Sebastian Firlik sfirlik@olx.pl

Dog classification project - final report.

1. Definition of the problem

For my capstone project I chose dog breed classification project. It was one of the projects, which were available in Udacity repo. This project was already started from the beginning -- I was given an initial version of Jupyter Notebook with this project with tips of how to organize my work and what should be the next steps in the process of project development.

The final application, which should be the outcome of this capstone project was human face detector and dog breed classification. In the first step the application should try to detect human face in the image. There are many ready-to-use face detectors, which even don't use Convolutional Neural Network to detect them, they just use some feature extractors or edge detectors, which are considered to be classical image processing methods. The next step would be to try to find a dog in the image, so one can use any neural network which was trained on ImageNet dataset. There are many dog classes and if an input image falls into any of these dog-related classes, it is then treated as a dog photo. If there is no human face and no dogs in the picture, the application should let us know that there is an error with the photo. In the other case, app should return predicted dog breed for either human face ("this human face resembles eg. labrador") or dog ("this dog is probably rottweiler").

The dataset used in this project consists of two parts:

a) Human faces dataset - it has 13233 human faces photos from both genders, some younger, some older. Examples of photos from this dataset will be shown in the next section.

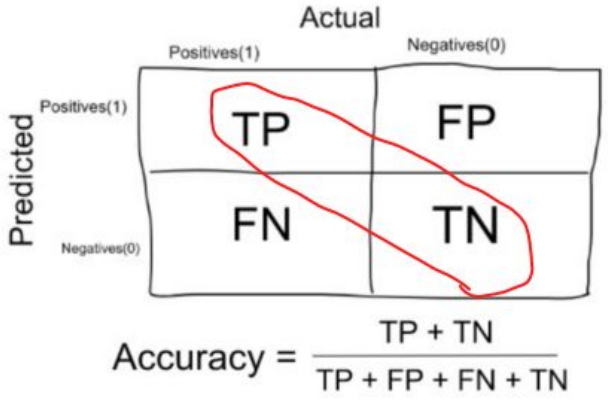
b) Dogs photos dataset - it has 8351 photos of dogs from 133 different breeds. Small analysis of this dataset, biggest and smallest classes, division into train, validation and test sets will be provided in section 2.

For both human face detector and dog detector there are many approaches. Firstly, I will describe possible solutions for human face detectors:

a) Knowledge-based approaches - in this type of algorithms we have a set of rules which indicate whether there is a face in the image or not.

So a typical face has a nose, ears, mouth and eyes. If we have all of these detected then it's probably a face in the image. I didn't try to implement this kind of face detector, because it would be too hard, too complicated and probably would require a lot of fine tuning, which is always bad if you want to do something quickly.
b) Feature-bases approaches - in this type of methods the algorithm tries to extract regions, which probably contain a face and then try to assess for example if this region is face-shaped.
c) Template matching approaches - this type of face detectors rely on a database of face pictures (or only some limited number of face templates) and tries to find correlation between the template and part of the image, which contains face. These methods are often inaccurate.
d) Appearance-based methods - this is the most modern type of face detectors. It tries to find a face by its appearance. These methods mainly consist of Machine Learning methods, or statistical approaches. CNNs are often finding faces by finding image regions, which look like faces.
Dog detectors are not used very often, so there are no popular and classical solutions to this problem. Instead, there are these appearance-based approaches, so Machine Learning methods.
The same goes for dog breed classification - this problem was not important in many areas, so the best way to classify dog by its breed is to learn a Convolutional Neural Network.

The most common metric in classification problems is just accuracy of the classifier.



source:
https://miro.medium.com/max/1064/1*5XuZ_86Rfce3qyLt7XMlhw.png

It is defined as number of correctly recognized examples divided by the population. If we only have positive examples, we can just divide number of detections by number of photos, so we will get percentage of proper classifications for dataset with just one class. Thi metric is also extendable to problems with many classes, like 133 dog breeds in this project.

2. Analyze the problem

As it was stated in the previous section, human face dataset consists of 13233 photos. These photos are grouped inside directories one directory per one person. Images can present only face, or sometimes the face can be partially covered, what makes it harder to detect it. Below are two examples of images from this dataset.

As far as I checked the human face dataset, most of the images contain clear shot of full face, so it was mostly very clean and proper to use them as input to neural network, or just use a ready solution from OpenCV library.
Dogs images were available in two datasets - some of them were part of ImageNet dataset - neural networks, which I tried to use for the transfer learning point of the project were always taught on ImageNet dataset.

In the ImageNet dataset, some photos, like these above, representing Husky dog are of different quality - some from zoom in, some from a bigger distance. It might be helpful to generate some characteristics of a breed, like, Husky

dog is mostly black with white snout.


