Vehicle Detection and Counting in Traffic Surveillance

Doan Viet Hung[1] and Le Nguyen Phuong Uyen[2†]

[1]Student ID: 21127289.
[2]Student ID: 21127476.
[1, 2]21TGMT, HCMUS, Nguyen Van Cu, HCMC, Vietnam.

Contributing authors: dvhung21@clc.fitus.edu.vn;
lnpuyen21@clc.fitus.edu.vn;
[†]These authors contributed equally to this work.

Abstract

Vehicle detection and counting in traffic surveillance systems play a crucial role in intelligent transportation management. With the increasing number of vehicles in urban areas, traditional manual methods for traffic monitoring have become inefficient and error-prone. Automated systems that leverage computer vision and machine learning techniques can provide real-time insights, helping reduce congestion, improve traffic signal control, and support urban planning. This report presents a comprehensive approach to vehicle detection and counting using video input from fixed surveillance cameras. We explore a pipeline that includes frame preprocessing, object detection (using YOLO-based models), vehicle tracking, and counting through virtual lines or regions of interest. The system is evaluated based on accuracy, processing speed, and robustness under varying weather and lighting conditions. The results demonstrate high potential for deployment in smart city infrastructure.

Keywords: Vehicle Detection, Vehicle Counting, Traffic Surveillance, Computer Vision, YOLOv8, Object Tracking, ByteTrack, Smart Transportation

1 Introduction

1.1 Motivation:

Urban traffic congestion is a growing problem in cities worldwide. As populations rise and urbanization accelerates, the number of vehicles on the road continues to increase. Traditional traffic monitoring relies heavily on manual observation or outdated sensors, which are neither scalable nor cost-effective. Delays, pollution, and inefficient use of infrastructure are just a few consequences of poor traffic management. To address this, automated vehicle detection and counting systems are gaining attention as vital components of smart transportation solutions.

Advances in computer vision, deep learning, and edge computing make it possible to process video data in real-time, detect vehicles with high accuracy, and generate actionable data for authorities. These systems reduce human error, lower operational costs, and provide continuous, scalable monitoring across large areas.

1.2 Practical Application:

The integration of computer vision and deep learning in traffic surveillance enables a wide range of practical applications:

- Traffic Flow Analysis: Understanding peak hours, congestion points, and route efficiency.
- Adaptive Traffic Signal Control: Adjusting light timing based on real-time vehicle count.
- Smart Parking and Tolling: Counting vehicles entering or leaving specific zones.
- Accident Detection and Response: Identifying abnormal traffic patterns or sudden stoppages.
- Urban Planning and Infrastructure: Using long-term traffic data to design better roads and public transport routes.
- Law Enforcement and Rule Violation Detection: Monitoring for illegal U-turns, wrong-way driving, or speed violations.

1.3 Problem Statement: Input, Output

This study focuses on the task of detecting and counting vehicles from video input collected by fixed traffic surveillance cameras.

- **Input:**
  - Continuous video feed or pre-recorded footage from a static camera facing a road or intersection.
  - (Optional) A predefined region of interest (ROI) or a virtual counting line.
- **Output:**
  - The number of vehicles crossing a virtual line or entering a specified ROI, segmented over time (e.g., per minute/hour).
  - (Optional) Vehicle type classification (e.g., car, truck, motorcycle).

– (Optional) Visual annotations of detection and tracking (bounding boxes, IDs).

The core challenges include maintaining detection accuracy under varying lighting and weather conditions, ensuring real-time performance, and handling occlusions and overlapping vehicles effectively.

2 Related Work

2.1 Memon et al. (2018) – A Video Based Vehicle Detection, Counting and Classification System:

Memon et al.[1] presents a cost-effective and vision-based system for vehicle detection, counting, and classification, addressing the limitations of traditional sensor-based traffic monitoring systems which are often expensive and maintenance-intensive. The proposed system operates on traffic video footage and utilizes background subtraction methods—specifically, the Gaussian Mixture Model (GMM)—to detect moving vehicles. Once foreground objects are extracted, the system processes their contours to classify vehicles based on size and shape into three categories: Low Transport Vehicle (LTV), Medium Transport Vehicle (MTV), and Heavy Transport Vehicle (HTV).

The architecture consists of three main modules: background learning, foreground extraction, and vehicle classification. For background subtraction, the system employs OpenCV's BackgroundSubtractorMOG2, which dynamically selects the optimal number of Gaussian distributions for each pixel and handles shadow detection. Foreground objects are enhanced using morphological operations like dilation and erosion to ensure accurate contour extraction. Vehicle detection and counting are based on tracking the centroid of each vehicle as it crosses a virtual line defined within a user-selected Region of Interest (ROI).

Two classification methods are compared in this work: Contour Comparison (CC) and a machine learning approach using Bag of Features (BoF) and Support Vector Machine (SVM). The CC method relies on manually defined thresholds of contour properties such as area, solidity, and aspect ratio. In contrast, the BoF-SVM method trains a classifier on SIFT features extracted from vehicle images. Through simulations on five traffic videos, it was found that the CC method consistently produced more accurate classification results, especially in terms of alignment with ground truth data.

While the system performs effectively under various daytime traffic conditions, limitations include a lack of occlusion handling, dependency on human-defined ROIs, and poor performance in low-light/nighttime scenarios. The authors suggest that future improvements could include real-time integration with microcontrollers, automated ROI detection, advanced feature extraction (e.g., color or shape descriptors), and machine learning enhancements to support classification in complex scenes, including nighttime environments.

2.2 Lin et al. (2021) - A Real-Time Vehicle Counting, Speed Estimation, and Classification System Based on Virtual Detection Zone and YOLO:

Lin et al. [2] proposes a robust, real-time system for vehicle counting, speed estimation, and classification in traffic surveillance. It aims to overcome limitations of traditional systems that struggle with real-time performance and accuracy under complex road conditions. The core innovation of the system is its integration of YOLOv4, a deep learning-based object detection algorithm, to detect and classify vehicles accurately in real-time from video feeds.

The system architecture consists of four main modules: vehicle detection, tracking, speed estimation, and classification. In the detection stage, YOLOv4 is used to identify vehicles in each frame, leveraging its high speed and accuracy. Detected vehicles are then tracked across consecutive frames using optical flow estimation and a Kalman filter to predict motion and maintain consistent tracking IDs. A region of interest (ROI) is defined such that when a vehicle crosses a virtual detection line within this area, it is counted.

For speed estimation, the authors introduce a method that calculates vehicle speed based on the displacement of bounding box centroids across frames, calibrated with the frame rate and a real-world scale factor. Classification is performed simultaneously by YOLOv4, which categorizes vehicles into types such as cars, trucks, buses, etc.

The system was tested on urban road video datasets and benchmarked using metrics such as Mean Average Precision (mAP), F1-score, and Intersection over Union (IoU). Results showed high accuracy in vehicle detection and classification, with a vehicle counting accuracy of over 96% and speed estimation error under 10%. The system is capable of real-time performance (processing speed 25 FPS) on standard GPU hardware.

In conclusion, the authors demonstrate that using deep learning (YOLOv4) significantly improves the robustness and reliability of traffic monitoring systems in dynamic environments. Future work could involve integrating license plate recognition and expanding detection categories using more advanced deep learning models like YOLOv5 or EfficientDet.

2.3 Shaikh M. Ali et al. (2022) - Smart Traffic Monitoring Through Pyramid Pooling Vehicle Detection and Filter-Based Tracking on Aerial Images:

Shaikh M. Ali et al. [3] proposes a smart traffic monitoring system tailored for aerial imagery, which is an increasingly relevant domain due to the growing use of drones and high-altitude cameras in traffic surveillance. The core challenge in aerial vehicle detection lies in the small object sizes, varying scales, and complex backgrounds. To address this, the authors present a pipeline that combines deep learning-based semantic segmentation with a novel filter-based object tracking algorithm.

The vehicle detection component is handled by PSPNet (Pyramid Scene Parsing Network), a deep convolutional neural network that improves object recognition by aggregating context information at multiple scales using pyramid pooling. This allows

the model to accurately segment small vehicles from aerial imagery, even under scale variation and occlusion. The segmentation output is then refined using a bounding box filter and morphological operations to improve object boundary definition.

Once vehicles are detected, the system uses a filter-based tracking method that leverages geometric constraints and spatial consistency to track the detected vehicles across frames. Unlike traditional object trackers that may rely heavily on appearance or deep features, this method is lightweight and optimized for efficiency, using centroid position updates and directional logic to maintain consistent vehicle identities.

Experimental evaluations were conducted using publicly available aerial traffic datasets. The system was evaluated using metrics such as Intersection over Union (IoU), detection accuracy, and tracking consistency. Results showed that the proposed approach achieves high segmentation accuracy and robust tracking, with superior performance compared to traditional methods when applied to small, moving vehicles in aerial views.

The paper concludes that combining deep semantic segmentation with a lightweight tracking filter can provide a reliable and efficient solution for aerial traffic monitoring. Future work may involve integrating more advanced deep learning models like DeepLabv3+, adding vehicle classification, or applying the system to real-time drone feeds.

