Politecnico di Torino

POLITECNICO DI TORINO

COLLEGIO DI INGEGNERIA INFORMATICA, DEL CINEMA E MECCATRONICA

Corso di Laurea Magistrale in Ingegneria Informatica

A new benchmark for Anomaly Segmentation in driving scenes, using the CARLA simulator

Relatore

Prof. Carlo MASONE
Phd. Shyam Nandan RAI

Candidato

Andrea PANI

SESSIONE DI LAUREA DI MARZO/APRILE 2025

Contents

Chapter 1

Introduction

Autonomous driving holds the keys to a future where driving time could be reallocated to other, more productive activities, where accidents due to human errors are minimized and where traffic could be optimized to reduce energy consumption. However, much progress still needs to be made for this to happen. In particular, handling of anomalous, hazardous objects, rarely seen by the vehicle during normal operation, is particularly difficult for the current state-of-the-art autonomous cars; this poses safety risks that sometimes lead to dangerous accidents, including pedestrian death. Furthermore, most deep learning models deployed on these cars are so called "black-box", because it's difficult to predict their behavior reliably and to identify what caused an accident after it happened. Research on these matters has been slow partly because it is difficult and expensive to produce datasets of road scenes with anomalies, making the development of deep-learning models to detect them difficult. One possibility to produce a high quantity of data with low economic cost is the use of graphic engines to simulate realistic environments with cities, vehicles and pedestrians. This thesis investigates the possibility of using CARLA (Car Learning to Act), a simulator based on Unreal Engine, to produce a dataset of road scenes filled with anomalous objects. The simulator, by offering software implementations of a variety of sensors like RGB cameras and LIDAR, often part of the perception modules of autonomous vehicles, offers also the possibility of recording different modalities of data. In the first part of this work a script is developed using the API provided by CARLA, that by taking user inputs and controlling the simulation as such, produces a dataset of road scenes in different maps and weather conditions. Then, after the collection of the dataset, some of the state-of-the-art models for anomaly segmentation on road scenes are tested on this dataset to assess their performance. In particular, the following chapters cover:

- Chapter 2, *Background*, includes an overview of Autonomous driving, an introduction to the task of Semantic segmentation and a description of all the features of the CARLA simulator
- Chapter 3, *Related works*, is focused on datasets and models in the task of Anomaly segmentation, that have been used or influenced this work
- Chapter 4, *Dataset creation* in which is described how the CARLA simulator was used to produce datasets with different sensor modalities and with or

without anomalies

- Chapter 5, *Experiments and results*, discusses how the datasets where used to benchmark models for the task of Anomaly segmentation, giving insights into their strength and weaknesses
- Chapter 6, *Conclusions*, wraps up the work, by hypothesizing on future directions that research on the matter discussed on this thesis could take

Chapter 2

Background

2.1 Autonomous Driving: an Overview

An Autonomous Driving System (ADS) [1], also known in popular culture with the somewhat misleading term "self-driving car", is a vehicle that is capable of operating with reduced or absent human input. Since according to a report [2] by the National Highway Traffic Safety Administration (NHTSA) as much as 94% of road accidents are caused by human errors, this paradigm shift promises a much more secure driving environment, other than related benefits such as reduced energy consumption, reduced stress, and increased productivity due to reallocation of driving time. However, despite business-driven promises, no company or organization is anywhere near achieving full autonomy, especially in challenging scenarios such as urban environments. An important comparison framework for ADSs is the one [3] proposed by the Society of Automotive Engineers (SAE) that defines 5 levels of autonomy:

- Level 0: no automation at all
- Level 1: primitive driving assistance methods such as Adaptive Cruise Control, Anti-Lock Braking Systems and Stability Control
- Level 2: partial automation with advanced assistance like Emergency Braking or Collision Avoidance System (CAS)
- Level 3: conditional automation, the driver can focus on other tasks while the car is operating but needs to quickly be able to take over the vehicle when prompted with an acoustic signal. Furthermore, the vehicle can only operate in "controlled environments", formally defined as Limited Operational Design Domains (ODDs), such as highways.
- Level 4: no human input is needed, but the vehicle can operate only in ODDs where special infrastructure or detailed maps exist. When departing from these areas, the vehicle must automatically park itself.
- Level 5: complete automation, the vehicle can operate by itself in all scenarios and weather conditions possible.

Figure 2.1: The 6 Levels of Autonomus Driving

For context, the Tesla Autopilot is considered by industry experts to be a Level 2 SAE technology [4], since it "requires active driver supervision and does not make the vehicle autonomous" as stated on Tesla's official website. Instead, Mercedes-Benz has recently announced its Level 3 DRIVE PILOT system on the luxury models S-class and EQS [5]. This newer version reaches the world record of up to 95 km/h of speed, but it can only operate on the German Autobahn network, legally allowing the driver to divert attention away from the vehicle. Regarding Level 4, companies such as Waymo and Baidu are testing driverless taxies, also called robo-taxies, in certain parts of the USA and China, respectively [6]. However, these early adoption efforts face challenges that range from social acceptance, cybersecurity, and production costs. Regarding Level 5, the Toyota Research stated that no automaker is even close to achieving full autonomy [7], despite announcements by business leaders.

2.1.1 System Architectures

ADS architectures can be classified at the macro-level by how vehicles interact with each other, and at the micro or vehicle level by how the different modules interact to enable autonomous driving. At the macro-level there are two main approaches followed: ego-only system or connected, multi-agent system. In a **Ego-Only System**, all the necessary operations for autonomous driving are carried out in the vehicle only, without relying on external elements or other vehicles. Most of the State-of-The-Art ADS fall into this category. In a **Connected Multi-Agent System** instead, the agents (vehicles) interact with each other using dedicated infrastructure sharing information about driving and the environment. This approach is mostly

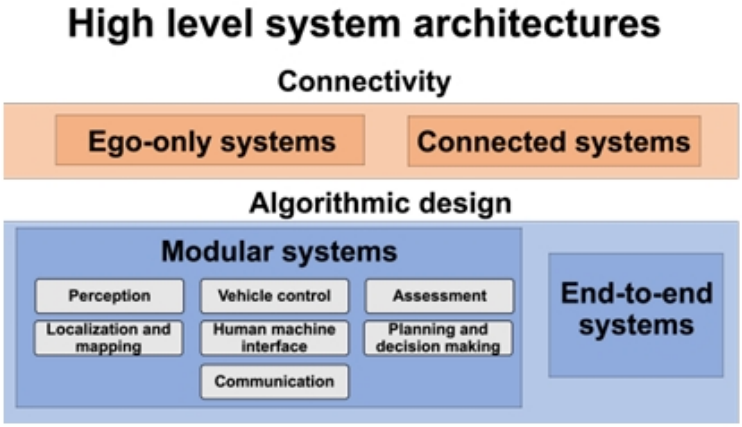


Figure 2.2: Classification of Automated Driving System Architectures

theoretical for now since it requires huge infrastructural changes, but holds the promise of solving ego-only vehicle problems such as sensing range, limited computational power and blind spots. At the vehicle level, the architectural choices used can be broadly categorized as modular systems or end-to-end driving systems, as shown in figure 2.3. In a **Modular system**, the architecture is structured as a pipeline that processes sensory inputs and produces actuator outputs. The core components of a modular ADS pipeline are:

- Localization and Mapping
- Perception
- Assessment
- Planning and Decision Making
- Vehicle Control
- Human-Machine Interface

The advantage of this architecture lies in the decomposition of a complex problem (automated driving) into a set of smaller ones, that in some cases have a complex and rich scientific literature on their own (for example, Computer Vision and Robotics). Apart from knowledge transfer, modularity allows for more safety, for example by adding a set of hard-encoded rules in the pipeline, and interpretability, as functions are self-contained in a module. Drawbacks of this architecture choice are error-propagation and over-complexity. For example, in the first ADS fatality caused by a Tesla car, a wrongly classified white trailer (as sky) was propagated from the Perception module up to the end of the pipeline, causing a severe system error. In a **End-to-end System** motion is generated directly from sensory inputs, using an all-in-one approach. In this architecture, motion can come in the form of a set discrete actions, such as accelerate or turn left, or as the continuous control of the vehicle peripherals. To implement and end-to-end pipeline, three main strategies are used:

- **Direct supervised deep learning** uses a model that learns directly from a human driver and can be trained offline. This method has poor generalization performance
- **Deep Reinforcement Learning** uses a model that learns by online training the optimal way of driving. However, urban driving has not been achieved yet using this method.
- **Neuroevolution** uses evolutionary algorithms to train a neural network model, without backpropagation. Real-world driving has not been achieved yet.

Generally speaking, the biggest problem of the end-to-end driving paradigm is the lack of interpretability and hard-coded safety measures, since the system works as a kind of "black box" in which is difficult to identify possible incident causes.

