Documentation

Lane Detection Using YOLO and Synthetic CARLA Data

Author:
Kamolov Makhmud

Budapest University of Technology and Economics
Department of Control for Transportation and Vehicle Systems

Course name:
Machine Vision

Date:
December 02, 2025

Contents

Chapter 1

Literature Review & Background

1.1 Introduction

Lane detection is one of the fundamental perception tasks in modern intelligent transportation systems and autonomous vehicles. Road lane markings provide essential information for maintaining the correct position of a vehicle within its lane, understanding road geometry, detecting curvature, and predicting the vehicle's safe driving corridor. Without reliable lane detection, many advanced driver assistance systems (ADAS) cannot function safely or effectively.

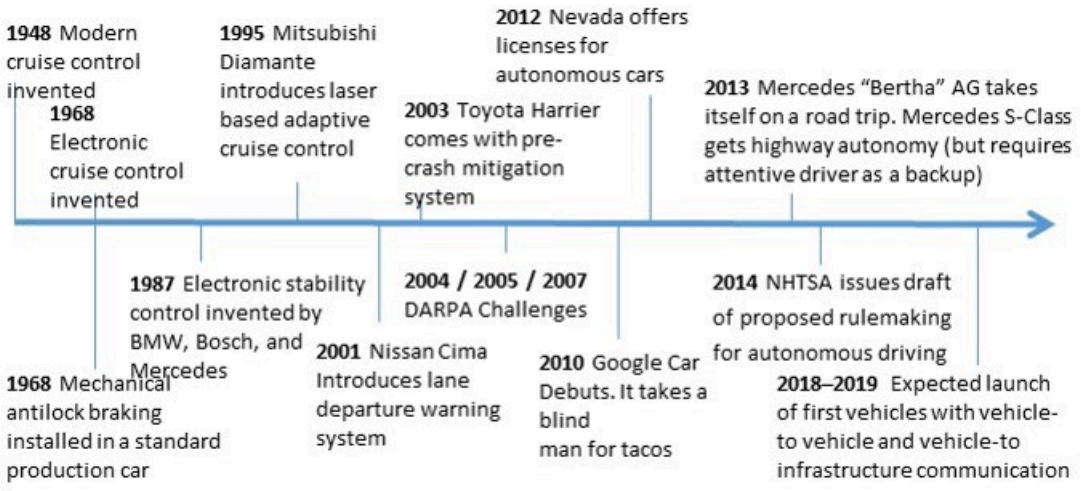

Figure 1.1: Historical Development of Autnomous Driving

Lane detection forms the basis of several widely used ADAS features. One of the most common is Lane Keeping Assist (LKA) [1], which continuously monitors the vehicle's position within the lane and applies small steering corrections to keep the vehicle centered. Another critical function is Lane Departure Warning (LDW) [2], which alerts the driver when the vehicle unintentionally drifts out of its lane, helping prevent accidents caused by drowsiness, distraction, or reduced visibility. In more sophisticated autonomous driving pipelines, lane information contributes to high-level perception and decision-making, supporting path planning, trajectory generation, and behavior prediction in complex road environments.

Figure 1.2: Lane Keeping Assist

Vision-based perception plays a central role in lane detection because cameras provide rich visual cues—such as lane paint, road texture, shadows, lighting conditions, and environmental context—that are difficult to capture with other sensors. Unlike LiDAR or radar, which detect geometric shapes or reflectivity, cameras can interpret subtle lane markings and distinguish them from other road features. As a result, vision is often the primary sensor modality for lane detection in both commercial ADAS systems and research-grade autonomous vehicles.

However, achieving robust lane detection is challenging due to varying lighting, weather conditions, worn-out lane markings, and occlusions from vehicles. This motivates the development of both traditional image-processing techniques and modern deep learning-based approaches, as well as the use of simulation environments—such as CARLA—to generate diverse training data and evaluate algorithms under controlled conditions. Lane detection therefore remains an active research area with high importance for improving road safety and enabling reliable autonomous driving.

1.2 Traditional Lane Detection Methods

Before the widespread adoption of deep learning, lane detection was primarily addressed using classical computer vision techniques. These methods rely on low-level image features such as edges, gradients, colors, and geometric shapes to identify lane markings on the road surface. Although simple and computationally efficient, traditional approaches often struggle in real-world driving environments due to their sensitivity to noise and varying visual conditions.

One of the earliest and most widely used methods is Canny edge detection, which extracts strong gradients in the image that potentially represent lane boundaries. The resulting edge map is then processed using the Hough transform, a geometric voting algorithm that identifies straight or slightly curved lines by detecting consistent edge patterns. Together, Canny and Hough form a classical pipeline for lane detection, especially on highways with clear and continuous markings.

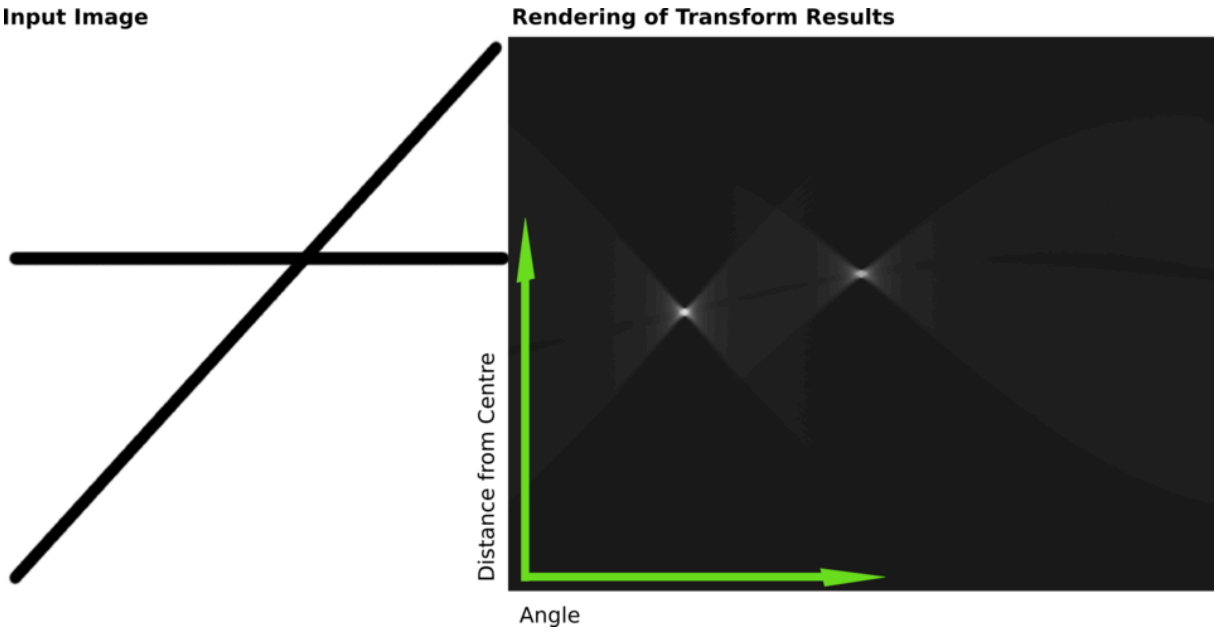

Figure 1.3: Hough Transform

Another group of techniques involves color thresholding, where specific color channels (typically white and yellow lane paints) are isolated using manual or adaptive thresholds. This helps remove irrelevant background pixels and highlight lane markings. After thresholding, morphological operations are often applied to clean the mask and remove noise.

To accommodate curved road segments, many classical systems apply polynomial fitting. After extracting candidate lane pixels, a second- or third-order polynomial is fitted to describe the lane geometry across the image. This allows the algorithm to predict lane curvature and estimate the vehicle's lateral position.

While these approaches are intuitive and fast, they suffer from several important limitations. Most classical methods assume high-contrast, well-defined lane markings, which is not always the case in real driving scenarios. Shadows from buildings, trees, and vehicles can significantly alter pixel intensities, causing edge detectors or thresholding methods to misinterpret shadow boundaries as lane lines. Sunlight reflections on wet or polished asphalt introduce bright streaks that resemble lane paint, confusing both edge- and color-based methods. Worn or faded lane markings reduce gradient strength and color consistency, often leading to missed detections. Adverse weather conditions—such as rain, fog, or nighttime driving—further degrade visibility, making it difficult for handcrafted features to reliably distinguish lanes from the background.

Due to these challenges, traditional vision techniques are increasingly being replaced by data-driven, learning-based approaches that can better handle variability and ambiguity in complex road environments.

1.3 Deep Learning-Based Lane Detection

With the rapid development of computer vision and autonomous driving technologies, deep learning has become the dominant approach for lane detection. Unlike traditional methods that rely on manually designed filters and geometric rules, deep learning models learn lane features directly from data. This allows them to capture complex patterns, adapt to diverse environments, and generalize more effectively across different road scenes.

Figure 1.4: Ultralytics YOLO icon.

