Report - Exercise 3

Yifei Liu, Klaus-Rudolf Kladny

March 2021

Problem 1. Building a 2 Stage 3D Object Detector

1.

For the implementation of the Intersection over Union (IoU) we used the library `shapely` which provides simple to use abstractions for polygons and can calculate the intersection and union of these. As this task requires computing the 3D IoU, this functionality was not sufficient. Hence, for calculating the 3D intersection volume V_{inter}, the intersection area A_{inter} of the polygons that make up the ground rectangles of the bounding boxes was determined and then multiplied by the height intersection value:

$$V_{inter} = A_{inter} \cdot max(0, min(h_1, h_2) - max(l_1, l_2)),$$

where h_1 and h_2 are the height coordinate values of the upper points of box 1 and box 2 and l_1 and l_2 are the lower height coordinate values of the boxes, respectively. Then, the 3D IoU could be computed as

$$IoU = \frac{V_{inter}}{V_1 + V_2 - V_{inter}},$$

where V_1 and V_2 are the volumes of box 1 and box 2, respectively. For improved readability of the code, we created an additional function `compute_single_iou()`, which computes the IoU for two individual bounding boxes.

In this manner, the average recall on the *val* data set could be computed:

$$\text{average recall} \approx 0.813$$

The code for this calculation can be found in `avg_recall.py` and contains a simple calculation of an arithmetic mean.

To see why the average recall is more appropriate for assessing the first stage proposals, let us compare them:

Precision:	Recall:
$\frac{\text{TP}}{\text{TP} + \text{FP}}$	$\frac{\text{TP}}{\text{TP} + \text{FN}}$

We see that while precision penalizes False Positives (FP), recall penalizes False Negatives (FN). The reason for using recall is apparent: As the coarse predictions by the first stage network will be refined by the second stage network, False Positives can still be sorted out by NMS and classification. For first stage False Negatives on the other hand there is no opportunity for correcting them in the second stage. In other words: If an actual object is not detected in the first stage already, it cannot be detected in the second stage anymore because only the predictions from the first stage will be used as input to the second stage.

2.

For this task, gaining speed was the hitch. We tried to use the predefined `numpy` vector operations as often as possible in order to circumvene for-loops. To this end, for every prediciton (which was the only required for-loop), we performed a rigid transformation on the entire point cloud, which can be formulated as a matrix multiplication. We divide this into a translation followed by a rotation. To further speed up this transformation, we utilized `numpy`'s `order` argument for `ndarrays`, as it is more efficient to multiply a matrix in row major format by a matrix in column major format than for other formats/combinations of formats.

Then, the scanning of the point cloud was much simpler as we could easily check for each coordinate of a transformed point if it was in the box like:

```
input  : Point p, Box b
if $abs(p.x) \leq \frac{b.width}{2} \wedge abs(p.y) \leq \frac{b.height}{2} \wedge abs(p.z) \leq \frac{b.length}{2}$ then
    return True;
else
    return False;
end
```

Algorithm 1: Find out if a point is contained in a box

In practice, of course, this was implemented in a vectorized fashion and not individually for each point.

One can see in Figure 1 that this approach indeed computes the RoIs in the desired manner. The images were created using the script `exercise3/visualize_task2.py`.