2.4 Chughtai and Jalal (2024) – Traffic Surveillance System: Robust Multiclass Vehicle Detection and Classification:

Chughtai and Jalal [4] introduces an efficient approach for real-time vehicle detection and counting using video-based traffic surveillance. The motivation arises from the increasing demand for intelligent transportation systems that can handle growing traffic volumes in urban areas without relying on expensive and hard-to-maintain hardware like sensors or radar. The authors propose a method that focuses on robustness, simplicity, and low computational cost, making it suitable for real-time applications on standard computing devices.

The system architecture includes three main stages: background subtraction, vehicle detection, and vehicle counting. Background subtraction is performed using a Gaussian Mixture Model (GMM), which helps isolate moving vehicles from the static background. Following this, the system applies contour-based analysis and morphological operations to detect the vehicles. Bounding boxes are drawn around detected vehicles, and their positions are analyzed in subsequent frames to determine their movement and prevent duplicate counting.

To ensure accurate counting, a virtual detection line is defined within the Region of Interest (ROI). A vehicle is counted only when its centroid crosses this virtual line, and duplicate counts are avoided by tracking movement direction and position consistency across frames. The system has been tested on various real-world traffic videos captured under different lighting conditions. Experimental results show that the method is able to detect and count vehicles with high precision and low false detection rates, achieving performance suitable for practical deployment.

In conclusion, the proposed method offers a reliable and resource-efficient solution for vehicle detection and counting in traffic surveillance systems. Although the current

work focuses on detection and counting, the authors note that future enhancements could include vehicle classification and more sophisticated tracking methods to handle occlusions and improve performance in crowded or complex scenarios.

3 Proposed Method

In this section, we describe the proposed method for vehicle detection and counting, based on the YOLOv8s model. The method is built upon transfer learning using a pretrained YOLOv8s backbone, followed by fine-tuning on the KITTI dataset. The key components of the method include model architecture, training configuration, and the composite loss function used during optimization. After being successfull in detection, we apply object tracking (ByteTrack) to ensure accurate counting without double-counting.

3.1 Model:

The detection framework adopted in this work is based on the **YOLOv8s** (You Only Look Once version 8, small variant) model developed by Ultralytics. YOLOv8s is a one-stage object detector designed for high-speed inference and competitive accuracy, making it suitable for real-time applications in traffic surveillance.

YOLOv8s comprises three main modules:

- **Backbone: Feature Extraction Powerhouse**
  - **CSPDarknet-based:** The backbone of YOLOv8 is a modified version of the CSPDarknet53 architecture, originally introduced in earlier YOLO versions like YOLOv4. This structure consists of 53 convolutional layers and leverages Cross-Stage Partial (CSP) connections to improve computational efficiency and gradient flow. In YOLOv8, the backbone has been refined to strike a balance between depth and speed, making it suitable for real-time applications.
  - **C2f Module:** A standout innovation in YOLOv8 is the replacement of the C3 module (used in YOLOv5) with the C2f module. The C3 module relied on outputting features from only the final Bottleneck layer, whereas the C2f module integrates outputs from multiple Bottleneck layers (each comprising two 3x3 convolutions with residual connections). This change enhances feature richness by capturing more granular details, improving detection performance without significantly increasing computational overhead.
  - **Convolutional Adjustments:** The initial 6x6 convolution in the stem of previous models has been replaced with a 3x3 convolution. This reduces the receptive field size early in the network, allowing finer feature detection and lowering computational complexity.
  - **SPPF:** An upgraded version of SPP.
- **Neck: Feature Aggregation and Multi-Scale Processing**
  - The Neck in YOLOv8 serves as an intermediary between the backbone and the head, aggregating features extracted from various layers of the backbone to enable multi-scale object detection. It adopts a Feature Pyramid Network

(FPN) and Path Aggregation Network (PAN) topology, which has been a staple in YOLO models since YOLOv4.
  - **Enhanced Multi-Scale Fusion:** The Neck combines high-level semantic features (from deeper layers) with low-level spatial details (from shallower layers), ensuring that the model can detect objects of varying sizes effectively. The PAN component further refines this process by facilitating bottom-up feature propagation, improving the localization of small objects.
  - **Efficiency Optimization:** While retaining the multi-scale capabilities of its predecessors, YOLOv8's Neck is optimized to minimize latency, making it more suitable for deployment on resource-constrained devices.
- **Head: Anchor-free detection with decoupled head**
  - The Head of YOLOv8 is where the model's predictions—bounding boxes, object classes, and confidence scores—are produced. Unlike earlier YOLO versions (e.g., YOLOv5 and prior), which relied on anchor-based detection, YOLOv8 adopts an anchor-free approach, marking a significant shift in design philosophy.
  - **Anchor-Free Detection:** Instead of predicting offsets from predefined anchor boxes, YOLOv8 directly predicts the center of an object. This eliminates the need to tune anchor box sizes and aspect ratios for specific datasets, simplifying the model and reducing the number of bounding box predictions. As a result, the Non-Maximum Suppression (NMS) post-processing step is faster, enhancing inference speed.
  - **Decoupled Head:** The Head is detachable and processes feature maps from the Neck to output bounding boxes and class probabilities independently. This decoupling improves flexibility and allows the model to adapt to various vision tasks beyond object detection, such as segmentation and classification.

The model takes an input image resized to 640×640 pixels and outputs a set of bounding boxes, class probabilities, and objectness scores. For this study, we use the pretrained weights from the COCO dataset to initialize the model and fine-tune it on the **KITTI dataset**, which consists of annotated street scenes with vehicles.

3.2 Training Configuration:

The training process involves fine-tuning the YOLOv8s model using images and annotations from the KITTI dataset. The dataset is converted to the YOLO format, containing separate directories for training and validation images and labels.

Key training parameters are summarized as follows:

- **Model:** YOLOv8s pretrained on COCO
- **Input Size:** 640×640 pixels
- **Epochs:** 100
- **Optimizer:** set `"auto"` (default in Ultralytics with fine-tune task is `AdamW`)
- **Batch Size:** Defined based on available hardware (CPU/GPU)
- **Confidence Threshold:** 0.5 (during inference)

The model is trained using the `model.train()` API from the Ultralytics library. After each epoch, training and validation losses are logged and visualized, including individual loss components for bounding box regression, classification, and distribution focal loss.

3.3 Loss Function:

YOLOv8 employs a composite loss function that integrates multiple components to jointly optimize object detection performance. The total loss L_{total} is expressed as:

$$L_{\text{total}} = \lambda_{\text{box}} L_{\text{box}} + \lambda_{\text{cls}} L_{\text{cls}} + \lambda_{\text{dfl}} L_{\text{dfl}}, \tag{1}$$

where λ_{box}, λ_{cls}, and λ_{dfl} are weighting coefficients (tuned internally in the YOLO framework), and the individual loss components are defined as follows:

- **Box Loss** (L_{box}): Measures the error between predicted bounding boxes and ground-truth boxes using IoU-based metrics such as CIoU or GIoU. This loss encourages accurate localization and overlap.
- **Classification Loss** (L_{cls}): Implements binary cross-entropy (BCE) loss to evaluate how accurately the model predicts object classes. It is computed over the detected bounding boxes for each object.
- **Distribution Focal Loss (DFL)** (L_{dfl}): A novel loss function used for bounding box regression, which models the box offsets as discrete probability distributions. It enhances the localization accuracy by providing finer granularity in box prediction.

The combination of these losses allows the model to effectively learn both classification and precise localization of vehicles in diverse traffic scenes.

3.3.1 Bounding Box Regression Loss ($\mathcal{L}_{\text{box}}$):

This loss penalizes inaccuracies in the predicted bounding box locations and sizes. YOLOv8 uses a variant of the Complete Intersection over Union (CIoU) loss, which goes beyond simple IoU by incorporating:

- **Overlap area (IoU):** Encourages high intersection between prediction and ground truth.
- **Center point distance:** Reduces the distance between the centers of predicted and ground-truth boxes.
- **Aspect ratio consistency:** Aligns the aspect ratios of prediction and target.

The CIoU loss is defined as:

$$\mathcal{L}_{\text{CIoU}} = 1 - \text{IoU} + \frac{\rho^2(\mathbf{b}, \mathbf{b}^{gt})}{c^2} + \alpha v, \tag{2}$$

where ρ is the Euclidean distance between the centers, c is the diagonal length of the smallest enclosing box, and v measures aspect ratio divergence. This formulation improves convergence and localization accuracy.

3.3.2 Classification Loss ($\mathcal{L}_{\text{cls}}$):

This loss measures how well the model classifies detected objects. YOLOv8 applies binary cross-entropy (BCE) loss across each class prediction using one-hot encoding:

$$\mathcal{L}_{\text{cls}} = -\sum_{i=1}^{C} \left[y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i) \right], \tag{3}$$

where C is the number of classes, y_i is the ground truth (0 or 1), and $\hat{y}_i$ is the predicted probability. The BCE formulation supports multi-label classification and helps the model learn robust object semantics.

Additionally, label smoothing is sometimes applied to prevent overconfidence in class predictions, enhancing generalization on unseen data.

