

Machine Learning Engineer Nanodegree

Capstone Project "Dog breed classification"

Franziska Zimmermann

December 28th, 2021

I. Definition

1. Project Overview

Identification and classification of objects in images is one of the most classical application areas of machine learning. Over the last decades neural architectures used to solve this task have taken an impressive evolution. Over time new publications brought in more and more smart ideas to process image information yielding better performances on benchmarks datasets like ImageNet. Some important milestones in this process of evolving cnn network architectures are LeNet (LeCun et al., 1998), AlexNet (Krizhevsky et al., 2009), VGGNet (Simonyan et al., 2014), GoogLeNet (Szegedy et al., 2014), ResNet (He et al., 2015), DenseNet (Huang et al., 2016) and ResNeXt (Saining et al., 2017).

Since I'm passionate about computer vision applications of machine learning algorithms I chose the Dog Breed classification project to explore how humans and dogs can be detected in images and how transfer learning can be used to predict the breed of a dog in an image reliably although this is a task that most humans (that are not dog experts) would fail in.

At first glance the problem of classifying dog breeds might seem not too relevant to solve in reality, however the classification of objects that vary only slightly in appearance (for example MRI images of different types of tumors) is in general a very important problem. Due to the availability of sufficient training data in this project the classification of dog breeds was chosen however knowledge gained during project development will surely be useful for more relevant problems in the future. Moreover this project allows us to understand the challenges involved in piecing together a series of models designed to perform various tasks in a data processing pipeline and using its individual strengths and weaknesses in a smart combination.

2. Problem Statement

The goal of this project is to build a pipeline 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. So as a first step, the algorithm has to decide whether a human or dog is present in a given image. In a second step it then decides about the breed of the apparent dog or the most resembling dog breed for a human.

For solving the task of detecting human faces in images OpenCV's implementation of Haar feature-based cascade classifiers will be used. For identifying whether a dog is present in an image a Pre-trained VGG 16 net (trained on ImageNet) will be used. The performance of both models will be evaluated on the given datasets.

For classifying dog breeds, we will first try to train a small CNN from scratch. For that we will investigate necessary/useful pre-processing steps for training and test images and selection of appropriate optimizer, loss function and training hyperparameters. The trained model will be evaluated on the given test set. Of course with this approach (training from scratch) we can't expect a very good accuracy of the trained model since the chosen network architecture will be probably too simple to represent this complex task. Of course we could invest in more time and resources for training a larger model from scratch but this is not necessary: Luckily there are several pre-trained bigger models available that contain lots of useful information in earlier layers that can be reused for the given problem. Therefore as a next step, transfer learning will be applied to get a better performing model for classifying dog breeds. Several pre-trained architectures will be finetuned on the given dataset of 133 dog breeds and evaluated on the test set. For performing transfer learning in a lightweight manner I will use Pytorch Lightning and will finetune a pre-trained Mobilenet, Resnet50 and ResNext101. Of course this again includes appropriate preprocessing of the images, selection of appropriate loss functions and optimizers and training hyperparameters.

Finally we'll put together the algorithms for identifying human faces, dogs and classifying dog breeds to build the final application and test it on some images the algorithm has never seen during training.

3. Metrics

The problem whether a human or dog is visible in a given image is a binary classification problem which can be evaluated by looking at the accuracy of the binary classifier. The problem which dog breed is shown in a picture of a dog is a multi-class (and single-label) classification problem which can be evaluated by measuring the accuracy of the classifier as well. Since distribution of samples among the 133 classes is approximately equal (see II.2) and classification results on all classes are equally important plain accuracy does the job and we don't have to think too much about macro f1 or precision/recall on special classes.

II. Analysis

1. Data Exploration

The datasets of humans and dogs were provided by Udacity and unfortunately I couldn't find any details about their origin. It contains 13233 human and 8351 dog images. The human images folder contains portrait photos of several prominent figures (sometimes only one picture per person, sometimes several). The dog images folder is split into a train, dev and test subdirectory. Each of these contains 133 subfolders for the various dog breeds. Each dog breed subfolder itself contains many different photos showing dogs from the respective breed. The photos have varying backgrounds and show the dogs sometimes in full-body mode, sometimes just in portrait mode. There are even some pictures that also contain a human or multiple dogs.



Image 1: Examples from training set

For an analysis about the distribution of train/test data among the 133 dog breeds please refer to section II.2 . As you can see from the example images shown in Image 1, the images have varying shapes. For feeding the images in a CNN (that is not fully convolutional) we need to resize all images to a common shape. Therefore I did some analysis on the heights/widths of the training images to come up with a good common image shape. Image 2 shows the width/height distribution of images in the train set.

For the training images the median of width/height distribution is approx. 1.13. (Later all images will be resized to this width/height ratio). Since it is a lot better to use downscaled images for a CNN classifier than upscaled ones we calculate the 5% quantile of heights of the training images (e.g. 95% of the training images have at least this height). This is 269. So we will later scale all images to this height.

