DOG BREED IDENTIFICATION

Machine Learning Capstone Project Report

Chaoyi Wang

Definition

Project Overview

Automatic image classification is one of the major topics in the research and application field. This category of techniques provides essential component for applications such as security surveillance, auto-pilot systems, product quality control, retail, radiography, scientific research, etc. Automatic image classification is commonly achieved by convolutional neural network (CNN), a deep learning framework.
The motivation for proceeding with this project is to learn and practice using convolutional neural network in image classification, starting with dog breeds.

Problem Statement

Dogs are people's best friends. However, there is literally countless number of dog breeds in the world. How can one tell what the breed of a dog is if first met or given a picture/video clip? This project aims to develop a deep leaning model using convolutional neural network framework that can distinguish a breed of a dog given a picture of the dog. The finished model should firstly feature the ability to distinguish whether the supplied picture is a dog or not. Secondly, the model should accurately identify dog breeds.

Metrics

The metrics for judging how the CNN model performs are validation loss values and prediction accuracy against test dataset. The validation loss value is defined by cross-entropy loss, which is also called the log loss or multi-class loss in some platforms (e.g. Kaggle). The loss value measures the performance of a classification model whose output is a probability value between 0 and 1. Cross-entropy values increases if the predicted probability diverges from the actual label[3].

Analysis

Data Exploration

The dataset is supplied by Udacity machine learning nanodegree program. The supplied datasets include a series of dog pictures with corresponding breeds and a series of human pictures.
The dog breed dataset includes the following components:

1. Train

    The training dataset contains 133 dog breeds with 30-70 pictures for each breed, there are 6680 dog images in total for training.

2. Test

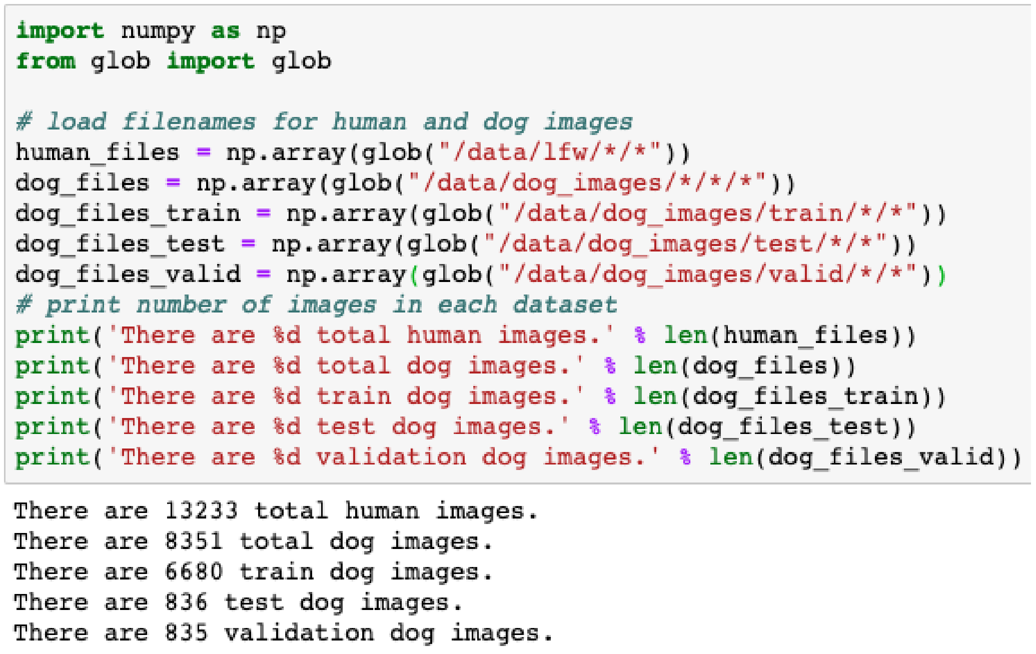
    The test dataset contains the same 133 dog breeds with 6-10 pictures for each breed, there are 836 dog images in total for testing.

3. Validation

    The validation dataset contains the same 133 dog breeds with 6-10 pictures for each breed, there are 836 dog images in total for validation.

The human dataset contains 13233 different pictures of human.

The dataset structure is shown in Figure 1. The class distributions in test, train, and validation datasets are shown in Figure 2. It seems that the class distributions of the dataset are quite uniform, with more uniformity in the validation dataset. Additionally, all the images are in RGB color space. All the human images feature a resolution of 250x250. Images of dogs feature various resolutions from ~200x~200 to ~1000x~1000.



```
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/*/*/*"))
dog_files_train = np.array(glob("/data/dog_images/train/*/*"))
dog_files_test = np.array(glob("/data/dog_images/test/*/*"))
dog_files_valid = np.array(glob("/data/dog_images/valid/*/*"))
# 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))
print('There are %d train dog images.' % len(dog_files_train))
print('There are %d test dog images.' % len(dog_files_test))
print('There are %d validation dog images.' % len(dog_files_valid))
```

```
There are 13233 total human images.
There are 8351 total dog images.
There are 6680 train dog images.
There are 836 test dog images.
There are 835 validation dog images.
```

Figure 1. Dataset structure.