One major family of approaches is object detection–based lane detection, where lane markings are treated as individual objects that the model must localize with bounding boxes. Modern detectors such as YOLO (You Only Look Once) operate in real time and can predict lane bounding boxes in a single forward pass. YOLO-based systems benefit from high prediction speed, robustness to noise, and the ability to detect multiple lane segments simultaneously. They also integrate contextual information from the entire image, helping them distinguish lane markings from shadows, road cracks, and other visual distractors. Although object detection does not provide pixel-level accuracy, it offers a lightweight and efficient solution for applications that require fast runtime, such as embedded ADAS systems.

A second major category involves semantic segmentation networks, which classify each pixel in the image as lane or non-lane. Models such as LaneNet, SCNN (Spatial Convolutional Neural Network), UNet, and more recent transformer-based architectures can produce highly detailed lane masks and accurately capture lane topology, even on curved or partially occluded roads. Segmentation-based methods typically achieve higher

accuracy than detection-based approaches but often require more computational resources and are slower during inference. Nonetheless, they are widely used in research environments and high-end autonomous driving systems where pixel-level information is critical.

Deep learning methods also enable the use of data augmentation and domain adaptation, allowing models to handle challenging conditions such as night driving, rain, fog, glare, and worn-out lane markings. Additionally, large-scale datasets—both real and synthetic—help networks learn features that remain stable across different road types, lighting conditions, and countries with varying lane styles.

Despite their strong performance, deep learning–based lane detectors rely heavily on diverse and high-quality training data. Collecting large labeled datasets from the real world is expensive and time-consuming, especially for pixel-level lane annotations. This challenge has encouraged the use of simulation environments such as CARLA, which can automatically generate perfectly labeled images for training and testing. Synthetic data allows researchers to simulate different weather, lighting, and traffic scenarios that might be difficult or unsafe to capture in real life.

Overall, deep learning has significantly advanced the state of lane detection, making it more reliable, adaptable, and scalable. By combining powerful neural architectures with synthetic data generation, modern systems achieve high robustness in complex conditions where traditional methods often fail.

1.4 Synthetic Data Generation with CARLA

A major challenge in developing robust lane detection systems is the need for large, diverse, and accurately labeled datasets. Real-world data collection is often time-consuming, expensive, and constrained by safety considerations. Moreover, manually labeling lane markings—especially at the pixel level—is extremely labor-intensive and prone to human error. These limitations have motivated researchers and engineers to explore synthetic data generation as an alternative source of training data.

Figure 1.5: CARLA Simulator logo.

The CARLA simulator (Car Learning to Act) provides a realistic, open-source virtual environment specifically designed for autonomous driving research. CARLA includes a wide variety of urban and rural road networks, detailed traffic scenarios, dynamic weather

conditions, and highly configurable sensor models. Because it runs on a deterministic rendering engine, CARLA can generate repeatable and customizable scenarios, which is ideal for creating large annotated datasets.

One of the key advantages of CARLA for lane detection research is its ability to produce automatic ground-truth labels. The simulator provides a semantic segmentation camera that assigns predefined class IDs to every pixel in the scene—for example, separating vehicles, pedestrians, sidewalks, and importantly, lane markings. These labels are generated internally by the engine and are perfectly accurate, overcoming the inconsistencies and ambiguities often present in human-annotated datasets. By extracting specific class IDs, such as the lane marking class, researchers can obtain clean binary masks that highlight lane pixels without any need for manual labeling.

CARLA also allows the creation of diverse training conditions that are difficult to capture consistently in the real world. Users can vary time of day, sun angle, cloudiness, rain intensity, wetness of the road, fog density, and traffic density. These variations are critical for building lane detection models that generalize well to challenging scenarios, such as nighttime driving, low-visibility weather, occlusions from vehicles, and shadows from buildings or trees. Synthetic data can therefore help overcome the domain gap that often limits traditional computer vision and machine learning approaches.

Furthermore, CARLA enables synchronized data collection from multiple sensors. Combining an RGB camera with a semantic segmentation camera allows the generation of paired datasets: for each RGB frame, there is a corresponding ground-truth lane mask. This pairing is essential for training supervised learning models, including detection-based networks such as YOLO or segmentation-based architectures like UNet or LaneNet.

Another important benefit of synthetic data is scalability. Large real-world datasets such as TuSimple, CULane, or BDD100K contain tens of thousands of samples, but generating similarly large datasets in CARLA is significantly faster and cost-effective. Automated scripts can generate thousands of images with diverse, random scenarios in minutes, making it easy to expand the dataset or focus on specific rare events.

In summary, synthetic data generation using CARLA addresses many of the challenges associated with real-world lane detection datasets. It provides accurate labels, diverse environmental conditions, reproducibility, cost efficiency, and seamless sensor synchronization. These advantages make CARLA a powerful tool for training, validating, and evaluating modern lane detection algorithms.

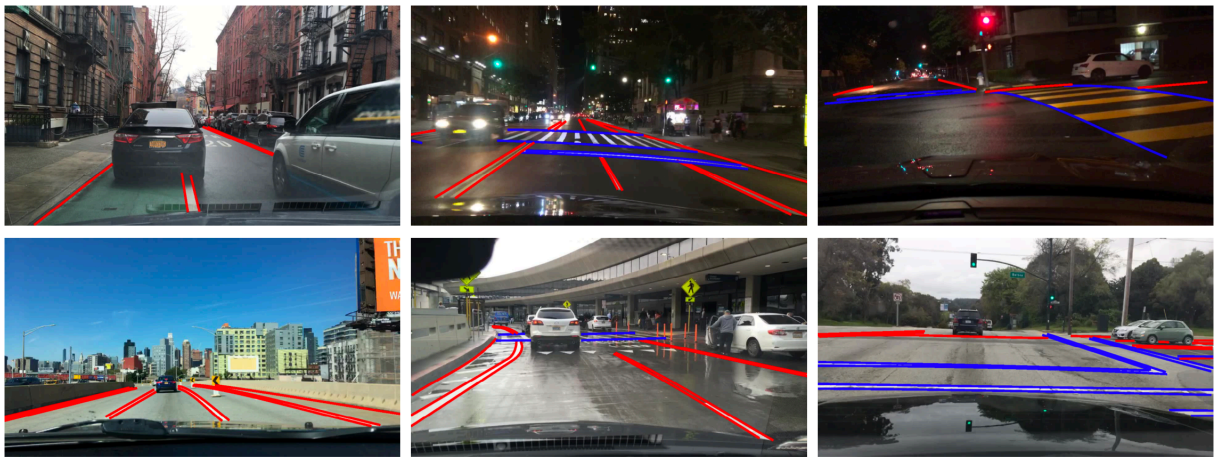

Figure 1.6: BDD100K Lane Markings

1.5 Summary of Related Works

Lane detection has been an active research topic for more than two decades, evolving significantly as computer vision and deep learning technologies have advanced. Early studies primarily focused on handcrafted feature extraction, relying on gradient-based edge detection and geometric constraints to identify lane markings. For example, many classical approaches combined Canny edge detection with the Hough transform to extract linear lane features on highways. While effective in controlled environments, these methods were highly sensitive to lighting variations, faded markings, and complex urban scenes.

As the limitations of traditional vision techniques became evident, researchers began exploring machine learning–based methods that could adapt to varying conditions. Before the deep learning era, some approaches incorporated probabilistic models, such as Kalman filters or particle filters, to track lanes across consecutive frames. These models improved temporal consistency but still depended heavily on handcrafted image features.

The introduction of convolutional neural networks (CNNs) marked a significant turning point. Early deep learning approaches such as LaneNet introduced pixel-wise lane segmentation using encoder–decoder architectures. This was followed by methods like SCNN (Spatial Convolutional Neural Network), which leveraged vertical and horizontal message passing to capture the elongated, thin structure of lane markings. These models demonstrated substantial improvements over traditional techniques, especially in challenging conditions such as shadows, occlusions, and curved roads.

More recent research has explored transformer-based architectures and multi-task learning frameworks. Models such as Ultra Fast Lane Detection (UFLD) and CondLaneNet emphasize real-time performance by optimizing network architectures to run efficiently on embedded hardware. Meanwhile, transformer-based models incorporate

global attention mechanisms, enabling them to reason about long-range dependencies, which is particularly useful for detecting lanes on complex road geometries.

Parallel to the advancement of deep learning algorithms, synthetic data has gained increasing attention as a mechanism for boosting model robustness and reducing reliance on costly real-world annotation. Studies leveraging simulators like CARLA, AirSim, and LGSVL show that synthetic images can effectively train models or serve as supplementary data to enhance performance under rare or hazardous scenarios. Domain randomization techniques have been explored to improve generalization when transferring models trained on synthetic data to real-world environments.

In the context of object detection–based methods, variants of YOLO have also been applied to lane detection by treating lane segments as detection targets. Although this approach is not as precise as segmentation-based methods, it offers real-time performance and simpler annotation requirements, making it suitable for lightweight ADAS applications. Research has demonstrated that bounding-box–based lane detectors can effectively capture lane presence, lane width, and position when trained with sufficiently diverse data, particularly when combined with synthetic datasets.