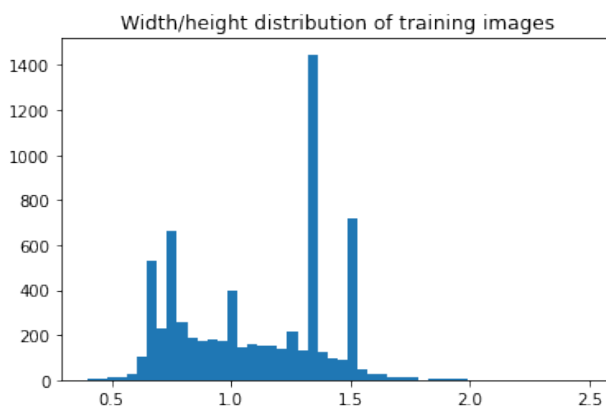


Image 2: Width/height distribution of images in train set

By looking at various images from the train dataset we can conclude that the dataset is sufficiently large and contains enough variation for training a model that can generalize well to real-world images not seen during training.

2. Exploratory Visualization

Here we want to take a closer look at the distribution of the different dog breeds in the train and test set. Image 3 and Image 4 show histograms of the class distribution in the respective sets. In the train set, the number of samples per dog breed category varies between approximately 30 and 80 but is in general quite balanced. So there are no classes with really little examples which makes the classification problem a bit easier (compared to a scenario with some really small classes in the train set). The distribution of classes in the test set seems to match the class distribution in the train set (probably a stratified sampling technique was applied during the train-test-split). For each class, we have between 3 and 10 examples. Therefore the test set seems to be well suited for estimating the quality of the classifier.

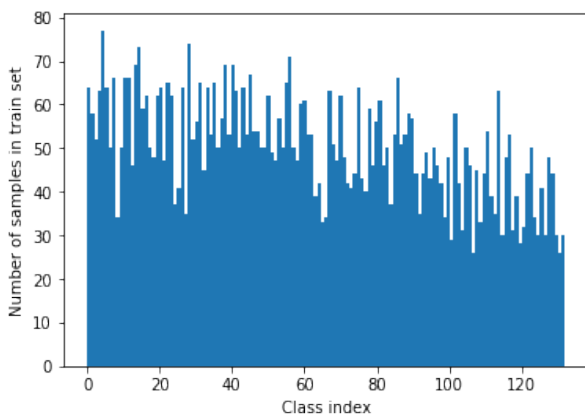


Image 3: class distribution train set

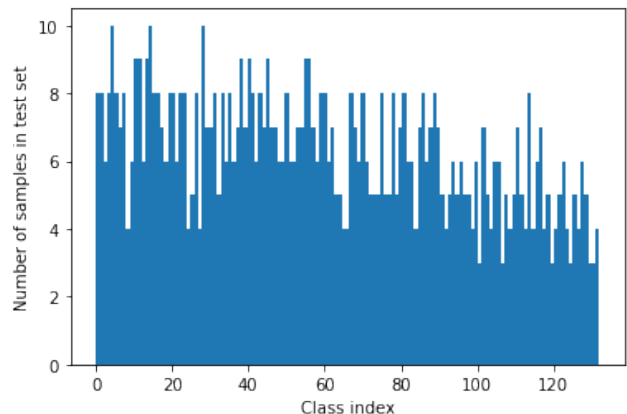


Image 4: Class distribution test set

3. Algorithms and Techniques

For solving the task of detecting human faces in images OpenCV's implementation of Haar feature-based cascade classifiers is used. Haar cascades are a very popular choice for face detection since the calculation is computationally cheap (first real-time face detection system!) and the results are usually quite good.

For identifying whether a dog is present in an image a Pre-trained VGG 16 net (trained on ImageNet) will be used. An ImageNet-pre-trained net can be used without finetuning/exchanging the last layer because among the classes the net was trained on, there are some dog classes (corresponding to the class indices 151-268 of ImageNet) and not-dog-classes (all other class indices). Therefore the prediction of the pre-trained net can be directly used to distinguish dog images from not-dog images.

For the task of classifying dog breeds we set up a benchmark by training a small CNN from scratch. In order to do that we did some statistical analysis on the image heights and widths appearing in the training set and made an informed choice on a common image height/width (which is necessary since the CNN is not fully convolutional). We also apply augmentation techniques to enlarge the training set a bit and make the model more robust. For details about the preprocessing and augmentation strategies used please refer to section III.1.