3.3.3 Distribution Focal Loss ($\mathcal{L}_{\text{dfl}}$):

YOLOv8 introduces a refined regression technique called **Distribution Focal Loss (DFL)** to improve the precision of bounding box coordinates. Rather than predicting coordinates directly, DFL models the offsets as discrete probability distributions over a range of bins.

For a given ground-truth coordinate y (e.g., box offset), DFL computes a weighted cross-entropy between the predicted distribution $\hat{p}$ and a soft label distribution centered around y:

$$\mathcal{L}_{\text{dfl}} = -\sum_{i=1}^{K} q_i \log(\hat{p}_i), \tag{4}$$

where q_i is the target probability for bin i, and K is the number of discrete bins.

This allows the model to learn from continuous-valued ground truth using a differentiable, probabilistic approach. It increases localization granularity and smooths the learning signal compared to traditional L_1 or L_2 losses.

3.3.4 Loss Synergy and Optimization:

Each loss component plays a synergistic role:

- $\mathcal{L}_{\text{box}}$ drives accurate spatial localization.
- $\mathcal{L}_{\text{cls}}$ ensures semantic discrimination between object classes.
- $\mathcal{L}_{\text{dfl}}$ refines the bounding box precision using probabilistic modeling.

Together, they form a comprehensive optimization objective for real-time object detection. During training, these losses are monitored separately to assess convergence and tuning. Loss curves (e.g., box loss, classification loss, DFL loss) are plotted to visualize training dynamics and identify potential overfitting or underfitting.

3.4 Tracking and Counting with ByteTrack:

To enable robust multi-object tracking and accurate vehicle counting, we incorporate the **ByteTrack** tracking algorithm in conjunction with YOLOv8s detection results. ByteTrack is a state-of-the-art object tracker known for its efficiency and high performance, especially in crowded and complex scenes.

3.4.1 ByteTrack Integration:

ByteTrack operates on frame-level detections provided by the object detector. For each video frame, the YOLOv8s model generates bounding boxes, class labels, and confidence scores. These detections are then passed to ByteTrack, which maintains consistent *Track IDs* across consecutive frames, enabling reliable tracking of individual vehicles over time.

Formally, given a set of detections $\mathcal{D}_t$ at time t, ByteTrack solves the data association problem by assigning detection-to-track matches using IoU and motion estimation. It also incorporates low-score detections to recover from temporary detection failures, increasing robustness.

3.4.2 Vehicle Counting Logic:

To count vehicles accurately, we define two virtual lines in the image:

- A **vertical line** positioned in the center of the frame
- A **horizontal line** near the bottom portion of the frame

An object is counted if it crosses one of these lines in a defined direction, and it has not been previously counted. The following conditions are verified before incrementing the count:

- The object's center (cx, cy) moves across the line compared to the previous frame (based on track history).
- The object has existed for at least three frames to avoid transient noise.
- The crossing occurred within a defined threshold distance from the line.
- The object has not yet been counted in the current session.

Each tracked object is associated with a memory record that stores its previous and current position, class ID, lifetime, and flags indicating whether it has already been counted in either direction. This is implemented using a Python dictionary indexed by *track ID*.

The count is updated per class using a dictionary of counters. Events are also logged in a CSV file (`count_log.csv`), which includes frame index, timestamp, object ID, class label, crossing type (vertical or horizontal), and coordinates.

3.4.3 Visualization and Output:

The system annotates each frame with:

- Bounding boxes and track IDs
- Labels showing class name, confidence, and ID
- Visual indicators for the vertical and horizontal counting lines
- Dynamic overlay showing real-time class-wise counts

The output video is written to disk, providing a visual summary of both detection and tracking performance, with real-time updates on counting statistics.

This combined YOLOv8s + ByteTrack pipeline demonstrates strong performance in vehicle tracking and counting tasks, maintaining track consistency even under partial occlusions or brief detection lapses, making it highly effective for traffic surveillance applications.

4 Experiments

4.1 Dataset:

The KITTI dataset [5] is one of the most widely used benchmark datasets for autonomous driving and computer vision research, particularly in tasks involving object detection, tracking, and scene understanding. It was collected using a suite of sensors mounted on a moving vehicle in Karlsruhe, Germany, and provides rich, high-resolution data representative of real-world urban environments.

For the purpose of this work, we utilize the `object detection` subset of KITTI, which includes annotated images suitable for training deep learning-based vehicle detection and counting systems.

4.1.1 Data Composition:

The dataset includes the following key components:

- **RGB Images:** High-resolution front-view camera images with a resolution of 1242×375 pixels, captured under diverse traffic and lighting conditions.
- **Annotations:** Each image is accompanied by a text file containing precise bounding box annotations and class labels.
- **Object Categories:** Annotated classes include `Car`, `Van`, `Truck`, `Pedestrian`, `Person Sitting`, `Cyclist`, `Tram`, and `Misc`.
- **Training and Testing Splits:** The dataset consists of 7,481 images for training and 7,518 for testing. Since the testing annotations are withheld, only the training split is typically used for both training and validation by creating a manual split.

4.1.2 Annotation Format:

Each annotation contains detailed information for every object instance:

- `Class Label`: Semantic class of the object (e.g., Car, Truck).
- `Truncation`: Degree of object truncation (float between 0 and 1).
- `Occlusion`: Level of object occlusion (0 = fully visible, 1 = partly occluded, 2 = largely occluded, 3 = unknown).

- `Alpha`: Observation angle of the object in the image.
- `Bounding Box`: Pixel coordinates $(x_{\min}, y_{\min}, x_{\max}, y_{\max})$ enclosing the object.
- `3D Dimensions and Location`: Includes height, width, length, and the object's location in camera coordinates.

In this work, only 2D bounding boxes and class labels are used for training the YOLOv8 model.

4.1.3 Relevance to Vehicle Detection and Counting:

KITTI is especially well-suited to the vehicle detection and counting task for several reasons:

- **Urban Traffic Scenarios:** The dataset captures a wide variety of urban scenes, including congested traffic, intersections, and occlusions—conditions frequently encountered in real-world surveillance.
- **High-Quality Annotations:** Accurate object boundaries and labels enable effective training and evaluation of detection models.
- **Class Diversity:** Multiple vehicle types (e.g., car, van, truck, tram) allow for fine-grained classification in counting applications.
- **Temporal Continuity:** Though not explicitly tracked, consecutive frames can be used for synthetic tracking benchmarks, especially in multi-object tracking (MOT) pipelines.

4.1.4 Challenges:

Despite its utility, KITTI presents several challenges:

- **Imbalanced Classes:** The majority of annotated objects are `Cars`, leading to class imbalance and underrepresentation of categories like `Cyclist` or `Truck`.
- **Occlusions and Truncations:** Urban environments result in frequent object occlusions, making detection and tracking difficult.
- **Fixed Viewpoint:** The dataset is collected from a single forward-facing camera, limiting the diversity of viewpoints.
- **Annotation Inconsistencies:** Some annotations may contain noise or subjective labeling, particularly for partially visible or distant objects.

4.1.5 Preprocessing and Format Conversion:

To use the KITTI dataset with YOLOv8, the annotations must be converted to YOLO format:

- Convert bounding boxes from $(x_{\min}, y_{\min}, x_{\max}, y_{\max})$ to normalized YOLO format: $(x_{\text{center}}, y_{\text{center}}, w, h)$, relative to image width and height.
- Map KITTI class labels to YOLO class indices (e.g., `Car` $\rightarrow$ 0, `Truck` $\rightarrow$ 1, etc.).
- Organize the dataset into separate directories for `images/train`, `images/val`, `labels/train`, and `labels/val`.

These steps ensure compatibility with Ultralytics' training pipeline and allow effective fine-tuning of the pretrained YOLOv8 model.

4.1.6 Augmentation:

To improve model generalization and robustness, especially under varied real-world conditions such as lighting changes and motion blur, we applied a diverse data augmentation pipeline during training. The augmentations were implemented using the `Albumentations` library, which provides high-performance and flexible image transformations. The transformations were applied only to training images, while validation images remained unchanged to ensure unbiased evaluation.

The augmentation strategy includes both geometric and photometric transformations. A horizontal flip was applied with a 50% probability to simulate mirrored traffic scenarios. Brightness and contrast adjustments were introduced randomly to mimic day-night variations and shadow effects. Motion blur (20% chance) simulated the blur caused by fast-moving vehicles or camera motion, while random gamma correction and CLAHE (Contrast Limited Adaptive Histogram Equalization) enhanced image contrast and emulated various lighting conditions.

To simulate camera variations and slight perspective distortions, an affine transformation was used, which includes random scaling (90–110%), translation (up to 5%), and small rotations (±10 degrees). Additionally, color-based augmentations were introduced using a OneOf wrapper, which randomly applied either hue-saturation-value adjustments or RGB color shifts to diversify the color spectrum of the dataset.

All augmentations preserved bounding box integrity using the BboxParams module in YOLO format, ensuring that the transformed labels remained consistent with the augmented images. This comprehensive augmentation pipeline enhances the model's ability to detect vehicles in various environmental and visual conditions, thus improving its real-world applicability.

