



Universiteit  
Leiden  
The Netherlands

# Data Science & AI

## YOLO-Based Fusion Models for Real-Time 3D Object Detection

Emmanouela Stranomiti

1st Supervisor:

Dr. Erwin M. Bakker

2nd Supervisor:

Prof. Dr. Michael S.K. Lew

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)

[www.liacs.leidenuniv.nl](http://www.liacs.leidenuniv.nl)

01/07/2025

## Abstract

Kieffer’s (2024) [1] open-source YOLO-LiDAR-Fusion framework, which formed the basis of this thesis, is expanded upon by this methodology. With an inference time of 0.2425 seconds, Kieffer’s initial pipeline produced 2D segmentation masks using YOLOv8m-seg and fused them with LiDAR data for 3D object detection. The results were moderate, with an average IoU of 20.28% for cars and 20.37% for pedestrians when filtering parameters were set to (20, 20). Several significant improvements were made in this thesis to increase the efficiency and accuracy of detection. The most recent segmentation-capable variant, YOLOv11m-seg, was used in place of the YOLOv8 backbone. Using the same filtering configuration, it produced faster inference times of 0.1008 seconds and higher detection scores of 26.98% (car), 33.26% (pedestrian), and 14.26% (cyclist). Additionally, preprocessing steps like depth filtering, erosion-based mask refinement, and PCA for 3D bounding box alignment were re-implemented and assessed. The most per-class average IoUs were 29.57% (car), 35.25% (pedestrian), and 16.57% (cyclist) after a more thorough grid search over erosion and depth values (ranging from 10 to 70) revealed class-specific optimal configurations. Furthermore, Kieffer’s original implementation was compared to the fusion pipeline’s testing across several YOLO variants, including YOLOv8m, YOLOv9e, and YOLOv11m. Our YOLOv8m-seg alone outperformed his version by +11.32% (car) and +3.81% (pedestrian), while also being nearly three times faster. The Nano version had the fastest runtime (0.0402s), while the Extra Large version had the highest car IoU (28.02%), according to our evaluation of the impact of YOLOv11 model scaling. Lastly, the thesis outlines limitations and suggests future work, including class-specific post-processing, expanding the model to detect additional classes from the KITTI dataset, and further training the model on other datasets, such as Waymo or NuScenes, to improve generalizability.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Related Work</b>	<b>3</b>
<b>3</b>	<b>Fundamentals</b>	<b>4</b>
3.1	Object Detection . . . . .	5
3.2	Popular Object Detectors . . . . .	5
3.2.1	Region-Based Detectors . . . . .	5
3.2.2	Regression-Based Detectors . . . . .	6
3.3	Performance Metrics . . . . .	8
3.3.1	Intersection over Union (IoU) . . . . .	8
3.3.2	Precision and Recall . . . . .	8
3.3.3	Average Precision . . . . .	9
3.3.4	Mean Average Precision . . . . .	10
3.4	YOLO . . . . .	11
3.4.1	YOLO Evolution . . . . .	11
3.4.2	YOLOv11 Architecture . . . . .	13
3.4.3	YOLOv11 Training, Segmentation Model, and Performance . . . . .	15
3.5	Clustering . . . . .	18
3.5.1	KMeans Clustering Algorithm . . . . .	19
3.5.2	DBSCAN Clustering Algorithm . . . . .	20
3.6	Sensor Calibration . . . . .	22
3.6.1	LiDAR-Camera Transformation . . . . .	22
<b>4</b>	<b>KITTI Dataset</b>	<b>24</b>
4.1	Dataset Presentation . . . . .	24
4.2	Dataset Evaluation . . . . .	25
<b>5</b>	<b>Methodology</b>	<b>26</b>
5.1	Baseline Model . . . . .	26
5.2	Our Work . . . . .	29
5.3	Preprocessing . . . . .	29
5.4	Calibration . . . . .	31
5.5	Fusion . . . . .	32
5.6	3D Bounding Boxes . . . . .	33
5.7	Model Configuration . . . . .	34
5.8	Risk Analysis and Mitigation . . . . .	35
5.8.1	Ethical Considerations . . . . .	35
<b>6</b>	<b>Experiment Results and Discussion</b>	<b>35</b>
6.1	Effect of Segmentation Mask Erosion and Depth Filtering . . . . .	35
6.2	Optimizing Segmentation Mask Erosion and Depth Filtering . . . . .	36
6.3	PCA for 3D Bounding Boxes . . . . .	39
6.4	YOLO Variant Evaluation . . . . .	40

6.5	YOLOv11 Scaling Evaluation . . . . .	42
6.6	Limitations and Future Work . . . . .	43
<b>7</b>	<b>Conclusion</b>	<b>44</b>
	<b>References</b>	<b>48</b>



# 1 Introduction

Over the past 20 years, autonomous vehicles (AVs) have advanced quickly thanks to major developments in deep learning, vehicle dynamics, and sensor technology. Both technological and economic factors are contributing to the growing interest in AVs. The global market for AVs was estimated to be worth 1.7 trillion USD in 2024, and is expected to reach 3.9 trillion USD by 2034, with a compound annual growth rate (CAGR) of 8.6% [2]. One of the main factors driving this increase is the belief that AVs will significantly lower the number of traffic-related deaths by removing human driving errors. Currently, it is estimated that over 1.3 million deaths per year are caused by human error, which accounts for 80-90% of the total traffic-related deaths. [3].

However, this requires understanding the driving environment, which includes extreme weather conditions like intense rain or strong sunlight, that can quickly damage sensors. In addition, it is difficult to anticipate the unpredictable behavior of other road users [4]. Furthermore, meeting the real-time and high-accuracy requirements of autonomous driving is challenging due to the enormous volume of input data required for object detection. Thus, accurate 3D object detection is essential for autonomous vehicles (AVs) to identify and track cars, pedestrians, and other obstacles, and to avoid them properly [5].

In order to obtain information about the environment, modern AVs frequently employ a variety of sensors, most notably RGB cameras and LiDAR (Light Detection and Ranging). RGB cameras provide rich texture of visual information, which helps with 2D object detection tasks. However, they are limited in their ability to comprehend the true size, shape, and position of objects in 3D space. This is especially problematic in complex scenes with objects that overlap or are at different distances.

Furthermore, RGB cameras are passive sensors, which means they only rely on ambient light to function. As a result, their performance can be greatly influenced by lighting conditions. Shadows, glare, low light, and bright reflections can distort the captured image, reducing detection accuracy. These limitations make it difficult for camera-only systems to reliably perceive the spatial structure of their surroundings, which is necessary for tasks like autonomous driving and 3D object detection. [5].

In contrast to RGB cameras, LiDAR uses lasers to detect its surroundings, which are less impacted by ambient light. This allows for precise measurements of an object’s distance. LiDAR sensors scan the environment with laser pulses, delivering precise range and 3D position of objects. High-end LiDAR (64-beam or more) yields detailed 3D point clouds, but low-resolution LiDAR (for example, 16 or 32 beams) provides much sparser data, making object recognition challenging if used in isolation. But, even with high-resolution LiDAR [6], the point clouds are extremely sparse when compared to the feature-rich and high-resolution information of an RGB image.

As a result, fusing depth information from LiDAR point clouds with feature-rich information from RGB images is expected to be very effective for enhancing 3D object classification and localization. The fused data mitigates the limitations of each of the individual sensors while enhancing overall detection performance [5]. However, the computational burden of several state-of-the-art techniques limits their applicability for real-time deployment in autonomous systems with limited resources [7]. There is an urgent need for lightweight alternatives that maintain high detection accuracy

while being compatible with embedded or low-power hardware platforms that often have limited computational resources.

## *Motivation*

This thesis investigates how contemporary lightweight YOLO-based architectures, such as YOLO-LiDAR Fusion [1] by Kieffer (2024), can be modified to enhance 3D object detection.

Kieffer’s (2024) [1] fusion pipeline showed that a modular and interpretable YOLO-based architecture can provide respectable performance, but its full potential was constrained by its dependence on YOLOv8. There is a timely opportunity to investigate whether incorporating these architectures into lightweight fusion pipelines can reduce the performance gap with more complex state-of-the-art models, given the introduction of newer YOLO models that greatly enhance feature representation and segmentation quality. In order to overcome the trade-off between accuracy and efficiency in LiDAR-camera fusion, this thesis makes use of the most recent developments in YOLO segmentation networks. It does this by methodically assessing their effects and comparing their performance on the KITTI dataset using an enhanced version of Kieffer’s pipeline. The objective is to assess whether such models can achieve or even match the performance of state-of-the-art techniques such as VPFNet [8], LoGoNet [9], or VirConv-S [10] while lowering computational resources.

### *Research Question:*

How can a LiDAR-camera fusion pipeline be adapted to work on a more recent YOLO model, and how does that affect its results?

## *Contributions*

This thesis investigates how a lightweight YOLO-based fusion pipeline [1] can be improved by using the latest YOLO model to achieve efficient and accurate 3D object detection. The primary contributions of our work are as follows:

1. This thesis uses Kieffer’s (2024) [1] LiDAR-camera fusion to make multiple contributions to the field of lightweight 3D object detection. As a starting point for our final performance comparison, we conduct a comprehensive literature review of recent state-of-the-art fusion models evaluated on the KITTI dataset, including LoGoNet, ViKIENet-R, and VirConv-S. The baseline model, created by Kieffer, is a modular fusion pipeline that serves as the basis for our work.
2. We replicate Kieffer’s pipeline and use the more recent YOLOv11m-seg model in place of the original YOLOv8m-seg backbone. Both accuracy and efficiency are significantly increased as a result of this change. Our implementation reduces inference time to 0.0948 seconds, more than twice as fast as Kieffer’s version (0.2472 seconds), while achieving average IoUs of 28.00% for cars, 31.41% for pedestrians, and 14.70% for cyclists with erosion and depth filtering set to (25, 25).

3. Next, we compare several YOLO variations using the same fusion configuration. The most balanced model overall is YOLOv9e-seg, which has the highest average IoU for cyclists (14.94%) and pedestrians (32.34%), while YOLOv11m-seg has the best result for cars (28.00%) with competitive runtime.
4. The LiDAR-camera fusion pipeline evaluates three different YOLO versions (YOLOv8, YOLOv9, and YOLOv11). YOLOv11m-seg produced the highest average IoU for cars (28.00%), whereas YOLOv8m-seg produced the average IoU results for pedestrians (90.05%) and cyclists (59.88%) with the shortest inference time (0.0836 s). This highlights the different strengths of each model based on the object class and application context.
5. We then investigate how model scaling affects the YOLOv11 family. While the Extra Large model marginally enhances car (28.02%) and pedestrian (31.90%) detection, we find that YOLOv11-Nano achieves the fastest inference time (0.0402 seconds) and the highest cyclist IoU (15.32%). All sizes, however, yield comparable outcomes, indicating that larger models offer diminishing accuracy returns.
6. Lastly, we discuss the limitations of the model and suggestions for future work, such as dataset expansion, addition of other classes, and the use of class-specific post-processing.

## 2 Related Work

This literature review will outline the principles and constraints of the most advanced techniques currently available for camera LiDAR sensor fusion methods for 3D object detection in autonomous driving tasks, evaluated on the KITTI dataset [11]. These serve as a comprehensive understanding of the various fusion techniques present in the literature that are utilized for 3D object detection. In addition, we introduce Kieffer’s work, which serves as a baseline to our model.

One of the state-of-the-art models for 3D object detection, by Wu et al. (2023) [10], presents the VirConv-S model, which uses Stochastic Voxel Discard (StVD) to reduce the computational cost and Noise-Resistant Submanifold Convolution (NRConv) to improve robustness against noisy inputs. By incorporating their VirConv, they create an effective pipeline, VirConv-L, based on an early fusion design. They then construct a high-precision pipeline called VirConv-T using a transformed refinement scheme. Finally, they create the semi-supervised pipeline VirConv-S using a pseudo-label framework. On the KITTI car 3D detection test leaderboard, their VirConv-L achieves 85% AP. Their VirConv-T and VirConv-S achieve a high precision of 86.3% and 87.2% AP, respectively, and are each currently ranked third and first.

VikIENet-R, proposed by Yu et al. (2025) [12], introduces a multi-modal feature fusion framework called the virtual key instance enhanced network (ViKIENet), which combines the features of LiDAR points and virtual key instances (VKIs) in a number of steps. Semantic key instance selection (SKIS), virtual-instance-focused fusion (VIFF), and virtual-instance-to-real attention (VIRA) are the three primary components of our contributions. Additionally, we suggest the expanded ViKIENet-R with VIFF-R version, which incorporates rotationally equivariant features. The results of the experiment demonstrate that ViKIENet and ViKIENet-R significantly enhance detection performance. It

attains an AP of 86.04% for automobiles on the KITTI dataset with an inference time of 0.06 seconds.

Xin Li et al. (2023) [9] present a new Local-to-Global fusion network (LoGoNet) that combines voxel-level global fusion and region-level local fusion. It uses Centroid Dynamic Fusion (CFD) and Feature Dynamic Aggregation (FDA) to improve geometric precision and semantic consistency. LoGoNet achieves 85.06% AP on the KITTI Dataset.

By using a voxel-to-point feature fusion technique, VPFNet (Voxel-to-Point Fusion Network) [8] by Zhu et al. (2023) offers a hybrid architecture that connects voxel- and point-based representations. This design enables the model to take advantage of the structured regularity of voxel-based processing while preserving the geometric accuracy of point-level features. The use of a spatial-aware transformer module, which improves long-range contextual reasoning across the 3D space, is a significant innovation in VPFNet. The model enhances object localization accuracy and semantic understanding by better aligning image and point cloud features. VPFNet achieves 83.21% AP on the car class on the KITTI benchmark.

A novel cross-modal 3D object detector called UPIDet is presented by Zhu et al. (2023) [13] with the goal of maximizing the image branch’s potential from two angles. First, normalized local coordinate map estimation is a brand-new 2D auxiliary task that UPIDet presents. With this method, sparse point clouds can be enhanced by learning local spatially aware features from the image modality. Second, they find that by using a concise and efficient point-to-pixel module, the gradients backpropagated from the image branch’s training objectives can improve the point cloud backbone’s representational capability. At the time of submission, the model placed second in the fiercely competitive KITTI benchmark cyclist class, achieving an AP of 74.32%.

This thesis is based on the lightweight fusion pipeline presented in Kieffer’s master’s thesis (2024) [1], which integrates LiDAR point clouds with the YOLOv8m-seg model. His method consists of depth pruning, segmentation-based filtering, and optional Principal Component Analysis (PCA) 3D bounding box refinement. The open-source pipeline offers a modular structure that is ideal for adaptation, even though it does not benchmark against the most advanced models on KITTI. In order to further investigate lightweight fusion strategies in real-time settings, this thesis expands parameter tuning, class-wise analysis, and SOTA benchmarking while replicating Kieffer’s architecture with the most recent YOLOv11m-seg variant.

### 3 Fundamentals

This section provides an overview of key concepts and terms related to the LiDAR and camera data fusion method for object detection. It starts with an overview of object detection, outlining the task’s objectives and challenges in both 2D and 3D contexts. Popular object detectors are reviewed, with a special emphasis on the YOLO (You Only Look Once) family. The section also goes over the performance metrics commonly used to evaluate object detection models, such as Intersection over Union (IoU) and Mean Average Precision (mAP). Furthermore, it describes key techniques such as clustering, which is required for post-processing segmented point clouds, and sensor calibration, which allows for accurate alignment of LiDAR and RGB camera data. These topics, taken together, provide the theoretical foundation for implementing and evaluating the proposed LiDAR-camera fusion pipeline.

