DOG BREED CLASSIFIER

Udacity Machine Learning Engineer Capstone Project Report

Photo by Karsten Winegeart on Unsplash

Rahul Chauhan

30.10.2020

1. DEFINITION

Project Overview

In this project we aimed to build a neural network based multiclass classifier that could identify a breed of dog if given a photo or image as input. If the photo or image contains a human face (or alien face), then the application will return the breed of dog that most resembles this person.

The project uses Convolutional Neural Networks (CNNs)! A pipeline is built to process real-world, user-supplied images. Given an image of a dog, the 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.

Problem Statement

Biodiversity is one of the most complex and vital features of our planet. Biodiversity is a modern term which simply means 'the variety of life on earth'. This variety can be measured on several different levels. Among these diverse animals we call one a man's best friend. Even though they are all individually different we can differentiate them from their visual and other features into different breeds. We wish to train a machine to be able to recognise and classify these numerous species. Also for fun, we will try to classify humans into dog breeds that the trained model thinks looks similar to them.

Metrics

We used accuracy as our metric for evaluation. Accuracy is the proportion of correct classifications among all classifications.

$$accuracy := \frac{\text{correct classifications}}{\text{number of classifications}}$$

Accuracy is a simple and useful measure if we have the same amount of samples per

class, which is the case with the dataset that we are working on.

When we use accuracy, we assign equal cost to false positives and false negatives. Since we do not prefer any one breed over others and misclassification of all breeds have similar penalty and correct classifications similar rewards, accuracy can give us a good idea about the effectiveness of our classifier.

2. ANALYSIS

Data Exploration

The dog_names variable stores a list of the names for the classes to use in our prediction model. Based on the path name, we see a total of 8351 images of dogs belonging to 133 different dog breeds which are then categorized into 6680, 835, and 836 images in training, validation, and testing.

Dog Images — The dog images provided are available in the repository within the Images directory further organized into the train, valid and test subfolders

Human Faces — An exhaustive dataset of faces of celebrities have also been added to the repository in the lfw folder

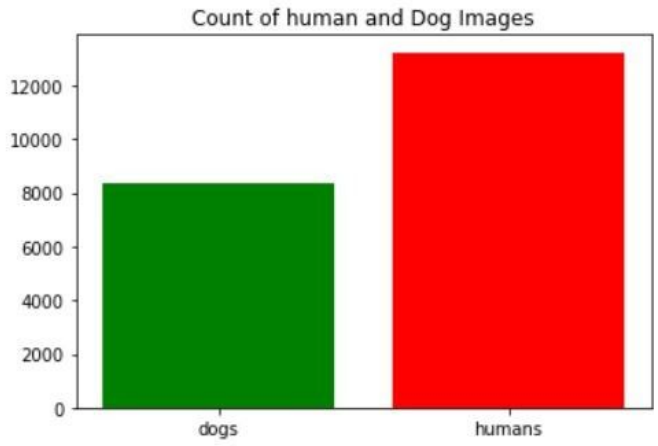
Haarcascades — ML-based approach where a cascade function is trained from a lot of positive and negative images, and used to detect objects in other images. The algorithm uses the Haar frontal face to detect humans. So the expectation is that an image with the frontal features clearly defined is required

Test Images — A folder with certain test images have been added to be able to check the effectiveness of the algorithm

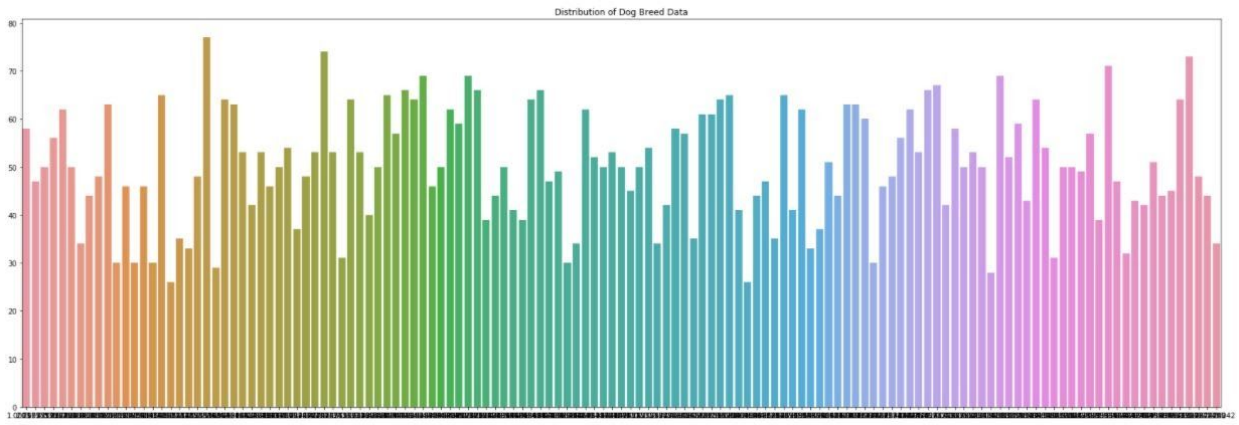
1. We check for samples available

```
import matplotlib.pyplot as plt
plt.bar(['humans', 'dogs'],[len(human_files), len(dog_files)], color=['red', 'green'])
plt.title('Count of human and Dog Images')
```

```
Text(0.5,1,'Count of human and Dog Images')
```