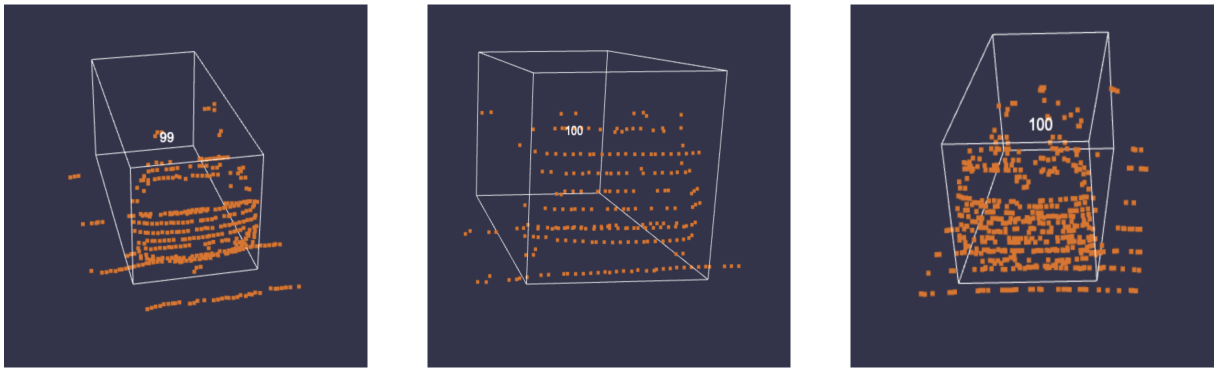


Figure 1: Pooled RoIs from three randomly selected frames. Frame 20, index 0 (bounding box parameters: `[ 5.14 1.45 19.2 1.56 1.65 3.69 -1.38]`) (Left), frame 2, index 0 (bounding box parameters: `[ 1.15 1.68 24.32 1.61 1.66 3.2 -1.52]`) (Middle), frame 100, index 0 (bounding box parameters: `[ 0.12 1.66 10.93 1.55 1.63 3.32 -1.57]`) (Right). These RoIs were generated from ground truth bounding boxes where the (default) values `delta = 1.0` and `max_points = 512` were used. Note that the visualized boxes are not enlarged by `delta`, but the visualized points are the points contained in the enlarged boxes.

3.

An example of such a *preprocessed* scene can be observed in Figure 2. It was created using the script `exercise3/visualize_task3.py`.

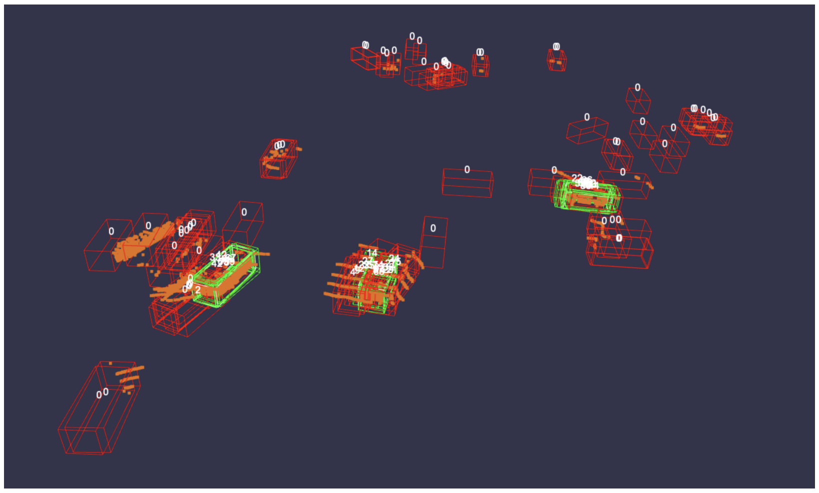


Figure 2: Final input-output pairs per scene for frame 20. The boxes are either ground truth boxes (green) or proposals (red) without any modification. Again, like in the visualizations for task 1.2, the visible points are from the pooled ROIs (but this time of the proposals) with `delta = 1.0` and `max_points = 512`.

Now to the explanation for why this sampling scheme is useful:

(1) If we were to just pick samples randomly (uniformly distributed), we would in most cases take much more background proposals as foreground proposals. This is due to the fact that we do not penalize the first-stage network for false positives (recall task 1.1).

But, for training the regression head, we also need many foreground objects as most background objects will not contain an object with high likelihood to begin with. In fact, as described in task 1.4 (a), we will only consider foreground objects for training the regression head.

