Machine Learning Engineer Nanodegree

Project Report

Dog Breed Classifier with CNNs

Hassan Raza Bukhari

30th October, 2020

Background and Motivation

There are currently over 300 identified dog breeds all around the world and identifying them is an interesting problem and this problem. Distinguishing different animals would be relatively easier but distinguishing precisely the breed of a dog is somewhat difficult and it is not immediately clear that which features to be used for this purpose and which algorithm might prove to be useful. Clearly this is a multi-class classification problem which must utilize a robust algorithm to be accurately identify the dog breed. Convolutional Neural Networks (CNN) has proved to be very effective for similar applications classifying different objects, so it would be a good idea to utilize that. It must also be noted that some dog breeds have very similar features and are very hard to distinguish even for humans as well. Therefore, CNN based classifier might also face issues in being able to distinguish certain dog classes from each other. Testing it on data would show much accurate the predictions are.

Introduction

Dog breed classification is a rather tricky classification problem because there are so many different breeds of dogs and that visually a lot of them look very similar, and some same breed dogs have different color. Thus, for the model to classify image of a dog properly there must be a robust algorithm to address this problem. For this project, Convolutional Neural Networks (CNNs) were utilized because it proves to be effective for image classification problems. CNNs were used in two ways, firstly a model was built and trained from scratch and then a model using transfer learning was utilized to maximize accuracy. In addition to classifying dog images based on the breed of the dog, the algorithm can also take as input an image of a human and classify it into any resembling dog breed. There are images of dogs of 133 different breeds in the dataset used for this project, although there are a lot more than that.

Problem Statement

In simple words the problem statement is, *When given input image of a dog, identifying the breed of that dog and when given image of a human, identifying the resembling dog breed.*

The classifier solves two problems

Dog Breed Classification: The classifier will be able to classify images of dogs into different breeds.

Human face classification: The classifier will classify human face image into resembling breed of dog.

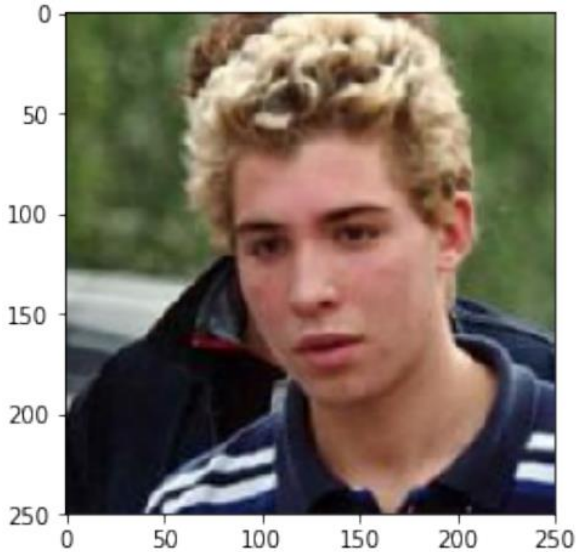
Datasets used

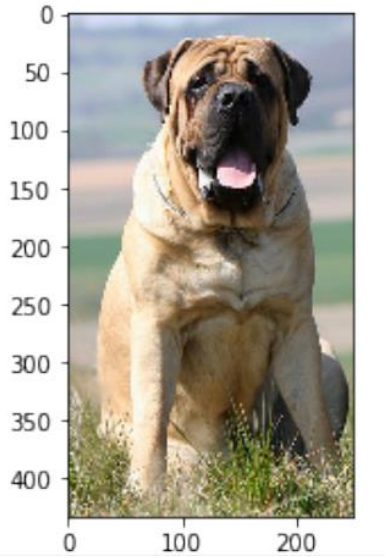
Two datasets were used during this project

Dog Dataset: Dog Dataset containing images of different dog breeds. It contains 8351 images having 131 different dog breeds. Images are divided into train and test datasets

Human Dataset: This dataset includes a set of human images to be used during the project. It contains 13233 images.

Example images from both datasets are shown below





h

Model Evaluation

For self-evaluation of model, since there are 133 different breeds of dogs in the dataset of this project, thus randomly guessing the breed of a dog would give us less than 1% accuracy. Therefore, our model built and trained from scratch using CNN should produce an accuracy of at least 10%, while the one with transfer learning should have accuracy of around 60%.

