TDT4265 - Computer Vision and Deep Learning

Assignment 4

Martin Johannes Nilsen

22nd April 2022

1 Object Detection Metrics

Task 1a

When talking about precision metrics, Intersection over Union can be useful to mention. Especially when we want to mesure how well a neural network is able to separate objects by using bounding boxes. In intersection over union we measure how much of the predicted bounding box overlaps with the ground truth, being the real bounding box. For illustrating this even further, I have drawn to boxes in a figure below. As we can see in the illustration, we have one red box behind a blue box. The intersection between them is illustrated by the color purple. To measure the prediction of the bounding box, we take the intersection of the two boxes (purple area), divided by the union of the two boxes (red and blue).

$$IoU = \frac{\text{area of overlap}}{\text{area of union}}$$

Figure 1: Intersection over Union

Task 1b

In this subsection we will introduce the two performance metrics **Precision** and **Recall**. The metrics are built up by different shares of *True* and *False Positives* and *Negatives*, and the defintions of these should be known before we introduce the equations. A **True Positive (TP)** is that we have predicted the result as a positive, agreeing with the ground truth and being a correct prediction. A **True Negative (TN)** is that we have correctly predicted the result to be negative. A **False Positive (FP)** is an incorrect predition, where we have predicted it to be positive, but the truth is that it is negative. And last, but not least, we have a **False Negative (FN)** which is that we have incorrectly predicted the result to be negative, when it actually is positive. Then we are ready for the formulas.

Precision How many of the positive predictions we have made are correct?

$$Precision = \frac{TP}{TP + FP}$$

Recall How well are we able to find all the positives there is?

$$Recall = \frac{TP}{TP + FN}$$

Task 1c

We are tasked to find the mean average precision between two precision-recall curves. I start off by plotting each of the given curves and calculate the Average Precision. Then we can calculcate the Mean Average Precision (mAP).

The average precision is calculated using the following equation

$$AP = \frac{1}{\text{number of points}} \sum_{r \in \{0.0, \ldots, 1.0\}} Precision$$

One thing to mention here is that the number of points is 11, being the elements

$$\{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$$

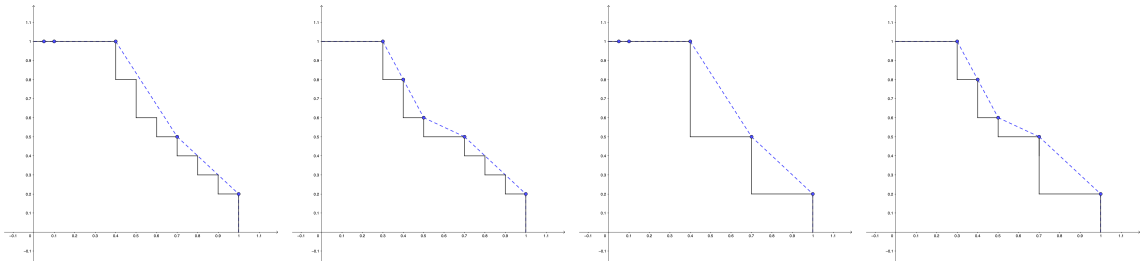
For calculating the average precision we sum the precision at each of these points and divide by the total number of points for averaging.

The mean average precision can then be calculated with the following equation

$$mAP = \frac{1}{\text{number of APs}} \frac{\text{sum of APs}}{\text{number of points in AP}}$$

For this task, I wanted to test out another aspect as well. Earlier, when calculating average precision, we have used interpolation. Therefore, I want to calculate the mAP with both an interpolated and non-interpolated curve in this task, for comparing the different results.

The task continue on the next page.



Class 1 | Class 2 | Class 1 interpolated | Class 2 interpolated

Figure 2: Plots of classes, both original and interpolated

Original

Class 1 yields the following average precision

$$AP = \frac{1}{11}(5 * 1.0 + 0.8 + 0.6 + 0.5 + 0.4 + 0.3 + 0.2) = \frac{7.8}{11} = 0.709$$

Class 2 yields the following average precision

$$AP = \frac{1}{11}(4 * 1.0 + 0.8 + 0.6 + 2 * 0.5 + 0.4 + 0.3 + 0.2) = \frac{7.3}{11} = 0.664$$

Which altogether gives a mean average precision of

$$mAP = \frac{1}{2}\frac{7.8 + 7.3}{11} = 0.686$$

Interpolated

Class 1 yields the following average precision

$$AP = \frac{1}{11}(5 * 1.0 + 3 * 0.5 + 3 * 0.2) = \frac{7.1}{11} = 0.645$$

Class 2 yields the following average precision

$$AP = \frac{1}{11}(4 * 1.0 + 0.8 + 0.6 + 2 * 0.5 + 3 * 0.2) = \frac{7}{11} = 0.636$$

Which altogether gives a mean average precision of

$$mAP = \frac{1}{2}\frac{7.1 + 7}{11} = 0.641$$

Which shows that the original and interpolated mAP show some variations, though not too much. One may still prefer the original one, but one thought would be to have accurate measures at each point, which could have made it even more precise.