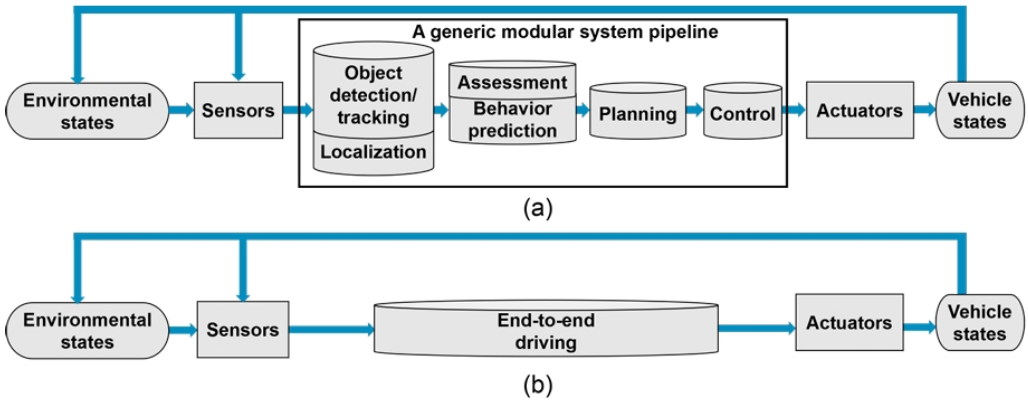


Figure 2.3: Flow diagram of (a) a Modular System and (b) and End-to-End Driving System

2.1.2 Sensors and Hardware

Every modern autonomous vehicle uses a wide variety of on-board sensors for operation; in most cases, high sensor redundancy is needed to guarantee robustness and reliability. Types of sensors used include exteroceptive sensors, used to perceive the environment, proprioceptive sensors, used for internal vehicle monitoring, communication arrays, actuators, and computational units. The most commonly used **Exteroceptive sensors** for perception are:

- **Monocular cameras**, that perceive color while not emitting any signal. Coupled with powerful 2-D Computer Vision algorithms, they can be a simple yet powerful sensing tool, although affected by illumination conditions
- **Event cameras** record data asynchronously for individual pixels with respect to visual stimulus. The output therefore consists of irregular data points / events caused by changes in brightness. Current event cameras have lower resolutions than standard cameras.

- **Radar** perceives the distance of objects by measuring the time and strength of emitted radio waves, returning to the source after bouncing back on object surfaces. It is simple, lightweight, and cost-effective.

- **LIDAR** has a very similar principle to that of the Radar, but uses lasers of ultraviolet, visible or near-infrared light instead of radio waves. Its accuracy under 200 meters is much stronger than that of Radar. LIDARs are affected by weather conditions such as rain, fog or snow and are generally larger than Radars, making them less practical.

In general, ultrasonic sensors like Radar and LIDAR serve to compensate for the shortcomings of cameras under certain conditions, for example at night. However, being active sensors, they can be affected and affect other systems as well. **Proprioceptive sensors** are crucial in determining speed, acceleration, yaw and other state parameters that require continuous monitoring to guarantee safe operation of a vehicle. Some examples of proprioceptive sensors are wheel encoders and IMU (Inertial Measurement Unit), used for odometry (the process of estimating the position and orientation of a moving object), tachometers to measure vehicle speed and altimeters for altitude.

2.1.3 Localization and Mapping

In an ADS driving scenario, **Localization** is defined as determining the precise position and orientation of the vehicle in its environment. This task is crucial in order to keep the lane, avoid obstacles and route planning. As shown in Figure 2.3, the Localization module is part of the driving pipeline of a Modular ADS. **Mapping** instead refers to the creation of a representation of the vehicle's surroundings, starting from onboard sensors data (like cameras and LIDAR); it provides the vehicle the context necessary to understand its environment, like roads, buildings and traffic signs. A simple strategy for Localization is **GPS-IMU fusion**: since using only position and orientation coming from the IMU unit leads to accumulated error over time, GPS readings are integrated to help correct them, leading to more reliable localization of the vehicle over time. However, the accuracy required by urban driving environments is too high for current GPS-IMU systems to be employed effectively, and their accuracy even drops in particularly dense urban scenarios or in the presence of tunnels. In **Simultaneous Localization and Mapping (SLAM)**, the task of location and mapping are combined, by simultaneously building the map during operation (online) and localizing the vehicle in it. The technique comes from robotics and is generally applied in closed environments, while its application in open scenarios like the one needed by ADS is for now inefficient. For this reason it is still common for autonomous cars to use pre-built maps; in a technique called **A-Priori Map Based Localization**, online data coming from sensors is compared to an already built map to find the best matching position. This method obviously requires the additional step of map creation, but what if the environment changes significantly over time? The map needs to be updated, an expensive operation. That's why SLAM techniques have the potential to replace a priori map-based methods if their performance increases in the future.

2.1.4 Perception

Perception is the ADS task of perceiving the environment with the goal of extracting useful information that the vehicle can use for safe navigation. It encompasses various different sub-tasks such as object-detection, semantic segmentation, road and lane detection, and object tracking. In **Object Detection**, the goal is to identify both the location and size of objects in the vicinity of a vehicle, be it static, like roads and traffic signs / lights, or dynamic, like vehicles and pedestrians. After the deep learning revolution in 2012, two different architectural style choices have been used to perform object detection:

- **Single stage detection** networks use a single neural network to produce object locations (bounding boxes) and labels at the same time. Models like You Only Look Once (YOLO), based on convolutional neural networks (CNNs) and more recently the DEtection TRansformer (DETR) are examples.

- **Region-proposal** networks employ a two stage process: first a network select regions of interest in an image, then another network categorizes them. An example is the Faster R-CNN models.

When comparing models for object detection that needs to be deployed in autonomous vehicles, not only accuracy in bounding boxes estimation needs to be taken into consideration. Computational power is extremely important, because the car needs to have time to react to objects; that is why a balance between at how many frame per second (FPS) a model can operate and its accuracy needs to be considered. In **Semantic Segmentation**, each pixel in an image is classified as belonging to a different class. This task is important for autonomous driving since some objects like roads, sidewalks and traffic lines are poorly defined by bounding boxes. Since this thesis work regards the creation of a dataset for semantic segmentation, and in particular with the goal of segmenting anomalies in road scenes, a more detailed analysis of architectures for semantic segmentation is found in Section 2.2. Often simply understanding the location of an object is not enough for safe driving, but also the speed and future direction of an object needs to be estimated using some kind of modeling, in order to prevent collision. In **Object tracking**, this problem is often tackled with the use of multiple sensor modalities, such as cameras and LIDAR, used together (sensor fusion). Likewise, often simply segmenting the drivable surfaces using semantic segmentation models is not enough: in the problem of **Road and Lane detection** a vehicle needs to understand the semantics of the road in order not to cause accidents.

2.1.5 Assessment

After perception of the environment has occurred, a robust ADS also needs to quantify uncertainties. The **Assessment** task consist in estimating future intentions of dynamic actors such as vehicle and pedestrians, with the goal of inferring the overall risk of the current situation. Assessment can take many forms:

Figure 2.4: Risk assessment for road scenes

- **Risk and uncertainty assessment** aims to assess the global risk level of a complex driving scene, composed, for example, of several vehicles or pedestrians. This method could greatly increase the overall safety of an ADS pipeline

- **Driving behavior and style recognition** aims to build models of human traits of drivers in order to predict future actions of vehicles

It is fair to say that these methodologies are of much importance for the future development of SAE 5 truly autonomous vehicles, but at current times research on these topics is still in its infancy and the problem of estimating driving behavior is far from solved.

2.1.6 Planning and Decision Making

Planning in an ADS is the task of, given a destination by the user of the system, to develop a program with the goal of reaching that location. In **Global planning**, the shortest route to the destination on a map must be found. This often takes the form of choosing the shortest point-to-point path in a directed graph. Efficient algorithms on this matter have been extensively investigated by research in various fields and are already utilized in our GPS systems today. In **Local planning**, the goal is to execute the global plan without failures. This implies finding trajectories to avoid obstacles, while also satisfying optimality criteria for the graph-based configuration space, using information coming from earlier modules in an ADS pipeline.

2.2 Semantic Segmentation

Image Segmentation [8] is a computer vision (CV) task that consists in partitioning images (or video frames) into a set of known classes or objects. It plays a crucial role in a broad range of applications such as medical imaging (E.g. tumor segmentation), autonomous vehicles (E.g. identifying drivable surface, vehicles and pedestrians), video surveillance and many more. Image Segmentation can be further subdivided into three slightly different tasks:

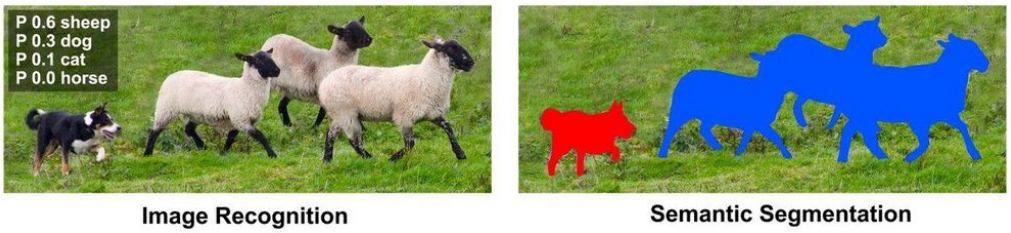


Figure 2.5: Difference between Image recognition and Semantic segmentation

- **Semantic segmentation**: assigns each pixel of an image to a set of known classes (E.g. human, car, tree), discarding the regions of no interest. Objects with the same class will have the same label.