Criterion for Accuracy

The criterion to estimate the accuracy used is simply the *ratio of number of images correctly classified to the total number of images classified.*

Project Design (Development Stages)

The following plan was followed for the development of project

1. Importing the datasets
2. Detecting Humans
3. Detecting Dogs
4. Creating a CNN to classify dog breeds (from scratch)
5. Creating a CNN to class dog breeds (from transfer learning)
6. Writing the algorithm
7. Testing the algorithm

1. Importing Datasets

The first stage in the development of any project is to obtain and import datasets that would be used for the project. The code below was used to first import the data and then list the total number of

```
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))
```

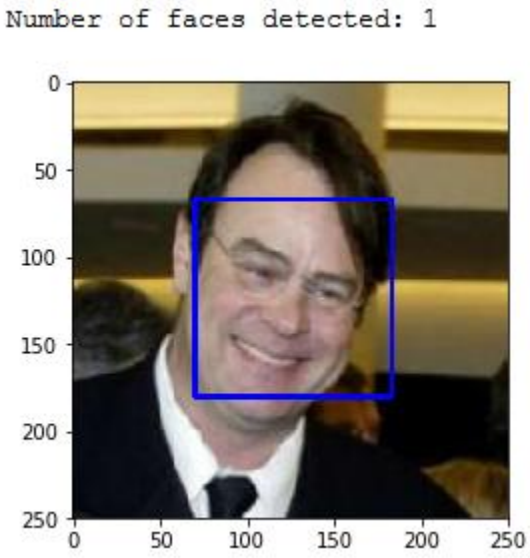
2. Detecting Humans

To detect human faces in an image a pre trained detector from Open CV was utilized. The exact detector used was **Haarcascade Frontal Face Detector.** The code used to import and make use of the pre trained detector is given below:

```
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()
```

When given a human image as input, the result looks something like this:



100 images from both datasets were supplied to this detector and it did not give very good results. It detected 98 out of 100 human faces which is pretty good, but it also detected 17 out of 100 images as human faces, which is pretty bad.

3. Detecting Dogs

The next step was to detect dogs when an image is supplied. The algorithm used for detecting human faces was good but it did make more mistakes than I expected. Therefore, to improve upon this, the model chosen for detecting dogs was a **pre trained VGG-16** model which proved to be very good.

The model was obtained from torchvision models using the code below:

```
import torch
import torchvision.models as models
from torchvision.models.vgg import model_urls

model_urls['vgg16'] = model_urls['vgg16'].replace('https://', 'http://')


# 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()
```

The actual detector or predictor was built after this stage. Several transformations were applied to the images so that they are suitable to be used with the model.

```
from PIL import Image, ImageFile
import torchvision.transforms as transforms
ImageFile.LOAD_TRUNCATED_IMAGES = True

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')

    normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],std=[0.229, 0.224, 0.225])

    transformations = transforms.Compose([transforms.Resize(size=(224, 224)),
                                          transforms.ToTensor(),
                                          normalize])

    img_t = transformations(img)[:3,:,:].unsqueeze(0)

    if use_cuda:
        img_t = img_t.cuda()

    result = VGG16(img_t)
```

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

This function returns a prediction between 0 – 999 which are 1000 possible categories from ImageNet. 151 to 268 of these prediction values correspond to dogs. Therefore, if value is between and including these values then we can say that the model predicted the image to contain a dog.

Thus, extending this concept to code the detector function which uses the previous function

```
def dog_detector(img_path):
    ## TODO: Complete the function.
    if VGG16_predict(img_path) in range(151,269):
        return True

    return False # true/false
```

On assessing the dog detector on 100 images from both datasets. It detected all 100 dog images correctly and made only one mistake with the human dataset.

4. Creating a CNN to Classify Dog Breeds (from Scratch)

Creating a CNN model and training it on the dataset from scratch would be the next step in the process.

