TDT4265 Computer Vision and Deep Learning

Trondheim, 2022

Eivind Dogger
eivinddo@stud.ntnu.no

Aleksander Vangen
aleksava@stud.ntnu.no

Ørjan Carlsen
orjanic@stud.ntnu.no

Preface

The project delivery consists of a video, source code and this report. In the project description, several tasks were requested to be included in the video, which was specified to have a maximum length of 8 minutes. Due to the time limit of the video, we couldn't provide a detailed explanation of all tasks included in the video. Thus, it was decided to use more time of the video for the tasks only included in the video, while there is included a more detailed analysis in the report for the tasks included in both the video and report. This is reflected through the fact that Task 1.1 has a more detailed analysis in the video, than the other tasks.

Contents

Part 2 - Model Creation **1**
Task 2.1 - Creating your First Baseline 1
Task 2.2 - Augmenting the Data 3
Task 2.3 - Implementing RetinaNet 4
Task 2.3.1 Feature Pyramid Network 4
Task 2.3.2 Focal Loss 6
Task 2.3.3 Deeper Convolutional Regression and Classification Heads 8
Task 2.3.4 Weight and Bias Initialization 10
Task 2.4 - Using the Exploration Knowledge 12
Tuning α 12
Tuning aspect ratios and sizes 13
Different number of aspect ratios 18
Task 2.5 - Extending the Dataset 20

Part 3 - Discussion and Evaluation **22**
Task 3.2 - Discussion and Qualitative Analysis 22
Data Augmentation 22
Focal Loss 23
Deeper Convolutional Heads 25
Task 3.3 - Final Discussion 25

Part 4 - Going Beyond **27**
Task 4.1 - Implementing BiFPN 27
Task 4.3 - Following your own idea 29
Task 4.4 - Comparing to State-of-the-Art 32

References **35**

Part 2 - Model Creation

Task 2.1 - Creating your First Baseline

The model from assignment 4 is adjusted to support a image resolution of 128x1024. In `configs/ssd300.py` the specifications of the anchors are changed into:

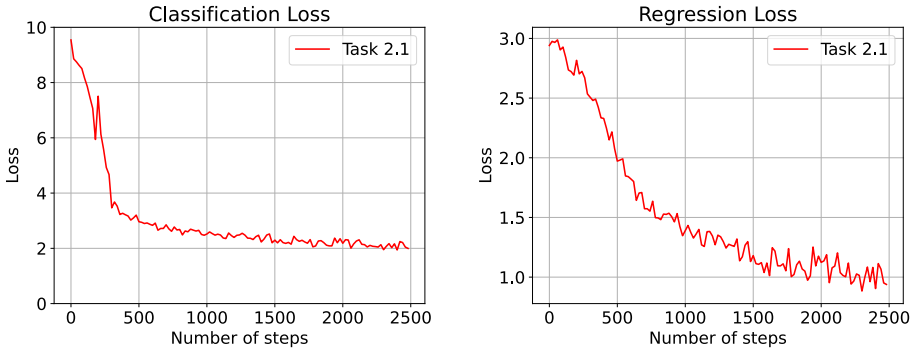
- `feature_sizes=[[32,256], [16,128], [8,64], [4,32], [2,16], [1,8]]`
- `strides=[[4,4], [8,8], [16,16], [32,32], [64,64], [128,128]]`
- `min_sizes=[[16,16], [32,32], [48,48], [64,64], [86,86], [128,128], [128,400]]`
- `aspect_ratios=[[2,3], [2,3], [2,3], [2,3], [2], [2]]`

There have also been minor modifications to the `BasicModel` in `ssd/modeling/bacbones/basic.py` to adjust the backbone model to the new image resolution. A technical description of the backbone-structure is shown in table 1. The config file for running this task is `configs/tdt4265.py`.

Table 1: Overview of the basic model used in Part 2 - Model Creation.

Layer	Filter	Output chans	Size	Stride	Pad	Activation
0	Conv2D	32	3x3	1	1	ReLU
	MaxPool	32	2x2	2	-	-
	Conv2D	64	3x3	1	1	ReLU
	Conv2D	128	2x2	2	0	ReLU
1	Conv2D	128	3x3	1	1	ReLU
	Conv2D	256	4x4	2	1	ReLU
2	Conv2D	256	3x3	1	1	ReLU
	Conv2D	128	4x4	2	1	ReLU
3	Conv2D	128	3x3	1	1	ReLU
	Conv2D	128	4x4	2	1	ReLU
4	Conv2D	128	3x3	1	1	ReLU
	Conv2D	64	2x2	2	0	ReLU
5	Conv2D	64	2x2	1	1	ReLU
	Conv2D	64	2x2	2	0	ReLU

The model was trained for 50 epochs and in figures 1a and 1b the classification and regression loss for this model is showed. In addition, a graph of the mAP@0.5:0.95 is showed in figure 2.



(a) Classification loss. (b) Regression loss.

Figure 1: Losses for the baseline model in Task 2.1 - Creating your First Baseline.

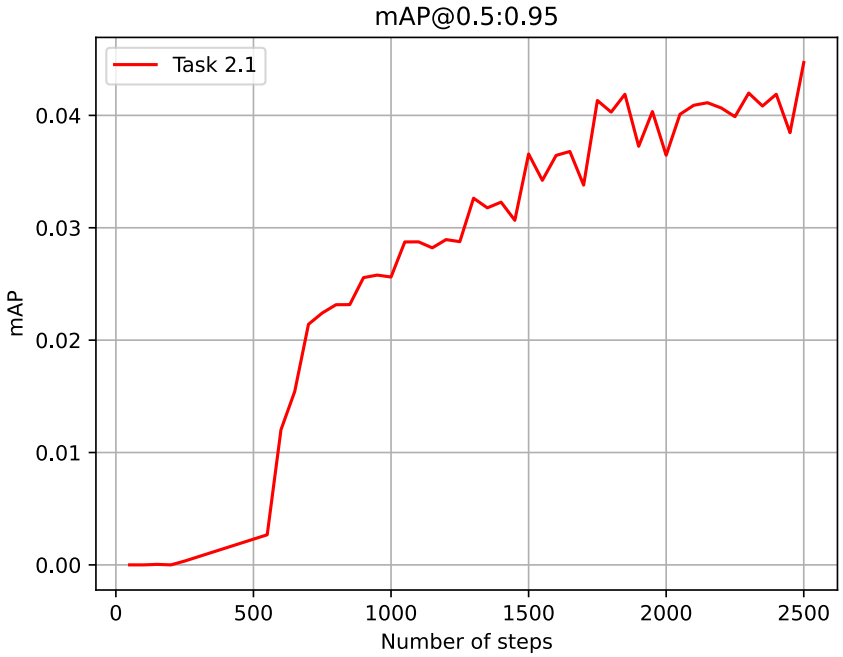


Figure 2: mAP@0.5:0.95 for the baseline model in Task 2.1 - Creating your First Baseline.

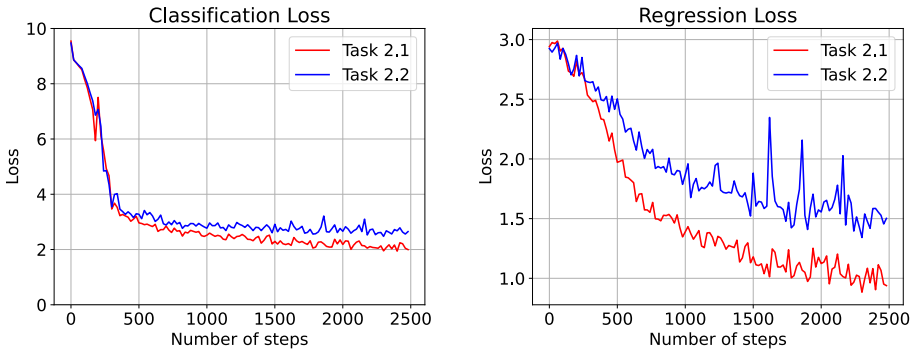
The resulting peak and average mAP is as shown in table 2. Table 2 also shows the average frames per second the network can handle after passing 100 frames through it, and the number of parameters.

Table 2: Statistics for the trained network from Task 2.1 - Creating your First Baseline.

mAP@0.5:0.95 (final)	0.0447
mAP@0.5:0.95 (peak)	0.0447
FPS	8.4393
Parameters	3 266 272

Task 2.2 - Augmenting the Data

The implementation of data augmentation for the training set is made in `configs/task2_2.py`. `RandomSampleCrop`, `RandomHorizontalFlip`, `RandomContrast`, `RandomBrightness`, which all are implemented in `ssd/data/transforms/transform.py`, were used in the data augmentation. The resulting loss plots are shown in figures 3a and 3b and the mean average precision is shown in figure 4. Table 2 summarizes the performance of the model. It can be seen that the model with data augmentation has a worse mean average precision. However, all the augmentations were kept in the model. A deeper analysis of this choice and the consequences is provided in Task 3.2 - Discussion and Qualitative Analysis.



(a) Classification loss.

(b) Regression loss.

Figure 3: Losses for the model with data augmentation in Task 2.2 - Augmenting the Data, compared to the model without data augmentation from Task 2.1 - Creating your First Baseline.

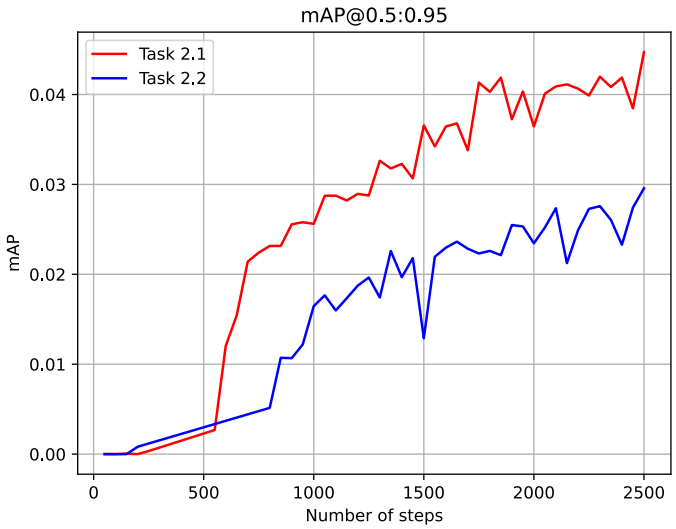


Figure 4: mAP@0.5:0.95 for the baseline model in Task 2.2 - Augmenting the Data, compared to the model without data augmentation from Task 2.1 - Creating your First Baseline.