- **Instance segmentation**: aims at segmenting countable objects by also distinguishing between different instances.

- **Panoptic segmentation**: combines both semantic segmentation and instance segmentation, distinguishing between "things", countable objects that need instance identification, and "stuff", amorphous regions like sky and grass that are a single instance.

2.2.1 Datasets

To train models for Image segmentation, one of the most popular datasets is the **PASCAL Visual Object Classes (VOC)**. It consists of two sets of images, 1,464 for the training and 1,449 for the validation set, plus an hidden test set for the benchmark challenge evaluation. Each of these images is annotated with one of 21 classes (E.g. vehicles, household, animals) in addition to a background class where for pixels not belonging to any of the classes. Another notorious dataset is the **Microsoft Common Objects in Context (MS COCO)**, consisting of 328k images of 91 classes of common objects in their natural context. With respect to datasets for road scenes understanding , **Cityscapes** [9] is without a doubt the most used. It consists of 5000 images with high-quality pixel-level annotations, plus an additional 20000 images with coarse (less precise) annotations. The images were recorded by using a moving vehicle with cameras, over the span of several months covering spring, summer and fall, in 50 cities in Germany. Images are annotated with 30 classes divided into eight categories: flat, construction, nature, vehicle, sky, object, human, and void. Since some classes are rarely present in images, in the official benchmark challenge the authors evaluate the submitted models only on a subset of 19 most frequent classes. The **BDD100K** dataset is a much larger dataset comprised of 100.000 images coming from 1100 videos, captured by crowd-sourced drivers across the world: this ensures stronger diversity with respect to Cityscapes. Unlike the former, however, it has only 10 classes to be used for Semantic segmentation, but captured images have a much higher variety of weather conditions (like fog and rain) and include night shots.

Figure 2.6: An example Image and Annotation coming from Cityscapes

2.2.2 Architectures

Before the advent of Deep Learning, Image segmentation was performed using various simpler methods, such as thresholding a grayscale image to distinguish between foreground and background, or grouping pixels similar in color, texture, and intensity in clusters using the K-Means algorithm. With the widespread use of Neural Networks, and in particular Convolutional Neural Networks (CNNs), several architectures were proposed over the years to perform segmentation harnessing the power of large quantities of data and deeper networks. One of the first such architectural patterns is the **Fully Convolutional Network (FCN)** that consists of a convolutional network (usually existing architectures like VGG16 or GoogLeNet) where all linear layers are stripped. As such, only convolutional layers remain, and the final segmentation map is produced by utilizing only a transpose convolution layer (de-convolution) that up-samples the image to the original resolution; one interesting feature of the FCN is that it can handle images of different shapes since it only uses convolutional layers. An evolution of this type of architecture is the **Encoder-Decoder model** which, instead of a single de-convolution, produces a segmentation map utilizing a series of such layers interposed by un-pooling layers. Using more parameters than an FCN increases the sensitivity to smaller details, making the Encoder-Decoder network perform better when small objects are involved, at the cost of higher memory and computational usage. Going forward, The **DeepLab** family of convolutional neural networks introduced a series of new features with respect to simple Encoder-Decoder architectures that made them capture finer details of images and handle images of higher resolutions:

- Atrous or Dilated convolutions make possible to combine information over a larger area of an image using the same weights as standard convolutions interspersed with zeros.
- Atrous Spatial Pyramid Pooling (ASPP), a technique consisting of multiple atrous convolutions in parallel with different dilation rates. This makes the model capture information at different scales simultaneously, improving segmentation accuracy.
- The usage of different backbones like Resnet or MobileNet that provide feature representation can influence model performance and efficiency, depending on the desired task

Another important leap was made by utilizing the revolutionizing Transformer and in particular Vision Transformer (ViT) architectures to image segmentation. Vi-

sion Transformers improve upon some of the shortcomes of convolutional networks; first, by leveraging their attention mechanism, they capture global dependencies between different or distant parts of an image, making them even more able to delineate complex objects than CNNs; second, each sub-task of image segmentation (Semantic segmentation, Instance segmentation and Panoptic segmentation) was approached by researching and building different specialized convolutional networks, while transformers offer a unified architecture that can tackle all simultaneously. One of such architectures is **MaskFormer** and its evolution **Mask2Former** [10], that introduce a single mask-level classification to handle both instance and semantic segmentation, where usually a per-pixel classification strategy was employed.

2.2.3 Metrics

A simple metric to evaluate models on Semantic Segmentation is the **Pixel accuracy (PA)**. It simply evaluates a model on the ratio of correctly classified pixels over the total number of pixels; for example, in a problem setting with K different classes:

$$PA = \frac{\sum_{i=0}^{K} p_{ii}}{\sum_{i=0}^{K} \sum_{j=0}^{K} p_{ij}}$$

where p_{ij} is the fraction of pixels labeled as class i while being from class j. When classes are not balanced, meaning that some classes account for significantly fewer labeled pixels than others, **Mean pixel accuracy (MPA)** can instead be used. It calculates the ratio of correctly labeled pixels for each class and then averages them:

$$MPA = \frac{1}{K} \sum_{i=0}^{K} \frac{p_{ii}}{\sum_{j=0}^{K} p_{ij}}$$

One of the most popular metrics is the **Intersection over Union (IoU)**, also called the **Jaccard Index**; It is defined as the area of intersection between the predicted segmentation map and the ground truth, divided by the area of union between the predicted segmentation map and the ground truth:

$$\begin{aligned} IoU = J(A, B) &= \frac{|A \cap B|}{|A \cup B|} \\ &= \frac{TP}{TP + FP + FN} \end{aligned}$$

where A and B denote the ground truth and the respective predicted segmentation map (with values ranging from 0 to 1) and:

- True positives (TP) are the fraction of pixels correctly labeled as belonging to the class
- False positives (FP) are the fraction of pixels incorrectly labeled as belonging to the class
- False negatives (FN) are the fraction of pixels incorrectly labeled as not belonging to the class

An averaged version over all classes of this metric is also widely used, called the **Mean IoU**.

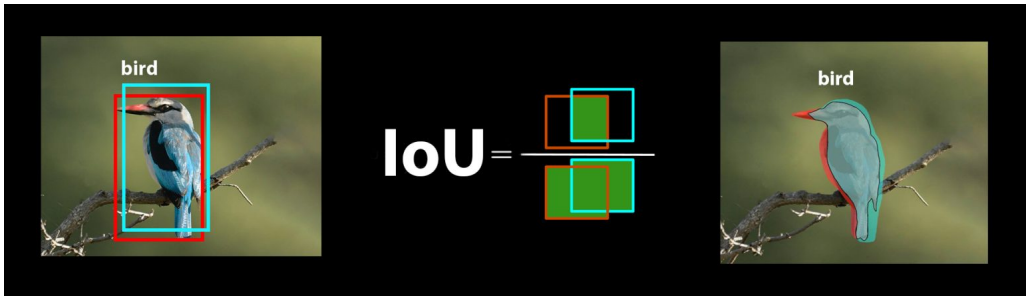


Figure 2.7: Intersection over Union metric

2.3 The CARLA simulator

CARLA (Car Learning to Act) [11] is an open-source driving simulator designed for research and development in the fields of autonomous driving, robotics, and artificial intelligence, developed by the Computer Vision Center (CVC) of Barcelona. CARLA offers out-of-the-box high-fidelity 3D assets like cars and buildings, customisable maps and implementations of sensors and control interfaces. The CARLA simulator consists of a scalable client-server architecture, where multiple user scripts written in the Python programming language can interact with the same server simulation, written in C++ and based on a modified build of the graphics engine Unreal Engine, developed by Epic Games. The usage of Unreal Engine as a core provides CARLA with high-quality graphics and photorealism, advanced physics simulation and AI control of NPCs (non-player characters), without the need to re-implement everything from the beginning.

2.3.1 World and Client

The **CARLA Client** is the module that the user runs in order to obtain information about the state of the simulation or to request changes to it. The client connects to the CARLA server, running the simulation, using an IP address and a port; the server could therefore be running on the same machine or in another machine on the network. As mentioned before, multiple clients can connect to the same server, although requiring synchronization to maintain a coherent simulation state. Some of the actions that the client can perform are loading maps, recording the simulation and retrieval of server information.

The **World** in CARLA is an object representing the actual state of the simulation. It also acts as a logical layer that exposes methods to spawn actors, change the weather and get information about objects placed in the world. There is only one world per simulation, and it will be destroyed and created new every time a new map is loaded. Another interesting concepts in CARLA are the **asynchronous** and **synchronous** mode. When the **asynchronous mode** is set, the server runs as fast as possible and handles client requests on the fly. This means that if the client is slower than the server to handle data, it will lose some information. When using the **synchronous mode**, instead, the server waits for a client "tick" to advance the state of the simulation; this ensures that the client processing finishes before the simulation can continue.