2 Implementing Mean Average Precision

Task 2f

Below you can see plot of the precision-recall curve. As requested, the final mAP over the dataset is 0.9066.

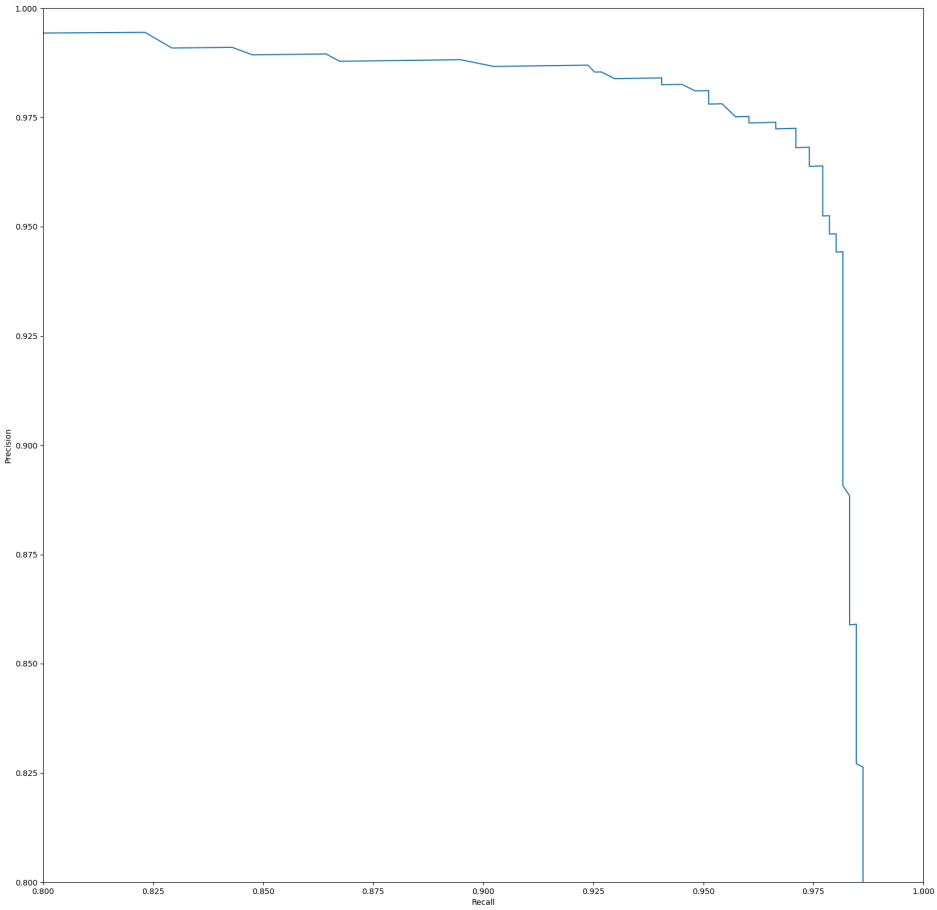


Figure 3: Precision-recall curve

3 Theory

Task 3a

The operation used to filter out the redundant overlapping boxes for an object is called non-max supression.

Task 3b

The resolution gets lower and lower as we move deeper into the network. As the lower resolution feature maps is used to detect larger objects, we have that the shallower layers to the left are used for detecting smaller objects. Therefore, the claim that the deeper layers (closer to the output) detect small objects is **False**.

Task 3c

One reason for the SSD to use different aspect ratio bounding boxes at the same location is to be able to find boundary box predictions for multiple types of objects. If you were to only find one particluar type of object, having only one aspect ratio on the bounding box would work fine if all the objects of that type would fit in the set shape. With the predicted bounding boxes covering multiple shapes, the model is better equipped for detecting multiple objects as well.

Another reason is that this optimizes the chances of finding the best fitting bounding box. A score is calculated for each of the aspect ratios, which can be further compared for seeing how much of the object is within each of them. This tells us more about the most fitting placement of the optimal bounding box.