Table 3: Statistics for the trained network from Task 2.2 - Augmenting the Data.

mAP@0.5:0.95 (final)	0.0296
mAP@0.5:0.95 (peak)	0.0296
FPS	8.3811
Parameters	3 266 272

Task 2.3 - Implementing RetinaNet

In this section, improvements to the network have been made iteratively, allowing an independent evaluation of each improvement.

Task 2.3.1 Feature Pyramid Network

A feature pyramid network, based on a pretrained Resnet34 model, was the first improvement to be made. The implementation of the feature pyramid network can be seen in `ssd/modeling/backbones/fpn.py`. All other parts of the detector was kept at the defaults from the SSD model. The config file for running this iteration is in `configs/task2_3_v1.py`.

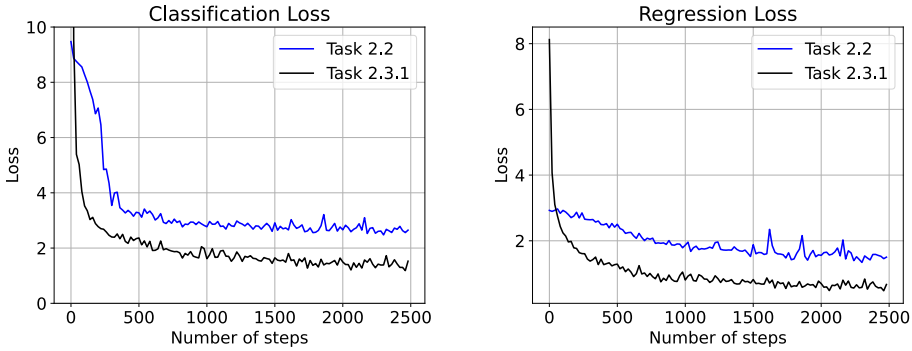
We wanted 6 output features from the model, but since the Resnet34 only has 4 we added two small layers as shown in table 4. The number of output channels for the last two layers were based on the number of

output channels in the Resnet34 model. Since it had the output channels `[64, 128, 256, 512]`, the pattern was continued and made the number of output channels of the last two layers 1024 and 2048.

Table 4: Layer 5 and Layer 6 that are added as additional layers to ResNet34 in order to get a 6 layer model.

Layer	Filter	Output channels	Size	Stride	Pad	Activation
5	Conv2D	512	3x3	1	1	ReLU
	Conv2D	1024	3x3	2	1	ReLU
6	Conv2D	1024	3x3	1	1	ReLU
	Conv2D	2048	3x3	2	1	ReLU

In order to implement the feature pyramid network, the function in PyTorch, `torchvision.ops.FeaturePyramidNetwork()` was used. The output channels was set to $C = 256$ from the feature pyramid network as in [1]. The resulting losses are shown in figures 5a and 5b, and the resulting mean average precision is shown in figure 6. The new network has a much higher increase in mAP early in the training. This is due to the ResNet34 in the backbone being pretrained.



(a) Classification loss.

(b) Regression loss.

Figure 5: Losses for the model from Task 2.3.1 Feature Pyramid Network, compared to the model from Task 2.2 - Augmenting the Data.

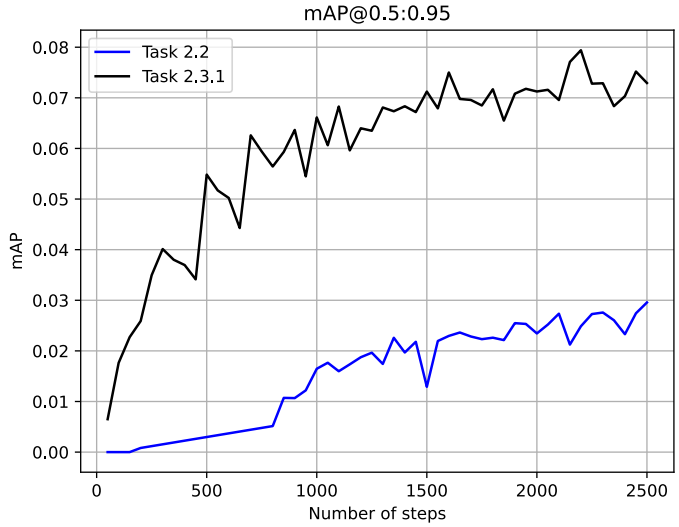


Figure 6: mAP@0.5:0.95 for the model from Task 2.3.1 Feature Pyramid Network, compared to the model from Task 2.2 - Augmenting the Data.

The resulting statistics for this configuration is shown in table 5. It is worth mentioning the increase in number of parameters. This allows for better precision as the model is able to pick up more complex patterns, but does also decrease the inference speed significantly.

Table 5: Statistics for the trained network from Task 2.3.1 Feature Pyramid Network.

mAP@0.5:0.95 (final)	0.0729
mAP@0.5:0.95 (peak)	0.0794
FPS	4.3091
Parameters	62 473 568

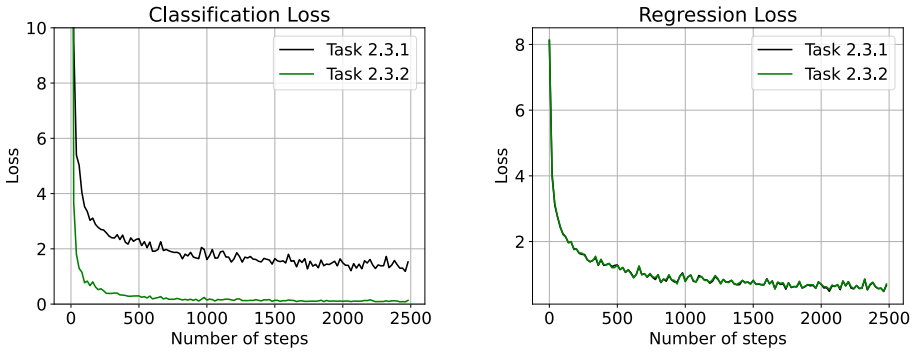
Task 2.3.2 Focal Loss

The goal of focal loss is to focus the training on misclassified examples, by reducing the loss from well-classified samples. The code is presented in `ssd/modeling/ssd_focal_loss.py`, and the config file is given in `configs/task2_3_v2.py`. The focal loss replaces the hard-negative mining and cross-entropy loss used in the earlier tasks, and it is calculated as shown in equation 1.

$$\mathrm{FL}(p, y) = -\sum_{k=1}^{K} \alpha_k \cdot (1 - p_k)^{\gamma} \cdot y_k \cdot \log(p_k) \tag{1}$$

The α_k-parameter, used to adjust the relative weighting on the loss between the different classes, is chosen to `alpha=[10, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000]`. This gives the background class a weight of one hundredth of the other classes, to adjust for the class imbalance. Furthermore, it is used `gamma=2` to reduce the loss from well-classified examples, as this is the default setting used in [1].

The regression loss, shown in figure 7b, is more or less unaffected by the change of `loss_objective`, as the focal loss only replaces the classification loss. The introduction of focal loss improved mAP from 7.29% to 8.17%, observed in figure 8. Because the regression loss only has marginal deviations from the model in Task 2.3.1 Feature Pyramid Network, the improved mAP is mainly due to a better classification, rather than a better detection. The implementation does not change the number of parameters, and obviously has minimal effect on inference speed. This is shown in table 6. A deeper analysis is included in Task 3.2 - Discussion and Qualitative Analysis.



(a) Classification loss.

(b) Regression loss.

Figure 7: Losses for the model from Task 2.3.2 Focal Loss, compared to the model from Task 2.3.1 Feature Pyramid Network.

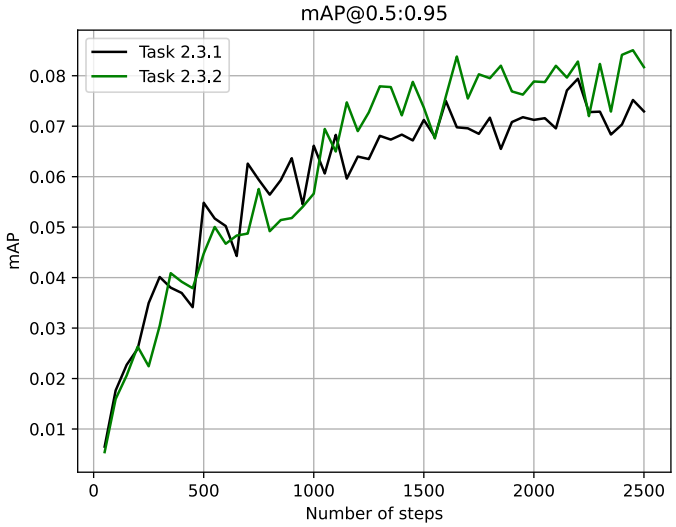


Figure 8: mAP@0.5:0.95 for the model from Task 2.3.2 Focal Loss, compared to the model from Task 2.3.1 Feature Pyramid Network.

