# KITTI Data Exploration and Preprocessing

The KITTI dataset is a popular benchmark for computer vision and autonomous driving, containing stereo images, optical flow sequences, visual odometry data, 3D object detection, and road/lane detection annotations collected from a vehicle equipped with cameras, Lidar, GPS, and IMU sensors. It serves as a standard for developing and evaluating algorithms in autonomous driving research.

The subset of the dataset provided for this task contains the following directories:

1. **image_02:** Contains the images taken from camera number "2".
2. **labels_02:** Contains the annotated labels for objects in images.
3. **calib:** Contains the extrinsic and intrinsic data
4. **oxts:** Contains GPS data collected during the driving sessions.
5. **velodyne:** This directory holds Velodyne Lidar point cloud data.

## Dataset Limitations and Errors

1. **Varying Lighting Conditions:** Differences in lighting conditions across the dataset introduce variability, which may affect algorithm performance and generalization.
2. **Urban Focus:** The dataset captures driving scenarios in urban environments only. While this urban focus is valuable for certain research tasks, it may not fully represent other driving conditions (e.g., rural roads, highways, extreme weather)

## Preprocessing

I have applied the following preprocessing steps for the the dataset:

1. **Downsampling using a Voxel Grid**
    a. **Description:** A voxel grid downsampling technique is applied to reduce the number of points in the point cloud. This method partitions the space into a 3D grid of voxels and condenses points within each voxel to a single point (often the centroid).
    b. **Purpose:** Reduces the computational load for subsequent processing and helps to mitigate the effects of noise in the raw data by averaging out points within each voxel.
2. **Ground Plane Removal using RANSAC**
    a. **Description:** A RANSAC (Random Sample Consensus) algorithm is used to identify and remove the ground plane from the point cloud. This method iteratively selects a random subset of points, fits a plane, and then removes points that fit the model within a defined threshold.
    b. **Purpose:** The removal of the ground plane focuses subsequent processing on objects of interest like vehicles and pedestrians, which is especially useful in tasks related to autonomous driving and navigation.

3. **Normal Estimation**
    a. **Description:** Normals are estimated for each point in the point cloud based on the local neighborhood of points. This typically involves fitting a plane to the nearest neighbors of each point and calculating the plane's normal vector.
    b. **Purpose:** Normal vectors are crucial for many 3D processing tasks, such as segmentation and surface analysis, and they facilitate the identification of geometric features.
4. **Data Augmentation through Random Rotations**
    a. **Description:** The point cloud is randomly rotated around the origin. This involves generating a random rotation matrix and applying it to all points in the cloud.
    b. **Purpose:** Enhances the robustness of models that will later use this data by simulating various orientations, helping the models generalize better to unseen data during testing or deployment.

## Object Detection Models

### Detecting Objects using Images

I trained a model which uses a Faster R-CNN with a ResNet-50. Faster R-CNN first uses a Region Proposal Network (RPN) to generate candidate object locations and then uses a RoI (Region of Interest) pooling layer to extract features for each candidate box. These features are then used to classify the object in the box and refine the box coordinates.

**Loss Functions:**

The Faster R-CNN model uses a combination of classification and regression losses. The classification loss computes the difference between expected and true class probabilities. The regression loss computes the difference between expected and actual bounding box adjustments.[1]

$$L(p_i, t_i, v_i) = \frac{1}{N_{cls}} \sum_i L_{cls}(p_i, p_i^*) + \lambda \frac{1}{N_{reg}} \sum_i p_i^* L_{reg}(t_i, v_i)$$

### Detecting Objects Using the Lidar point cloud

To detect objects in using the Lidar point cloud, I decided to use SECOND (Sparsely Embedded Convolutional Detection) model. For this purpose, I installed OpenPCDet library, which provides an implementation of SECOND among other models.

OpenPCDet is an open-source library for 3D object detection based on point cloud data, providing a collection of state-of-the-art models and tools for efficient development and evaluation in autonomous driving and related applications. It supports various popular datasets, including KITTI, and offers modular and extensible code to facilitate research and development in 3D perception.[2]

### Difference between Lidar object detection and image object detection neural networks

The main architectural difference between Lidar object detection neural networks and image object detection neural networks lies in the nature of the input data and the corresponding processing techniques. Lidar object detection networks are designed to handle sparse and irregular point cloud data, which provide 3D spatial information about the environment. These networks often use architectures that can effectively process and extract features from 3D point clouds, such as voxel-based methods, point-based methods, or a combination of both. Voxel-based methods convert the point clouds into a structured grid of voxels, enabling the use of 3D convolutions to extract features, while point-based methods, like PointNet and PointNet++, directly operate on the point clouds to capture local and global structures.

On the other hand, image object detection neural networks are designed to work with dense and regular 2D grid data, which is inherently different from the sparse 3D data provided by Lidar. These networks typically employ convolutional neural networks (CNNs) to extract hierarchical features from images. Popular architectures, such as Faster R-CNN, YOLO, and SSD, utilize 2D convolutions to process the images and generate region proposals or predict bounding boxes and class labels directly from the image features. The regular structure of image data allows for the use of well-established convolutional operations, pooling layers, and other techniques that efficiently capture spatial hierarchies and patterns in the data.