Task 3d

The main difference between SSD and YOLOv1/v2 is the use of single- vs multi-scale feature maps. SSD uses multi-scale feature maps, but YOLOv1/v2 on the other hand uses only a single-scale feature map. One thing to mention is the performance on predictions of smaller objects. The implementation of multi-scale feature maps in SSD may limit its ability to detect small objects if it does not fit in a feature map. YOLOv1/v2 does not have this problem as they use a single feature map.

Task 3e

For calculating the total amount of anchor boxes for a feature map, we take the resolution of the feature map times the number of different anchors. This can be

written as the following equation

$$\text{Number of achor boxes} = H \cdot W \cdot k,$$

where H is the height of the feature map, W the width and k the number of anchors.

For the given feature map of resolution 38×38, with 6 anchors, we get

$$38 \cdot 38 \cdot 6 = 8664$$

anchor boxes.

Task 3f

For this subtask we extend the equation from task 3e, summing up the number of anchor boxes of each feature map. This yields the following equation:

$$38 \cdot 38 \cdot 6 + 19 \cdot 19 \cdot 6 + 10 \cdot 10 \cdot 6 + 5 \cdot 5 \cdot 6 + 3 \cdot 3 \cdot 6 + 1 \cdot 1 \cdot 6 = 11640$$

anchor boxes.

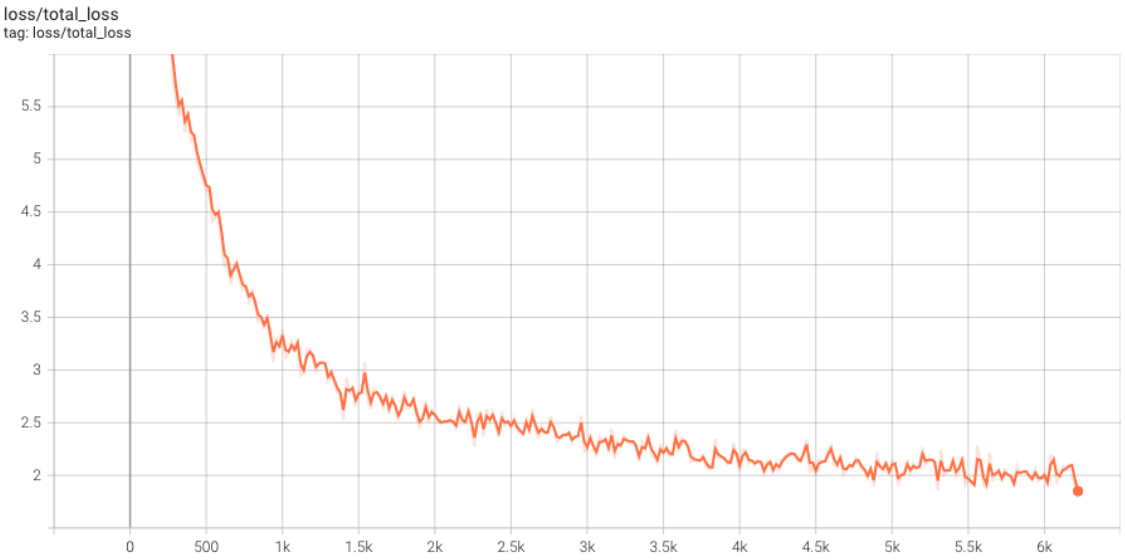
4 Implementing Single Shot Detector

Task 4b

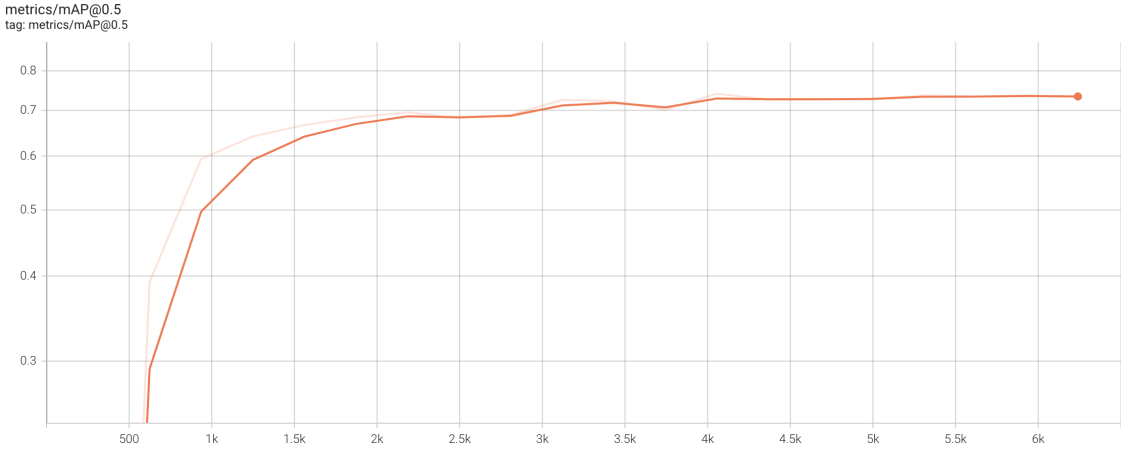
As we have 10 000 images in the dataset, one can calculate the iterations by taking the size of the dataset, divided by batch size. For each epoch you iterate over all the batches. This means that with batch size 32, yielding 312 batches, the model need to converge at 75-77% during 20 epochs.