Loading and Preprocessing Data

To do this, firstly train, test and validation datasets are loaded and some transformations are done on them to make them suitable for the model.

```
import os
from torchvision import datasets

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

batch_size = 20
num_workers = 0


train_path = '/data/dog_images/train'
val_path = '/data/dog_images/valid'
test_path = '/data/dog_images/test'

normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                 std=[0.229, 0.224, 0.225])

train_dataset = datasets.ImageFolder(train_path,  transforms.Compose([
            transforms.RandomResizedCrop(224),
            transforms.RandomHorizontalFlip(),
            transforms.RandomRotation(15),
```

```
            transforms.ToTensor(),
            normalize,
        ]))

val_dataset = datasets.ImageFolder(val_path,  transforms.Compose([
            transforms.Resize(size=(224,224)),
            transforms.ToTensor(),
            normalize,
        ]))

test_dataset = datasets.ImageFolder(test_path,  transforms.Compose([
             transforms.Resize(size=(224,224)),
            transforms.ToTensor(),
            normalize,
        ]))

train_loader = torch.utils.data.DataLoader(train_dataset, batch_size= 20, num_workers = 0, shuff
le = True)

val_loader = torch.utils.data.DataLoader(val_dataset, batch_size= 20, num_workers = 0, shuffle
= False)

test_loader = torch.utils.data.DataLoader(test_dataset, batch_size= 20, num_workers = 0, shuffle
= False)

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

Model Architecture

After that the actual model architecture was defined.

```
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

        self.conv1 = nn.Conv2d(3, 36, 3, padding=1)
        self.conv2 = nn.Conv2d(36, 64, 3, padding=1)
```

```
        self.conv3 = nn.Conv2d(64, 128, 3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(28*28*128, 512)
        self.fc2 = nn.Linear(512, 133)
        self.dropout = nn.Dropout(0.25)
        self.batch_norm = nn.BatchNorm1d(512)

    def forward(self, x):

        x = self.conv1(x)
        x = F.relu(x)
        x = self.pool(x)

        x = self.conv2(x)
        x = F.relu(x)
        x = self.pool(x)

        x = self.conv3(x)
        x = F.relu(x)
        x = self.pool(x)

        x = x.view(-1, 28*28*128)

        x = self.fc1(x)
        x = self.batch_norm(x)
        x = F.relu(x)

        x = self.dropout(x)
        x = self.fc2(x)
        x = F.relu(x)

        return x

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

# instantiate the CNN
model_scratch = Net()

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

Three convolutional networks were used in this architecture, each having a stride equal to 1 and kernal size equal to 3. They have output sizes 36, 64 and 128 respectively. Pooling layer of (2,2) is used wheras ReLU activation from nn being utilized. Lastly we have two fully connected layers that has output size of 133 which is kept such that it is equal to the number of dog breeds in the dataset.

Loss function and Optimizer

CrossEntropyLoss function was used as a loss function while SGD optimizer was used as an optimizer.

```
import torch.optim as optim

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

### TODO: select optimizer
optimizer_scratch = optim.SGD(model_scratch.parameters(), lr = 0.03)
```

Training the model

After that the model was trained so that it can learn to distinguish different breeds of dogs

```
def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
    """returns trained model"""
    # initialize tracker for minimum validation loss
    valid_loss_min = np.Inf

    for epoch in range(1, n_epochs+1):
        # initialize variables to monitor training and validation loss
        train_loss = 0.0
        valid_loss = 0.0

        ###################
        # train the model #
        ###################
        model.train()
        for batch_idx, (data, target) in enumerate(loaders['train']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## find the loss and update the model parameters accordingly
            ## record the average training loss, using something like
            ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss))
            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()
            train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss))

        ######################
        # validate the model #
        ######################
        model.eval()
```

```
        for batch_idx, (data, target) in enumerate(loaders['valid']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## update the average validation loss
            output = model(data)
            loss = criterion(output, target)
            valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_loss))


        # print training/validation statistics
        print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
            epoch,
            train_loss,
            valid_loss
            ))

        ## TODO: save the model if validation loss has decreased
        if valid_loss < valid_loss_min:
            print("Saving model. Validation loss has decreased to ",valid_loss)
            torch.save(model.state_dict(), save_path)
            valid_loss_min = valid_loss


    # return trained model
    return model


# train the model
model_scratch = train(10, 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'))
```

For the training, 10 epochs were used.