As a baseline model, I trained a small CNN from scratch. When designing the model architecture for that I had in mind that the net mustn't be too deep/wide (have too many parameters), otherwise training will take a lot of time but it must be complex enough s.th. it can capture enough image details to be able to perform the difficult classification task of classifying dog breeds. I decided to use 3 convolutional blocks, each consisting of 2 relu-activated 3x3 convolutions followed by a 2x2 maxpooling and dropout layer. During these 3 convolution-maxpooling blocks the height and width of intermediate output tensors decreases while its number of channels increases. This follows the well known idea in literature that deeper layers contain "more aggregated" information while early layers capture the basis features in finer grained details. I decided to use relu activation functions since this has a reduced likelihood of the gradient to vanish. Using convolutional layers for feature extraction (instead of fully connected layers) is very popular in image processing since they

extract location invariant information and have far less parameters than fully-connected layers since weights are shared. The max-pooling layers are needed to reduce the height/width of the feature map. I also added dropout layers to introduce a bit of regularization and make the net more robust against overfitting. After the 3 conv-pooling blocks there are 2 linear layers and one dropout layer as "classification head" followed by a log-softmax function that returns log-probabilities as output of the net.

I used the NLLoss function during training because the output of the net contains log-probabilities of the respective classes (we applied the log-softmax activation in the last layer!). (If the output of the net would be unnormalized scores then the CrossEntropy would be the appropriate loss to use.) I decided to go for an Adam optimizer since it's a very popular optimization algorithm with many advantages compared to SGD: According to (Kingma et al., 2014) "the magnitudes of parameter updates are invariant to rescaling of the gradient, its stepsizes are approximately bounded by the stepsize hyperparameter, it does not require a stationary objective, it works with sparse gradients, and it naturally performs a form of step size annealing." Since the results from training a small model from scratch were not too good (which was to be expected since the net is not big enough to do well on this complex task), I used transfer learning as a next step to improve.

When deciding for a pre-trained architecture to use you have to take two criteria into account: The top1/top5 accuracies obtained on the pre-training dataset (ImageNet), and the training/inference speed (that heavily depends on the number of weights the model contains). When you look at Image 5 you want to choose a model in the upper left corner with a top1-accuracy as good as possible but small number of operations. In this regard ResNeXt-101 yields a pretty good trade-off.

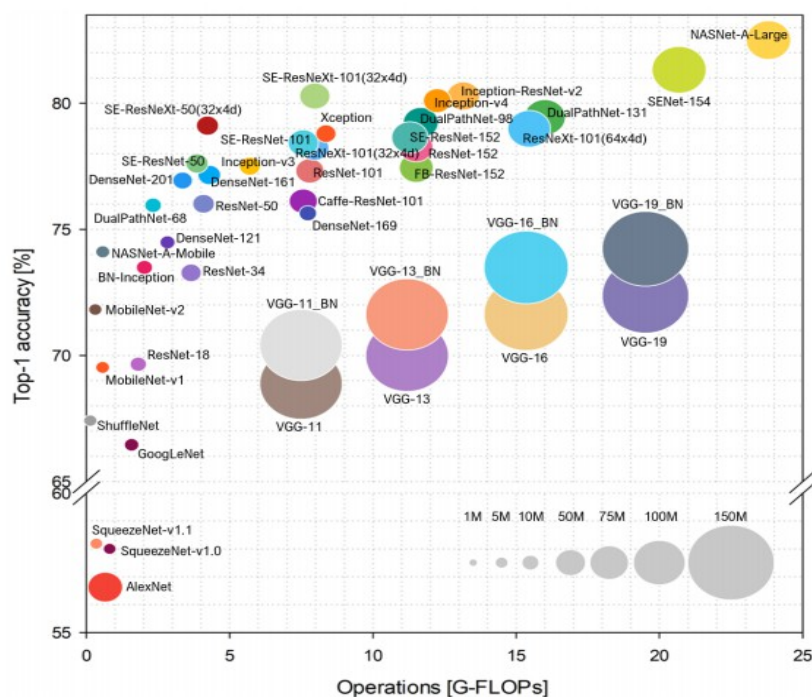


Image 5: Overview of operations, model size and top-1 accuracy of popular network architectures, source: <https://towardsdatascience.com/how-to-choose-the-best-keras-pre-trained-model-for-image-classification-b850ca4428d4>

ResNeXt has a smart architecture that combines good ideas from several architectures used before: having a shortcut from the previous layer to the next to enable better gradient flow (from ResNet), repeating layers to build a deep model with a lot of learning capacity (from VGG) and usage of a split-transform-merge practice (split the input to multiple blocks and merging them later, from InceptionNet). Compared to InceptionNet, ResNeXt uses the same topology for parallel blocks that's why fewer parameters for defining the architecture are required while more layers can be added.

Fortunately, ResNeXt is available as ImageNet-pre-trained model in pytorch. The images we want to classify into dog breeds look quite similar to the ImageNet images, meaning both sets contain photos of real objects.

Moreover in the ImageNet pre-training the model learned already to distinguish 118 different dog types. Of course, the model was focusing not only on dogs back then and we also have different dog breed categories now, but the pre-training task still contained a bit of the task we have to solve now. Therefore we can expect that the features contained in the pre-trained model can be re-used very well even up to deeper layers. Therefore I decided to re-use the full pre-trained net and only replaced the very last linear layer with a few new linear layers where the final one matches the number of dog categories. I finetuned first only these new layers (keeping the well initialized pre-trained features in earlier layers fixed), later I allowed updating also the earlier layers a bit with a lower learning rate. For comparison of several model architectures (and out of curiosity how well a smaller model would perform) I also finetuned a ResNet50 and a Mobilenet. I used again the Adam optimizer but this time with a CrossEntropy loss since the last layer of the net outputs raw unnormalized scores.