Table 6: Statistics for the trained network from Task 2.3.2 Focal Loss.

mAP@0.5:0.95 (final)	0.0817
mAP@0.5:0.95 (peak)	0.0851
FPS	4.3083
Parameters	62 473 568

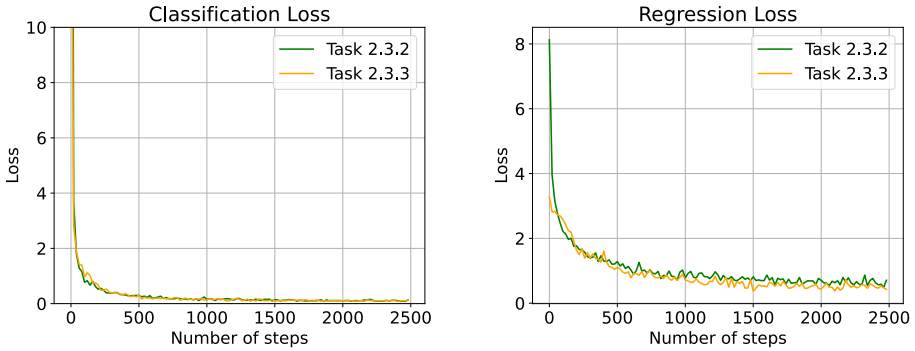
Task 2.3.3 Deeper Convolutional Regression and Classification Heads

In this section, the single-layer regression and classification output heads were replaced with deeper convolutional networks. The code is implemented in `ssd/modeling/retina_net.py`, and the config file is `configs/task2_3_v3.py`. The `SSD300` model is replaced with the new `RetinaNet` model.

Two new output heads are designed, one for the regression and one for the classification. These two new sub-networks are used on all feature map outputs from the FPN, and implemented according to [1]. The only difference between our implementation and the suggested implementation in the paper is that the final *sigmoid* is dropped.

The losses of the new model are plotted against the old model in figures 9a and 9b. As seen from figure 9b, this makes the initial regression loss a lot lower. For most of the training however, the losses are very similar to the ones in Task 2.3.2 Focal Loss. The extra convolutional layers gives a

slightly slower training the first epochs, but the new model quickly surpasses the previous model in terms of mAP, as seen in figure 10. The new output heads increases the size of the model, which affects the inference speed, as seen in table 7. A deeper analysis is included in Task 3.2 - Discussion and Qualitative Analysis.



(a) Classification loss.

(b) Regression loss.

Figure 9: Losses for the model from Task 2.3.3 Deeper Convolutional Regression and Classification Heads, compared to the model from Task 2.3.2 Focal Loss.

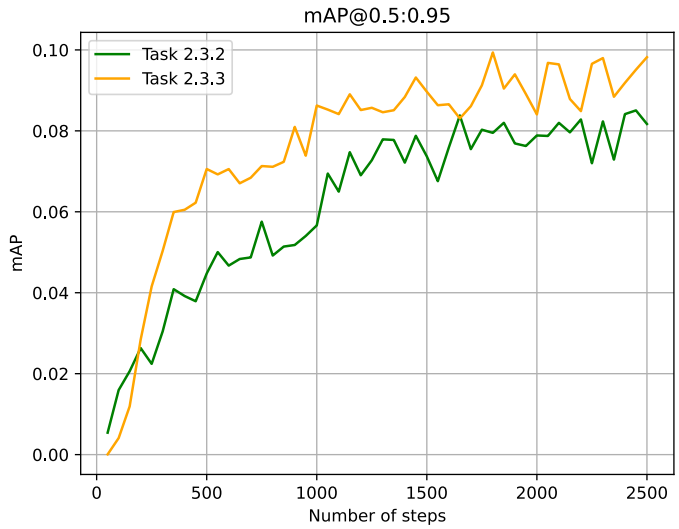


Figure 10: mAP@0.5:0.95 for the model from Task 2.3.3 Deeper Convolutional Regression and Classification Heads, compared to the model from Task 2.3.2 Focal Loss.

Table 7: Statistics for the trained network from task Task 2.3.3 Deeper Convolutional Regression and Classification Heads.

mAP@0.5:0.95 (final)	0.0982
mAP@0.5:0.95 (peak)	0.0994
FPS	1.7446
Parameters	66 415 438

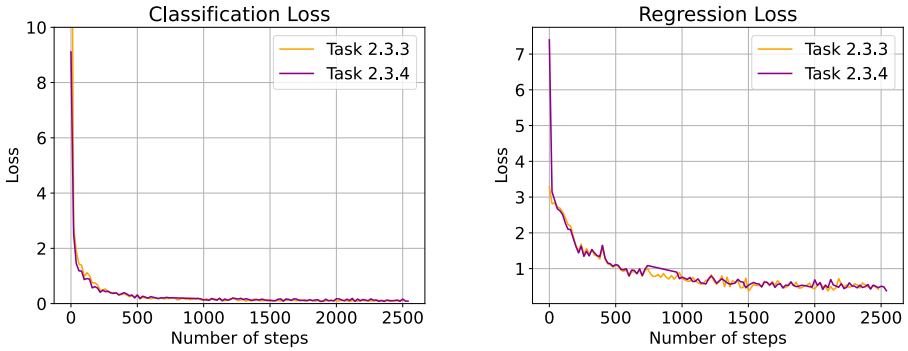
Task 2.3.4 Weight and Bias Initialization

Weight initialization is introduced to improve the performance of the model in the initial stages. The code is implemented in `ssd/modeling/retina_net.py`, and the config file is `configs/task2_3_v4.py`.

The weight initialization of the convolutional heads was implemented slightly differently from the approach described in [1], but the key ideas are kept. Instead of using normal distribution with $\sigma = 0.01$, as suggested in [1], we used Kaiming initialization of the weights. It is stated in [2] that this approach is better suited for deep layers than a Gaussian distribution. In addition, the authors of [2] point out that the Kaiming initialization is better for nonlinear activation functions, than the Xavier initialization that was in the starter code. Thus, there was made a choice to use the Kaiming initialization. The weights of the last layer for both the classification head and the regression head are also initialized, even though this is not specified in [1]. This choice was made as it reduced the initial loss. Finally, the bias of the background class in the final layer of the convolutional head was set according to equation 2, which is different from the formula stated in [1]. However, both formulas are introduced to prevent the large number of background classes to give a potentially destabilizing classification loss in the first iterations of training.

$$b = \log\left(p \cdot \frac{K-1}{1-p}\right) \tag{2}$$

Showed in figure 11a, the initial classification loss is decreased with the weight initialization. With the model from Task 2.3.3 Deeper Convolutional Regression and Classification Heads the classification loss was ≈20.82 for the first iteration, while the classification is ≈9.11 with the model from Task 2.3.4 Weight and Bias Initialization. This indicates that the weight initialization works. Similar results can also be observed for the mean average precision, plotted in figure 12. The mAP after the first epoch is ≈0.00006 figure 10 without weight initialization, and ≈0.00132 with weight initialization.



(a) Classification loss. (b) Regression loss.

Figure 11: Losses for the model from Task 2.3.4 Weight and Bias Initialization, compared to the model from Task 2.3.3 Deeper Convolutional Regression and Classification Heads.

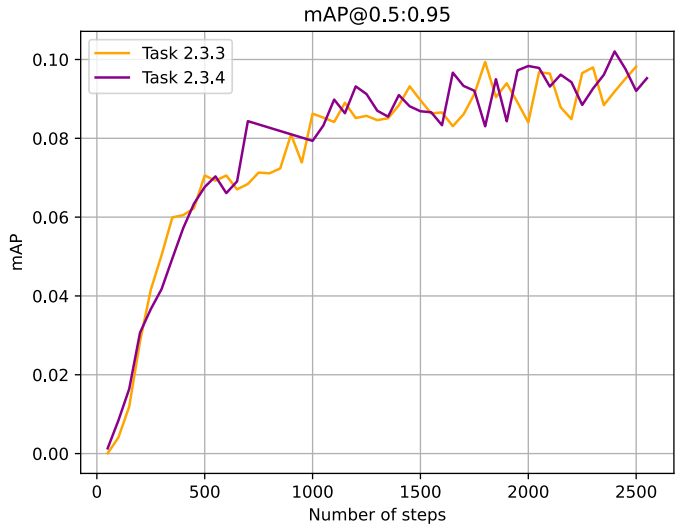


Figure 12: mAP@0.5:0.95 for the model from Task 2.3.4 Weight and Bias Initialization, compared to the model from Task 2.3.3 Deeper Convolutional Regression and Classification Heads.

Table 8: Statistics for the trained network from Task 2.3.4 Weight and Bias Initialization.

mAP@0.5:0.95 (final)	0.0953
mAP@0.5:0.95 (peak)	0.1020
FPS	1.7558
Parameters	66 415 438

Task 2.4 - Using the Exploration Knowledge

All improvements discussed in this section are applied to the final `RetinaNet` from Task 2.3 - Implementing RetinaNet.

In order to specialize the model to the dataset, several approaches was tried, but we struggled to find changes that actually improved the model. Only one of the changes, when trained on the large dataset, gave a slight improvement. The main changes that were tried was tuning the α in the equation 1 since there are a lot more cars and persons than i.e. busses. Another change was based on the aspect ratios of the anchor boxes, and trying to find a combination of aspect ratios that better resembles the actual aspect ratios in the dataset. In addition to this, the size of the different anchor boxes were changed. Finally, a more complex model for the `AnchorBoxes` class, and classification and regression heads were implemented. The following subsections will go through the different improvements that was tried.

Tuning α

In equation 1 the α parameter is originally set to 10 for the background class and 1 000 for the other classes. This is to get a lower loss value for the background class as it is the dominating class (much more background than labelled objects). However, as can be seen in figure 13, the non-background classes aren't evenly distributed themselves either. This led to the theory that one could reduce α-value for car and person since there is a lot more of these classes than the others. This change was just performed temporarily by modifying the `SSDFocalLoss` class in `ssd_focal_loss.py`.

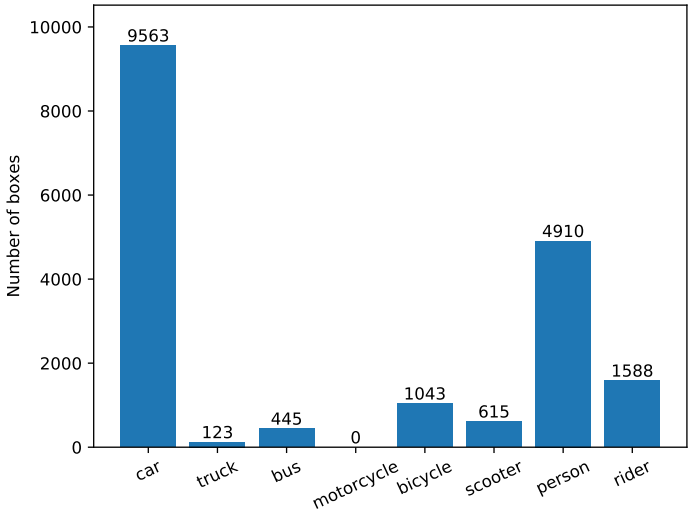


Figure 13: Distribution of the different classes in the non-updated dataset.

