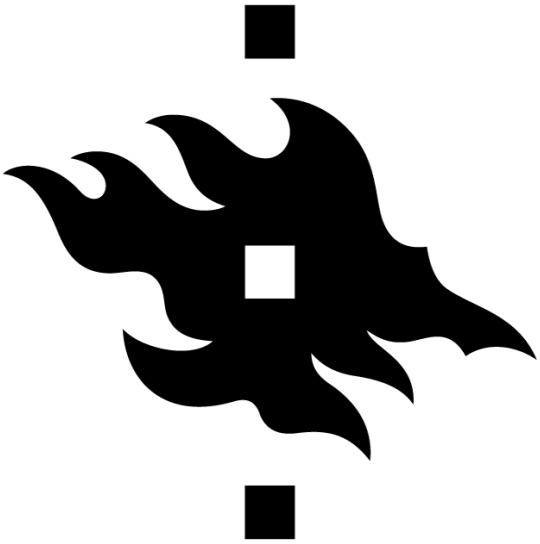
University of Helsinki



Report of the project of Deep Learning

Image project

Students:
Riccardo Menoli (Student-id: 014963230)
Stefano Lia (Student-id: 014962723)
Tafseer Ahmed (Student-id: 014973194)

2018/2019

Contents

1 About the project

To do the project we used **Google colab**, the Google's free cloud service for AI developers. The reasons that convince us to use this tool are the possibility to deploy both python and jupyter notebook file and the availability of the GPU. In the zip file you can find three different directories:

1. **fast-ai**: which contains the results obtained with fast-ai tools;
2. **pyorch**: which contains all the other trials we performed and the final chosen algorithm.

Notice that two algorithms are provided for the final solution. Further explanation about that can be found in the last section (section 4).

You can find the pth and ptk files in the GitHub repository in the *models* directory.

2 Data Management

To start the project the first thing to do was finding the proper way to handle the data. In this section we explain all the problems we faced and the corresponding found solution during this process.

2.1 Creation of the Dataset

The first problem was the **creation of the Dataset**. As the guide of PyTorch and the exercise suggested we created a custom Dataset following this guide. Based on this we organized the pandas Dataframe in the following way:

- one column containing the path of the image;
- one column for each class. If the sample (the row) belongs to one on more class, it has 1 in the corresponding column which represents the belonging class;
- one column containing the representing image in the PIL format;
- one column containg one list whose values are the classes which the example belongs to.

2.2 Grayscale problem

Some images are in grayscale. This is a problem because the `in_channels` parameter of the convolutional layer requires a fixed number of channels to work properly. Due to the fact that the majority of the images are colored, we decided to **convert** the greyscale images from one channel to three channels. This can be easily done using the pytorch function `convert("RGB")`, after opening the image. This operation is done when an image is taken from the dataset (in `__getitem__` function).

2.3 Missing values problem

Some images in the dataset miss at least one label or are complete without them. If we plot these images we can easily notice that some of them can be considered incorrectly unlabeled. In fact, there are some images, in which can be reasonable to have one (or more) of the considered classes, with some missing labels (i.e. im13 represents a river but it has no labels or im19 which represents clouds).
Our first idea was to remove these images in order to not add possible undesirable noise to the data. However, doing this we force the model to make **always** predictions. This is not always the best choice. In fact, some images does not contain any class and have an empty prediction is the maximum we can achieve. So, if we eliminate all the images without label the model will learn also to make always a prediction (which makes the model less precise). Since labeling all the images

which suppose to have a class was too expensive, we decided to train the models on the whole training set.
The division between training and test set is respectively 90% of the data for the training and 10% for the test. Finally, plotting the test set we notice that there are several images for which any of the class can be assigned. So, we assumed reasonable to mantain all the images without any label in the training set instead only a part of that. In other words, we assumed that the distribution of the images is similar between training and test set.

2.4 Imbalanced data

The dataset is strongly imbalanced. What we mean here is the fact that we have just a few examples for some classes. In fact, the distribution of the training set looks like this:

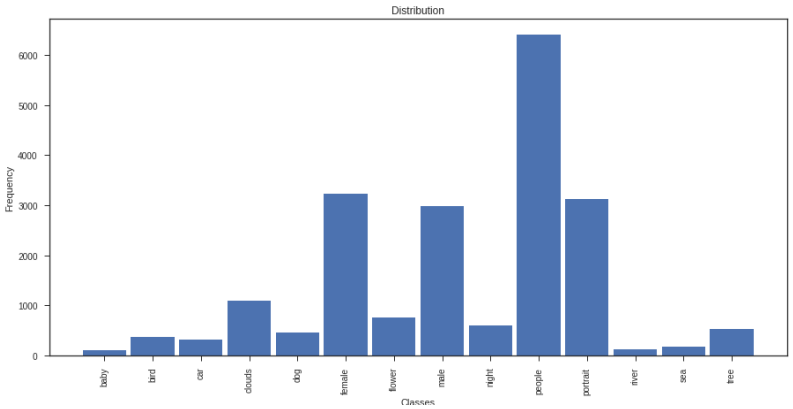


Figure 1: Distribution of the classes

The main consequence of this problem is that the network could have difficulties to predict correctly the classes with a few examples in the training set. First we tried to penalize the loss function accordingly to the number of samples for each class by using a *weights vector*. However, this didn't produce a good result. Secondly, we tried to find a good threshold for the predictions. Each model that we tried has as final output a fully connected layer with 14 neurons (because we have 14 different classes). The activation function used in this layer is the sigmoid. So, finding the right threshold helps the model out to find the correct labels for each sample. The idea was to request a minor activation for neurons that represent classes with few examples. Vice versa for the classes with many examples.
We started considering as a good threshold the proportion of the data in the training set. For example, if the class 'bird' is the label of the 30% of the dataset, the started threshold for that class is 0.3 (so to predict 1 for this class the output of the neuron should be higher respect to this value). After that we have adjusted a bit this threshold in order to have better precision than recall. Our idea was to make the predictions when there is a high activation of the corresponding neuron.