$$312 \cdot 20 = 6240$$

My implementation reached a mean average precision of 74% during 6 000 iterations. The two requested graphs, respectively for the total loss and mean average precision with an IoU-threshold of 0.5 (mAP@0.5), is shown below.



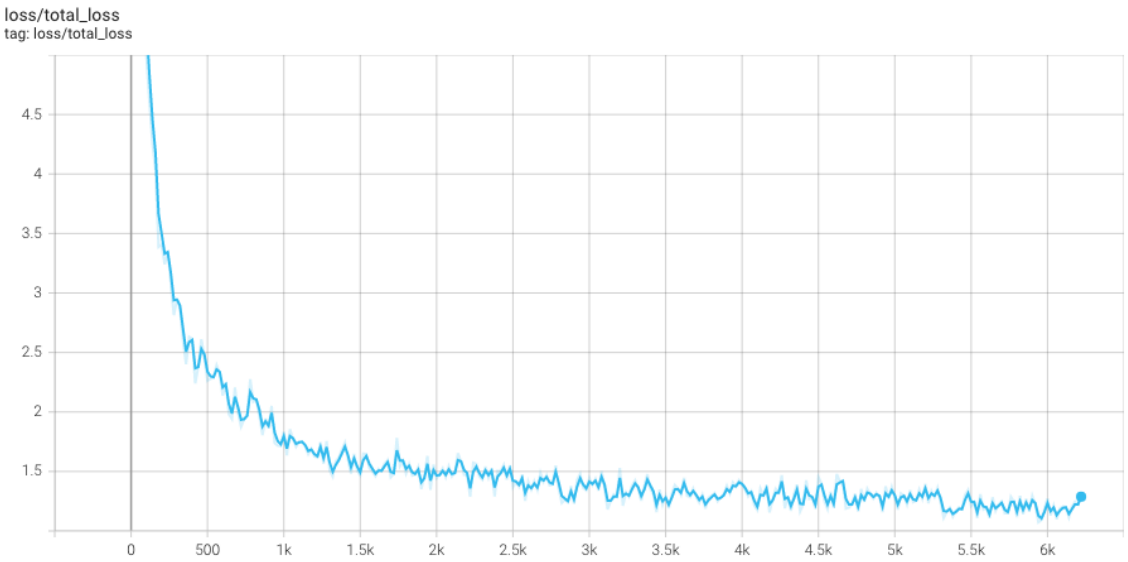
Total loss during training



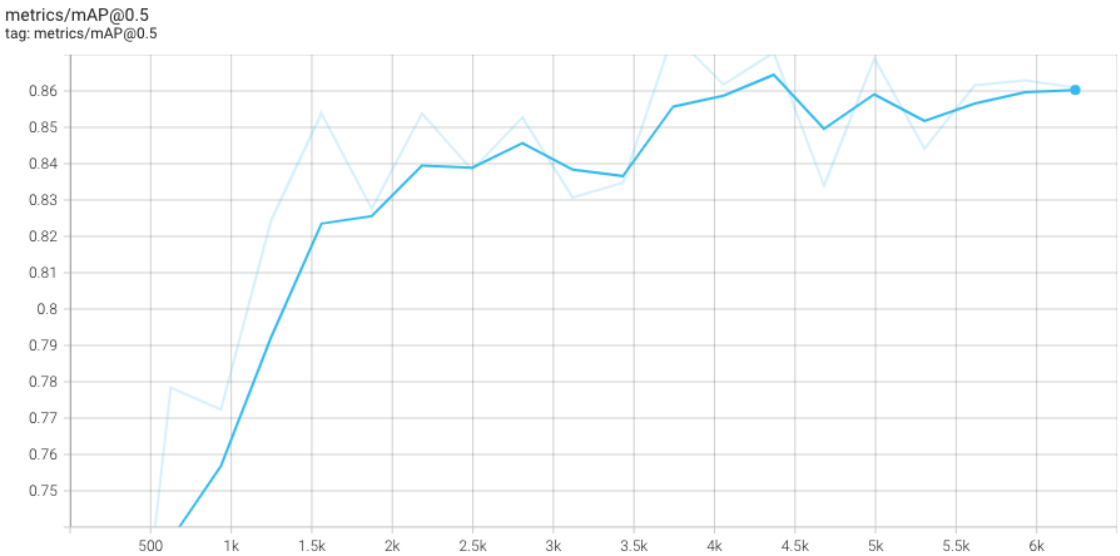
mAP@0.5 during training

Task 4c

Using the same logic from the last subtask, reaching a mean average precision before 10 000 iterations would mean a maximum of 32 epochs using a batch size of 32. I started off changing the learning rate to 0.0026 and changing the optimizer to Adam. This yielded some effect. In addition I changed the mean and std to the values recommended by PyTorch. When running this I noticed that a problem was detecting smaller items, and I therefore halved the two smallest anchor boxes. With these implementations I managed to get to 85-87% already after 20 epochs, with only small changes to the layers. I made multiple attempts at trying to add batch normalization and dropout to the model, without significant effect. This only lead to the model having a harder time converging, and as I had already gotten 85% I did not experiment any further with this. The total loss and mAP@0.5 are shown below, and the final layer structure of the model are shown on the next page.



Total loss during training



mAP@0.5 during training

Important parameters in the config

Learning rate	0.0026
Batch size	32
Epochs	32
Mean	[0.485, 0.456, 0.406]
Std	[0.229, 0.224, 0.225]
Out channels	[128, 256, 128, 128, 64, 64]
Aspect ratios	[15,15],[30,30],[111,111],[162,162],[213,213],[264,264],[315,315]
Weight decay	0.0005