The first attempt was at setting $\alpha_{car} = 800$ and $\alpha_{person} = 900$. This resulted in the mAP values given in table 9, which are a bit worse than before changing the alpha's, but the performance for the two classes was mostly the same.

Max mAP	0.0971
Final mAP	0.0971

Table 9: mAP after training a model with lowered α-weighting for the car and person classes.

Since the first attempt ended in a bit worse results than the final RetinaNet implementation in Task 2.3.4 Weight and Bias Initialization, another attempt was setting $\alpha_{car} = 1\,100$ and $\alpha_{person} = 1\,050$. This increases the weight, and therefore loss, of these classes. The maximum mAP achieved is listed in table 10, which is a bit higher than in table 8. However, the increase is so little that we concluded that these small changes to α didn't have any significant effect. By plotting the average precision (AP) for the car class, which had the largest change in α, one can also see that the change had little to no effect. This plot is shown in figure 14

Tuning aspect ratios and sizes

All previous models had been implemented with aspect ratios of 1:2, 2:1, 1:3, 3:1 and two anchor boxes with aspect ratio 1:1. In addition, the variable `min_sizes` was unchanged from the basic SSD300 model. In order to get a

Max mAP	0.1050
Final mAP	0.0928

Table 10: mAP after training a model with increased α-weighting for the car and person classes.

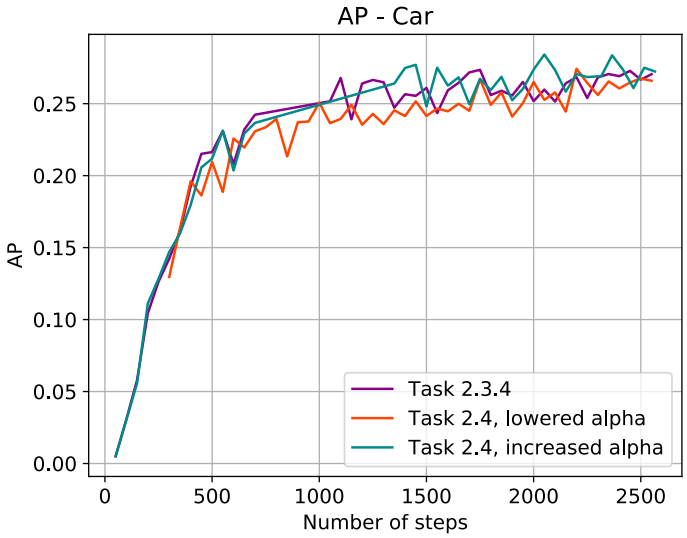


Figure 14: AP for the car class with different α-weight for the car and person classes.

grasp of what modifications should be done to improve the anchor boxes, we made a comparison of the anchor boxes created by the model and the ground truth bounding boxes. Table 11 shows the size and area of the different anchor boxes in the model from Task 2.3.4 Weight and Bias Initialization. These values are generated using `scripts/analyze_anchor_box_coverage.py`. As one can see, the smallest anchor boxes that are created are 256px^2. In figure 15, the average size of the anchor boxes for each of the classes is shown. Based on this, one can see that for the person class, the average bounding box size is smaller than the smallest anchor boxes. In addition, the larger classes, trucks and busses, have average sizes of 5700px^2 and 4400px^2 respectively, which is much less than the larges feature maps in table 11. This pointed in the direction that more of the anchor boxes would fit better if they were all a bit smaller.

In addition to changing the `min_sizes` parameter to adjust the size of the anchor boxes, the aspect ratios had to be fitted for the dataset as well. They could be adjusted with different aspect ratios for different feature maps in order to have some feature maps with small and narrower boxes, for i.e. persons, and other for wide and large objects, like trucks.

Table 11: Sizes of anchor boxes for the model developed in Task 2.3.4 Weight and Bias Initialization.

Feature map	**Box number**	**(width, height)**	**Area**
Feature map 0	Box 1	(16.0, 16.0)	256.0
	Box 2	(22.63, 22.63)	512,0
	Box 3	(11.31, 22.63)	256,0
	Box 4	(22.63, 11.31)	256,0
	Box 5	(9.24, 27.71)	256,0
	Box 6	(27.71, 9.24)	256,0
Feature map 1-4	⋮	⋮	⋮
Feature map 5	Box 1	(128.0, 128.0)	16 384,0
	Box 2	(226.27, 128.0)	28 963,1
	Box 3	(90.51, 181.02)	16 384,0
	Box 4	(181.02, 90.51)	16 384,0
	Box 5	(73.9, 221.7)	16 384,0
	Box 6	(221.7, 73.9)	16 384,0

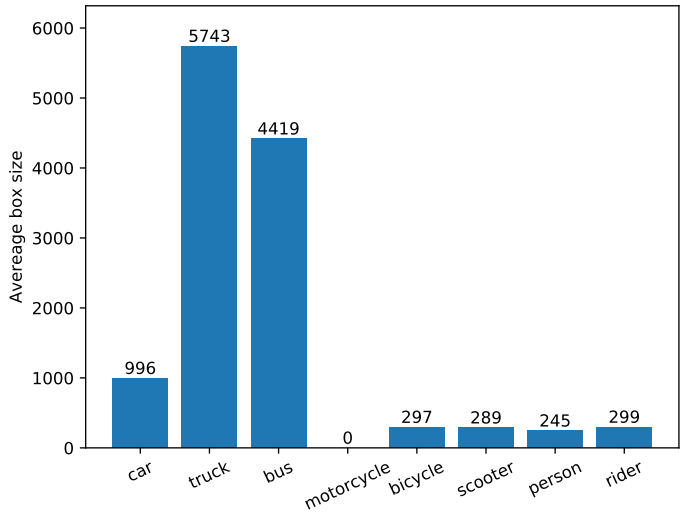
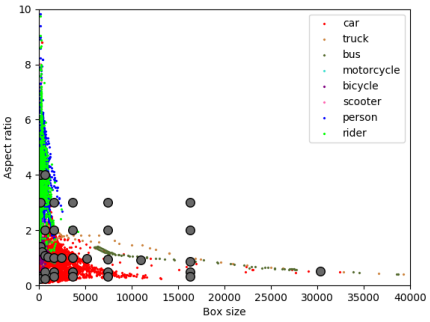
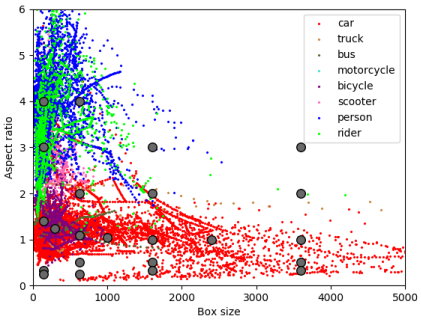


Figure 15: The average box size for the different classes.

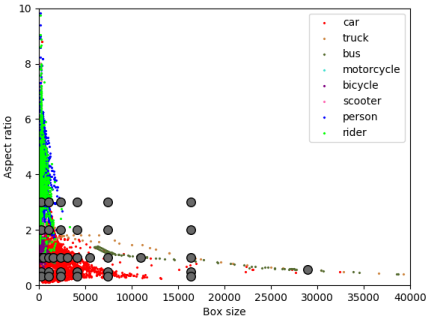
In order to get an intuition of how to adjust the different sizes and aspect ratios, a plot of the aspect ratios versus the bounding box size was created. The plot was created as part of the dataset exploration, and now overlaid with the anchor boxes used. This is created by running `scripts/analyze_anchor_box_coverage.py`. The anchor boxes of the feature map was tuned by changing `aspect_ratios` and `min_sizes` in order to get a better fit over the distribution of the dataset. Figure 16 shows the difference between the old and the modified anchor boxes.



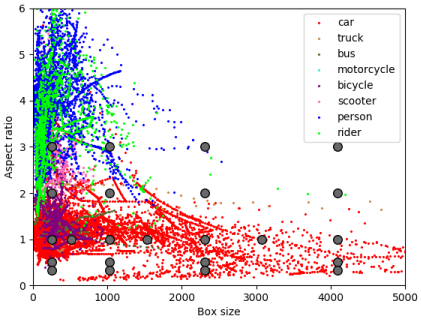
(a) Improved anchor boxes.



(b) Improved anchor boxes, zoomed.



(c) Original anchor boxes.



(d) Original anchor boxes, zoomed.

Figure 16: The distribution of ground truth box size and aspect ratio overlaid by the box size and aspect ratio of the anchor anchor boxes (gray/black dots).

The improved version in figures 16a and 16b has anchor boxes and mean sizes given by:

- `min_sizes=[[14, 10], [26, 24], [40, 40], [60, 60], [84, 88], [120, 136], [134, 420]]`
- `aspect_ratios=[[3, 4], [2, 4], [2, 3], [2, 3], [2, 3], [2, 3]]`

Since the `min_sizes` variable no longer has the same value for width and height for all feature maps, the boxes in the first and second feature map do not have 1:1 boxes. This is seen in figure 16b, where there is a curve to the anchor boxes in the lower left corner. This was done to better fit the data in this region, as the aspect ratio of the first feature map was changed from [2,3] to [3,4]. From the color-labeling in the plot, the aspect ratios line up with the bicycle and rider classes, so we had a hypothesis that this change would also increase the detection of these classes. The config file for testing this implementation is `configs/task2_4_min_sizes.py`.

There are no bicycles in the validation set, so no improvement could be seen here. However, the rider class had a significant improvement compared to the network from Task 2.3.4 Weight and Bias Initialization, as shown in figure 17. This came at the cost of reduced average precision for the person and bus classes. The person class was the most surprising, as the anchor boxes in figure 16 was moved more towards the blue dots (persons), but its average precision was still reduced. The reduction in AP for bus and person is shown in figures 18a and 18b. In the end, the mAP ended up worse than before implementing these hypothesized improvements, and the result can be seen in figure 19.