Also, the classification head needs a good balance between foreground and background proposals as the dataset will be highly imbalanced otherwise. In the imbalanced case and without any other precautions, we could expect the classification head to generally be biased towards predicting everything as non-objects which in a real traffic scenario could be dangerous.

(2) We can see from Figure 2 that the majority of negative bounding boxes are actually easy backgrounds (those having almost no intersection with groundtruth), thus it is beneficial for our network to be able to quickly and precisely learn to identify these easy background. If we don't sample easy backgrounds, our model can only learn representations of hard backgrounds and use these representations to infer whether an easy background is negative sample, which is indirect and can cause difficulties and decrease test classification accuracy. In a word, during validation or test, the model bears the risk of identifying an easy background as a positive sample.

(3) If we set a single IoU threshold as 0.5 for separating foreground and background proposals, we would consider samples that have great uncertainty in their labels if their maximum IoU would be ≈ 0.5. Hence, the quality of these labels would be poor and could be noise information which causes the model thrashing while learning.
We need to match the groundtruth with its highest IoU proposal because in some rare cases there is just no foreground proposal for a certain groundtruth, but we still want to predict it correctly, so we force

a proposal to match this groundtruth and hope by stage-2 refinement it can predict this groundtruth correctly.

4.

For the regression loss we first filter the positive samples where IoU ≥ 0.55 and then compute the loss like $L_{reg} = L_{location} + 3 * L_{size} + L_{rotation}$.

For the classification loss, we first filtered the positive samples where IoU ≥ 0.6 and negative samples where $IoU \leq 0.45$ and then computed the loss using the formula $L_{cla} = L_{pos} + L_{neg}$. Note that no numpy implementation was used in this part since numpy is not supported on GPU. If numpy is used in the loss functions, there would be errors like *RuntimeError: All input tensors must be on the same device. Received cuda:0 and cpu.*

5.

Name	BEV NMS	3D NMS	e_mAP, m_mAP, h_mAP
G52_0611-2324_default_f1014		✓	88.281, 84.379, 78.146
G52_0611-2324_default_f1014 Codalab		✓	87.02, 81.52, 75.78
G52_0624-2143_default_BEV_61b36	✓		87.652, 83.458, 77.306
G52_0624-2143_default_BEV_61b36 Codalab	✓		87.68, 81.95, 76.18
Expected Baseline Reference	✓		81.34, 72.35, 71.11

Table 1: Performance of different NMS methods.

For the implementation of the BEV IoU, we created a new function `get_bev_iou(pred, target)` that works similar to `get_iou(pred, target)` from task 1. But, as the name already indicates, here we restricted the calculation to the ground rectangles of the boxes without considering the height parameters.

The IoU can also be calculated in 3D with the implementation used in the previous exercises. However, there are Pros and Cons that come with that decision:

An argument in favor of using 3D IoU for Non-maximum suppression (NMS) is that two objects that are on top of each other can be recognized, which is obviously not the case for 2D BEV NMS.

An argument against using 3D IoU is that in practice, it is unlikely that two vehicles are stacked on top of each other and even if this happens to be the case (e.g. vehicle trucks), it does in most cases not play an important role for the decision making in a traffic situation. In general, one can expect it to be more likely that cars will be predicted on top of each other even though there is just a single car. 2D NMS can generally prevent this from occuring.

Our experiments show that the performance is almost not influenced whether we use 3D NMS or bird's eye view NMS. The results are depicted in Table.1. We can see from the table bird's eye view has lower the validation accuracy but higher test (codalab) accuracy, so we choose to adhere to bird's eye view NMS in our models in problem 2.

6.

Figure 3 shows the loss and precision curves and Figure 4 shows the scene visualization of beginning, midway, and end of the training cycle.