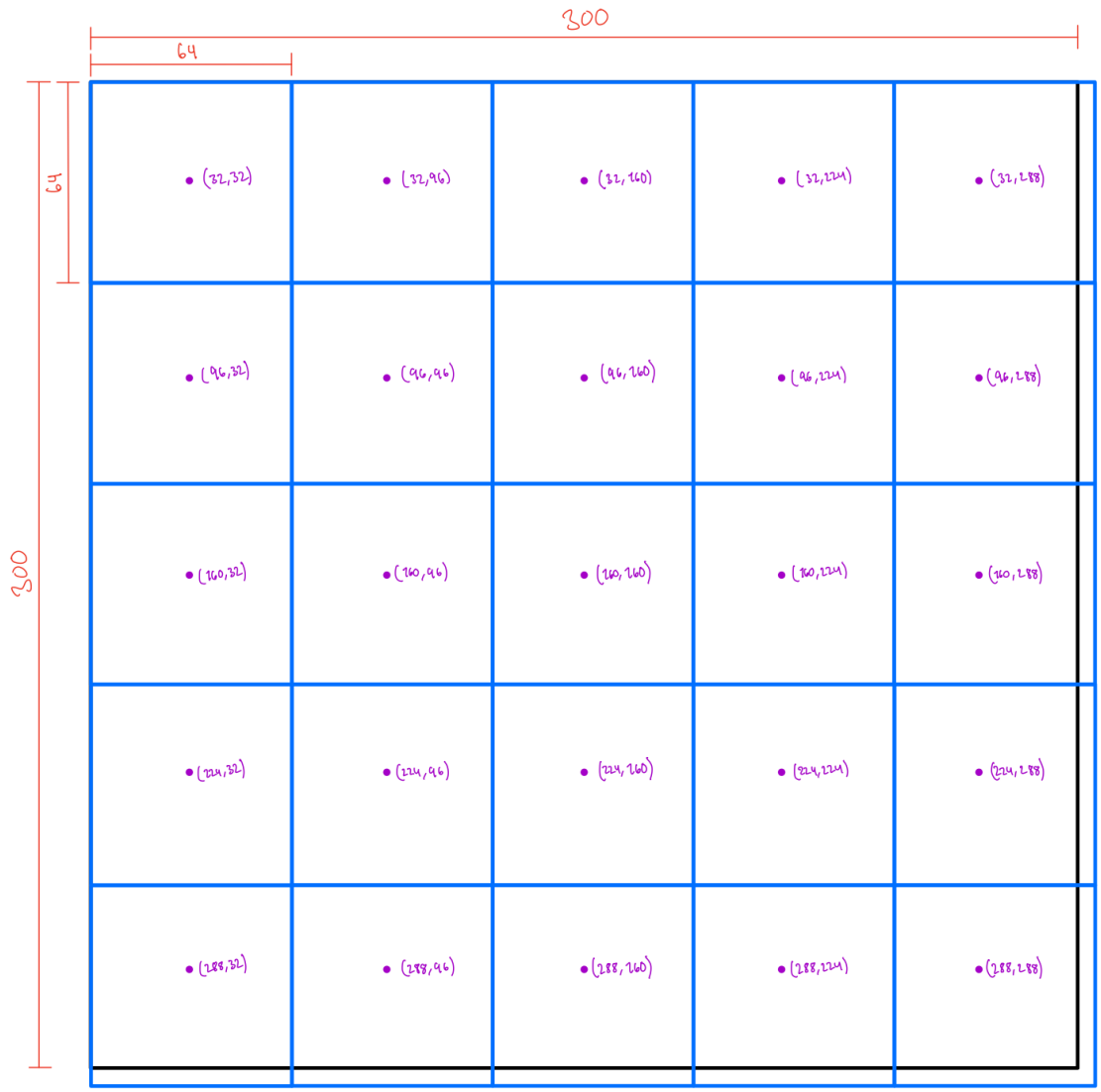
Model structure

Is Output	Layer Type	Number of Filters	Stride
	Conv2D	32	1
	MaxPool2D	–	2
	ReLU	–	–
	Conv2D	64	1
	MaxPool2D	–	2
	ReLU	–	–
	Conv2D	64	1
	ReLU	–	–
Yes - Resolution: 38×38	Conv2D	output_channels[0]	2
	ReLU	–	–
	Conv2D	256	1
	ReLU	–	–
Yes - Resolution: 19×19	Conv2D	output_channels[1]	2
	ReLU	–	–
	Conv2D	256	1
	ReLU	–	–
Yes - Resolution: 10×10	Conv2D	output_channels[2]	2
	ReLU	–	–
	Conv2D	256	1
	ReLU	–	–
Yes - Resolution: 5×5	Conv2D	output_channels[3]	2
	ReLU	–	–
	Conv2D	256	1
	ReLU	–	–
Yes - Resolution: 3×3	Conv2D	output_channels[4]	2
	ReLU	–	–
	Conv2D	256	1
	ReLU	–	–
Yes - Resolution: 1×1	Conv2D	output_channels[5]	2

Task 4d

In this task, we are to show the placement of anchor box centers, and calculate the size of the anchor boxes used for the feature map with 5x5 resolution, and 64x64 stride.