Testing the model

The model was tested to estimate the loss on test data and also to estimate the accuracy.

```
def test(loaders, model, criterion, use_cuda):

    # monitor test loss and accuracy
    test_loss = 0.
    correct = 0.
    total = 0.
```

```
    model.eval()
    for batch_idx, (data, target) in enumerate(loaders['test']):
        # move to GPU
        if use_cuda:
            data, target = data.cuda(), target.cuda()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the loss
        loss = criterion(output, target)
        # update average test loss
        test_loss = test_loss + ((1 / (batch_idx + 1)) * (loss.data - test_loss))
        # convert output probabilities to predicted class
        pred = output.data.max(1, keepdim=True)[1]
        # compare predictions to true label
        correct += np.sum(np.squeeze(pred.eq(target.data.view_as(pred))).cpu().numpy())
        total += data.size(0)

    print('Test Loss: {:.6f}\n'.format(test_loss))

    print('\nTest Accuracy: %2d%% (%2d/%2d)' % (
        100. * correct / total, correct, total))

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

The model gave 10% accuracy which was the minimum criterion decided for the scratch model to be acceptable.

5. Creating a CNN to Classify Dog Breeds (with Transfer Learning)

Transfer learning will be used to maximize the accuracy for our classifier. The minimum accuracy decided for this was 60%, although this proved to be much better than that.

Loading Data

The same dataloaders were copied from the previous model

```
loaders_transfer = loaders_scratch.copy()
```

Model Architecture

For the transfer learning **pretrained resnet50** model was used. It was imported from torchvision models as well.

```
import torchvision.models as models
```

```
import torch.nn as nn

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

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

Since this was an image classification problem therefore the pretrained resnet model was chosen because it very effective model when it comes to image classification. Briefly describing the steps taken to make it applicable for our dataset.

- Obtained the pretrained resnet model
- Looked at the model architecture to get an idea of output features
- The output features must be 133, i.e equal to our classification classes, thus that was done

Loss function and optimizer

Same loss function and optimizer were used as the scratch model

Training the model

Model was trained using the same function used for the scratch model

```
model_transfer = train(21, loaders_transfer, model_transfer, optimizer_transfer, criterion_transfe
r, use_cuda, 'model_transfer.pt')
```

Testing the model

Similarly, it was tested to find out its accuracy after the training was complete.

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

The accuracy came out to be **85%** which was better than I expected as it was only trained for 21 epochs.

Predicting dog breed with the model

The model was now trained and ready to put in action, so a predictor was coded so that it can be used to make it easier or the application to predict using the model

```
### TODO: Write a function that takes a path to an image as input
### and returns the dog breed that is predicted by the model.
data_transfer = loaders_transfer
# list of class names by index, i.e. a name can be accessed like class_names[0]
class_names = [item[4:].replace("_", " ") for item in data_transfer['train'].dataset.classes]

from PIL import Image
import torchvision.transforms as transforms

def preprocess_image(img_path):
```

```
    img = Image.open(img_path).convert('RGB')
    prediction_transform = transforms.Compose([transforms.Resize(size=(224, 224)),
                                transforms.ToTensor(),
                                normalize])

    img = prediction_transform(img)[:3,:,:].unsqueeze(0)
    return img

def predict_breed_transfer(img_path):
    # load the image and return the predicted breed
    img = preprocess_image(img_path)
    if use_cuda:
        img = img.cuda()

    result = model_transfer(img)
    prediction = torch.max(result,1)[1].item()
    return class_names[prediction]
```

Writing the algorithm

The algorithm must have the following properties

- 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.

A simple app was coded that would take an image path as input and perform predictions based on the conditions mentioned above

