Deep Learning for Autonomous Driving Lab 1

Team 30
Alan Paul
Loïc Houmard

March 28, 2022

Code structure

In order to run our code properly, you should have the same structure of the files as the one shown above (i.e. all the python files in the codes directory). Our submission doesn't include the provided files (data_utils.py. and load_data.py.) which must be added in the codes directory in order to be included correctly. The code should be run from within the codes directory and the data directory should be in the parent directory, otherwise the paths should be changed.

```
/
├── codes
│   ├── 3dvis.py
│   ├── data_utils.py
│   ├── load_data.py
│   ├── task1.py
│   ├── task2.py
│   ├── task3.py
│   └── task4.py
└── data
    ├── problem_4
    │   └── ...
    ├── data.p
    └── demo.p
```

Problem 1. Bird's eye view

For this first task, we had to show an elevated 2D view of the point cloud. To achieve that, we first round our x and y coordinates to the given resolution (0.2m along each axis in our case). Then, we extract the minimum and maximum value along each axis and use that to compute the number of pixels (the shape of our image), which will only depend on the difference between the max and the min

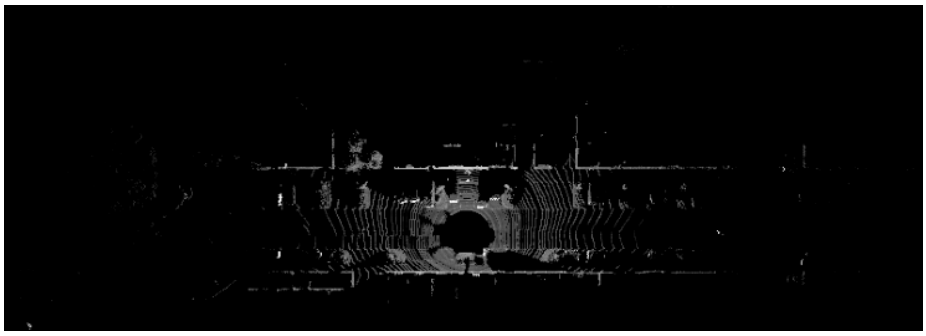


Figure 1: Birds eye view of the data (rotated by 90 degrees for visualization purpose)

and the resolution. Then for every point in the point cloud, we compute to which pixel it corresponds to and if our new point has a higher intensity than what was previously stored, we update the pixel value. We then use matplotlib to show the image (Figure 1).

Problem 2. Visualization

2.1 Point Cloud Projection

In this second task, we first had to project the point cloud given in the world (velodyne) coordinate system into the image of camera 2. For that, we first transform the world points into homogeneous coordinates by stacking a 4th dimension with 1s. Then, we use the extrinsic matrix to go from the world coordinate to the camera 0 coordinate system. We then filter the points and keep only the ones being in front of the camera. We also keep the z-value so that later on, when projecting the points into the camera, if 2 points happened to be mapped to the same pixel, we use the one being closest to the camera. We then use the intrinsic matrix (which is given from camera 0 to camera 2) to project the points from the camera coordinate system into the image coordinate system. We then divide by the last column and keep only the 2 first components to have back our 2D points in inhomogeneous coordinates. We then round the values to have valid pixel locations. Afterwards, we filter the points which are not in the image range. We then use the semantic labels to put the correct color on each of the pixels corresponding to these points, using the closest one if two happened to be projected at the same location. For that, we use the same masks used to filter the points on the labels and then use the color map to map the labels to BGR color which we transform in RGB. We finally draw the image using matplotlib (Figure 2).

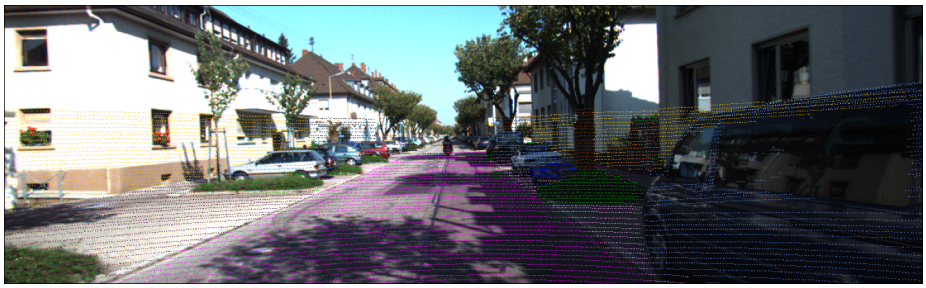


Figure 2: Point cloud projected into image 2 with semantic color.

