Training Traffic Sign Recognition Network

Albert Gregus

November 2021

1 Code structure

In order to be able to easily set the and find different parameters of the machine learning project I used config files (eg. *config/config_files/base.yaml*) (different experiments should have different config files). It contains the parameters in embedded directories in a well separated manner. The configs are read by the Config object and formed into an embedded Namespace.

All the elements of the ML project (model, optimizer, loss etc.) are inside the *ml* folder under the corresponding subdirectories. The subdirectories contain a switch inside their *__init__.py* file for getting the elements, and can also contain custom elements. The switch first tries to load builtin elements from the installed packages according to the name and parameters in the config (eg. 'resnet18' with pretrained=True from torchvision.models), then tries to load custom elements (if there are any).

The whole experiment is contained inside a Solver object, which initializes all the elements and controls the training/inference. This way the hyperparameter optimizer can instantiate whole experiments with different hyperparameters. The hyperparameters can be set in the config file (eg. learning rate with loguniform distribution between 0.001 and 0.01).

2 Dataset

The data has highly imbalanced classes, some having less than 10 and some having more than 150 examples. In order to make sure that all the classes will be represented in the validation set I used stratified split.

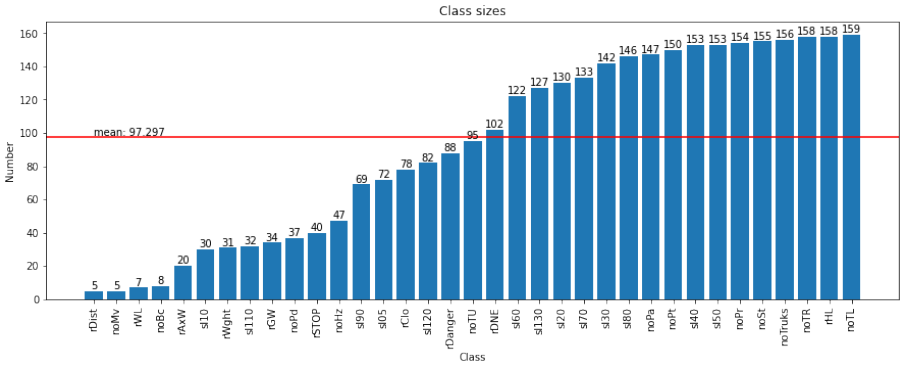

Figure 1: class imbalance

A way of dealing with class imbalance is over- and undersampling. I sampled the classes to have the same number of examples: the mean example number (but I oversampled at most 10 times to avoid overfitting). The sampling was remade randomly after every epoch, this way every element from the large classes are used during training (and not just a subsample of the classes).

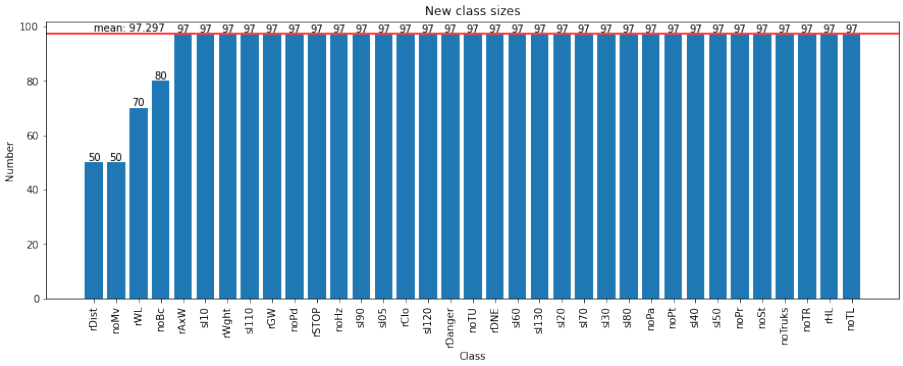

Figure 2: class sizes after resampling

To further lessen the impact of class imbalance I used heavy data augmentation and to improve the training normalized the input.
The augmentations are:

- random brightness, contrast, saturation and hue change
- random rotation
- random perspective change
- random crop

Finally, I also used FocalLoss, a variant of CrossEnropyLoss designed for detection tasks with class imbalance.

3 Metrics

As accuracy is erroneous for imbalanced classes, I used F1 score as a goal metric, and also calculated Precision and Recall. To give all classes the same weight, and avoid bias due to class imbalance I calculated them for each class separately, and averaged the metrics across classes ('macro' averaging). In order to calculate this, all predictions and groundtruths should be saved during the epoch. For large datasets this would be intractable, but for a small dataset like this it does not cause memory problems.

To easily find problems during the training I drew and saved the confusion matrix and by class F1 scores after every epoch.

As the dataset contained very similar class examples, it is possible that the validation examples are very similar to the training examples, and the model overfitted (the model learned to 99+% F1 score on the val set). It could be tested on a separated test set.

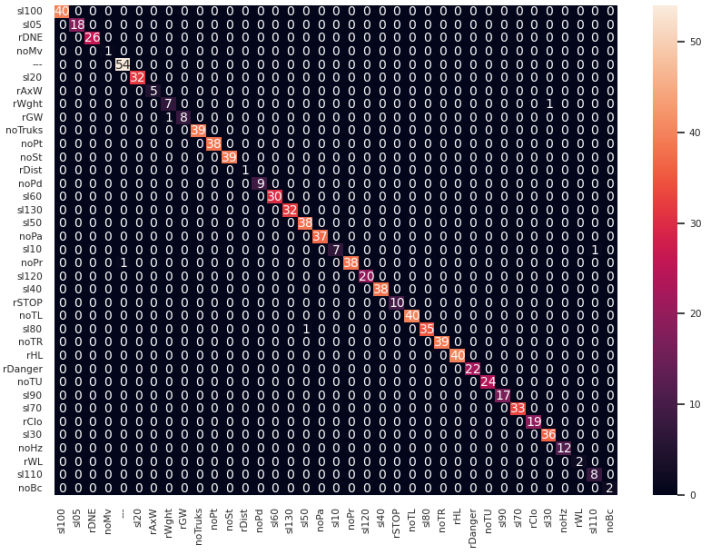

Figure 3: confusion matrix of a trained model

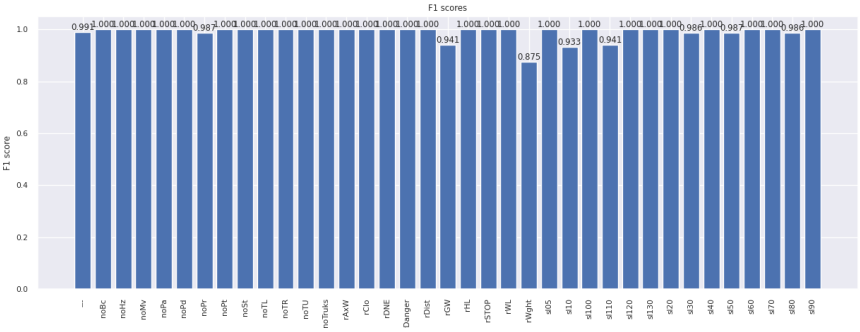

Figure 4: by class F1 score