Overall, the evolution of lane detection research reflects a shift from handcrafted vision algorithms toward data-driven deep learning models capable of handling complex and diverse road environments. Recent studies also highlight the growing importance of simulation platforms like CARLA to address dataset limitations, generate controlled variations, and facilitate reproducible experimentation. These trends form the foundation and motivation for the lane detection approach implemented in this project.

Chapter 2

Detailed Documentation of Code

2.1 Overview of System Architecture

The overall system developed in this project follows a structured pipeline designed to generate synthetic training data from the CARLA simulator, convert the collected information into a usable machine-learning dataset, and train a YOLO-based lane-marking detector. The architecture consists of three major components: the simulation environment, the dataset generation module, and the machine-learning training module. Each component communicates through well-defined file structures and synchronized data-flow steps to ensure accurate and reproducible results.

The first component of the system is the CARLA simulation environment, which provides a controllable virtual world containing multiple towns, diverse weather conditions, and dynamic traffic scenarios. The system spawns an ego vehicle equipped with two sensors: a forward-facing RGB camera and a semantic segmentation camera. Both sensors are attached to the same vehicle using identical spatial transforms to guarantee perfect pixel alignment. The world is executed in synchronous mode, meaning each simulation tick generates one pair of RGB and semantic frames. This deterministic execution ensures that the two cameras capture the same scene at precisely the same moment. The simulation module also monitors the ego vehicle's speed, and data is recorded only when the vehicle is moving above a predefined threshold, preventing duplicate images during stop phases or low-speed idle states.

The second component is the dataset generation subsystem, implemented in Python using CARLA's client API, OpenCV, and standard file operations. RGB images are stored in a dedicated directory, while semantic segmentation frames are post-processed to extract lane-marking pixels based on the class ID associated with lane markings in CARLA's semantic labeling. The system can generate either bounding-box labels for YOLO detection or pixel-wise masks for segmentation models. In the case of YOLO training, lane masks are binarized, and connected components are extracted to compute bounding boxes in YOLO's normalized `(x_center, y_center, width, height)` format. The RGB images and their corresponding label files are then organized into the folder structure expected by the Ultralytics YOLO framework, with separate training and validation splits.

The final component is the YOLO training module, which operates independently of CARLA. The generated dataset is loaded into the Ultralytics YOLO training interface, where the model learns to detect lane-marking segments directly from the synthetic data.

Training results, including model weights, training curves, confusion matrices, precision–recall plots, and validation metrics (mAP, precision, recall), are saved to persistent storage for later evaluation. This modular design allows the model to be retrained or extended without re-running the simulation, enabling rapid experimentation and efficient iteration.

Together, these components form a complete system architecture: CARLA provides high-quality synthetic visual data, the dataset generation module converts this data into a structured machine-learning format, and the YOLO training module produces a real-time lane-detection model. The separation of responsibilities between simulation, data processing, and model training ensures flexibility, reproducibility, and clarity in the overall workflow.

2.2 Detailed Documentation of Data Creation Module (*dataset_ creation_ substuff.py*)

This section provides a complete explanation of the data generation subsystem used for creating RGB images and lane masks from the CARLA simulator. The module is responsible for initializing the simulation environment, configuring sensors, synchronizing data acquisition, and storing processed images. All descriptions in this section refer to the code contained in *dataset_ creation_ substuff.py*.

2.2.1 Module overview

The data-creation subsystem is implemented using Python, CARLA's Python API, NumPy, and OpenCV. Main purpose of this file is to organize some basic functions and make *data_ creation.py* more clean. It is designed to:

- Connect to a running CARLA simulation instance
- Configure the world (map, weather, traffic manager, synchronous mode)
- Spawn an ego vehicle
- Attach an RGB camera and a semantic segmentation camera
- Capture synchronized frames
- Extract lane markings from semantic labels
- Save aligned RGB + lane-mask image pairs for dataset creation

The system is designed to run deterministically using synchronous mode, ensuring that RGB and semantic frames correspond to the same simulation tick.

2.2.2 Global Variables and Imports

The module begins by importing required libraries:

- `cv2` and `numpy` for image processing
- `carla` for simulation
- `hashlib` it was used for debugging purposes (unused at the moment)

It also initializes global variables such as `last_hash`, `count_rgb`, `img_hash`, and `creating`, which control sampling frequency and file-saving logic. These variables are used to ensure that images are not saved too frequently.

2.2.3 Connecting to the CARLA Server

The system connects to the CARLA simulator via:

```
client = carla.Client("localhost", 2000)
client.set_timeout(10.0)
```

This creates a TCP connection to an already running CARLA UE4 instance. The timeout ensures that the system does not freeze when the simulator is temporarily unresponsive.

2.2.4 Loading the World and Weather Profiles

The module selects **Town10HD** and loads it:

```
town_name = 'Town10HD'
world = client.load_world(town_name)
static_weathers = [ carla.WeatherParameters.ClearNoon,
                    carla.WeatherParameters.CloudyNoon,
                    carla.WeatherParameters.WetNoon,
                    carla.WeatherParameters.WetCloudyNoon,
                    carla.WeatherParameters.MidRainyNoon,
                    carla.WeatherParameters.HardRainNoon,
                    carla.WeatherParameters.SoftRainNoon,
                    carla.WeatherParameters.ClearSunset,
                    carla.WeatherParameters.CloudySunset,
                    carla.WeatherParameters.WetSunset,
                    carla.WeatherParameters.WetCloudySunset,
                    carla.WeatherParameters.MidRainSunset,
                    carla.WeatherParameters.HardRainSunset,
                    carla.WeatherParameters.SoftRainSunset]
```

It also defines **14 different weather presets**, including clear, cloudy, rainy, wet, and sunset conditions. These will later be cycled to generate diverse datasets.

2.2.5 Configuring Traffic Manager and Synchronous Mode

To generate deterministic image pairs, the world is placed in synchronous mode:

```
settings = world.get_settings()
settings.synchronous_mode = True
settings.fixed_delta_seconds = 0.05
world.apply_settings(settings)
```

This ensures:

- Each `world.tick()` generation produces exactly one frame
- Sensors produce synchronized RGB and semantic images
- The simulation advances only when controlled by the script

The Traffic Manager (TM) is configured with:

```
tm.set_synchronous_mode(True)
tm.set_global_distance_to_leading_vehicle(1.0)
tm.global_percentage_speed_difference(0)
```

This makes traffic predictable and consistent.

2.2.6 Blueprint Library and Spawn Points

The module loads the blueprint library and extracts spawn points:

```
blueprint_library = world.get_blueprint_library()
spawn_points = world.get_map().get_spawn_points()
```

These are used by the main dataset-creation script to spawn the ego vehicle and background traffic actors.

2.2.7 Camera Sensor Configuration

Two camera blueprints are prepared:

RGB Camera

```
cmr_rgb = blueprint_library.find('sensor.camera.rgb')
cmr_rgb.set_attribute('image_size_x', '800')
cmr_rgb.set_attribute('image_size_y', '600')
cmr_rgb.set_attribute('fov', '90')
```

Semantic Segmentation Camera

```
bp_seg =
blueprint_library.find('sensor.camera.semantic_segmentation')
bp_seg.set_attribute('image_size_x', '800')
bp_seg.set_attribute('image_size_y', '600')
```

```
bp_seg.set_attribute('fov', '90')
```

Both sensors are configured with the same resolution and field-of-view to ensure pixel-perfect alignment.

Camera Transform

Both sensors share the same mounting position:

```
relative_transform = carla.Transform(
    carla.Location(x=1.5, z=1.7),
    carla.Rotation(pitch=0)
)
```

This places the camera slightly above and ahead of the ego vehicle, producing a realistic driver-view perspective.

2.2.8 Ego Vehicle Blueprint

The Lincoln MKZ model is used as the ego vehicle:

```
ego_bp = blueprint_library.find('vehicle.lincoln.mkz_2020')
ego_bp.set_attribute('role_name', 'hero')
```

2.2.9 RGB Image Saving Function

The `save_image` function handles RGB frame storage:

```
def save_image(image, image_numbers, weather_number):
    global count_rgb
    array = np.frombuffer(image.raw_data, dtype=np.uint8)
    array = np.reshape(array, (image.height, image.width, 4))
    rgb = array[:, :, :3]
    cv2.imshow("CARLA Camera View", rgb)
    cv2.waitKey(1)
    count_rgb += 1
    if count_rgb == 20 and image_numbers < 100 and creating:
        img = np.frombuffer(image.raw_data,
dtype=np.uint8).reshape(image.height, image.width, 4)
        filename = f"dataset/rgb/{town_name}_{weather_number}
_{image_numbers}.png"
        cv2.imwrite(filename, img)
        print("image", image_numbers)
```

Key behaviors:

- Converts CARLA's BGRA buffer into a NumPy array
- Displays the live camera feed for debugging
- Increments count_rgb each frame

- Saves RGB images only when count_rgb == 20 → avoids saving 20 frames per second

Filename pattern:

```
dataset/rgb/{town_name}_{weather_number}_{image_numbers}.png
```

This ensures each image is associated with its town and weather condition.