2.2 Bounding Boxes Projection

In this sub-task, we were asked to project the 3D bounding boxes into image 2. In our solution, we first compute the extrinsic matrix from camera 0 to the velodyne coordinate system by taking the inverse of the matrix going the other way around. Then, for each bounding boxes, we only keep the values which are useful (x, y, z of center of bottom face, width, height, length and y-rotation). Then, for each bounding box, we compute the location of their 8 corners in their own coordinate system (centered at the center of the bounding box). We then compute an extrinsic matrix going from the bounding box coordinate system to the camera 0 coordinate system using the center of the bounding box as translation vector and the y-rotation to create the rotation matrix around the y-axis. We then use our extrinsic matrix previously computed to go from camera 0 to the velodyne coordinate system. We then use the same method as in task 2.1 to project all our bounding boxes' corners into camera 2. Note that we could avoid to project from camera 0 to velodyne to then project back to camera 0, but we kept our code like this for modularity and because we had implemented it this way at first because we had misunderstood the way the intrinsic matrix was given to us. Once we have all the corners in image 2, we simply plot the lines joining the adjacent corners on top of our previous image. We decided to plot the bounding boxes according to the semantic color of the object they represent (here blue since the semantic color for car is blue) since it was not specified in the assignment. The result is shown in Figure 3.

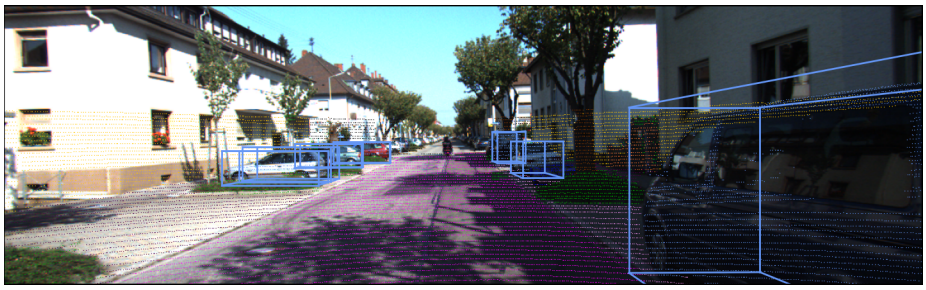


Figure 3: Bounding boxes projected on top of point cloud into image 2.

2.3 3D visualization

In this sub-task, we had to complete the provided code to use the semantic colors for the point clouds and show the bounding boxes in 3D. For the semantic color, for every point, we simply compute the BGR color using the color map, transform it to RGB just as in part 2.1, normalize the values in range [0, 1] and then use the set_data method provided. For the 3D bounding boxes, we just use our previous method to compute the corners location in the world coordinate system and then use the provided function update_boxes. The result is shown in Figure 4

