# IN5400 - Mandatory 1 Report

Inger Annett Grünbeck - student nr. 62065

March 2021

## Task 1

The model was trained and validated on a local GPU: NVIDIA GeForce RTX 2060, with 6 GB GPU memory. The code was therefore not tested on the cluster, but worked ok on the local GPU (including the GUI). The root directories in the script were afterwards adapted to the dataset's location on the cluster (root_dir = '*/itf-fi-ml/shared/IN5400/dataforall/mandatory1/VOCdevkit/VOC2012/*'), and the local root directory used by me has been hashed out.

In order to train and validate the model, script Main_task1.py has to be run. Running Main_task1.py will train the model, validate it, save the scores, and return the learning curves (the loss and the mAP), the tailacc and the GUI of the best scoring images. **In order to run the script without calling the GUI at the end, line 375-383 have to be removed**. The GUI can also be called independently of the training, by running GUI.py. See Section 1.2 for more information on the GUI.

Main_task1.py, together with the other scripts, can be found in folder "Code". Paths (exluding the path to the VOC dataset) in the codes are all written relative to the "Code" folder. The best performing model(best_entire_model.pth) and it's state (best_model_state.pth) were stored in folder "saved_model".

In Main_task1.py, I added some new methods in addition to those already existing:

- **save_scores**: sorts the scores of the images of each class descending. The filenames of the images are also sorted based on the scores. Both the scores and filenames are saved as .npy - files in folder "saved_scores".

- **test_train_curve**: Plots either the loss curves of the test and training set, or the mAP.

- **tailacc**: calculates the average tailacc(t) of the classes for t $\in [0.5, t_{max_x f(x)}\rangle$.

## The Setup

The implementation of the last layer of the model was performed as demonstrated in earlier weekly exercises:

```
model = models.resnet18(pretrained=True)
num_ftrs = model.fc.in_features
model.fc = nn.Linear(num_ftrs, config['numcl'])
model.fc.reset_parameters()
```

The size of the input was set to be the same as the original pre-trained model's layer. The output was adjusted to the correct number of classes in the VOC dataset (=20).

The seeds and other configuration parameters were set to be:

```
np.random.seed(9400)
torch.manual_seed(9400)

config['use_gpu'] = True
config['lr']=0.005
config['batchsize_train'] = 16
config['batchsize_val'] = 64
config['maxnumepochs'] = 35
config['scheduler_stepsize'] = 10
config['scheduler_factor'] = 0.3
```

The optimizer and scheduler were defined as follows:

```
optimizer = optim.SGD(model.parameters(), lr=config['lr'], momentum=0.9)

scheduler = lr_scheduler.StepLR(optimizer=optimizer,
                                step_size=config['scheduler_stepsize'],
                                gamma=config['scheduler_factor'],
                                last_epoch=-1)
```

### The Loss Function

BCEWithLogitLoss was chosen as loss function. This loss combines a Sigmoid layer with the BinaryCrossEntropy Loss in one single layer, creating a more numerically stable version of the BinaryCrossEntropy Loss (see `https://pytorch.org/docs/stable/generated/torch.nn.BCEWithLogitsLoss.html` for more documentation). The unreduced loss can be expressed as:

$$l(x,y) = L = \{l_1, ..., l_N\}^T, l_n = -w_n[y_n * log(\sigma(x_n)) + (1 - y_n) * log(1 - \sigma(x_n))], \quad (1)$$

where N corresponds to the batch size. The applied loss in this project was not unreduced. Instead *l(x,y)* was defined as the *mean(L)*.