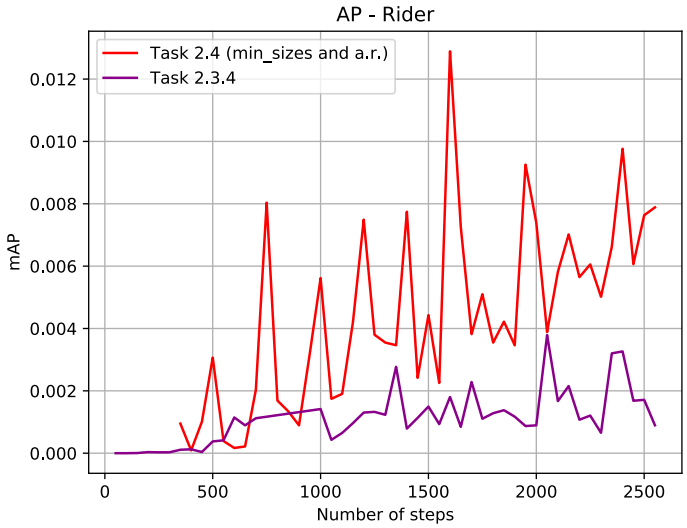
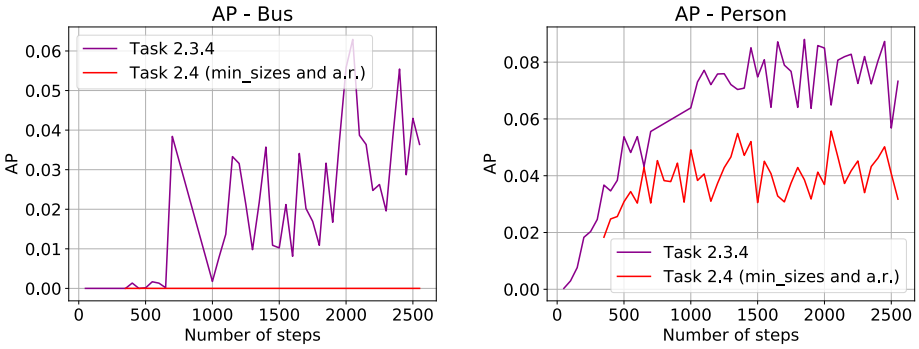


Figure 17: Average precision for the rider class. Comparison of the model from Task 2.3.4 Weight and Bias Initialization, with and without specialized anchor boxes.



(a) Class bus. (b) Class person.

Figure 18: Average precision for the bus and person classes. Comparison of the model from Task 2.3.4 Weight and Bias Initialization, with and without specialized anchor boxes.

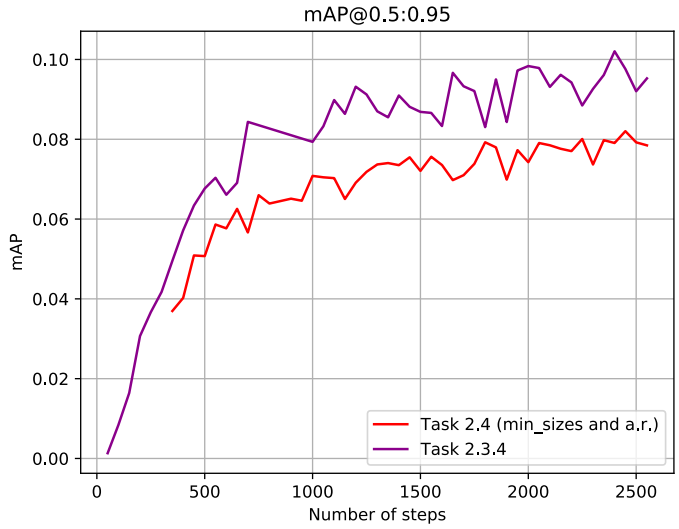


Figure 19: mAP comparison of the model from Task 2.3.4 Weight and Bias Initialization, with and without specialized anchor boxes.

Different number of aspect ratios

From figure 16 one can see that several of the combinations of aspect ratios and sizes are in areas where there are no similar bounding boxes in the dataset. In case these anchor boxes led to false positives in the network, we tried to remove these anchor boxes. Since this led to different number of anchor boxes per feature map, the classification and regression heads

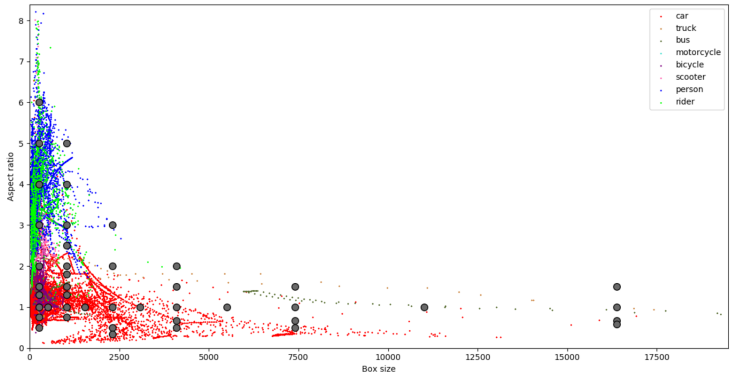


Figure 20: Scatter plot showing the ratio between box sizes and aspect ratio of the different classes, as well as the anchor boxes for the model in black.

had to be changed as well. A second classification and regression head was implemented in addition to the one developed in Task 2.3.3 Deeper Convolutional Regression and Classification Heads. Then there were no longer one common classification/regression head for the last layer of all feature maps, but one for feature maps with A_1 anchor boxes and another for feature maps with A_2 anchor boxes. This implementation was done in the class `RetinaNet2` in `ssd/modeling/retina_net2.py`. The file `ssd/modeling/anchor_boxes2.py` was also created as a modified version of the original anchor boxes.

This allowed the removal of some of the anchor boxes in some of the aspect ratios in order to i.e. not look for high and narrow object in the feature maps detecting large objects.

In addition the aspect ratios of the larger size boxes were shrunk in to fit the model better as seen in figure 20, and the aspect ratios used was

```
aspect_ratios=[[0.5, 0.75, 1.3, 1.5, 2, 3, 4, 5, 6],
    [0.75, 1.3, 1.5, 1.8, 2, 2.5, 3, 4, 5], [2, 3], [1.5, 2],
    [1.5, 2], [1.2, 1.5]]
```

From before, the class `AnchorBoxes` created boxes in aspect ratios 1:2 and 2:1 when given an aspect ratio of 2. `AnchorBoxes2` however is manually designed to only include one of 1:2 or 2:1 for some of the aspect ratios since both sides are not necessary for all feature maps. The config file for running this model is `configs/task2_4_ar.py`.

The new model did perform slightly better than the model from Task 2.3.4 Weight and Bias Initialization as seen in figure 21, where both were trained on the larger dataset. However the performance got worse for small and medium objects as seen in figure 22

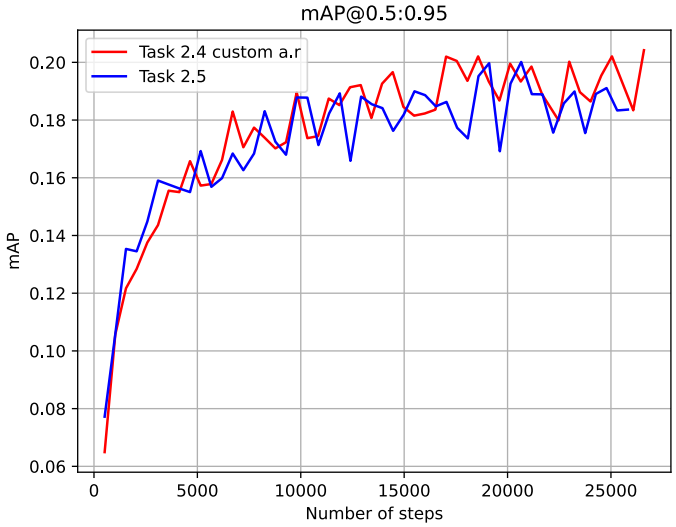
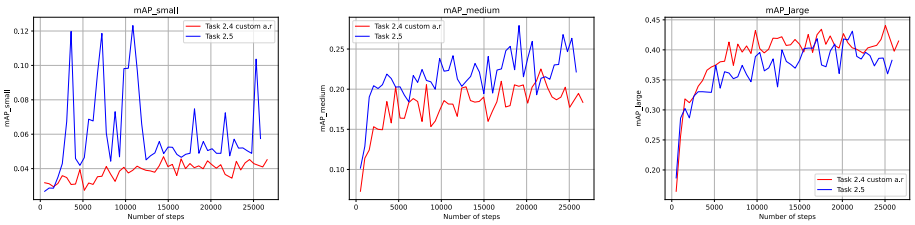


Figure 21: mAP comparison of the model from Task 2.3.4 Weight and Bias Initialization, with and without specialized anchor boxes, both trained on the larger/updated dataset.



(a) mAP small. (b) mAP medium. (c) mAP large

Figure 22: Mean average precision for the small, medium and large objects. Comparison of the model from Task 2.3.4 Weight and Bias Initialization, with and without specialized anchor boxes.

Task 2.5 - Extending the Dataset

The model from Task 2.3.4 Weight and Bias Initialization was trained on the extended dataset. The config file is `configs/task2_5.py`. The result is shown in figure 21.

These results shows the importance of large amounts of training data. The large amount of data makes the model able to see several angles, types and variations of the different classes in order to be able to classify the classes better. With the smaller dataset, there is a quite limited amount of instances of the classes, especially some of them like trucks, buses, motorcycles and

scooters. The model then only have a few images of these to train on, and they may all be the same vehicle in several video-frames, so the model won't be good at i.e. other truck-types. By having a larger and more diverse dataset, the classifier performs much better as it sees a lot more instances of the different classes.