2.2.10 Semantic Segmentation Callback

This function extracts lane pixels:

```
semantic_map = img_array[:, :, 2]  # R channel contains class ID
lane_id = 24
lane_mask = (semantic_map == lane_id).astype(np.uint8) * 255
```

CARLA encodes semantic class IDs in the **red channel** of the BGRA image. Lane marking ID = 24.

The result is a clean binary mask where:

- 255 = lane pixel
- 0 = background

The saving condition matches the RGB camera:

```
if count_rgb == 20 and image_numbers < 100 and creating:
```

This guarantees matching RGB/seg pairs.

Output file:

```
dataset/seg/{town_name}_{weather_number}_{image_numbers}.png
```

2.2.11 Summary of Data Flow

Input

- CARLA world
- RGB camera data
- Semantic segmentation camera data
- Weather and map parameters

Processing

- Synchronization using fixed ticks
- Extraction of lane pixels from semantic data

- Frame rate reduction using counter
- Weather iteration and map cycling

Output

```
/dataset/rgb/*.png #  - raw visual images
/dataset/seg/*.png #  - binary lane masks
```

Each image pair is perfectly aligned and used later to generate YOLO labels or segmentation training masks.

2.3 Detailed Documentation of the Main Data Creation Script (*dataset_ creation.py*)

This section describes the main data-collection script responsible for running the CARLA simulation, synchronizing sensor data, filtering frames based on vehicle motion, and saving aligned RGB–semantic image pairs. The explanations refer to the contents of *dataset_ creation.py*.

2.3.1 Purpose of the Script

While *dataset_ creation_ substuff.py* contains reusable components such as blueprints, camera configurations, and image-processing utilities, the main script presented here orchestrates the full data generation pipeline. Its purposes are:

1. Start and control the simulation loop
1. Spawn background traffic and the ego vehicle
1. Attach and synchronize sensors
1. Cycle through selected weather conditions
1. Save perfectly aligned RGB + semantic frames only when the vehicle is moving
1. Produce 100 paired images per weather condition

This script acts as the "engine" of the dataset-generation system.

2.3.2 Loading System Components

The script imports key variables and functions from the helper module:

```
from dataset_creation_substuff import (
    world, tm, settings, blueprint_library, tm_port, spawn_points,
```

```
    relative_transform, ego_bp, cmr_rgb, save_image, bp_seg,
    semantic_callback, static_weathers
)
```

These include:

- The CARLA world object
- Traffic Manager configuration
- Vehicle and sensor blueprints
- Predefined relative camera transforms
- Weather presets
- Image saving callbacks

This modular design isolates configuration from runtime logic.

2.3.3 Spawning Background Traffic

Up to ten random vehicles are spawned using:

```
actors = []
vehicle_blueprints = blueprint_library.filter('*vehicle*')
for i in range(10):
    v = world.try_spawn_actor(random.choice(vehicle_blueprints),
random.choice(spawn_points))
    if v:
        actors.append(v)
```

Each vehicle is assigned a random spawn point. Successfully spawned vehicles are stored in `actors` for later cleanup. This background traffic adds realism to the dataset by introducing diverse occlusions and scene compositions.

2.3.4 Spawning the Ego Vehicle

The ego vehicle is spawned from its blueprint:

```
while True:
    ego_vehicle = world.try_spawn_actor(ego_bp,
random.choice(spawn_points))
    if ego_vehicle:
        actors.append(ego_vehicle)
        break
```

A loop ensures that a valid spawn point is found. The ego vehicle is included in the `actors` list for destruction during cleanup.

2.3.5 Enabling Autopilot for All Actors

All spawned vehicles are placed in autopilot mode:

```
for v in actors:
    v.set_autopilot(True, tm_port)
```

The Traffic Manager is already configured for synchronous mode, ensuring consistent and deterministic behavior during simulation.

2.3.6 Sensor Initialization and Queues

The script uses Python queues to guarantee **strict frame-level synchronization:**

```
rgb_queue = queue.Queue()
segm_queue = queue.Queue()
```

Two sensors are attached to the ego vehicle:

RGB Camera

```
camera_rgb = world.spawn_actor(cmr_rgb, relative_transform,
attach_to=ego_vehicle)
camera_rgb.listen(rgb_queue.put)
```

Semantic Segmentation Camera

```
cam_seg = world.spawn_actor(bp_seg, relative_transform,
attach_to=ego_vehicle)
cam_seg.listen(segm_queue.put)
```

Each camera pushes frames into its respective queue. Using queues ensures that the main loop processes one RGB frame paired with exactly one semantic frame.

2.3.7 Main Simulation Loop

The outer loop iterates through every predefined weather condition:

```
for i in static_weathers:
    world.set_weather(i)
```

For each weather profile, the system collects up to 100 synchronized image pairs.

2.3.8 Synchronous Tick and Frame Retrieval

Inside the weather loop:

```
world.tick()
rgb_image = rgb_queue.get()
seg_image = segm_queue.get()
```

Because the world is in synchronous mode, `world.tick()` guarantees that both cameras capture frames at the same simulation time step.

Frame Synchronization Check

```
if rgb_image.frame != seg_image.frame:
    print(f"Frame mismatch: RGB {rgb_image.frame}, SEG {seg_image.frame}")
    continue
```

If the two sensors produce mismatched frame IDs, the frame is discarded. This prevents misaligned RGB–mask pairs and ensures dataset integrity.

2.3.9 Vehicle Speed Filtering

To avoid saving identical images when the ego vehicle is stopped or crawling slowly, the script computes vehicle speed:

```
vel = ego_vehicle.get_velocity()
speed_ms = math.sqrt(vel.x**2 + vel.y**2 + vel.z**2)
speed_kmh = speed_ms * 3.6
```

Only frames captured at speed ≥ 1 km/h are accepted:

```
if speed_kmh < MIN_SPEED_KMH:
    continue
```

This prevents dataset redundancy and ensures more diverse driving scenes.

2.3.10 Saving RGB and Semantic Images

When the vehicle is moving, the script saves:

RGB Camera

```
save_image(rgb_image, image_numbers, a)
```

Where:

- `image_numbers` indexes the current dataset segment
- `a` identifies the weather condition

Semantic Mask Image

```
if semantic_callback(seg_image, image_numbers, a):
    image_numbers += 1
```

The callback returns `True` when a corresponding lane mask has been successfully saved, ensuring the RGB and mask remain synchronized.

2.3.11 Weather Switching Logic

After capturing 100 pairs for the current weather:

```
if image_numbers == 100:
```

```
        image_numbers = 0
        break
```

The script resets the counter and moves on to the next weather preset.

2.3.12 Cleanup and Graceful Shutdown

The `finally` block ensures that all actors and sensors are safely destroyed:

```
camera_rgb.stop(); camera_rgb.destroy()
cam_seg.stop(); cam_seg.destroy()
for a in actors: a.destroy()
```

The world and Traffic Manager are returned to asynchronous mode:

```
tm.set_synchronous_mode(False)
settings.synchronous_mode = False
world.apply_settings(settings)
```

All OpenCV windows are closed.

2.3.13 Summary of System Behavior

dataset_ creation.py coordinates the entire data generation pipeline:

1. Initializes simulation and sensors
1. Enforces strict synchronization via tick-based execution and frame matching
1. Filters out idle frames using real vehicle speed
1. Saves clean RGB + lane-mask pairs
1. Cycles through multiple weather types
1. Produces consistent datasets across environments

This script, together with the helper module, creates a complete and reproducible dataset generation system suitable for training both detection- and segmentation-based lane detection models.

2.4 Google Colab Training Pipeline (*Machine_ Vision_ HW.ipynb*)

This section documents the full model-training workflow implemented inside the Google Colab environment. The notebook *Machine_ Vision_ HW.ipynb* contains the complete process used to prepare the dataset, generate YOLO-format labels, configure training parameters, and execute training on GPU.

The purpose of this section is to describe how the data generated in CARLA is transformed into a usable format and how the YOLOv8 model is trained in a cloud-based environment.

2.4.1 Motivation for Using Google Colab

Google Colab provides:

- A free NVIDIA T4 GPU with 16 GB VRAM
- Direct access to Google Drive for persistent storage
- A controlled environment where all dependencies can be installed
- Protection against local system limitations (VRAM, CPU, disk space)

Because the synthetic dataset contains thousands of images, Colab enables faster training and reliable storage of trained model files.

2.4.2 Dataset Upload and Extraction

The first section of the notebook navigates to the working directory and extracts the uploaded `.zip` file containing RGB images:

```
%cd /content
!unzip "/content/drive/MyDrive/rgb.zip" -d dataset_raw
```

This creates:

```
dataset_raw/
   rgb/
   seg/   (after label-generation step)
```

This mirrors the folder structure produced by the CARLA data-creation scripts.

