Machine Learning Engineer Nanodegree

Capstone Project

Marco Viscardi
February 5th, 2020

I. Definition

Project Overview

Convolutional Neural Networks (CNN) project with the goal of understanding the challenges involved in piecing together a series of models designed to perform various tasks in a data processing pipeline.
I chose this project because I´m very interested in machine vision and thus in improving my knowledge and experience with CNN.

Problem Statement

Machine vison project developing a pipeline that can be used within a web or mobile app 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's breed.
Both detection problem (detecting humans and dogs) and classification problem (identifying dog's breed) will be solved using CNN models (three different models): one model for detecting humans, one model for detecting dogs and one model for classifying dog's breed or identify the resembling dog's breed. This last model will be trained using the given dataset.

Metrics

The evaluation metric key parameter is the image classification accuracy on the test set (the models are trained and selected minimizing the validation loss at each epoch).
This is a standard approach when training CNN with large image datasets where there is no category of specific images with a dominant incidence and the variety of the datasets is thus wide enough not to enter in an accuracy paradox.

The CNN will be also tested with random images not coming from the original dataset provided by Udacity.

II. Analysis

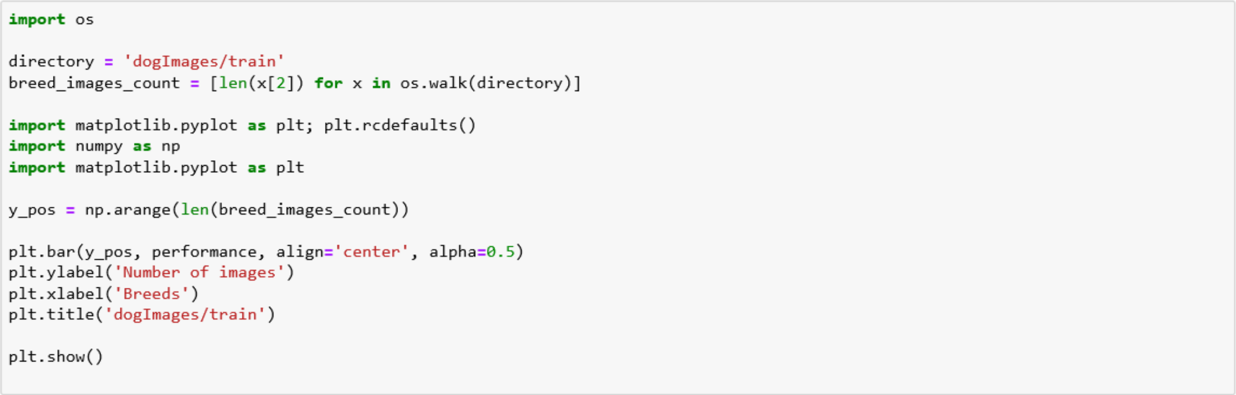
Data Exploration

The dataset provided by Udacity consists in:

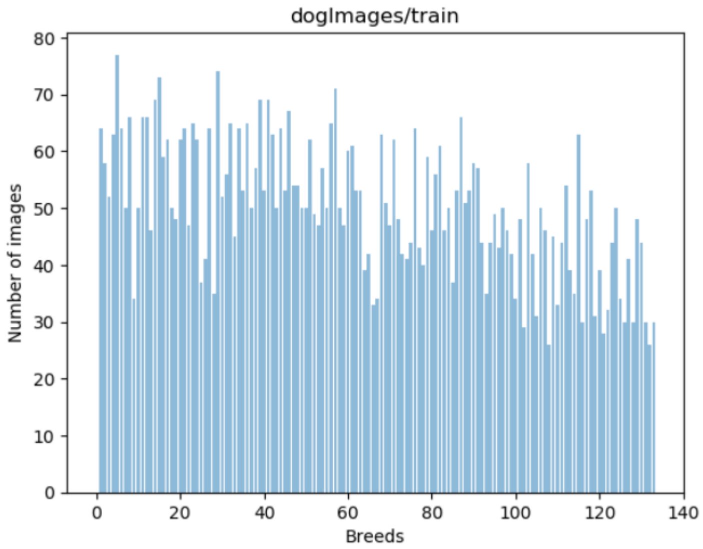
- One folder (dogImages) containing 8.351 pictures of 133 different dog´s breeds divided in test, train and validation datasets.

- A second folder (lfw) containing 13.234 pictures of 5.749 different people.

The following graph is showing that, as expected, there is no category of specific images with a dominant incidence within our dogImages dataset:

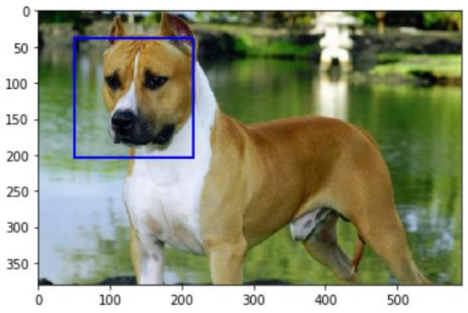


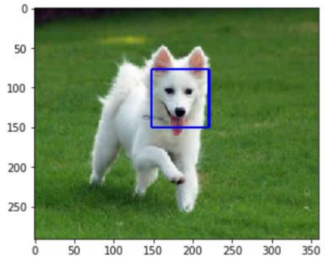
```
import os

directory = 'dogImages/train'
breed_images_count = [len(x[2]) for x in os.walk(directory)]

import matplotlib.pyplot as plt; plt.rcdefaults()
import numpy as np
import matplotlib.pyplot as plt

y_pos = np.arange(len(breed_images_count))

plt.bar(y_pos, performance, align='center', alpha=0.5)
plt.ylabel('Number of images')
plt.xlabel('Breeds')
plt.title('dogImages/train')

plt.show()
```

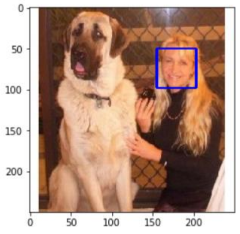


Exploratory Visualization

As visible in the examples here below the dogs' pictures can have different sizes. Image preprocessing/standardization is thus needed prior using the dataset for training the networks. Humans could also be present within the dogs' pictures.





