DOG BREED IDENTIFICATION

Machine Learning Capstone Project Report

Suganthan Arulvelan

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.

In [2]:
```
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.
```

Figure 1. Dataset structure.

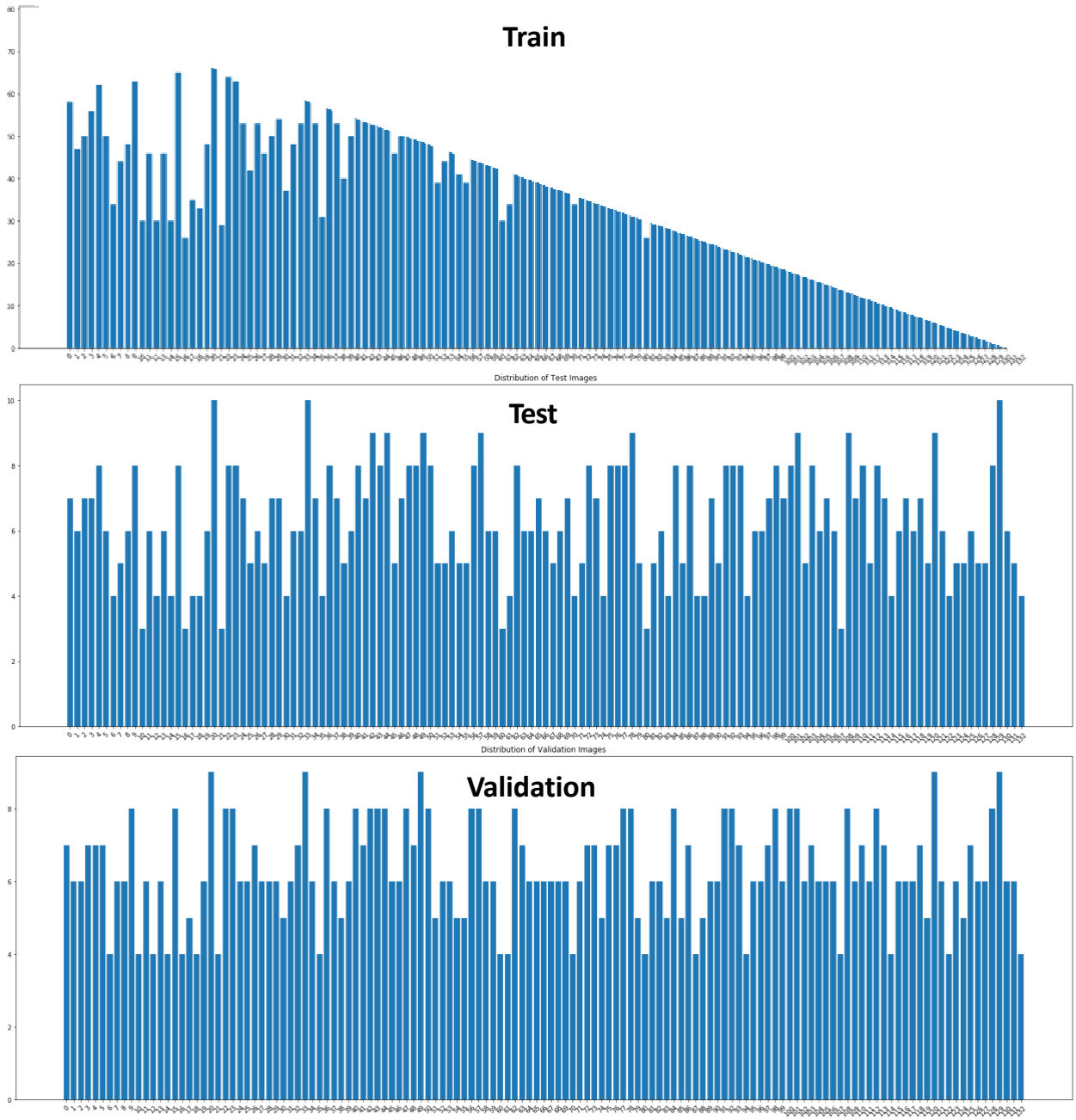


Figure 2. The class distribution in train, test, and validation datasets.

Exploratory Visualisation

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.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.
2. 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 VGN11 model. The flow chart is presented as Figure 5.