4.2 Metrics:

To assess the effectiveness of our object detection model on the KITTI dataset, we employ a range of standard evaluation metrics including precision, recall, F1-score, and confusion matrix analysis. These metrics are computed over the validation set using the predictions from the YOLOv8s model.

Table 1 Validation of YOLOv8s performance

Metric	Value
Precision (B)	0.939
Recall (B)	0.904
mAP50 (B)	0.948
mAP50-95 (B)	0.769
Fitness Score	0.787

mAP50(B): The mean Average Precision at IoU threshold 0.5 is 94.8%, indicating excellent detection quality. This metric reflects the average precision of the model when predictions are considered correct if they overlap with ground truth by at least 50%. A value this high shows that the model is highly effective at localizing vehicles reasonably accurately, and is likely well-tuned for the dataset.

mAP50-95(B): This is a more stringent metric that averages the AP over multiple IoU thresholds ranging from 0.5 to 0.95. A score of 76.9% still reflects strong overall performance, but also highlights that the model may be slightly less accurate in cases requiring very precise localization. In practice, this drop is typical and acceptable, especially if the application (like traffic analytics) can tolerate small variations in bounding box sizes.

Fitness: The fitness score is a composite metric used during model training to evaluate how well the model balances between precision, recall, and mAP. A score of 0.787 shows that the model is well-optimized overall. This value helps guide the training process and hyperparameter tuning. While not an absolute performance indicator, it suggests the model is performing well in terms of both classification and localization.

4.2.1 Precision–Recall Characteristics:

Precision (P) measures the proportion of correctly predicted positive samples among all positive predictions:

$$\text{Precision} = \frac{TP}{TP + FP}$$

$\Longrightarrow$ This value indicates that when the model predicts a bounding box, it is correct 93.9% of the time. High precision means the model produces few false positives—i.e., it rarely identifies a vehicle when none exists. In a traffic surveillance context, this is crucial because it prevents over-counting or misidentifying vehicles, which could lead to misleading traffic flow analysis or congestion estimates.

Recall (R) quantifies the model's ability to detect all relevant instances:

$$\text{Recall} = \frac{TP}{TP + FN}$$

$\Longrightarrow$ A recall of 90.4% demonstrates that the model is able to detect the majority of actual objects in the scene. In simpler terms, out of all the real vehicles present in the images, 90.4% were correctly detected by the model. This is essential in applications like traffic monitoring and vehicle counting, where missing a vehicle (false negatives) can skew statistical data or compromise system reliability.

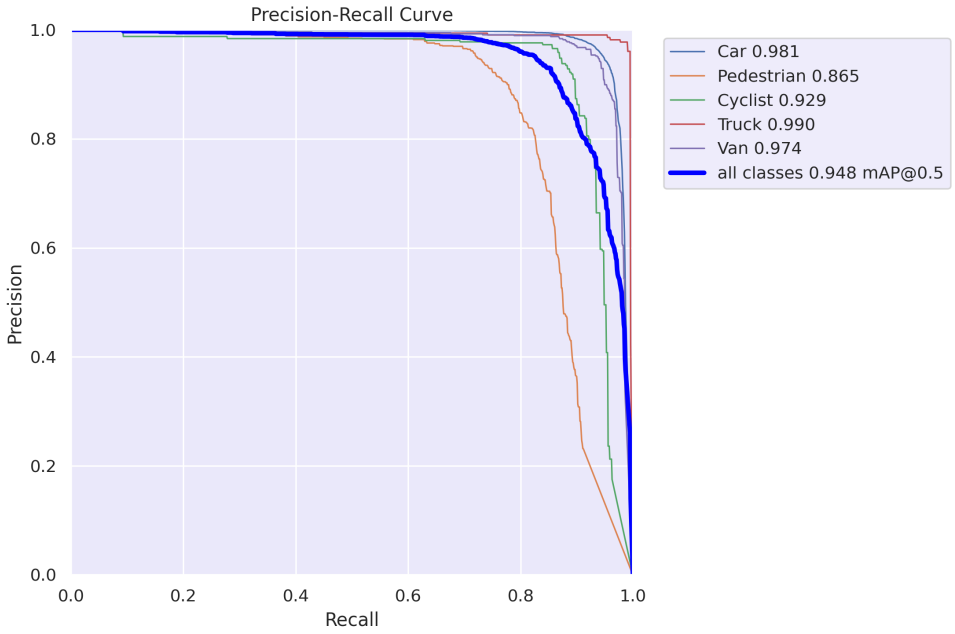


Fig. 1 Precision–Recall (PR) Curve

As shown in the Precision–Recall (PR) curve (Figure 1), the model performs exceptionally well in detecting vehicles, with particularly high average precision for Truck (0.990), Car (0.981), and Van (0.974), indicating excellent precision and recall across these classes. Cyclist detection is also strong (0.929), though slightly less stable at higher recall levels. Pedestrian detection lags behind (0.865), with a notable drop in precision at high recall, likely due to occlusion and smaller object sizes. The overall mAP0.5 of 0.948 confirms that the model is highly effective for vehicle detection in traffic surveillance, though further optimization may be needed for robust detection of pedestrians and cyclists.

4.2.2 F1 Score and Confidence Thresholding:

The F1 score balances precision and recall, defined as:

$$\mathrm{F1} = 2 \cdot \frac{\mathrm{Precision} \cdot \mathrm{Recall}}{\mathrm{Precision} + \mathrm{Recall}}$$

Figure 2 illustrates the F1-score as a function of the confidence threshold. The optimal threshold is indicated by the peak of this curve, which is used for filtering predictions during inference.

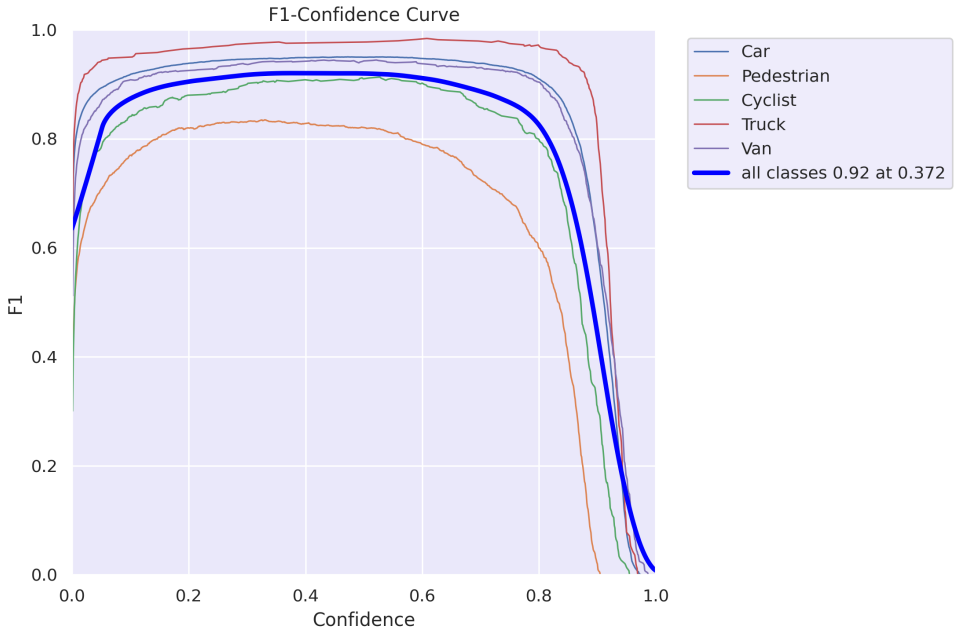


Fig. 2 F1 Score vs. Confidence Threshold

The F1-Confidence Curve provides insights into the balance between precision and recall across varying detection confidence thresholds for each class. The F1 score, which is the harmonic mean of precision and recall, peaks when both metrics are optimally balanced. In this plot, the overall maximum F1 score across all classes is 0.92, achieved at a confidence threshold of 0.372, suggesting this is the ideal point to filter detections for optimal performance.

Among individual classes, Truck consistently outperforms others, maintaining an F1 score near 1.0 across a wide confidence range, followed by Car and Van, both showing stable high F1 scores, indicating reliable detection even at higher confidence thresholds. Cyclist follows with decent performance, though with a more noticeable dip at high thresholds. Pedestrian detection, however, displays a lower and less stable F1 curve, peaking lower and declining more steeply, reflecting earlier findings that it is the most challenging class for the model.

Overall, this analysis confirms the model is highly effective in detecting vehicles and moderately reliable for cyclists, but improvements may be needed in detecting pedestrians with both precision and recall in balance.

4.2.3 Precision and Recall vs. Confidence:

Precision and recall curves plotted against the model's confidence scores (Figures 3, 4) help determine the impact of the confidence threshold on prediction quality. As confidence increases, precision generally improves while recall tends to decrease.