### 3.1 Object Detection

The process of identifying and categorizing several objects for a given input data set, like an image or a point cloud, is known as object detection. The main objective of object detection is to identify the objects in the input data and surround them with labeled bounding boxes. The location of the object within the coordinate system of the corresponding input data is indicated by these bounding boxes. Each detected object's corresponding category—such as car, pedestrian, or bicycle—is specified by the labels. Object detection is an essential part of autonomous cars that enables them to identify an ideal and collision-free path by detecting objects such as other cars, people, and traffic signs.

### 3.2 Popular Object Detectors

Prior to 2014, hardware components and deep learning were not yet advanced enough to enable quick and precise object detection. There were three primary steps in traditional object detection methods:

1. Region selection involves inspecting images with a multi-scale sliding window approach to locate regions of objects of different sizes. However, this approach is computationally demanding and frequently results in duplicate detections.
2. Feature extraction is performed after object localization to provide representations for object recognition. Creating an accurate feature descriptor that recognizes any kind of object in images with contrasting backgrounds, perspectives, and lighting can be challenging.
3. Classification assigns labels to recognized objects based on their extracted features.

The manual feature extractors used in the conventional object detection techniques are unreliable and computationally expensive. Deep learning is now utilized for object recognition in order to get around these shortcomings of conventional techniques. Deep learning is used by two different kinds of object detectors: one-stage and two-stage detectors.

#### 3.2.1 Region-Based Detectors

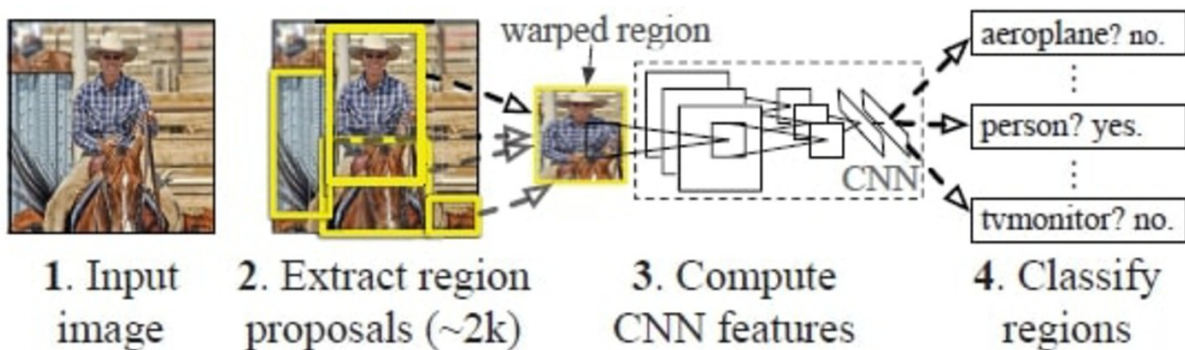


Figure 1: RCNN Model Architecture [14].

Two-stage object detectors separate the task into object localization and classification. This indicates that the detector initially generates ideas for possible object locations. The object is then categorized by the detector using the corresponding category and the extracted features within the generated regions. These detectors, also known as region-based frameworks, have the benefit of high detection accuracy but the drawback of faster inference. Ross Girshick et al.’s Region-based Convolutional Neural Network (RCNN) [14] is the first object detection model to successfully use deep learning. Figure 1 displays a general overview of the model. Three primary modules make up the RCNN model [14], [15]:

1. Selective search extracts category-independent region proposals, which function as potential detections for the next module. In this first module, RCNN recognizes about 2000 regions based on variations in textures, colors, and scales.
2. To fit the CNN input, the region proposals are first rescaled to fixed length vectors and then processed by a large CNN. The CNN then extracts features from each region.
3. The third module employs class-specific Support Vector Machines (SVM) to determine whether an object is present in a region. The SVM-generated bounding boxes are refined using a linear regression model.

Although the RCNN model has shown significant improvements in accuracy and speed, it remains inefficient when processing each region separately by the CNN, resulting in redundant computations and storage requirements. Ross Girshick proposed the Fast RCNN model [16] to enhance the previous RCNN object detector. The image is processed through a CNN to create a convolutional feature map, which is then used to derive region proposals. A fixed-length feature vector is then extracted from each region proposal by a Region of Interest (RoI) pooling layer. Following that, these vectors are processed by a few fully connected layers, one of which generates four values that establish the bounding box corners and the other of which generates soft-max probability estimates over the selection of class labels.

The RCNN model is greatly enhanced by the Fast RCNN model in terms of training performance, efficiency, and inference speed. Nevertheless, Fast RCNN continues to produce region proposals through selective search, which is slow and computationally costly. Faster RCNN [17], a novel Region Proposal Network (RPN) that is a fully connected CNN, was introduced by Shaoqing Ren et al. to address this drawback. Like the initial stage of the Fast RCNN model, the RPN creates region proposals straight from the shared feature map produced by a CNN. To put it simply, the RPN instructs the Fast RCNN module where to look by feeding its output into the detector. Since the RPN module generates region proposals significantly faster than selective search, the Faster RCNN model can detect objects in real time. Additionally, accuracy and end-to-end training are enhanced by Faster RCNN.

### 3.2.2 Regression-Based Detectors

Two-stage detectors, such as the R-CNN variants, are typically slower but more accurate because they first produce region proposals before refining them in a second stage. Through a single inference



pass, one-stage detectors such as YOLO and Single Shot MultiBox Detector (SSD) directly predict boxes on a dense grid of locations, sacrificing some accuracy in exchange for significantly faster speed [18].

One-stage object detectors, also known as region proposal-free frameworks, work by integrating the tasks of object classification and object localization. Therefore, region proposals are no longer required prior to the detection process, and bounding box coordinates and class scores can be directly predicted from the input image with just one neural network pass. CNNs based on regression are used in these one-stage detectors. The process of predicting a continuous value from one or more input variables, also referred to as features, is called regression. These continuous values are the size and bounding box coordinates used in object detection.

You Only Look Once (YOLO), introduced by Joseph Redmon et al. [18], was the first pertinent regression-based object detector. After predicting the objects' bounding box coordinates, it calculates the likelihood that they belong to a particular category. YOLO employs features from the entire image to make its detections, as opposed to region-based detectors. In order to accomplish this, the model divides the image into a grid, with each cell predicting bounding box coordinates and confidence scores for each category. The bounding box with the highest Intersection over Union (IoU) with the other boxes in the cell is chosen, along with the class with the highest confidence score in each cell.

Wei Liu et al. [19] introduced the Single Shot Multi Box Detector (SSD), another pertinent regression-based object detector. The SSD model employs a feed-forward CNN that generates fixed-size bounding boxes and confidence scores for object classes inside of them, much like the YOLO model does. The final detections are then obtained by applying a non-maximum suppression step that eliminates redundant boxes based on IoU. Anchor boxes, which are predefined boxes of different sizes and aspect ratios positioned at various points on the feature maps, are used in the SSD model in contrast to YOLO. Because of this, SSD can extract features with varying aspect ratios and sizes, ensuring high recognition accuracy. In terms of speed and accuracy, the SSD model performs better than the simple YOLO model.

Over time, the YOLO model was further refined to address its shortcomings and enhance its overall functionality. Section 3.4 goes into further detail about YOLOv12, which is the most recent version as of the writing of this thesis.

One-stage designs are favored for on-vehicle deployment due to the real-time demands of driving. YOLO is incredibly quick and adaptable to new domains thanks to its straightforward "you only look once" design [18]. For instance, the original YOLO v1 achieved 45 frames per second on PASCAL VOC using a Titan X GPU, and a faster variant reached 155 frames per second using the same hardware. At the time, this achieved the highest detection accuracy among real-time systems.

### 3.3 Performance Metrics

Several metrics can be used to assess how well object detection frameworks work. By using the three primary outputs of object detectors—object label (class), bounding box, and confidence score—these metrics assist scientists in comparing various models on a variety of datasets. The following subsections list the most widely used evaluation metrics. First, it’s important to explain the fundamental ideas behind these metrics [20]:

- **Ground Truth (GT)**: The manually annotated object classes and bounding boxes in a dataset’s images make up the ground-truth data. The model is trained using this data, and its performance is subsequently assessed by evaluating its predictions with the ground-truth annotations.
- **True Positive (TP)**: The model accurately predicts bounding boxes that match GT bounding boxes and classes.
- **False Positive (FP)**: The model may predict a bounding box that does not exist in the GT data or incorrectly detects an existing one.
- **False Negative (FN)**: The model misses some objects in the GT data, resulting in their detection failure.
- **True Negative (TN)**: The model accurately predicts the absence of an object when it is not detected or present in the GT data. This concept is not applicable in object detection due to the infinite number of bounding boxes that cannot be detected within an image.

#### 3.3.1 Intersection over Union (IoU)

The definitions above require defining “*correct detection*” and “*incorrect detection*”. One common approach is to use intersection over union (IOU). The measurement is based on the Jaccard Index, a coefficient of similarity between two sets of data. In the object detection scope, the IOU calculates the overlapping area between the predicted and ground-truth bounding boxes ( $B_p$  and  $B_{gt}$ ), divided by the area of the union between them. Thus, the IOU is defined in Equation 1 and is illustrated in Figure 2.

$$J(B_p, B_{gt}) = \text{IOU} = \frac{\text{area}(B_p \cap B_{gt})}{\text{area}(B_p \cup B_{gt})} \quad (1)$$

The IoU value can be compared to a threshold  $t$  to determine whether a detection is correct or incorrect. If  $\text{IOU} \geq t$ , the detection is considered correct (true positive). If  $\text{IOU} < t$ , the detection is considered incorrect (false positive).

#### 3.3.2 Precision and Recall

Object detection frameworks do not use true negatives (TN), so metrics like the true positive rate, false positive rate, and Receiver Operating Characteristic (ROC) curves should be avoided [21].



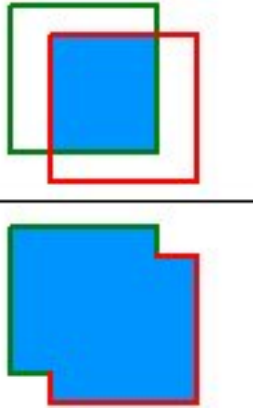
$$IOU = \frac{\text{area of overlap}}{\text{area of union}} = \frac{\text{area of overlap}}{\text{area of union}}$$


Figure 2: Intersection over union (IOU) metric [20].

Object detection methods are evaluated primarily based on precision ( $P$ ) and recall ( $R$ ). Instead, object detection methods are evaluated primarily based on precision ( $P$ ) and recall ( $R$ ).

Equation 2 defines precision as the accuracy of the model’s predictions. Precision measures the model’s ability to make accurate positive predictions. A high precision indicates that a model’s predictions are usually accurate.

$$P = \frac{TP}{TP + FP} = \frac{TP}{\text{all detections}} \quad (2)$$

Equation 3 defines recall as the model’s ability to predict all relevant objects, specifically GT bounding boxes. Recall measures a model’s accuracy in predicting relevant objects in an image, as opposed to those it misses. A high recall indicates that the model accurately captures the majority of relevant objects.

$$R = \frac{TP}{TP + FN} = \frac{TP}{\text{all ground truths}} \quad (3)$$

Precision and recall are critical metrics for object detection. A successful model should capture all ground truths ( $FN = 0$ ) and accurately predict all relevant objects ( $FP = 0$ ). A good model maintains high precision as recall increases. The precision-recall curve shows the relationship between precision and recall for different confidence levels in the model-generated bounding boxes. The area under the curve (AUC) represents high precision and requires a specific confidence threshold. A high AUC indicates good model performance across different confidence thresholds.

### 3.3.3 Average Precision

In practice, the precision  $\times$  recall plot can be jagged, making it difficult to accurately measure AUC. To avoid zigzag behavior, the precision  $\times$  recall curve is processed before estimating the AUC. The Average Precision (AP) metric evaluates the detection of various models and is commonly used

for comparison. The AUC of the precision-recall curve represents the model’s AP and summarizes its performance as a scalar value. There are two methods for doing so: 11-point interpolation and all-point interpolation.

The 11-point interpolation summarizes the precision  $\times$  recall curve by averaging maximum precision values at 11 evenly distributed recall levels  $[0, 0.1, 0.2, \dots, 1]$  (see Equation 4) [20].

$$\text{AP}_{11} = \frac{1}{11} \sum_{R \in \{0, 0.1, \dots, 0.9, 1\}} P_{\text{interp}}(R), \quad (4)$$

where,

$$P_{\text{interp}}(R) = \max_{\tilde{R}: \tilde{R} \geq R} P(\tilde{R}). \quad (5)$$

To smooth the curve, the AP is calculated using the maximum precision  $P_{\text{interp}}(R)$  with a recall value greater than  $R$ , rather than the observed recall level at each step.

By employing all recall points rather than just 11 fixed points, the all-point interpolation smoothes the precision-recall curve like that of the 11-point interpolation. Equation 6 illustrates how the all-point interpolation method [20] is used to calculate the AP, as seen in Equation 6.

$$\text{AP}_{\text{all}} = \sum_n (R_{n+1} - R_n) P_{\text{interp}}(R_{n+1}), \quad (6)$$

where,

$$P_{\text{interp}}(R_{n+1}) = \max_{\tilde{R}: \tilde{R} \geq R_{n+1}} P(\tilde{R}). \quad (7)$$

According to the definition given above, the AP is calculated by interpolating the maximum precision  $P_{\text{interp}}(R_{n+1})$  observed at each level rather than just a few points, whose recall value is greater than  $R_{n+1}$ .

### 3.3.4 Mean Average Precision

The mean AP (mAP) metric evaluates the accuracy of object detectors across all classes in a given database. Equation 8 calculates the mAP, which is the average AP across all classes [20].  $\text{AP}_i$  represents the AP in the  $i^{\text{th}}$  class, and  $N$  represents the total number of classes evaluated.

$$\text{mAP} = \frac{1}{N} \sum_{i=1}^N \text{AP}_i, \quad (8)$$

### 3.4 YOLO

In 2015, Joseph Redmon et al. [18] introduced the YOLO (You Only Look Once) object detection model, which relies on a single neural network for fast performance. YOLO has evolved into a cutting-edge model, including Ultralytics' YOLOv11 model [22], which includes detection, segmentation, tracking, and classification capabilities.

YOLO models are known for their fast, accurate, and comprehensive object detection approach. The YOLOv1 model revolutionized object detection by treating it as a single regression problem. The model uses a single neural network to process the entire image, resulting in faster and more accurate results than region proposal methods. YOLOv1 had 24 convolutional layers, followed by two fully connected layers. The input image was divided into a  $7 \times 7$  grid. Each grid cell predicted two bounding boxes and class probabilities, simplifying the object detection pipeline and increasing speed.

Figure 3 depicts a simplified output prediction of the YOLOv1 model using a  $3 \times 3$  grid with three classes and one class per grid cell. Each grid cell generates an 8-value vector ( $P_c, b_x, b_y, b_h, b_w, c_1, c_2, c_3$ ) containing confidence score, bounding box dimensions, and class probabilities. This simplified approach yields  $3 \times 3 \times 8$  output predictions for a single image [23].