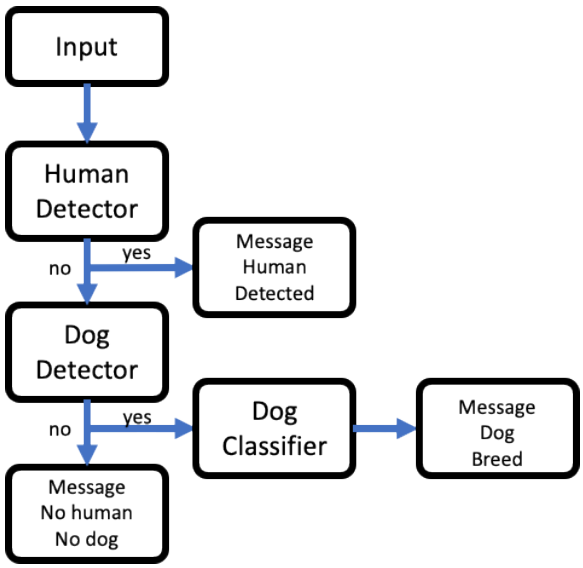


Figure 5. Dog classifier flow chart

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.

In [13]:
```
import os
from torchvision import datasets

### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes
train_transforms = transforms.Compose([transforms.Resize(224),
                                       transforms.CenterCrop(224),
                                       transforms.RandomHorizontalFlip(),
                                       transforms.RandomVerticalFlip(),
                                       transforms.RandomRotation(16),
                                       transforms.ToTensor(),
                                       transforms.Normalize([0.485, 0.456, 0.406],
                                                            [0.229, 0.224, 0.225])])

test_transforms = transforms.Compose([transforms.Resize(224),
                                      transforms.CenterCrop(224),
                                      transforms.ToTensor(),
                                      transforms.Normalize([0.485, 0.456, 0.406],
                                                           [0.229, 0.224, 0.225])])
train_data = datasets.ImageFolder( '/data/dog_images/train', transform=train_transforms )
test_data = datasets.ImageFolder( '/data/dog_images/test', transform=test_transforms )
valid_data = datasets.ImageFolder( '/data/dog_images/valid', transform=test_transforms )

batch_size = 20
num_workers = 0

train_loader = torch.utils.data.DataLoader(train_data,batch_size=batch_size, num_workers=num_workers,shuffle=True)
valid_loader = torch.utils.data.DataLoader(valid_data,batch_size=batch_size, num_workers=num_workers,shuffle=False)
test_loader = torch.utils.data.DataLoader(test_data,batch_size=batch_size, num_workers=num_workers,shuffle=False)

loaders_scratch = {
    'train': train_loader,
    'valid': valid_loader,
    'test': test_loader
}
```

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.

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 categories[4]

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.

Results

Model Evaluation and Validation

As mentioned in the refinement section, the scratch CNN model only features a poor prediction accuracy of 11% (Figure 7). 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 VGN11 model, after 20 epochs of training. Training loss and validation loss are reduced to ~1.72 and ~1.53 (Figure 8, 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 70%. 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 9.

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

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

```
Epoch: 1        Training Loss: 4.868939         Validation Loss: 4.988751
Validation loss decreased (inf --> 4.988751).  Saving model ...
Epoch: 2        Training Loss: 4.648924         Validation Loss: 4.352670
Validation loss decreased (4.988751 --> 4.352670).  Saving model ...
Epoch: 3        Training Loss: 4.453243         Validation Loss: 4.261151
Validation loss decreased (4.352670 --> 4.261151).  Saving model ...
Epoch: 4        Training Loss: 4.360094         Validation Loss: 4.266086
Epoch: 5        Training Loss: 4.282887         Validation Loss: 4.222140
Validation loss decreased (4.261151 --> 4.222140).  Saving model ...
Epoch: 6        Training Loss: 4.198280         Validation Loss: 4.049032
Validation loss decreased (4.222140 --> 4.049032).  Saving model ...
Epoch: 7        Training Loss: 4.111806         Validation Loss: 4.063550
Epoch: 8        Training Loss: 4.023125         Validation Loss: 3.865681
Validation loss decreased (4.049032 --> 3.865681).  Saving model ...
Epoch: 9        Training Loss: 3.931028         Validation Loss: 4.251086
Epoch: 10       Training Loss: 3.866554         Validation Loss: 3.736705
Validation loss decreased (3.865681 --> 3.736705).  Saving model ...
Epoch: 11       Training Loss: 3.791135         Validation Loss: 4.187199
Epoch: 12       Training Loss: 3.712351         Validation Loss: 3.922077
Epoch: 13       Training Loss: 3.629020         Validation Loss: 3.924531
Epoch: 14       Training Loss: 3.521366         Validation Loss: 3.775950
Epoch: 15       Training Loss: 3.452189         Validation Loss: 3.551825
Validation loss decreased (3.736705 --> 3.551825).  Saving model ...
Epoch: 16       Training Loss: 3.362992         Validation Loss: 3.798125
Epoch: 17       Training Loss: 3.275536         Validation Loss: 3.074409
Validation loss decreased (3.551825 --> 3.074409).  Saving model ...
Epoch: 18       Training Loss: 3.188429         Validation Loss: 3.105656
Epoch: 19       Training Loss: 3.102381         Validation Loss: 2.883252
Validation loss decreased (3.074409 --> 2.883252).  Saving model ...
Epoch: 20       Training Loss: 2.990252         Validation Loss: 2.869104
Validation loss decreased (2.883252 --> 2.869104).  Saving model ...
```