Part 3 - Discussion and Evaluation

Task 3.2 - Discussion and Qualitative Analysis

We have chosen three modeling decisions from the project, where we have provided a deeper qualitative analysis. The three decisions that will be discussed are choice of data augmentation, the implementation of focal loss and the implementation of deeper output heads.

Data Augmentation

As described in Task 2.2 - Augmenting the Data, several data augmentations were tested. The tested CPU transforms were `RandomHorizontalFlip` and `RandomSampleCrop`, as suggested in the task. The GPU transforms that were tested were random `RandomContrast` and `RandomBrightness`. It was chosen to use all the augmentations mentioned above, even though it resulted in worse mean average precision compared to the basic model without augmentations. This is shown in figure 4. The mean average precision decreased significantly, from 4.47% to 2.96%.

As seen in figures 3a and 3b, the loss with these augmentations is higher than with the basic model from Task 2.1 - Creating your First Baseline. It is natural with a higher loss, as the augmentations results in a more diversified training set. A higher loss is in itself not a problem. The positive effect of this is that the more diversified training set makes the model less likely to overfit during training. This is particularly important when training on a small dataset, which was done before the extended dataset was utilized. However, it is not straight forward to include as much augmentation as possible to prevent overfitting. Too much data augmentation might destroy the training set by making it too different from the original data. Thus, it was important for us to find out if we had included too much data augmentation, given that the mean average precision decreased.

After selective testing, only `RandomHorizontalFlip` was able to improve mAP compared to Task 2.1 - Creating your First Baseline. We did, however, have a hypothesis that the data augmentation would work better on a more complex and pretrained backbone. At this point in the development process, the model was looking quite different from what the final model would look like. Thus, it was chosen to proceed with all the data augmentation, and do a test of the effect after the implementation of Task 2.3 - Implementing RetinaNet. Figure 23 shows the mAP of the RetinaNet, trained once with all data augmentations and trained once with only `RandomHorizontalFlip`. This clearly shows that the data augmentation improves the performance significantly.

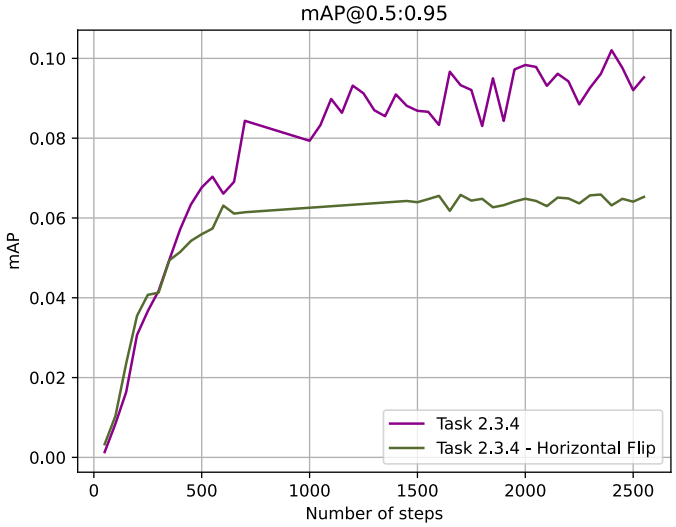


Figure 23: mAP@0.5:0.95 for the resulting RetinaNet after Task 2.3 - Implementing RetinaNet. RetinaNet is trained once with `RandomSampleCrop`, `RandomHorizontalFlip`, `RandomContrast`, `RandomBrightness` as data augmentation (Task 2.3.4), and trained once with only `RandomHorizontalFlip` as data augmentation (Task 2.3.4 - Horizontal Flip).

By this test, we do conclude that there wasn't introduced too much data augmentation, as the more diversified training set gave a much higher precision. It is possible that the model in Task 2.2 - Augmenting the Data would have achieved more or less the same precission as the model in Task 2.1 - Creating your First Baseline, if it was given more epochs of training. Given that the model wasn't pretrained when augmentation was introduced, the constantly changing training set might have made the training go slowly.

Focal Loss

Focal loss, described in Task 2.3.2 Focal Loss, is implemented to reduce the loss from well-classified samples and rather focus on the hard ones. This is done by reducing the weighting of a class, as it is predicted more and more accurate. The calculation of focal loss, shown in equation 1, does also have the feature to counter class imbalance. This is done to adjust α, such that classes with significantly more samples are given a relatively lower weighting in the calculation of the loss. In our case, this is done by assigning the background class a low α-value. To improve focal loss further, it could also be possible to discriminate between the other classes as well, and not just between foreground and background.

The property of focal loss to give well classified classes a lower impact,

is illustrated by figure 24. When the AP for the person class reaches ≈0.06, the slope slacks off quite instantly. This compared to the model from Task 2.3.1 Feature Pyramid Network, without focal loss implemented, which has a steady increase throughout the whole training process.

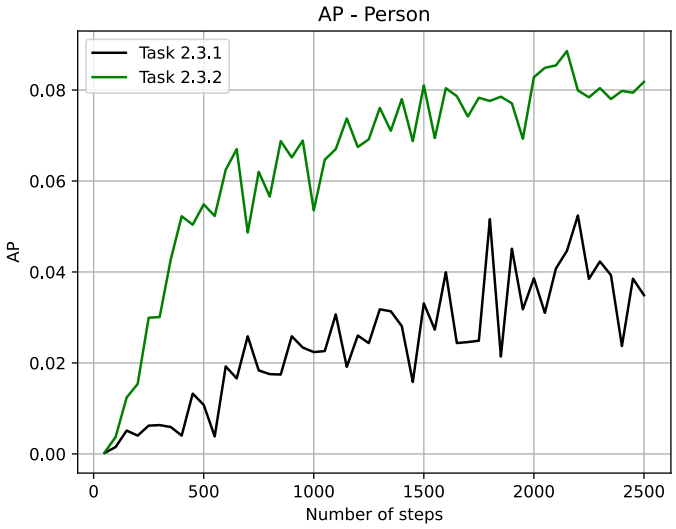


Figure 24: Image showing the average precision of person classifications for the model Task 2.3.1 Feature Pyramid Network and Task 2.3.2 Focal Loss.

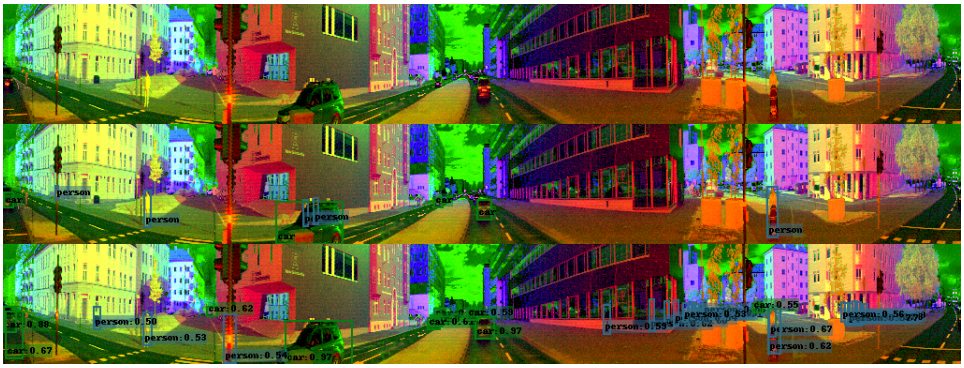


Figure 25: Image showing the detection of the model from Task 2.3.2 Focal Loss. The first image is the raw data. The second image shows ground truth labeling. The third image shows the detection and classification made by the model.

Deeper Convolutional Heads

Deeper convolutional heads were implemented on the output from the FPN in Task 2.3.3 Deeper Convolutional Regression and Classification Heads. There was developed one regression head and one classification head, which were used on all feature maps from the FPN-backbone.

The purpose of introducing deeper convolutional heads is to make the model able to pick up more complex patterns from the backbone. This is illustrated in figure 10, where the achieved mean average precision is much higher than with single layer heads. Another effect of the deeper heads is that training might be a bit slower in the beginning. This is due to the fact that there are more weights to learn. This is also seen in figure 10, as the model with single-layer output heads is better the first 200 iterations.

The performance of the model with deep convolutional heads is shown in figure 26. We chose to use the same image as in the analysis of focal loss, to illustrate the difference the deep convolutional heads make up. The model has clearly improved the ability to detect persons. The model from Task 2.3.2 Focal Loss detects persons most places there are vertical lines, like window frames or lamp posts. However, with the deep heads, the model is better suited to detect the complex shape of people.

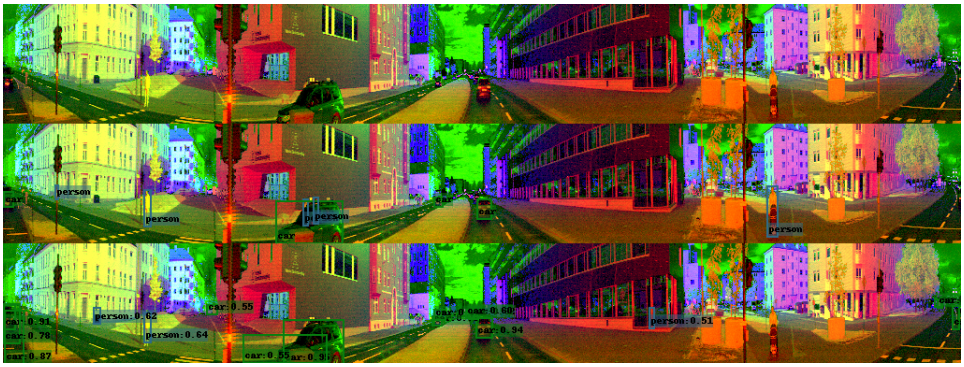


Figure 26: Image showing the detection of the model from Task 2.3.3 Deeper Convolutional Regression and Classification Heads. The first image is the raw data. The second image shows ground truth labeling. The third image shows the detection and classification made by the model.