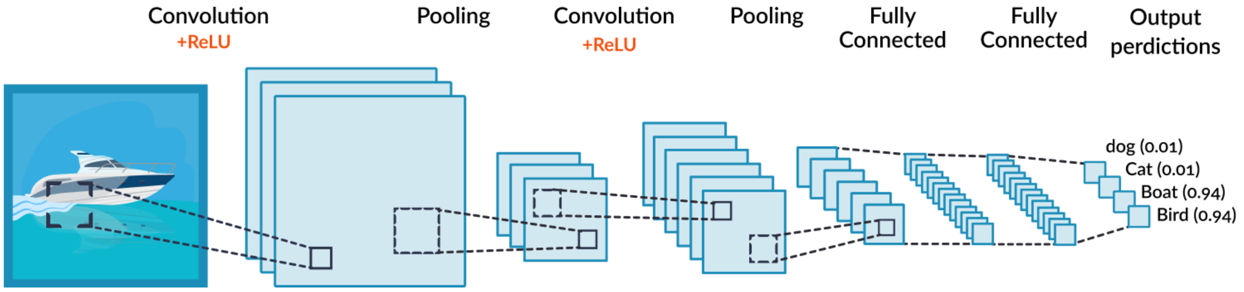


Algorithms and Techniques

For the detection problem (detecting the presence of humans or dogs within the picture) existing pre-trained models and algorithms can be used such as OpenCV face_cascade classifier (for human detection) and VGG16 CNN (for dogs detection).

The dog´s breed classification problem will instead require to develop and train a feed-forward classifier using transfer learning and the dogs images dataset provided by Udacity. The pre-trained model of choice is Resnet50 but also other pre-trained models available in Pytorch could be used (VGG16, etc…).
As a learning experience also a completely new CNN will be developed and trained on the dogs' dataset with the aim of achieving at least 10% accuracy on the test dataset.

A CNN architecture gradually takes the spatial data (original pixel values) and converts the array to a representation that instead encodes the content of the image (final class scores).
As shown here below it does that mainly applying in alternate sequence a series of Convolutional and Pooling filters.



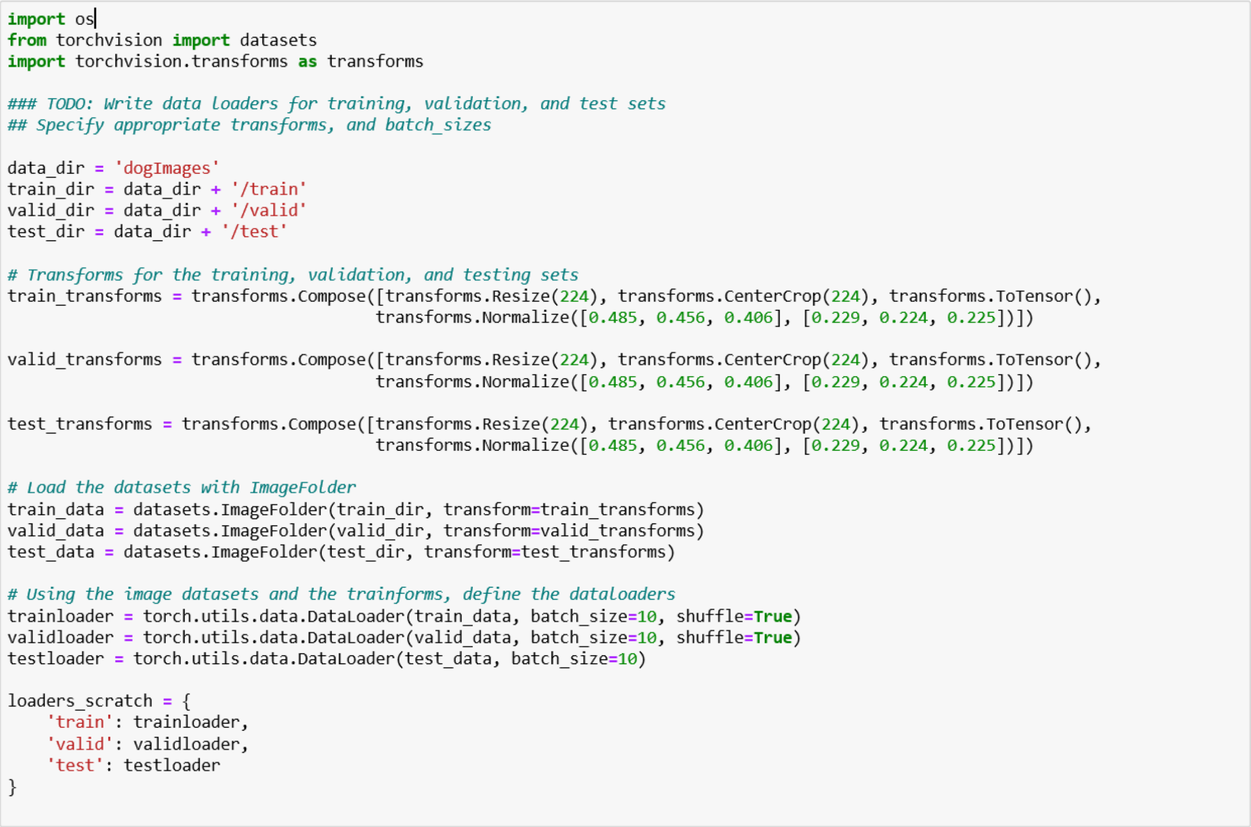
Convolutional filters are responsible for finding patterns in the image. This means that the higher the number of object categories the higher the number of required convolutional filters (bigger stack with high dimensionality). Higher dimensionality means more parameters, which could lead to overfitting. Pooling layers are thus used to reduce the dimensionality.