The code was implemented similar to pytorch's loss-function:

```
class BCEWithLogitsLoss(nn.modules.loss._Loss):

    def __init__(self, weight: Optional[Tensor] = None, size_average=None,
                 reduce=None, reduction: str = 'mean',
                 pos_weight: Optional[Tensor] = None) -> None:
        super(BCEWithLogitsLoss, self).__init__(size_average, reduce, reduction)
        self.register_buffer('weight', weight)
        self.register_buffer('pos_weight', pos_weight)

    def forward(self, input: Tensor, target: Tensor) -> Tensor:
        assert self.weight is None or isinstance(self.weight, Tensor)
        assert self.pos_weight is None or isinstance(self.pos_weight, Tensor)
        return torch.nn.functional.binary_cross_entropy_with_logits(input, target,
                                                                    self.weight,
                                                                    pos_weight=self.pos_weight,
                                                                    reduction=self.reduction)
```

## The Learning Curve, mAP and the Tail Accuracy

According to the task-instructions, the model's learning curve, mAP and accuracy of the upper tail were implemented and included in the report. The results can be viewed in the plots below. Figure 1 illustrates the loss-curves, Figure 2 the mAP of the model and Figure 3 the Tailacc(t). A mean average precision (mAP) of approximately 0.84 was achieved by the model during validation, in addition to a loss $\approx 0.08$. Further, a nearly linear increase in average tailaccuracy can be observed, proving the highest scored samples/images were mostly classified correctly.

