Udacity Machine Learning Nanodegree

Dog Breed Classification using CNN Report

Abhishek Chaturvedi

May, 2020

Definition

Project Overview:

CNN have been used to great effect in application such as object classification, scene recognition and other applications. Dog Breed Classification is a very specific and well known application of Convolutional Neural Network. It falls under the category of fine-grained image classification problem, where inter-class variations are small and often one small part of the image considered makes the difference in the classification.

Image classification has uses in most of the industries today. Some of these are the manufacturing industry, IT industry and Medical Science. Dog Breed Classification is especially useful for the animal services which work round the clock to provide shelter, food and families for the animals. Most of the animals which are rescued are dogs, in such a case an app like this which can tell about a dog's breed just by a picture comes in handy. The goal of the project is to identify the breed of a dog whose picture is given as input using Deep Learning techniques.

One of the main reasons, why I went for this project was to understand CNN in depth. The projects aim at identifying the breed of the dog in the input image. The model has been trained on 133 different dog breeds and hence it aims to predict the dog's breed from among these breeds. Additionally it can also identify a human image, and tells which breed of dog that image most resembles to.

Problem Statement:

The task at hand is to build a CNN model that accepts user-supplied images, and in return performs three tasks:

Dog face detector: The model outputs the breed of the dog, if the image is in fact belongs to a dog.

Human face detector: if the input is a human image, the model will output the breed with which the given human image most resembles with.

Exception case: The model will display a suitable message when the image contains neither a dog, nor a human.

Metrics:

I find Accuracy metric as the most suitable evaluation metric for the model. Accuracy is the ratio of number of correct predictions to the total number of input samples. Accuracy works best when the data is balanced. In our case the classes are approximately evenly distributed over training, validation and testing sets and hence we can use accuracy.

$$Accuracy = \frac{TrueNegatives + TruePositive}{TruePositive + FalsePositive + TrueNegative + FalseNegative}$$

$$\text{Precision} = \frac{True\ Positive}{True\ Positive + False\ Positive}$$

$$\text{Recall} = \frac{True\ Positive}{True\ Positive + False\ Negative}$$

- Precision is defined as the percentage of your results which are relevant
- Recall is defined as percentage of total relevant results correctly classified by your algorithm

Analysis

Data Exploration:

- **Datasets and Inputs:**

  Both the training and testing datasets comprise of images. The datasets have been provided by Udacity. The input format must be of image type. There are 2 datasets, namely Dog images dataset and Human images dataset.

  - **Human Images Dataset:** The dataset contains a total of 13233 human images. These images have different background and angles. All the images are of size 250x250.
  - **Dog Images Dataset:** This dataset consists of a total of 8351 dog images, which are distributed among 133 different dog breeds. These images are further distributed into train, test and validate set,

with 6,680 images in train set, 835 images in valid set and 836 images in test set.

- **Visualisation**

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.
```

1 Total number of dog and human images

```
# Data Exploration
total_len = len(image_datasets['train']) + len(image_datasets['test']) + len(image_datasets['valid'])
print("Total number of classes: " , len(image_datasets['train'].classes))
print("Total number of train records: " , len(image_datasets['train']))
print("Total number of test records: " , len(image_datasets['test']))
print("Total number of valid records: " , len(image_datasets['valid']))
print("Length of total dataset: " , total_len)
```

```
Total number of classes:  133
Total number of train records:  6680
Total number of test records:  836
Total number of valid records:  835
Length of total dataset:  8351
```

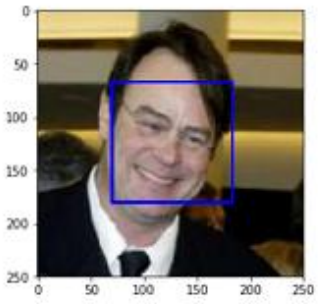
2 Division of dog dataset into train/valid and test dataset

Exploratory Visualization:

The human and dog detector can identify the images of humans and dogs respectively, The Human detector has the following code and outcome:

In [3]:
```
import cv2
import matplotlib.pyplot as plt
%matplotlib inline

# extract pre-trained face detector
face_cascade = cv2.CascadeClassifier('haarcascades/haarcascade_frontalface_alt.xml')

# load color (BGR) image
img = cv2.imread(human_files[0])
# convert BGR image to grayscale
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# find faces in image
faces = face_cascade.detectMultiScale(gray)

# print number of faces detected in the image
print('Number of faces detected:', len(faces))

# get bounding box for each detected face
for (x,y,w,h) in faces:
    # add bounding box to color image
    cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)

# convert BGR image to RGB for plotting
cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

# display the image, along with bounding box
plt.imshow(cv_rgb)
plt.show()
```

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



Algorithms and Techniques

The CNN model has the following code:

In [27]:
```
import torch.nn as nn
import torch.nn.functional as F