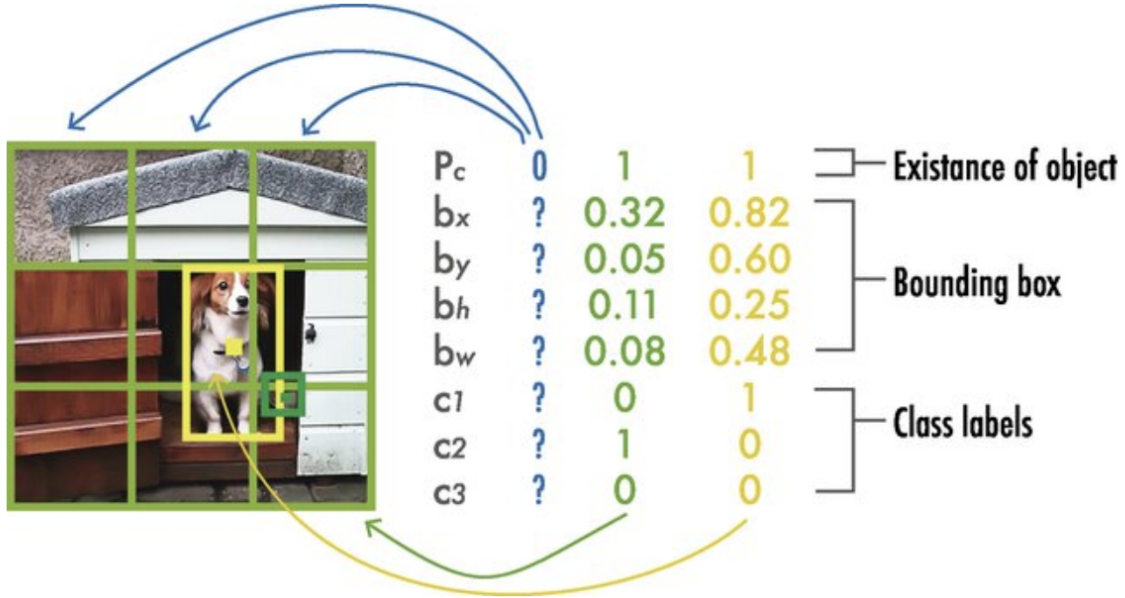


Figure 3: YOLOv1 output predictions visualized [23].

#### 3.4.1 YOLO Evolution

The YOLOv1 model has been enhanced over time to overcome limitations and improve performance. Newer YOLO models aim to improve accuracy while maintaining speed [23].

- **YOLOv2:** YOLOv2, which was released in 2016, sought to increase accuracy without sacrificing speed. High-resolution classifiers, anchor boxes to improve bounding box prediction, and batch normalization to lessen model overfitting were all introduced. The backbone of YOLOv2 was Darknet-19, which included five max-pooling layers after 19 convolutional layers. In addition to YOLOv2, a new model named YOLO9000 was unveiled. It made it possible to train the model with a mix of classification and detection data. Furthermore, over 9000 categories could be detected by the YOLO9000 model.
- **YOLOv3:** With the release of YOLOv3 in 2018, the deep backbone Darknet-53—which has 53 convolutional layers with residual connections—was built on the achievements of its predecessors. Additionally, by detecting three boxes at three different scales, YOLOv3 enhances multi-scale predictions, enabling the model to identify objects at various scales within the same image. This enhancement greatly increased the accuracy of small object detection by enabling more detailed boxes. YOLOv3 greatly increased accuracy while preserving the speed advantage of earlier iterations.
- **YOLOv4:** When YOLOv4 was released in 2020, it significantly improved the detection accuracy and feature extraction capabilities. Three components—the head, neck, and backbone—were used to describe object detection model architectures at the time. A modified Darknet-53 with Cross Stage Partial (CSP) connections and a Mish activation function served as the foundation for the YOLOv4 model. For improved performance in complex environments, YOLOv4 also includes methods like DropBlock regularization, Mosaic data augmentation, Spatial Pyramid Pooling (SPP), and CIoU (Complete Intersection over Union) loss.
- **YOLOv5:** YOLOv5, created by Ultralytics in 2020, gained widespread popularity despite not being formally included in Joseph Redmon’s original YOLO series because of its practical enhancements and easy-to-use implementation. Although it was created using PyTorch rather than Darknet, it includes many of the enhancements from YOLOv4. In order to strike a balance between speed and accuracy, YOLOv5 offered several model sizes (nano, small, medium, large, and extra large) and concentrated on streamlining the deployment process. In addition to being able to classify and segment instances, YOLOv5 offers a more user-friendly interface for training and deployment.
- **YOLOv6 & YOLOv7:** Although they were not created by Ultralytics, these versions, which were released in 2022, featured additional architectural improvements. EfficientRep, a new backbone based on RepVGG that uses higher parallelism to increase processing speed during inference, was introduced by YOLOv6. With its emphasis on extended efficient layer aggregation (E-ELAN), YOLOv7 produced the fastest and most accurate object detection model available at the time by enabling more effective training.
- **YOLOv8:** YOLOv8, which Ultralytics released in 2023, supported a wide variety of vision AI tasks and included new features and enhancements for improved performance, flexibility, and efficiency. In contrast to its predecessors, YOLOv8 boasts an anchor-free split Ultralytics head, cutting-edge neck and backbone architectures, and an optimized accuracy-speed tradeoff, which makes it perfect for a variety of applications.

- **YOLOv9 & YOLOv10:** YOLOv9 and YOLOv10, which were released in late 2023 and 2024, respectively, brought new innovations that were centered on transformer-based enhancements and efficiency. YOLOv9 presents novel techniques such as the Generalized Efficient Layer Aggregation Network (GELAN) and Programmable Gradient Information (PGI). YOLOv10, developed by Tsinghua University researchers using the Ultralytics Python package, offers improvements in real-time object detection by introducing an End-to-End head that does away with the need for Non-Maximum Suppression (NMS). Both versions showed improved generalization across various datasets and pushed for additional accuracy gains on metrics like COCO.
- **YOLOv11:** As of writing this thesis, the most recent version of the YOLO family is YOLOv11, released by Ultralytics in 2025 [22]. For more accurate object detection and complex task performance, it uses an enhanced backbone and neck architecture that improves feature extraction capabilities. In order to deliver faster processing speeds and preserve the best possible balance between accuracy and performance, it introduces improved architectural designs and training pipelines. By using 22% fewer parameters than YOLOv8m and achieving a higher mean Average Precision (mAP) on the COCO dataset, YOLOv11m is computationally efficient without sacrificing accuracy, thanks to improvements in model design. For optimal flexibility, YOLOv11 can be easily implemented in a variety of settings, such as edge devices, cloud platforms, and systems that support NVIDIA GPUs. YOLOv11 is made to handle a wide range of computer vision problems, including object detection, instance segmentation, image classification, pose estimation, and oriented object detection (OBB).
- **YOLOv12:** Additionally, in 2025, YOLOv12 was released, introducing attention-based mechanisms like Residual Efficient Layer Aggregation Networks (R-ELAN) and Area Attention. These techniques were included to preserve real-time performance while increasing detection accuracy. The instance segmentation feature is still unstable, especially for lightweight variants like YOLO12s-seg, despite the architecture supporting tasks like classification, pose estimation, and segmentation. Users have reported training collapse and inconsistency. Therefore, until more improvements are made, YOLOv12 is not yet advised for segmentation tasks. As a result, this thesis will not take into account this Yolo version.

### 3.4.2 YOLOv11 Architecture

Building on the improvements made in previous YOLO versions, such as YOLOv8, YOLOv9, and YOLOv10, the architecture of YOLOv11 is intended to maximize speed and accuracy. The C3K2 blocks, SPFF (Spatial Pyramid Pooling Fast), and sophisticated attention mechanisms like the C2PSA block are the primary architectural innovations in YOLOv11, which improve its capacity to process spatial information while preserving high-speed inference [24].

The YOLO architecture is built around three core components. The backbone extracts features from raw image data using convolutional neural networks, resulting in multi-scale maps. The neck component serves as an intermediate processing stage, using specialized layers to aggregate and enhance feature representations at various scales. The head component predicts and generates outputs for object localization and classification using refined feature maps. YOLO11 builds on YOLOv8 by introducing architectural innovations and parameter optimizations, resulting in superior

detection performance.

The backbone is a key component of the YOLO architecture, extracting features from input images at different scales. Stacking convolutional layers and specialized blocks generates feature maps with varying resolutions.

YOLOv11 follows the same structure as its predecessors, using initial convolutional layers to downsample images. The feature extraction process relies on these layers, which gradually reduce spatial dimensions and increase the number of channels. YOLO11 introduces the C3k2 block, a significant improvement over the previous versions' C2f block [24]. The C3k2 block offers a more efficient implementation of the Cross-Stage Partial (CSP) Bottleneck. Unlike YOLOv8, it uses two smaller convolutions rather than a single large one. The "k2" in C3k2 refers to a smaller kernel size, allowing for faster processing while maintaining performance [24]. YOLO11 retains the Spatial Pyramid Pooling Fast (SPPF) block from previous versions, but adds a new Cross-Stage Partial with Spatial Attention (C2PSA) block [24].

The neck combines features on various scales and sends them to the head for prediction. The process involves upsampling and concatenating feature maps from various levels to capture multi-scale information effectively. YOLO11 makes a significant change to the neck by replacing the C2f block with the C3k2 block. The C3k2 block optimizes feature aggregation performance by increasing speed and efficiency. YOLO11's neck incorporates an improved block after upsampling and concatenation, leading to faster and more efficient performance [24].

The head of YOLOv11 is in charge of making the final predictions in terms of object detection and classification. It uses feature maps from the neck to generate bounding boxes and class labels for objects in the image. In the head section, YOLOv11 employs multiple C3k2 blocks to efficiently process and refine the feature maps. C3k2 blocks are distributed across multiple pathways in the brain, processing multi-scale features at various levels. The C3k2 block is flexible depending on the value of the c3k parameter. When c3k is false, the C3k2 module operates similarly to the C2f block, with a standard bottleneck structure. When c3k is set to true, the C3 module replaces the bottleneck structure, allowing for more detailed and complex feature extraction [24].

Following the C3k2 blocks, YOLOv11's head includes several CBS (Convolution-BatchNorm-Silu) [24] layers. These layers refine feature maps by extracting relevant features for accurate object detection, batch normalizing data flow, and using the Sigmoid Linear Unit (SiLU) activation function for non-linearity to enhance model performance. CBS blocks are essential for feature extraction and detection. They ensure that refined feature maps are passed to later layers for bounding box and classification predictions.

Conv2D layers are used at the end of each detection branch to condense features into bounding box coordinates and class predictions. The final Detect layer consolidates predictions, including:

- Coordinates of the bounding box for localizing objects in the picture.
- Objectness scores that show whether objects are present.



- Class scores, which are used to identify the detected object’s class.

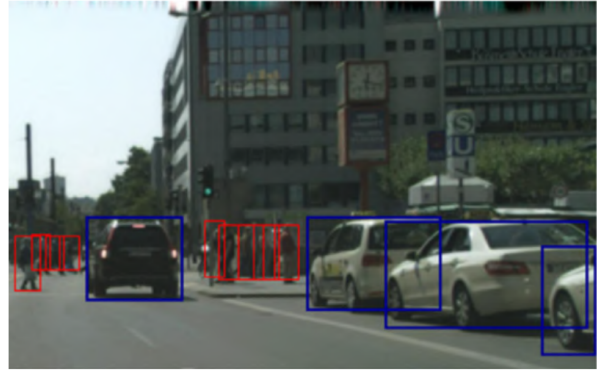
### 3.4.3 YOLOv11 Training, Segmentation Model, and Performance

A variety of hyperparameters and configurations are employed throughout the training process in the YOLO model training settings [22]. These settings affect the model’s performance, speed, and accuracy. Key training parameters include batch size, learning rate, momentum, and weight decay. The choice of optimizer, loss function, and training dataset composition can all have an impact on the training process. Careful tuning and experimentation with these settings are essential for achieving peak performance. Augmentation techniques are critical for improving the robustness and performance of YOLO models by introducing variability into the training data, allowing the model to generalize more effectively to new data.

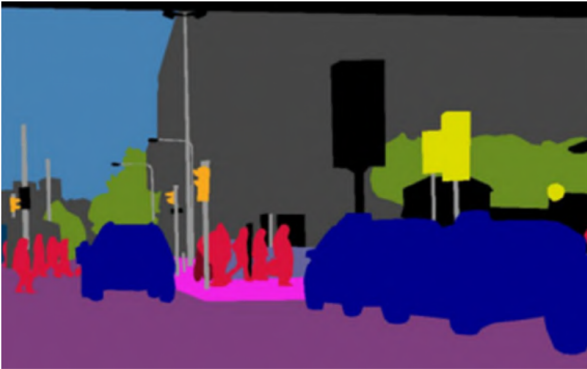
Semantic segmentation is a computer vision task that assigns class labels to each pixel in an input image. Semantic segmentation divides images into meaningful parts based on visible object shapes [25]. Figure 4c illustrates semantic segmentation, which provides precise boundary lines around an object in addition to its location and class. Figure 4d demonstrates instance segmentation, which identifies each object in an image, even if they are close together and have the same label.



(a) Original image



(b) Object detection



(c) Semantic segmentation



(d) Instance segmentation

Figure 4: Comparison of various computer vision tasks [25]

YOLOv11 also includes an instance segmentation variant, YOLOv11-seg, which is based on the core architecture of the YOLOv11 object detection model [22]. Both models use the same backbone, however, YOLOv11-seg adds segmentation-specific features such as a refined neck and dedicated segmentation heads that generate high-quality instance masks. These heads are designed to increase mask precision, especially for small and overlapping objects. Similar to its detection counterpart, YOLOv11-seg comes in a variety of model sizes, allowing users to strike a balance between computational efficiency and segmentation performance.

Ultralytics' latest YOLO models provide state-of-the-art (SOTA) performance across a variety of tasks, including detection, segmentation, pose estimation, tracking, and classification [22]. Figure 5 shows that the different YOLO models have varying mean Average Precision (mAP) scores on the COCO dataset [26], depending on their runtime performance.

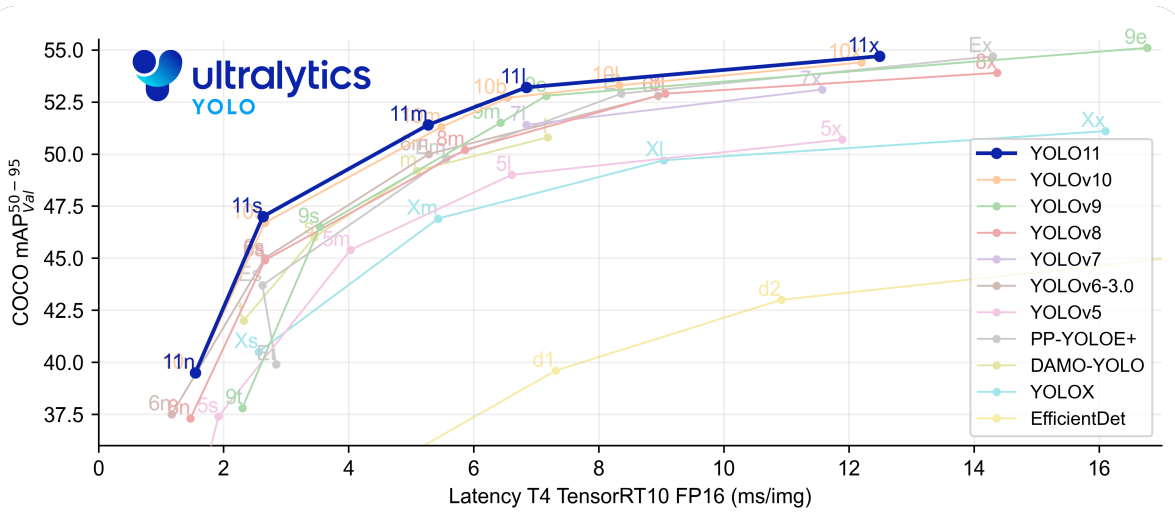


Figure 5: YOLO model comparisons [22].