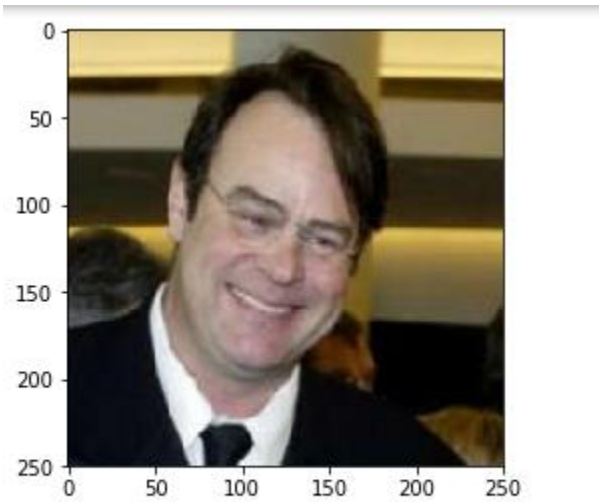
```
def run_app(img_path):
    ## handle cases for a human face, dog, and neither
    img = Image.open(img_path)
    plt.imshow(img)
    plt.show()

    if dog_detector(img_path):
        print("Dog was detected in the image")
        print("The dog breed is found to be {}".format(predict_breed_transfer(img_path)))
    else:
        print("Human detected in the image")
        print("The resemling dog breed is {}".format(predict_breed_transfer(img_path)))
```

Testing the algorithm

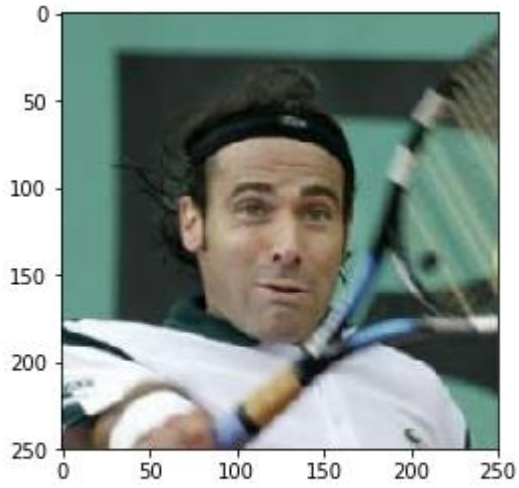
The algorithm was tested on a few images from both datasets

```
for file in np.hstack((human_files[:3], dog_files[:3])):
    run_app(file)
```

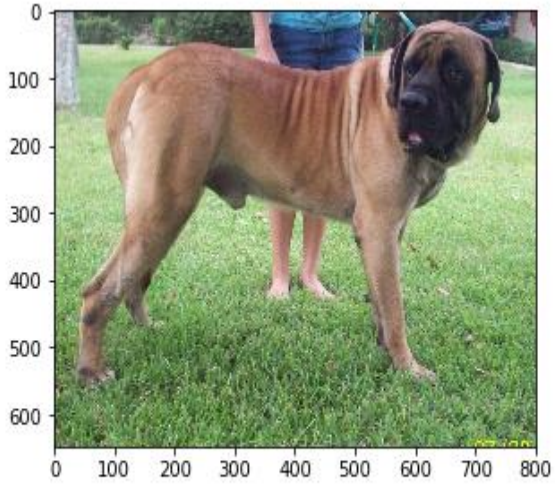
The results are



```
Human detected in the image
The resemling dog breed is Chihuahua
```

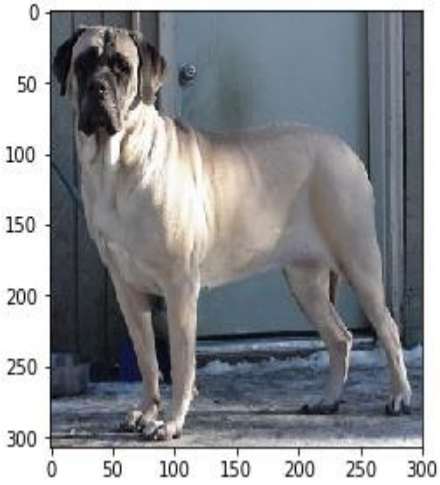


```
Human detected in the image
The resemling dog breed is Basenji
```



```
Dog was detected in the image
The dog breed is found to be Bullmastiff
```

```
Dog was detected in the image
The dog breed is found to be Bullmastiff
```



Improvement of algorithm

Algorithm can be improved by doing the following things

1. Using a model with more layers like resnet101

2. Tuning the hyperparameters systematically
3. Training for more epochs, as this was trained for less number of epochs

References

1. **Base Repository**: Original GitHub repository used as a base for this project.
2. **Domain Knowledge:** Using Convolutional Neural Networks to classify dog breeds
3. **Datasets:** Dog Dataset and Human Dataset
4. **Benchmark Models:** Deep Learning and Transfer Learning approaches for Image Classification