Benchmark

The feedforward classifier generated from Resnet50 is our benchmark model as Resnet50 was already pretrained with a large image dataset including dogs and humans.
Transfer learning is providing Resnet50 with additional capabilities on our specific dogs' dataset granting that the accuracy on the test set is 60% or greater (vs at least 10% accuracy of the completely new CNN).

III. Methodology

Data Preprocessing



```
import os
from torchvision import datasets
import torchvision.transforms as transforms

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

data_dir = 'dogImages'
train_dir = data_dir + '/train'
valid_dir = data_dir + '/valid'
test_dir = data_dir + '/test'

# Transforms for the training, validation, and testing sets
train_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])])

valid_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])])

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])])

# Load the datasets with ImageFolder
train_data = datasets.ImageFolder(train_dir, transform=train_transforms)
valid_data = datasets.ImageFolder(valid_dir, transform=valid_transforms)
test_data = datasets.ImageFolder(test_dir, transform=test_transforms)

# Using the image datasets and the trainforms, define the dataloaders
trainloader = torch.utils.data.DataLoader(train_data, batch_size=10, shuffle=True)
validloader = torch.utils.data.DataLoader(valid_data, batch_size=10, shuffle=True)
testloader = torch.utils.data.DataLoader(test_data, batch_size=10)

loaders_scratch = {
    'train': trainloader,
    'valid': validloader,
    'test': testloader
}
```

All Pytorch pre-trained models expect input images normalized in the same way, i.e. mini-batches of 3-channel RGB images of shape (3 x H x W), where H and W are expected to be at least 224. The images have to be loaded into a range of [0, 1] and then normalized using mean = [0.485, 0.456, 0.406] and std = [0.229, 0.224, 0.225].

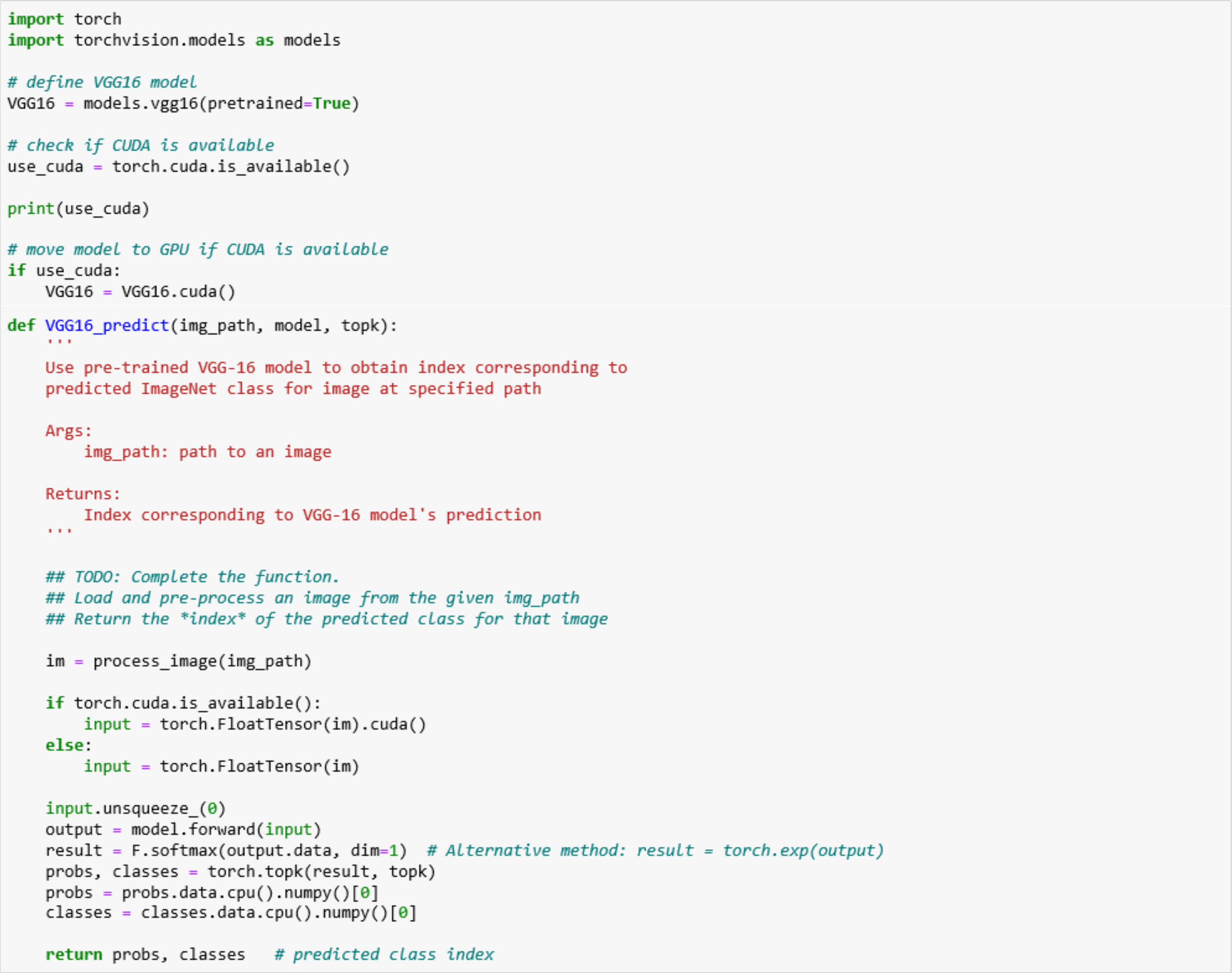
I´ve thus applied all these transformations in order to generate a dataloader suitable for both pre-trained models (when using transfer learning) and new models.