2. Distribution of breed in dog_photos



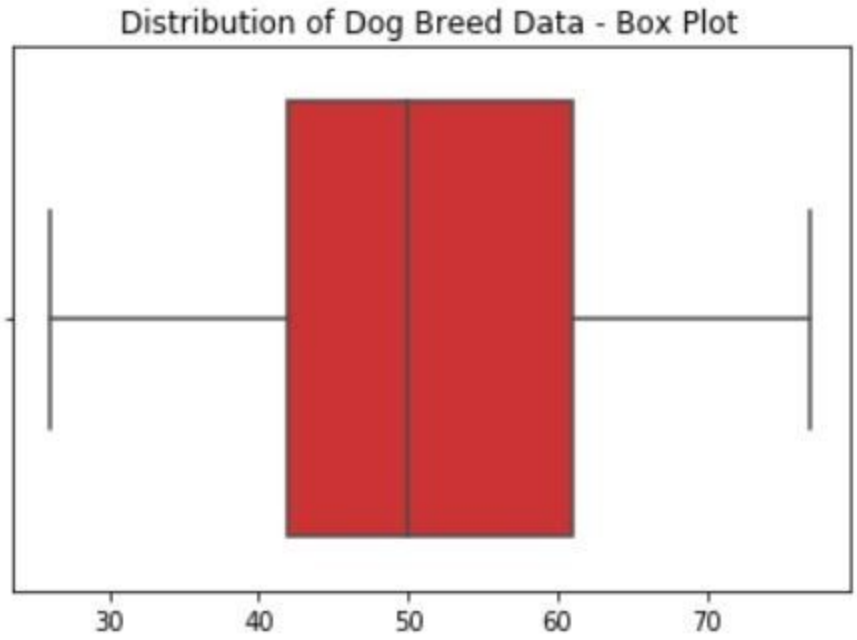
3. Check for outliers

```
print("Standard Deviation = ", np.std(count))
```

```
Standard Deviation =  11.8191997117
```

```
sns.boxplot(x=count, palette="Set1").set_title("D
```

```
Text(0.5,1,'Distribution of Dog Breed Data - Box
```



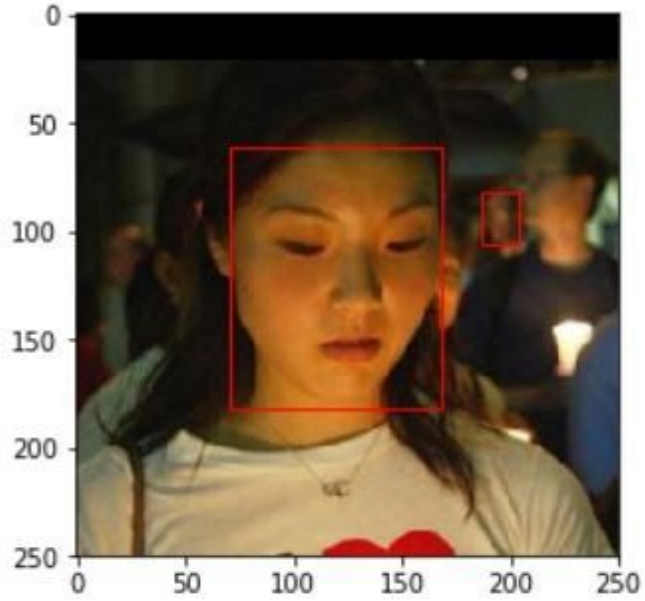
Exploratory Visualisation

1. Found humans with both dogs and humans

```
#images with detected human faces by opencv in dog dataset in first 50 images

for img_path in dog_files_short[:50]:
    if face_detector(img_path):
        img = cv2.imread(img_path)
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        faces = face_cascade.detectMultiScale(gray)
        for (x,y,w,h) in faces:
            cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)
        cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        plt.imshow(cv_rgb)
        plt.show()
```