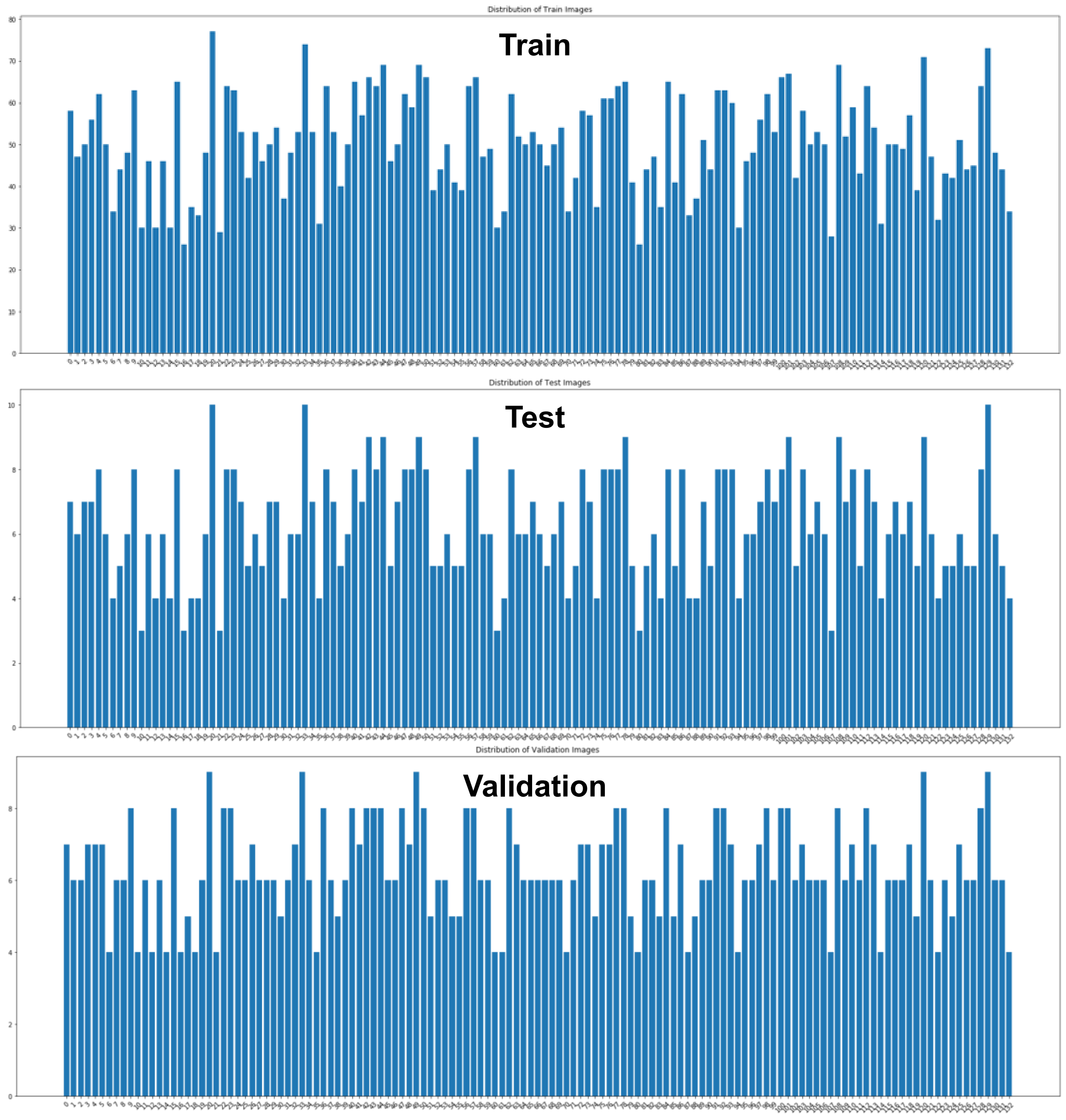


Figure 2. The class distribution in train, test, and validation datasets.

Exploratory Visualization

The majority of the images contains a single portrait of a dog of the corresponding breed. However, the dataset contains a small portion of images that include both dogs and humans, multiple dogs with different breeds (Figure 3). The challenge is how to successfully detect whether there is a dog in the image and how not to identify humans as dogs, or vice versa.

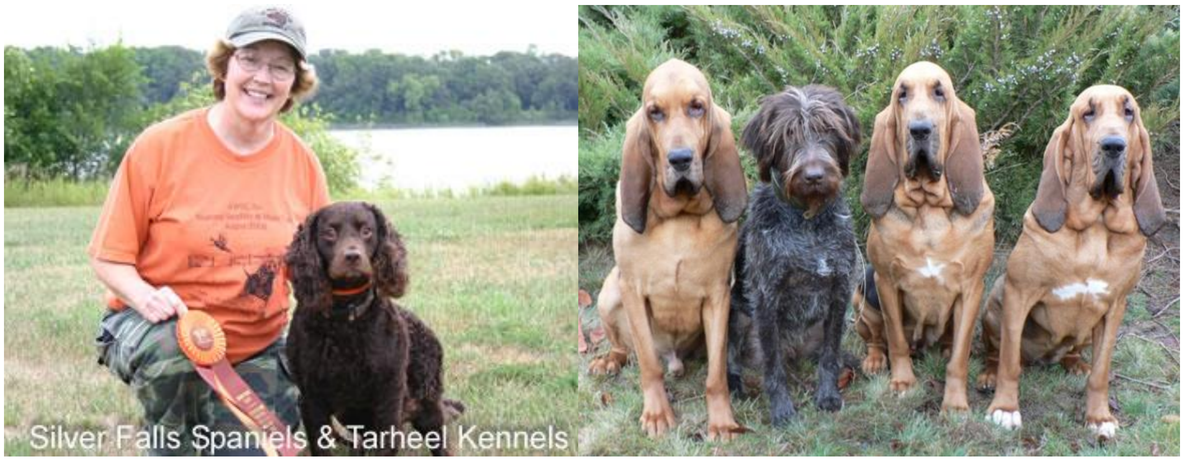


Figure 3. Pictures in training dataset of dog images showing human with dogs and multiple dogs with different breeds.

The human dataset is similar structured with most pictures featuring a single human portrait in the middle of the frame. There are some pictures showing multiple humans in a frame, but one human is emphasized (Figure 4).

Figure 4. A picture in the human dataset containing multiple faces.