Dataset augmentation transformations such as random scaling, cropping, and flipping help the network generalize leading to better performance. The validation and testing sets are anyway used to measure the model's performance on data it hasn't seen yet. For this reason no scaling or rotation transformations should be applied, but it´s anyway needed to resize and then crop the images to the appropriate size. I´ve done just that on the full dataset.

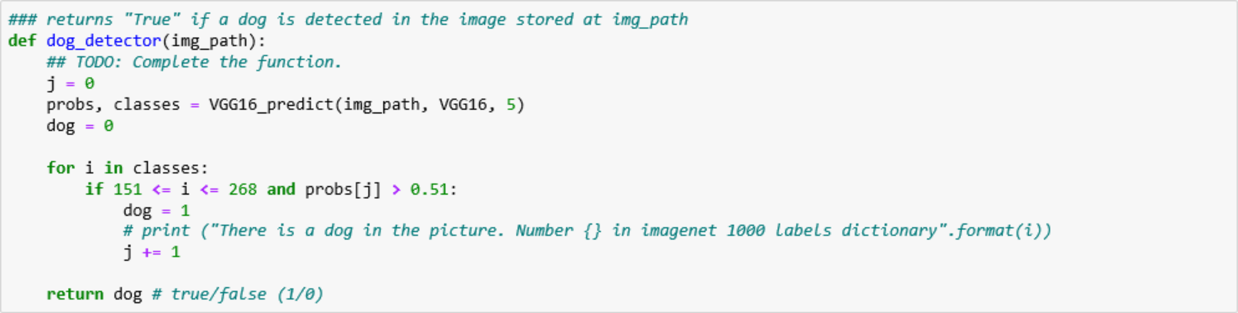
Implementation

I started developing the required dog_detector and face_detector functions using the existing pre-trained VGG16 CNN model (for dogs detection) and OpenCV face_cascade classifier (for humans detection).

Dog_detector:

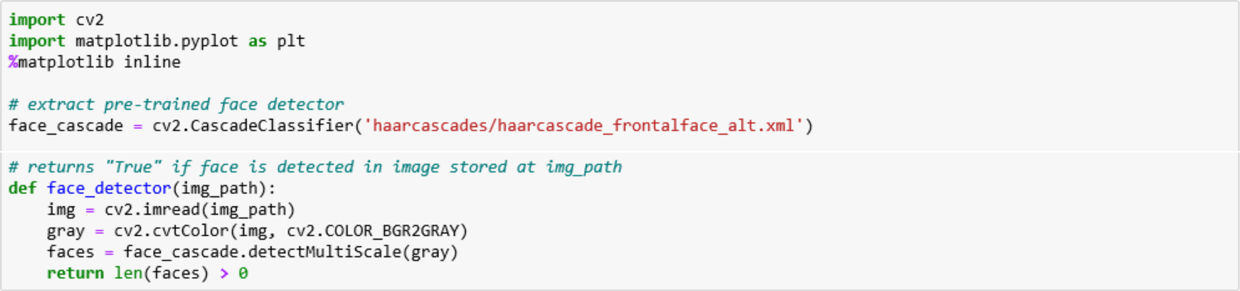


```
import torch
import torchvision.models as models

# define VGG16 model
VGG16 = models.vgg16(pretrained=True)

# check if CUDA is available
use_cuda = torch.cuda.is_available()

print(use_cuda)

# move model to GPU if CUDA is available
if use_cuda:
    VGG16 = VGG16.cuda()
def VGG16_predict(img_path, model, topk):
    '''
    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

    im = process_image(img_path)

    if torch.cuda.is_available():
        input = torch.FloatTensor(im).cuda()
    else:
        input = torch.FloatTensor(im)

    input.unsqueeze_(0)
    output = model.forward(input)
    result = F.softmax(output.data, dim=1)  # Alternative method: result = torch.exp(output)
    probs, classes = torch.topk(result, topk)
    probs = probs.data.cpu().numpy()[0]
    classes = classes.data.cpu().numpy()[0]

    return probs, classes   # predicted class index
```



```
### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    ## TODO: Complete the function.
    j = 0
    probs, classes = VGG16_predict(img_path, VGG16, 5)
    dog = 0

    for i in classes:
        if 151 <= i <= 268 and probs[j] > 0.51:
            dog = 1
            # print ("There is a dog in the picture. Number {} in imagenet 1000 labels dictionary".format(i))
            j += 1

    return dog # true/false (1/0)
```

Face_detector:



```
import cv2
import matplotlib.pyplot as plt
%matplotlib inline

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