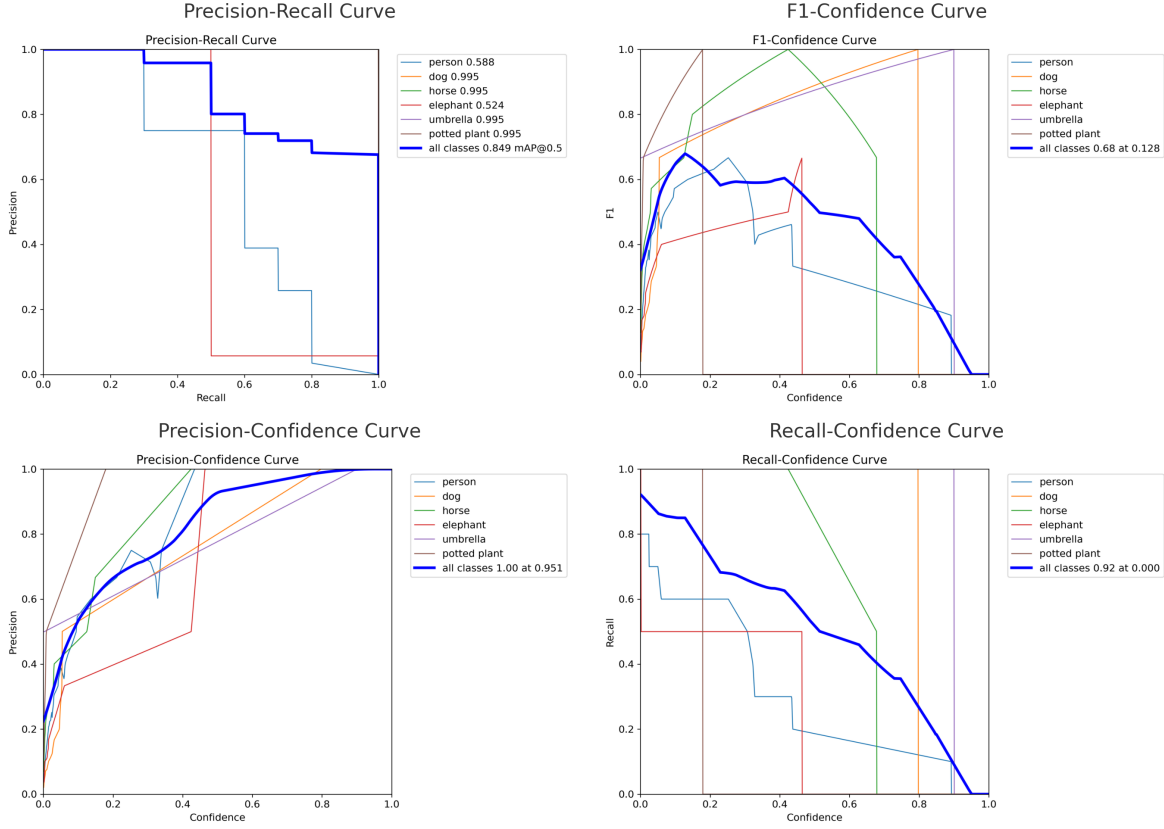


Figure 6: Evaluation metrics for the YOLOv11m-seg model on a subset of the COCO dataset consisting of 4 validation images and 17 object instances. The plots summarize detection performance across six object classes: *person*, *dog*, *horse*, *elephant*, *umbrella*, and *potted plant*. **Top-left:** Precision-Recall curve showing overall mAP@0.5 of 0.849, with high scores for most classes except for *person* and *elephant*, which exhibit weaker precision and recall. **Top-right:** F1-Confidence curve indicating that the optimal confidence threshold for balancing precision and recall is approximately 0.128, yielding a peak F1-score of 0.68. **Bottom-left:** Precision-Confidence curve demonstrating strong precision calibration, with perfect precision (1.00) achieved at high confidence levels. **Bottom-right:** Recall-Confidence curve revealing that recall is maximized at lower confidence values, gradually decreasing with increased thresholds. These metrics suggest that YOLOv11m-seg is highly effective on well-defined object classes such as *dog* and *potted plant*, but less reliable for complex or variable classes like *person*, highlighting the importance of confidence threshold tuning in real-world deployment scenarios.

The YOLOv11m-seg model’s evaluation curves on a limited subset of the COCO dataset provide important information about the model’s class-wise detection behavior and confidence calibration (see Figure 6). With a mAP@0.5 of 0.849, the Precision-Recall (PR) curve shows a strong overall detection capability. The *person* and *elephant* classes show steeper drop-offs, indicating less consistent localization or classification for these categories, whereas the majority of classes maintain high precision and recall throughout. The model appears to maximize recall in safety-critical applications by achieving the optimal trade-off between precision and recall at relatively low confidence levels, as indicated by the F1 vs. Confidence curve’s peak at a threshold of about 0.128. While the

Recall-Confidence curve demonstrates that recall is maximized at lower confidence levels, the Precision-Confidence curve demonstrates that confidence scores are well-calibrated, with perfect precision (1.00) at high thresholds (0.95). These patterns show that the model is very certain of its accurate predictions and cautious when unsure, which is a good quality for real-time detection systems.

Table 1: Evaluation results of YOLOv11m-seg on a subset of the COCO dataset (4 images, 17 instances).

Class	Precision	Recall	mAP@0.5	mAP@0.5:0.95
<b>person</b>	0.589	0.600	0.588	0.293
<b>dog</b>	0.550	1.000	0.995	0.796
<b>all classes</b>	0.574	0.850	0.849	0.652

The results in Table 1, which measures per-class detection performance, enhances the curve-based evaluation. The *dog* class, in particular, achieves almost perfect values: a recall of 1.00, mAP@0.5 of 0.995, and mAP@0.5:0.95 of 0.796. The model performs well across all metrics. Accordingly, YOLOv11m-seg appears to be particularly well-suited for the detection of distinct, non-overlapping objects with a high degree of visual consistency. The *person* class, on the other hand, records much lower scores—precision and recall both close to 0.59, and a sharp decline to 0.293 in mAP@0.5:0.95—indicating challenges with accurate localization and classification at more stringent IoU thresholds. The model’s resilience and adaptability are demonstrated by the aggregated metrics across all classes (Precision: 0.574, Recall: 0.850, mAP@0.5: 0.849, mAP@0.5:0.95: 0.652), which show that even with a small sample size, the model maintains high recall and strong average precision. The difference between the easy and difficult classes, however, highlights the necessity of class-specific augmentation or tuning techniques, particularly for occluded or visually complex categories like humans.

This thesis uses the YOLOv11 model because it strikes a good balance between detection accuracy and real-time performance, making it an appropriate foundation for the subsequent LiDAR-camera fusion stage.

### 3.5 Clustering

Clustering is used in Kieffer’s fusion pipeline [1] to isolate the most relevant point group from the filtered 3D LiDAR points within each segmented object mask, thereby reducing outliers and refining object localization. Clustering is an unsupervised learning technique that groups data points based on similarities, revealing natural patterns and structures. Unsupervised learning is a type of machine learning that analyzes unlabeled data for insights without prior training. The following steps are typically followed in clustering [27]:

1. Extract and select the most representative features from the original dataset.
2. Design the clustering algorithm based on the problem characteristics.

3. Evaluate the clustering results to determine the algorithm's effectiveness.
4. Give a practical interpretation of the clustering results.

Clustering algorithms rely heavily on distance and similarity. Distance metrics such as the *Minkowski distance* (see Equation 9), which includes both *Euclidean distance* (when  $p = 2$ ) and *Manhattan distance* (when  $p = 1$ ), can reveal patterns and relationships in numerical data.

$$dist(x_1, x_2) = \left( \sum_{j=1}^d |x_{1j} - x_{2j}|^p \right)^{\frac{1}{p}} \quad (9)$$

Where  $x_1$  and  $x_2$  are  $d$ -dimensional data points,  $x_{1j}$  and  $x_{2j}$  are the  $j$ -th components of  $x_1$  and  $x_2$ , and  $p$  determines the distance metric type (Euclidean or Manhattan).

Similarity metrics, such as Jaccard Similarity (defined in Equation 10), are better suited for categorical data as they assess the similarity of data points based on qualitative features.

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \quad (10)$$

where  $|X|$  denotes the number of elements in the set  $X$ .

These metrics influence cluster formation and interpretation across various data types, significantly impacting algorithm output [27].

Evaluation indicators evaluate clustering algorithms' effectiveness and validity, allowing for a better understanding and comparison of performance. There are two main types of evaluation indicators:

- **Internal Evaluation Indicators** analyze clustering data to assess cohesion and separation. Algorithm comparisons may not always be definitive.
- **External Evaluation Indicators** are considered the gold standard for comparing algorithms as they objectively measure clustering quality using known ground truth.

### 3.5.1 KMeans Clustering Algorithm

KMeans, also known as centroid-based clustering or partition-based clustering, is one of the most widely used clustering algorithms.

The KMeans algorithm requires an initial input of the number of clusters ( $k$ ). The primary steps of the KMeans algorithm are as follows:

1. **Initialization:** Choose  $k$  distinct centroids, also known as means, at random from the collection of data points.

2. **Assignment:** Use a distance metric, such as the Euclidean distance described earlier, to assign each data point to its nearest centroid.
3. **Update:** Determine the mean of each point inside each cluster and designate it as the cluster's new centroid.
4. **Repeat:** Until the centroids no longer change significantly or a maximum number of iterations is reached, steps two and three are repeated.

Similar to the majority of other partition-based clustering algorithms, KMeans offers the following benefits [27]:

- KMeans has a computational complexity of  $O(k * n * t)$ , where  $k$  is the number of clusters,  $n$  is the number of total data points, and  $t$  is the number of iterations, making it highly efficient.
- The implementation of KMeans is also relatively straightforward, and numerous libraries already have the algorithm built in to them.

However, the algorithm also has a number of limitations and disadvantages [27]:

- The specification demands that the number of clusters  $k$  is predetermined.
- Due to the initialization procedure, the clustering result is sensitive to the number and selection of clusters.
- The algorithm may converge on a local optimum rather than a global optimum, which is undesirable.
- Since it assumes that clusters are spherical and of comparable size, it is not appropriate for non-convex data and is somewhat sensitive to outliers.

### 3.5.2 DBSCAN Clustering Algorithm

An algorithm that falls under the Density-Based clustering category is called Density-based Spatial Clustering of Applications with Noise (DBSCAN). DBSCAN can find clusters of any shape and does not require a predetermined number of clusters, in contrast to centroid-based methods like KMeans [27].

DBSCAN uses two parameters to generate clusters:

- The maximum distance between two points for them to be regarded as belonging to the same neighborhood is defined by **eps** (epsilon), which also defines the radius of a neighborhood around a point. Different clusters may combine into one if the *eps* is too big. If it is too small, there may be fewer clusters because more points may be regarded as outliers.
- The smallest number of points in a neighborhood (within an *eps* radius) is defined by **min\_samples**. There may be fewer clusters overall if *min\_samples* is higher because denser regions are needed to form a cluster. There may be more clusters overall when a smaller value is assigned because fewer points are taken into account in the same neighborhood.

The DBSCAN algorithm uses three different kinds of data points to form clusters:

1. Points with at least *min\_samples* within their *eps* radius are known as **core points**.
2. Points that belong to a core point's neighborhood but lack sufficient points within their *eps* radius to qualify as core points themselves are known as **border points**.
3. Points that do not fit the definition of a core or border point are known as **noise points**.

The following are the DBSCAN algorithm's primary steps:

1. **Core Point Identification:** List all of the dataset's core points.
2. **Cluster Expansion:** To create or enlarge a cluster, recursively add points that are within an *eps* radius of a core point or its neighboring points.
3. **Noise Handling:** Designate as noise the remaining points that are inaccessible from any core point.

Similar benefits are provided by DBSCAN and other density-based clustering algorithms [27]:

- **Adaptability:** Able to recognize groups of any size or shape.
- **Robustness to Noise:** Capable of managing noise points and outliers efficiently without clumping them together.
- **Automatic Cluster Number Determination:** It is not necessary to predetermine the number of clusters.
- **Time Complexity:**  $O(n \log n)$ , where  $n$  is the total number of data points, is the time complexity, which is typically efficient for large datasets.

DBSCAN does, however, have certain restrictions and disadvantages [27]:

- **Parameter Sensitivity:** The *eps* and *min\_samples* parameters, which must be carefully selected depending on the dataset characteristics, have a significant impact on DBSCAN.
- **Challenge with Varying Density:** DBSCAN has trouble clustering data with different densities, which leads to subpar clustering outcomes.
- **Memory Usage:** For large datasets, achieving  $O(n \log n)$  frequently requires significant memory consumption.

### 3.6 Sensor Calibration

LiDAR and camera sensors are calibrated to get the transformation from one coordinate system to another so that each point can be represented in either coordinate system. Therefore, it is possible to translate a 3D LiDAR point into a 2D image coordinate system and, in turn, obtain the 3D coordinates of an image pixel in the LiDAR coordinate system. When combining LiDAR and camera data for 3D object detection or tracking, this procedure, known as extrinsic calibration of the sensors, is essential to getting good results.

In addition to extrinsic calibration, the camera sensor's intrinsic calibration is crucial. It alludes to the internal characteristics of the camera, including the distortion coefficients, focal length, and optical center. As a result, the intrinsic camera calibration only needs to be done once and does not require the LiDAR sensor. A 3D point in the camera's field of view is projected onto the 2D image plane according to the intrinsic parameters.

Pixel coordinates in the image plane ( $x_{min}$ ,  $y_{min}$ ,  $x_{max}$ ,  $y_{max}$ ) define a box in 2D image object detection. Outputting a 3D bounding box in the real world (typically in the LiDAR or camera coordinate system) with its center location ( $x, y, z$ ), dimensions (width, height, length), and orientation (yaw angle) is the aim of 3D object detection [28].

#### 3.6.1 LiDAR-Camera Transformation

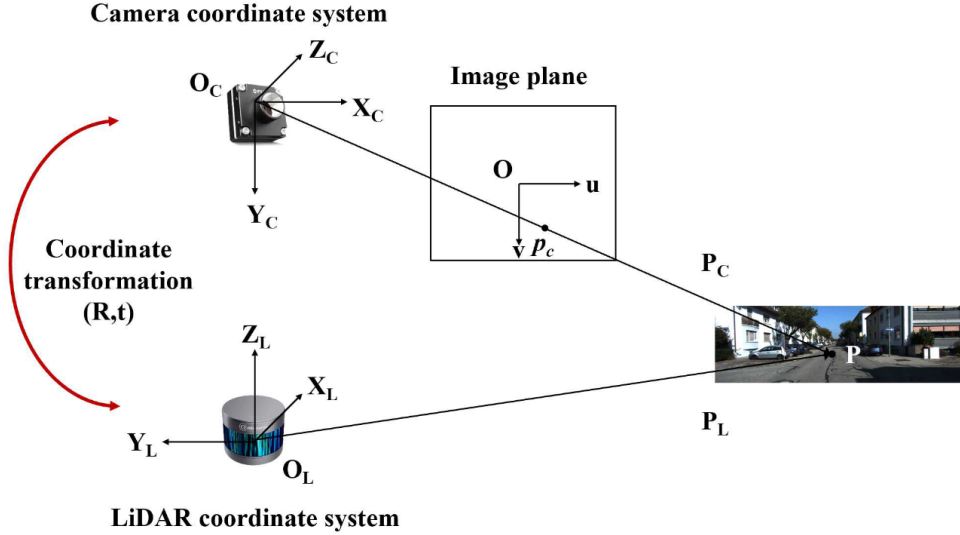


Figure 7: LiDAR and camera sensor transformation [29].