2. Explored Neural Network based algorithms for face detection




```
In [8]: def face_detector_mtcnn(img_path):
            pixels = plt.imread(img_path)
            faces = mtcnn_detector.detect_faces(pixels)
            return len(faces) > 0
```

Algorithms and Techniques

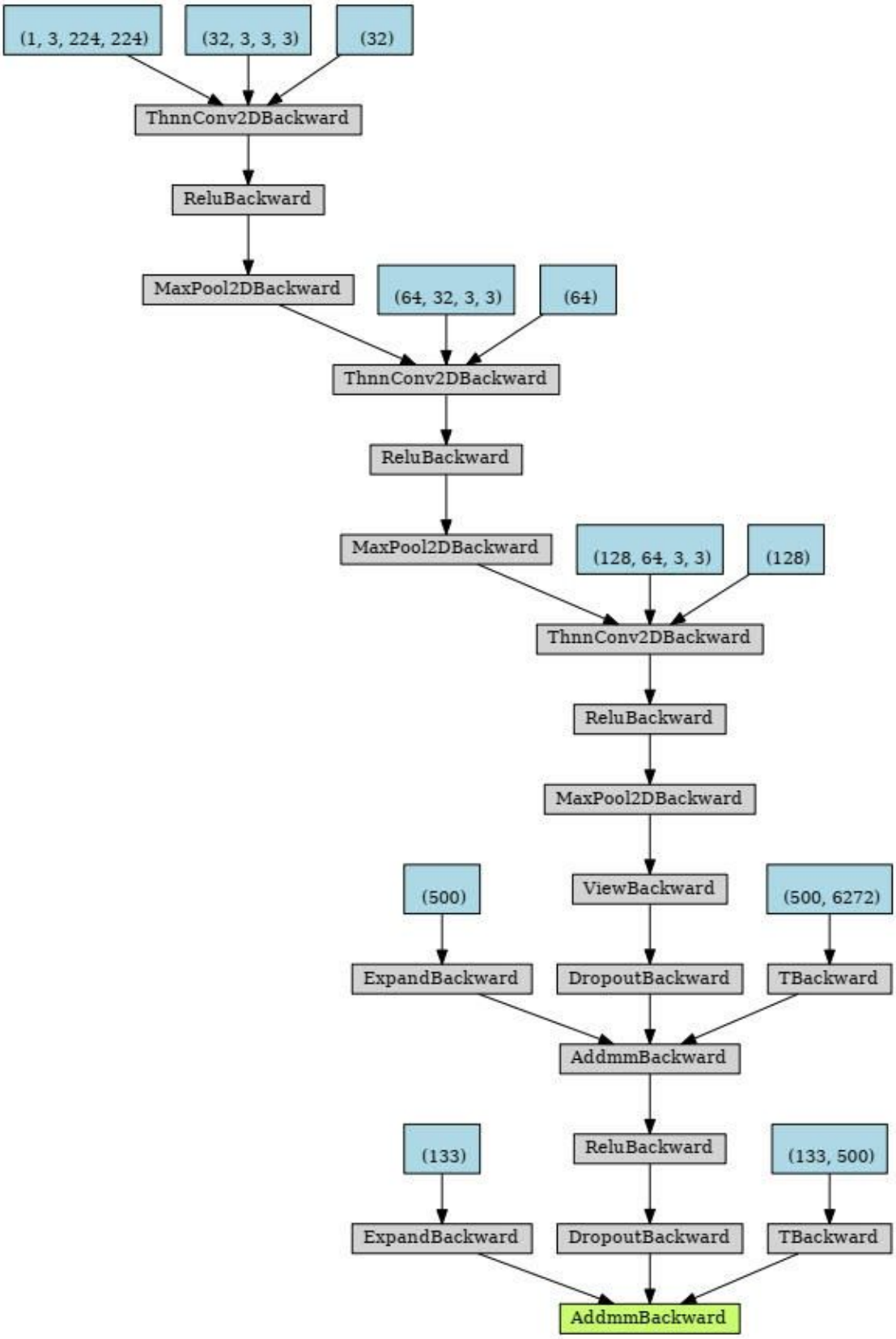
1. Created a basic convolutional neural network from scratch with the following architecture.