2.4.3 Automatic YOLO Label Generation

The notebook includes a custom label-generator function (Cell 1) which:

1. Reads each RGB image
1. Finds the corresponding semantic mask
1. Extracts all lane-marking components
1. Computes bounding boxes around lane segments
1. Saves `.txt` annotation files in YOLO format

The key steps are:

```
img = cv2.imread(img_path)
mask = cv2.imread(mask_path, cv2.IMREAD_GRAYSCALE)
mask = (mask > 0).astype("uint8") * 255      # binary lane mask
contours, _ = cv2.findContours(...)
x, y, w, h = cv2.boundingRect(cnt)
```

Once the bounding boxes are normalized, the annotations are written into the standard YOLO format:

```
<class_id> <x_center> <y_center> <width> <height>
```

The label generator is executed with:

```
if __name__ == "__main__":
    create_labels()
```

This fully automates label production for thousands of frames.

2.4.4 Train–Validation Split

To train YOLO properly, the dataset must be split into two subsets. The notebook implements an 80/20 split:

```
train_imgs = images[:train_split]
val_imgs   = images[train_split:]
```

Images and labels are copied to their respective folders:

```
dataset/
   images/train/
   images/val/
   labels/train/
   labels/val/
```

This matches Ultralytics YOLO's required directory layout.

2.4.5 Creating the Dataset Configuration (.yaml)

A custom configuration file is generated dynamically:

```
yaml_text = f"""
path: {DATASET_ROOT}

train: images/train
val: images/val

names:
  0: lane_marking
"""
```

YOLO uses this YAML file to locate training images, validation images, and class names.

2.4.6 Installing Ultralytics and Initializing YOLOv8

The notebook installs the Ultralytics framework:

```
!pip install -q ultralytics
from ultralytics import YOLO
```

This version includes YOLOv8-nano, YOLOv8-small, and other pre-trained models.

2.4.7 Training the YOLO Model

Training is executed directly in Google Drive so results persist even if the session disconnects:

```
!yolo detect train \
  data=carla_lane.yaml \
  model=yolov8s.pt \
  epochs=40 \
  imgsz=512 \
  batch=16 \
  device=0 \
  project=PROJECT_DIR \
  name="exp_t4" \
  exist_ok=True
```

Key configuration details:

- **model = yolov8s.pt** A small and efficient architecture suitable for real-time lane-mark detection.
- **epochs = 40** Enough for convergence given synthetic data quality.
- **batch = 16** Fits into the Colab T4's VRAM while maximizing throughput.
- **project = PROJECT_DIR** (Google Drive) Ensures that best.pt, last.pt, metrics, plots, and logs are permanently stored.

During training, YOLO outputs:

- loss curves
- mAP metrics
- precision–recall charts
- confusion matrix
- validation predictions

All of these are saved automatically.

2.4.8 Running Inference on Validation Data

To visually verify that the model detects lane markings:

```
!yolo detect predict \
  model=/content/drive/MyDrive/carla_lane_yolo/exp_t4/weights/best.pt \
  source=/content/dataset/images/val \
  conf=0.3 \
  save=True \
  project=/content/drive/MyDrive/carla_lane_yolo \
  name=exp_t4_preds \
  exist_ok=True
```

This produces annotated images in:

```
/MyDrive/carla_lane_yolo/exp_t4_preds/
```

Each image contains bounding boxes around detected lane segments, confirming that the model is functioning correctly.

2.4.9 Summary of the Colab Pipeline

The Google Colab notebook handles the entire machine-learning stage of the project:

1. Imports and decompresses the CARLA-generated dataset
1. Generates YOLO label files from semantic lane masks
1. Splits data into training and validation sets
1. Creates YOLO configuration (YAML)
1. Installs and initializes YOLOv8
1. Trains the detector directly on GPU
1. Saves model weights and training statistics into Google Drive
1. Runs inference to visualize detection results

This workflow complements the data-creation scripts by providing a reliable, automated environment for training and evaluation.

Chapter 3

Results

This section presents the outcomes of the lane-detection pipeline, including the results of the dataset-generation process, the YOLOv8 training performance, and the visual evaluation of the model's predictions. The objective of this section is to demonstrate that the system successfully detects lane markings across multiple towns, weather settings, and viewpoints generated in the CARLA simulator.

3.1 Dataset Generation Results

Using the scripts described in Section 2, a total of 8,400 image pairs were collected from six CARLA towns under fourteen different weather conditions. Each sample contains:

- a high-resolution RGB image (800×600),
- a corresponding semantic lane mask, and
- a set of automatically generated YOLO bounding-box annotations.

The synchronization mechanism ensured that every RGB frame corresponded to the exact semantic frame of the same simulation tick.

Vehicle-speed filtering effectively removed idle frames, ensuring the dataset contained only meaningful driving scenes.

Figure samples below demonstrate correct alignment between RGB images and binary lane masks.

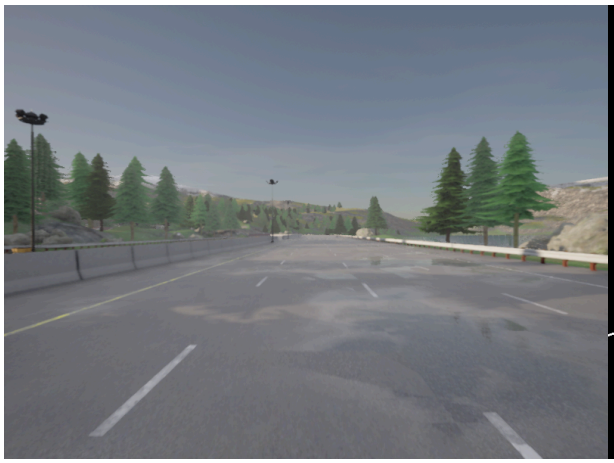

Figure 3.1: Town04_9_77.png RGB

Figure 3.2: Town04_9_77.png Lane Mask

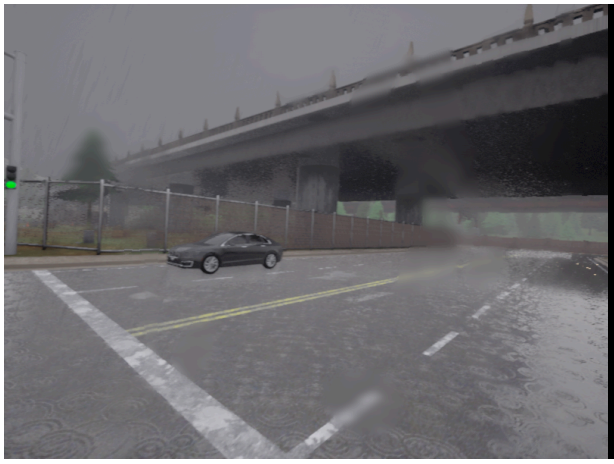

Figure 3.3: Town05_12_32.png RGB

Figure 3.4: Town05_12_32.png Lane Mask

3.2 YOLO Training Results

The YOLOv8s model was trained in Google Colab using an NVIDIA T4 GPU.
The final training configuration consisted of:

- **Model:** YOLOv8s (pretrained on COCO)
- **Epochs:** 40
- **Batch size:** 16
- **Image size:** 512×512
- **Optimizer:** SGD with warm-up
- **Dataset:** Custom CARLA lane dataset (train/val split: 80/20)

All model outputs—including weights, loss curves, and performance plots—were automatically saved to Google Drive, ensuring reproducibility.

3.2.1 Quantitative Evaluation

YOLO's built-in evaluation module reports the following key metrics:

- **Precision (P):** The fraction of detected boxes that correctly correspond to lane markings

-**Recall (R):** The fraction of ground-truth lane segments that are successfully detected

- **mAP50:** Mean Average Precision at IoU ≥ 0.5
- **mAP50-95:** Stricter mAP using IoU thresholds from 0.5 to 0.95