A setup with LiDAR and camera sensors that record a scene is shown in Figure 7. In the camera coordinate system, a point  $P$  in the 3D scene is represented by  $P_C$  in the camera coordinate system and in the LiDAR coordinate system by  $P_L$ . The extrinsic parameters  $\mathbf{R}$  and  $\mathbf{t}$  are used to perform the coordinate system transformation. Both coordinate systems' origins are represented by

the letters  $O_C$  and  $O_L$ , which stand for the camera ( $C$ ) and LiDAR ( $L$ ) respectively. The three coordinates of a point-  $X$ ,  $Y$ , and  $Z$ - are used to represent it.

A point's position is denoted as

$$P_C = [X_C \ Y_C \ Z_C]^T$$

in camera coordinates and

$$P_L = [X_L \ Y_L \ Z_L]^T$$

in LiDAR coordinates. Additionally, point  $P$  projected onto the image plane as

$$p_C = [u \ v]^T.$$

The projection between the 3D points  $P_C$  in camera coordinates and the 2D point  $p_C$  on the image plane, or the **intrinsic** parameters [29], is defined by Equation 11 :

$$Z_C \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & s & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_C \\ Y_C \\ Z_C \end{bmatrix} = \mathbf{K}_C \begin{bmatrix} X_C \\ Y_C \\ Z_C \end{bmatrix} \quad (11)$$

$Z_C$  stands for the depth scale factor in Equation 11,  $f_x$  and  $f_y$  for the focal length in pixels on the x- and y-axes, respectively,  $(u_0, v_0)$  for the camera's optical center, and  $s$  for the skew coefficient, which is non-zero if the image axes are not perpendicular. Therefore, the projection matrix from 3D camera coordinates to 2D image coordinates on the image plane is the matrix  $\mathbf{K}_C$ .

Equation 12 describes the **extrinsic** parameters [29], or the transformation between the point  $P_L$  in LiDAR coordinates and the point  $P_C$  in camera coordinates:

$$\begin{bmatrix} X_C \\ Y_C \\ Z_C \end{bmatrix} = \mathbf{R} \begin{bmatrix} X_L \\ Y_L \\ Z_L \end{bmatrix} + \mathbf{t} = [\mathbf{R} \ \mathbf{t}] \begin{bmatrix} X_L \\ Y_L \\ Z_L \\ 1 \end{bmatrix} \quad (12)$$

$\mathbf{R}$  is the rotation matrix and  $\mathbf{t}$  is the translation vector between the LiDAR and camera coordinate systems in Equation 12. Let  $\mathbf{T} = [\mathbf{R} \ \mathbf{t}]$  stand for the total extrinsic parameters, which is the sum of these values.

Equations 11 and 12 [29] can be combined to determine the overall transformation between a 3D LiDAR point  $P_L$  and the corresponding 2D image plane point  $p_C$ :

$$Z_C \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \mathbf{K}_C [\mathbf{R} \ \mathbf{t}] \begin{bmatrix} X_L \\ Y_L \\ Z_L \\ 1 \end{bmatrix} \quad (13)$$

Equation 13 uses intrinsic and extrinsic parameters to define the mapping of 3D points from the LiDAR coordinate system to 2D points on the image plane (pixel coordinates) of the camera and the other way around.



## 4 KITTI Dataset

This project tested and developed the model using the KITTI 3D Object Detection dataset [11]. Despite being over a decade old, the KITTI dataset remains a popular benchmark for comparing various 3D object detection models. Through the use of transformer-based architectures and sophisticated fusion techniques, recent state-of-the-art models on the KITTI dataset, including VirConv-S [10], ViKIENet-R [12], and UPIDet [13], have achieved over 85% AP for the car class on the moderate setting. With techniques like UPIDet and VPFNet also providing competitive performance for cyclists and pedestrians, these models exhibit a strong balance between accuracy and efficiency.

### 4.1 Dataset Presentation

The dataset includes 7481 training images and 7518 test images, along with point clouds and calibration files. The training data includes labels for each image, created by manually labeling objects in a 3D point cloud and projecting them back into the image [11].

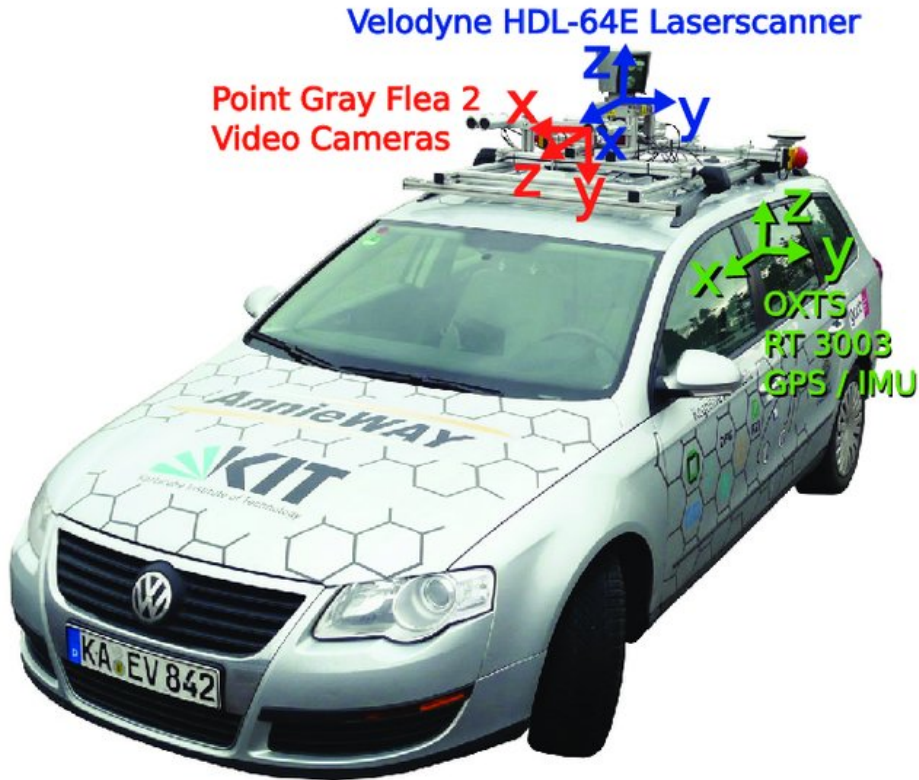


Figure 8: KITTI car sensor setup, the Volkswagen Passat station wagon features four video cameras (two color and two grayscale), a rotating 3D laser scanner, and a GPS/IMU inertial navigation system. [30].

Figure 8 depicts the car’s sensor setup. The equipment includes two high-resolution color and



grayscale video cameras, a Velodyne laser scanner (LiDAR), and a GPS system. The cameras and LiDAR capture 10 frames per second (FPS). Calibration files for transforming coordinate systems between cameras, LiDAR, and GPS are also provided. This project relies solely on data from the center color camera *Cam 2* and the LiDAR scanner (Velodyne laser scanner) mounted in the car.

## 4.2 Dataset Evaluation

The KITTI dataset offers a development kit [30] with tools and explanations for evaluating the model’s 3D object detection performance on the dataset. The model must save each processed frame’s output in a text file with the following information for each detected object:

- **Object class:** specifies the type of the detected object. The evaluation script only considers the classes *Car*, *Pedestrian*, and *Cyclist*. Other classes are marked as *DontCare* and ignored during valuation.
- **2D Bounding Box:** includes the pixel coordinates of the top-left and bottom-right corners projected onto the image from the 3D bounding box.
- **3D Bounding Box:** includes the 3D bounding box dimensions (height, width, and length in meters), and its ground center.
- **Confidence Score:** is a floating number that indicates the likelihood of detecting an object belonging to a specific class (higher scores are better).

The object class and confidence score are derived from the YOLOv11-seg model output. The object class is only refactored to match the expected string by the KITTI evaluation script. The model’s fusion process generates bounding box information.

The KITTI dataset includes three difficulty levels for their 3D object detection benchmark. Each object in the dataset is assigned a difficulty level based on specific criteria [11]. Table 2 summarizes the criteria used to categorize objects into difficulty levels. The minimum bounding box height measures an object’s projected 2D bounding box, indicating its distance. The maximum occlusion level indicates whether an object is fully visible in an image or if it is obscured by other objects or plants. The final criterion is the maximum truncation, which determines how much an object is cut off by the image boundaries.

Difficulty	Min. bounding box height	Max. occlusion level	Max. truncation
Easy	40 Px	Fully visible	15%
Moderate	25 Px	Partly occluded	30%
Hard	25 Px	Difficult to see	50%

Table 2: Difficulty levels based on bounding box height, occlusion, and truncation.

Each of the three object classes used in the evaluation process has three different difficulty levels. The KITTI dataset evaluates model detection performance using Average Precision (AP), resulting

in a total of 9 different AP values.

## 5 Methodology

This methodology is based on Kieffer’s (2024) [1] open-source YOLO-LiDAR-Fusion framework, which served as the foundation for this thesis. While the original repository contains a basic fusion pipeline that combines LiDAR and RGB data with YOLO-based object detection, this work includes several enhancements to improve detection accuracy and efficiency. Notably, the YOLOv8 model used in the baseline was replaced with YOLOv11m-seg, the most recent segmentation-capable version as of writing. Preprocessing steps have also been streamlined to make it easier to understand the steps involved in the fusion process, such as erosion-based segmentation mask refinement, depth-based filtering of LiDAR points, and Principal Component Analysis (PCA) bounding box alignment. Furthermore, a broader grid search was used to optimize erosion and depth parameters for each object class. The thesis provides a direct comparison with Kieffer’s work to better understand the architectural trade-offs between the YOLO models. It also offers additional evaluations of various YOLO versions (v8m, v9e, v11m) to directly compare the YOLO evolution in relation to Kieffer’s fusion strategy and provide a comparison of the two YOLOv8 models. Lastly, the YOLOv11 model sizes (from Nano to Extra Large) were also evaluated to test the effect of scaling the size of the model on the fusion pipeline.

### 5.1 Baseline Model

The baseline model [1] uses a YOLOv8-based fusion pipeline to detect 3D objects by combining RGB images and LiDAR data. The complete model pipeline can be seen in Figure 9. The process starts with YOLOv8-seg, which uses instance segmentation on the input RGB image to extract object-specific masks. These masks provide pixel-level precision, which is then used to match detected objects with 3D LiDAR points. The point cloud is filtered to only include points in the camera’s field of view. The remaining 3D points are projected onto the 2D image plane using the calibration matrix. Points within a segmentation mask are considered part of the detected object. A 3D bounding box is created for each object using a simple axis-aligned method that takes into account the min and max coordinates of the associated points. The final detection output consists of bounding boxes and class labels, which are assessed using Intersection over Union (IoU) and inference time.

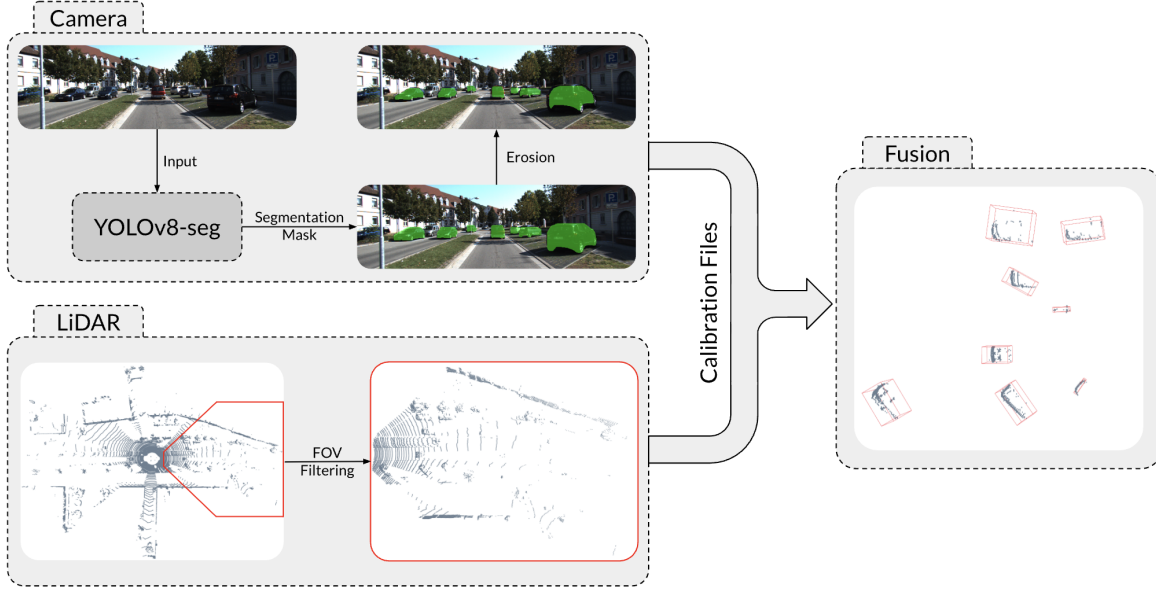


Figure 9: Baseline model pipeline. [1].

The baseline model pipeline (see Figure 9) takes image, point cloud, and LiDAR calibration files as input and returns IoU scores, bounding boxes, and class labels for detected objects.

Figure 9 shows that the image is first fed into the YOLO model, which extracts the segmentation mask, label, confidence score, and tracking ID for each detected object. The segmentation masks are then eroded to enhance fusion accuracy. The LiDAR point cloud for the image is filtered to include only points within the camera's Field-of-View (FOV). The fusion step involves calibrating the inputs from the two streams in order to align the data, and then the pipeline proceeds by projecting the filtered 3D points onto the image plane using the transformation matrix obtained from calibration files. The segmentation mask's projected points are identified and used to generate 3D bounding boxes. Principal Component Analysis (PCA) with  $z$ -axis rotation is applied to attempt to improve spatial alignment by reducing misalignment of the real-world objects. Finally, the model is evaluated based on the IoU score for each image, along with its inference time.

**Step 1:** Original image from the KITTI dataset that is fed into the suggested model.



**Step 2:** Retrieve each object's segmentation masks from the YOLOv11 model. Each segmentation mask should be eroded according to an erosion factor that affects how much the mask shrinks.



**Step 3:** Each 3D LiDAR point whose projection is inside the eroded segmentation mask should be fused. Use the depth-based filtering that makes use of DBSCAN to filter the fused 3D LiDAR points.



**Step 4:** Using PCA with z-axis rotation, create 3D bounding boxes that enclose the filtered 3D LiDAR points.



**Step 5:** The YOLOv11 model detects and localizes the objects with a bounding box, predicted class label, and IoU score.



Figure 10: An example of every fusion step in the model adapted for YOLOv11.

Figure 10 shows the full fusion process along with step-by-step visualizations. The original image is shown first in the sequence, and then segmentation masks are projected and eroded. The 3D points are then projected onto the image after being fused. These points are filtered, after which the bounding boxes are made and superimposed using Principal Component Analysis (PCA). Lastly, the model detects and localizes the object in the image, displaying a bounding box, class label prediction, and IoU score.

## 5.2 Our Work

This thesis compares Kieffer’s (2024) [1] fusion pipeline using the YOLOv8m-seg model, by using his pipeline with the YOLOv11m-seg model and streamlining the code.

In comparison to this baseline, the approach used in the current thesis builds on and streamlines the baseline’s code. First, the YOLO model was upgraded from YOLOv8 to YOLOv11m-seg, which has a more advanced architecture and higher segmentation accuracy. Second, the model’s segmentation masks erosion and depth filtering, are mathematically the same, however, the process is streamlined to reduce unnecessary function calls and adds more error checks. This work uses Kieffer’s [1] Principal Component Analysis (PCA) with z-axis rotation to generate 3D bounding boxes that are aligned with the object’s dominant orientation. This enables more accurate spatial localization, particularly for elongated or rotated objects. This work expands on the baseline by applying a broader grid search to optimize the erosion and depth filtering parameters for each object class. Kieffer’s work explores erosion and depth factor values between 10-30 and 5-20, respectively. This work explores erosion and depth factor values, both ranging from 5-70. Another significant advancement is the systematic comparison of various YOLO variants (v8, v9, and v11) and model sizes for the YOLOv11 model (from Nano to Extra Large) to investigate the trade-offs between accuracy and inference time.