(a) A vehicle

(b) A walker

Figure 2.8: Example of CARLA actors

2.3.2 Actors and Blueprints

Actors are the objects in a CARLA simulation that can perform actions and affect other actors as well. Actors in CARLA include vehicles, walkers (pedestrians), and even sensors, traffic signs and traffic lights; some of this actors can be spawned in the simulation (vehicles, walkers, sensors) while traffic signs and lights are fixed from the start. The actors that can be spawned in the running simulation are described by a **Blueprint**, which essentially defines, using the Blueprint scripting language and classes of Unreal Engine, its properties that depend on the type of actor considered. Example of modifiable properties are color for vehicles, speed for walkers and resolution for a camera sensor. Actors like vehicles and walkers consist also of a 3-D model with animations, as exemplified in figure 2.8. All of the blueprints available in CARLA can be accessed by using an object called **Blueprint Library**, which accepts blueprint ids and gives back a blueprint for the associated actor (which is in turn used by the World object for spawning). As an example, to spawn the vehicle in figure 2.8, a query to the Blueprint library for the actor with id equal to "vehicle.lincoln.mkz_2020" would be needed, while to spawn a pedestrian (also in figure 2.8) the id to be used would be "walker.pedestrian.0001".

2.3.3 Maps and Traffic

In CARLA, Actors are placed inside a **Map**, which comprises both a 3-D model of the town, constructed using Unreal Engine, and the road annotation, which consists of an OpenDRIVE file, a standard file format used for describing road networks.

The Python API of CARLA, used by the client to control the simulation, relies on the OpenDRIVE definition of roads, lanes and junctions to make its decisions. The CARLA distribution comes with these, uninspiringly named, maps:

- **Town01**: A small, simple town with a river and several bridges.
- **Town02**: A small simple town with a mixture of residential and commercial buildings.
- **Town03**: A larger, urban map with a roundabout and large junctions.
- **Town04**: A small town embedded in the mountains with a special "figure of 8" infinite highway.
- **Town05**: Squared-grid town with cross junctions and a bridge. It has multiple lanes per direction. Useful to perform lane changes.
- **Town06**: Long many lane highways with many highway entrances and exits. It also has a Michigan left.
- **Town07**: A rural environment with narrow roads, corn, barns and hardly any traffic lights.
- **Town10**: A downtown urban environment with skyscrapers, residential buildings and an ocean promenade.

Each **Map** in CARLA has an associated set of **Spawn points**, which are locations in the Map recommended to spawn Actors to (and especially a Vehicle), in order to place them well with respect to each other. Another important concept related to Maps is the **Waypoint**, nothing more than a 3D point with information about the lane which is included into; they can be, however, combined to define paths into the road for vehicles to follow. Finally, when a number of Vehicles are placed into the simulation, how can they be moved? CARLA provides the **Traffic Manager**, a module that controls certain vehicles in a simulation from the client side. The Traffic Manager provides a CARLA simulation with realistic traffic conditions in an easy configurable way, for example it can activate lights of vehicles, set the minimum distance vehicles should maintain between them and the desired maximum speed. Another way of moving a vehicle that guarantees the user a more fine-grained control is by using **Agents**. An Agent allows a vehicle to either follow a random, endless route or take the shortest route to a given destination by obeying traffic lights and reacting to other obstacles on the road. An agent is more versatile than the standard Traffic manager since the user can control target speed, braking distance and tailgating behavior (and more) of the vehicle, achieving more complex and realistic driving behaviors. CARLA provides implementations for a Basic Agent, that roams around the map or reaches a target destination in the shortest distance possible by avoiding other vehicles and responding to traffic lights but ignoring stop signs, and a Behavior Agent, that can reach a target destination in the shortest distance possible by following traffic lights, signs, and speed limits while tailgating other vehicles.

(a) Town01

(b) Town10

Figure 2.9: Example of CARLA Maps

2.3.4 Weather

Weather conditions in CARLA come in the form of adjustable parameters that can alter the appearances of Maps by utilizing the underlining graphics power of Unreal Engine. These parameters are grouped into an object that needs to be passed to the World in order for the change to take effect, although, obviously, each Map has its own default weather configuration. Parameters include:

- Cloudiness: ranging from 0 or clear sky to 100 or completely cloud sky.
- Precipitation: rain intensity values, going from 0 (no rain) to 100 (heavy rain)
- Fog density: fog concentration or thickness, with values ranging from 0 to 100.
- Wind intensity: controls the strength of the wind, from 0 (no wind) to 100 (strong wind)

So far in the latest CARLA version weather conditions only affect RGB cameras, without affecting actors physics or other sensors (for example, a strong wind will only move trees of a map, without affecting vehicles trajectories). For convenience, CARLA already comes with a series of Weather presets ready be used: these are 27 in total, and each one is a combination of one of three daylight conditions (Sunset, Noon and Night) with weather (Clear, Cloudy, Soft rain, Mid rain, Hard rain, Wet cloudy and Wet). More complex weather scenarios, like weather changing as time passes in the simulation, needs to be scripted.

2.3.5 Sensors and Data

As said before, **Sensors** in CARLA are Actors, although a special type able to measure and stream simulation data. Since there are a lot of different types of sensors (which I will briefly present), also the type of data varies accordingly. The moment in which a sensor gathers data also varies by type: some collect data on every simulation step, others only when a certain event is registered. Despite their differences, each sensor follows a similar life-cycle pattern:

- **Setting**, in which the user sets the Blueprint attributes, such as the sensor tick, which specifies when the sensor should record data out of every simulated second. Other attributes are sensor-dependent and can greatly customize each type of sensor to the user's needs.
- **Spawning**, in which the sensors are placed in the simulation. Unlike others, however, sensors need to be attached to an actor, which is usually a vehicle.
- **Listening**, in which sensors process the collected data using a user-defined callback. The processing frequency can also be altered.
- **Data Storing**, in which processed data is stored on disk. Data can be tagged using information like frame and timestamp, for smart browsing in the future.

The CARLA library divides each sensor type into three categories: **Cameras**, **Detectors** and **Other**.
Cameras capture the simulated world from their point of view. Currently, there are 6 different cameras implemented:

- The **RGB camera** acts as a regular camera capturing images from the scene. Modifiable parameters for this camera include shutter speed, gamma and ISO. A set of post-processing effects can also be applied to enhance the image realism.
- The **Semantic camera** classifies every object in sight by displaying it in a different color according to its tags. Each tag corresponds to a class and is determined by the folder in which the object 3D model is located inside the project structure. This tag is encoded by the CARLA server in the image's red channel. The library also provides utilities to convert each tag to a color, for example, using the Cityscapes palette.
- The **Instance segmentation camera** classifies every object in the field of view both by class and also by instance ID. This implies that every object in the simulation is identified by both its class and its specific instance ID. The image tags are encoded, like in the Semantic camera, using the red channel, while the object ID is split between the blue and green ones.
- The **Depth camera** provides an image of the scene codifying the distance of each pixel to the camera (also known as depth buffer or z-buffer) to create a depth map of the elements inside. For visualization's sake, the image can be saved using the Depth and Logarithmic Depth color palettes.

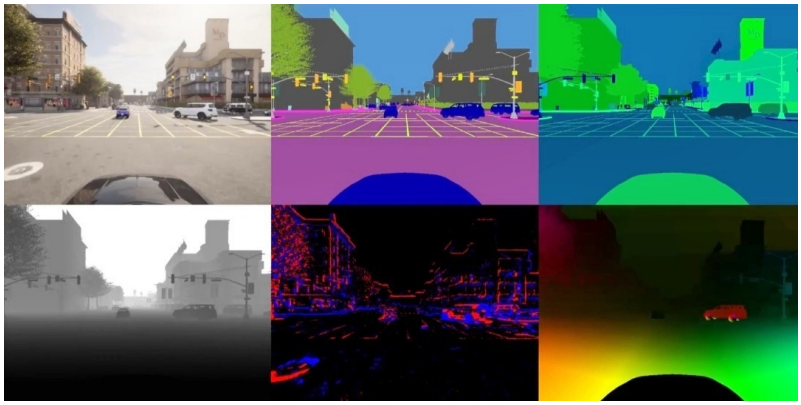


Figure 2.10: From left to right, top to bottom: RGB camera, Semantic camera, Instance camera, Depth camera, DVS camera and Optical flow camera

- The **Optical flow camera** captures the motion perceived from the point of view of the camera. Every pixel recorded by this sensor encodes the velocity of that point projected to the image plane.
- The **DVS camera**, also known as an Event camera, is a sensor that works radically differently from a conventional camera. Instead of capturing intensity images at a fixed rate, it measures changes of intensity asynchronously, in the form of a stream of events, encoding per-pixel brightness changes. This sensor can be customized by attributes such as positive and negative thresholds and refractory period.

Sensors in the category **Detectors** register data when the Actor they are attached to registers a specific event:

- The **Collision detector** registers an event each time its parent actor collides against something in the simulated world.
- The **Lane invasion detector** registers an event each time its parent actor crosses a lane marking, using the OPENDRIVE data associated with the current map.
- The **Obstacle detector** registers an event every time its parent actor has an obstacle ahead its way. Customizable attributes includes hit radius and detection distance.