I started off by calculating the centers of each anchor box. As the given feature map resolution is 5x5, we divide the 300x300px image into a grid of 5x5 tiles of size 64x64 because of the nature of striding. Further on we know that the centre point is in centre of the tile and get that the first centre point is (32, 32). Because of the 64x64 stride, we have that the next centre is 64px to the right at (32, 96). This applies for both x and y direction. As you can see, even though the tiles go past the border of the image, the anchor centres is all located within the image frame, but a little shifted to the right border. On the next page we calculate the anchor box sizes and visualize all anchor boxes.



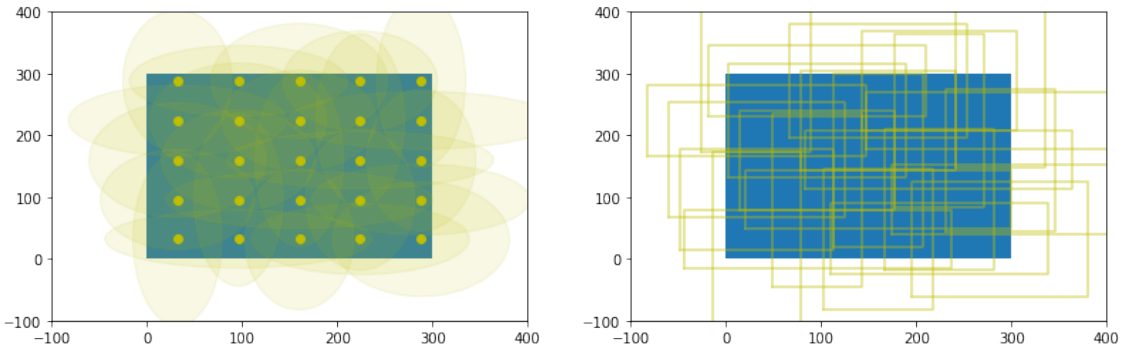
We continue with the task, where we are asked to find the 6 anchor box sizes used at each anchor box centre. For calculating these, we are given 4 equations, where two of them are applied twice. These are:

- [min_size, min_size]
- $[\sqrt{min\_size \cdot next\_min\_size}, \sqrt{min\_size \cdot next\_min\_size}]$
- $[min\_size \cdot \sqrt{\text{aspect\_ratio1}}, min\_size \,/\, \sqrt{\text{aspect\_ratio1}}]$
- $[min\_size \,/\, \sqrt{\text{aspect\_ratio1}}, min\_size \cdot \sqrt{\text{aspect\_ratio1}}]$
- $[min\_size \cdot \sqrt{\text{aspect\_ratio2}}, min\_size \,/\, \sqrt{\text{aspect\_ratio2}}]$
- $[min\_size \,/\, \sqrt{\text{aspect\_ratio2}}, min\_size \cdot \sqrt{\text{aspect\_ratio2}}]$

We solve them with aspect ratio $[2, 3]$, $min\_size = 162$ and $next\_min\_size = 213$:

- $[162, 162]$
- $[\sqrt{162 \cdot 213}, \sqrt{162 \cdot 213}] = [186, 186]$
- $[162 \cdot \sqrt{(2)}, 162/\sqrt{(2)}] = [229, 115]$
- $[162/\sqrt{(2)}, 162 \cdot \sqrt{(2)}] = [115, 229]$
- $[162 \cdot \sqrt{(3)}, 162/\sqrt{(3)}] = [281, 94]$
- $[162/\sqrt{(3)}, 162 \cdot \sqrt{(3)}] = [94, 281]$

This yields us 6 boxes per location, which can be confirmed in the notebook `visualize_priors.ipynb` as it it yields that there are to be 150 $(25 \cdot 6)$ anchor boxes with 6 boxes per location.



Task 4e

After running the command

```
python demo.py configs/task4e.py demo/mnist demo/mnist_output
```

I got the following labeled images

15 output images, labeled by a model with mAP@0.5 of 0.8734 on validation set

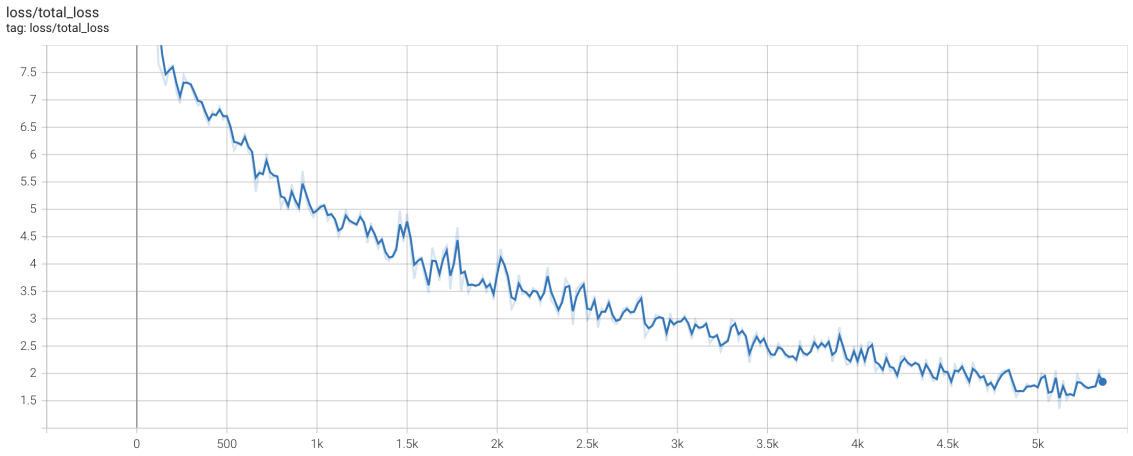
As you may observe from the images above, it seems like the number it have missed the most is the number 1. In addition to this, it seems like some numbers are not recognized if they are located too close to other numbers. Nevertheless, I would say this perform pretty good, and does confirm that the validation accuracy of 87.34% is quite close to the performance in practice.

Task 4f

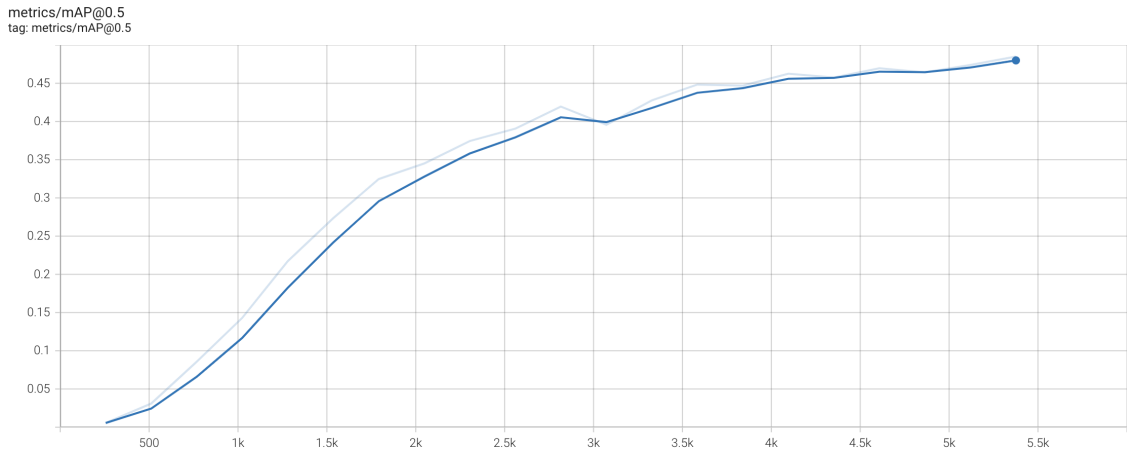
In this task we were asked to train the VGG model provided in the handout code, on the PASCAL VOC dataset. This was done by running the command

```
python train.py configs/voc_vgg.py
```