   A Convolutional Neural Network (ConvNet/CNN) is a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other. The pre-processing required in a ConvNet is much lower as compared to other classification algorithms. While in primitive methods filters are hand-engineered, with enough training, ConvNets have the ability to learn these filters/characteristics.

   The objective of the Convolution Operation is to extract the high-level features such as edges, from the input image.

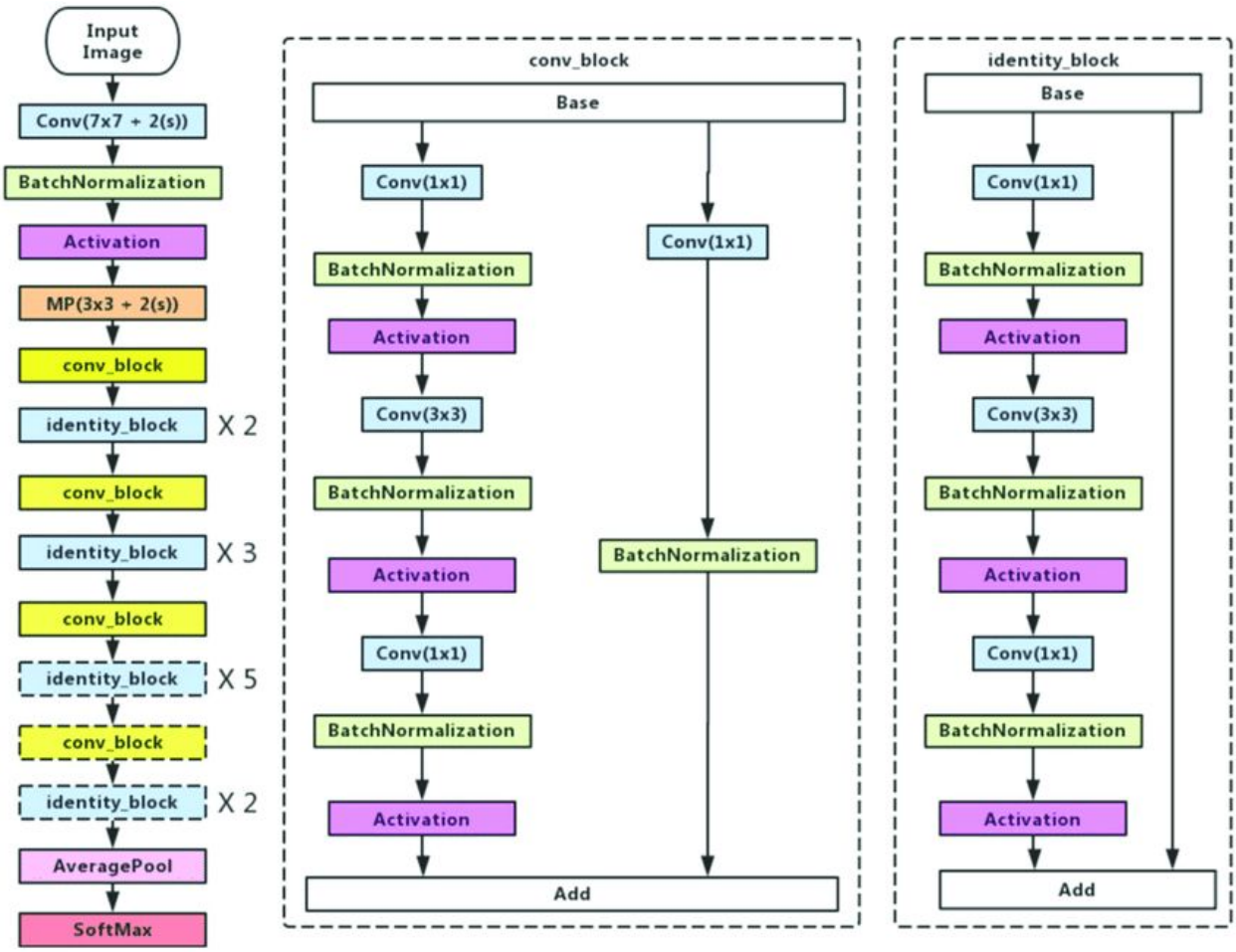
   Similar to the Convolutional Layer, the Pooling layer is responsible for reducing

the spatial size of the Convolved Feature. This is to decrease the computational power required to process the data through dimensionality reduction. Furthermore, it is useful for extracting dominant features which are rotational and positional invariant, thus maintaining the process of effectively training the model.