Features	Results
Precision	0.922
Recall	0.733
mAP50	0.83
mAP50-95	0.669

Table 3.1: Results of the model

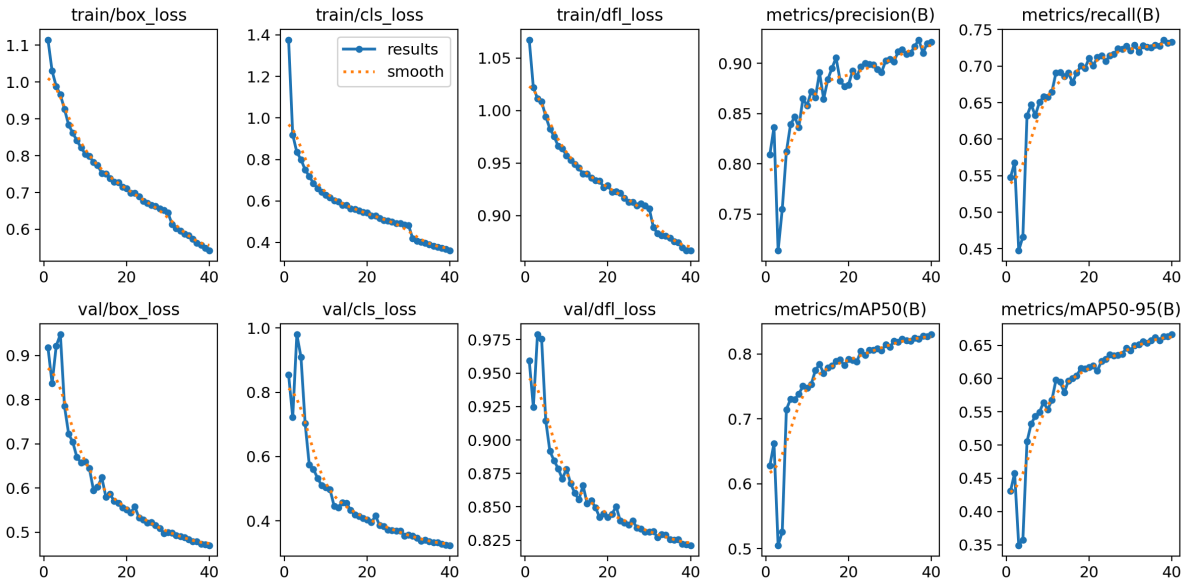

Figure 3.5: Results of the model

These metrics indicate that the model successfully learned the appearance of lane markings even across different towns and environmental conditions. As expected for a single-class detector trained on synthetic data, the model converged quickly and maintained stable validation performance.

epoch	time	train/box_loss	train/cls_loss	train/dfl_loss	metrics/precision(B)	metrics/recall(B)	metrics/mAP50(B)	metrics/mAP50-95(B)	val/box_loss	val/cls_loss	val/dfl_loss	lr/pg0	lr/pg1	lr/pg2
1	192.325	1.11502	1.37637	1.06728	0.80917	0.54766	0.62765	0.43062	0.91817	0.85472	0.9595	0.000663492	0.000663492	0.000663492
2	380.024	1.03031	0.91881	1.02184	0.83628	0.56803	0.66231	0.45773	0.83651	0.72205	0.92453	0.00129724	0.00129724	0.00129724
3	569.464	0.98855	0.83443	1.01146	0.71351	0.44737	0.50519	0.34911	0.92183	0.9801	0.97894	0.00189798	0.00189798	0.00189798
4	754.893	0.96564	0.79947	1.00843	0.75478	0.4665	0.52577	0.3574	0.94768	0.90934	0.97543	0.0018515	0.0018515	0.0018515
5	939.9	0.92612	0.75048	0.99426	0.81186	0.63224	0.71449	0.50518	0.78504	0.70355	0.91441	0.001802	0.001802	0.001802
6	1120.53	0.88361	0.71848	0.98261	0.83911	0.64731	0.73023	0.53163	0.72264	0.57399	0.89159	0.0017525	0.0017525	0.0017525
7	1302.43	0.86208	0.68558	0.97543	0.84654	0.63251	0.72983	0.54368	0.70406	0.56101	0.88461	0.001703	0.001703	0.001703
8	1481.09	0.84076	0.66141	0.96617	0.83637	0.65052	0.73828	0.54951	0.66957	0.53302	0.87853	0.0016535	0.0016535	0.0016535
9	1659.82	0.82194	0.6433	0.9639	0.86477	0.65821	0.7505	0.56403	0.65728	0.50964	0.87095	0.001604	0.001604	0.001604
10	1839.87	0.80503	0.62844	0.95707	0.8578	0.6578	0.74759	0.5534	0.66035	0.50351	0.87821	0.0015545	0.0015545	0.0015545
11	2016.04	0.79874	0.61748	0.95265	0.87214	0.66456	0.75324	0.56742	0.64478	0.49797	0.86759	0.001505	0.001505	0.001505
12	2189.93	0.78249	0.60169	0.94873	0.86566	0.69112	0.77466	0.5981	0.59513	0.44632	0.8602	0.0014555	0.0014555	0.0014555
13	2365.5	0.77388	0.59577	0.94549	0.89077	0.69121	0.78408	0.59489	0.60284	0.44176	0.85531	0.001406	0.001406	0.001406
14	2542.11	0.75291	0.57977	0.93959	0.86446	0.68598	0.77025	0.57888	0.62393	0.4576	0.86598	0.0013565	0.0013565	0.0013565
15	2716.39	0.75124	0.57976	0.93957	0.8838	0.69057	0.77862	0.59685	0.58009	0.45518	0.85246	0.001307	0.001307	0.001307
16	2890.83	0.73887	0.56404	0.9359	0.89508	0.67758	0.7815	0.6006	0.5863	0.43329	0.85476	0.0012575	0.0012575	0.0012575
17	3063.6	0.72874	0.56083	0.93375	0.90544	0.69074	0.78846	0.6044	0.57068	0.41907	0.84945	0.001208	0.001208	0.001208
18	3236.09	0.72728	0.55275	0.93323	0.88265	0.70054	0.79081	0.61517	0.5664	0.41417	0.84226	0.0011585	0.0011585	0.0011585
19	3409.79	0.71489	0.54556	0.92696	0.87693	0.6969	0.78256	0.61475	0.55574	0.40762	0.84484	0.001109	0.001109	0.001109
20	3584.89	0.71102	0.5429	0.92864	0.87858	0.71105	0.79217	0.61678	0.55099	0.4041	0.84206	0.0010595	0.0010595	0.0010595
21	3762.89	0.69903	0.52802	0.92257	0.89234	0.70076	0.79048	0.6191	0.54497	0.39487	0.84461	0.00101	0.00101	0.00101
22	3938.74	0.69825	0.52816	0.9229	0.88684	0.71202	0.78807	0.61164	0.55771	0.41588	0.85018	0.0009605	0.0009605	0.0009605
23	4114.46	0.68929	0.51717	0.92174	0.89638	0.71445	0.80401	0.62635	0.53282	0.38581	0.83951	0.000911	0.000911	0.000911
24	4289.85	0.67616	0.50618	0.91686	0.90021	0.70689	0.7978	0.6295	0.52897	0.38242	0.83796	0.0008615	0.0008615	0.0008615
25	4465.39	0.67154	0.50488	0.91307	0.89898	0.71407	0.80599	0.63618	0.52125	0.37264	0.83671	0.000812	0.000812	0.000812
26	4638.74	0.66581	0.49915	0.9129	0.89838	0.71659	0.8061	0.63441	0.52225	0.36978	0.8397	0.0007625	0.0007625	0.0007625
27	4812.13	0.66289	0.49356	0.90953	0.89406	0.72376	0.80863	0.63541	0.51561	0.36903	0.83425	0.000713	0.000713	0.000713
28	4987.47	0.65658	0.49236	0.91136	0.89125	0.72348	0.80492	0.63652	0.50915	0.36934	0.83345	0.0006635	0.0006635	0.0006635
29	5160.57	0.65221	0.48525	0.90939	0.9026	0.7273	0.8146	0.64624	0.49751	0.35348	0.8312	0.000614	0.000614	0.000614

Table 3.2: Results at each epoch

3.3 Qualitative Evaluation

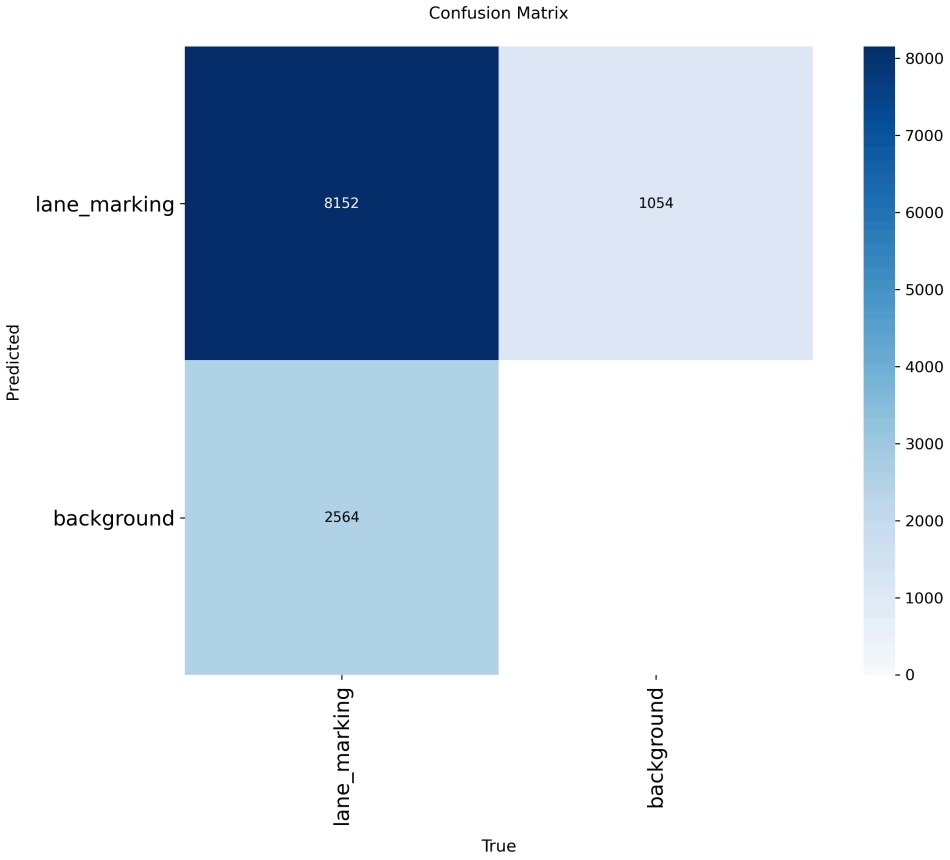

Figure 3.6: Confusion Matrix

A qualitative evaluation was performed by running the trained model on validation images:

```
yolo detect predict --model best.pt --source val_folder
```

The output images were automatically saved and inspected manually.