Finally, sensors in the category **Others** include proprioceptive and ultrasonic sensor implementations:

- The **LIDAR** simulates a real rotating LIDAR by using ray-casting, a computer graphics and computational geometry technique used to determine the visibility of objects in a scene by tracing rays from a point of origin. This implementation contains a lot of customizable attributes, such as the number of points generated per second, the number of lasers used (channels), the detection range and the rotational frequency. Unfortunately, as of now, this version of LIDAR is not influenced by weather conditions of the simulation like a real sensor would.

- The **Semantic LIDAR** functions the same as the standard LIDAR, but includes the instance and semantic ground truth (the tag and instance ID of the Semantic/Instance camera) with the collected points.
- The **Radar** generates a conical field of view, which is translated into a 2D point map representing detected object along with their speed relative to the parent actor. Points collected this way consist of polar coordinates, distance and velocity.
- The **GNSS sensor** reports the current Global Navigation Satellite System (GNSS) position of its parent actor.
- The **IMU sensor** provides measures that a real accelerometer, gyroscope and compass would retrieve for the parent actor.

Chapter 3

Related works

In this chapter, datasets and methods aimed at Anomaly segmentation are introduced. This task is defined as precisely segmenting anomalies, that are patterns and elements which deviate from normality, by assigning an anomaly score to each pixel of an image. Segmenting anomalies is important because most state-of-the-art models for Semantic Segmentation are trained on a closed-set of well defined classes, while in the real world previously unseen objects can appear at any time; the failure to at least acknowledge the potential hazards can lead to unpredictable and potentially dangerous consequences.

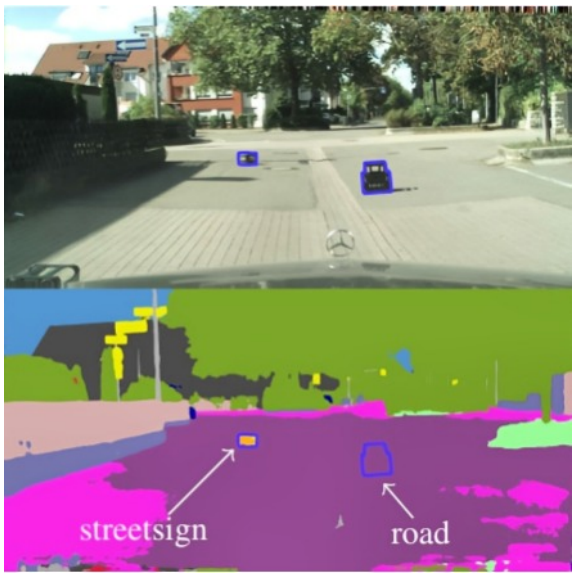


Figure 3.1: Example of a Segmentation model failing to detect objects on the road

3.1 The Fishyscapes Benchmark

Fishyscapes [12] was the first public benchmark for Anomaly segmentation of driving scenes. It consists of an evaluation suite and two different datasets: FS Static and FS Web. Each of these datasets is based on the validation set of Cityscapes (probably the most notorious and used dataset for Semantic Segmentation of road scenes)

(a) Image with overlayed object (b) Segmentation mask

Figure 3.2: Fishycapes Benchmark, FS Static example from the validation set

on which a series of objects have been overlayed on the original images, producing synthentic new images: in FS Static, objects from the PASCAL VOC not found in Cityscapes have been overlayed, while in FS Web objects are first crawled from the Internet using a changing list of keywords, processed and the overlayed. The interesting part of FS Web is that, at least in the original authors' intention, the dataset is updated every couple of months with new objects, making it dynamically changing. The benchmark, acknowledging that most methods used for Anomaly segmentation produce uncertainty scores, uses a threshold to transform these into binary scores that indicate wether a pixel of the image is anomalous or not. In the related paper, the authors also proceed to test the state-of-the-art models of the time on Fishyscapes, concluding that most of the methods required some modification of the loss that reduced Semantic Segmentation performance and that models trained with supervision from OOD (Out of Distribution) data consistently outperformed unsupervised equivalents.

3.2 The SegmentMeIfYouCan Benchmark

The SegmentMeIfYouCan benchmark [13] builds upon Fishyscapes and other previous road anomaly datasets, trying to improve the diversity of scenarios and objects included in the images. This works comes with a public leaderboard, an evaluation suite and defines two tasks to tackle, with associated datasets: Anomalous Object Segmentation and Road Obstacle Segmentation, both consisting of real collected images, unlike Fishyscapes where the images are synthetic. In Anomalous Object Segmentation, any object that doesn't belong to any of the classes used in the training process (in this case, the one from Cityscapes) can appear anywhere in the image; the associated dataset, called RoadAnomaly21, contains 100 images with an increased (from previous works) amount of different objects, 26. Since, according to the authors, sometimes defining what an anomaly is can be "fuzzy", the Road Obstacle Segmentation task considers every object present on the road, be it from known or unknown classes, an anomaly. The associated dataset, RoadObstacle21, contains 371 images with 31 different objects. The SegmentMeIfYouCan benchmark also defines two different metrics to challenge submitted models: one, similar to Fishyscapes, evaluates anomaly scores on a per-pixel bases, another, instead, focuses on evaluating them on a component level. The component level metrics are

(a) RoadAnomaly21

(b) RoadObstacles21

Figure 3.3: Examples from the test sets of the SegmentMeIfYouCan benchmark

needed for small anomalies, that by occupying a small set of pixels do not contribute that much to a negative per-pixel score, if mistaken for normal objects. Finally, the paper's authors tested various types of models on RoadAnomaly21 and RoadObstacle21, finding that most methods show a significant drop in performance when evaluated on the component-wise metrics; clearly, the authors concluded, there is still much work to do to deliver models able to be deployed safely for autonomous tasks.

3.3 The StreetHazards dataset

StreetHazards [14] is an Anomaly Segmentation dataset that is built using the CARLA simulator, by leveraging its customization capabilities to insert a wider variety of anomalies in a realistic scenario. According to the authors of this paper, the usage of a simulated environment solves the problem of inconsistent lightning and textures that synthetic datasets based on composed images, such as Fishyscapes (where objects from different sources are overlayed with a real image) present, giving the model unwanted cues that the object is, in fact, anomalous. The authors used 3 maps from CARLA to collect a training set of 5125 images and an additional one for the 1,031 validation images, leveraging inbuilt RGB cameras sensors and Semantic cameras. All the training and validation images do not have inserted anomalies. The test set filled with anomalies is instead built on two different maps from the ones used for training/validation, and it contains 1500 images in total.

3.4 Detecting outliers in mask-level predictions

In a work titled "On Advantages of Mask-level Recognition for Outlier-aware Segmentation" [15], researchers used a Mask2Former architecture [**cheng2021mask2former**] combined with a custom "anomaly detector" that extends the capabilities of the standard mask-based architecture from simple semantic segmentation to what they called "outlier-aware semantic segmentation". The authors proposed (among others) two different approaches that use a scoring function that maps each pixel, coming from predictions by Mask2Former, to its out-of-distribution score. The first approach is called **Anomaly of Ensembled Mask-wide predictions (AEM)** and

is formally defined as:

$$s^{AEM}(\mathbf{X})[w,h] = -\max_{k=1..K} \sum_{i=1}^{N} \mathbf{M}_i[w,h] * P_i(Y=k|\mathbf{M}_i)$$

where $\mathbf{X}$ is the input image of size WIDTHxHEIGHT, (w, h) is a pixel belonging to that image, $\mathbf{M}$ is a pixel mask of size WIDTHxHEIGHT from Mask2Former, $P_i(Y=k|\mathbf{M}_i)$ is the class prediction distribution for a mask. Additionally, K is the number of semantic classes of the semantic segmentation task and N is the total number of output masks of Mask2Former. The second approach is called **Ensemble over Anomaly scores of Mask-wide predictions (EAM)**

$$s^{EAM}(\mathbf{X})[w,h] = \sum_{i=1}^{N} \mathbf{M}_i[w,h] * (-\max_{k=1..K} P_i(Y=k|\mathbf{M}_i))$$

using the same notation as before. According to the results presented by the researchers who authored the paper, the EAM scoring function outputs lower anomaly scores on semantic boundaries between in-linear and outlier pixels. This helps EAM reduce false-positives in these regions with respect to the AEM approach.

3.5 Mask2Anomaly

Mask2Anomaly [16] is an anomaly recognition model that leverages the mask-base segmentation novelties introduced first in MaskFormer and refined in Mask2Former [10], adapting it to let it perform well not only on closed-set in-distribution (ID) data, but also on out-of-distribution (OOD), not seen before during training, data. Mask2Anomaly can jointly perform three different anomaly tasks with a single common architecture that are, in order of complexity:

- **Anomaly segmentation (AS)**, that focuses on segmenting object from classes that were absent during training, generating an output map that identifies the anomalous pixels
- **Open-set semantic segmentation (OSS)**, that evaluates a segmentation model's performance on both anomalies and known classes. OSS ensures that when training an anomaly segmentation model, its performance on known classes remains unaffected.
- **Open-set panoptic segmentation (OPS)**, that simultaneously segments distinct instances of unknown objects and performs panoptic segmentation for the known classes.