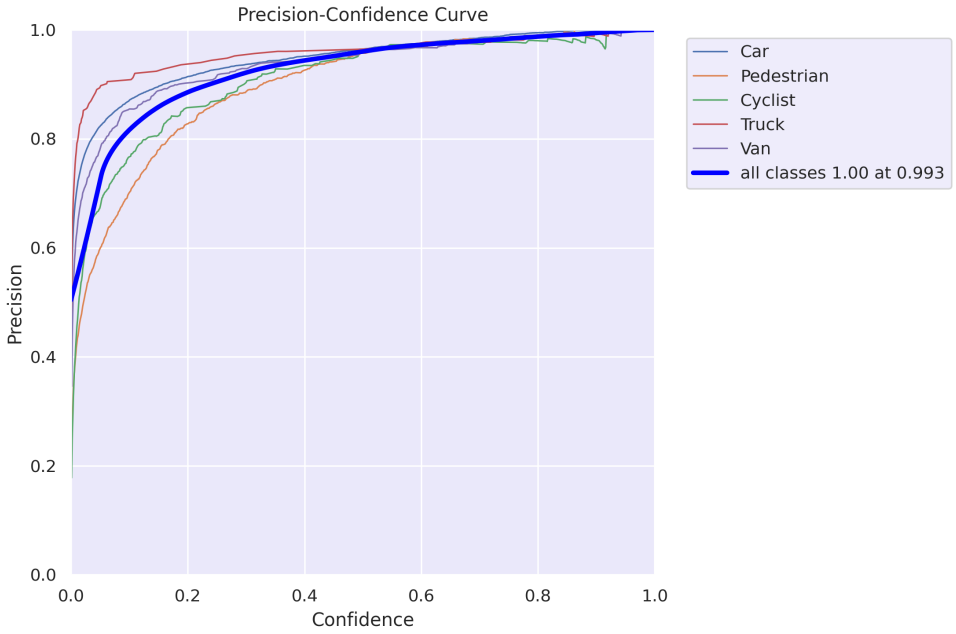


Fig. 3 Precision vs. Confidence

In figure 3, we observe that precision increases with confidence, reaching 1.00 at a confidence threshold of 0.993 for all classes. This trend indicates that when the model is highly confident about a detection, it is almost always correct. The Truck class maintains near-perfect precision throughout, while Car and Van also show consistently high precision. However, the Pedestrian and Cyclist classes have lower precision at low confidence levels, improving steadily as the confidence threshold increases—highlighting the trade-off between detecting more objects and maintaining correctness.

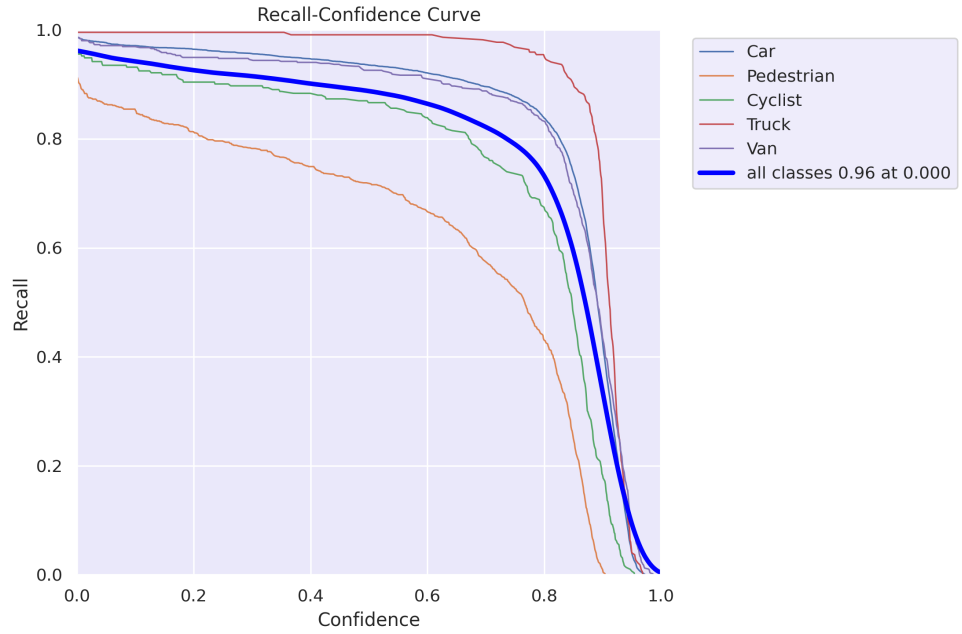


Fig. 4 Recall vs. Confidence

Conversely, in figure 4, recall decreases as confidence increases, with a maximum value of 0.96 at confidence = 0.000, meaning the model captures nearly all true positives when every low-confidence prediction is included. This comes at the cost of lower precision, as seen in the previous curve. Truck again shows strong performance, maintaining high recall across a broader confidence range, while Pedestrian recall declines steeply with increasing confidence—indicating that the model often detects pedestrians with lower confidence, and filtering out low-confidence predictions may lead to missed detections.

These two plots highlight the classic precision-recall trade-off, where setting a lower confidence threshold improves recall but may hurt precision, and vice versa. Choosing the optimal threshold depends on the specific application's tolerance for false positives versus false negatives.

4.2.4 Confusion Matrix Analysis:

The confusion matrix (Figure 5) provides detailed insights into the model's per-class performance. Each cell (i, j) represents the number of instances of class i predicted as class j.

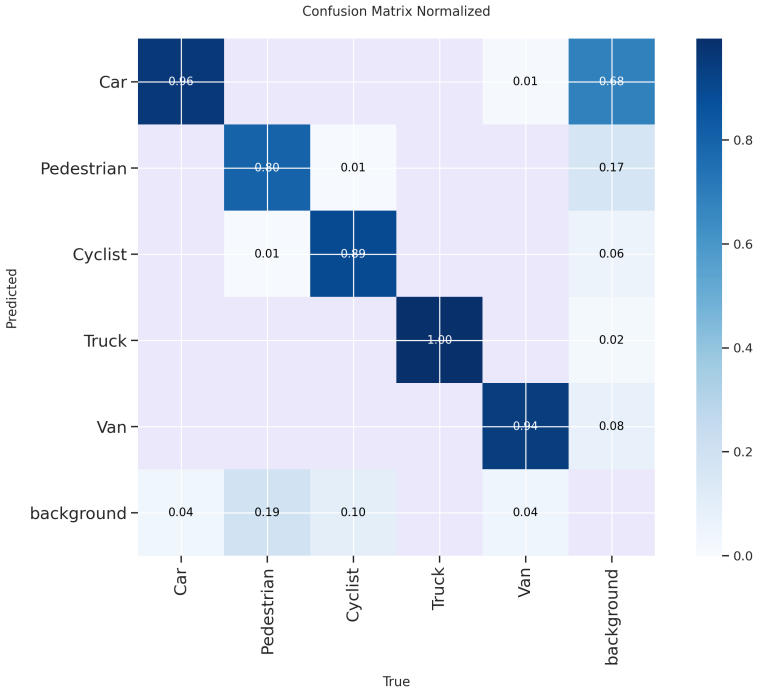


Fig. 5 Normalized Confusion Matrix

The normalized confusion matrix provides a detailed overview of how well the model distinguishes between different classes by comparing the predicted labels against the ground truth. The diagonal values represent correct predictions, while off-diagonal values indicate misclassifications. The model performs exceptionally well on Truck with a perfect classification rate of 1.00, and similarly high accuracy on Car (0.96) and Cyclist (0.89). Pedestrian detection, while still decent at 0.80, suffers from notable confusion with the background (0.17), suggesting that the model sometimes fails to detect pedestrians or misclassifies them as background clutter—possibly due to occlusion or scale variation. Van also has lower reliability, with a correct classification rate of 0.84, and a small portion misclassified as background (0.08) or as other vehicle classes (e.g., Car or Truck).

Additionally, background false positives appear for most classes, especially Pedestrian (0.19) and Cyclist (0.10), which reflects the model occasionally detecting objects where none exist. These findings align with earlier precision-recall analysis: the model is highly confident and accurate with large, distinct classes like Trucks and Cars, but struggles more with smaller or less distinct classes like Pedestrians and Cyclists. Targeted data balancing, better augmentation, or class-specific tuning may improve performance on these challenging categories.

4.2.5 Aggregate Metrics:

Figure 6 summarizes the overall performance using class-wise average precision (AP), mean average precision (mAP), and mean recall. These metrics confirm that the YOLOv8s model achieves competitive performance, suitable for downstream tasks such as vehicle counting and tracking.

Class	Images	Instances	Box(P	R	mAP50	mAP50-95)
all	1497	7631	0.939	0.904	0.948	0.769
Car	1338	5693	0.947	0.949	0.981	0.864
Pedestrian	349	873	0.917	0.756	0.865	0.537
Cyclist	209	292	0.928	0.884	0.929	0.697
Truck	208	220	0.961	0.991	0.99	0.904
Van	421	553	0.942	0.942	0.974	0.843

Fig. 6 Validation Metrics Summary

4.3 Experimental Setup:

Our project will run all on Kaggle. The instruction below will guide you how to download dataset (KITTI), fine-tune the model again if you want to work on another dataset (for instance, UA-DETRAC, Visdrone,...) or reuse my pretrained model results to test on my/your reality urban traffic video.

a) Step 1: Create/Import notebook

Access this google drive link and download `.ipynb` file: https://drive.google.com/file/d/1hVbCIQ7l8qr9f2s2OYQxLBvr9UKtZ-Fz/view?usp=sharing.