(1, 3, 224, 224)
(32, 3, 3, 3)
(32)
ThnnConv2DBackward
ReluBackward
MaxPool2DBackward
(64, 32, 3, 3)
(64)
ThnnConv2DBackward
ReluBackward
MaxPool2DBackward
(128, 64, 3, 3)
(128)
ThnnConv2DBackward
ReluBackward
MaxPool2DBackward
(500)
ViewBackward
(500, 6272)
ExpandBackward
DropoutBackward
TBackward
AddmmBackward
(133)
ReluBackward
(133, 500)
ExpandBackward
DropoutBackward
TBackward
AddmmBackward

2. ResNet50 Architecture

ResNet-50 is a convolutional neural network that is 50 layers deep. We can load a pre trained version of the network trained on more than a million images from the ImageNet database. The pretrained network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals.



ResNet50 Architecture : Creative Commons License

Source:
https://www.researchgate.net/figure/Left-ResNet50-architecture-Blocks-with-dotted-line-represents-modules-that-might-be_fig3_331364877

Benchmark

We used the thresholds provided by Udacity to benchmark our model.

We were required to obtain an accuracy of 10% and above for the scratch implementation and an accuracy of 60% and above for the transfer learning implementation.

We also compared our results with those available on Kaggle for a similar problem and used the 75th percentile result (77 % accuracy) as our benchmark.

3. METHODOLOGY

Data Preprocessing

Since most machine learning models require data in a certain format before they can begin analysing them, we need to preprocess data before feeding it to the model.

We also normalised and resized the data before feeding it into the network.

We performed data augmentation to increase robustness of our model. It can help to increase the amount of relevant data in your dataset. This is related to the way with which neural networks learn. A convolutional neural network that can robustly classify objects even if it is placed in different orientations is said to have the property called invariance. More specifically, a CNN can be invariant to translation, viewpoint, size or illumination (Or a combination of the above). This essentially is the premise of data augmentation. In the real world scenario, we may have a dataset of images taken in a limited set of conditions. But, our target application may exist in a variety of conditions, such as different orientation, location, scale, brightness etc. We account for these situations by training our neural network with additional synthetically modified data.

Finally we converted them to tensors as per demands of the models.

```
normalisation = transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
data_transformers = {'train': transforms.Compose([transforms.Resize(size=224),
                                    transforms.CenterCrop((224,224)),
                                    transforms.RandomHorizontalFlip(),
                                    transforms.ToTensor(),
                                    normalisation]),
                     'test': transforms.Compose([transforms.Resize(size=224),
                                    transforms.CenterCrop((224,224)),
                                    transforms.RandomHorizontalFlip(),
                                    transforms.ToTensor(),
                                    normalisation]),
                     'valid': transforms.Compose([transforms.Resize(size=224),
                                    transforms.CenterCrop((224,224)),
                                    transforms.RandomHorizontalFlip(),
                                    transforms.ToTensor(),
                                    normalisation])
                      }
```

Implementation

1. We use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images. OpenCV provides many pre-trained face detectors, stored as XML files on Github.

```
# 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
```

2. Here, we use a pre-trained ResNet-50 model to detect dogs in images. Our first line of code downloads the ResNet-50 model, along with weights that have been trained on ImageNet, a very large, very popular dataset used for image classification and other vision tasks. ImageNet contains over 10 million URLs, each linking to an image containing an object from one of 1000 categories. Given an image, this pre-trained ResNet-50 model returns a prediction (derived from the available categories in ImageNet) for the object that is contained in the image.

```
# define ResNet50 model
ResNet50_model = ResNet50(weights='imagenet')
```

```
### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    prediction = ResNet50_predict_labels(img_path)
    return ((prediction <= 268) & (prediction >= 151))
```