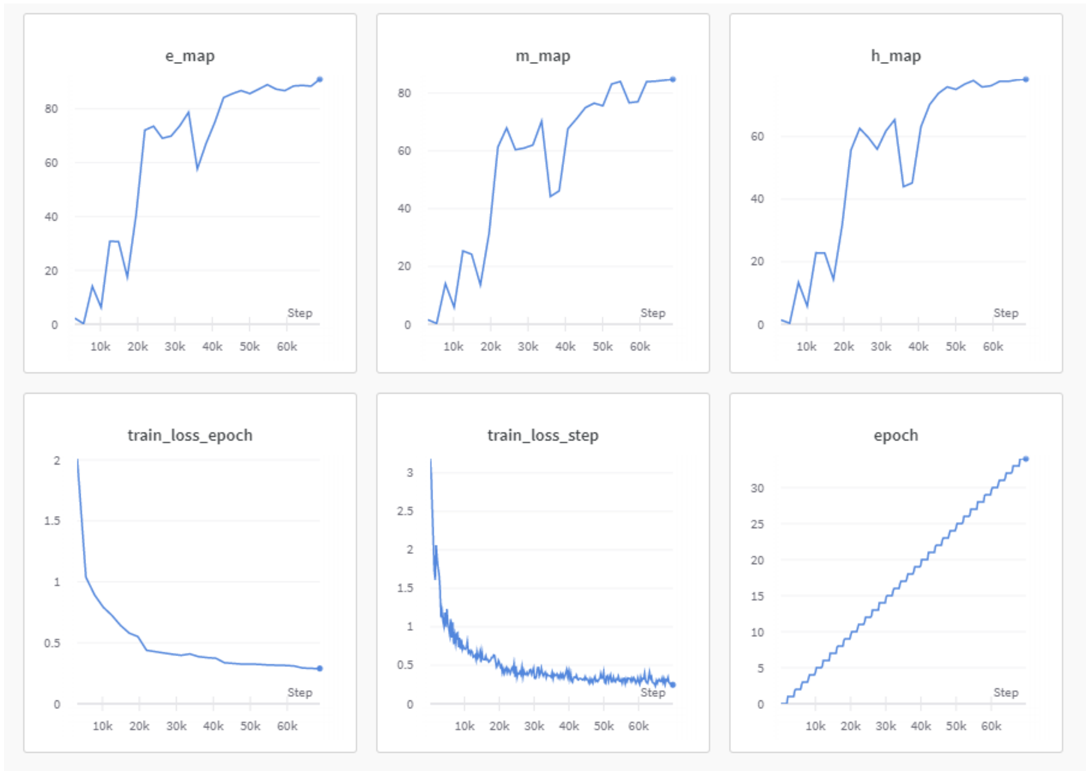
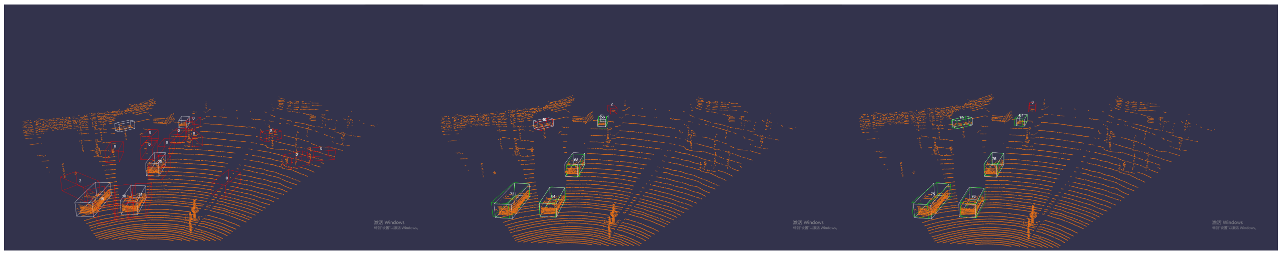


Figure 3: Loss and accuracy curves of the baseline model



(a) beginning (epoch 0) (b) midway (epoch 17) (c) end (epoch 34)

Figure 4: visualization scenes