Enhancements towards detecting anomalies made by Mask2Anomaly with respect to Mask2Former have been made at the architectural, training and inference level. First, the architecture of Mask2Former was modified extending the masked-attention mechanism (MA), that attends primarily to foreground regions of images, with a global masked-attention (GMA), that also considers background regions. This was

necessary because anomalies may also appear in the background, still this novelty does not significantly impact the benefits of Mask2Former's masked-attention in terms of training convergence and semantic segmentation performance. Second, during training, a mask contrastive learning procedure is utilized in order to maximize the difference between anomaly scores for OOD data, which should be high, with scores for ID data, which should be as low as possible. This is performed using an additional outlier dataset and performing an additional fine-tuning procedure on top of training, that consists in combining the training ID images with objects cut from the OOD data (anomaly-mix). Lastly, for inference, a refinement mask is used that filters out most false-positives (FP) from the predicted anomaly-mask. This approach was inspired by Panoptic-segmentation (that groups objects in a scene in "things", countable objects, or "stuff", amorphous regions). The filtering builds a binary mask where the unwanted "stuff" pixel masks have value 0 and 1 otherwise. Since this approach is tailored for road-type scenes and most obstacles are placed on the road, the "Road" regions are excluded from the filtering of the refinement-mask.

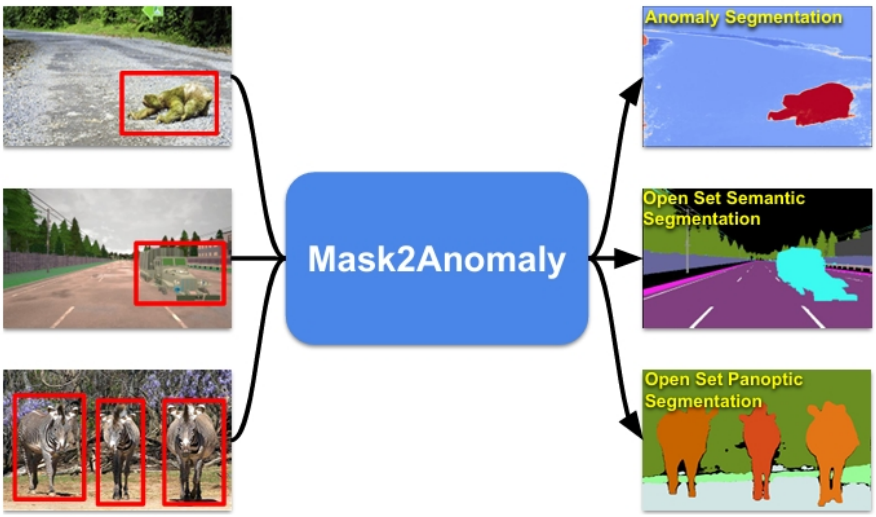


Figure 3.4: Mask2Anomaly

Chapter 4

Dataset Creation

In this chapter, I will describe how the CARLA simulator was used to collect the datasets used in training and evaluating models for anomaly segmentation. The main desired characteristics for the datasets that guided the work and influenced choices made were the following:

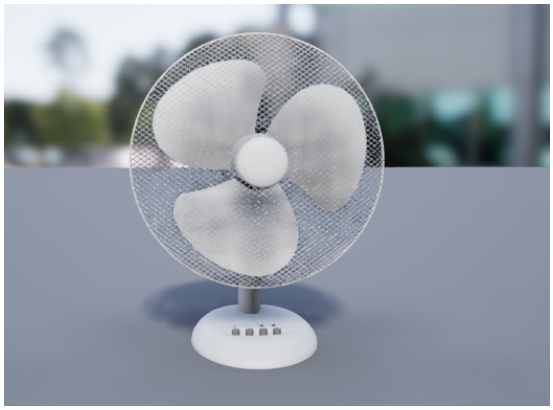
1. Given that most of the datasets for anomaly segmentation in road scenes consist of new objects, unseen during training, mostly placed on the road, to expand the notion of what an anomaly is. For example, to also consider as anomalies objects already seen during training, but in strange configurations, like a fallen tree on the road or an improperly parked car.

2. Most of the datasets collect only RGB data, and the ground truth needs to be derived from manual labeling or from some automated procedure (usually less precise and realistic). Since CARLA provides implementations for other sensors like LIDAR (although imperfect), to expand the data collection to consider other modalities like 3-D point clouds.

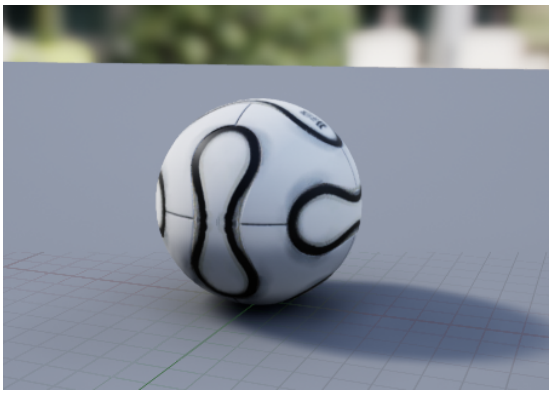
3. Collecting a continuous stream of data just as a car on the road would. This could enable models that can leverage temporal information to recognize anomalies better.

4.1 Adding objects to CARLA

The CARLA build already provides objects called **Props**, which model the various structures and items that can be found on or near roads, such as benches, boxes, bins, debris or trash. They are spawned in the simulation as actors, like vehicles and walkers, by selecting the corresponding Blueprint from the Blueprint library. However, not all the props were considered useful for our scenario: some objects were too big and designed to be spawned as map-customization tools outside the road (like, for example, food carts and fountains), while others focused only on certain categories like road construction items and garbage. For this reason, I decided to try and add new 3-D models to hopefully enhance the different categories of objects that could be spawned in the simulation. To this end, was extremely useful the website CGTrader, a 3-D marketplace for content creators to share their work for



(a) A fan 3-D model



(b) A football 3-D model

Figure 4.1: Examples of objects added into CARLA

others to buy; apart from paid content, there is also an extensive library of free-to-use objects to download after registering to the site. Each downloaded object needed to be in the FBX (Filmbox) format, which is the format adopted by Unreal Engine, the backbone of CARLA, for importing 3-D models. Not all the models downloaded from CGTrader were ready to be imported nicely into Unreal Engine; some came with embedded textures, a type of file that specifies the surface of the 3-D model, and some came with textures that needed to be manually imported and added to the model in Unreal Engine. In order to define the surface properties of a 3-D model or Mesh, Unreal Engine uses a structure called a Material, which can control and combine textures to create a final surface for the mesh. Its main controllable attributes are:

- **Base color**: the main color or texture of the surface.
- **Metallic**: defines how metal-like the surface is, from 0 (similar to plastic) to 1 (completely metallic)
- **Roughness**: how shiny or matte the surface is, from 0 (completely shiny / reflective like glass) to 1 (completely matte or completely not reflective)
- **Normal**: A texture that adds small surface details (like bumps or grooves) without modifying the underlining geometry (useful, for example, to recreate brick or wood textures)

This Material structure was therefore used to combine the various downloaded textures to create a final realistic surface to be added to the 3-D skeleton of an object. These newly added objects needed to be registered in a JSON file in order for them to be included in the Blueprint library as props and become spawnable inside CARLA maps.

4.2 The CARLA Client

In this section, I will describe how the CARLA client, the script that controls the CARLA simulation, was crafted in order to allow enough versatility to record both

the datasets for training, with no anomalies, and the one for evaluation, with added anomalies. After development, its main features were to control the simulation by connecting to a CARLA server, to spawn a variable numbers of actors like vehicles and walkers, to attach sensors for recording data and to spawn anomalies (props), all in varying maps and weather conditions. This section continues by describing the different parameters that the client scrips offers to customize the simulation, the steps that it follows to achieve the desired behavior, and finally the characteristics of the training and evaluation datasets that were collected by using it.

4.2.1 Client parameters

The user of the client script can specify parameters and configuration for the simulation in two different "places": a JSON file, mainly used for lengthy and more structured data, and command-line arguments to be specified when running the script. The JSON file is to be structured in this way (in order for the script to process it correctly):

- **"maps"**: this field is an array of strings in which the names of selected maps for the simulation should be specified (e.g. Town01, ..., Town10)
- **"weather"**: this field is an array of strings in which the names of the desired weather conditions for the simulation should be specified (e.g. ClearSunset, CloudyNoon, SoftRainNight). They correspond to weather presets offered by CARLA.
- **"props"**: this field is an array of strings in which the names of the desired props for the simulation should be specified (e.g. fan, football, mattress). In particular, the Blueprint library uses this strings to locate the 3-D prop model to spawn in CARLA.
- **"vehicles"**: this field is an array of objects. Each object should specify a vehicle configuration, composed of the blueprint id of the car for the blueprint library (e.g vehicle.tesla.model3) and a list of sensor objects. Each sensor is also defined by its blueprint id (e.g sensor.camera.rgb or sensor.other.lidar), the "transform" field that specifies the x-y-z location in space relative to the parent object (the car), the "parameters" field that specifies values for the sensor parameters (which differ greatly by sensor, e.g field of view for the RGB camera and rotational frequency for the LIDAR) and a "name" which is used to create different directories to save data. In summary, each car should have its own sensors configured, because differences in size between 3-D models make impossible to attach the sensors always in the same place in every car.