The images from Stanford Dog Dataset were similar to the ones from ImageNet, but included 133 different dog breeds. Some of the photos were also zoomed in, so one couldn't see the whole dog, just his head, or a bit zoomed out, like the one on the right.
The classes were mostly balanced, so even though there are 133 different races, none of them is really dominating any others, so metric like accuracy should be sufficiently good for this classification problem.
The biggest class is Alaskan Malamute, which has 96 images. It is just a little more than 1% of the whole dataset, so this dataset is considered very balanced. The smallest class is the mexican hairless dog, named Xoloitzcuintli. It contains only 33 images, which is more than 33% of the biggest class -- this ratio is very good. The biggest downside of this dataset is that these classes are all very small and some of the dog breeds are very similar to each other. Such small classes are often impossible to be learnt, because neural network can't generalise a lot from 40 examples. Usually, neural network learnt from scratch would require several hundreds of examples per class to be useful in classification. But thanks to pretrained networks and much bigger ImageNet dataset, it was possible to learn a network like VGG16 to predict proper dog breed from an image of a dog.

3. Implement the algorithm

There was Jupyter Notebook, provided by Udacity team where I had this project outlined. There was a clear flow and I could implement this project step by step, but with only a small help from the tips between cells and questions, which I had to answer. It was a very pleasant way of implementing the dog breed classification algorithm.
The first step was to find, how many images are there in both human face dataset, as well as dogs dataset.
Then, there was a step of building a human detector. The default solution to this task was to use Haar feature based face detector, which was working pretty well. My task was to see, how well does this face detector work by

testing it on first 100 images from human faces dataset and dog dataset (in the ideal world it should return 100 faces in human dataset and 0 faces in dog dataset). After conducting this test I was able to tell that this algorithm made only one mistake on human dataset, so it found 99 faces, but made also 18 mistakes on dog dataset. It detected 18 human faces on photos of dogs.

The photo above is the only one, that face detector couldn't mark correctly. It might be the fault of the blue glasses, because that's something unnatural and usually, people don't have blue glasses.

The next step was to build a dog classifier. In this exercise I was told to use some pretrained network, because ImageNet dataset's classes between index 151 and 258 inclusive were classes with dog breeds. I used VGG16 network, which was proved multiple times to be a very good network for various classification tasks and was reasonably good on ImageNet dataset. There are some newer and better in terms of loss function or accuracy networks, but VGG16 is used very widely and it was a very good choice for a starting point for a dog detector function.

Next step was to make a function, which will get image path as input parameter and that will return index of predicted class. I could use this function to classify all the images in a given folder without using DataLoaders. I could also use it to test how well is the network working with my own local images. The function was predicting the image class, by firstly reading the image file, then normalizing it, resizing and unsqueezing, just as VGG16 input should look like. Then I was told to write a function which will return True if class label is in range of dog breeds and False otherwise.

The third section of the Jupyter Notebook was about creating a Convolutional Neural Network from scratch to predict dog's breed. The first step in building such a solution is to load all the images from dog dataset as DataLoaders. It is a very convenient way of defining datasets: I provided path to train set and this structure automatically inferred what's the correct class for all the images. There is also an option to define pipeline of transformations to the dataset, including data augmentation, resizing and converting to PyTorch tensor

format. There is also an option to shuffle the data, to specify batch_size, and number of workers (so processes). This step of defining the DataLoaders was also the step of preprocessing the data. For the train loader I used RandomResizedCrop with size 224, so it was cropping the training image with size 224x224. Thanks to that, there were not only beautiful pictures, but also a bit distorted, or cropped with a weird angle. Then I used RandomHorizontalFlip, so there was a chance that the image will be flipped horizontally. Dog flipped like that is still a dog, so it was a good way to augment the data. Then I converted the image to a PyTorch Tensor and normalized it, according to the tutorial for training networks. In this way I was able to normalize the data, so I could use it for both transfer learning and learning CNN from scratch. For valid loader I resized the images to 256x256 and then cropped 224x224 in the center of the image. Then I also converted it to Tensor and normalized in the same way. In test loader I just resized the images to 224x224 and then converted to Tensor and normalized.