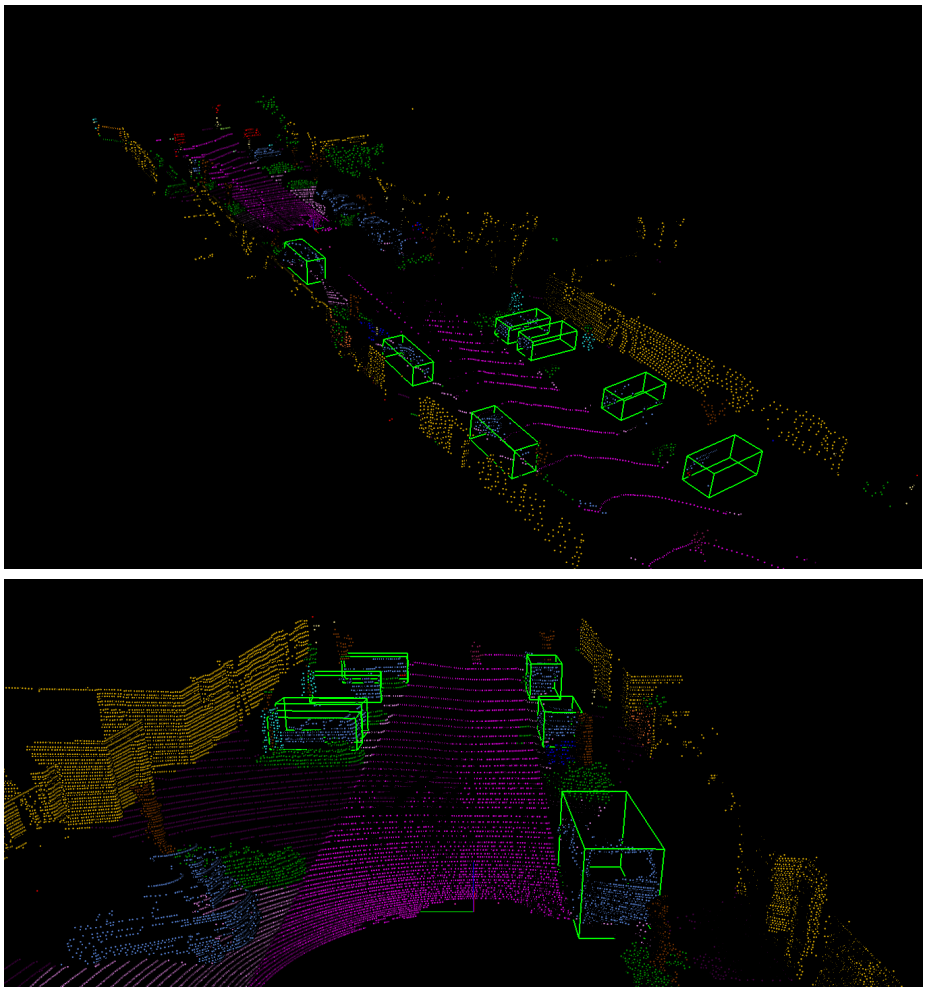


Figure 4: 3D visualization of the scene from 2 different viewpoints.

As we can see, most of the cars in the range of the LiDAR which are in the camera field of view (and hence which are visible in the visualization) are correctly identified. There are only a few points which belong to a car on the right that our network failed to identify (1 car). However, when we look at the image (Figure 3), we see that there are quite a few cars further away which are visible in the image but which are not in the point cloud and which are not identified.

Problem 3. ID Laser ID

For this question, we had to find from which of the 64 channels a point in the point cloud came from, color them with different color and project them into camera 2. For that, we first compute the elevation angle of every point using their norm and their z-coordinates and simple trigonometry. We then divide the total range into 64 bins of equal angle and for each point we check to which bin it corresponds to (same idea as the way we assign every point to a pixel in task 1). The only "problem" with our implementation is that the point having the maximum angle would be assigned to channel 64 (and therefore we would have 65 channels numbered from 0 to 64), hence we assign it to the 64th channel (channel number 63, when 0-indexed). Then we alternate between four colors (green, blue, red, yellow) using modulo on their assigned channel to have their colors. We then project each point and draw it with its assigned color using the methods from task 2. The result is shown in Figure 5).

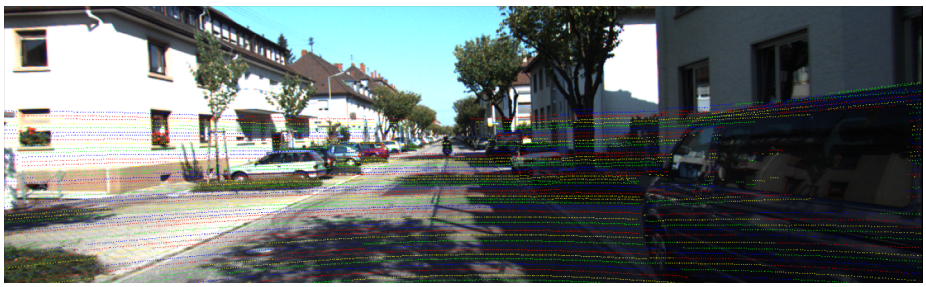


Figure 5: Projected point cloud colored according to their laser ids.

Problem 4. Remove Motion Distortion

For this task, we had to "untwist" the point clouds that are distorted due to motion.

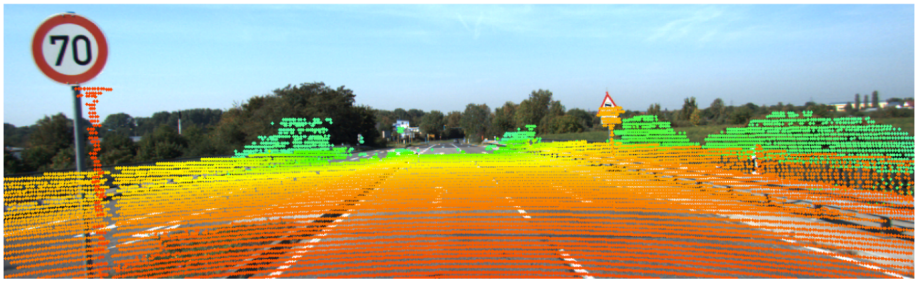
Firstly, to get a feel for the motion distortion for different velocities, we visualize different distorted point clouds projected onto their corresponding images. To colour the points based on distance, we use the depth_color() function. To project the points onto the image, we use the functions we wrote for the previous tasks.

To remove the motion distortion, our first step would be to identify the time differences from the time the image click was triggered by the velodyne, to the time the laser corresponding to each 3D point was fired. For this, we first compute the angle of each 3D point relative to the forward axis (only on the x-y plane, ignoring elevation) using the arctan2 function of numpy. We then compute the angle between the start of the velodyne scan to the point the image click was triggered. This angle can range between 0 and 360 degrees. Using these two quantities along with the timestamp of image trigger and start of scan, we can compute the time difference between each 3D point's laser and the image trigger.