This file contains vital source code including: commands for installing necessary libraries, preprocessing the raw dataset with augmentation, model training, validation, and inference. Depends on your needs, set your notebook's status as `Private/Public`.

b) Step 2: Adding input

In this project, we will need 2/3 different types of datasets:

- Raw dataset for training: Here i use `KITTI` but you can change to any other dataset you like, as long as it is still related to vehicles and traffic.
- Sample video for testing: Actual video of urban traffic, highway system or any video that containing vehicles.
- **(Optional)** My YOLOv8s fine-tune results on KITTI: This folder contains all vital stuffs you need (training results and validation results) to run demo immediately in order to save your precious time. In subfolder *`"Train"`*, there is **weights file** (`best.pt`), you can use it to continue training on the other datasets or making predictions on testing images/videos.

There are 2 options for you to add input (all three types) into your project: Using code or adding manually. For raw dataset (KITTI), it is better to take

advantage of available functionality of Kaggle, just use `"Add Input"` to find the dataset looks like figure 7 and press the plus button. We done here.

Fig. 7 KITTI Dataset

For sample testing video and YOLOv8s fine-tune results, if you want to reuse mine, it is easier when applying code for downloading straight forward to the project (I have set both as `Public`). For sample video:

```
import kagglehub

# Download latest version
path = kagglehub.dataset_download("einsamerwolf/sample-video-mp4")

print("Path to dataset files:", path)
```

And for YOLOv8s fine-tune results:

```
import kagglehub

# Download latest version
path = kagglehub.dataset_download("einsamerwolf/einsamerwolf/
yolov8s-results-on-kitti")

print("Path to dataset files:", path)
```

c) Step 3: Choosing hardware

- **NVIDIA Tesla P100:**

  The NVIDIA Tesla P100 is one of the earlier high-performance data center GPUs based on the Pascal architecture. In Kaggle, it is typically provided as a single GPU instance with 16 GB of HBM2 memory. The P100 excels in double-precision (FP64) and single-precision (FP32) performance, which makes it suitable for a wide range of scientific computing and deep learning tasks. With a memory bandwidth of 732 GB/s, the P100 allows efficient access to large datasets, reducing bottlenecks during training.

  It has 3584 CUDA cores, and performs best with moderately large models such as YOLOv5s/yolov5m, EfficientNet, or medium-sized transformers.

While the P100 does not offer Tensor Cores (which are available from the Volta architecture onward), it remains powerful for standard PyTorch or TensorFlow workloads that don't rely heavily on mixed-precision training. On Kaggle, the P100 delivers reliable and consistent performance for most deep learning tasks, although it is slower than newer generation GPUs when using large batch sizes or high-resolution inputs.

- **NVIDIA Tesla T4x2:**

  The Tesla T4x2 setup in Kaggle refers to a dual-GPU environment featuring two NVIDIA Tesla T4 GPUs, each with 16 GB of GDDR6 memory, providing a total of 32 GB across both GPUs. The T4 is based on the more recent Turing architecture and features Tensor Cores, which enable accelerated mixed-precision (FP16) computation — a major advantage for deep learning workloads using frameworks that support automatic mixed precision (AMP) like PyTorch and TensorFlow.

  Each T4 has 2560 CUDA cores and 320 Tensor Cores, offering efficient training and inference for deep learning models, especially when using FP16 precision. The T4x2 configuration supports multi-GPU parallelism, which is beneficial for training large models or large datasets more quickly, provided the user implements data parallelism correctly (e.g., using torch.nn.DataParallel or torch.distributed). Additionally, the T4 is optimized for energy efficiency and is often faster than the P100 in real-time inference and modern DL frameworks that leverage Tensor Core acceleration.

- **Summary and Use Case:**

  Use the P100 for simpler setups, standard training, and if your model does not leverage Tensor Cores or mixed precision.

  Use T4x2 for large models, multi-GPU training, and faster training with AMP (FP16) — ideal for recent architectures like YOLOv5/YOLOv8, Transformers, and diffusion models.

  For inference or training with very large batches, T4x2 can often outperform the P100 due to the double memory and parallel processing advantage.

d) Step 4: Start our notebook

After prepairing enough datasets, just starts session and runs cell by cell normally, or you can choose ***Run all*** mode and wait for the results. However, please remember these drawbacks of Kaggle while running notebook in order to avoid time-wasting and unwanted situations:

- Limited Session Time: GPU sessions (P100 or T4x2) are limited to 9 hours, and TPU sessions to even less. Sessions automatically disconnect after inactivity or timeout, which can interrupt long training runs unless checkpointing is implemented.
- No Persistent GPU Access: You only have 30 hours to access GPU if you do not link to Colab Pro. You cannot reserve or choose the GPU type in free-tier environments. Sometimes, you may get a CPU-only environment even when selecting GPU — and there's no retry queue.

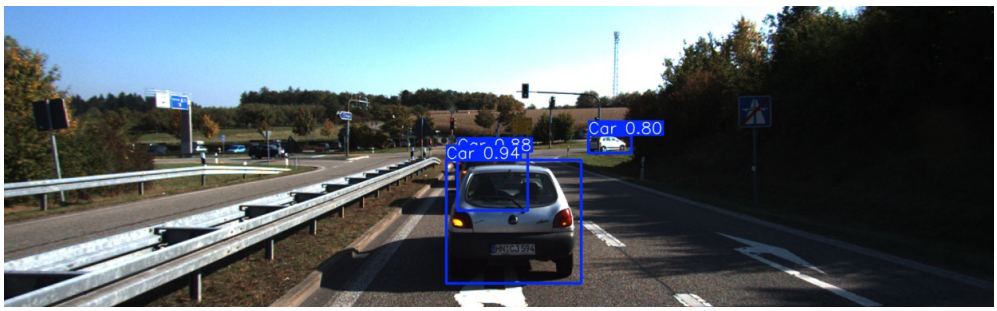
- No Real-Time Background Execution: Unlike services like Google Colab Pro or cloud VMs, Kaggle shuts down sessions when the notebook tab is closed, even if the kernel is running. There's no ability to run long-term background jobs or daemons.
- Storage and Disk I/O Constraints: While you get 20GB of temporary working space, it's erased after each session ends. Downloading and unpacking large datasets from Kaggle Datasets or external sources can be slow, and datasets must be reloaded each time unless you're using Kaggle Datasets directly.
- No Direct Internet Access: Kaggle notebooks do not have internet access by default for security reasons. You cannot use `pip install` from private repos, access online APIs, or download pre-trained weights/models from external sources unless they are Kaggle-hosted or manually uploaded.
- Lack of Full System Control: You cannot customize the environment deeply: kernel-level changes, driver updates, Docker usage, or installing system-level packages are restricted. Limited `apt-get` functionality makes it hard to install external dependencies like some CV libraries or external DB connectors.
- Limited TPU and Multi-GPU Control: Though T4x2 (2 GPUs) is available, parallelism is not automatic — you must manually code DataParallel. There's no official TPU support, unlike Google Colab.
- No Real-Time Logging or Monitoring Tools: You cannot use TensorBoard, WandB, or MLFlow easily for live logging or monitoring, especially due to the lack of persistent ports or background processes.
- Kernel Restarts and No Save-on-Crash: If your notebook crashes or restarts, unsaved outputs are lost. Long training jobs must implement manual saving (e.g., model.save() after each epoch), or progress will be lost.

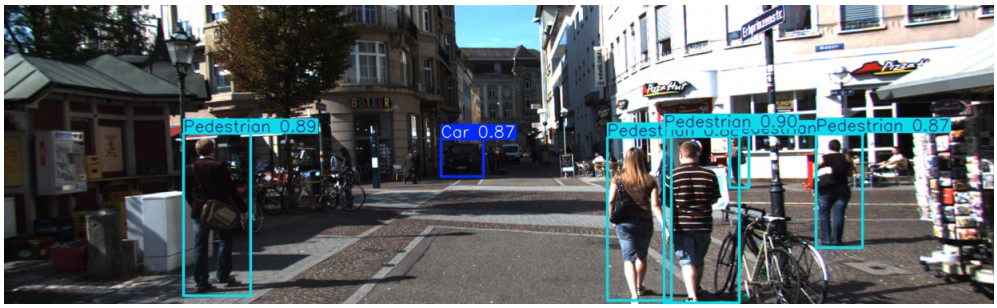
Kaggle is excellent for competitions, quick prototyping, and reproducible research, but it is not ideal for long training, production workflows, or complex environments. Users needing more control or reliability often move to Google Colab Pro, AWS EC2, Paperspace, or on-premise servers.