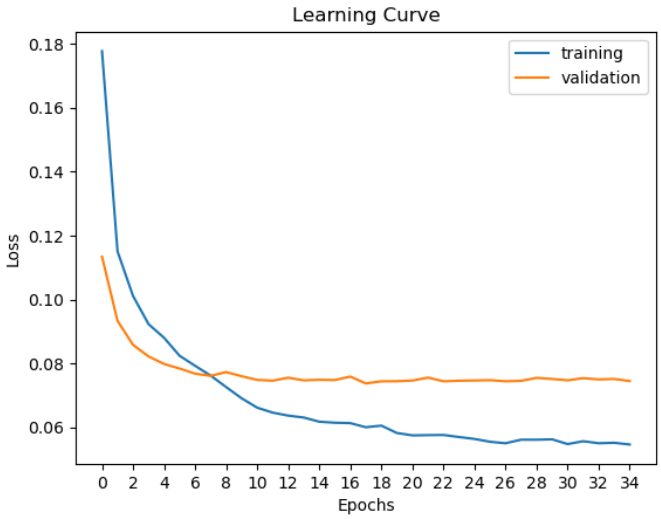


*Figure 1: The Learning curve*

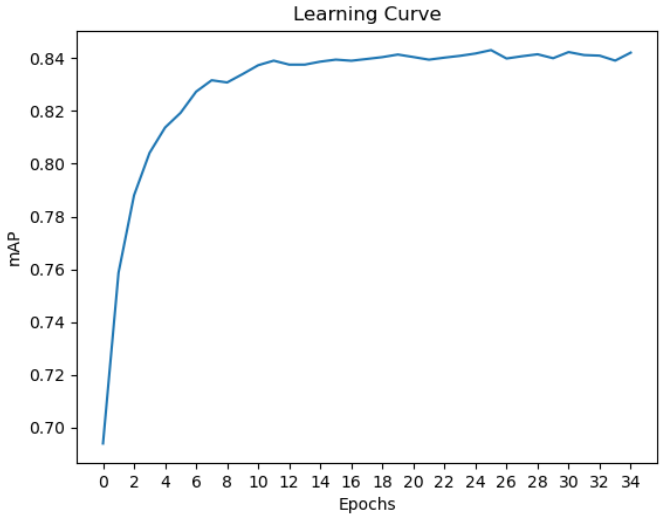


*Figure 2: The mAP of the model*

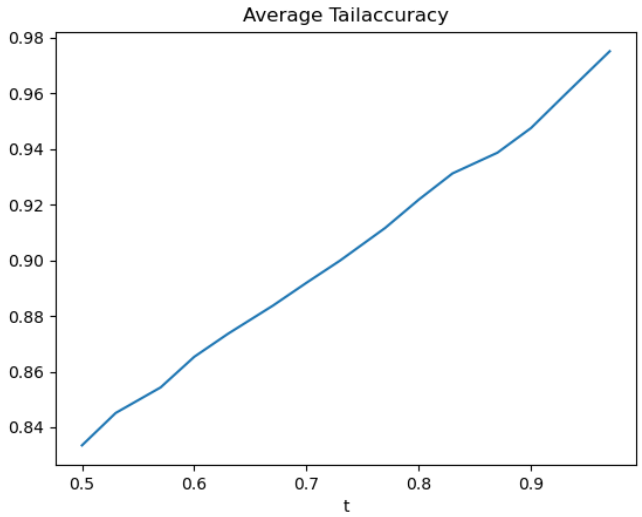


*Figure 3: The average accuracy of predictions in the upper tail across classes (Tailacc(t))*

## The GUI

The GUI was created using the PySimpleGUI - package, as recommended in the mandatory exercise. The script was based upon multiple example scripts from PySimpleGUI's documentation, and can be run by calling GUI.py. In addition to PySimpleGUI, Pillow and numpy were imported. The packages were used to handle the images, and to load and handle the scores and filenames, respectively. A requirements - file for GUI.py can be found in the zip-folder handed in. The file can be run using the command *pip install -r requirements_gui.txt* using the conda prompter (or similar).

When called, the GUI lets you choose which class to display (all classes are available). The ten highest and lowest scoring images from the chosen class will be displayed. Below the

images of three arbitrary classes were included.