After having trained a net that does a really good job of classifying dog breeds the final step was only to put everything together that we have until now: first the Haar feature-based cascade classifier and the pre-trained VGG16 net are used to detect whether a dog or a human face is present in the image. As a second step, the image is fed into the trained dog breed classifier to predict the dog breed. The user is shown the image together with a print statement saying “Hello dog” or “Hello human” (depending on whether a dog or human face was detected in step 1) and “You look like a ...” giving the predicted (most resembling) dog breed.

4. Benchmark:

As a benchmark model, I trained a really small and simple CNN architecture (consisting of a few conv and max pooling layers, for details about the architecture see section III.2) from scratch. I decided for this as a benchmark model since there was no pre-trained net out there that would do prediction using exactly those target classes provided in the dataset and training/finetuning a bigger model would have been more time consuming, complicated and resource intense. I trained my baseline model for 100 epochs with NLLoss using a learning rate of 0.0001 and Adam optimizer. When evaluating the trained model on the test set I obtained an accuracy of 22% which is not bad compared to a random guess on the 133 classes (and taking into account that the model architecture was really small). However this accuracy is very too small to be an acceptable solution for the given problem of classifying dog breeds reliably.

III. Methodology

1. Data Preprocessing

Before feeding the image to the Haar feature-based cascade classifier we will convert it to gray scale (since color is not important for finding faces).

In order to be able to apply the Pre-trained VGG 16 net (trained on ImageNet) to our images, we have to do the same pre-processing to our images as it was done to the ImageNet images during initial training.

Therefore we resize it (to height/width 256), center crop (to height/width 224) and normalize (with $\text{mean}=[0.485, 0.456, 0.406]$, $\text{std}=[0.229, 0.224, 0.225]$) the images appropriately.

When training the small CNN from scratch for the baseline model and doing transfer learning for the final model I was free to choose my own pre-processing and augmentation techniques and used the following pre-processing and augmentation for both:

For the training images the median of width/height distribution is approx. 1.13. Therefore I chose to resize all images in a way that their width/height ratio matches this. Since it is a lot better to use downscaled images for a CNN classifier than upscaled ones I decided to take as a common height the 5% quantile of heights of the training images (e.g. 95% of the training images have at least this height). This leaves us with a final image size of $\text{height}=269$ and $\text{width}=1.13 \cdot 269=303.97 \sim 304$. To get all images to the approximately desired size I resized them in a way that their smaller edge is 304 (while maintaining their aspect ratios). In order to get all images to the desired aspect ratio: Since stretching the images to the desired aspect ratio could destroy the visual appearance of the objects I used cropping ($\text{height}=269$ and $\text{width}=304$). Since during training some augmentation is a good idea anyway I used RandomCrop for the train set. For the test set I used the normal CenterCrop. For the train set I additionally applied augmentation via random horizontal flips and rotation up to 20 degrees. This helps improving generalization abilities of the trained model and avoids overfitting (since the training dataset is with 6680 samples not very large for this difficult task of distinguishing 133 classes).

For choosing the batch size, I took into account that when you choose it too small you won't fully use the computational speedups from the parallelism of GPUs and moreover when choosing a bigger batch size you will more likely end up in a globally optimal solution. However, a really large batch_size can slow down the training process empirically, meaning that with a small batch size you will faster end up in a quite "good" solution than when searching with a big batch size for an more optimal solution. Smaller batch sizes also have the advantage of better generalization of the model. Therefore I decided to use a "middle large" batch size of 16.

2. Implementation:

For the Haar feature-based cascade classifier I used OpenCV's implementation (detectMultiScale method of CascadeClassifier, see https://docs.opencv.org/3.4/d1/de5/classcv_1_1CascadeClassifier.html).

For applying the pre-trained VGG16 net I used the pre-trained pytorch model from torchvision.models (see <https://pytorch.org/vision/stable/models.html>). If the prediction of the net was between 151 and 268 (inclusive), then we interpret this as "dog present" otherwise as "no dog present". The reason for that is that the keys 151-268 correspond to dog breeds in the ImageNet classes.