```
# 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
```

I then started developing the new CNN architectures for the dog´s breed classification problem.
I found a 2014 study from the University of Toroto (by Turing Award winners Geoffrey Hinton and Yoshua Bengio): "Dropout: A Simple Way to Prevent Neural Networks from Overtting" In this study they describe the following CNN architecture tested with good results on The Street View House Numbers (SVHN) DataSet (color images of house numbers collected by Google Street View):

"*...The best architecture that we found uses three convolutional layers each followed by a max-pooling layer. The convolutional layers have 96, 128 and 256 filters respectively. Each convolutional layer has a 5 × 5 receptive field applied with a stride of 1 pixel. Each max pooling layer pools 3 × 3 regions at strides of 2 pixels. The convolutional layers are followed by two fully connected hidden layers having 2048 units each. All units use the rectified linear activation function. Dropout was applied to all the layers of the network with the probability of retaining the unit being p = (0.9, 0.75, 0.75, 0.5, 0.5, 0.5) for the different layers of the network (going from input to convolutional layers to fully connected layers). In addition, the max-norm constraint with c = 4 was used for all the weights. A momentum of 0.95 was used in all the layers. These hyperparameters were tuned using a validation set. Since the training set was quite large, we did not combine the validation set with the training set for final training. We reported test error of the model that had smallest validation error...*"

I thus started developing a similar architecture for my completely new CNN.

In parallel I started the development and training of the benchmark feed-forward classifier based on the Resnet50 pre-trained CNN model following these steps:

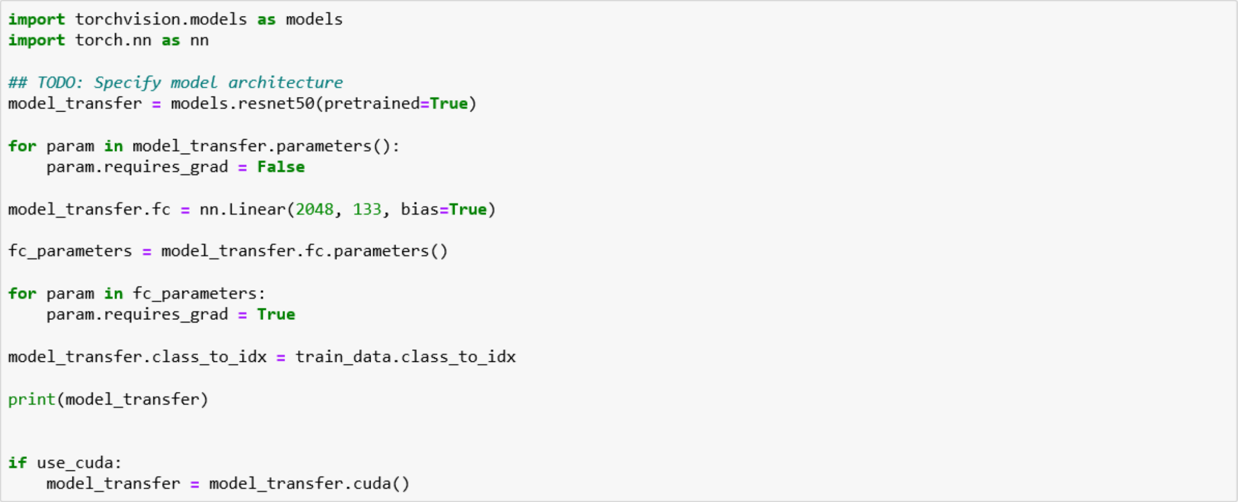
- ➢ Loading of the pre-trained Resnet50 network
- ➢ Definition of a new feed-forward network adding a new feedforward classifier to the existing Resnet50 convolutional layers.
- ➢ The parameters of the feedforward classifier are appropriately trained, while the parameters of the feature network (Resnet50) are left static.

Refinement

With my preliminary new CNN model I could not reach the expected results in terms of accuracy on the test dataset (at least 10% accuracy). By trial and error I simplified the model reaching this final solution implemented here below. It has the same number of convolutional and max pooling layers, same number of filters (with different dimensions but same strides) and I have applied dropout only on the fully connected layers (still with probability = 0.5).
In order to avoid running out of CUDA memory I also had to change the dataloaders batch_size parameter to 10 (I started with 64).

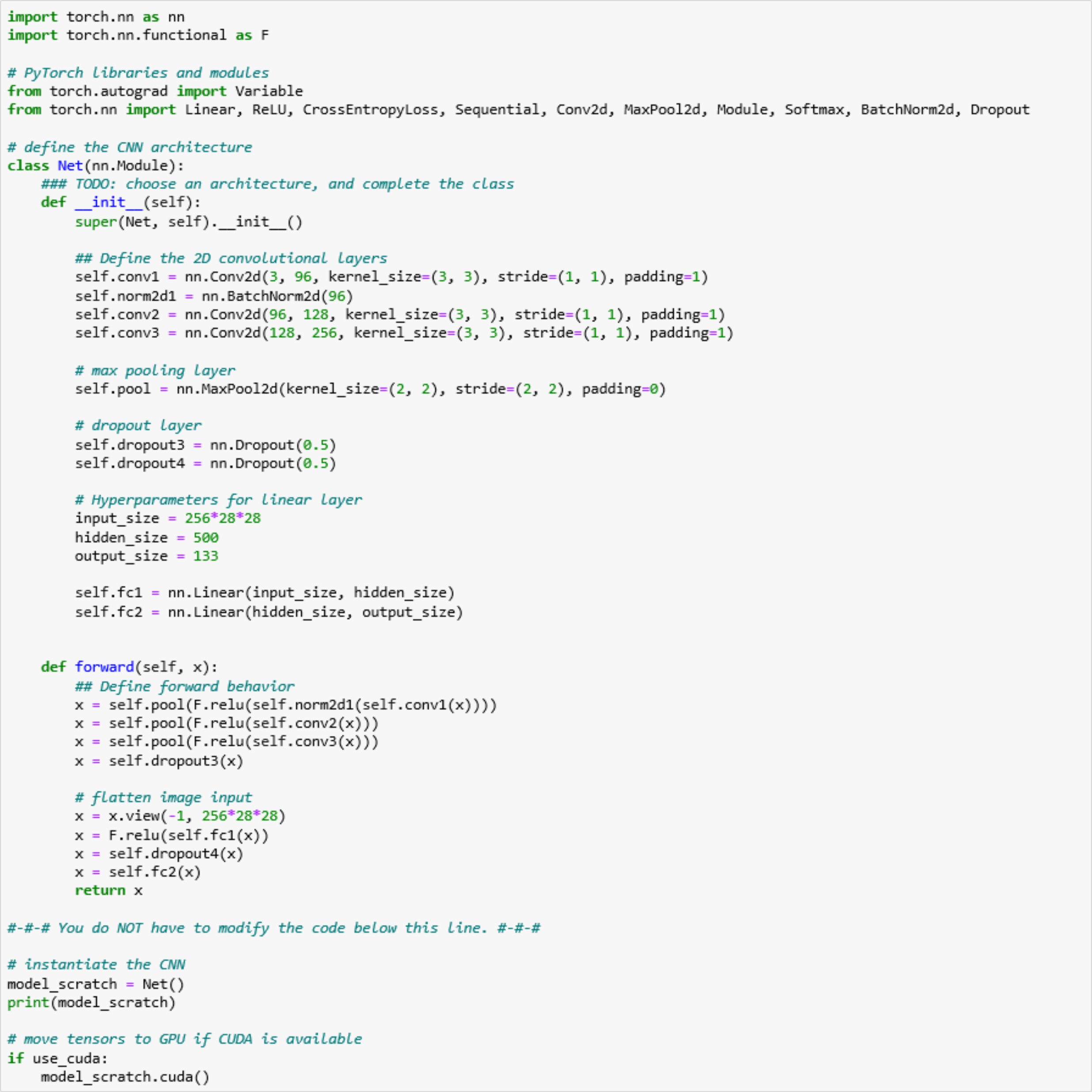
The benchmark feed-forward classifier based on the Resnet50 pre-trained CNN model was instead performing well from the beginning adding just one new linear layer after the Resnet50 convolutional layers (81% accuracy on the test dataset). No refinement activities were needed. Here below the implementation.

New feed-forward classifier based on Resnet50:



```python
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, bias=True)

fc_parameters = model_transfer.fc.parameters()

for param in fc_parameters:
    param.requires_grad = True

model_transfer.class_to_idx = train_data.class_to_idx

print(model_transfer)


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

New CNN made from scratch:

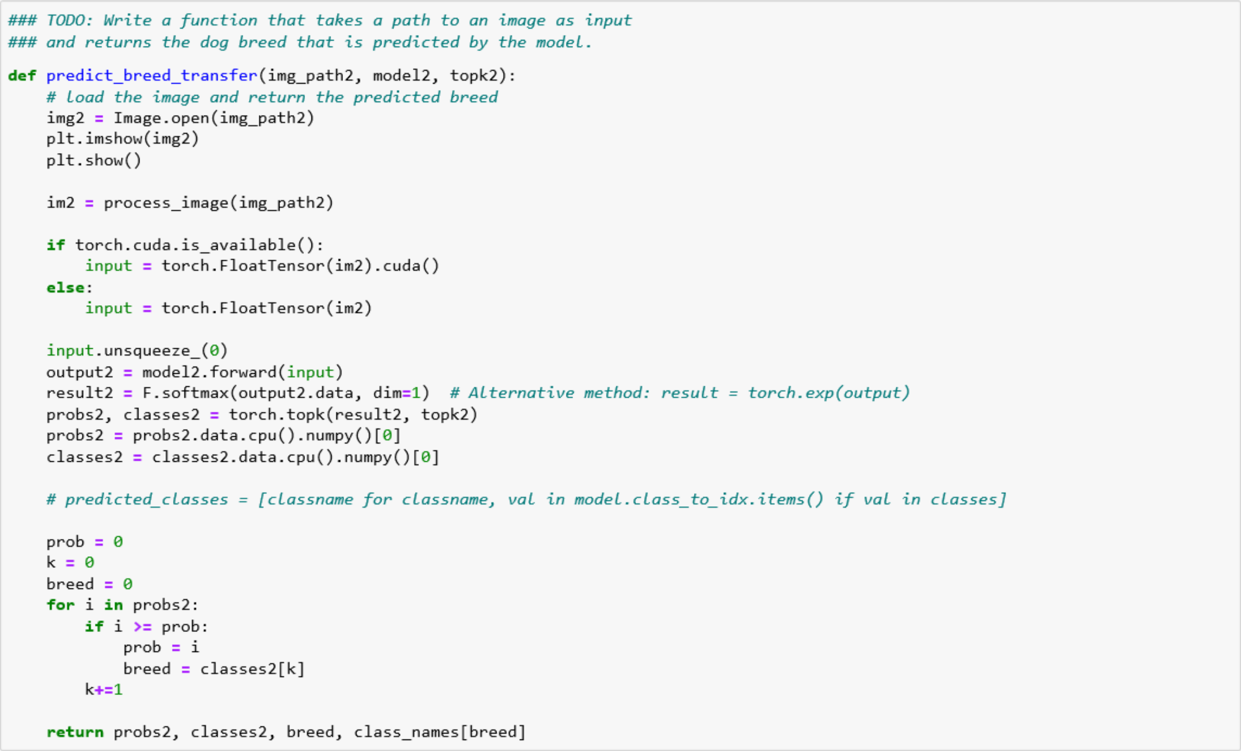


```
import torch.nn as nn
import torch.nn.functional as F

# PyTorch libraries and modules
from torch.autograd import Variable
from torch.nn import Linear, ReLU, CrossEntropyLoss, Sequential, Conv2d, MaxPool2d, Module, Softmax, BatchNorm2d, Dropout

# define the CNN architecture
class Net(nn.Module):
    ### TODO: choose an architecture, and complete the class
    def __init__(self):
        super(Net, self).__init__()

        ## Define the 2D convolutional layers
        self.conv1 = nn.Conv2d(3, 96, kernel_size=(3, 3), stride=(1, 1), padding=1)
        self.norm2d1 = nn.BatchNorm2d(96)
        self.conv2 = nn.Conv2d(96, 128, kernel_size=(3, 3), stride=(1, 1), padding=1)
        self.conv3 = nn.Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=1)

        # max pooling layer
        self.pool = nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2), padding=0)

        # dropout layer
        self.dropout3 = nn.Dropout(0.5)
        self.dropout4 = nn.Dropout(0.5)

        # Hyperparameters for linear layer
        input_size = 256*28*28
        hidden_size = 500
        output_size = 133

        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, output_size)


    def forward(self, x):
        ## Define forward behavior
        x = self.pool(F.relu(self.norm2d1(self.conv1(x))))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = self.dropout3(x)

        # flatten image input
        x = x.view(-1, 256*28*28)
        x = F.relu(self.fc1(x))
        x = self.dropout4(x)
        x = self.fc2(x)
        return x

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

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

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

IV. Results

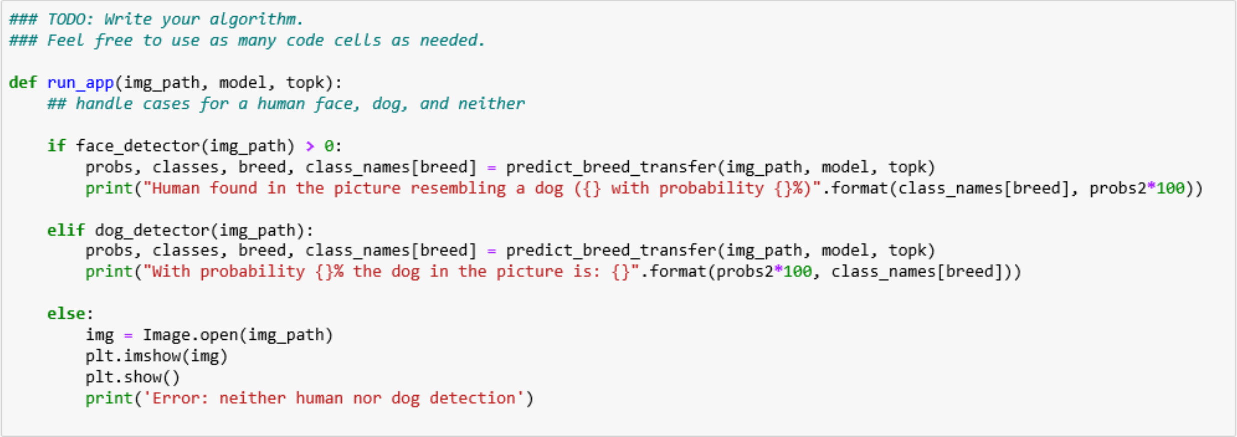
Model Evaluation and Validation

I then developed the function predict_breed_transfer for the identification of an estimate of the canine's breed (it will be used with the feed-forward classifier based on Resnet50).



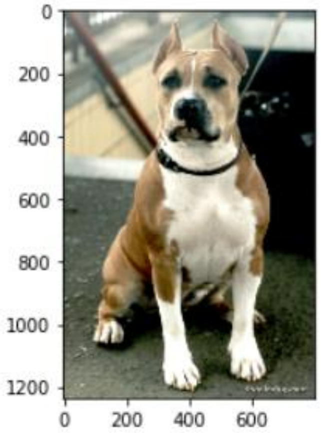
```python
### TODO: Write a function that takes a path to an image as input
### and returns the dog breed that is predicted by the model.