2.5 Batching, scaling and hyperparameters

We divided the data in this way: 90% of the data for the training and 10% for the validation set. We used a batch size of 64 images. We chose 64 for the following reasons. First, since we have 180000 images, having this batch size allows us to have enough updates for each epoch to get low the loss function. Second, a batch size too small would make the training really slow and variable. Finally, a bigger batch size lead us to CUDA memory problems. The **learning rate** is chosen automatically by the **Adam optimizer**. We didn't change it because the results were good. Finally, we added some transformations to the images - when they are taken only from the training set - to improve the results (e.g. RandomHorizontalFlip and RandomRotation).

2.6 Splitting strategy

To have trustable results we split the data in training and validation set making sure that each set contains the same proportion of samples for each class. Further information can be found in `image_project_pytorch.ipynb`.

3 Models

We tried the following different models:

1. Custom Inception Net: a custom version of the Inception_v1 network;
2. Custom Resnet: a custom version of the Resnet 101;
3. Resnet50 (pretrained);
4. Models with FAST.AI tool.

For each model we will show the comparison between validation and training loss by plotting them. The final models are chosen in the epoch when the validation error is lowest. This is done to avoid overfitting and get the most general solution (early stop).

3.1 Custom Inception Net

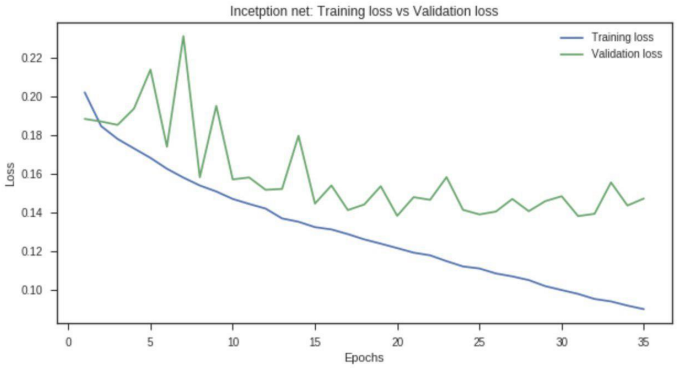


Figure 2: Custom Inception Net loss

The best model was saved at the 32th epoch. The model has been trained for 35 epochs.

3.2 Custom Resnet

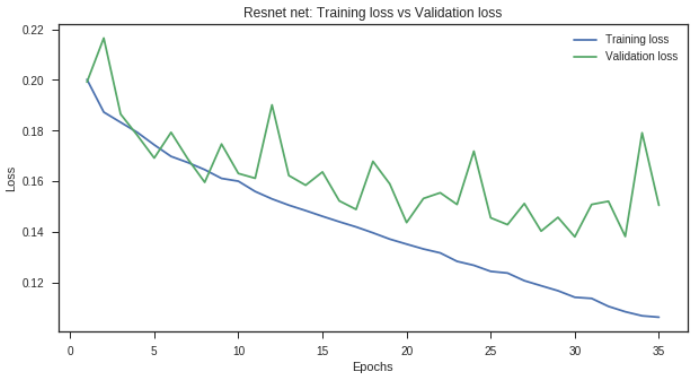


Figure 3: Custom Res Net loss

The best model was saved at the 31th epoch. The model has been trained for 35 epochs.

3.3 Pretrained Resnet50

Figure 4: Pretrained Resnet50

In the Pretrained Res Net we changed the last fully connected layer in order to have only 14 output neurons. Despite the network was already trained, we freezed the first six layers and we trained the last ones for 20 epoch. The best model was saved at the 19th epoch. So, it seems reasonable assuming that - with more trining epochs - the model can achieve more accurate results.

3.4 Comparison among the last three models

We compared the models using the micro-average F1-score on the Validation set:

1. F1-score Custom Inception Net: 0.4728875826598089;
2. F1-score Custom Res Net: 0.47159558702453486;
3. F1-score Pretrained Res Net: 0.665258711721225.

The best model, according to the micro average F1-score, is the Pretrained Res Net.

3.5 FAST.AI

Finally we tried to use the **fast.ai** deep learning library. This tool is built on top PyTorch and, by using this, we tried two other models: Resnet-18 and Resnet-152.
We also attach the code of these two models (as it is explained in section 1). The best one between them is Resnet-18, since it achieves a better micro average f1-score. The results are the following:

- F1-score of Resnet-18: 0.742385;
- F1-score of Resnet-152: 0.724525.

4 Conclusion and results

Since we used two different approaches (PyTorch and Fast.AI) we decided to provide two final algorithms: the pretrained resnet-50 (for the PyTorch approach) and resnet-18 (for the FAST.AI). These are the best models in terms of micro-average F1-score.

So, you can find two different models in the GitHub repository:

If we have to choose one solution, according to the results, we will choose the FAST.AI solution.