# define the CNN architecture
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(in_channels = 3 , out_channels = 32 , kernel_size = 3 , stride = 1 , padding = 1) # Takes in (224
        self.conv2 = nn.Conv2d(in_channels = 32 , out_channels = 64 , kernel_size = 3 , stride = 1 , padding = 1) # Takes in (11.
        self.conv3 = nn.Conv2d(in_channels = 64 , out_channels = 128 , kernel_size = 3 , stride = 1 , padding = 1) # Takes in (5
        self.conv4 = nn.Conv2d(in_channels = 128 , out_channels = 256 , kernel_size = 3 , stride = 1 , padding = 1) # Takes in (.
        self.conv5 = nn.Conv2d(in_channels = 256 , out_channels = 512 , kernel_size = 3 , stride = 1 , padding = 1) # Takes in (

        # dropout layer (p=0.3)
        self.dropout = nn.Dropout(0.3)

        # linear layers
        self.fc1 = nn.Linear(in_features = 512*7*7 , out_features = 512)
        self.fc2 = nn.Linear(in_features = 512 , out_features = 256)
        self.out = nn.Linear(in_features = 256 , out_features = 133)


    def forward(self, x):
        ## Define forward behavior
        #input layer
        x = x

         # hidden conv layer 1
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x , kernel_size = 2 , stride = 2)

         # hidden conv layer 2
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x , kernel_size = 2 , stride = 2)

         # hidden conv layer 3
        x = F.relu(self.conv3(x))
        x = F.max_pool2d(x , kernel_size = 2 , stride = 2)

         # hidden conv layer 4
        x = F.relu(self.conv4(x))
        x = F.max_pool2d(x , kernel_size = 2 , stride = 2)

        # hidden conv layer 5
        x = F.relu(self.conv5(x))
        x = F.max_pool2d(x , kernel_size = 2 , stride = 2)

        # hidden linear layer 1
        x = x.reshape(-1 , 512*7*7) #must be flattened for 1st linear layer
        x = F.relu(self.fc1(x))
        x = self.dropout(x)

        # hidden linear layer 2
        x = F.relu(self.fc2(x))
        x = self.dropout(x)

        # output layer
        x = self.out(x)

        return x
```

My CNN architecture consists of 5 Convolutional layers and 3 Fully Connected Linear Layers. Each convolutional layer is followed by a max_pool2d function which has a kernel size = 2 and stride = 2, similarly each Linear layer except the output layer is followed by a dropout layer with a dropout probability of 0.3.

For the first conv layers there are 3 in-channels and 32 out channels, after that I just kept increasing the channels by a factor of 2. Each conv layer has a kernel size = 3 , stride = 1 and padding = 1.

After the tensor passes through the conv layers it reaches the 1st linear layer, there I flatten the tensor (the logic of flattening is described below) and pass it through the 1st linear layer, after that the tensor passes through a dropout layer and then again to a linear layer and finally to the output

layer. Since the number of output classes are 133 (i.e the number of dog breeds in dataset) Hence the output layer has 133 out-features. Along with this I have included a signature function which prints "Abhishek's net" when print(model_scratch) is used , and in order to provide a summary of my model , I have used torchsummary module which is very similar to Keras's model.summary()

- **Logic for tensor flattening:**
  - Suppose we have N x N input.
  - Suppose we have f x f filter.
  - Suppose we have a padding of p and a stride of s.

The output size O is given by this formula:

$$O = \frac{n - f + 2p}{s} + 1$$

This value will be the height and width of the output. However, if the input or the filter isn't a square, this formula needs to be applied twice, once for the width and once for the height.

Now we started with a tensor of (224 x 224) hence after the 1st conv. layer its size is:

O1 = ((224 - 3 + 2*1) / 1) + 1 = 224

But when this O1 passes through the max_pool2d layer the output becomes:

O2 = ((O1 - 2 + 0) / 2) + 1 = ((224 - 2) / 2) + 1 = 112 ,

This is also evident from the model summary cell.

Hence like this the tensor reaches the 5th conv. layer as (14 x 14) and after that it passes through the final max_pool2d function it becomes (7 x 7), hence to flatten the tensor I use (512 (output_channels of last conv layer) x 7 x 7)

The second part consists of fine tuning a ResNet50 model. The screenshot for the code is attached below:

In [40]:
```
import torchvision.models as models
import torch.nn as nn

## TODO: Specify model architecture
model_transfer = models.resnet50(pretrained = True)

'''https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html'''

# freeze the layers of already trained resnet50 model to avoid furthur trainin
for param in model_transfer.parameters():
    param.requires_grad = False

# remove the last fc layer and add a new one insted which has 133 out_features
num_ftrs = model_transfer.fc.in_features
model_transfer.fc = nn.Linear(num_ftrs , 133)

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

In the code I just froze the architecture and weights of the ResNet50 model in order to avoid any further training, and then I replaced the last layer with a custom layer with total number of out channels = 133 i.e. equal to the total breeds of dogs in dataset.