3. Then we created a model from scratch to classify our images

```
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, 32, 3, stride=2, padding=1)
        self.conv2 = nn.Conv2d(32, 64, 3, stride=2, padding=1)
        self.conv3 = nn.Conv2d(64, 128, 3, padding=1)

        self.pool = nn.MaxPool2d(2,2)

        self.fc1 = nn.Linear(128*7*7, 500)
        self.fc2 = nn.Linear(500, 133)

        self.dropout = nn.Dropout(0.3)

    def forward(self, x):
        ## Define forward behavior
        x = F.relu(self.conv1(x))
        x = self.pool(x)
        x = F.relu(self.conv2(x))
        x = self.pool(x)
        x = F.relu(self.conv3(x))
        x = self.pool(x)

        ## flatten
        x = x.view(-1, 7*7*128)
        x = self.dropout(x)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x
#-#-# You so NOT have to modify the code below this line. #-#-#

# instantiate the CNN
model_scratch = Net()
```

4. We then used a pretrained model for the same purpose of identification.

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

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

for param in model_transfer.parameters():
    param.requires_grad = False

model_transfer.fc = nn.Linear(2048, 133)

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

5. We wrote a algorithm to classify and give us the predicted output

- 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.

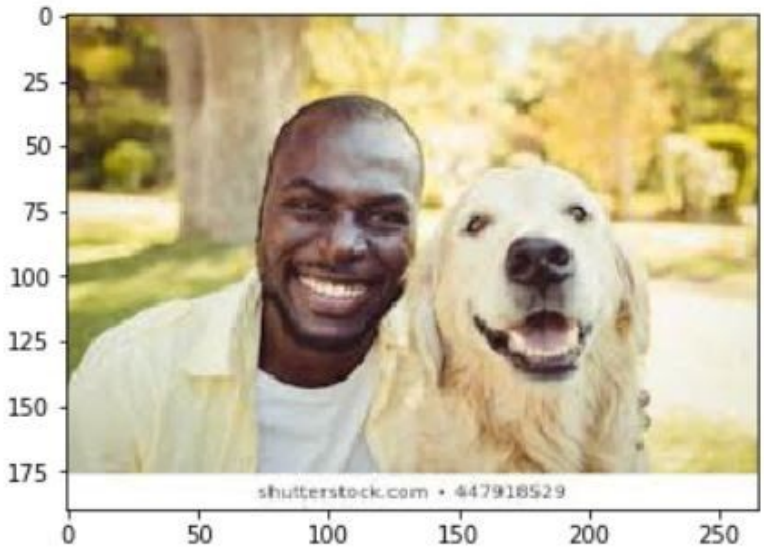
```
def run_app(img_path):
    ## handle cases for a human face, dog, and neither
    # detect human
    if (face_detector(img_path)):
        resemble_breed_index = predict_breed_transfer(img_path)
        temp_img = Image.open(img_path)
        plt.imshow(temp_img)
        plt.show()
        print(f' Hey! human! I am a {breed_names[resemble_breed_index][4:]}, you know, you kinda look like me')
    # 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(f'This collections of transistors says that I am a {breed_names[breed_index][4:]}, what do you think
    else:
        temp_img = Image.open(img_path)
        plt.imshow(temp_img)
        plt.show()
        print('Who let the dogs out!!!')
```

4. RESULTS

Model Evaluation and Validation

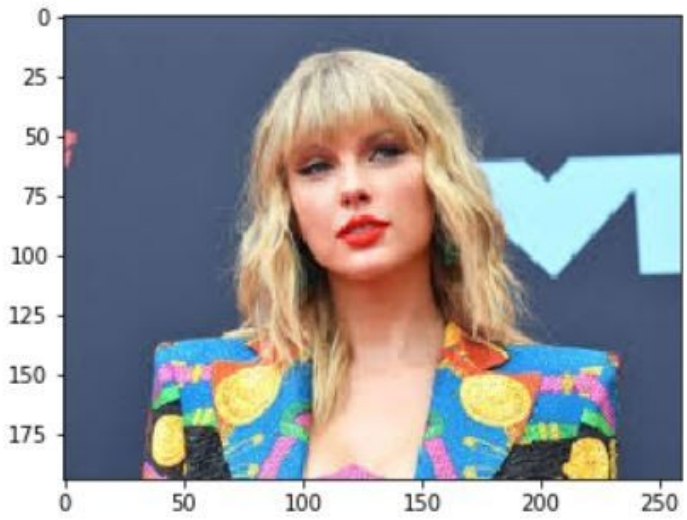
We take our new algorithm for a spin! What kind of dog does the algorithm think that a man looks like? If we have a dog, does it predict the dog's breed accurately? If we have a cat, does it mistakenly think that our cat is a dog?

```
#An image with both man and a dog.
run_app('my_dog_photos/mananddog.jpg')
```