Figure 7. The training log and test accuracy of the scratch CNN model.

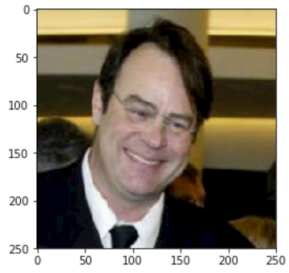
In [21]:
```
# train the model
model_transfer = train(20, loaders_transfer, model_transfer, optimizer_transfer, criterion_transfer, use_cuda, 'model_transfer
.pt')

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

```
Epoch: 1        Training Loss: 6.998098         Validation Loss: 4.745387
Validation loss decreased (inf --> 4.745387).  Saving model ...
Epoch: 2        Training Loss: 3.796046         Validation Loss: 5.601173
Epoch: 3        Training Loss: 2.770375         Validation Loss: 2.577967
Validation loss decreased (4.745387 --> 2.577967).  Saving model ...
Epoch: 4        Training Loss: 2.190026         Validation Loss: 3.430122
Epoch: 5        Training Loss: 1.695532         Validation Loss: 1.437858
Validation loss decreased (2.577967 --> 1.437858).  Saving model ...
Epoch: 6        Training Loss: 1.397064         Validation Loss: 1.521878
Epoch: 7        Training Loss: 1.225878         Validation Loss: 0.426645
Validation loss decreased (1.437858 --> 0.426645).  Saving model ...
Epoch: 8        Training Loss: 0.948300         Validation Loss: 1.271680
Epoch: 9        Training Loss: 0.871231         Validation Loss: 1.788332
Epoch: 10       Training Loss: 0.742894         Validation Loss: 0.562407
Epoch: 11       Training Loss: 0.723900         Validation Loss: 0.654009
Epoch: 12       Training Loss: 0.587596         Validation Loss: 0.280533
Validation loss decreased (0.426645 --> 0.280533).  Saving model ...
Epoch: 13       Training Loss: 0.451573         Validation Loss: 0.423101
Epoch: 14       Training Loss: 0.461241         Validation Loss: 0.642274
Epoch: 15       Training Loss: 0.394667         Validation Loss: 0.015940
Validation loss decreased (0.280533 --> 0.015940).  Saving model ...
Epoch: 16       Training Loss: 0.357921         Validation Loss: 0.029224
Epoch: 17       Training Loss: 0.340305         Validation Loss: 0.014211
Validation loss decreased (0.015940 --> 0.014211).  Saving model ...
Epoch: 18       Training Loss: 0.291707         Validation Loss: 0.588398
Epoch: 19       Training Loss: 0.294063         Validation Loss: 0.152412
Epoch: 20       Training Loss: 0.248706         Validation Loss: 0.012623
Validation loss decreased (0.014211 --> 0.012623).  Saving model ...
```

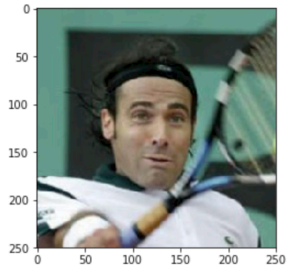
Figure 8. The training log and test accuracy of the transferred VGN11model.

In [27]:
```
## 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 detected!
```



```
This person looks like  American staffordshire terrier


Human detected!
```



```
This person looks like  Chihuahua
```

Figure 9. Final algorithm and a sample result

Justification

The best prediction accuracy I have achieved is 70% 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/