## Sensor Fusion

### Overview of Sensor Fusion

Sensor fusion is the process of integrating sensory data from multiple sources to produce more accurate, reliable, and comprehensive information than what could be derived from any single sensor alone. I utilized data from both RGB cameras and Lidar sensors to enhance object detection capabilities. The fusion algorithm leverages the strengths of each sensor, combining the rich color and texture information from the RGB images with the precise depth and spatial information from Lidar point clouds.

### Fusion of RGB and Lidar Detections

The fusion algorithm I developed combines detections from both Faster R-CNN and SECOND models. First, the 3D bounding boxes from the Lidar detections are projected onto 2D image planes. This involves transforming the 3D coordinates into 2D coordinates while preserving the spatial relationship. Once both sets of detections are in the same 2D coordinate system, they are concatenated into a unified set of bounding boxes, labels, and scores. This fused detection set benefits from the complementary strengths of both sensors, resulting in more robust and reliable object detection.

To merge the detections, I use a weighted average of the confidence scores from both sensors to determine the final confidence score for each fused detection. The bounding boxes are adjusted by calculating the intersection-over-union (IoU) between overlapping boxes and merging them accordingly. This approach ensures that the final detections are accurate and take into account the contributions from both RGB and Lidar data.

# Calibration

### Overview

Sensor calibration is crucial in achieving accurate measurements and reliable data from various sensors used in autonomous systems and computer vision applications. It ensures that the sensors' measurements are accurate and aligned, which is essential for tasks like object detection, mapping, and navigation. Intrinsic calibration involves determining the internal parameters of a sensor, such as focal length, optical center, and lens distortion, to correct for any optical imperfections. Extrinsic calibration, on the other hand, involves establishing the position and orientation of the sensor in relation to a reference coordinate system, which is vital for integrating data from multiple sensors.

### Camera Calibration

I have implemented an algorithm for camera calibration that leverages a checkerboard pattern to determine the camera's parameters. The process begins by capturing images of a checkerboard from a camera and detecting the inner corners of the checkerboard pattern. These detected points are then used as image points. OpenCV's *calibrateCamera* function is then employed to compute the intrinsic and extrinsic camera parameters. The *calibrateCamera* function estimates the following parameters:

1. ***Camera Matrix*:** This matrix includes the focal length and optical center of the camera.
2. **Distortion Coefficients:** These coefficients account for lens distortion, which causes straight lines to appear curved in the image. Correcting this distortion is crucial for accurate image representation.
3. **Rotation Vectors:** These vectors describe the orientation of the camera relative to the checkerboard, allowing for the alignment of the camera's coordinate system with the real-world coordinate system.

4. **Translation Vectors:** These vectors indicate the position of the camera relative to the checkerboard, which is necessary for determining the camera's location in space.

The mathematical foundation of this calibration algorithm involves solving a system of equations that relate 2D image points to their corresponding 3D object points in real space. This is achieved through an optimization process that minimizes the re-projection error, which is the difference between the projected points and the detected image points.

**Camera/Lidar Extrinsic calibration:**

Camera/LiDAR extrinsic calibration involves determining the spatial relationship between a camera and a LiDAR sensor. This calibration is crucial for sensor fusion applications where data from both sensors need to be accurately aligned in a common coordinate system.

In developing my sensor fusion algorithm, I utilized the calibration parameters provided with the kitti dataset to make sure the data from the two sensors are aligned. Specifically, I used the camera matrix (`P2`) and the transformation matrix from LiDAR to camera coordinates (`Tr_velo_to_cam`). These parameters were crucial in transforming 3D bounding boxes from LiDAR space to 2D bounding boxes on the camera image plane.

## Wrapping in ROS2 Nodes

I have created a ROS-2 package designed for object detection and sensor fusion, utilizing data from both images and point clouds. The package contains three primary nodes: `image_detection_node`, `pc_detection_node`, and `fusion_node`. The `image_detection_node` processes image data to detect objects using a pre-trained Faster R-CNN model, while the `pc_detection_node` handles LiDAR point clouds for object detection and get detection using SECOND model. The `fusion_node` integrates the detection results from both image and point cloud data to provide a comprehensive understanding of the environment.

The nodes mentioned above works only when there are images and point clouds published in the respective topics. To test the nodes functioning, I have implemented two more nodes which are `image_publisher_node` and `pc_publisher_node`. These nodes publish a sample image and point cloud in the respective topics.

## Unit Testing

Unit testing is a crucial aspect of validating the correctness and reliability of the individual components within the object detection and sensor fusion pipeline. In this project, I utilized the pytest framework to conduct thorough unit tests, focusing on ensuring that each module performs its designated function accurately.

The unit testing process involved creating test cases that assess the core functionalities of each component. These tests were designed to validate the accuracy and reliability of the code by comparing the actual outputs against expected results. Any discrepancies identified during testing were promptly addressed, leading to refinements and improvements in the respective modules.