Task 3.3 - Final Discussion

One of the greater weaknesses of the model is the inference speed, which has mainly decreased throughout the whole development process. When the RetinaNet is done, the inference speed is 1.76, showed in table 8. If the model is to be used on an autonomous vehicle, this is nowhere near good enough. Thus, it should have been taken more into consideration during the development process, than what we did. A good example is

the implementation of the backbone of the RetinaNet in Task 2.3.1 Feature Pyramid Network. Here, the final two layers were chosen to have respectively 1024 and 2048 output channels. However, we did test implementations with fewer layers, for example 128 and 64 output channels in the last two layers. This was disregarded due to a slightly worse mAP. Seen in retrospect, the better choice would probably be to use fewer output channels, as the layers with 1024 and 2048 output channels increases the number of parameters in the model drastically. Doing a change of output channels 1024 to 128 and 2048 to 64 would have reduced the number of parameters from 62 473 568 to 29 515 040 at that point in the development process. This parameter reduction should have been given a higher priority than a minimally better precision, to increase the inference speed.

Another weakness in the development process might be having to decide augmentation of data as one of the first stages. The choice of data augmentation was done in Task 2.2 - Augmenting the Data, a point in the development process where the model was looking very differently from the final model at the end. It was discussed in Task 3.2 - Discussion and Qualitative Analysis that the instant effect of the data augmentation was a much worse performing model. However, we showed in figure 23 that the final model performed significantly better with data augmentation. This shows that deciding what data augmentation to use, at a point in the process where the model is far from complete, might lead to a wrong decision. Thus, it might be argued that the data augmentation should have been done at a later stage in the development process.

Part 4 - Going Beyond

Task 4.1 - Implementing BiFPN

A bi-directional feature pyramid network was implemented as described in [3], with a few modifications to make it fit with our model. The implementation is done in `ssd/modeling/bacbones/bifpn.py`, and the config file is `configs/task4_1.py`. Note that the code in `bifpn.py` is inspired by an open github repository, which is linked in the top of `bifpn.py`.

The model is built on the final model from Task 2.3.4 Weight and Bias Initialization, but the backbone is changed. `BiFPN` replaced the previous `FPN` class from Task 2.3.1 Feature Pyramid Network, and is still implemented on top of parts of a pretrained ResNet34. However, there were required some modifications to the ResNet34 to make the feature sizes in the `BiFPNLayer` match. In the new implementation, the last two convolutional layers of ResNet34 are dropped, before it is added six new layers to produce the input to the BiFPN layers. These layers are called $P_{3,\dots,8}$. Note that [3] only introduces $P_{3,\dots,7}$, as the authors want to produce five different feature maps from the backbone. Thus, we introduced P_8 to get six feature maps to feed the convolutional heads. All feature maps are implemented to contain equally many channels. This is done because outputs are computed based on inputs from different layers, and the Resize() in equation 3 only adjusts for resolution differences. The BiFPN is implemented with three successive BiFPN-layers, as done in [3]. *Fast normalized fusion* is used as weighted feature fusion method, following the pattern shown for P_6 in equation 3.

$$P_6^{td} = \text{Conv}\left(\frac{w_1 \cdot P_6^{in} + w_2 \cdot \text{Resize}(P_7^{in})}{w_1 + w_2 + \epsilon}\right) \tag{3}$$

$$P_6^{out} = \text{Conv}\left(\frac{w_1' \cdot P_6^{in} + w_2' \cdot P_6^{td} + w_3' \cdot \text{Resize}(P_5^{out})}{w_1' + w_2' + w_3' + \epsilon}\right) \tag{4}$$

In addition, the weights in the BiFPN-layers are initialized using the Kaiming initialization. As suggested in [3], the convolutions in the BiFPN-layers are done using depthwise separable convolutions, meaning there is one depthwise convolution and one pointwise convolution, followed by a batch normalization and ReLU-activation.

In [3], the authors observe an improved average precision with fewer parameters, when replacing FPN with BiFPN. Note that, in the paper, this is done using a different backbone that the FPN and BiFPN are implemented on top of, than the backbone we have implemented BiFPN on top of. Despite of this, we are able to observe some similar results. The statistics of the new model are reported in table 12, showing a significant decrease in parameters from Task 2.3.4 Weight and Bias Initialization.

Table 12: Statistics for the trained network from Task 4.1 - Implementing BiFPN.

mAP@0.5:0.95 (final)	0.0836
mAP@0.5:0.95 (peak)	0.0870
FPS	3.3927
Parameters	5 382 734

However, we do not observe an improved mAP, as the authors of [3] did. To compare the difference in performance between FPN and BiFPN, the RetinaNet is trained both with our FPN and BiFPN as backbone, with the mean average precision plotted in figure 27. We see that the model with BiFPN in the backbone needs more iterations before it achieves results of significance. This is probably due to the changes in the ResNet34 that BiFPN is built upon. In the new model, several of the pretrained layers are removed, and replaced with six new layers to produce $P_{3,\ldots,8}$. This shift away from the pretrained layers is probably the reason why the model with BiFPN is training so slowly for the first 500 iterations. Thus, we don't believe that the shift from FPN to BiFPN itself is the reason for worse performance, but that the changes in ResNet34 is to blame. This hypothesis could have been tested further by keeping more layers of the pretrained ResNet34 and adjusting them to the input we wanted for the BiFPN layers. However, due to time limitations of the project, this is not implemented.

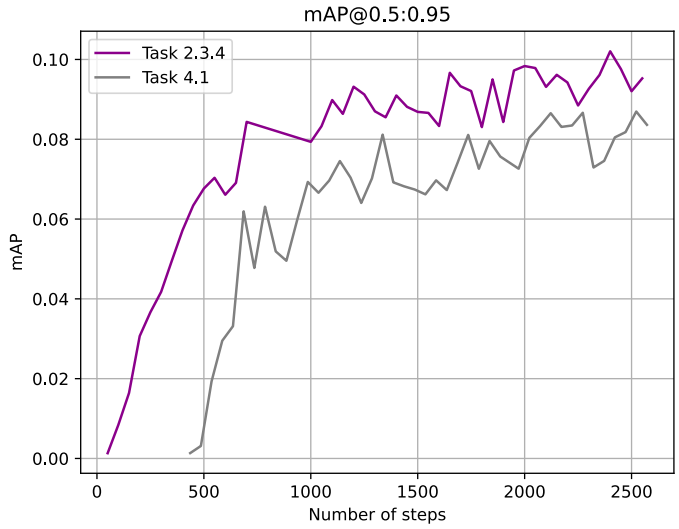
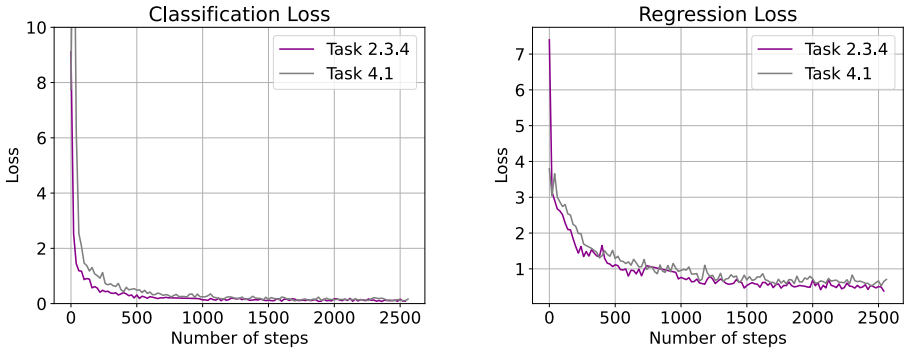


Figure 27: mAP for the model from Task 2.3.4 Weight and Bias Initialization with FPN in the backbone, compared to the model from Task 4.1 - Implementing BiFPN where there is used BiFPN in the backbone.



(a) Classification loss.

(b) Regression loss.

Figure 28: Losses for the model from Task 2.3.4 Weight and Bias Initialization with FPN in the backbone, compared to the model from Task 4.1 - Implementing BiFPN where there is used BiFPN in the backbone.

Task 4.3 - Following your own idea

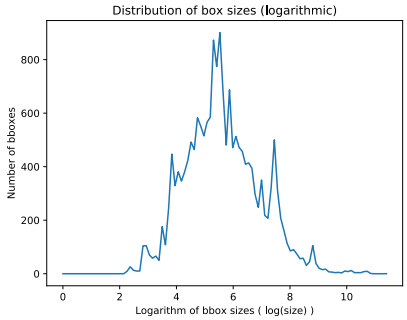
The task is quite open, and due to the lack of improvements from Tuning aspect ratios and sizes, where we tried to change the aspect ratios and min-sizes, we wanted to see if this tuning could be implemented as an automated part when initializing the network. The goal was to initialize the anchor boxes at the beginning by doing an automated analysis of the annotation data in order to find the dominating box sizes and the aspect ratios of the dataset. The backbone still had 6 layers, so the automated anchors had to be distributed over 6 feature maps, but the number of aspect ratios per feature map is customizable. This made the configuration of the anchor-boxes as simple as:

```
anchors = L(AnchorBoxesCustom)(
   image_shape="${train.imshape}",
   aspect_ratios_per_size = 5,
   scale_center_variance=0.1,
   scale_size_variance=0.2,
   annotation_path='data/tdt4265_2022/train_annotations.json'
)
```

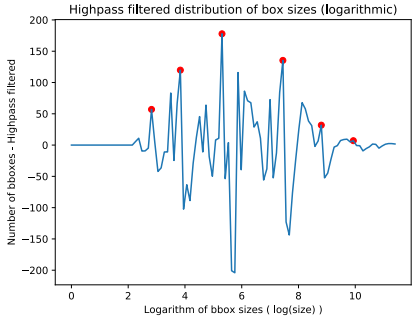
as shown in the config file `configs/task4_3.py`.

The method is based on a peak-finder for finding ideal sizes, and then k-means in order to find the ideal aspect ratios for each size level.

A logspace from 0 to the maximum ground truth bounding box size is created. Then the sizes of all the bounding boxes are put into bins made from the logspace. This leads to the graph in figure 29a.



(a) Distribution of bounding box sizes in a logarithmic scale.



(b) Distribution of bounding box sizes in a logarithmic scale after highpass filtering.

Figure 29: Distribution of bounding box sizes.