His experiments are conducted on a high-end desktop equipped with an Intel Core i9-13900K CPU (24 cores, up to 5.5 GHz), an NVIDIA GeForce RTX 4090 GPU with 24GiB of GDDR6X VRAM, and 98GiB RAM. In contrast, our experiments are carried out on an Apple MacBook Pro with the M4 Max chip. The M4 Max has a 12-core CPU (eight performance cores and four efficiency cores), a 40-core integrated GPU, and a unified memory architecture featuring high-bandwidth, low-latency LPDDR5X memory.

Kieffer’s YOLOv8m-seg configuration excluded cyclists due to consistent misclassifications and low IoU scores, resulting from the YOLOv8m-seg model’s inability to differentiate between cyclists and stationary bicycles [1]. Despite similar limitations, we decided to keep the cyclist class in our evaluation. The significance of cyclists as vulnerable road users in urban settings, where precise detection is essential for safe autonomous navigation, served as the driving force behind this decision. Even though the cyclist class was not specifically used to fine-tune our YOLOv11m-seg model, its inclusion in the evaluation enables us to evaluate the model’s generalization abilities and identify performance gaps in underrepresented or unsupported classes. Furthermore, by presenting results for cyclists, we provide a more comprehensive understanding of the model’s advantages and disadvantages, laying a stronger foundation for future research that aims to enhance detection for all relevant traffic participants.

## 5.3 Preprocessing

The model begins by preparing the LiDAR and camera data that will be used as input. The image is fed directly into the YOLOv11 segmentation model without any preprocessing. The point cloud is filtered by a function that keeps only the points within the image’s field of view and discards the rest, such as those behind the car. A calibration function that makes use of the calibration matrices



between the LiDAR and camera sensors first defines the image boundaries before converting the 3D LiDAR points into 2D image coordinates.

To remove noise and irrelevant data, a boolean mask is used to filter 2D points that are within image boundaries and have a minimum clip distance to the sensor. The function returns both filtered 2D points and corresponding 3D points. Using the KITTI dataset [11], this preprocessing step reduces the point cloud size from 122000 to 18500 points, improving inference speed by approximately 85%.

The YOLOv11-seg model can produce inaccurate segmentation masks, leading to 3D background points being mistaken for object points. Some cars’ masks are too large, revealing background information like the road or walls behind them. Erosion shrinks the mask while maintaining its shape, removing background points that impede the fusion process. The visible eroded part of each detected object is discarded, leaving only the reduced mask as the final segmentation mask. Segmentation masks typically contain enough points to accurately predict 3D bounding boxes even after erosion and fusion.

The model calculates the area of the segmentation mask (polygon) before performing erosion. The function requires an *erosion\_factor* as input to determine the amount of erosion on a polygon. A higher erosion factor preserves more of the segmentation mask’s original area, resulting in less erosion. The parameter is scaled based on the polygon’s area to ensure proportional erosion.

Equation 14 describes how the scaled erosion factor is calculated. Smaller polygons are less eroded than larger ones due to a smaller scale factor. A scaled factor of 0 indicates that the polygon has not been eroded. However, this implies that the polygon area is always zero, which is not the case.

$$scaled\_erosion\_factor = \frac{\sqrt{polygon\_area}}{erosion\_factor} \quad (14)$$

After that, a binary mask is made, with black pixels filling the remainder of the image and white pixels filling the polygon. The scaled erosion factor that was previously calculated is then used to define the erosion kernel size. The erosion kernel is a small matrix that shrinks regions in a binary image by removing pixel edges. Erosion reduces the white area of the mask (polygon) based on the kernel, revealing the contours of the new polygon. Contours are curves that connect continuous points on the boundary of a shape with the same color, in this example, white. To preserve the most significant part of the original polygon during erosion, the largest contour is chosen based on its area.

After fusing 3D points with an eroded segmentation mask, the point cloud still contains noise. Objects’ 3D points are identified, but some may not belong due to reflection or obstacles. The LiDAR laser may pass through or be reflected by car windows, resulting in points that are not on the car’s surface. These points should not be considered for 3D bounding box creation.

The filtering process employs the *DBSCAN* clustering algorithm to group points based on their depth values and select the cluster with the most points. The DBSCAN algorithm requires two parameters, *eps* and *min\_samples*. The function begins by calculating *min\_samples*, which is either

1% of the detected object’s points or 5, whichever is greater.

Each point’s Euclidean distance from the LiDAR scanner in the  $xy$ -plane—apart from the  $z$ -coordinate—is used to calculate depth values. This method collects depth data that can be used to eliminate background and noise points, which are typically located farther away. When the objects were far to either side of the LiDAR scanner, this method had limitations because initially, only the  $x$ -coordinate (which points to the front) was taken into account.

The DBSCAN algorithm is then applied to the depth values by passing the calculated *min\_samples* value and a predefined *eps* value as a parameter to the filtering function. Only the cluster with the most points is chosen after the resultant clusters are arranged in descending order according to their point totals.

Noise points labeled  $-1$  by the DBSCAN algorithm are removed from the largest cluster. The depth values of the largest remaining cluster are analyzed to determine its minimum and maximum depths. The depth values are adjusted using a *depth\_factor* parameter passed to the function. The *depth\_factor* control adjusts the depth range aggressively. A larger *depth\_factor* leads to a wider depth range, resulting in less aggressive filtering and more points retained. A smaller *depth\_factor* tightens the depth range, resulting in more aggressive filtering and removal of points outside the range.

A depth mask is then made using the modified depth range. Only the points within the designated depth range are retained thanks to this mask, which filters the points in the chosen cluster according to their depth values.

## 5.4 Calibration

Calibration of the LiDAR and camera is necessary for accurate model performance. The model employs a calibration object that includes all required calibration matrices (transformation, rotation, and projection). The *LiDAR2Camera* class uses a utility function from the KITTI development kit [30] to parse calibration files and create a dictionary.

The calibration matrices are utilized by a function that converts 3D points from the LiDAR coordinate system to 2D points in the image frame. The function uses the *LiDAR2Camera* object and 3D point cloud as input and returns 2D pixel coordinates in the image frame. The *LiDAR2Camera* class and function were implemented from the YOLO-LiDAR Fusion repository on GitHub [1].

The rotation from the reference camera coordinate system to the camera coordinate system that is used (*Cam 2*) is represented by the  $3 \times 3$  matrix  $R$  in Equation ???. The KITTI dataset’s calibration file contains this rotation matrix.

Equation ??? illustrates how to transform the matrix  $R$  into a  $4 \times 4$  homogeneous transformation matrix  $R_{\text{homo}}$  by first adding a row of zeros  $[0 \ 0 \ 0]$  and then adding the column  $[0 \ 0 \ 0 \ 1]^T$ .

Since a homogeneous transformation matrix is uniformly sized and thus easily combines with other homogeneous matrices, it makes it easier to compute, represent, and concatenate multiple transformations. Equation ?? displays the  $3 \times 4$  matrix  $P$ , which is the projection of the used camera’s 3D camera coordinates to 2D image coordinates. The KITTI dataset’s calibration files contain this projection matrix. The matrix  $K_C$  in Equation 13 is represented by the left  $3 \times 3$  part of the matrix  $P$ . The translation vector (extrinsic parameters) of the used camera is included in the right  $3 \times 1$  part of the matrix  $P$ .

The  $3 \times 4$  matrix  $PR$ , which is displayed in Equation ??, is then obtained by combining the projection matrix  $P$  with the homogeneous transformation matrix  $R_{\text{homo}}$  by a dot product.

To project 3D LiDAR points onto the reference camera’s 2D image plane, several types of transformations are performed using the calibration matrices included with the dataset. To convert LiDAR coordinates to camera coordinates, we apply the transformation matrix  $V2C$  to encode rotation and translation between the two. This  $3 \times 4$  matrix is extended to a  $4 \times 4$  homogeneous matrix  $V2C_{\text{homo}}$  by adding an extra row, allowing for seamless matrix manipulation with homogeneous coordinates.

To project the transformed 3D points into the image plane, we multiply the projection matrix  $PR$ , which encodes the camera’s intrinsic parameters by  $V2C_{\text{homo}}$  from the final transformation matrix  $T$ . This matrix enables direct mapping from LiDAR coordinates to image frames.

Each 3D LiDAR point  $[x \ y \ z]$ , is first converted to homogeneous coordinates  $[x \ y \ z \ 1]^T$  and then transformed by  $T$  into a new vector  $[u \ v \ w]^T$ . To perform perspective division, we normalize the coordinates with  $w$ , resulting in the 2D pixel coordinates  $[x' \ y']$ , as seen in Equation 15. This process allows for precise spatial alignment of 3D LiDAR data and 2D image content.

$$x' = \frac{u}{w}, \quad y' = \frac{v}{w} \quad (15)$$

## 5.5 Fusion

The fusion function accepts both a 3D point cloud and its corresponding 2D points obtained through filtering and projection. The segmentation mask from the YOLOv11-seg model is required, as well as the original image matching the point cloud. For each detected object in the image, the fusion function returns the 3D coordinates of its corners within the 3D bounding box.

The first step in fusion is to erode the segmentation mask and determine if it is too small for further processing. If this occurs, the function returns, and the current object is not fused with any points.

The main fusion process involves identifying 3D LiDAR points that are compromised due to eroded segmentation. To achieve this, a single-channel grayscale mask of the same size as the image is created, containing only black pixels, resulting in a 2D array with zero values. The eroded segmentation polygon is drawn on top of the grayscale image with an intensity of 1, giving every



pixel in its region a value of 1 instead of 0 in the image’s 2D array. This effectively indicates the region of the segmentation polygon in the image.

The 2D coordinates received as input are converted from floating point to integer values for further processing. The integer coordinates are used to determine which points correspond to the mask’s filled polygon. This check creates a boolean array with truth values indicating whether a pixel is inside the polygon. To select 3D points within a polygon, use a boolean array containing point indices. Index the array to select the points. This indexing only works when the arrays of 2D and 3D points have the same length and order. For example, a 2D point at index  $i$  in the 2D points array corresponds to a 3D point at index  $i$  in the 3D points array.

After fusion, the 3D points of the processed object are stored in an array. Some points should not be included in the 3D bounding box because they are not part of the object. The points are filtered using DBSCAN’s depth filtering approach explained earlier in section 3.5.2. After filtering, we determine if there are enough points to create an accurate 3D bounding box. This applies to at least four remaining points that are not co-planar, meaning they do not share the same z-coordinates.

## 5.6 3D Bounding Boxes

The fusion process is complete once all 3D points of a detected object are identified and filtered. The model still needs to create 3D bounding boxes, which is the final step. The model distinguishes between *person* and other classes, such as *car*, *motorcycle*, *bicycle*, *truck*, *bus*, and *train*, when generating bounding boxes.

The 3D bounding box for the person class is computed using *Principal Component Analysis* (PCA) without additional processing. PCA is a statistical technique that identifies the smallest bounding box for a set of 3D points. This technique creates oriented bounding boxes with minimal volume while enclosing all 3D points. PCA-generated 3D bounding boxes are more accurate and better represent the shape of the detected object compared to those created without it.

The model utilizes the *Open3D* library [31] to generate 3D bounding boxes. First, a point cloud is generated from the array of 3D points provided as input. This array is the result of the fusion process following depth filtering. The built-in Open3D function *get\_oriented\_bounding\_box()* creates the oriented bounding box with PCA. The eight corners of the bounding box are extracted and converted into 2D points. Converting to 2D points allows for the visualization of edges over images. The 2D points are only used for visualization, while the 3D coordinates of the bounding box are used to calculate parameters like its center and dimensions.

The model uses a similar approach to create bounding boxes for classes other than *person*, but discards the rotation around the z-axis. To create a 3D point cloud and oriented bounding box, use the built-in *Open3D* function. The initial steps remain unchanged. The next step involves extracting the rotation matrix from the oriented bounding box, which describes its orientation along three axes. The matrix is modified to keep only the x- and y-axis rotations while discarding the z-axis rotation. To create a new oriented bounding box, use the center, dimensions, and

adjusted rotation matrix from the previous one. Following this step, the function is similar to creating a PCA bounding box. The eight corners are extracted, converted to 2D coordinates, and returned.

## 5.7 Model Configuration

The following sections, including parameter configuration, model evolution, and final results, used the medium size of the YOLOv11 model, unless otherwise specified.

The YOLOv11m-seg model was pre-trained on the *Microsoft Common Objects in Context* (COCO) dataset [26], which contains 328000 labelled images for instance segmentation across 91 categories. The YOLOv11-seg model does not require retraining as the dataset already includes the most important categories for object detection in autonomous vehicles, such as cars, pedestrians, bicycles, and trucks. Our YOLOv11m-seg model was not optimized on the "cyclist" class, despite the fact that the KITTI dataset contains explicit annotations for this class. Our experiments' model was pretrained on the COCO dataset [26], which lacks a specific "cyclist" category. No further training was done to match the class mappings to KITTI's label taxonomy. Any detections in this category are therefore based on general features learned for related classes like "person" or "bicycle," as the model lacks specialized supervision for identifying cyclists. In spite of this drawback, we decided to keep the cyclist class in the evaluation process to see how well the model applies to this class in the absence of focused training.

Kieffer's [1] fusion pipeline has two runtime parameters that affect erosion and filtering processes. These two parameters can significantly impact the model's performance.

- The **erosion factor** affects the segmentation mask erosion process. The erosion rate of a mask determines the number of points considered during the fusion process.
- **depth factor** determines the aggressiveness of 3D point filtering. The range of distances between the LiDAR scanner and a 3D point determines whether the point is part of the detected object. The algorithm adjusts maximum and minimum distances to determine if a point belongs in the object cluster or should be excluded. The depth factor determines the filtering process's level of aggressiveness by adjusting its range.

Kieffer's YOLO-LiDAR-Fusion model [1] used an erosion factor of 25 and a depth factor of 20, which were set to serve as the starting point for further parameter tuning. These values were selected based on preliminary observations and because they performed well across different object classes.

The 3D Intersection-over-Union (IoU) between the predicted bounding boxes and the ground truth (GT) bounding boxes from the validation set was used to assess the model's runtime parameter combinations. Each bounding box's size, ground center, and y-axis rotations (yaw) in 3D camera coordinates are used to calculate this value. The IoU of the oriented bounding boxes is then calculated by dividing their intersection volume by their union volume after they have been transformed

into axis-aligned boxes.

Every GT bounding box in an image is compared to the bounding box of the detected object for each detected object in that image. The GT box is correctly matched with the detected box that has the highest IoU. After that, these IoU values are stored so that, for every input image of the dataset, a list of IoU values between matching GT and prediction boxes is produced. Since the IoU computation is only carried out for GT boxes and their matching prediction boxes, the prediction box is simply discarded in situations where the suggested model predicts objects without a GT label.

## 5.8 Risk Analysis and Mitigation

Some potential risks that could impact the model’s assessment were identified during the parameter tuning and fusion process. The risks listed below have been mitigated:

- **Overfitting parameters:** If the erosion and depth factors are adjusted especially for a validation set, there is a chance that the results do not apply to other scenes or conditions. **Mitigation:** To capture a range of scenarios, a diverse validation subset from the KITTI dataset was used. Furthermore, performance was assessed using the average IoU across all images.
- **Inaccurate LiDAR-to-image Projections:** Inaccurate calibration data or errors in the transformation matrix  $T$  may cause misaligned projections and skewed evaluation metrics. **Mitigation:** Every matrix was loaded from the KITTI calibration files and manually checked.