Algorithms and Techniques

The solution of this project will include a trained detector using CNN to detect dog breeds based on the supplied picture. CNN stands for convolutional neural network, a deep learning framework which is widely used in image classifications. Convolutional layers are one of the basic components of a CNN model. CNN uses multiple layers of convolutional filters to extract features out of an image such as vertical/horizontal edges, groups, certain shapes, and colors, etc. A complete CNN model also features hyperparameters such as number of epochs, batch size, pooling, number of layers, weight, and dropouts. Example explanations of some hyperparameters are listed below[5]:

1. Number of epochs: the number of times the entire training set pass through the neural network.

2. Number of hidden layers: the number of convolutional and linear layers specified in the CNN model, more layers can result in higher computational cost, less layers can result in underfitting.
3. Dropout: a preferable regularization technique to avoid overfitting in deep neural networks. The method simply drops out units in neural network according to the desired probability.

The three hyperparameters listed above have been examined in this project.
CNN works extremely well with image classification applications because it follows a hierarchical model which resembles human brain. The established model features fully connected layers where all the neurons are connected to each other with specified outputs[6]. Images are composed of clouds, edges, colors, etc. These features are easily extractable by convolutional filters in the CNN models. Therefore, CNN is exceptionally effective in image classification applications.
Specifically, the input picture will go through two detectors, namely, human detector and dog detector. If human is detected in the picture, the algorithm will respond by displaying the image with a text describing the most resembled dog breed. If a dog is detected in the picture, the algorithm will display the image with a text indicating which breed the dog should belong to. If no human or dog can be recognized by the model, a text will be printed out indicating no human or dog is detected in the image. The human detector is constructed using haarcascade pre-trained model, the dog detector is constructed using VGG16 pre-trained model. The dog breed classifier is a transferred pre-trained ResNet50 model. The flow chart is presented as Figure 5.

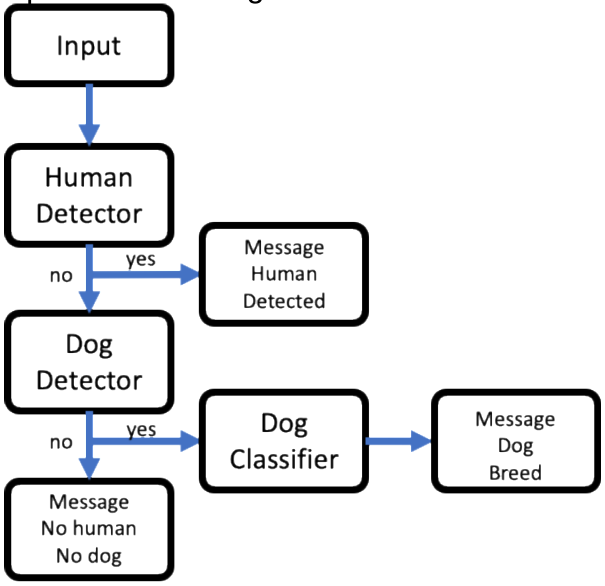


Figure 5. Dog classifier flow chart

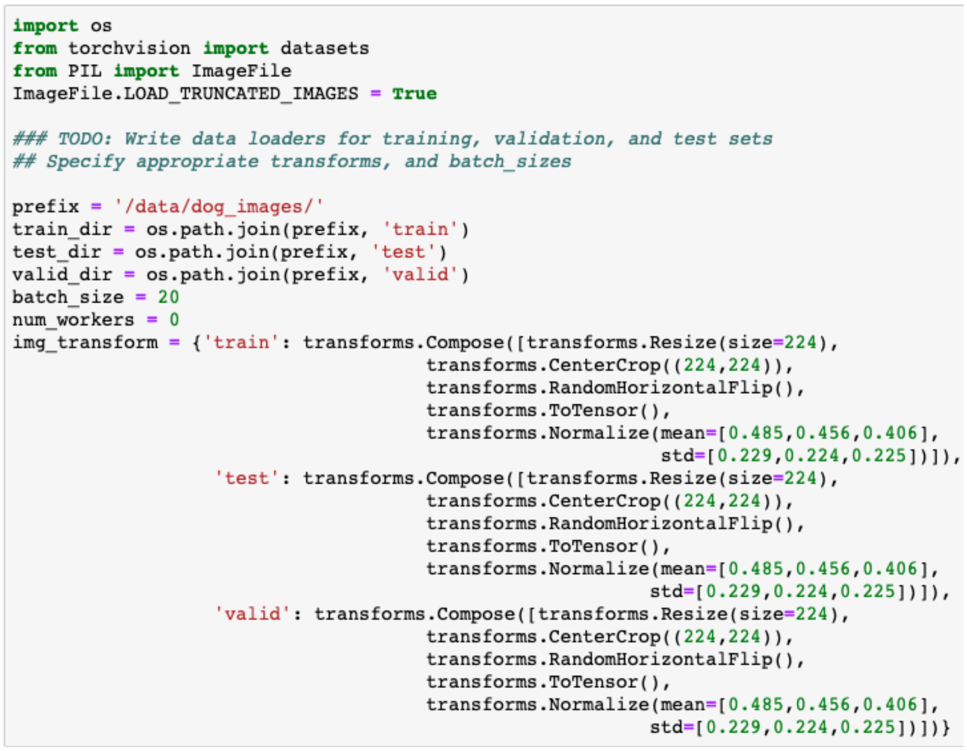
Benchmarks

The benchmark for the model can be referenced to the Kaggle leaderboard for dog breed identification competition. The target for this model is to reach a multiclass loss score less than 0.01, which is in the top 100 of the competition. The other benchmark will be 90% prediction accuracy, which will be used as the upper limit. The benchmark set by Udacity will be 60% prediction accuracy, which will be used as the lower limit. The final performance of the model will sit in between the two limits.