def predict_breed_transfer(img_path2, model2, topk2):
    # load the image and return the predicted breed
    img2 = Image.open(img_path2)
    plt.imshow(img2)
    plt.show()

    im2 = process_image(img_path2)

    if torch.cuda.is_available():
        input = torch.FloatTensor(im2).cuda()
    else:
        input = torch.FloatTensor(im2)

    input.unsqueeze_(0)
    output2 = model2.forward(input)
    result2 = F.softmax(output2.data, dim=1)  # Alternative method: result = torch.exp(output)
    probs2, classes2 = torch.topk(result2, topk2)
    probs2 = probs2.data.cpu().numpy()[0]
    classes2 = classes2.data.cpu().numpy()[0]

    # predicted_classes = [classname for classname, val in model.class_to_idx.items() if val in classes]

    prob = 0
    k = 0
    breed = 0
    for i in probs2:
        if i >= prob:
            prob = i
            breed = classes2[k]
        k+=1

    return probs2, classes2, breed, class_names[breed]
```

Last function to be developed was the run_app algorithm which is combining all the functions previously developed in order reach the requested app functionality (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).

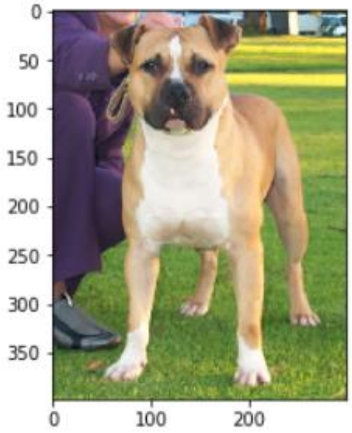


```python
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.

def run_app(img_path, model, topk):
    ## handle cases for a human face, dog, and neither

    if face_detector(img_path) > 0:
        probs, classes, breed, class_names[breed] = predict_breed_transfer(img_path, model, topk)
        print("Human found in the picture resembling a dog ({} with probability {}%)".format(class_names[breed], probs2*100))

    elif dog_detector(img_path):
        probs, classes, breed, class_names[breed] = predict_breed_transfer(img_path, model, topk)
        print("With probability {}% the dog in the picture is: {}".format(probs2*100, class_names[breed]))