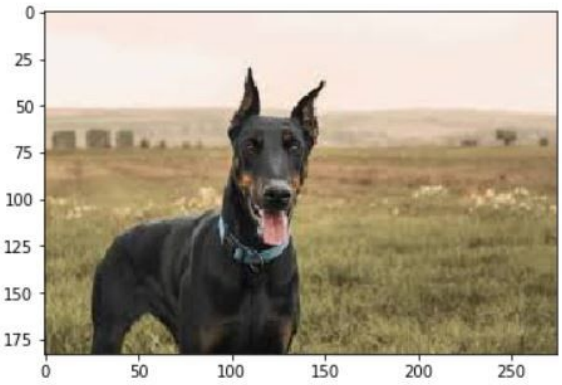

```
Hey! human! I am a Great_pyrenees, you know, you kinda look like me
```

```
#Taylor Swift
run_app('my_dog_photos/taylor.jpg')
```



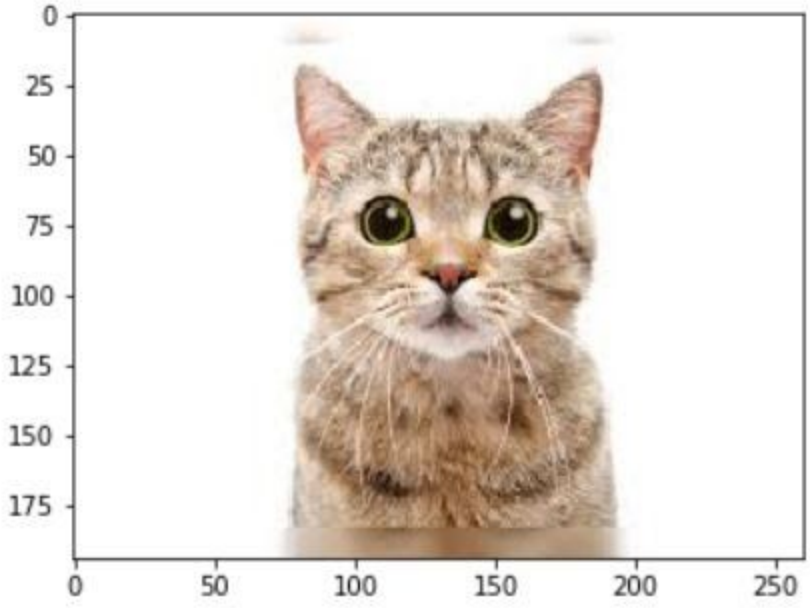

```
Hey! human! I am a Chinese_crested, you know, you kinda look like me
```

```
#Doberman
run_app('my_dog_photos/doberman.jpg')
```




```
This collections of transistors says that I am a Doberman_pinscher, what do you think, is it correct
```

```
#cat
run_app('my_dog_photos/cat.jpg')
```




```
Who let the dogs out!!!
```

Justification

The output accuracy is good given the small number of epochs the algorithms ran on, we can see that the training or validation error hadn't reached saturation yet. Simply training further might improve the accuracy. (I had to stop it at lower epochs because of time constraints and the fact that this workspace stops if left idle or connection is lost.

Other hyperparameters such as batch size can be tuned to further improve the accuracy

More dog images can increase the accuracy

5. CONCLUSION

The project was fun and packed with things to learn. Although we did good enough given the time constraints, we could also have looked more deeply into the images themselves that we used for training the dog breeds. We could have looked at a confusion matrix to see which images were giving the biggest errors in the validation data in order to identify possible noise. Maybe some of these images were too blurred and the model was generalizing well but being deceived by unclear images.

We could check the training images with random sampling to see the quality of the images and delete images that were badly focused or with more than one breed of dog, i.e. reduce noise in the training data.

We could check to see if there were sufficient training images of each breed of dog and that the image classes were balanced overall in terms of training numbers.

Following the above areas I'm sure we could increase the testing accuracy of the model to above 90%.

To summarise possible improvements are:

- analysis of images used to train and validate the model
- data augmentation
- fine tune hyperparameters

6. REFERENCES

https://docs.opencv.org/3.4/d7/d8b/tutorial_py_face_detection.html

https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

https://neurohive.io/en/popular-networks/vgg16/

https://towardsdatascience.com/an-overview-of-resnet-and-its-variants-5281e2f56035