**Label: Horse**

*Figure 4: Top ten highest scoring images in the horse category*

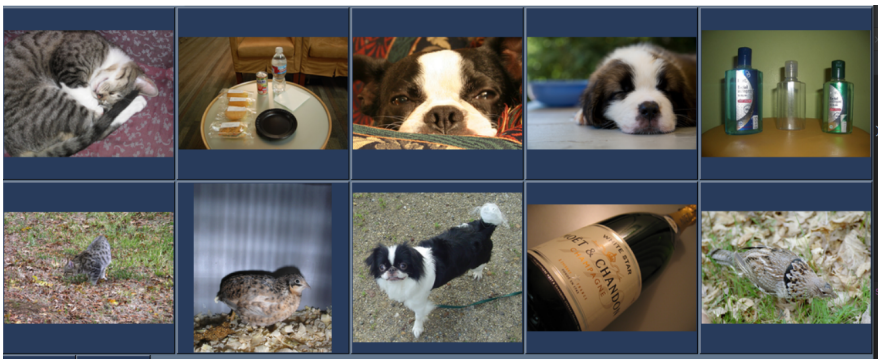

*Figure 5: The ten lowest scoring images in the horse category*

**Label: Potted plant**

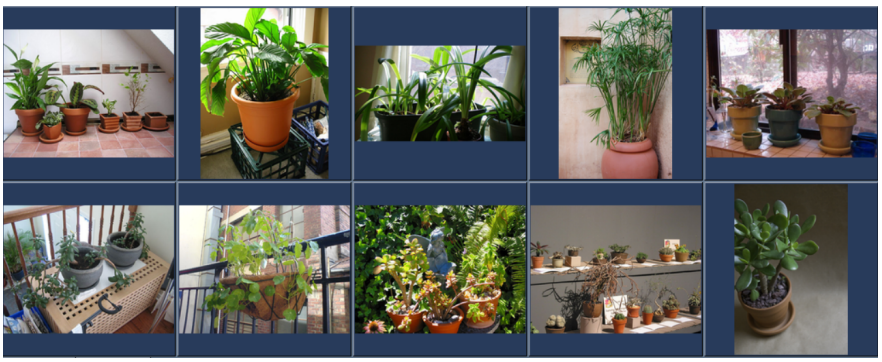

*Figure 6: Top ten highest scoring images in the potted plant category*

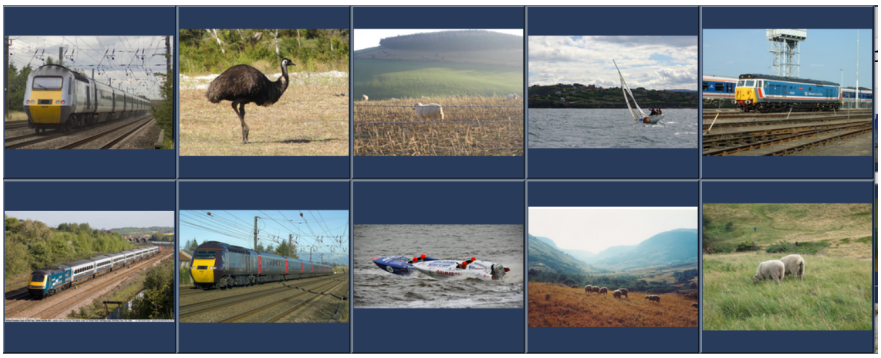

*Figure 7: The ten lowest scoring images in the potted plant category*

**Label: People**

*Figure 8: Top ten highest scoring images in the people category*

*Figure 9: The ten lowest scoring images in the people category*

# Task 2

This task was attempted, but not completed. The report will try and explain what was done, and parts of the thought process. The explanation includes the code parts that were written. The started on code can also be found in file "Main_task2.py" on folder "Code".

## The Weighted Convolution

The new convolution layer, which standardized the original weights, was implemented:

```
class wsconv2(nn.Conv2d):
    def __init__(self, in_channels, out_channels, kernel_size, stride,
                 padding, dilation=1, groups=1, bias=None,
                 copied_weights=None, copied_bias=None, eps=1e-12):
        super(wsconv2, self).__init__(in_channels, out_channels, kernel_size,
        stride, padding, dilation, groups, bias)

        self.eps =eps
        self.weights = copied_weights
        self.bias = copied_bias



    def forward(self, x):
        weight = self.weights
        std = weight.std(dim=(1, 2, 3))
        weight = weight / (sqrt(std**2 + self.eps))
        return torch.nn.functional.conv2d(x, weight, self.bias,
        self.stride, self.padding, self.dilation, self.groups)
```

Further, the first part of bntoWSconverter was implemented, passing the original convolutions weights, bias and input to the new convolution:

```
        if isinstance(module, nn.Conv2d):

            lastwasconv2= True

            usedeps= 1e-12

            module_weights = module.weight
            module_bias = module.bias

            newconv = wsconv2(in_channels=module.in_channels,
            out_channels=module.out_channels, kernel_size=module.kernel_size,
            stride=module.stride, padding=module.padding,
            dilation=module.dilation, groups=module.groups, bias=module.bias,
            eps=usedeps, copied_weights=module_weights, copied_bias=module_bias)

            setbyname2(model, nm, newconv)
```

## The Batch Normalization

The actual batch normalization was not implemented, as I did not finish up the new expressions $\hat{\alpha}, \hat{\beta}, \hat{\sigma}$ and $\hat{\mu}$. I tried to determine the definitions by solving

$$y(W, \alpha, \beta, \sigma, \mu) = y(\hat{W}, \hat{\alpha}, \hat{\beta}, \sigma, \mu). \tag{2}$$

(And the equivalent for $\hat{\sigma}, \hat{\mu}$.) I was able to determine expressions for these variables, but not by exclusively using their original counterpart and $n_c$. Multiple other variables were involved:

Case $(\hat{\alpha}, \hat{\beta})$:

$$\hat{\alpha} = (\alpha * n_c(\mu - z))/(\mu * n_c - z), \tag{3}$$

and

$$\hat{\beta} = \beta. \tag{4}$$

Case $(\hat{\sigma}, \hat{\mu})$:

$$\hat{\mu} = \mu + (1/n_c - 1) * z, \tag{5}$$

and

$$\hat{\sigma} = \sigma. \tag{6}$$

As these expressions didn't seem correct to me, I did not finish the implementation. A step by step solution on the thought process for this task would be much appreciated, as it is an interesting topic!