For training a small CNN from scratch I implemented a DogDataset class that inherits from the Dataset class of pytorch and used the Dataloader functionality from pytorch. For the preprocessing of the training and test images I used two separate composed transformations (see class torchvision.transforms.Compose). The architecture of the small CNN that I trained from scratch was:

- nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, stride=1, padding=0)
- nn.ReLU()
- nn.Conv2d(in_channels=32, out_channels=32, kernel_size=3, stride=1, padding=0)
- nn.ReLU()
- nn.MaxPool2d(kernel_size=2)
- nn.Dropout(0.1)
- nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=1, padding=0)
- nn.ReLU()
- nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, stride=1, padding=0)
- nn.ReLU()
- nn.MaxPool2d(kernel_size=2)
- nn.Dropout(0.1)
- nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, stride=1, padding=0)
- nn.ReLU()
- nn.Conv2d(in_channels=128, out_channels=128, kernel_size=3, stride=1, padding=0)
- nn.ReLU()
- nn.MaxPool2d(kernel_size=2)
- nn.Dropout(0.1)
- flatten tensor
- nn.Linear(128* 30* 34, 128)
- nn.ReLU()
- nn.Dropout(0.3)
- nn.Linear(128, number_of_classes_in_train_set)
- nn.LogSoftmax(dim= 1)

I trained my baseline model for 100 epochs with NLLLoss using a learning rate of 0.0001 and Adam optimizer.

For developing the final model using transfer learning I decided to use pytorch lightning (and its CLI).

PyTorch Lightning is a high-level programming layer built on top of PyTorch. It makes building and training models faster, easier, and more reliable. Inspired by an example in the pytorch lightning examples folder (https://github.com/PyTorchLightning/pytorch-lightning/blob/master/pl_examples/domain_templates/computer_vision_fine_tuning.py) I made the files

"lightning_module_and_datasets.py" that contains definition of data loaders, model architecture, optimizers and training functionality. For running the training I made the script "lightning_cli_script.py".

I implemented a DogBreedDataModule inheriting from pytorch_lightning.LightningDataModule that

contains the same pre-processing as described earlier and offers a `train_dataloader`, `val_dataloader` and `test_dataloader` method. I implemented a `MilestonesFinetuning` class (that inherits from `pytorch_lightning.callbacks.finetuning.BaseFinetuning`) to control the freezing/unfreezing of certain model parts during training. Finally I implemented a `TransferLearningModel` (inheriting from `pytorch_lightning.LightningModule`) that contains the model itself, initialization of optimizer, learning rate schedule and loss function and training, validation and test functionality. When initializing the `TransferLearningModel` you can pass it a “backbone” parameter and depending on that you can perform finetuning on different backbone models. When building up the internal model in the `__build_model` method of `TransferLearningModel` the last layer of the backbone model will be discarded and instead a Linear layer, a ReLU and two more linear layers will be added as new classification head.

Since the output scores are unnormalized the `CrossEntropyLoss` function will be used. The milestones parameter of `TransferLearningModel` controls after how many epochs the 5 last layers of the backbone model and the remaining layers of the backbone model will be unfrozen.

I decided to use the lightning cli since it “provides a standardized way to configure experiments using a single file that includes settings for Trainer as well as the user extended `LightningModule` and `LightningDataModule` classes. The full configuration is automatically saved in the log directory. This has the benefit of greatly simplifying the reproducibility of experiments.” (see https://pytorch-lightning.readthedocs.io/en/stable/common/lightning_cli.html).

I trained all models using the Adam optimizer, a multi-step learning rate scheduler, a start learning rate of $1e-3$, `lr_scheduler_gamma=1e-1`, `milestones=(2, 4)` and `batch_size 32`.

I finetuned a `mobilenet_v2` for 180 epochs, a `resnet50` for 40 epochs and a `resnext101_32x8d` backbone for 40 epochs. The number of epochs needed to get convergence in the train loss and validation accuracy were found by experiments and looking at the respective plots in tensorboard (see Image 6 and Image 7).