Observations

- The model consistently detected white lane markings on asphalt roads.
- Bounding boxes were correctly aligned around lane-marking segments, even when the lanes were partially occluded by traffic.
- The detector performed reliably under clear, cloudy, wet, and sunset weather conditions.
- Detection confidence remained high on straight and curved road sections.
- Failure cases occurred mainly in:

overexposed regions (strong sun glare)

extremely dark nighttime scenes (limited visibility)

situations where lane markings were heavily worn or occluded.

Examples

Figure 3.7: Batch2 Valitadion labels

Figure 3.8: Training image boxes

These visualizations confirm that the YOLO model correctly identifies lane-marking segments across a wide variety of scenarios.

3.4 Discussion of Results

The results demonstrate that YOLO is capable of robustly detecting lane markings at the object-level. While YOLO provides only bounding boxes rather than pixel-exact lane shapes, the system successfully localizes regions of interest that correspond to lane boundaries. This makes the method suitable for downstream preprocessing tasks in lane-keeping assistance systems or for driving-policy pipelines that expect coarse spatial lane cues.

However, the model does not generate continuous lane polylines, which are often required in advanced ADAS or autonomous-driving stacks. For fine-grained lane geometry extraction, a segmentation-based approach (e.g., U-Net or LaneNet) would be more appropriate.

Despite this limitation, the overall performance indicates that the synthetic CARLA dataset, combined with YOLOv8s, provides a reliable and computationally efficient solution for lane-marking detection.

Bibliography

[1] B. Görg, Ed., *Vehicle Assist Systems with Focus on the Traffic Lane Keeping Function*. Cham: Springer, 2018.

[2] M. Moretti, Ed., *Collision and Lane Departure Warning Systems for the Trucking Industry: Cost-Benefit Analyses*. Hauppauge, NY: Nova Science Publishers, 2021.

List of Figures

List of Tables

Appendix A

dataset_creation.py

```
1  import queue
2  import random
3  import math
4  import cv2
5
6  MIN_SPEED_KMH = 1.0
7  image_numbers = 0
8
9
10 # ---- getting all required stuff ----
11 from dataset_creation_substuff import (world, tm, settings,
   blueprint_library, tm_port, spawn_points,
12                             relative_transform, ego_bp, cmr_rgb,
   save_image, bp_seg,
13                             semantic_callback, static_weathers)
14
15 # ---- Spawn traffic vehicles ----
16 actors = []
17 vehicle_blueprints = blueprint_library.filter('*vehicle*')
18 for i in range(10):
19     v = world.try_spawn_actor(random.choice(vehicle_blueprints),
   random.choice(spawn_points))
20     if v:
21         actors.append(v)
22
23 # ---- Spawn ego vehicle ----
24 while True:
25     ego_vehicle = world.try_spawn_actor(ego_bp,
   random.choice(spawn_points))
26     if ego_vehicle:
27         actors.append(ego_vehicle)
28         break
29
```

```
30  # ---- setting up autopilot
31  for v in actors:
32      v.set_autopilot(True, tm_port)
33
34
35  rgb_queue = queue.Queue()
36  segm_queue = queue.Queue()
37
38  # ---- attaching camera to ego vehicle ----
39  # camera_ins = world.spawn_actor(cmr_ins_seg, relative_transform,
    attach_to=ego_vehicle)
40  camera_rgb = world.spawn_actor(cmr_rgb, relative_transform,
    attach_to=ego_vehicle)
41  # print(f"Camera {camera_ins.id} attached to vehicle
    {ego_vehicle.id}")
42
43  # ---- creating new window to show images from camera
44  image_numbers = 0
45  # camera_ins.listen(lambda image: save_image(image, typ="ins"))
46  camera_rgb.listen(rgb_queue.put)
47
48  cam_seg = world.spawn_actor(bp_seg, relative_transform,
    attach_to=ego_vehicle)
49  cam_seg.listen(segm_queue.put)
50
51
52  # ---- simulation will start ----
53  print("Simulation running... Press Ctrl+C to stop.")
54  a = 0
55  try:
56      for i in static_weathers:
57          world.set_weather(i)
58          while True:
59              world.tick()
60              rgb_image = rgb_queue.get()
61              seg_image = segm_queue.get()
62
63              if rgb_image.frame != seg_image.frame:
64                  print(f"Frame mismatch: RGB {rgb_image.frame}, SEG
    {seg_image.frame}")
```

```
                continue

                # ---- compute vehicle speed ----
            vel = ego_vehicle.get_velocity()
            speed_ms = math.sqrt(vel.x ** 2 + vel.y ** 2 + vel.z ** 2)
            speed_kmh = speed_ms * 3.6

            # ---- skip saving if vehicle is basically stopped ----
            if speed_kmh < MIN_SPEED_KMH:
                # optional: print occasionally for debugging
                # print(f"Skipping frame {rgb_image.frame}, speed={speed_kmh:.2f} km/h")
                continue
            # ---- car is moving → save images ----
            save_image(rgb_image, image_numbers, a)  # your existing RGB save
            if semantic_callback(seg_image, image_numbers, a):
                image_numbers += 1
            if image_numbers == 100:
                image_numbers = 0
                break
        a+=1
except KeyboardInterrupt:
    print("Stopping simulation...")
finally:
    camera_rgb.stop()
    camera_rgb.destroy()
    # camera_ins.stop()
    # camera_ins.destroy()
    cam_seg.stop()
    cam_seg.destroy()
    for a in actors:
        a.destroy()
    tm.set_synchronous_mode(False)
    settings.synchronous_mode = False
    world.apply_settings(settings)
    cv2.destroyAllWindows()
    print("Clean shutdown complete.")
```

Appendix B

dataset_creation_substuff.py

```
1  import cv2
2  import numpy as np
3  import carla
4  import hashlib
5
6  last_hash = None
7  count_rgb = 0
8
9  img_hash = 1
10 creating = True
11
12
13 # ---- Client is set inside local machine at port 2000 ----
14 client = carla.Client("localhost", 2000)
15 client.set_timeout(10.0)
16
17 # --- Reset world and TM (to run again without closing
   CarlaUE4.exe) ---
18 town_name = 'Town10HD'
19 world = client.load_world(town_name)
20
21 static_weathers = [ carla.WeatherParameters.ClearNoon,
22                     carla.WeatherParameters.CloudyNoon,
23                     carla.WeatherParameters.WetNoon,
24                     carla.WeatherParameters.WetCloudyNoon,
25                     carla.WeatherParameters.MidRainyNoon,
26                     carla.WeatherParameters.HardRainNoon,
27                     carla.WeatherParameters.SoftRainNoon,
28                     carla.WeatherParameters.ClearSunset,
29                     carla.WeatherParameters.CloudySunset,
30                     carla.WeatherParameters.WetSunset,
31                     carla.WeatherParameters.WetCloudySunset,
32                     carla.WeatherParameters.MidRainSunset,
```

```
                    carla.WeatherParameters.HardRainSunset,
                    carla.WeatherParameters.SoftRainSunset]


# try:
#     current_map = world.get_map().name.split('/')[-1]
#     world = client.load_world(current_map)
#     print(f"Reloaded map: {current_map}")
# except RuntimeError as e:
#     print(f"Map reload failed: {e}")


# ---- Traffic manager is reset ----
tm = client.get_trafficmanager(8000)
tm.set_synchronous_mode(False)

settings = world.get_settings()
settings.synchronous_mode = True
settings.fixed_delta_seconds = 0.05
world.apply_settings(settings)
print(world.get_map().name)
print(client.get_available_maps())
blueprint_library = world.get_blueprint_library()
spawn_points = world.get_map().get_spawn_points()
def world_recreate(new_map, worl, setting, actors):
    for a in actors:
        a.destroy()
    setting.synchronous_mode = False
    worl.apply_settings(settings)
    worl.load_world(new_map)
    settings.synchronous_mode = True
    worl.apply_settings(settings)

# --- Traffic Manager setup ---
tm_port = tm.get_port()
tm.set_synchronous_mode(True)
tm.set_global_distance_to_leading_vehicle(1.0)
tm.global_percentage_speed_difference(0)