Methodology

Data Preprocessing

The training, test, validation images are resized and center cropped into 224x224 pixels, then randomly flipped in the horizontal direction before transforming into tensors. The transformed data are organized into train, test, and validation directories, respectively. The corresponding reprocessing code blocks are shown in Figure 6.
The reason for resizing, cropping is to achieve a uniform input data format. The random flipping increases the data variation in terms of features, thus making the dataset more robust.

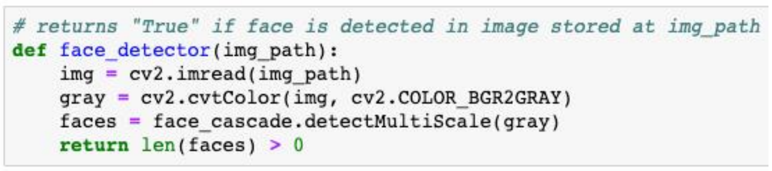


```
import os
from torchvision import datasets
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

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

prefix = '/data/dog_images/'
train_dir = os.path.join(prefix, 'train')
test_dir = os.path.join(prefix, 'test')
valid_dir = os.path.join(prefix, 'valid')
batch_size = 20
num_workers = 0
img_transform = {'train': transforms.Compose([transforms.Resize(size=224),
                                     transforms.CenterCrop((224,224)),
                                     transforms.RandomHorizontalFlip(),
                                     transforms.ToTensor(),
                                     transforms.Normalize(mean=[0.485,0.456,0.406],
                                                          std=[0.229,0.224,0.225])]),
                 'test': transforms.Compose([transforms.Resize(size=224),
                                     transforms.CenterCrop((224,224)),
                                     transforms.RandomHorizontalFlip(),
                                     transforms.ToTensor(),
                                     transforms.Normalize(mean=[0.485,0.456,0.406],
                                                         std=[0.229,0.224,0.225])]),
                 'valid': transforms.Compose([transforms.Resize(size=224),
                                     transforms.CenterCrop((224,224)),
                                     transforms.RandomHorizontalFlip(),
                                     transforms.ToTensor(),
                                     transforms.Normalize(mean=[0.485,0.456,0.406],
                                                         std=[0.229,0.224,0.225])])}
```

Figure 6. Data preprocessing code block

Implementation

Human detector: The image is firstly feed to a human detector function to detect if a human face is presented. The pre-trained model haar-cascade classifiers is implemented to achieve this functionality. Figure 7 shows the code block defining the human detector and a sample result.



```python
# 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
```

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

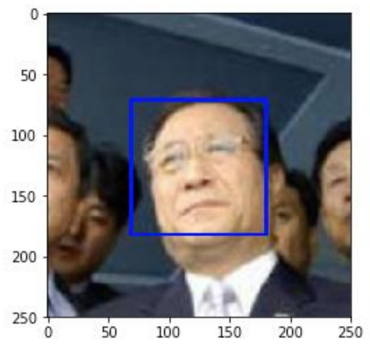
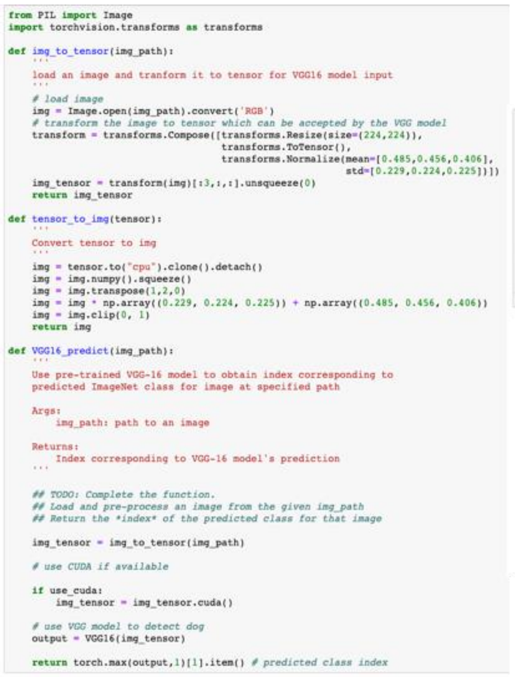
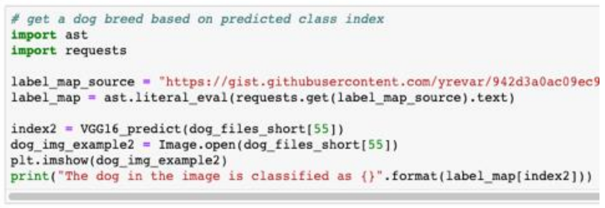


Figure 7. Human detector function and a sample result.

Dog detector: If no human face is detected by the human detector, the image will be passed to the dog detector to see if a dog is presented. The dog detector is constructed using a VGG16 pre-trained model which can identify classes in the 1000 categories4. Figure 8 shows the code block defining the dog detector and a sample result. Additionally, VGG16 can be used directly for dog breed identification (Figure 7), however, ResNet50 is going to be used as the pre-trained transfer training model in the final dog breed classifier.