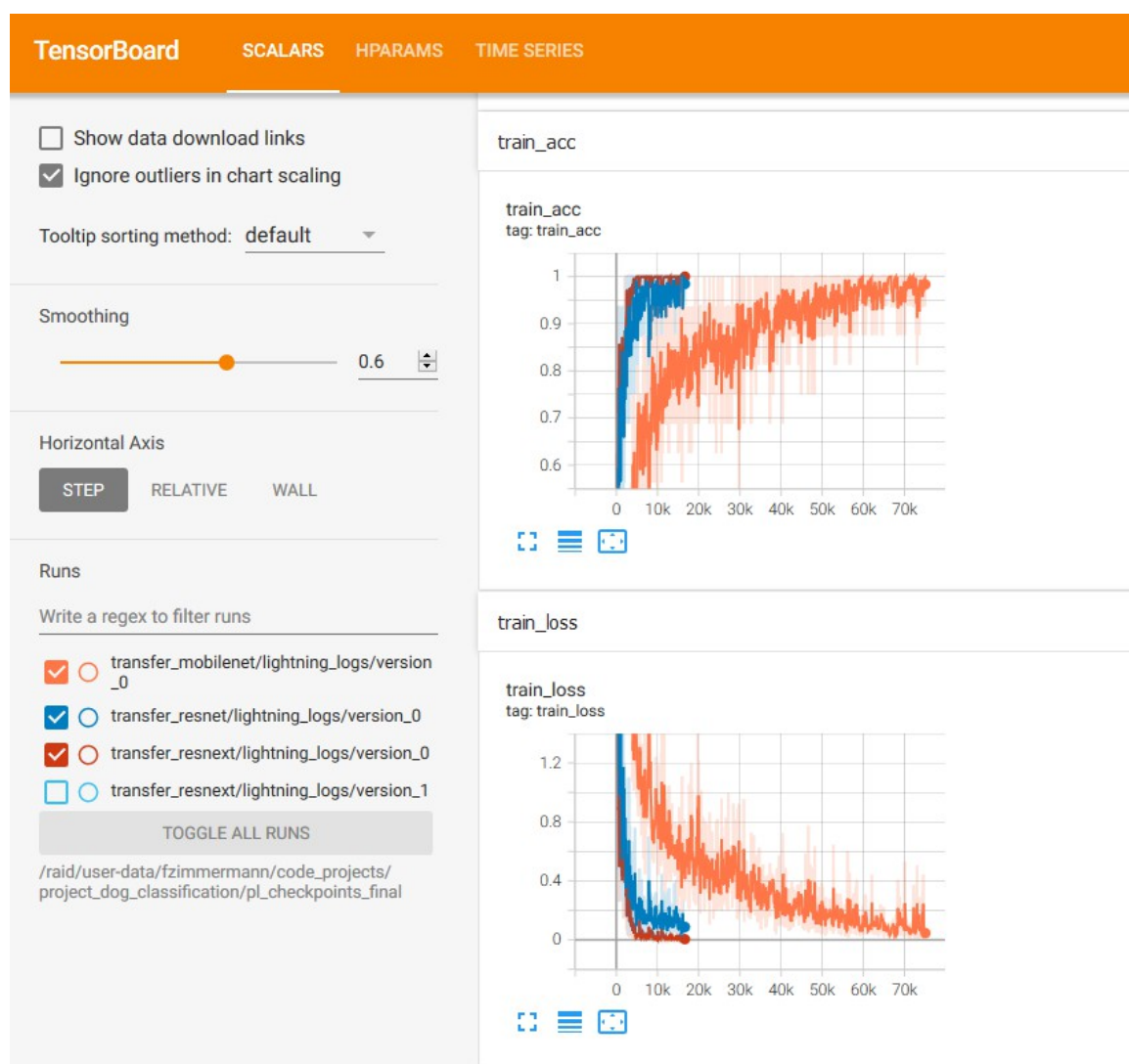


Image 6: Tensorboard plot of accuracy and loss on train set

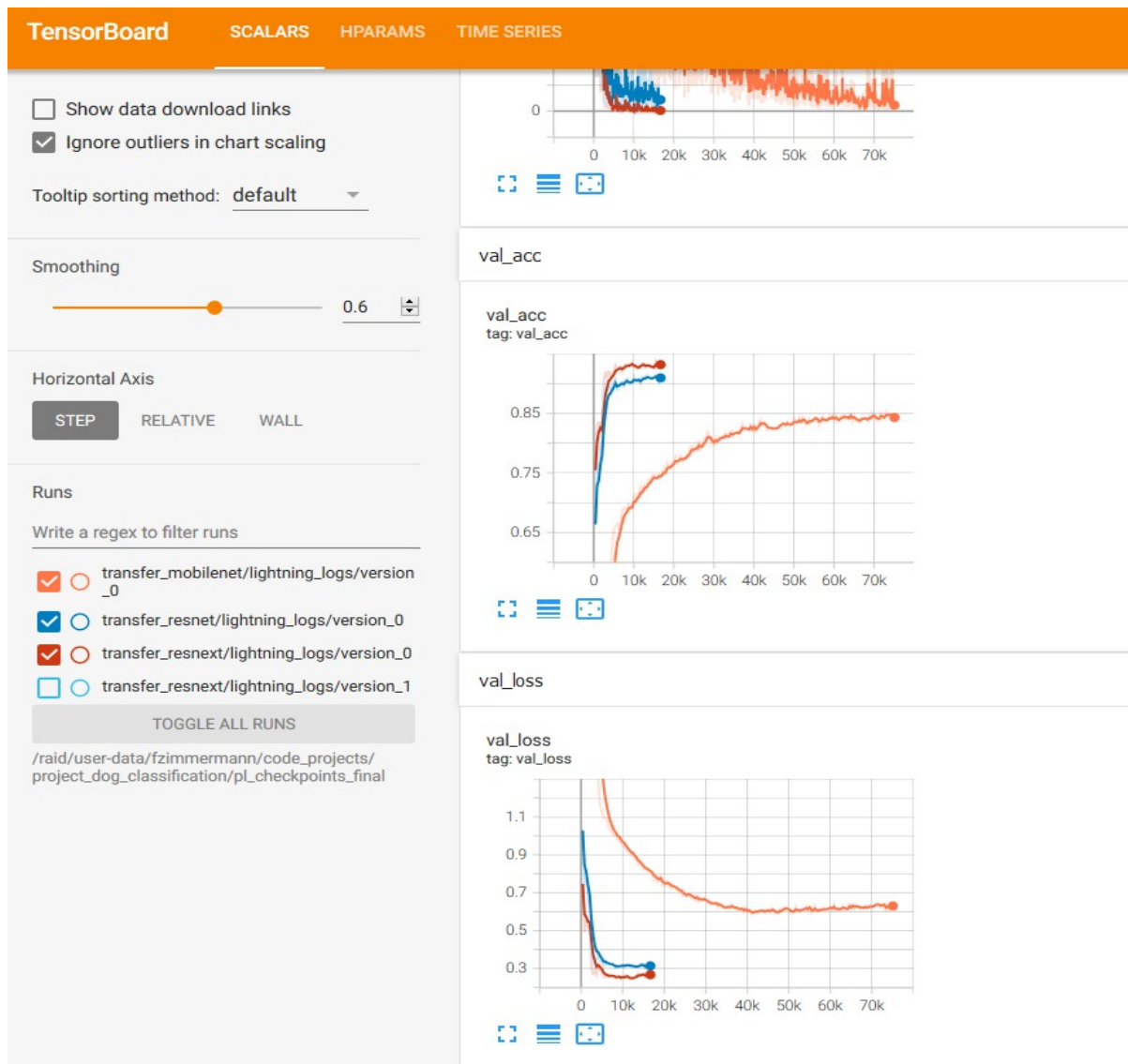


Image 7: Tensorboard plot of accuracy and loss on validation set

3. Refinement

I started off with training a small CNN from scratch. The performance was very good compared to a random guess but far not good enough to yield a reliable prediction of the dog breed from an image. I then finetuned 3 different backbone models pre-trained on ImageNet on our dog dataset. First, I accidentally didn't freeze the backbone net during the first epochs of finetuning which had catastrophic effects: the pre-trained weights were messed up in the beginning and there was basically no use of the pre-trained weights at all which gave the same results as training a big net from scratch in the beginning (so not very good!). But after finding this bug and introducing a proper freezing of the backbone net during the first finetuning epochs results improved significantly from the first epochs on. Regarding the hyperparameters my first guesses were quite good already that's why I didn't have to modify a lot there. Using the plots in tensorboard I found out the number of epochs needed to archive convergence on the training loss and validation accuracy. Pytorch lightning includes a functionality that always saves the (in terms of validation accuracy) best model so far which is very handy.

IV. Results

1. Model Evaluation and Validation

When inspecting the validation accuracy plots in tensorboard (Image 7) you can see that finetuning resnext101_32x8d gives the best performance on the validation set (~0.93), followed by resnet50 (~0.91). mobilenet_v2 gives only about 0.85 for the validation accuracy. This order was kind of to be expected when you look at the number of operations/parameter size and Image-Net-top-1 accuracies of the pre-trained backbone models (Image 5). Therefore I decided to use the model with the resnext101_32x8d backbone as final model. As stated in section III.3 Pytorch Lightning automatically saves the (in terms of validation accuracy) best model that is seen over the various epochs in the given training folder. Later you can evaluate this checkpoint on the given test set. When evaluating the resnext101_32x8d backbone model on the test set I got an accuracy of 0.92 which is a very good result given the difficult task of distinguishing dog breeds (that look really similar in many cases). It's also amazingly good compared to the poor baseline result of 0.22 accuracy on the test set.

