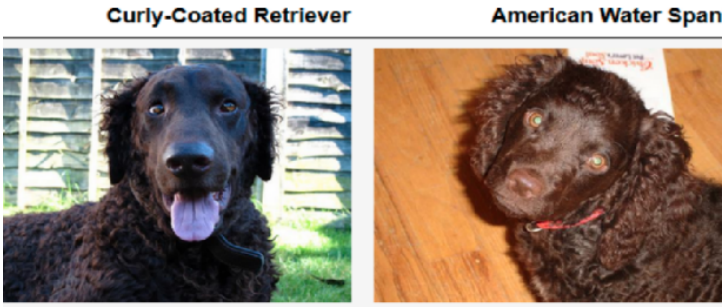
Machine Learning Engineer Nanodegree — Nilesh Sonawane

PROJECT REPORT — May 06, 2020

CNN Project: Dog Breed Classifier

Project Overview

Humans are excellent at vision, our brain is very powerful in visual recognition. Given a dog, one can easily detect it's breed the only condition is you be aware of all the dog breeds on this planet! Now this becomes a quite challenging task for a normal human. Consider you love a specific dog breed(say, labrador) and want to adopt a dog of the same breed, you go to shop and bring home your lovely new friend. How do you know if that's the correct dog breed you have got?? Many times it becomes hard for humans to identify the dog's breed. For example how about classifying the two dog images given below.



So, this becomes a tough job. Here is when the need for Machine Learning/Deep Learning comes into the picture. Computer Vision helps you build machine learning models where you train a model to recognize the dog breed! This makes your job easy! CNN was a huge invention in deep learning, this gave a boost to lots of applications in visual recognition. This project takes you through how to build a CNN from scratch and leverage some of the popular CNN architectures for out image classification task.

Problem Statement

Aim of the project was to build ML model that can be deployed and used within a web app to process real world, user-supplied images. Three main tasks:-

- *Human Face detector* - If given an image, model outputs True if Human face is detected, else returns False
- *Dog detector* - If given an image, model outputs True if dog is detected else outputs False.
- *Dog breed detector* - If given an image, model predicts the dog breed associated with that that image. If it detects dog, it will predict dog breed, if human face is detected, it predicts the dog breed that resembles with the human face.

Dataset and Inputs (Data Exploration)

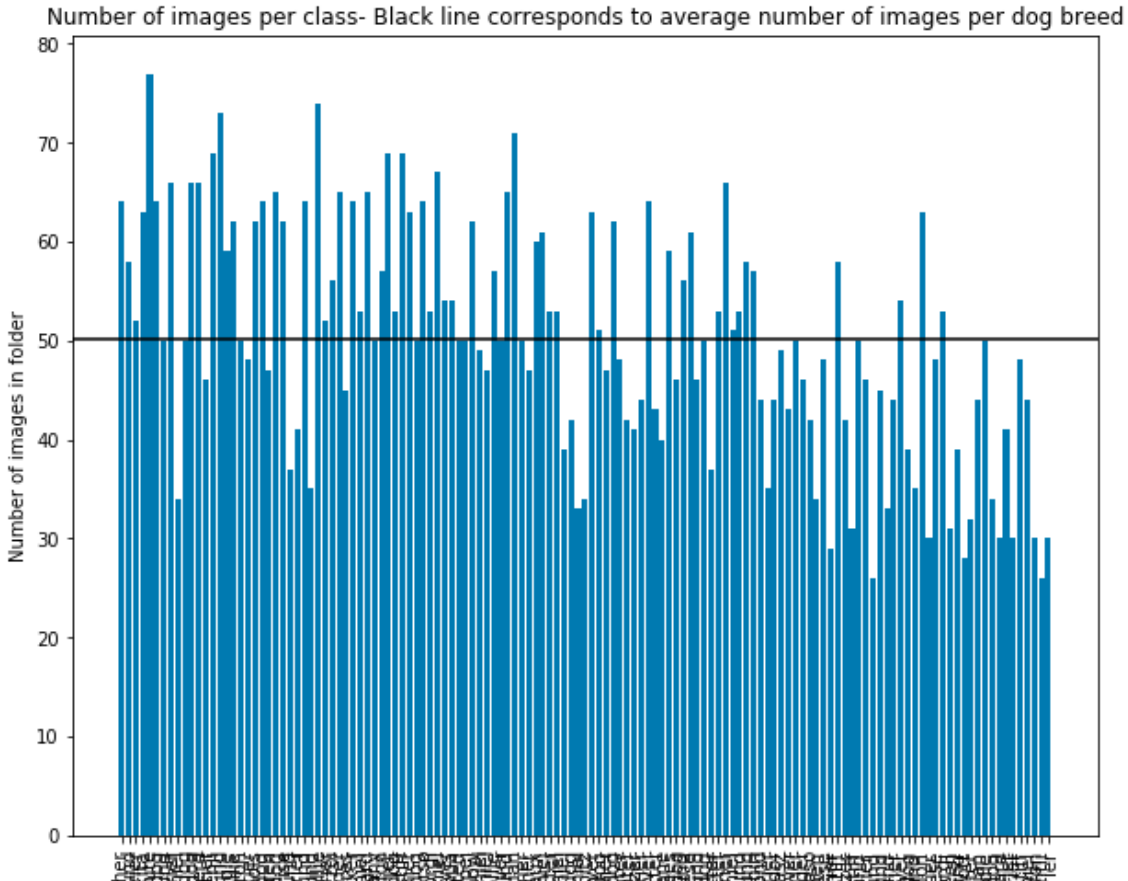
Inputs - Input type for this project must be an Image.
Dataset -