From figure 29a one can see there are some peeks along the general *wide* peak that covers the entire distribution. In order to isolate the peaks, a highpass filter was applied using `signal.butter()` from `scipy`. Then, the 6 highest peaks (because of 6 feature maps) with sufficient distance between them (9 logspace bins) are found, and they can be seen in figure 29b.

By taking the exponent of the logarithmic bounding box sizes of the peaks, the most dominant sizes are found. The ground truth bounding boxes are then divided into the closest 6 groups based on their size being closes to the one of the 6 *peak sizes*. For each of these, `KMeans` from `sklearn` is used to group the aspect ratios into `aspect_ratios_per_size` number of *cluster centers*.

The result of this algorithm is a method that takes annotations and based on this outputs the 6 different sizes (similar to the 6 feature maps of *normal* anchors) and the `aspect_ratios_per_size` best fitting aspect ratios for each of the size levels. In order to make this automatic creation of anchors work, a special anchor box class was also implemented: `AnchorBoxesCustom` in `ssd/modeling/anchor_boxes_custom.py`. This creates bounding specialized for each of the sizes and aspect ratios. In addition to the found aspect ratios, two boxes are also created with aspect ratio $1:1$. One of them has the ideal size found and the other one has twice the size. This is to distribute the anchor boxes over several size-levels than just the 6 found earlier. In order to analyze how the automated method places the anchor boxes, one can look at a plot of the aspect ratios versus box sizes, as shown earlier, but for the anchors created now. This figure is shown in figure 30, and one can see that the placed anchor does in fact fit well with the data itself.

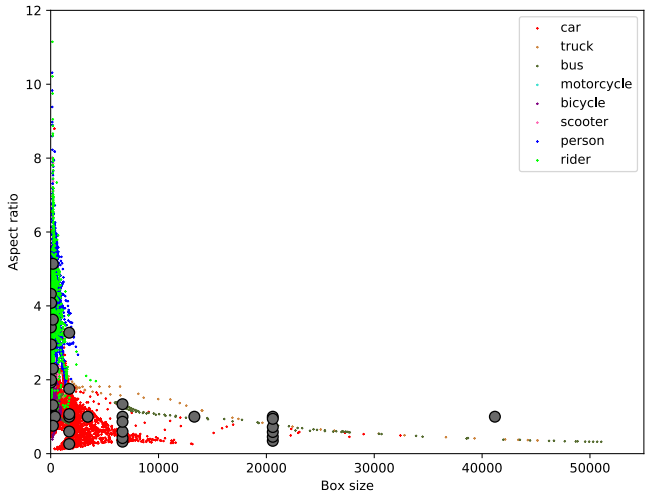


Figure 30: Distribution of anchor boxes with the automated placement.

The problem with this algorithm is that the feature sizes no longer match the output of the downsampling done while passing an image through the feature pyramid network that is the backbone of the model. This mismatch is however solved with a interpolation in the `forward`-function of the feature pyramid network backbone. In order to not affect the old models, this is implemented in `FPN2` in `ssd/modeling/backbone/fpn2.py`.

The results of training this model on the original dataset for 50 epochs is a final mAP of 0.0215 (peak mAP: 0.0238), and a comparison with the model from Task 2.3.4 Weight and Bias Initialization is shown below in figure 31. The performance is a lot worse, and it struggles to detect objects. This shows that the automated method may not work as good as humans doing a solid analysis of the dataset. In addition, the introduced interpolation steps in the network might affect the performance as well.

The performance, in terms of mAP and FPS, in addition to the number of parameters for this model i shown in table 13

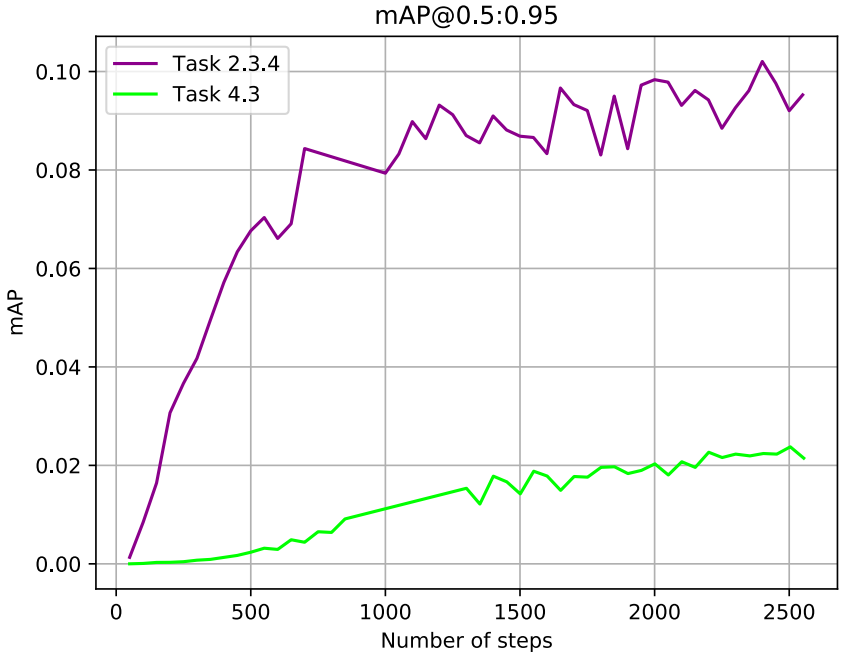


Figure 31: Comparison of the model from Task 2.3.4 Weight and Bias Initialization with and without automated anchor box configuration.

Table 13: Statistics for the trained network from Task 4.3.

mAP@0.5:0.95 (final)	0.0215
mAP@0.5:0.95 (peak)	0.0238
FPS	1.2182
Parameters	66 635 303

Task 4.4 - Comparing to State-of-the-Art

When comparing with a state-of-the-art detector, we chose to compare our model to a detector with many different features. This was done to see how a different architecture could improve the performance. We compared our model with the Faster R-CNN (X101-FPN, 3x) [4]. The Faster R-CNN code from Detectron2 was used to transfer learn the TDT4265 dataset, and this was done following the official Colab Tutoral from Detectron2 [5]. The notebook to reproduce the results is given in `notebooks/task4_4.ipynb`, and the predictions on one of the images in the validation set is shown in figure 32.

(a) Without colors.

(b) With colors.

Figure 32: Prediction from Faster R-CNN (X101-FPN, 3x) on one of the images in the validation set.

The architecture of the network is showed in figure 33. Faster R-CNN is built using a region proposal network (RPN) on top of the CNN. This backbone enables the possibility to propose areas in the image where the object most likely is located. The feature maps and the proposed regions are merged using region of interest (RoI) pooling. The purpose of this is to covert all the proposals to fixed shapes, and make predictions.

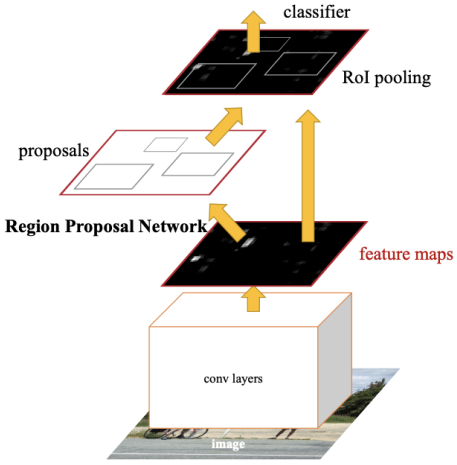


Figure 33: Architecture of Faster R-CNN (X101-FPN, 3x) implemented in [5].

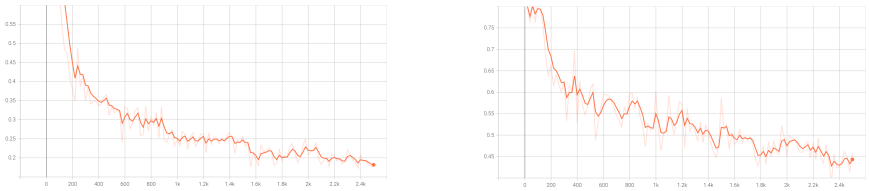
The Faster R-CNN (X101-FPN, 3x) was trained in the TDT4265 dataset, and tested on the validation set. The performance is summarized in figure 34. The training was done with 2500 iterations and a batch size of 32, to make the results comparable with our own results. The final mean average precision is 13.3%, which is considerably better than the best result we

have achieved of 9.53%. It is worth noticing that this is achieved using ≈105 million parameters, compared with our model using ≈66 million parameters. However, there is an implementation detail regarding the number of parameters that could have helped improve our own model. The Faster R-CNN (X101-FPN, 3x) uses a ResNeXt101 in the backbone, compared with our ResNet34. The ResNeXt introduces *cardinality*, meaning that the convolutional layers are split in several parallel paths. Cardinality is a method to reduce the number of parameters in the network. Thus, using ResNeXt in our own model could have allowed a higher inference speed, which is an issue with our model discussed in Task 3.3 - Final Discussion.

```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.133
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.290
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.099
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.142
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = -1.000
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = -1.000
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.066
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.165
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.178
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.178
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = -1.000
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = -1.000
```

Figure 34: The performance of Faster R-CNN (X101-FPN, 3x) from [5].

Figure 35 shows the classification and regression losses of the model. Compared to figure 11, the losses from Faster R-CNN doesn't seem to be as stagnated as with the previous models. We only trained for 2500 iterations since it's the same as all the other models we have trained on the small dataset, but the Faster R-CNN model could probably have gotten even better performance if left to train even longer.



(a) Classification loss. (b) Regression loss.

Figure 35: Losses for the Faster R-CNN (X101-FPN, 3x).

References

[1] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 2999–3007, 2017.

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1026–1034, 2015.

[3] Mingxing Tan, Ruoming Pang, Quoc V. Le, and Jian Sun. Efficientdet: Scalable and efficient object detection. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, page 10781–10790, 2020.

[4] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1137 – 1149, 2015.

[5] Yuxin Wu, Alexander Kirillov, Francisco Massa, Wan-Yen Lo, and Ross Girshick. Detectron2. `https://github.com/facebookresearch/detectron2`, 2019.