### 5.8.1 Ethical Considerations

Advanced object detection models, such as the one used in this project, can raise ethical concerns, particularly if used for tasks like surveillance. These systems can track people and vehicles, which may jeopardize personal privacy if used inappropriately. This project only uses public, anonymized data from the KITTI dataset, so no personal or sensitive information is involved. Still, KITTI may not represent all real-world conditions, so the model may not be accurate in all situations. Given that the project relies on open-source tools, we take care to use them responsibly and ensure that they function as expected. While this work is intended for research purposes, its application in real-world settings needs extensive testing to ensure that it is used ethically.

## 6 Experiment Results and Discussion

### 6.1 Effect of Segmentation Mask Erosion and Depth Filtering

This experiment investigates the effect of segmentation mask erosion and depth filtering on object detection performance in the YOLOv11m-seg model using Kieffer’s fusion pipeline [1]. Specifically, the model’s performance without filtering (erosion = 0, depth = 0) was compared to the setup

with an erosion factor of 20 and a depth factor of 20, as those were the optimal settings found in Kieffer’s thesis [1].

Source	Filtering Setup	Car Avg. IoU (%)	Pedestrian Avg. IoU (%)	Cyclist Avg. IoU (%)	Inference Time (s)
Kieffer (2024)	No Filtering (0, 0)	5.90	1.08	N/A	0.2237
Kieffer (2024)	Filtering (20, 20)	20.28	20.37	N/A	0.2425
Ours	No Filtering (0, 0)	5.61	0.80	3.30	<b>0.0916</b>
Ours	Filtering (20, 20)	<b>26.98</b>	<b>33.26</b>	<b>14.26</b>	0.1008
<b>Best Avg. IoU: 0.8902 @ Filtering (20, 20), Image ID: 002019</b>					

Table 3: Comparison of average IoU and inference time across filtering setups from related work (using YOLOv8m-seg) and our work (using YOLOv11m-seg). Bold values indicate the best-performing setup.

Table 3 compares average IoU scores and inference times between our method and a related work using various filtering configurations. In the absence of filtering, both methods perform poorly: our approach achieves 5.61% for cars, 0.80% for pedestrians, and 3.30% for cyclists, whereas Kieffer (2024) reports 5.90% for cars and 1.08% for pedestrians. This demonstrates the ineffectiveness of raw segmentation masks and depth maps without refinement. After applying erosion and depth filtering (20, 20), our method shows significant improvement, reaching 26.98% for cars, 33.26% for pedestrians, and 14.26% for cyclists, outperforming Kieffer (2024), which achieved 20.28% and 20.37% for cars and pedestrians, respectively (no cyclist results reported). Inference time also favors our pipeline, with a significantly shorter runtime of 0.1008s versus 0.2425s in the related work. Notably, image index 002019 demonstrated the highest single-frame performance, with an overall average IoU of 0.8902 under the (20, 20) configuration.

These improvements are due to filtering mechanisms that improve the quality of fused features. Erosion removes loosely segmented pixels near object boundaries, which frequently correspond to background noise or incorrectly labeled regions. Since LiDAR points are more consistently aligned with object geometry inside objects, this helps focus attention on those areas. As for far-range LiDAR points, depth filtering removes those that are less likely to be connected to objects that are visible in the image, particularly for cyclists and pedestrians who usually take up shorter depth ranges. These preprocessing techniques enhance the fusion output’s spatial consistency and improve detection.

## 6.2 Optimizing Segmentation Mask Erosion and Depth Filtering

Kieffer’s (2024) [1] work optimized erosion and depth filtering parameters using an exhaustive evaluation of a defined range of values on the KITTI validation set. The initial range was determined through empirical testing on a few selected samples, and it was then iteratively expanded as IoU improvements were observed. His final parameter search space included erosion values ranging from

10 to 30 and depth values from 5 to 20. Each combination was tested on all 3,740 validation frames using heatmap visualizations to determine both the best-frame IoUs and the average IoUs per object class. The evaluation revealed that the erosion factor had a greater impact on car detection, whereas depth filtering had a greater influence on pedestrian detection. Based on these findings, Kieffer proposed a general-purpose configuration of erosion 25 and depth 20 for YOLOv8m-seg. Again, the cyclist class was excluded from his analysis due to the reasons mentioned before.

In contrast, our YOLOv11-seg model approach broadens and refines this methodology by addressing a larger and more granular parameter space. The tested values were the following:

- **Erosion Factor:** 10 to 70 (steps of 5)
- **Depth Factor:** 5 to 70 (steps of 5)

This extended grid provides a more detailed understanding of how different classes react to filtering. Furthermore, we assessed average IoU scores for the cyclist class.

This experiment aims to assess the impact of these parameters on the detection of distinct object classes using two primary metrics: the average IoU, which represents overall detection consistency across all objects, and the best IoU, which indicates the highest-quality detection of any single object.

According to the findings shown in Table 4, object classes react differently to depth and erosion settings. This is most likely due to their size, shape, and the way they appear in LiDAR images.

Table 4: Best Performing Erosion and Depth Settings Per Class

Class	Avg IoU (%)	Best Parameters (Erosion, Depth)
<b>Car</b>	29.57	(70, 70)
<b>Pedestrian</b>	35.25	(15, 20)
<b>Cyclist</b>	16.57	(70, 70)

For the *Car* class, the highest values (70, 70) produced the best average IoU for cars. This suggests that stronger filtering is more reliable over many examples, even if it slightly reduces peak performance.

The *Pedestrian* class performed better with lower erosion and depth (15, 20), achieving an average IoU of 35.25%. Pedestrians are smaller and more prone to erosion, so light filtering helps them maintain their shape.

The *Cyclist* class had the weakest results with an average of just 16.57%. This class is more difficult to detect, most likely because cyclists appear as a combination of body and bike, which confuses the model. Even the best settings (70, 70) did not yield significant results. Extra techniques may be required to improve performance in this class, such as *Class-Specific Post-Processing*, where different filtering or additional refinement steps are applied for the *Cyclist* class to improve its performance.

Table 5 recommends different parameter settings based on the goal. These recommendations are based on how erosion and depth values influenced detection results for each object class. They provide practical guidance based on their findings. Rather than simply showing raw numbers, it explains how to apply the results based on the detection goal. It also shows how erosion and depth tuning can be tailored to the requirements of various objects in real-world applications.

Table 5: Recommended Parameter Settings by Use Case

Use Case	Best Params (Erosion, Depth)	Rationale
Maximize <b>best-case detection</b>	(10, 15)	Achieves highest single-object IoU (96.55%) for Car; best for peak detection
Prioritize <b>Pedestrian IoU</b>	(15, 20)	Pedestrian average IoU peaks at 35.25%; low erosion preserves human contours
Improve <b>Cyclist robustness</b>	(65, 60) and (70, 70)	Cyclist detection is weak overall; slightly better average IoU at higher erosion/depth
Maximize <b>overall consistency</b>	(25, 25)	Achieves highest overall average IoU (89.02%); balanced across all classes with reasonable runtime

The first recommendation (10, 15) is for detecting a single object as accurately as possible. This setting resulted in the highest IoU for a car: 96.55%. The low erosion keeps the object’s shape very detailed, while the light depth filtering prevents useful information from being lost. This works well for large, clearly visible objects such as cars, particularly in good weather.

For pedestrians, (15, 20) provides the best results. These lower values preserve people’s small, narrow shapes. Higher erosion levels tend to remove thin parts such as arms and legs, making it more difficult to detect pedestrians accurately. Light filtering is sufficient to reduce noise while maintaining detection.

Cyclists are the most difficult to identify. The optimal settings were (65, 60) and (70, 70). These are high values, indicating significant erosion and depth filtering. Although performance remains low, this setting makes a minor improvement. One possible explanation is that cyclists frequently appear alongside bikes, causing clutter in the point cloud. Although it would only be marginally helpful, strong filtering might help distinguish significant features from the background.

The final setting (25, 25) is ideal for achieving good average detection across all objects. This combination produced the highest average IoU. It balances erosion and depth filtering so that the model removes some noise while retaining enough detail. It’s an excellent choice for general use, particularly when there are a variety of object types in the scene.

The erosion and depth parameters were set at **(25, 25)** for the following experiments because they provide a fair trade-off between processing stability and detection quality.

### 6.3 PCA for 3D Bounding Boxes

The PCA with z-axis rotation experiment attempted to see whether Principal Component Analysis (PCA) could improve 3D bounding box generation by better aligning boxes with the orientation of detected objects in the fused LiDAR-camera space.

The baseline method relied on axis-aligned 3D bounding boxes generated directly from 2D segmentation masks and projected LiDAR points. These boxes are simple to create, but they are frequently misaligned with the true orientation of elongated or rotated objects (such as pedestrians or cyclists). To address this, PCA was applied to each detected object’s clustered 3D points to determine the principal axes of variance. A bounding box was then fitted and rotated along the  $z$ -axis to reflect the object’s dominant orientation in the horizontal plane.

Kieffer (2024) [1] investigated PCA-based 3D bounding box alignment in his YOLOv8m-seg pipeline using erosion and depth parameters set to (20, 20). According to his findings, PCA reduced the IoU for cars (16.37%) while increasing the IoU for pedestrians to 28.13%, implying a benefit for smaller, vertically oriented objects. Using PCA, his reported inference time increased to 0.2425 seconds. Interestingly, Kieffer concluded that bounding boxes created without PCA were more accurate for the Car class, whereas PCA provided only marginal improvements for pedestrians. He also noted that the addition of tracking and segmentation refinement in subsequent pipeline stages reduced IoU, particularly for pedestrians.

Our PCA experiments were run on the YOLOv11m-seg model, with the erosion and depth filtering parameters set to (25, 25). The primary direction of spatial variance was determined by performing PCA on the corresponding cluster of 3D LiDAR points for each detected object. A rotated bounding box was then created by aligning its orientation with the principal axis of the horizontal ( $z$ ) plane. The goal was to reduce misalignment between bounding boxes and real-world object orientations, especially for elongated or rotated objects like pedestrians and cyclists. The experiment also sought to improve spatial accuracy and evaluate the trade-off between accuracy gains and the computational overhead caused by the PCA transformation.



Table 6: Impact of Applying PCA to YOLO-Based Models (Parameter Settings at (20, 20) for Kieffer’s model, and at (25, 25) for our model.

Model	PCA	Car Avg. IoU (%)	Pedestrian Avg. IoU (%)	Cyclist Avg. IoU (%)	Inference Time (s)
YOLOv8m-seg (Kieffer)	No	20.28	20.37	N/A	0.2416
	Yes	16.37	28.13	N/A	0.2425
YOLOv11m-seg (Ours)	No	<b>28.00</b>	31.41	<b>14.70</b>	<b>0.0948</b>
	Yes	19.73	<b>31.43</b>	9.27	0.2107

The impact of using Principal Component Analysis (PCA) for 3D bounding box construction in two distinct model configurations—YOLOv11m-seg (our implementation) and YOLOv8m-seg (from Kieffer’s work), is presented in Table 6. The experiment compares performance across object classes and inference time, using erosion and depth filtering at (25, 25) for our model and (20, 20) for Kieffer’s model.

The results in Table 6 also show that applying PCA with z-axis rotation to the YOLOv11 model (with erosion and depth filtering set to 25) results in mixed performance across object classes. Car detection drops in terms of average IoU from 28.00% to 19.73%, implying that PCA introduces bounding box misalignment or a loss of structural information for larger objects when rotated along the  $z$ -axis. Pedestrian detection shows a negligible improvement, up from 31.41% to 31.43%, which may be within the margin of error and might not be practically significant. This suggests that PCA does not significantly improve pedestrian detection accuracy. Finally, cyclist detection also declines, with the average IoU falling from 14.70% to 9.27%, indicating that the PCA transformation may cause more harm than good for smaller or irregularly shaped objects. Furthermore, the inference time more than doubles from 0.0948 to 0.2107 seconds, which further diminishes PCA’s relevance in real-time applications.

PCA displays a similar pattern in Kieffer’s YOLOv8m-seg implementation: the Pedestrian IoU improves significantly from 20.37% to 28.13%, while the Car IoU falls from 20.28% to 16.37%. Because of the frequent misclassifications brought on by label mismatches, Kieffer did not include the cyclist class in his evaluation. With or without PCA, his inference time is still rather high (0.242s), most likely because of the more intricate model architecture and more processing power of the GPU (RTX 4090 as opposed to the Apple M4 Max chip used in our tests).

All things considered, these findings lend credence to the idea that PCA is only conditionally advantageous and adds significant computational costs. PCA may be useful for pedestrian detection (as demonstrated more clearly in Kieffer’s results than in ours), but it is not appropriate for real-time multi-class detection tasks due to its detrimental effects on cyclist and car IoU and its longer inference time. PCA was therefore left out of our final pipeline.

## 6.4 YOLO Variant Evaluation

This experiment evaluates the performance of three YOLO model variants: *YOLOv8m-seg*, *YOLOv9e-seg*, and *YOLOv11m-seg*. All models were evaluated with the same preprocessing parameters (an

erosion factor of 25 and a depth threshold of 25). The purpose of this experiment is to determine how improvements in YOLO architecture across versions affect segmentation quality for various object types, while also taking inference time as a practical performance metric.

We also include results from Kieffer’s (2024) final *YOLOv8m-seg* model pipeline, which included depth filtering, PCA-based 3D bounding boxes, and segmentation refinement, in addition to our own experiments. This serves as a useful guide for understanding how modifications to the architecture and post-processing affect performance in different experimental configurations.

Table 7: Per-Class Average IoU and Inference Time Across YOLO Variants (parameters set to (20,20) for Kieffer’s model, and (25,25) for our models)

Model Variant	Car Avg. IoU (%)	Pedestrian Avg. IoU (%)	Cyclist Avg. IoU (%)	Inference Time (s)
YOLOv8m-seg (Kieffer)	16.37	28.13	N/A	0.2472
YOLOv8m-seg (ours)	27.69	31.94	14.63	<b>0.0836</b>
YOLOv9e-seg (ours)	27.99	<b>32.34</b>	<b>14.94</b>	0.1900
YOLOv11m-seg (ours)	<b>28.00</b>	31.41	14.70	0.0948

Table 7 shows that all three models perform reasonably well overall, but their performance differs depending on the class and metric.

In the *Car* class, *YOLOv11m-seg* outperformed the other models, with the average IoU of 28.00%. This suggests that YOLOv11’s updated architecture improves the detection and localization of large, well-defined objects such as cars.

For the *Pedestrian* class, *YOLOv9e-seg* had the highest average IoU (32.34%), indicating that it performs more consistently across multiple pedestrian instances than the other two variants. This could be due to regularization improvements or feature refinement in YOLOv9’s backbone, which aids in smaller, thinner shapes.

For the *Cyclist* class *YOLOv9e-seg* slightly led in Average IoU at 14.94%. This demonstrates that *YOLOv9e-seg* is slightly more consistent across samples for detecting cyclists when compared to the other two variants. *YOLOv11m-seg* had slightly lower scores in both metrics, implying that its improvements may not be as applicable to more complex or mixed-shape classes, such as cyclists.

Kieffer’s final YOLOv8m-seg pipeline, on the other hand, performs noticeably worse in terms of Pedestrian IoU (28.13%) and Car detection (16.37%). Cyclist results are not reported by his model for the reasons mentioned previously. Furthermore, our implementation of *YOLOv8m-seg* is both significantly faster (0.0836s vs. 0.2472s) and achieves significantly higher IoU values than

Kieffer’s version (Car: 27.69% vs. 16.37%, Pedestrian: 31.94% vs. 28.13%). This discrepancy is probably due to the differences in processing techniques, especially regarding the application of PCA.