- ***Human images***: Human images are distributed in **5749 folders** named after human names like “Julianne_Moore”, “Dan_Ackroyd”, etc. There are a total of **13233 human face images**. Images are not evenly distributed among the folders. Link to the dataset is hyperlinked here.
- ***Dog images***: Dog images are distributed in 3 main folders named “train”,“test” and “valid” for training, testing, and validation respectively. Furthur all of these folders are again distributed into **133 folders** representing dog breeds. Hence our dog dataset has 133 classes ie.breeds(“Mastiff”, “Bichon_frise”, etc).

Information about the data-

- Total number of **human face images: 13233**
- Total number of **human face folders: 5749**
- Total numner of folders in 'dog_images:' 3
- Folders in 'dog_images': train,test,valid
- **Total folders(breed classes)** in 'train, test, valid': **133**
- Total images in /dog_images/**train : 6680**
- Total images in /dog_images/**test : 836**
- Total images in /dog_images/**valid : 835**

Distribution of Dog Breeds in training Dataset (average was 50.22 samples per class): Class names are not visible properly but we can see the average samples per class represented by a black line parallel to the class axis:



Model Evaluation Metrics

As we saw that our dataset is imbalanced, accuracy is not the correct model evaluation metric. Hence I chose to calculate average precision recall numbers for model evaluation. I have also calculated confusion matrics. Going ahead:

	Actual -- True/False	
Predicted -- Positive/Negative	True Positive	False Positive (Type I)
	False Negative (Type II)	True Negative

- Precision: TruePositives / (TruePositives + FalsePositives)
- Recall: TruePositives / (TruePositives + FalseNegatives)

Human Face Detection

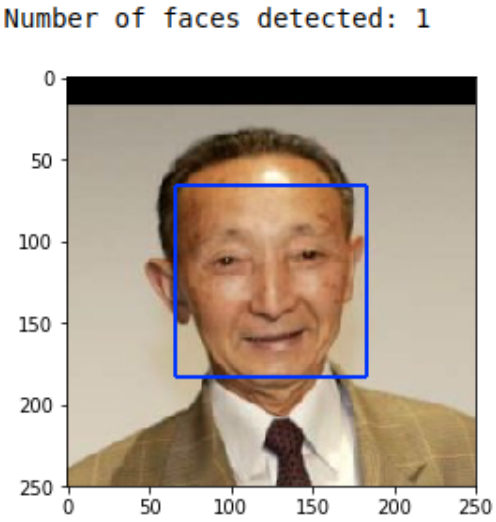
I used ***OpenCV's Haar Cascade Classifier*** to detect human faces. This OpenCV's classifier is trained on many images with positive points(with face) and negative points(without a face) labels. The detectMultiScale returns a list of 4 bounding box coordinate values for all detected faces in the same image. It is a standard practice to convert RGB image into a grayscale image for all the face detection algorithms, so make sure to convert your image.

```
import cv2
face_cascade = cv2.CascadeClassifier('haarcascade_frontalface.xml')
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  #returns bool
```

Model Performance: Haar Cascade Classifier has a good performance on data:

- Percentage of human faces detected in human images data: **98.74**
- Percentage of human faces detected in dog images data(incorrect detections): **10.83**

Sample ***result*** given by Haar Cascade Classifier:



Dog Detection

I have used transfer learning for detecting dog in a given picture. I tried to use the *VGG16 and ResNet50* pre-trained model, which are trained on *10M images of ImageNet data for 1000 classes*. I have downloaded this model from torchvision. The **VGG16 model** worked better on our dog dataset. We need to load and transform the image in the required format(eg. image size, convert to RGB, normalize data). This *"load_transform_image" function is also used during testing our final workflow for one image*. Also made a dog detector that returns *"True" if a dog is detected* in the image passed to this function. Following is the code:

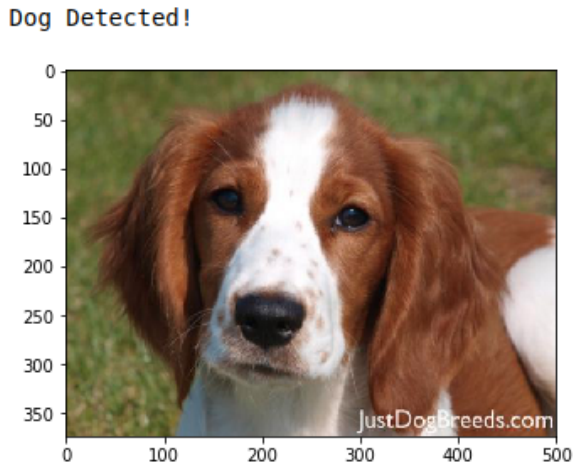
```
import torch
from PIL import Image
import torchvision.transforms as transforms
import torchvision.models as models
# define VGG16 model
VGG16 = models.vgg16(pretrained=True)
# check if CUDA is available
use_cuda = torch.cuda.is_available()
# move model to GPU if CUDA is available
if use_cuda:
    VGG16 = VGG16.cuda()
def load_transform_image(img_path):
    '''
    Used load & transform image for prediction on single 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])
    img_transform = transforms.Compose([
                        transforms.Resize(size=(224, 224)),
                        transforms.ToTensor(),
                        normalize])
    img = img_transform(img)[:3,:,:].unsqueeze(0)
    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
    '''
    image = load_transform_image(img_path)
    if use_cuda:
        image = image.cuda()
    output = VGG16(image)
    return torch.max(output,1)[1].item()
def dog_detector(img_path):
    prediction = VGG16_predict(img_path)
    return (prediction>=151 and prediction<=268)
```

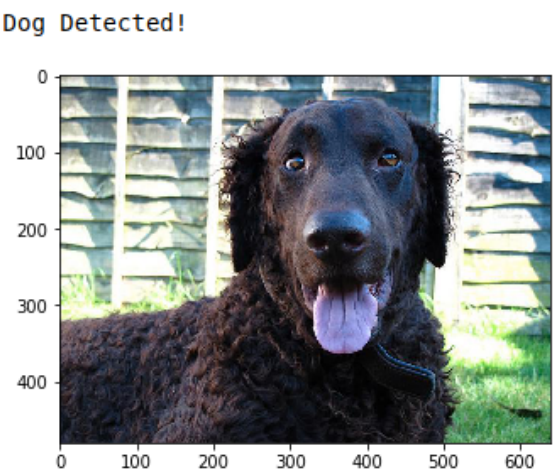
Model Performance: to save time, we can get the detection done for first 100 images in both datasets.

- Percentage of dogs detected in human image data(incorrect detections): 1.0%
- Percentage of dogs detected in dog image data: 100.0%

Different trained architectures like Inception-v3, GoogleNet, etc can be used.

Sample **results** from *dog_detector:*





Data Loader

We need to prepare our data for training the model. I have performed data normalization, splitting, and arranging data in the required format for training and testing our model! Data augmentation is an important factor while training your NN, this adds more robustness to your model and makes it learn from different variations of our data. Basically data augmentation helps in adding variation to our data. Given is the code for generating data loader.

```
import os
from torchvision import datasets
import torchvision.transforms as transforms
data_dir = '/data/dog_images/'
train_dir = os.path.join(data_dir, 'train/')
valid_dir = os.path.join(data_dir, 'valid/')
test_dir = os.path.join(data_dir, 'test/')
normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],std=[0.229, 0.224, 0.225])
preprocess_data = {'train': transforms.Compose([transforms.RandomResizedCrop(224),
                                                transforms.RandomHorizontalFlip(),
                                                transforms.ToTensor(),normalize]),
                   'valid': transforms.Compose([transforms.Resize(256),
                                                transforms.CenterCrop(224),
                                                transforms.ToTensor(),normalize]),
                   'test': transforms.Compose([transforms.Resize(size=(224,224)),
                                               transforms.ToTensor(), normalize])}
train_data = datasets.ImageFolder(train_dir, transform=preprocess_data['train'])
valid_data = datasets.ImageFolder(valid_dir,transform=preprocess_data['valid'])
test_data = datasets.ImageFolder(test_dir, transform=preprocess_data['test'])
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
}
```