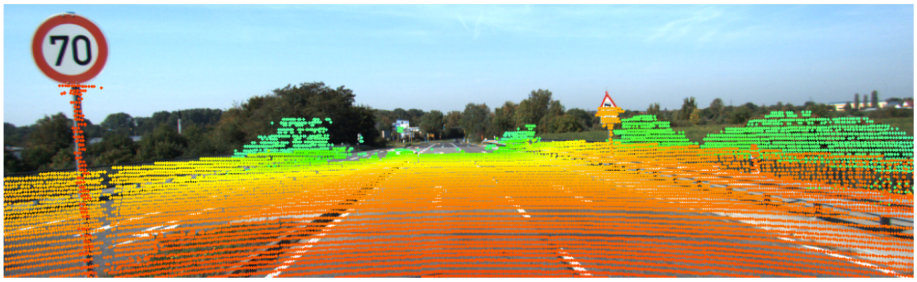
This allows us to calculate the amount of translation distortion and rotation distortion present in the point cloud by multiplying with the translation and angular velocities provided by the IMU sensor. Before this multiplication, we had to transform the point cloud to the IMU's coordinates using the given calibration matrix.

With this information, we built the correction matrix consisting of the Rotation matrix and Translation matrix and retrieved the corrected point cloud in the IMU coordinate system. Finally, we transformed these points back to the Lidar coordinates and then used our previous visualization code to view the point cloud in the image. Figure 6 and 7 show two of our results.

(a) Image



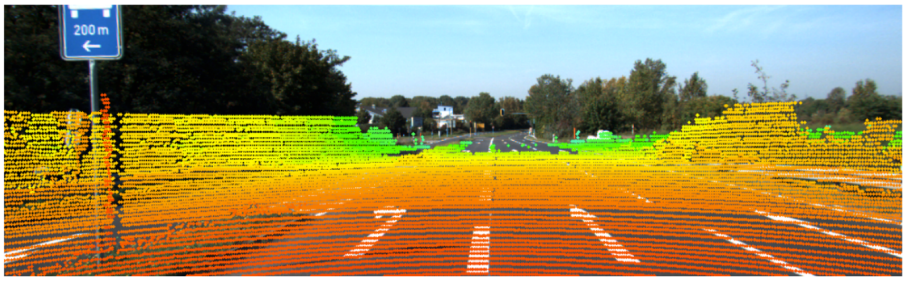
(b) Direct projection without correcting motion distortion



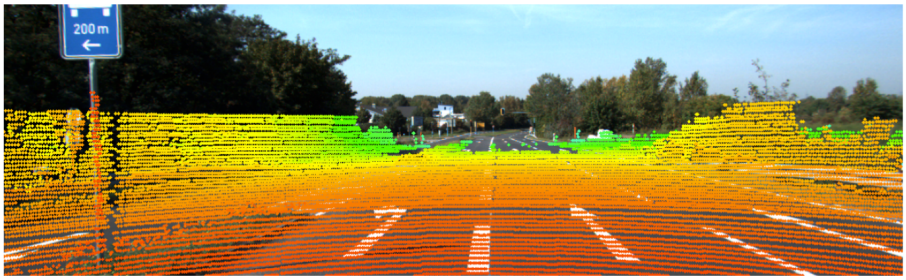
(c) Project after correcting motion distortion

Figure 6: An example ("0000000037") showing how motion distortion looks like when projected to an image (b) and how it looks like when corrected (c). The sign post on the left best illustrates the correction.

(a) Image



(b) Direct projection without correcting motion distortion



(c) Project after correction motion distortion

Figure 7: An example ("0000000077") showing how motion distortion looks like when projected to an image (b) and how it looks like when corrected (c). The sign post on the left best illustrates the correction.

Problem 5. Bonus Questions

1. When we are closer, we cover a wider angle of laser emissions and hence we are more likely to get some laser coming into our eyes or even more than just 1 laser which would lead to more power reaching our eyes. Additionally, the closer we get to a laser source, the greater is its intensity.

2. On a wet road, the water acts as a specular surface that reflects light like a mirror which is very different than in normal conditions. This reflection can create challenges for the two sensors. For the camera, it creates reflections which look like the surrounding objects (and hence our system could believe that there are more objects than there actually are or that some parts of the roads belong to an object making it look closer). For the LiDAR, it can reduce the range of the sensor because some of the laser reflects off the water on the road and go away from the sensor instead of coming back, making our system having more problems seeing the surface of the road. The lasers can also be distorted by the droplets (this has more to do with rain than wet roads, but both often appear at the same time).

3. Since they are not co-centered, some points captured by the LiDAR with different angles might be projected into the same pixel in the image. This problem is more likely to happen if the two sensors are far from each other.