Overall, our *YOLOv8m-seg* has the fastest inference speed at 0.0836 seconds per image, making it ideal for real-time applications. *YOLOv11m-seg* is slightly slower (0.1034 s), but it has the best overall performance in the Car class and shows great promise as a high-performing, and relatively fast model. *YOLOv9e-seg* is strong in some detection metrics, but has the slowest inference time at 0.1900 seconds (more than double the time when compared to the other variants), which may be a concern for real-time systems. Thus, the YOLOv11 model variant will be used for the next experiment, due to its balance between accuracy and inference time.

## 6.5 YOLOv11 Scaling Evaluation

Using the *YOLOv11-seg* architecture, this experiment investigates how model size affects segmentation performance. To ensure consistency, five model sizes were tested (Nano, Small, Medium, Large, and Extra Large) with the same preprocessing parameters (Erosion = 25, Depth = 25). The goal was to see how increasing model capacity affects detection quality across object classes and inference time. Table 8 shows the results of the experiment.

Table 8: Performance Comparison of Different YOLOv11 Model Sizes (Erosion = 25, Depth = 25)

Model Size	Car Avg. IoU (%)	Pedestrian Avg. IoU (%)	Cyclist Avg. IoU (%)	Inference Time (s)
Nano	27.82	30.72	<b>15.32</b>	<b>0.0402</b>
Small	27.86	31.45	14.77	0.0546
Medium	28.00	31.41	14.70	0.0948
Large	28.01	31.60	14.71	0.1012
Extra Large	<b>28.02</b>	<b>31.90</b>	15.21	0.1543

In the *Car* class, the *Extra Large* model had the highest average IoU (28.02%), indicating more consistent performance across samples with larger model capacity. In the *Pedestrian* class, the *Extra Large* model again had the highest Average IoU (31.90%). This demonstrates that deeper models provide only slight improvements in overall pedestrian detection consistency. In the **Cyclist** class, the *Nano* model had the highest Average IoU (15.32%), surpassing all larger variants. This unexpected result could be attributed to overfitting or diminishing returns in larger models that handle smaller, less frequent classes. It also suggests that lightweight models can better capture cyclist features when regularization effects are dominant.

In terms of efficiency, the *Nano* model outperformed all others, with an inference time of 0.0402 seconds per image. While the Extra Large model had slightly better average performance in some categories, it was nearly four times slower (0.1543 seconds). The *Medium* and *Large* models provided reasonable tradeoffs between speed and accuracy.

The findings indicate that, as the model size increases, the inference time also increases. However, the change in average IoU score is negligible. Therefore, the pipeline can scale well across model sizes without suffering excessive computational costs.

## 6.6 Limitations and Future Work

The proposed approach produces promising results in terms of average IoU and inference speed, but it has a number of limitations that affect its overall performance and generalizability.

One limitation is the limited amount of time and computational resources available during the development of this project. It was not feasible to test additional parameter settings or train the models on larger datasets. This means that some improvements may have been overlooked simply because there was insufficient time and computational power to test them.

Several improvements for future work are possible. One promising approach is to use class-specific post-processing. The current approach uses the same processing steps for all object types, regardless of size, shape, or appearance. However, different classes have unique visual features. It is possible to improve segmentation accuracy and reduce false positives by designing post-processing steps that are specific to each class (for example, adjusting erosion or depth thresholds). For example, pedestrians may benefit from less erosion to maintain their slim contours, whereas cars may benefit from stronger filtering to reduce background noise.

Furthermore, expanding the model to detect additional classes from the KITTI dataset, such as trucks, vans, trams, and sitting pedestrians, would increase its practical utility. These object types are common in real-world driving scenarios, and including them would enable the model to handle more complex scenes. It would also provide a more comprehensive understanding of the model’s strengths and weaknesses across various object types. Additionally, calculating mAP scores and AP scores for each class on all of the KITTI dataset’s difficulty levels (Easy, Medium, Hard) would further enhance understanding of the model’s capabilities.

Another improvement would be to train and test the model on larger and more diverse datasets, such as the *Waymo Open Dataset* [31] or *NuScenes* [32], to improve model generalizability. These datasets include a wider range of driving environments and sensor configurations. *Waymo* and *NuScenes* provide more data per scene and cover rare cases such as congested urban areas or nighttime driving, which KITTI does not.

Moreover, temporal information could be used to implement live object tracking. Currently, each frame is processed separately, which may result in video streams flickering or missing detections. The model can track objects more accurately over time by combining data from several consecutive frames. This would decrease noise and increase stability, particularly for objects that move quickly or are only partially visible. In real-time systems, reliable tracking is just as crucial as single-frame accuracy, so this research would also be valuable.

## 7 Conclusion

This thesis adapted a lightweight approach, presented by Kieffer (2024) [1], to a more recent YOLO model for 3D object detection by combining LiDAR and camera data with segmentation masks. The goal was to simplify the fusion pipeline while achieving strong performance at a low computational cost. The YOLOv11 segmentation model was integrated into an existing fusion pipeline that uses erosion and depth filtering to remove irrelevant regions and extract 3D points that correspond to detected objects.

Extensive testing was done to optimize the erosion and depth filtering parameters used to project 2D segmentation masks into 3D space. The experiment revealed their effect on the average IoU for three object classes: *Car*, *Pedestrian*, and *Cyclist*. These results were compared to Kieffer’s work, and several parameter recommendations were made for various use cases, such as maximizing best-case detection, consistency, and class-specific performance.

Following parameter optimization, the project compared multiple YOLO model variants (*YOLOv8m*, *YOLOv9e*, and *YOLOv11m*). The experiments revealed that *YOLOv11m* produced the best results for detecting cars, while *YOLOv9e* performed slightly better for detecting pedestrians and cyclists. Our *YOLOv8m* model was the fastest, but performed poorly compared to the other two variants. *YOLOv11m* had the best overall performance in the Car class and was less than 1% shy for the other classes when compared to the *YOLOv9e* model in terms of average IoU. In addition, the *YOLOv11m* offers a good balance between accuracy and inference time as it was twice as fast as the *YOLOv9e* variant. In contrast, Kieffer’s model underperformed compared to all three of our model variants, both in terms of average IoU score and inference time.

Furthermore, the *YOLOv11* model sizes (from *Nano* to *Extra Large*) were investigated for the trade-offs between speed and accuracy. The findings revealed that all the versions provided strong average IoUs while maintaining fast inference times. There was not much difference in performance in terms of average IoU scores, but the inference time increased as the model size increased. The *YOLOv11* models all delivered a balanced performance, demonstrating that the proposed pipeline can scale across model sizes without incurring significant complexity overhead.

In conclusion, the thesis offers the following contributions:

- We conduct a thorough literature review of the most recent state-of-the-art fusion models evaluated on the KITTI dataset, including LoGoNet, ViKIENet-R, and VirConv-S, which serve as reference points for our overall performance comparison. The literature review also introduces the baseline model.
- The LiDAR-camera fusion pipeline from Kieffer’s thesis [1] was replicated, and by substituting YOLOv11m-seg for the YOLOv8-based segmentation model, we were able to achieve notable gains in detection speed and accuracy. An average IoU of 28.00% for cars, 31.41% for pedestrians, and 14.70% for cyclists was attained by our implementation with erosion and depth filtering (25, 25). Its reduced inference time of 0.0948 seconds was more than twice as quick as Kieffer’s pipeline (0.2472s).

- We compared YOLO variants—YOLOv8m-seg (ours and Kieffer’s), YOLOv9e-seg, and YOLOv11m-seg—using the same fusion setup. Our YOLOv9e-seg model had the best pedestrian IoU (32.34%) and cyclist IoU (14.94%), while the YOLOv11m-seg had the best car IoU (28.00%) and competitive speed, making it the most balanced choice overall.
- We investigated the effect of scaling the YOLOv11 model. We discovered that the YOLOv11-Nano variant had the fastest inference time (0.0402 seconds) and the highest average cyclist IoU (15.32%). In contrast, the Extra Large model had marginally higher IoUs for cars (28.02%) and pedestrians (31.90%). However, the different scales of the model produced very similar results, and only indicate that the inference time increases as the model size increases.
- We provide limitations to our work and suggest several recommendations for future improvements. In order to better represent real-world driving scenarios, future work should test additional parameter configurations, apply class-specific post-processing to increase segmentation accuracy, and expand object classes beyond the current three classes (*Car*, *Pedestrian*, and *Cyclist*). Generalizability could be further improved by assessing performance on all KITTI difficulty levels as well as other sizable datasets like Waymo or NuScenes. Furthermore, incorporating temporal data for object tracking would enhance performance and stability in real-time applications.

Overall, this project provides a valuable starting point for real-time LiDAR-camera fusion and establishes the foundation for further improvements, including multi-class detection, class-specific post-processing, dataset expansion, learned fusion strategies, and live object tracking. With these enhancements, the model may be a better option for real-world applications in AVs.

## References

- [1] T. Kieffer, “Lidar-camera fusion for 3d object detection in autonomous driving systems,” master’s thesis, University of Luxembourg, Faculty of Science, Technology and Medicine, August 2024.
- [2] GlobeNewswire, “Self-driving cars market forecast at \$3.9 trillion by 2034; logistics and delivery segment to grow at fastest cagr of 14% as e-commerce and last-mile innovations accelerate,” Apr. 2025. Accessed: 2025-04-15.
- [3] European Agency for Safety and Health at Work, “Human error,” Aug. 2022. Accessed: 2025-04-15.
- [4] L. Liu, S. Lu, R. Zhong, B. Wu, Y. Yao, Q. Zhang, and W. Shi, “Computing systems for autonomous driving: State of the art and challenges,” *IEEE Internet of Things Journal*, vol. 8, no. 8, pp. 6469–6486, 2021.
- [5] H. Liu, C. Wu, and H. Wang, “Real time object detection using lidar and camera fusion for autonomous driving,” *Scientific Reports*, vol. 13, p. 8056, May 2023.
- [6] X. Dang, Z. Rong, and X. Liang, “Sensor fusion-based approach to eliminating moving objects for slam in dynamic environments,” *Sensors*, vol. 21, no. 1, 2021.
- [7] Z. Liu, H. Tang, A. Amini, X. Yang, H. Mao, D. L. Rus, and S. Han, “Bevfusion: Multi-task multi-sensor fusion with unified bird’s-eye view representation,” in *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 2774–2781, 2023.
- [8] H. Zhu, J. Deng, Y. Zhang, J. Ji, Q. Mao, H. Li, and Y. Zhang, “Vpfnet: Improving 3d object detection with virtual point based lidar and stereo data fusion,” *IEEE Transactions on Multimedia*, vol. 25, pp. 5291–5304, 2023.
- [9] X. Li, T. Ma, Y. Hou, B. Shi, Y. Yang, Y. Liu, X. Wu, Q. Chen, Y. Li, Y. Qiao, and L. He, “Logonet: Towards accurate 3d object detection with local-to-global cross-modal fusion,” in *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 17524–17534, 2023.
- [10] H. Wu, C. Wen, S. Shi, X. Li, and C. Wang, “Virtual sparse convolution for multimodal 3d object detection,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 21653–21662, June 2023.
- [11] A. Geiger, P. Lenz, and R. Urtasun, “Are we ready for autonomous driving? the kitti vision benchmark suite,” in *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3354–3361, 2012.
- [12] Z. Yu, B. Qiu, and A. W. H. Khong, “Vikienet: Towards efficient 3d object detection with virtual key instance enhanced network,” in *Proceedings of the Computer Vision and Pattern Recognition Conference (CVPR)*, pp. 11844–11853, June 2025.



- [13] Y. Zhang, Q. Zhang, J. Hou, Y. Yuan, and G. Xing, “Unleash the potential of image branch for cross-modal 3d object detection,” in *Advances in Neural Information Processing Systems* (A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, eds.), vol. 36, pp. 51562–51583, Curran Associates, Inc., 2023.
- [14] R. Girshick, J. Donahue, T. Darrell, and J. Malik, “Rich feature hierarchies for accurate object detection and semantic segmentation,” in *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 580–587, 2014.
- [15] R. Kaur and S. Singh, “A comprehensive review of object detection with deep learning,” *Digital Signal Processing*, vol. 132, p. 103812, 2023.
- [16] R. Girshick, “Fast r-cnn,” in *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 1440–1448, 2015.
- [17] S. Ren, K. He, R. Girshick, and J. Sun, “Faster r-cnn: Towards real-time object detection with region proposal networks,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 6, pp. 1137–1149, 2017.
- [18] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You only look once: Unified, real-time object detection,” 2016.
- [19] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, *SSD: Single Shot MultiBox Detector*, p. 21–37. Springer International Publishing, 2016.
- [20] R. Padilla, S. L. Netto, and E. A. B. da Silva, “A survey on performance metrics for object-detection algorithms,” in *2020 International Conference on Systems, Signals and Image Processing (IWSSIP)*, pp. 237–242, 2020.
- [21] J. Hanley, “The meaning and use of the area under a receiver operating characteristic (roc) curve,” *Radiology*, vol. 143, pp. 29–36, 05 1982.
- [22] Ultralytics, “Yolov11 - ultralytics.” <https://docs.ultralytics.com/>, 2025. Accessed: 2025-05-25.
- [23] J. Terven, D.-M. Córdova-Esparza, and J.-A. Romero-González, “A comprehensive review of yolo architectures in computer vision: From yolov1 to yolov8 and yolo-nas,” *Machine Learning and Knowledge Extraction*, vol. 5, no. 4, pp. 1680–1716, 2023.
- [24] G. Jocher and J. Qiu, “Ultralytics yolo11,” 2024.
- [25] S. Hao, Y. Zhou, and Y. Guo, “A brief survey on semantic segmentation with deep learning,” *Neurocomputing*, vol. 406, pp. 302–321, 2020.
- [26] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, and P. Dollár, “Microsoft coco: Common objects in context,” 2015.
- [27] D. Xu and Y. Tian, “A comprehensive survey of clustering algorithms,” *Annals of Data Science*, vol. 2, 08 2015.

- [28] M. Simon, S. Milz, K. Amende, and H.-M. Gross, “Complex-yolo: An euler-region-proposal for real-time 3d object detection on point clouds,” in *Computer Vision – ECCV 2018 Workshops: Munich, Germany, September 8-14, 2018, Proceedings, Part I*, (Berlin, Heidelberg), p. 197–209, Springer-Verlag, 2018.
- [29] X. Li, Y. Xiao, B. Wang, H. Ren, Y. Zhang, and J. Ji, “Automatic targetless lidar-camera calibration: a survey,” *Artif. Intell. Rev.*, vol. 56, pp. 9949–9987, September 2023.
- [30] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun, “Vision meets robotics: The kitti dataset,” *International Journal of Robotics Research (IJRR)*, 2013.
- [31] P. Sun, H. Kretzschmar, X. Dotiwalla, A. Chouard, V. Patnaik, P. Tsui, J. Guo, Y. Zhou, Y. Chai, B. Caine, V. Vasudevan, W. Han, J. Ngiam, H. Zhao, A. Timofeev, S. Ettinger, M. Krivokon, A. Gao, A. Joshi, Y. Zhang, J. Shlens, Z. Chen, and D. Anguelov, “Scalability in perception for autonomous driving: Waymo open dataset,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.
- [32] H. Caesar, V. Bankiti, A. H. Lang, S. Vora, V. E. Liong, Q. Xu, A. Krishnan, Y. Pan, G. Baldan, and O. Beijbom, “nusenes: A multimodal dataset for autonomous driving,” in *CVPR*, 2020.