application did tell me it's neither a dog nor a human face.



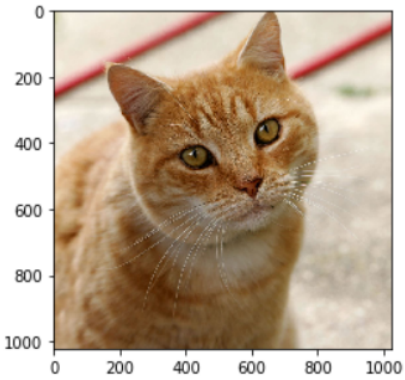

```
A dog is detected in the file additional_test_images\image_7.jpg, and its predicted breed is Golden retriever.
```

(a) Home-made test 1




```
A human face is detected in the file additional_test_images\image_3.jpg, and it resembles a dog of breed Lowchen.
```

(b) Home-made test 2




```
Error: Neither dog nor human is detected in the file additional_test_images\image_9.jpg
```

(c) Home-made test 3

Figure 10: Home-made tests