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

def img_to_tensor(img_path):
    '''
    load an image and tranform it to tensor for VGG16 model input
    '''
    # load image
    img = Image.open(img_path).convert('RGB')
    # transform the image to tensor which can be accepted by the VGG model
    transform = transforms.Compose([transforms.Resize(size=(224,224)),
                                    transforms.ToTensor(),
                                    transforms.Normalize(mean=[0.485,0.456,0.406],
                                                         std=[0.229,0.224,0.225])])
    img_tensor = transform(img)[:3,:,:].unsqueeze(0)
    return img_tensor

def tensor_to_img(tensor):
    '''
    Convert tensor to img
    '''
    img = tensor.to("cpu").clone().detach()
    img = img.numpy().squeeze()
    img = img.transpose(1,2,0)
    img = img * np.array((0.229, 0.224, 0.225)) + np.array((0.485, 0.456, 0.406))
    img = img.clip(0, 1)
    return img

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_tensor = img_to_tensor(img_path)

    # use CUDA if available

    if use_cuda:
        img_tensor = img_tensor.cuda()

    # use VGG model to detect dog
    output = VGG16(img_tensor)

    return torch.max(output,1)[1].item() # predicted class index
```



```python
# get a dog breed based on predicted class index
import ast
import requests

label_map_source = "https://gist.githubusercontent.com/yrevar/942d3a0ac09ec9
label_map = ast.literal_eval(requests.get(label_map_source).text)

index2 = VGG16_predict(dog_files_short[55])
dog_img_example2 = Image.open(dog_files_short[55])
plt.imshow(dog_img_example2)
print("The dog in the image is classified as {}".format(label_map[index2]))
```