Thus now we have ***train_loader***, ***valid_loader***, and ***test_loader*** with data stored in batches along with defined preprocessing and data augmentation steps. Explanation for steps performed in data preparation:

- I have used ***RandomiResizedCrop*** for *training data* which resized all the training images to (224,224), it also makes a random crop of the original image so that our model is able to learn complex variations in data. I have flipped the data horizontally using ***RandomHorizontalFlip***(p=.5) which will flip half images horizontally to add more variation to original training data. Almost all the images in the train folder are straight(dogs are aligned in a straight manner) so using a horizontal flip and not using a vertical flip was more sensible. I have also normalized all the channels in images using ***standard Normalization***.
- Used ***CenterCrop*** of size (224,224) for *validation data*, as most of the images have a dog face in the center thus this will help in good validation accuracy!
- ***Resized*** *test images* to (224,224), no other transformation done here as we will test our model on raw data

Dog Breed Classification

For this task I have built *CNN from scratch in Pytorch*. Here I created a ***3-layer CNN with Relu activation***. Using different kernel sizes, strides, padding, and Max-Pooling for each layer, the size of the original image (224,224) has been reduced to (7,7) and the original depth of 3 has been transformed to 128: (224,224,3) -> (7,7,128). In this way we have extracted the spatial features from a given image! We increase the depth or add more filters so that the network can learn more significant features in the image and **generalize better**.

Code for building model from scratch

```
import torch.nn as nn
import torch.nn.functional as F
# define the CNN architecture
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # Conv Layers
        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)
        # maxpool
        self.pool = nn.MaxPool2d(2, 2)
        # fc layers
        self.fc4 = nn.Linear(7*7*128, 2048)
        self.fc5 = nn.Linear(2048, 512)
        self.fc6 = nn.Linear(512, 133)    #number of classes = 133
        # dropout
        self.dropout = nn.Dropout(0.25)    #dropout of 0.25
        # batchNorm layers
        self.batch_norm = nn.BatchNorm1d(512)

    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.fc4(x))
        x = self.dropout(x)
        x = F.relu(self.batch_norm(self.fc5(x)))
        x = self.dropout(x)
        x = self.fc6(x)
        return x
# instantiate the CNN
model_scratch = Net()
```

Used **Cross-Entropy Loss** as a **cost function** and **Adam** to be the **optimizer,** after training for ***50 epochs*** with *batch size 20* on train data located in the dog image dataset, I got `***Training Loss: 4.1504***` and `***Validation Loss: 3.7211***`

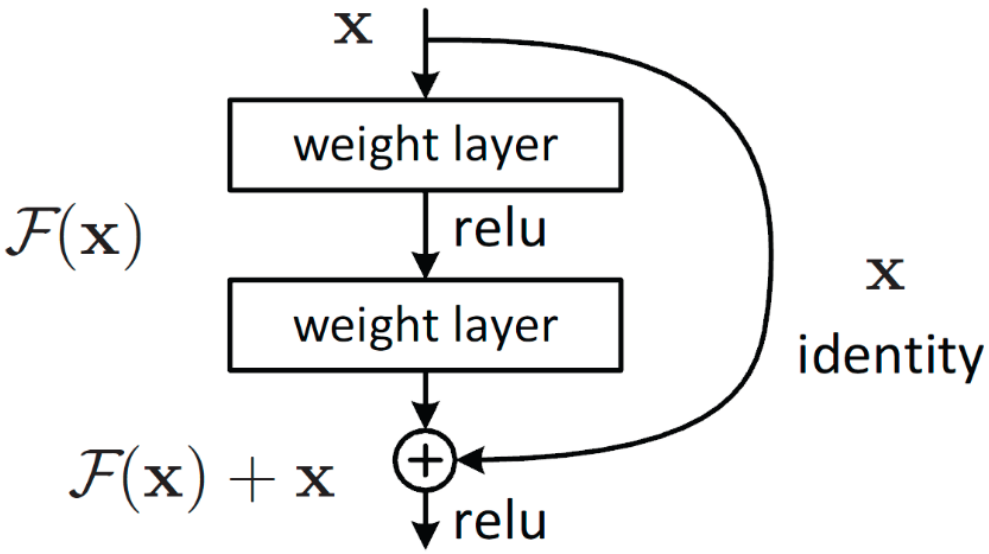
Model ***Performance*** of this model (*model_scratch*) on *test data*:

- Test Loss: 3.820199
- Test Accuracy: **11%** (89/836)

Next, **transfer learning** is used to accomplish the same task! I chose ***ResNet101*** for this multi-class classification task.