I used this finetuned resnext101_32x8d then in my final overall model that contains beside the dog breed classifier the face detection and dog detection part. Since the face detection and dog detection algorithm were fully pre-trained (and not finetuned on the given training data) for evaluating these algorithms it is not important whether images belong to the given train or test set. I evaluated both using the first 100 images found in the folder. The face detection algorithm detected a human face in 97.0% of human images and in 26.0% of the dogs images, so it has a very high recall and also a reasonably good precision. The dog detection algorithm detected a dog in 87.0% of dog images and in 0.0% of the human images, so it has a very good precision and a reasonably good recall. When looking at the evaluation of the 3 algorithmic parts individually we can conclude that each part does its job quite well to very good which gives us confidence that also the overall system will work well.

However, to verify that we will test it using a variety of example images that were neither in the train nor test set used so far. Table 1 shows an overview of the images tried, the true breed, the human versus dog prediction as well as the predicted breed.

As a first step, I tried the algorithm on 3 dog and 3 human images. The algorithm was able to classify all human images as humans (although I chose on purpose one image of a lady that looks a bit "dog like") and all dog images as dogs. Moreover all dog breeds of real dog images were classified correctly. For the human images the humans really resemble the predicted dog breeds.

As a second step, I tried the algorithm on 8 images of dogs that look really similar to a different breed than they actually belong to. Even in those very difficult cases (where a human would likely fail) the algorithm finds the correct breed in 50% of all cases which shows that it has better performance than an experienced human classifier.

As a third step, I tested the robustness of the model showing it a poodle shown in side view and a poodle with clothes. The poodle from the side and the poodle with clothes are both classified as dogs with breed Bichon Frise which is not too bad since Bichon Frises look really similar to poodles.