Benchmark

For this problem, I will be using VGG-16 as the benchmark model. VGG-16 is a CNN architecture which was used to win ILSVR(ImageNet) competition in 2014.VGG-16 consists of an architecture and weights which have already been trained on ImageNet which is the most popular dataset used for image classification. It makes the improvement over AlexNet by replacing large kernel-sized filters (11 and 15 in the first and second convolutional layer, respectively) with multiple 3x3 kernel - sized filters one after another. VGG16 was trained for weeks and was using NVIDIA Titan Black GPU. Below is the architecture of a standard VGG16 model.

I tried to beat the existing VGG16 model which got an accuracy of about 27% on test set , the dataset which was used to do the training and testing consisted of 120 dog breed classes , here is the link to the paper(ref 1)

Methodology

Data Pre-processing

The pre-processing part consisted of splitting the dataset into train, test and validation sets. After splitting the data I applied transformations to the data which augmented the data. Below is the code used for transformations:

```
# build the transforms dictionary
transforms = {
    'train' : transforms.Compose([transforms.CenterCrop(224),
                                  transforms.Resize(224),
                                  transforms.RandomHorizontalFlip(p = 0.5),
                                  transforms.RandomRotation(degrees = (0 , 30)),
                                  transforms.ToTensor(),
                                  transforms.Normalize(mean = [0.485, 0.456, 0.406], std = [0.229, 0.224, 0.225])]),

    'valid' : transforms.Compose([transforms.Resize(224),
                                  transforms.CenterCrop(224),
                                  transforms.ToTensor(),
                                  transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])]),

    'test' : transforms.Compose([transforms.Resize(224),
                                  transforms.CenterCrop(224),
                                  transforms.ToTensor(),
                                  transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])]),
}
```

I performed data augmentation for the train, test and valid set using the following methods:

- Train set : I first used CenterCrop to Crop the given Image at the centre , the I resized the image to 224x224 the performed RandomHorizontalFlip with a probability of 0.5 , the I used RandomRotation with a minimum degree of 0 and a max degree of 30.
- Finally I converted the augmented image to a tensor and then normalized it.
- Valid and Test sets : For these sets I did not perform augmentations as the model does not train on these images, I only cropped the images in this set at the centre and the resized them to 224x224 before converting them to tensor and normalizing them
- I selected a batch size of 50 , which I think is neither too high or too low
- Finally I used shuffle on only the train loader as the purpose of shuffling is to avoid the model to train on images belonging to some specific classes, hence it has no effect on predictions.

Implementation

I divided the project into 3 parts-

- Pre-processing: In this step I imported all the required datasets and libraries. After that I will pre-process the data.
- Data Splitting: In this step I spliced the input data into train, validate and test datasets, and then performed image augmentation on train/validate/test sets.
- Model training and evaluation: This step comprised of the following:
    - Created a dog detector using VGG16 model.
    - Trained, tested and validated a CNN model built from scratch.
    - Again created a CNN model, but this time using transfer learning with ResNet50, and finally trained, tested and validated the data on this model.
    - Evaluated the model by displaying of appropriate messages as output.

Results

Model Evaluation and Validation

The first task I did was to make a CNN model from scratch and train, validate and test the model. After training the model for 40 epochs I got an accuracy of a mere 11%, although it satisfies the pass criterion of (accuracy > 10%) but it is bad.

After this I used a pretrained ResNet50 model as my prediction model and just replaced the last linear layer with a custom linear layer which had 133 out-features corresponding to the 133 dog breeds classes in the dataset. This model performed fairly well and achieved an accuracy of 78%, which I was completely satisfied with

Justification

The model used in transfer learning achieved a quiet good accuracy, it is very high than the passing accuracy of 60%, and hence I think the model did quite well, however there is still a chance of improvement. These points must be kept in mind while trying to improve the model:

- Hyperparameter optimization can increase accuracy.
- A bigger training dataset and better data augmentation might result in better training and hence better predictions.
- Using more epochs for training might result in better trained model.
- Other evaluation metric can be tried
- Model can be designed in such a way that it shows the next most closest breed of the dog.

References

1. http://noiselab.ucsd.edu/ECE228_2018/Reports/Report18.pdf
2. https://github.com/udacity/machine-learning/blob/master/projects/capstone/report-example-1.pdf
3. https://deeplizard.com/learn/video/cin4YcGBh3Q