Reason for selecting ResNet:

- It is widely known that ResNets give outstanding performance in image classification. ResNet is built with "***Residual Blocks***" which are basically the skip connections between layers. This creates an identity connection between initial layers to the final layers, reducing the risk of vanishing /exploding gradient problem that also helps in reduced risk of underfitting and overfitting on training data! Resnet101 is a 101 layers deep Neural network hence capturing granular spatial information from the image. CNN gives better feature representation for each input image and residual blocks generate the identity connections which makes gradient flow easy, and thus ResNet helps to boost up classification task.
- Also I have **removed** the **last fully-connected layer** of pre-trained Resnet and added our *custom* fully-connected at the last which is intended to output ***133 sized vector***.



Residual Block

Model Architecture:

```
import torchvision.models as models
import torch.nn as nn
model_transfer = models.resnet101(pretrained=True)
for param in model_transfer.parameters():
    param.requires_grad = False
#replacing last fc with custom fully-connected layer which should output 133 sized vector
model_transfer.fc = nn.Linear(2048, 133, bias=True)
#extracting fc parameters
fc_parameters = model_transfer.fc.parameters()
for param in fc_parameters:
    param.requires_grad = True
```

After training for just ***20 epochs***, *batch size 20* on train data located in the dog image dataset, I got `***Training Loss: 1.4732***` and `***Validation Loss: 0.9333***`. If trained for more epochs loss will reduce and the model will learn as much possible from data!

Model Performance on test data: I chose *Precision* and *Recall* numbers as our evaluation metrics. This is because our data had an imbalance, precision-recall always gives us a good overview of model performance!

```
from sklearn.metrics import confusion_matrix, precision_recall_fscore_support
cm = confusion_matrix(ground_truths, predictions)
precision = np.mean(precision_recall_fscore_support(ground_truths, predictions)[0])
recall = np.mean(precision_recall_fscore_support(ground_truths, predictions)[1])
```

- ***Precision:*** *0.8039343488*
- ***Recall:*** *0.78137904284*

These numbers talk a lot about our breed classifier! **0.8 precision** indicates that our model is predicting the results with the precision on 80% that means our model is 0.8 precise(***80% of the prediction are correct***), which is really a good thing! ***Recall of 0.78*** is not bad considering that I have trained the model on *20 epochs only*. These evalution numbers can be boosted by training the model for *more number of epochs* and changing few other characteristic mentioned in improvements section.

Dog App

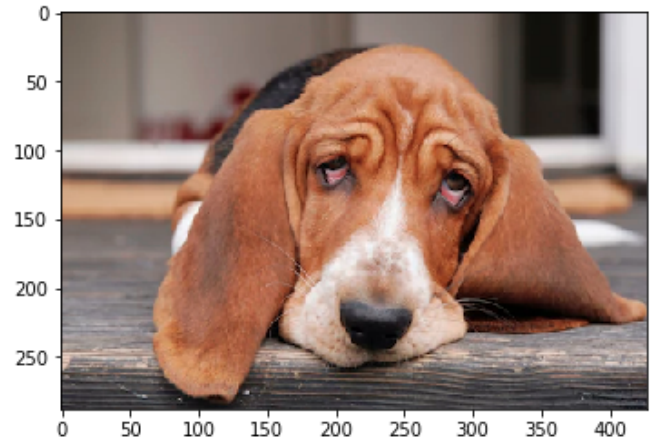
Now we finally make a user ready dog app function that can process real world user supplied images to detect wheather the image contains human or dog and then furthur resulting in dog breed predicted by the model!

This app or the entire workflow can be used to make a web application too.