4.4 Experimental Results:

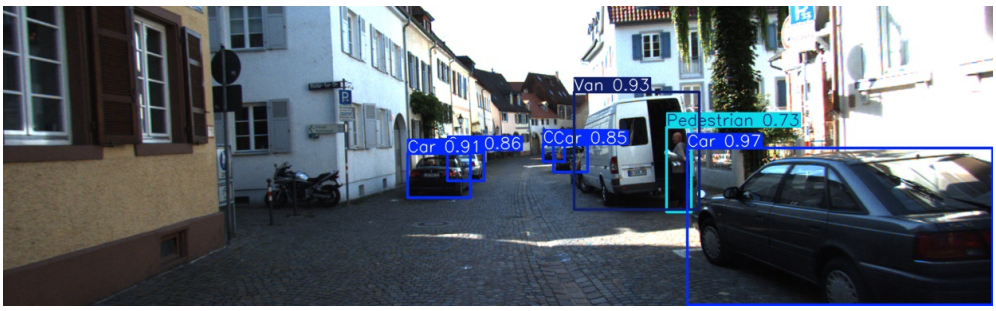
4.4.1 KITTI Dataset:

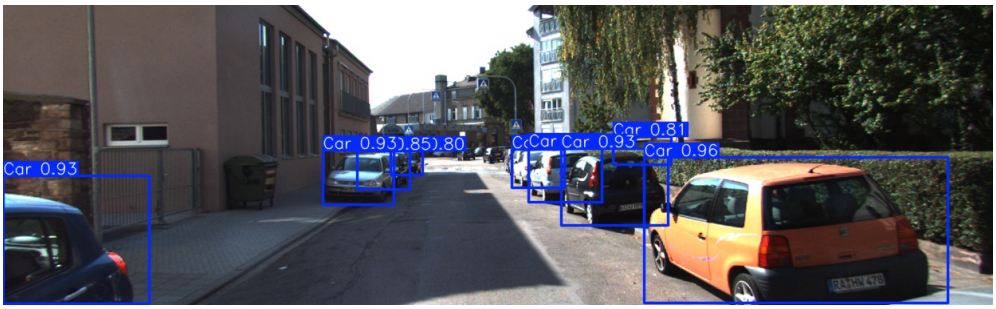
These are some images in validation set to test. It can be observed that the precision is pretty high.


Car 0.80
Car 0.94


Pedestrian 0.89
Car 0.87
Pedestrian 0.90
Pedestrian 0.87


Truck 0.90
Car 0.91
Van 0.96





4.4.2 Actual Urban Traffic Video:

This is the actual urban traffic video: https://drive.google.com/drive/folders/169QScIp2mRp5CdwmRJWyNOQXmqCHP6M8?usp=drive_link

4.5 How To Run Your Code:

This section provides step-by-step instructions for reproducing our results. All experiments are conducted using Kaggle's GPU-accelerated environment, and the codebase is compatible with Ultralytics YOLOv8 and ByteTrack implementations.

a) Setup Environment:

The code is designed to run in Kaggle Notebooks with the following libraries pre-installed:

- `ultralytics`
- `opencv-python`
- `pandas`, `numpy`, `matplotlib`, `seaborn`, `os`, `tqdm`, `albumentations`

If running locally, install seperate dependencies (not included in primal python version) using:

```
pip install ultralytics opencv-python
```

b) Download the Dataset:

On Kaggle, the dataset is automatically attached via Kaggle Datasets. Otherwise, download and prepare the KITTI dataset in YOLO format.

Just follow the instruction in **Subsection** 4.3 to create a complete datasets. I recommend you to use my datasets because i have prepair and edit the source code as suitable as possible.

c) Train the YOLOv8 Model:

After prepairing carefully, our progress will follow this pipeline:

YOLOv8 (Detection) →ByteTrack (Tracking) →Polygon Zone Checking (Counting) →Video Annotation (Output).

Remember that .yaml file format is very important for guiding model to train accurately. Make sure that your .yaml file containing the right directories to your training images/labels

```
from ultralytics import YOLO

model = YOLO("yolov8s.pt")
results = model.train(
    data=yaml_file_path,
    epochs=100,
    imgsz=640,
    device=0,
    patience=20,
    batch=32,
    optimizer='auto',
    lr0=0.001,
    lrf=0.1,
    seed=0
)
```

After training, the best weights will be saved in:
`runs/detect/train/weights/best.pt`

d) Run Inference and Counting with ByteTrack:

To perform tracking, labeling and counting using ByteTrack integrated with YOLOv8, you have to install:

```
!pip install supervision
!pip install ultralytics opencv-python labelme numpy
```

The script includes:

- Loading YOLOv8 model
- Performing frame-by-frame detection
- Applying ByteTrack to maintain consistent IDs
- Counting objects in real-time and total objects crossing virtual lines (polygon left/polygon right)

To ensure that the input images are clear and suitable for recognition, we implemented the function **apply_clahe_brightness(image)**. This function enhances the brightness and local contrast of the image using the CLAHE (Contrast Limited Adaptive Histogram Equalization) technique, applied only to the lightness channel (L-channel) to avoid altering the original colors.

```
def apply_clahe_brightness(image):
    lab = cv2.cvtColor(image, cv2.COLOR_BGR2LAB)
    l, a, b = cv2.split(lab)

    clahe = cv2.createCLAHE(clipLimit=3.0, tileGridSize=(8, 8))
    cl = clahe.apply(l)

    merged = cv2.merge((cl, a, b))
    return cv2.cvtColor(merged, cv2.COLOR_LAB2BGR)
```

The process described involves several steps for tracking and counting vehicles in a video using various libraries and techniques.

First, the video is loaded from a specified path, and the first frame is read to obtain the resolution and frames per second (FPS). A `VideoWriter` object is created to save the processed output in `.mp4` format. The libraries used include `cv2` (OpenCV) for image and video processing, `numpy` for handling coordinate arrays, **collections.deque** for storing vehicle movement trajectories, and `supervision.ByteTrack` for multi-object tracking.

Next, a fine-tuned YOLOv8s model is loaded and run on a GPU (CUDA) to accelerate the detection process. The goal is to detect vehicles, with the default class ID 0 representing `cars`. The model performs inference on each frame to predict bounding boxes for detected objects. Two predefined polygon zones are then set up to represent the left and right lanes, each with specific coordinates. These zones are visually outlined on the video using `cv2.polylines`, helping to visualize the designated counting areas.

For object tracking and counting, **ByteTrack** is employed to assign a unique ID to each detected vehicle across frames. A track memory list keeps track of each vehicle's center points to visualize their trajectories over time. Additionally, a lane record stores the ID of each vehicle along with the lane where it first appeared. Counters for the total vehicles entering the left and right lanes `(count_left and count_right)` are maintained. During each frame's processing loop, the model's detections are filtered to exclude non-car objects (objects with class ID not equal to 0) and detections with low confidence scores (less than 0.3). The detections are converted into the required format and passed to ByteTrack, which updates the positions and assigns IDs. The center point of each detection is then checked

against the predefined polygons using `cv2.pointPolygonTest`. If a vehicle enters a zone for the first time, the respective counter increases.

The code visualizes the tracking information by drawing bounding boxes, IDs, and direction labels—either **"IN"** if the vehicle is still inside the zone, or **"OUT"** if it has left. It also draws the movement trails from stored points in track memory. To optimize memory usage, IDs no longer present are removed from the track memory.

Finally, the program overlays statistical information, such as the total number of vehicles that have entered each lane and how many vehicles are currently inside each zone. Each processed frame is saved to the output video. Once all frames are processed, the resources are released with `cap.release()` and `out.release()`.

In many computer vision pipelines, especially those built within environments like Kaggle Notebooks or Jupyter, video outputs are commonly generated using the `cv2.VideoWriter` class from the OpenCV library. However, the videos created using this method often use default codecs such as `MJPG` or `XVID`, which may not be widely supported across all media players or browsers. This incompatibility can lead to playback errors, particularly when attempting to view the results directly on platforms like Kaggle or when downloading the video for review.

To resolve this issue, we employ the following `ffmpeg` command to convert the raw output video into a more universally compatible format:

```
!ffmpeg -y -i /kaggle/working/output_counted.mp4 -vcodec libx264 -
acodec aac output_fixed.mp4
```

This command re-encodes the video using the `libx264` codec for video and `aac` for audio. H.264 (libx264) is one of the most widely used and supported video codecs, ensuring compatibility across most browsers, video players (e.g., VLC, Windows Media Player), and web platforms (e.g., YouTube, GitHub, Kaggle).

In addition to enhancing compatibility, this conversion step helps correct structural issues that may arise during video generation. It is not uncommon for videos written by OpenCV to be malformed, missing headers, or even completely unreadable (e.g., 0-byte files). `ffmpeg` corrects these issues by remuxing and properly encoding the video, ensuring that the output is playable and structurally valid.

Another practical benefit of this conversion is file compression. Videos encoded with H.264 typically achieve better compression rates than MJPG or XVID, resulting in significantly smaller file sizes without compromising visual quality. This is particularly advantageous when storing or sharing videos in cloud-based environments with size restrictions.