Input image	True breed	True species	Predicted breed
	Affenpinscher	Dog	Affenpinscher
	Bull_terrier	Dog	Bull_terrier
	Collie	Dog	Collie
	Human	Human	
	Human	Human	
	Human	Human	

	english_springer_daniel	Dog	English_springer_spaniel
	welsh_springer_daniel	Dog	Ibizan_hound
	italian_greyhound	Dog	Italian_greyhound
	whippet	Dog	Italian_greyhound
	boston_terrier	Dog	Boston_terrier

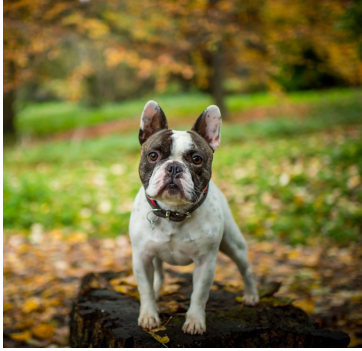
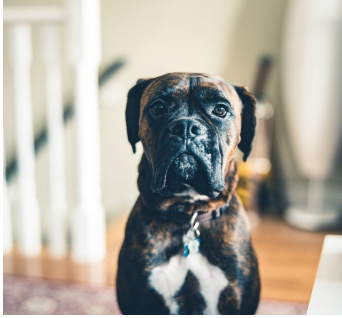



	french_bulldog	Dog	Boston_terrier
	boxer	Dog	Cane_corso
	cane_corso	Dog	Cane_corso
 <small>shutterstock.com · 2083976233</small>	poodle	Dog	Bichon_frise
	poodle	Dog	Bichon_frise

Table 1: Results of the final model on several example images

We've seen that the model works well on a variety of new, sometimes even tricky images which shows us that the model generalizes well to unseen data and performs remarkably well even on in terms of dog breed “tricky” images that would probably fool a human. So results from the model can be trusted most of the times.

2. Justification

The final dog breed classifier has an accuracy of 0.92 which is a very good result compared to the poor baseline result of 0.22 accuracy on the test set. As discussed in the section IV.1 also the other algorithmic parts (human face detector and pre-trained dog-detector) have high precision and recall which makes the overall model significant enough to solve the given problem.

V. Conclusion

1. Free-Form Visualization

Here we try what happens when we feed the algorithm images with humans and dogs or images with multiple different dog breeds. We try 2 images with a human and a dog present and one image showing 4 different dogs belonging to 3 breeds. Table 2 shows the results. We can see that the algorithm classifies all images as “dog” which is understandable even in those cases where a human is visible as well because the check for “dog” is carried out first in the implementation and if a dog is found, the image is classified as “dog”. The predicted breeds of the dogs present in the 2 images that also contain a human, however, are not understandable and far away from being correct. In the case of the image with 4 dogs present a breed that is not present at all was predicted which is also not correct. So we can conclude that our current model is not able to handle pictures with humans and dogs or multiple dog breeds reliably. If we'd like to have that we should convert the current image classification algorithm to an object detection algorithm that is able to locate different object types in different parts of the image (see section IV.3).

2. Reflection

In this project we've build a system that can tell whether a human or dog is visible in an image and determine the breed of the dog or the most resembling dog breed for a human. It was interesting to see that a simple VGG pre-trained on Image-Net does a very good job in determining whether a dog is present in an image. I was also surprised to see how well the good old Haar feature-based cascade classifier works. It was implementation-wise nice to design and train an own model from scratch however it felt a bit useless when you know the power of transfer learning already. But this project showed us once again that by using features pre-trained on a really large dataset you can gain lots of performance especially under the constraints of little training data for the actual problem and little computational resources (that don't allow training a big model from scratch).

3. Improvement

An accuracy of over 90% on the test set shows that the trained classifier for predicting dog breeds is quite good already. However, from the tensorboard logs (Image 6 and 7) we can see that we're facing some overfitting during training. As a remedy you could increase the amount of training data (or do even more augmentation) and add more regularization during training (like more dropout or weight_decay for example). With that we could maybe still increase the performance on the test set a bit.

A bigger problem however is the performance of the current algorithm on images that show humans and dogs or multiple different dog breeds (as shown in section V.1). In those cases there's plenty of room for improvement. Currently, the algorithm is built under the assumption that an image is either “human” or “dog” and shows exactly one dog breed. To resolve this we could either convert the problem into a multi-label classification problem or use an object detection algorithm that can locate objects with various types. The second idea would be more appealing since the user then could also see where the algorithm has found a human or dog in the image or which dog belongs to which breed in an image with multiple dogs. We could for example use a one-shot object detector like YOLOv3 ([arXiv:1804.02767](https://arxiv.org/abs/1804.02767)) which has the advantage of being very fast (compared to a more traditional two-stage object detector).




Input image	True breed	True species	Predicted breed
	poodle	Dog	Welsh_springer_spaniel
	Poodle	Dog	Bull_terrier
	Rottweiler_chihuahua_poodle	Dog	Chow_chow

Table 2: Results of the final model on example images with humans and dogs or multiple dogs