Results

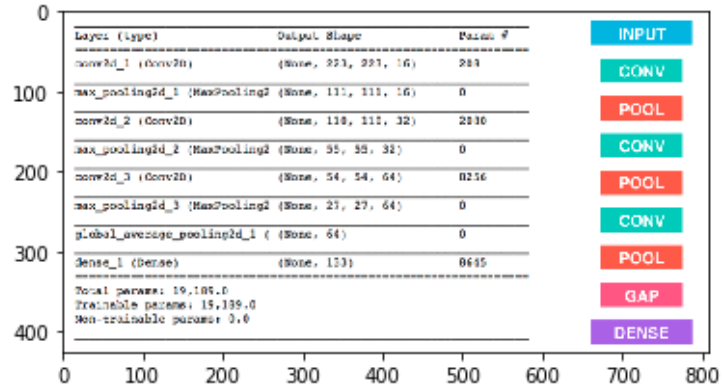
Given below are some interesting results produced by our workflow. All the images are *my inputs* which are *not the part of training data*!

Dog Detected!



This is a "Basset hound" kind of dog!

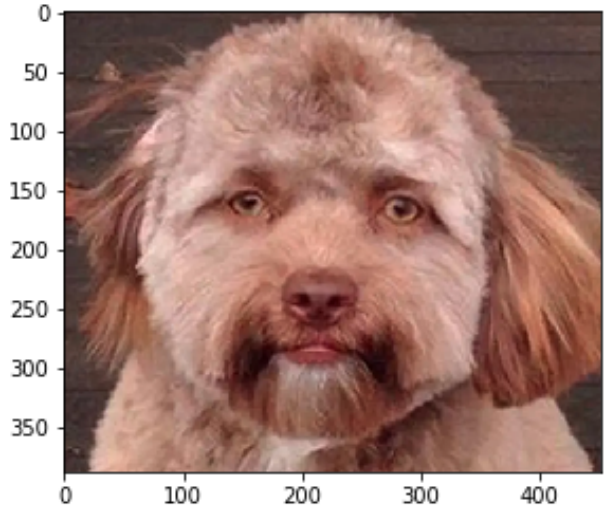
Ooops! Nothing to detect!



Dog Detected!

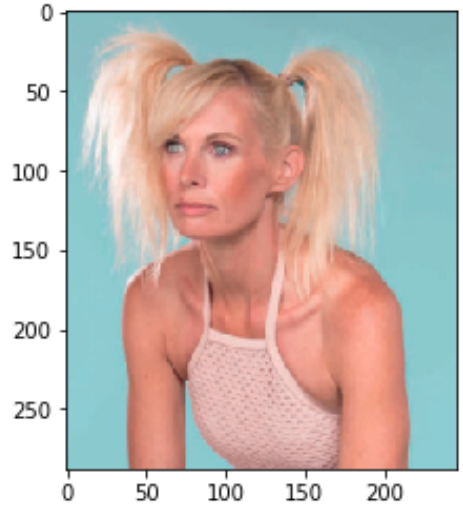
This is a "Chinese shar-pei" kind of dog!

Dog Detected!



This is a "Havanese" kind of dog!

Human Detected!



Dog Detected!

This is a "Poodle" kind of dog!

Interesting, this human looks like "Chinese crested"!

Improvements

There is still a huge scope of improvement given below are the few points that can be considered at this moment:

- Training on ***more data*** can help in this case, also augmentation of the data may help like cropping images to correct areas may help.
- To have perfectly labeled and more data in dog_images/train will surely help. Also Manually adding some variation(***noise***) to the data such as adding false images or images of humans that look like dogs in the human dataset.
- ***Hyperparameter*** tuning always helps! :)
- Using a more ***powerful pre-trained model*** may also help!

Conclusion

I got to explore the data on a huge scale. Playing with data and creating a model from scratch was interesting! CNN is an important invention in the field of Machine Learning. One major point to notice was how modern deep learning frameworks have made our work easy, we can train our model in a handful(very few) lines of code. Used PyTorch, which is a great programming framework! Got to learn how transfer learning can help us in our application.

Another interesting thing to notice is how a well-trained model(machine) sometimes gets even better than humans at generalizing the features.

References

1. OpenCV's Haar cascades classifier for face detection:
   - https://docs.opencv.org/trunk/db/d28/tutorial_cascade_classifier.html
2. Pre-trained Pytorch models(Torchvision.model) :
   - https://pytorch.org/docs/master/torchvision/models.html
3. ImageNet: http://www.image-net.org/
4. Precision and Recall: https://towardsdatascience.com/beyond-accuracy-precision-and-recall-3da06bea9f6c
5. CNN: https://cs231n.github.io/convolutional-networks/
6. Original Project Repo: https://github.com/udacity/deep-learning-v2-pytorch/tree/master/project-dog-classification
7. Resnet101: https://arxiv.org/abs/1512.03385