In summary, converting the output video using `ffmpeg` is an essential post-processing step that improves compatibility, playback reliability, and storage efficiency. It ensures that results generated during evaluation and inference are accessible, portable, and easy to share or present.

e) Output Files:

The following outputs are generated:

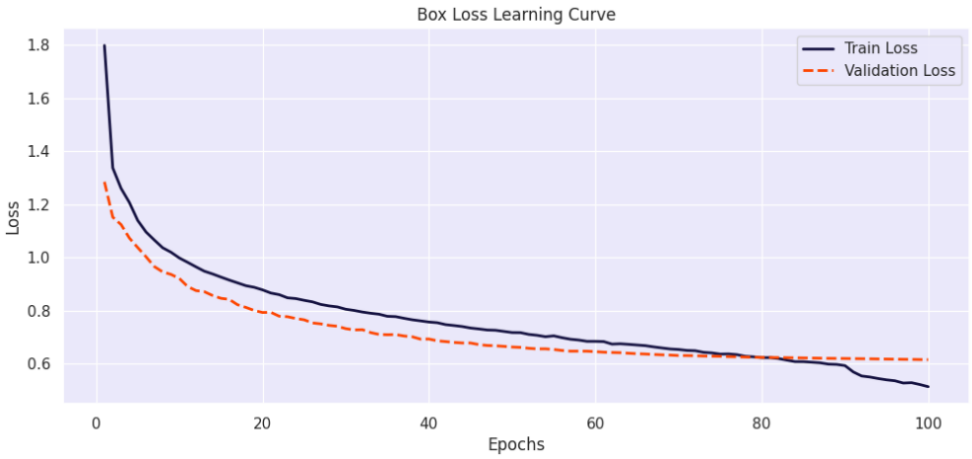
- Training and Validation results: `runs/detect/train and val/`
- Tracked video with annotations: `outputs/tracked_video.mp4`

- Count logs per frame: `outputs/count_log.csv`

f) Folder Structure:

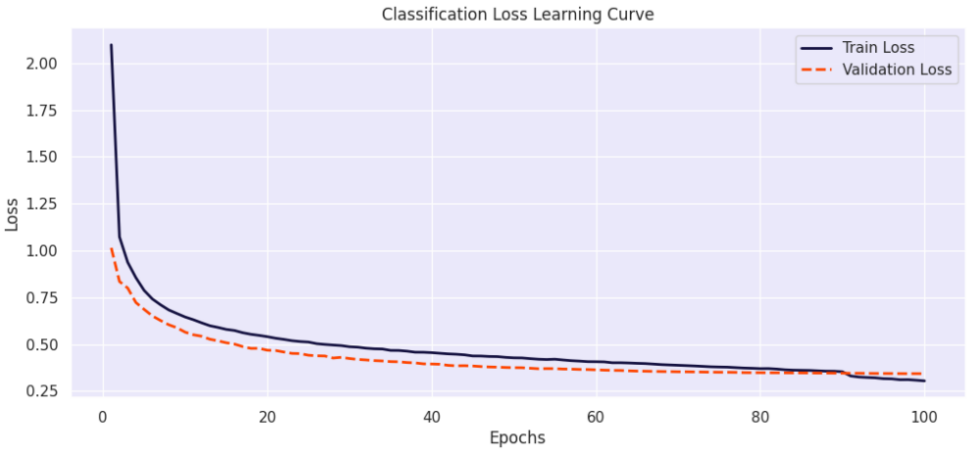
```
Kaggle/working
|
|-- kitti.yaml
|-- data_preprocessing (KITTI)/
|   |-- images/
|   |   |-- train/
|   |   |-- Val/
|   |-- labels/
|       |-- train/
|       |-- Val/
|-- runs/
|   |-- detect/
|       |-- train/
|       |-- Val/
|       |-- Predict/
|-- outputs/
    |-- tracked_video.mp4
    |-- count_log.csv
```

5 Evaluation

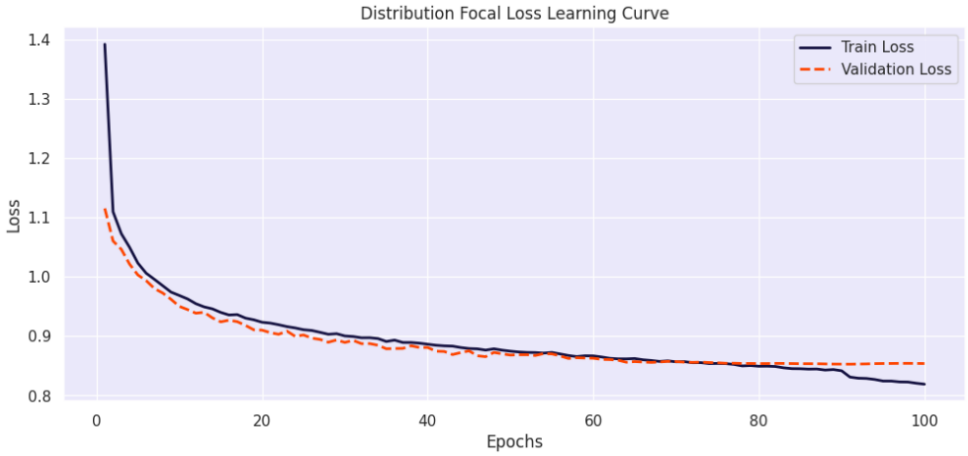


The Box Loss learning curve demonstrates the model's performance in bounding box regression over 100 epochs. Initially, the training loss starts at around 1.8 and

decreases sharply in the first few epochs, while the validation loss begins at approximately 1.3 and follows a similar downward trend. Both curves continue to decline steadily, with the training loss ending slightly below 0.55 and the validation loss around 0.6. The two curves remain close throughout the training process, suggesting stable learning and good generalization, with no significant overfitting, as the validation loss consistently tracks the training loss.



The learning curve shows the classification loss for both the training and validation sets over 100 epochs. At the start, the training loss is significantly higher, around 2.1, but it decreases rapidly within the first few epochs. The validation loss follows a similar trend, starting around 1.0 and also dropping quickly. As training progresses, both curves gradually converge, with the training loss ending slightly below 0.3 and the validation loss just above 0.3. The close alignment of the two curves throughout the process indicates good generalization, with no signs of overfitting, as the validation loss remains consistently close to the training loss.



The Distribution Focal Loss learning curve illustrates the model's performance in refining bounding box predictions over 100 epochs. The training loss starts at about 1.39 and decreases sharply in the first few epochs, while the validation loss begins around 1.11 and follows a similar trend. Both losses then gradually decline, converging closely as training progresses. By the final epochs, the training loss is slightly above 0.82, while the validation loss remains around 0.85. The consistent closeness of the two curves indicates stable learning and strong generalization, with no evident overfitting throughout the training process.

6 Conclusion

The YOLOv8 model, after fine-tuning on the KITTI dataset, demonstrates outstanding object detection performance. The results show a high precision of 0.939, reducing the number of false positive predictions, while a recall of 0.904 indicates the model's ability to detect most real-world objects. The value of mAP @ 0.5 of 0.948 reflects a strong recognition capacity at the average IoU threshold, and the mAP @ 0.5: 0.95 score of 0.769 shows that the model maintains high accuracy at multiple IoU levels. The overall score of 0.787 confirms that the model has been well trained and remains stable in the KITTI dataset. However, when applied to real-world videos, it is inevitable that some vehicles may be partially occluded, hindering detection, which could result in missed detections and potential inaccuracies in recognition and counting.

7 Future Development Directions

In the future, the model can be expanded and improved in the following directions:

- **Enhancing training data**: Incorporate images from various environments and weather conditions to improve generalization capability.

- **Optimizing the model for real-time performance**: Apply quantization or pruning techniques to reduce model size and increase inference speed, enabling deployment on edge devices.
- **Multi-model ensemble**: Using multiple YOLOv8 variants with different configurations to improve stability and accuracy.
- **Deployment in real-world environments**: Test the model on live traffic camera video streams to evaluate its performance outside the laboratory and in real world scenarios.
- **Using the model to detect lanes**: instead of manually drawing them improves both accuracy, saving time and efficiency in counting the number of vehicles in each lanes.

References

[1] Memon, S., Bhatti, S., Thebo, L.A., Talpur, M.M.B., Memon, M.A.: A video based vehicle detection, counting and classification system. I.J. Image, Graphics and Signal Processing **10**(9), 34–41 (2018)

[2] Lin, J., Yang, Y., Han, L., Zhang, M., Li, H.: A real-time vehicle counting, speed estimation, and classification system based on yolo object detection algorithm. Mathematical Problems in Engineering **2021**, 1–14 (2021) https://doi.org/10.1155/2021/9912534

[3] Ali, S.M., Usama, S.M., Asif, W., Rauf, A., Usama, M., Zubair, M., Anwar, S.: Smart traffic monitoring through pyramid pooling vehicle detection and filter-based tracking on aerial images. IEEE Access **10**, 108014–108026 (2022) https://doi.org/10.1109/ACCESS.2022.3212550

[4] Chughtai, B.R., Jalal, A.: Traffic surveillance system: Robust multiclass vehicle detection and classification. Unpublished Preprint, ResearchGate (2024)

[5] Geiger, A., Lenz, P., Urtasun, R.: Are we ready for autonomous driving? the kitti vision benchmark suite. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 3354–3361. IEEE, ??? (2012). https://doi.org/10.1109/CVPR.2012.6248074 . http://www.cvlibs.net/datasets/kitti/