Next section was a task of building own model architecture. I tried many different architectures, but mostly they were wrong and couldn't learn anything more than 4% of accuracy on test set. I tried a CNN with three convolutional layers, one with 16 filters, then 32 and then back to 16 filters, but this network was too simple for this task. Then I tried to use five conv layers, but this one was learning for a very long time and wasn't tuned well. Then I tried to use dropout on two final, Linear layers (especially on the first one, the second one couldn't have dropout because it was output layer). I specified too hight probability of dropping a neuron, which was 0.7 and the network couldn't learn anything once again. Then I dropped the number of conv layers to three, I made max pooling after each one and used batch normalization so the network won't overfit to training set. I used dropout on the first Linear layer, but with probability of 0.3 and this time the results were very nice and satisfying the minimal accuracy needed to pass the exercise, which was 10%. For the training to succeed, I had to define loss function and optimization algorithm. I chose loss function, which is very popular for multiclass classification problems, so CrossEntropyLoss. It is widely used so I didn't change it between approaches. As for the optimizing algorithm, I tried to use Adam algorithm, which is usually the default choice, due to the fact that designer shouldn't be forced to tune the hyperparameters of this algorithm, but it failed in the first approach and then I changed it to Stochastic Gradient Descent algorithm. I tried to use it with Nesterov's momentum with momentum rate of 0.9 but it didn't make it work better, so I dropped this idea and sticked to classical SGD. I tried to use different learning rates. Firstly, I used 0.1, then 0.01 but both of them weren't good enough to bring test set accuracy to 10% after 100 epochs. Then I tried to use 0.05, so in the middle of these two

values and it worked very nice. I got 16% of accuracy on test set, which was way higher than proposed 10%.
Next step was to implement dog breed classifier using transfer learning. First step was analogous to training CNN from scratch. I defined train, valid and test loaders with the same transformations as in the previous step. I used batch size of 64 due to the number of images in each bucket -- thanks to that the learning process was far shorter. Next I imported pytorchvision's already pretrained models.
As on pyTorch documentation website, there are many different pretrained models available to use. There were also best scores, achieved by these architectures on ImageNet dataset. I tried some of them and changed the data loaders accordingly to accepted input of given architecture. I tried to use Inception v3 model, which had one of the best scores as top 5 error and top 1 error. But it took a lot of time to train this one and I couldn't make it work. Training didn't make the loss on validation set lower, so I switched the model to a different one. Next I tried Resnet18, but it was also unable to learn dog breeds properly. Next step was to try VGG16 as this was the network that I used for dog detection problem in previous steps. After first promising epoch, it broke. The loss on both training set and validation set flew to nan, so the network couldn't converge at all. It seemed to me like a problem with regularization, so I changed it to VGG16_bn, so VGG16 with batch normalization applied. This one worked like a charm and I sticked to it and learnt it for 30 epochs. Thanks to the knowledge transferred from ImageNet learning process, it was able to distinguish between dog races easily and nicely. As for the optimization algorithm and loss function, the loss function was the same as in the previous step, but I changed the optimization aklgorithm. I used CrossEntropyLoss as the loss function and SGD with Nesterov momentum as the optimization algorithm. I picked the momentum term equal to 0.95 and it worked pretty well.
The last steps after training and testing this transfer learning model was to build an app, which tries to detect human face in the image. If it fails to do so, it tries to detect a dog. If there is no human face and no dog in the image, it should print an error. First thing to do was to create a dictionary mapping indexes of classes returned from the network into names of dog breeds, which were provided in the directory names. After that, I built a function which used dog detector from second part of the notebook, then face detector from the beginning of this project and then for both of these options, this image is fed into the transfer learning model. For humans, it prints the information about which dog breed does the human face resemble mostly and for dogs it predicts its breed.

4. Results

For the human face detector, the results were pretty nice. 99 human faces were detected correctly and 14 dogs were classified as human faces. It had very good accuracy for human faces, but it can make errors on pictures, which contain random, different image. Even though, error rate of 14% is acceptable and is good enough to stop searching for a better solution.
VGG16 dog detector (learnt on ImageNet) This approach classified 94% of dogs as dogs (so only 6 errors) and 0% of humans as dogs, so it was working pretty well.
My created from scratch CNN had accuracy of 16% on test set. It was better than the threshold for passing this exercise, which was set to 10%.
The pretrained VGG16_bn, fine tuned to my dog dataset got 83% on test set. The threshold for passing this exercise was set to 60%, so VGG16 with batch normalization was a lot better than I thought. It generalised the information about so many different dog breeds very well. Examples of good cases and mistakes are shown below:


This man resembles Alaskan Malamute the most.


American water spaniel was classified as curly-coated retriever. These dog breeds are really similar to each other, they are both brown and curly haired so to distinguish between these two cases we would need a lot more training

examples. But I think that in this specific case even human would have problems whether the dog in the left photo is an American water spaniel or pup of curly-coated retriever.

5. Conclusion

Thanks to this project I learnt a lot how to use PyTorch, how to use transfer learning, how to define my own neural network. I was able to choose proper optimization algorithm, loss function and tune hyperparameters of the network. I think that even though the network doesn't work for every case, it works pretty well. I think that both dog breed detector, as well as predicting the dog breed, which is the closest to a human face features are working pretty well. They are enjoyable and accuracy of these systems is high enough to be used by any people. The next step to make this application work better would be to deploy the model to Sagemaker and then use API Gateway to build a web app, using these models. It would be then very nice app, which could be a nice addition to my portfolio.