    else:
        img = Image.open(img_path)
        plt.imshow(img)
        plt.show()
        print('Error: neither human nor dog detection')
```

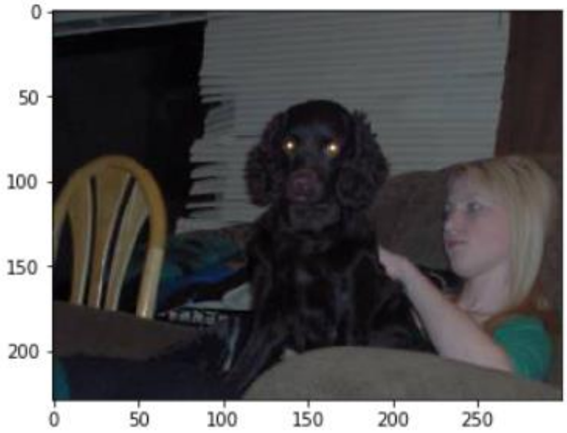
Here following there are some results obtained passing random pictures from the given dataset.



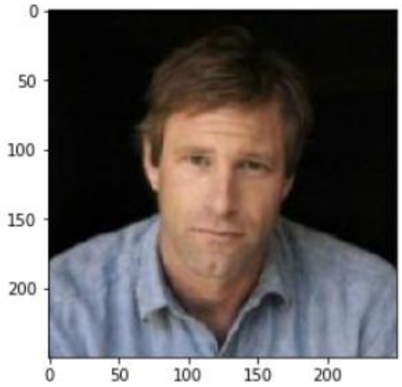
```
With probability [62.111004]% the dog in the picture is: American staffordshire terrier
```



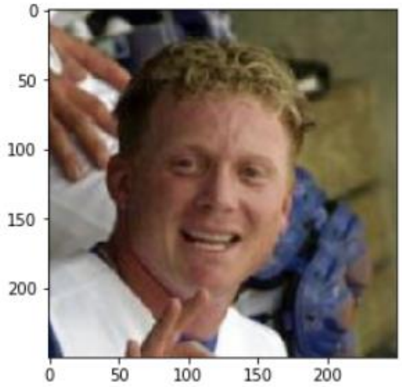
```
With probability [61.240685]% the dog in the picture is: American staffordshire terrier
```



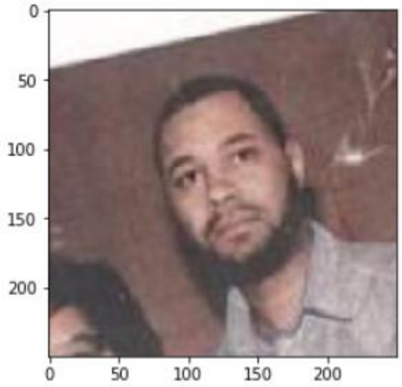
```
With probability [4.707436]% the dog in the picture is: Boykin spaniel
```



```
Human found in the picture resembling a dog (French bulldog with probability [17.257223]%)
```



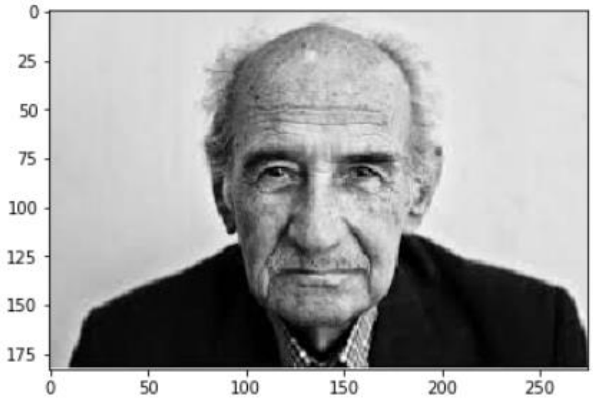
```
Human found in the picture resembling a dog (American water spaniel with probability [17.257223]%)
```



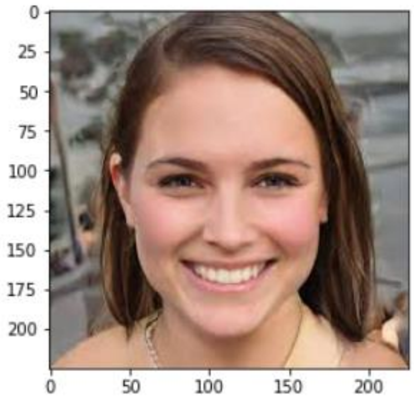
```
Error: neither human nor dog detection
```

The final model and algorithm seem to be reasonable and aligning with solution expectations but the performances are not perfect as demonstrated by the results in pictures 3 and 6 here above. The model is having more uncertainty or it´ s even failing when the image is blurry or containing disturbance elements (such as a human together with a dog).

The model shows acceptable performances also with unseen data (still some failures when disturbance elements are introduced):



```
Human found in the picture resembling a dog (Neapolitan mastiff with probability [17.257223]%)
```



```
Human found in the picture resembling a dog (English toy spaniel with probability [17.257223]%)
```

```
With probability [17.257223]% the dog in the picture is: French bulldog
```

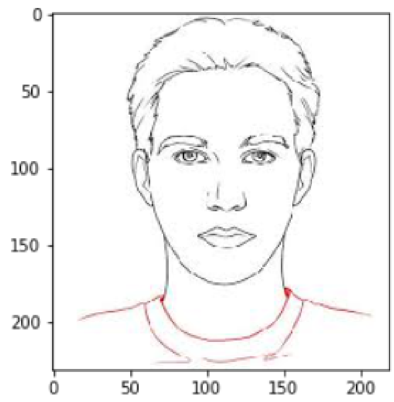
```
Error: neither human nor dog detection
```

```
Error: neither human nor dog detection
```

V. Conclusion

Free-Form Visualization

The algorithm seems to be able to detect humans also when simple drawings (with few graphic features) instead of pictures are provided:



```
Human found in the picture resembling a dog (English toy spaniel with probability [17.257223]%)
```

Also the CNN model seems to be able to identify the resembling dog breed with a similar confidence level as from previous examples.

Reflection

This project is demonstrating how powerful CNN architectures are in solving machine vision problems. It´s not easy to develop and put together all the models and functions in a working pipeline but in the end the results are quite impressive.

I´m quite surprised by the performances that can be achieved with transfer learning using just few lines of code and a “rather small” training/validation datasets (and training time).

This is encouraging me in continuing my learning path with CNN and computer vision topics.

Improvement

The algorithm works well in classifying dog´s pictures and creates funny suggestions for the dog´s breeds that are mostly resembling the human pictures. There are still anyway some imprecisions (as expected).

Three possible points for improvement could be:

- expanding the training and validation datasets with more pictures and re-train the network. The higher the quantity of data the better the outcome of the training.

- including within the dataset also "difficult" pictures where the dog is not in a natural position or where other disturbance factors may lead to a wrong classification. Edge cases are of course the most difficult to be identified.

- training another feedforward classifier (using a different pre-trained model and transfer learning) and then run the two networks in parallel comparing their outcomes prior issuing a final classification result.