```
The dog in the image is classified as bull mastiff
```

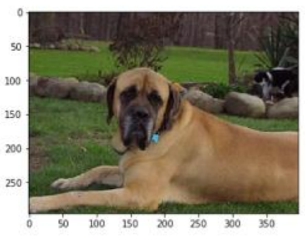


Figure 8. Dog detector using VGG16 model and a sample prediction using one of the input images.

Dog breed classifier: if a dog is detected by the dog detector, the image will be forwarded to the dog breed classifier. In this project, I first create a scratch CNN model for predicting the dog breed. The structure for the scratch CNN model is described below:

1. First convolution layer uses 32 filters, max pooling and stride reduced the image size to 56x56
2. Second convolution layer uses 64 filters, max pooling and stride reduced the image size to 14x14
3. Third convolution layer uses 128 filters and max pooling reduced the image size to 7x7
4. Two linear layers are assigned, with 133 as the output size.
5. Dropout is set to be 0.3 to avoid overfitting.

However, the scratch model is a simple CNN which has a high possibility of not achieving the accuracy metric (60%-90%) that is proposed in this project.

Refinement

Several structures have been tested with the scratch CNN model. Specifically, the number of filters (8 vs 16 in the first layer), pooling and striding (striding=0 vs striding=2). The scratch model is trained using GPU mode in Udacity container environment. The worst and best accuracy after 20 epochs is 8% and 11%, respectively. No drastic improvement is found using limited filters variations and pooling strategies. Therefore, I conclude that the structure of the scratch CNN is too basic for achieving higher accuracy. Because the accuracy metric achieved by the scratch CNN model is a far cry from the proposed accuracy (60%-90%) in this project. Therefore, using a more established pre-trained model makes sense. In this project, ResNet50 pre-trained model is used to serve as the real dog breed classifier. The ResNet50 model is tuned to match the 133 class outputs for this project. The ResNet50 model should theoretically produce much higher accuracy.

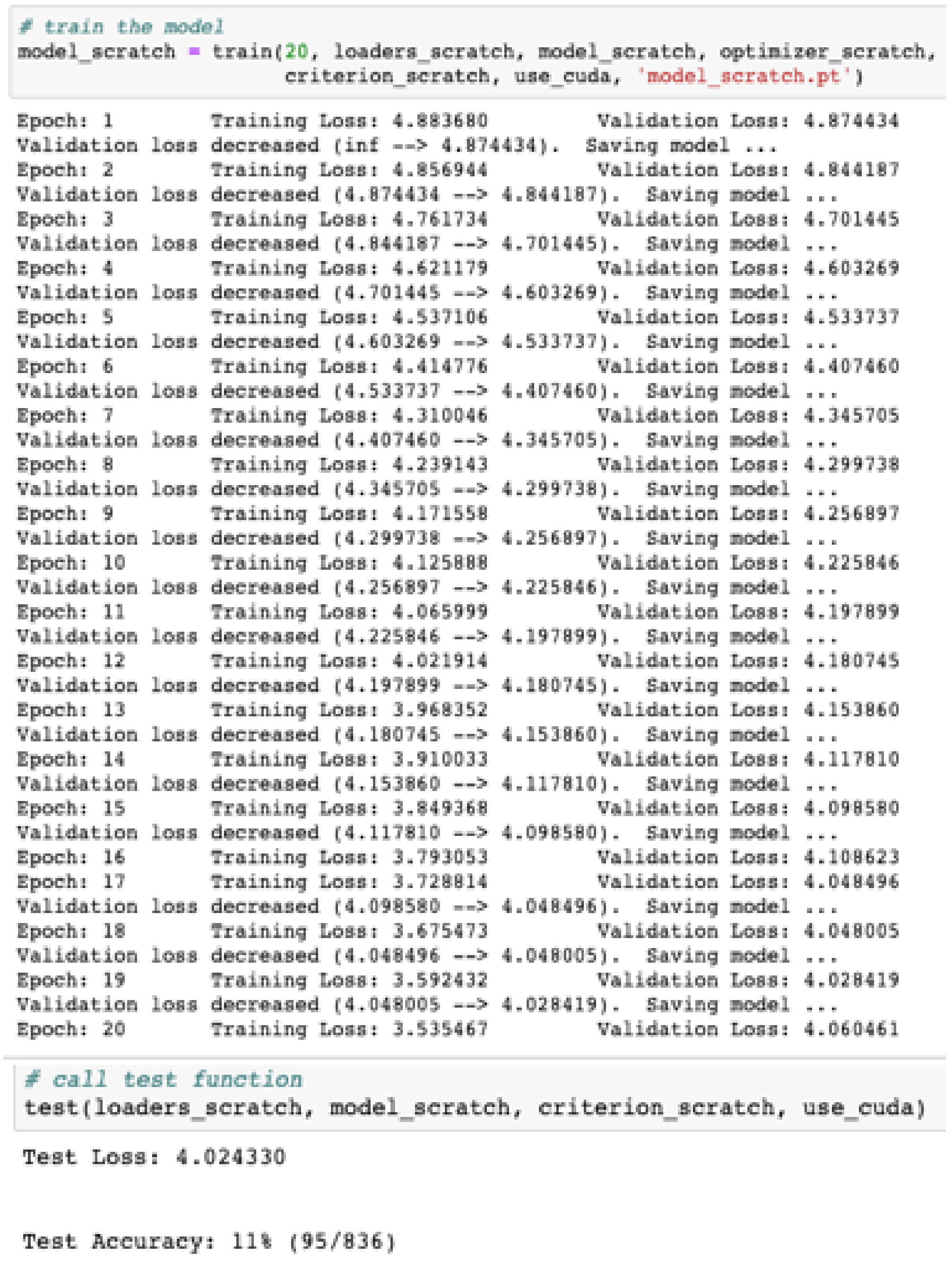
Results

Model Evaluation and Validation

As mentioned in the refinement section, the scratch CNN model only features a poor prediction accuracy of 11% (Figure 9). Specifically, after 20 epochs of training, the training loss and validation loss are reduced to ~3.53 and ~4.06, respectively. However, these are still very poor performance values comparing to some of the best metrics on Kaggle platform, which features a loss score of 0.01. The result indicate that the scratch model is too simple in terms of architecture. Additionally, the size of training data can be a caveat to a fresh model like this.

However, by implementing the ResNet50 model, after 20 epochs of training. Training loss and validation loss are reduced to ~1.72 and ~1.53 (Figure 10), which is a significant improvement comparing to the scratch CNN model. The transferred model also shows a test loss of ~1.51 and an accuracy of 77%. This result suggests a significant advantage

of using a pre-trained CNN model in terms of general image classification than building a CNN model from scratch.
The final algorithm and a sample prediction result are shown in Figure 11.



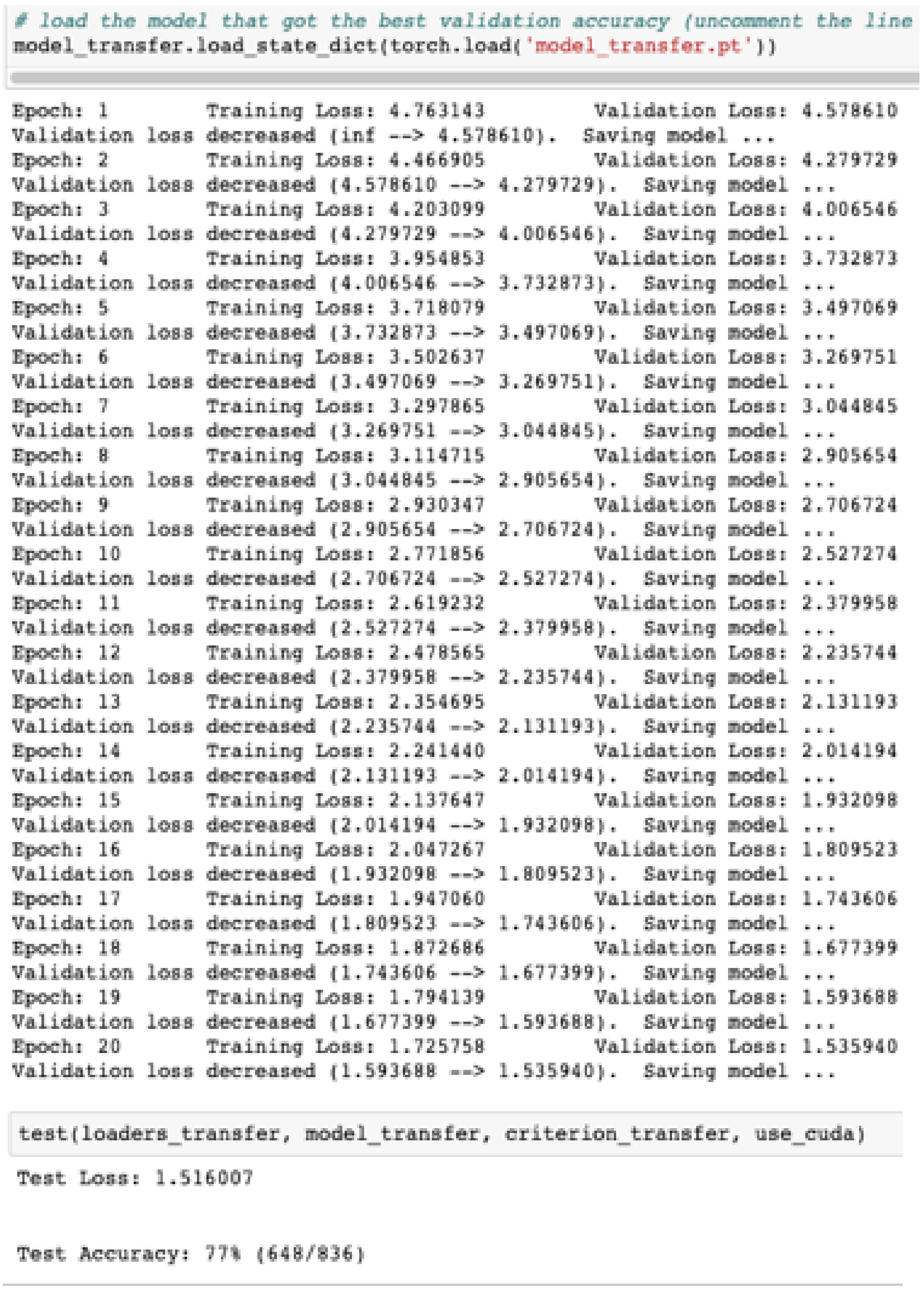
```
# train the model
model_scratch = train(20, loaders_scratch, model_scratch, optimizer_scratch,
                      criterion_scratch, use_cuda, 'model_scratch.pt')
```