```

```
# ---- setting up the instance segmentation camera ----
cmr_ins_seg = blueprint_library.find('sensor.camera.instance_segmentation')
cmr_ins_seg.set_attribute('role_name', 'camera')
cmr_ins_seg.set_attribute('image_size_x', '800')
cmr_ins_seg.set_attribute('image_size_y', '600')
cmr_ins_seg.set_attribute('fov', '90')

# ---- setting up the rgb camera ----
cmr_rgb = blueprint_library.find('sensor.camera.rgb')
cmr_rgb.set_attribute('role_name', 'camera')
cmr_rgb.set_attribute('image_size_x', '800')
cmr_rgb.set_attribute('image_size_y', '600')
cmr_rgb.set_attribute('fov', '90')

# Create semantic segmentation camera
bp_seg = blueprint_library.find('sensor.camera.semantic_segmentation')
bp_seg.set_attribute('image_size_x', '800')
bp_seg.set_attribute('image_size_y', '600')
bp_seg.set_attribute('fov', '90')









# ---- relative transform of the camera of ego vehicle ----
relative_transform = carla.Transform(
    carla.Location(x=1.5, z=1.7),
    carla.Rotation(pitch=0)
)

ego_bp = blueprint_library.find('vehicle.lincoln.mkz_2020')
ego_bp.set_attribute('role_name', 'hero')
def save_image(image, image_numbers, weather_number):
    global count_rgb
    array = np.frombuffer(image.raw_data, dtype=np.uint8)
    array = np.reshape(array, (image.height, image.width, 4))
    rgb = array[:, :, :3]
    cv2.imshow("CARLA Camera View", rgb)
```

```
111         cv2.waitKey(1)
112         count_rgb += 1
113         if count_rgb == 20 and image_numbers < 100 and creating:
114             img = np.frombuffer(image.raw_data,
    dtype=np.uint8).reshape(image.height, image.width, 4)
115             filename = f"dataset/rgb/{town_name}_{weather_number}
    _{image_numbers}.png"
116             cv2.imwrite(filename, img)
117             print("image", image_numbers)
118
119
120     def semantic_callback(image, image_numbers, weather_number):
121         global count_rgb, town_name
122         # Raw data is a flat uint8 array. Each pixel = 4 bytes (BGRA).
123         img_array = np.frombuffer(image.raw_data, dtype=np.uint8)
124         img_array = img_array.reshape((image.height, image.width, 4))
125
126         # The R channel contains the semantic tag ID directly.
127         # (CARLA encodes class IDs into the red channel)
128         semantic_map = img_array[:, :, 2]  # R channel in BGRA order
129         lane_id = 24  # lane marking semantic ID (check below)
130         lane_mask = (semantic_map == lane_id).astype(np.uint8) * 255
131
132         # Now semantic_map[y,x] is the class ID (integer)
133         # Example: 7 = lane marking, 1 = building, etc.
134         # cv2.imwrite(f"images/lane_{image.frame}.png", lane_mask)
135         if count_rgb == 20 and image_numbers < 100 and creating:
136             filename = f"dataset/seg/{town_name}_{weather_number}
    _{image_numbers}.png"
137             cv2.imwrite(filename, lane_mask)
138             print("seg")
139             count_rgb = 0
140             return True
141         else:
142             return False
143
144
```

Appendix C

Machine_Vision_HW.ipynb

```
# -*- coding: utf-8 -*-
"""Machine Vision HW.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1HAnO2QUS6XYKl3GZX7mTEHdSgd22dAkl
"""

# Commented out IPython magic to ensure Python compatibility.
# %cd /content

!unzip "/content/drive/MyDrive/rgb.zip" -d dataset_raw

import os
import cv2
import numpy as np

IMAGES_DIR = "dataset_raw/rgb"
MASKS_DIR = "dataset_raw/seg"
LABELS_DIR = "dataset_raw/labels"

os.makedirs(LABELS_DIR, exist_ok=True)

LANE_CLASS = 0  # one class only

def create_labels():
    img_files = sorted([f for f in os.listdir(IMAGES_DIR) if f.endswith(".png")])

    for file in img_files:
        img_path = os.path.join(IMAGES_DIR, file)
```

```
        mask_path = os.path.join(MASKS_DIR, file)

        # load RGB to know dimensions
        img = cv2.imread(img_path)
        h, w = img.shape[:2]

        # load mask (grayscale)
        mask = cv2.imread(mask_path, cv2.IMREAD_GRAYSCALE)

        # binary mask (255 = lane, 0 = background)
        _, binary = cv2.threshold(mask, 1, 255, cv2.THRESH_BINARY)

        # find all lane blobs
        contours, _ = cv2.findContours(binary, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)

        label_lines = []

        for cnt in contours:
            x, y, bw, bh = cv2.boundingRect(cnt)

            # skip tiny noise
            if bw < 5 or bh < 5:
                continue

            # convert to YOLO format
            x_center = (x + bw / 2) / w
            y_center = (y + bh / 2) / h
            width = bw / w
            height = bh / h

            label_lines.append(f"{LANE_CLASS} {x_center} {y_center}
{width} {height}")

        # save label file
        label_path = os.path.join(LABELS_DIR, file.replace(".png",
".txt"))
        with open(label_path, "w") as f:
            f.write("\n".join(label_lines))

```

```
69          print(f"Labeled: {file} → {len(label_lines)} boxes")
70
71  if __name__ == "__main__":
72      create_labels()
73
74  import os
75  import random
76  import shutil
77
78  # paths
79  RGB_DIR     = "dataset_raw/rgb"
80  LABELS_DIR  = "dataset_raw/labels"
81  IMAGES_OUT  = "dataset/images"
82  LABELS_OUT  = "dataset/labels"  # to not overwrite original
83
84  train_ratio = 0.8  # 80% train, 20% val
85
86  train_img_dir = os.path.join(IMAGES_OUT, "train")
87  val_img_dir   = os.path.join(IMAGES_OUT, "val")
88  train_lbl_dir = os.path.join(LABELS_OUT, "train")
89  val_lbl_dir   = os.path.join(LABELS_OUT, "val")
90
91  os.makedirs(train_img_dir, exist_ok=True)
92  os.makedirs(val_img_dir,   exist_ok=True)
93  os.makedirs(train_lbl_dir, exist_ok=True)
94  os.makedirs(val_lbl_dir,   exist_ok=True)
95
96  # all pngs
97  images = [f for f in os.listdir(RGB_DIR) if
    f.lower().endswith(".png")]
98  images.sort()
99  random.shuffle(images)
100
101 split_idx = int(len(images) * train_ratio)
102 train_files = images[:split_idx]
103 val_files   = images[split_idx:]
104
105 def move_files(file_list, img_dest, lbl_dest):
106     for f in file_list:
107         img_src = os.path.join(RGB_DIR, f)
```

```
108         lbl_src = os.path.join(LABELS_DIR, f.replace(".png",
    ".txt"))
109
110         img_dst = os.path.join(img_dest, f)
111         lbl_dst = os.path.join(lbl_dest, f.replace(".png", ".txt"))
112
113         if not os.path.exists(lbl_src):
114             print(f"Label missing for {f}, skipping.")
115             continue
116
117         shutil.copy2(img_src, img_dst)
118         shutil.copy2(lbl_src, lbl_dst)
119
120     print(f"Copied {len(file_list)} images to {img_dest}")
121
122 move_files(train_files, train_img_dir, train_lbl_dir)
123 move_files(val_files,   val_img_dir,   val_lbl_dir)
124 print("Done splitting.")
125
126 # Where your dataset is in Colab (after unzipping & splitting)
127 DATASET_ROOT = "/content/dataset"
128
129 # Where to save ALL training outputs (weights, results.csv, plots)
    on Drive
130 PROJECT_DIR = "/content/drive/MyDrive/carla_lane_yolo"
131
132 yaml_text = f"""
133 path: {DATASET_ROOT}
134
135 train: images/train
136 val: images/val
137
138 names:
139   0: lane_marking
140 """
141
142 with open("carla_lane.yaml", "w") as f:
143     f.write(yaml_text)
144
145 !cat carla_lane.yaml
```

```
146
147 !pip install -q ultralytics
148 from ultralytics import YOLO
149
150 !yolo detect train \
151   data=carla_lane.yaml \
152   model=yolov8n.pt \
153   epochs=40 \
154   imgsz=512 \
155   batch=32 \
156   device=0 \
157   project="$PROJECT_DIR" \
158   name="exp_t4" \
159   exist_ok=True
160
161 !yolo detect predict \
162   model=/content/drive/MyDrive/carla_lane_yolo/exp_t4/weights/
    best.pt \
163   source=/content/dataset/images/val \
164   conf=0.3 \
165   imgsz=640 \
166   device=0 \
167   save=True \
168   project=/content/drive/MyDrive/carla_lane_yolo \
169   name=exp_t4_preds \
170   exist_ok=True
```