As there was 256 batches, I ran the training for 20 epochs for a total of 5120 iterations. After training for a little more than 5K iterations, the final validation mAP@0.5 was at 48.49%. The curves for total loss and mAP@0.5 are plotted below.



Total loss during training

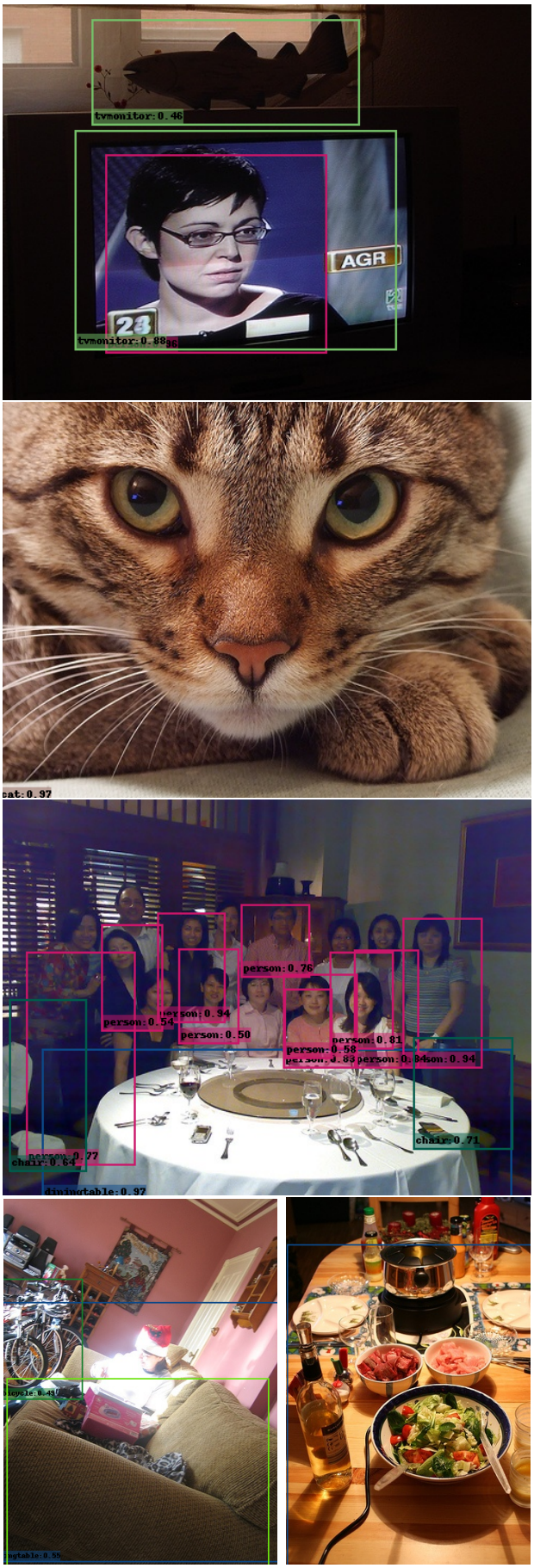


mAP@0.5 during training

Further on, we were to test the model on a collection of 5 test-images. This was done by running the following command

```
python demo.py configs/vog_vgg.py demo/voc demo/voc_output
```

which yielded 5 labeled images. These can be found on the next page. The model seems to be getting a lot of them correctly, almost better on the test-images than the precision of 50% would suggest. Though, It should be mentioned that some objects are labeled completely wrong, overall a decent performance.


tvmonitor:0.46
AGR
23
tvmonitor:0.88
cat:0.97
person:0.76
person:0.54
person:0.94
person:0.50
person:0.81
person:0.58
person:0.84
person:0.94
chair:0.71
person:0.77
chair:0.64
diningtable:0.97