```
Epoch: 1        Training Loss: 4.883680        Validation Loss: 4.874434
Validation loss decreased (inf --> 4.874434).  Saving model ...
Epoch: 2        Training Loss: 4.856944        Validation Loss: 4.844187
Validation loss decreased (4.874434 --> 4.844187).  Saving model ...
Epoch: 3        Training Loss: 4.761734        Validation Loss: 4.701445
Validation loss decreased (4.844187 --> 4.701445).  Saving model ...
Epoch: 4        Training Loss: 4.621179        Validation Loss: 4.603269
Validation loss decreased (4.701445 --> 4.603269).  Saving model ...
Epoch: 5        Training Loss: 4.537106        Validation Loss: 4.533737
Validation loss decreased (4.603269 --> 4.533737).  Saving model ...
Epoch: 6        Training Loss: 4.414776        Validation Loss: 4.407460
Validation loss decreased (4.533737 --> 4.407460).  Saving model ...
Epoch: 7        Training Loss: 4.310046        Validation Loss: 4.345705
Validation loss decreased (4.407460 --> 4.345705).  Saving model ...
Epoch: 8        Training Loss: 4.239143        Validation Loss: 4.299738
Validation loss decreased (4.345705 --> 4.299738).  Saving model ...
Epoch: 9        Training Loss: 4.171558        Validation Loss: 4.256897
Validation loss decreased (4.299738 --> 4.256897).  Saving model ...
Epoch: 10       Training Loss: 4.125888        Validation Loss: 4.225846
Validation loss decreased (4.256897 --> 4.225846).  Saving model ...
Epoch: 11       Training Loss: 4.065999        Validation Loss: 4.197899
Validation loss decreased (4.225846 --> 4.197899).  Saving model ...
Epoch: 12       Training Loss: 4.021914        Validation Loss: 4.180745
Validation loss decreased (4.197899 --> 4.180745).  Saving model ...
Epoch: 13       Training Loss: 3.968352        Validation Loss: 4.153860
Validation loss decreased (4.180745 --> 4.153860).  Saving model ...
Epoch: 14       Training Loss: 3.910033        Validation Loss: 4.117810
Validation loss decreased (4.153860 --> 4.117810).  Saving model ...
Epoch: 15       Training Loss: 3.849368        Validation Loss: 4.098580
Validation loss decreased (4.117810 --> 4.098580).  Saving model ...
Epoch: 16       Training Loss: 3.793053        Validation Loss: 4.108623
Epoch: 17       Training Loss: 3.728814        Validation Loss: 4.048496
Validation loss decreased (4.098580 --> 4.048496).  Saving model ...
Epoch: 18       Training Loss: 3.675473        Validation Loss: 4.048005
Validation loss decreased (4.048496 --> 4.048005).  Saving model ...
Epoch: 19       Training Loss: 3.592432        Validation Loss: 4.028419
Validation loss decreased (4.048005 --> 4.028419).  Saving model ...
Epoch: 20       Training Loss: 3.535467        Validation Loss: 4.060461
```

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

