# AER1515 - Perception for Robotics

## Assignment II

Object Detection and Instance Segmentation


- Mahesh Sudhakar
MEng | ECE
1004807328

**Summary:**

This assignment focuses on 2D object detection and instance segmentation using the depth data. Motivated by the KITTI vision challenge for object detection and tracking, disparity map between the corresponding images from left (p2) and right (p3) camera are provided. Depth map for all images are estimated using the given disparity and calibration information, and the results are attached along the submission as advised. An off-the-shelf configured 2D object detection algorithm YOLO v3 pre-trained on COCO dataset is provided for direct implementation. By tuning the confidence and non-maxima suppression threshold values, all cars in the frame of left image are detected and attached in this report. For each detection of objects with label 'car', the average depth is calculated neglecting the values with zero depth. Then a heuristic way for instance segmentation is implemented, to calculate the range of depth values to be considered as that of car, based on the average depth. More detailed information regarding the implementation is provided under each corresponding section. The code used for this assignment is attached along the report with a READ_ME.txt file.

## Part 1: Depth estimation

Motion between a corresponding pair of pixels (pixel difference) in stereo images are used to estimate depth in vision problems using the inverse co-relation between them. The disparity map and calibration information for images in both test and train dataset are provided along the 'kitti_dataHandler.py' python file which had built-in functions to import the calibration classes. Focal length (*f*) and baseline (*B*) for each image is calculated and displayed as part of the code. Then the depth map is calculated using the triangulation formula,

$$\text{Depth } z = f*B / disparity$$

which shows the inverse proportionality between the depth and disparity. As the ground truth data provided uses a LiDAR based depth estimation technique, values over a particular distance are 0. To mimic the same behaviour, we set the depth to be 0, for all pixels whose initial depth were estimated to be greater than 80m or less than 10cm.

During the training process, the calculated depth is compared with the ground truth data, using a Mean Squared Error (MSE) estimation. MSE is the sum of the squared difference between two provided images considering their input dimensions are same. The lower the MSE error, the similar the images. This metric is employed to evaluate the calculated depth map as all further steps depend on them.

### Part 2: 2D Bounding Box Estimation

YOLO is a family of open source real-time object detection algorithm. It is used in many computer vision projects for its low computation cost without much loss in accuracy. YOLO v3 being the latest version, uses a single deep convolutional neural network (DarkNet) with 106 layers that splits the input image into multiple grids. This version demonstrates better

performance in detecting small objects, as detection is performed in three layers over the network at different dimension of images. YOLO v3 is pre-trained on COCO dataset (which has 80 classes) and provided for direct implementation as a config file along its weights.

The algorithm is tuned on the training dataset to be able to detect all the cars present in the left image. Two input parameters namely, the confidence score and the non-maxima suppression threshold are tuned for this purpose on the training data and applied on the test data. The confidence score is set to be '0.5' to be able to detect the cars with lower confidence scores as well, since test image 11 has two cars at very far away distance. The non-maxima threshold is set to '0.65' to get a single tight bounding box around the detected cars in the image. The python code 'part2_yolo.py' provided is modified to be able to identify only the car class (classID = 2) as traffic lights were detected in some images. The obtained co-ordinates of the bounding boxes are saved as a numpy array in the same folder as the output images of YOLO, in order to be able to access them in future. The test images along the detected 2D bounding boxes are attached below for reference.

Test image '`000011.png`'

```
Bounding box co-ordinates are:
[[534, 185, 38, 22], [592, 178, 28, 10]]
```

Note: The third far away car could not be detected correctly even after tuning process.

Figure 1: `000011.png`

Test image '`000012.png`'

```
Bounding box co-ordinates are:
[[482, 186, 62, 45]]
```

Note: The rear side of the car in this test image could not be detected.


Figure 2: `000012.png`

Test image '`000013.png`'

```
Bounding box co-ordinates are:
[[106, 180, 318, 117], [816, 167, 68, 60]]
```

Note: Tuned not to detect traffic lights.


Figure 3: `000013.png`

Test image '`000014.png`'

```
Bounding box co-ordinates are:
[[501, 176, 39, 31]]
```


Figure 4: `000014.png`

Test image '`000015.png`'

```
Bounding box co-ordinates are:
[[1037, 172, 193, 104], [96, 184, 202, 70]]
```


Figure 5: `000015.png`

## Part 3: Instance Segmentation

For each of the test image, the calculated depth map is loaded (as image) along the bounding box co-ordinates (as numpy array). A dummy mask is generated as the shape of the depth map with values of all pixels to be 255. For each of the bounding box within a test image, the centroid is calculated and displayed along its depth value. Then, for each pixel within the detected bounding box, the depth value is appended into an array for estimating the average depth of the object. Here, the depth values corresponding to '0' are neglected (as they may be more than 80m away) as they should not influence the average depth calculation.

A heuristic way to determine the depth threshold for each bounding box is designed based on the range of its corresponding calculated average depth. A maximum and minimum range is set to determine the pixels of the instance (i.e. the car), based on this depth threshold value using the formulae,

```
depth_threshold_min = (1-depth_threshold)*avg_box_depth
depth_threshold_max = (1+depth_threshold)*avg_box_depth
```

where `depth_threshold` is determined based on the average depth.

The pixel values within this range is set to be '0' on the dummy mask generated initially. As this is iterated over a loop for each bounding box detected, we get the instance segmentation mask as per the requirements mentioned in the assignment document. If the bounding box detected has no depth value within it (all 0's) i.e., if the car is far away, it's made to by-pass the loop with a warning message.

### Other ways of implementation tried:

Other ways to threshold the instance mask using Otsu's adaptive thresholding method and mean adaptive thresholding (implemented in OpenCV), have been tried over the weeks. As this requires the bounding box as a separate image in grayscale format, the conversion back to the RGB mask was costly and no better results were generated. So, this idea of implementation was discarded, but may be investigated in the future.

Another idea was to calculate the depth value at the centroid of the bounding box and to set the threshold range corresponding to that. Though this idea may work for the five test images provided, it was discarded as it assumes the centroid of the bounding box to be at the object. An interesting observation made with the segmentation results is that, the pixels of car at windshields (or other see-through area), if has the background environment in it, have much higher depths. So, if that region or the region with sunlight reflection falls in this centroid position, then this idea may fail. So, the basic method proposed in the assignment document was implemented.

The segmentation results along the calculated depth maps are provided in the corresponding folders as advised.