```
Test Loss: 4.024330


Test Accuracy: 11% (95/836)
```

Figure 9. The training log and test accuracy of the scratch CNN model.



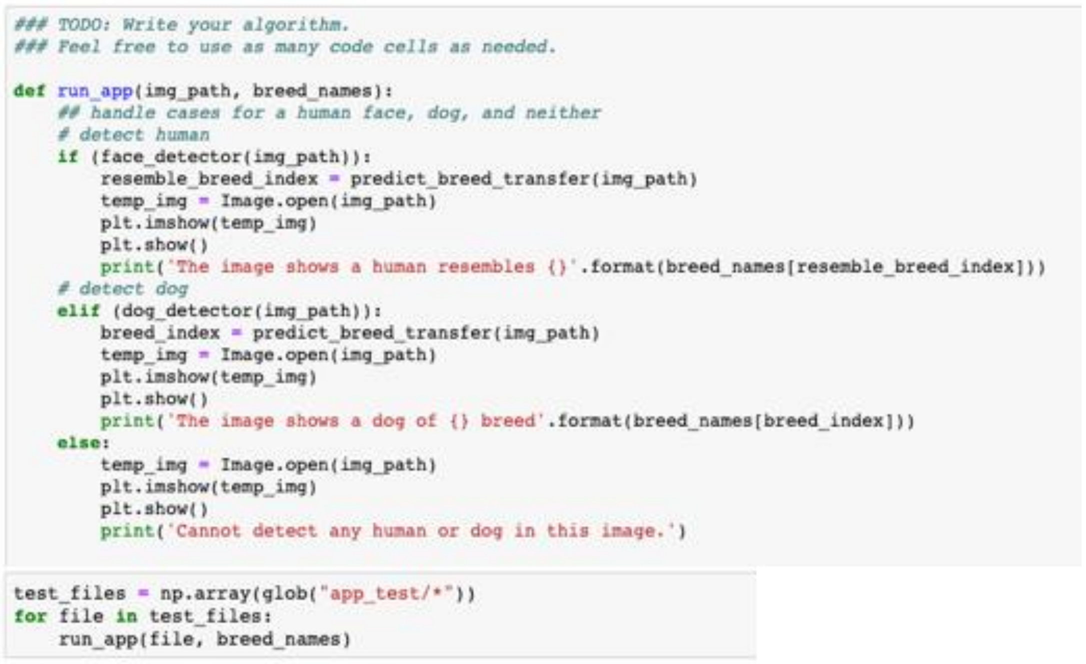
```
# load the model that got the best validation accuracy (uncomment the line
model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

```
Epoch: 1        Training Loss: 4.763143         Validation Loss: 4.578610
Validation loss decreased (inf --> 4.578610).  Saving model ...
Epoch: 2        Training Loss: 4.466905         Validation Loss: 4.279729
Validation loss decreased (4.578610 --> 4.279729).  Saving model ...
Epoch: 3        Training Loss: 4.203099         Validation Loss: 4.006546
Validation loss decreased (4.279729 --> 4.006546).  Saving model ...
Epoch: 4        Training Loss: 3.954853         Validation Loss: 3.732873
Validation loss decreased (4.006546 --> 3.732873).  Saving model ...
Epoch: 5        Training Loss: 3.718079         Validation Loss: 3.497069
Validation loss decreased (3.732873 --> 3.497069).  Saving model ...
Epoch: 6        Training Loss: 3.502637         Validation Loss: 3.269751
Validation loss decreased (3.497069 --> 3.269751).  Saving model ...
Epoch: 7        Training Loss: 3.297865         Validation Loss: 3.044845
Validation loss decreased (3.269751 --> 3.044845).  Saving model ...
Epoch: 8        Training Loss: 3.114715         Validation Loss: 2.905654
Validation loss decreased (3.044845 --> 2.905654).  Saving model ...
Epoch: 9        Training Loss: 2.930347         Validation Loss: 2.706724
Validation loss decreased (2.905654 --> 2.706724).  Saving model ...
Epoch: 10       Training Loss: 2.771856         Validation Loss: 2.527274
Validation loss decreased (2.706724 --> 2.527274).  Saving model ...
Epoch: 11       Training Loss: 2.619232         Validation Loss: 2.379958
Validation loss decreased (2.527274 --> 2.379958).  Saving model ...
Epoch: 12       Training Loss: 2.478565         Validation Loss: 2.235744
Validation loss decreased (2.379958 --> 2.235744).  Saving model ...
Epoch: 13       Training Loss: 2.354695         Validation Loss: 2.131193
Validation loss decreased (2.235744 --> 2.131193).  Saving model ...
Epoch: 14       Training Loss: 2.241440         Validation Loss: 2.014194
Validation loss decreased (2.131193 --> 2.014194).  Saving model ...
Epoch: 15       Training Loss: 2.137647         Validation Loss: 1.932098
Validation loss decreased (2.014194 --> 1.932098).  Saving model ...
Epoch: 16       Training Loss: 2.047267         Validation Loss: 1.809523
Validation loss decreased (1.932098 --> 1.809523).  Saving model ...
Epoch: 17       Training Loss: 1.947060         Validation Loss: 1.743606
Validation loss decreased (1.809523 --> 1.743606).  Saving model ...
Epoch: 18       Training Loss: 1.872686         Validation Loss: 1.677399
Validation loss decreased (1.743606 --> 1.677399).  Saving model ...
Epoch: 19       Training Loss: 1.794139         Validation Loss: 1.593688
Validation loss decreased (1.677399 --> 1.593688).  Saving model ...
Epoch: 20       Training Loss: 1.725758         Validation Loss: 1.535940
Validation loss decreased (1.593688 --> 1.535940).  Saving model ...
```

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

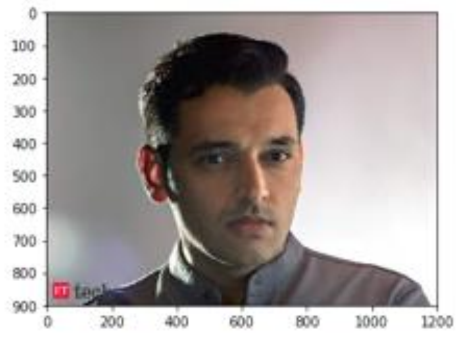
```
Test Loss: 1.516007


Test Accuracy: 77% (648/836)
```

Figure 10. The training log and test accuracy of the transferred ResNet50 model.



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

def run_app(img_path, breed_names):
    ## handle cases for a human face, dog, and neither
    # detect human
    if (face_detector(img_path)):
        resemble_breed_index = predict_breed_transfer(img_path)
        temp_img = Image.open(img_path)
        plt.imshow(temp_img)
        plt.show()
        print('The image shows a human resembles {}'.format(breed_names[resemble_breed_index]))
    # detect dog
    elif (dog_detector(img_path)):
        breed_index = predict_breed_transfer(img_path)
        temp_img = Image.open(img_path)
        plt.imshow(temp_img)
        plt.show()
        print('The image shows a dog of {} breed'.format(breed_names[breed_index]))
    else:
        temp_img = Image.open(img_path)
        plt.imshow(temp_img)
        plt.show()
        print('Cannot detect any human or dog in this image.')
```

```python
test_files = np.array(glob("app_test/*"))
for file in test_files:
    run_app(file, breed_names)
```



```
The image shows a human resembles 072.German_shorthaired_pointer
```

Figure 11. Final algorithm and a sample result

Justification

The best prediction accuracy I have achieved is 77% and the best test loss score is ~1.51. These results are still a fair distance away from the proposed performance. I have concluded the following ways to potentially improve the accuracy of my implementation.

1. Given more training time and epochs, the accuracy can be improved.
2. Larger input dataset for training can improve the accuracy of the model.
3. Manipulation on the training images to make input dataset more robust.
4. Play with the layer structures and hyperparameter tuning.
5. Examine the relevance of a different pretrained model.

References

1. Paul Viola and Michael J. Jones. Robust real-time face detection. International Journal of Computer Vision, 57(2):137–154, 2004.
2. Rainer Lienhart and Jochen Maydt. An extended set of haar-like features for rapid object detection. In Image Processing. 2002. Proceedings. 2002 International Conference on, volume 1, pages I–900. IEEE, 2002.
3. ml-cheatsheet, https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html
4. VGG16 1000 categories, https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a
5. A Walkthrough of Convolutional Neural Network — Hyperparameter Tuning, https://towardsdatascience.com/a-walkthrough-of-convolutional-neural-network-7f474f91d7bd
6. Overview of convolutional neural network in image classification, https://analyticsindiamag.com/convolutional-neural-network-image-classification-overview/