The command-line arguments that the user can specify when running the script are instead:

- **–host** and **–port** define the IP address of the machine running the CARLA server (that can be the same running the client, or another, potentially in another network) and the TCP port used for connection.

- –**verbose** specifies the verbosity of the script output. Choices are between 0, completely silent, 1 that displays basic simulation infos like iteration and vehicles number, 2, that displays also debug info like when there is a spawn collision or the elapsed seconds for each iteration.

- –**config-file** that specifies the name of the JSON file containing the simulation configuration

- –**runs** that specifies the number of times the simulation will run (iterations).

- –**seconds** specifies how long each run should last. It is worth noting that this is not real time but simulated time, therefore influenced by how fast the CPU and GPU is processing the simulation, and it can vary greatly each time.

- –**vehicles** specifies the number of vehicle actors that will be spawned in each simulation run.

- –**walkers** specifies the number of walker actors that will be spawned in each simulation run.

- –**record** specifies if simulation should be recorded using the sensors specified in the vehicle configuration.

- –**folder** specifies the name of the folder in which recorded sensor data should be stored.

- –**game-window** opens a window that displays a camera following the ego-vehicle in its path.

- –**anomalies** specifies the maximum number of anomalies that will be spawned in each run. Since all anomalies are spawned in the same place, values for this parameter should stay low in order to not block the car in its path (for example, 1 to 3 anomalies).

4.2.2 Client steps

The algorithmic steps that the CARLA client follows serve two purposes: first they initialize the simulation with the desired configuration, then they loop as long as the number of iterations desired is reached, called a "game loop". More specifically, the script first gathers parameters from the command line, process them, then proceeds to read the JSON file, as the path of this file needs to be also specified from the terminal. If the configuration is found correctly, the script tries to connect to the CARLA server using the IP address and port specified by the user and retrieves the corresponding World and Blueprint library objects that are to be used to find blueprints and spawn actors. The script then loops as long as the number of –runs that the user specified is reached, each time repeating the following steps.
First, a map and a weather condition is randomly extracted from the "maps" and "weather" configuration fields; this has the intent of recording data as diverse as

possible within the tools provided by the CARLA simulator. The ego-vehicle configuration is then chosen randomly from the possible "vehicles" configurations specified; using the corresponding blueprint ID the vehicle blueprint is retrieved by the Blueprint library and then spawned using a random spawn point in the map, by means of the World object. If the weather condition includes Night or Rain, the vehicles lights are turned on. A Behavior Agent is used to control the vehicle, and as a destination to reach a random spawn point of the map is chosen. The simulation is then populated with as many vehicles and walkers (pedestrians) as the user specified: blueprints are once again randomly extracted from the ones provided by CARLA in the Blueprint library and the associated attributes are randomly assigned (color for cars, as an example) to enhance the diversity of actors in the scenes. If the user wants to record some data, using the "–record" option, the list of sensors is retrieved from the vehicle configuration; each sensor needs a callback that specifies what to do with the collected data produced on the CARLA server, for this I defined a callback for each sensor (using guidelines and examples from the CARLA documentation) that generally does some processing to the data and then saves it on disk. This way, each sensor recording is tagged and placed in a directory named after the sensor using the format "RUN_MAP_WEATHER_FRAME", where "RUN" is the iteration number, "MAP" and "WEATHER" are the selected map and weather condition of the iteration and "FRAME" is the frame number, . As an example, an image from an RGB Camera tagged as "1_Town05_WetCloudyNight_00024873.png" was collected on the first iteration, the map was Town05 and the weather preset was WetCloudNight.
The last initialization step occurs only if the user has specified the number of anomalies desired, using the "–anomaly" argument; in this case, a random spawn point is chosen from the pre-defined path that the Agent plans to follow and the anomalies are spawned there. This way, the car will always reach the anomalies and bump into them. As anomalies are nothing but prop objects in the CARLA simulation, they are spawned using the Blueprint library and ids provided by the user in the JSON file, field "props".

In the "game-loop", the client script advances the simulation by communication with the server in synchronous mode. The Agent controls the ego-vehicle physics moving it one waypoint at a time following its pre-planned path. If there is no anomaly placed in the agent's path, data is always recorded at the specified rate; if anomalies are placed, instead, data is collected only when the anomalies are near the ego-vehicle, by appearing in its field of view. More specifically, this behavior is implemented by using an Instance camera with a custom callback that compares actor ids of the prop anomalies with actor ids collected in data: if a match is found, the client is notified by means of a flag and it starts recording, and when the anomaly disappears from view, it stops again. This design choice was necessary to collect small episodes for the anomaly dataset consisting of the ego vehicle reaching and colliding with the anomalies. The game-loop can end in two ways: if the user has specified a number of "–seconds", the run will only last that amount and then end, if not, it will end when the Agent has reached its destination. Before starting a new iteration, a cleanup phase is executed when all vehicles, walkers and sensors are "destroyed" from the simulation. In summary, the client script repeats initial-

ization steps and the game-loop for every iteration and when the desired number of iterations (runs) is reached, it disconnects from the CARLA server and it ends.

4.3 Recording configuration

Here are listed all the parameters that I specified in the JSON file and on the command-line during the recording of the two datasets, which I named CARLA-Normal and CARLA-Anomaly. Some of these parameters are in common between the two datasets, some are not. Starting with sensors in the vehicle configuration:

- 3 **RGB Cameras**: one in the center, called RGBCenterFront, one on the left, RGBLeftFront and one on the right, RGBRightFront, of the vehicle. The image resolution set was 1280x720 and the field of view (FOV) was set to 90 degrees. I also set the sensor attribute "motion blur" to 0 to obtain sharper images
- 1 **LIDAR** sensor on top of the vehicle, LIDARTop. The LIDAR was configured following suggestions from CARLA example scripts with 64 channels (the number of lasers used) generating 500000 points per second, covering a range of 100 meters. To generate a full-circle point-cloud per simulation step, the rotational frequency of the LIDAR is always set as the number of frames per second of the CARLA client (fps).
- 3 **Semantic/Anomaly cameras** and 1 **Semantic LIDAR** to capture ground truth for cameras and LIDAR. Here, the configuration changes between the two datasets. In CARLA-Normal the ground truths consist of semantic maps with values ranging from 0 to 28, indicating to which of the 29 CARLA classes each pixel belongs to, recorded using Semantic cameras. In CARLA-Anomaly the ground truths consist of anomaly binary maps, where each pixel is 0 if not anomalous and 1 if anomalous, plus additional semantic maps similar to CARLA-Normal but with an added 30th anomaly class (indexed 29). The anomaly ground truths were recorded using an Instance camera with a custom callback that I named an "Anomaly" camera to differentiate between the two. The Anomaly camera uses the IDs of anomalies (which are nothing but prop actors with an associated ID) to produce the two segmentation maps starting from the IDs of the actors in the scene that a standard Instance camera captures.

The other parameters in the JSON configuration were:

- **"maps"**: Town01, Town02, Town03, Town04, Town05, Town06, Town07 and Town10HD (all of the maps provided by CARLA)
- **"weather"**:

  {Sunset, Noon, Night} × {Clear, Cloudy, SoftRain, MidRain, HardRain, WetCloudy, Wet}

  (all the Weather parameters provided by CARLA)

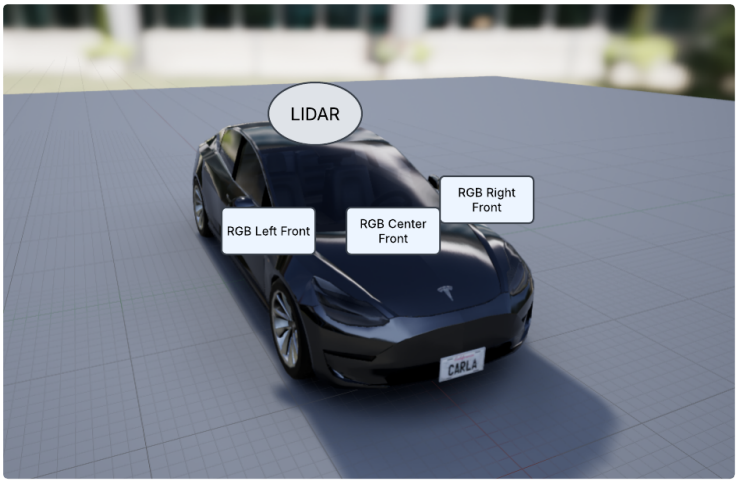


Figure 4.2: Configuration used to record the dataset

- **"props"**: baseballbat, basketball, beerbottle, bicycle, dumbbell, football, mattress, skateboard, television, tire, woodpalette, pillow, guitar, servicetrolley, fan, ladder, officechair, warningaccident, warningconstruction, constructioncone, streetbarrier, plasticchair, trashbag, box01, colacan, bin, clothcontainer, container, glasscontainer, trashcan01, wateringcan, haybale, trafficcone01 *(these are the objects that I added into CARLA plus other prop objects already packaged that I considered viable as road obstacles)*
- **"vehicles"**: Tesla Model3 with the sensor configuration previously described

and the command-line arguments were:

- –**runs** 50 for CARLA-Normal and 25 for CARLA-Anomaly
- –**seconds** 0 since each simulation lasts as longs as the vehicle reaches its destination
- –**vehicles** 50 for CARLA-Normal and 0 for CARLA-Anomaly. This was the only way I found to prevent other vehicles to hit anomalies before the ego-vehicle.
- –**walkers** 30
- –**record** set to record the data using the specified sensors
- –**anomalies** 3 for CARLA-Anomaly, clearly none for CARLA-Normal

4.4 CARLA-Normal

CARLA-Normal is the dataset used to train models on in-distribution data. Of the 50 CARLA simulation episodes recorded, 40 were selected for the training portion

of the dataset and the remaining 10 for validation. This way, CARLA-Normal training consists of a total of 6.099 images, combined from the 2.033 images or frames collected by each one of the three RGB cameras (Center, Left and Front), and the corresponding 2.033 point-cloud recording of the LIDAR sensor. Likewise, CARLA-Normal validation consists of 549 images per sensor; by combining, once again, images from each RGB camera we obtain 1.647 images in total. Each image and point-cloud has an associated ground-truth, recorded using the corresponding Semantic cameras and LIDAR.

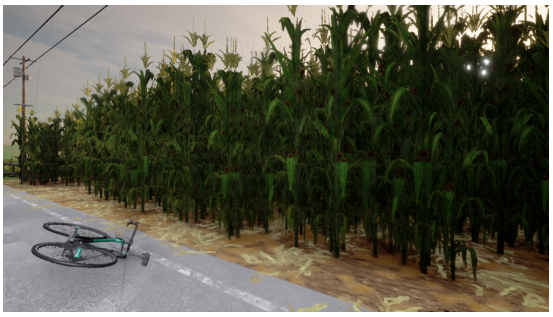
(a) RGB image (b) Semantic ground-truth

Figure 4.3: Example of image ground-truth pair from CARLA-Normal

4.5 CARLA-Anomaly

CARLA-Anomaly is the dataset used to evaluate models on out-of-distribution (OOD) data. The 25 CARLA simulation episodes produced 1.273 sensor recordings, that, in the case of images, account for a total of 3.819 images by combining all the RGB cameras. Each image is paired with the corresponding semantic and anomaly mask (as explained in the previous sections); the point-cloud recordings also have semantic ground truth as in CARLA-Normal. Each episode depicts the ego-vehicle reaching and colliding with a variable number of anomalous objects (set to 3 maximum in the configuration). As sensors in the script stop recording once the anomalies stop being in the field of view of the ego-vehicle, each episode has a short recorded duration. As shown in Figure 4.4, where the ego-vehicle is reaching a bike object that was added to CARLA, sometimes only certain cameras are actually recording anomalies because of their different placements. Therefore, not all frames in CARLA-Anomaly (be it images or point-clouds) contain anomalies, but most are.

(a) Left-Front RGB Camera view

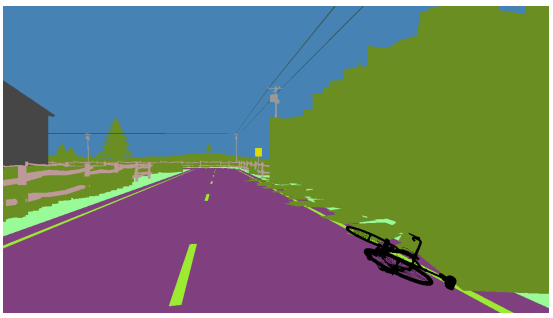


(b) Right-Front RGB Camera view

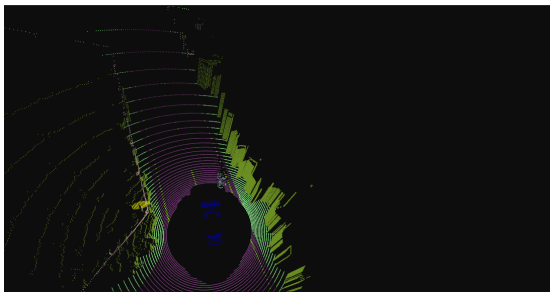
(c) Center-Front RGB Camera view

Figure 4.4: Example of recorded frame in the CARLA-Anomaly dataset

(a) A visualization of an anomaly mask



(b) A visualization of a segmentation mask



(c) A visualization of a Semantic LIDAR point-cloud

Figure 4.5: Example of ground truths of CARLA-Anomaly

Chapter 5

Experiments and Results

In this chapter, I will describe how some models were first trained using CARLA-Normal and then evaluated on the task of Anomaly segmentation on CARLA-Anomaly. All the training procedures and experiments were carried out on a machine located in the Visual And Multimodal Applied Learning Laboratory (VANDAL) of Politenico di Torino, equipped with an NVIDIA RTX 3090 graphics card (GPU).

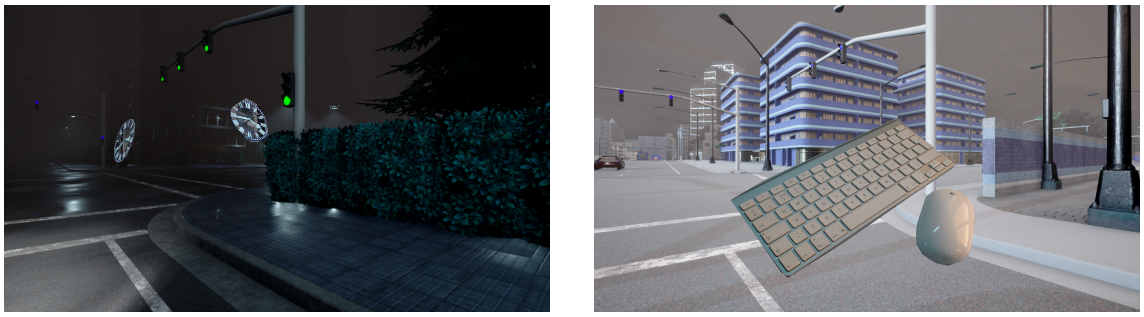
Parameter	**Value**
BATCH_SIZE	4
LEARNING_RATE	0.0001
WEIGHT_DECAY	0.05
ITERATIONS	60000
CROP_SIZE	(360, 760)
BACKBONE	Swin-B

Table 5.1: Training configuration

5.1 Training

Since methods based on ensembles of masks, namely AEM and EAM, implement a novel scoring anomaly function on top of existing mask transformer architectures, in order to perform experiments on CARLA-Anomaly was first necessary to train a Mask2Former model from scratch using the corresponding in-distribution dataset CARLA-Normal. The Mask2Anomaly method is also based on Mask2Former, but as highlighted in the "Related works" section, it modifies the original architecture by substituting Masked-attention with a tweaked version called Global masked-attention, therefore needing a different training procedure. The two Mask2Former variants were trained using the same configuration and hyperparameters listed in Table 5.1 for 60.000 iterations using the AdamW optimizer. Since the RTX 3090 comes with 24 GB of VRAM, the batch size was reduced to just 4 images per batch and the crop size was set to 360x760 to make model and data fit into memory. After training,

the models obtained similar performances on the CARLA-Normal validation set: the "vanilla" Mask2Former scored an mIoU of 77.1% while the Mask2Anomaly variant scored a slightly less 76.9%. Additionally, Mask2Anomaly's variant of Mask2Former requires fine-tuning on outlier images to achieve full potential on anomaly segmentation. As such, 300 images from the COCO dataset with respective annotation were used; during this process, images from CARLA-Normal and COCO where mixed together (anomaly-mix), as shown in Figure 5.1, and the model updated. In the end, the model did not suffer from a drop in mIoU on ID data, but performances on anomalies where greatly improved. Figure 5.3 (heatmaps on the anomaly scores, thresholded at 0.5, blue is ID and red is OOD) shows how Mask2Anomaly handles the scene differently before and after finetuning: the bicycle is initially unrecognized as anomaly (as it should, since the Bicycle class is part of CARLA's training classes), but after finetuning, it is labeled strongly as anomaly. Interestingly, the finetuned version suffers from false-positives in this scene, labeling the region between the road and the crops as anomalous.



(a) (b)

Figure 5.1: Example of mixed images from CARLA-Normal with COCO objects, used during the fine-tuning of Mask2Anomaly

In Figure 5.2 a series of predictions by the trained Mask2Former are shown. The first pair of images show that the model performs really well in in-distribution CARLA-Normal data. The second and third pair instead show of the model handles data from CARLA-Anomaly: when the anomalous object is a bike, an object that belongs to the training classes but still classifies as anomaly because of its road placement, it is correctly segmented by Mask2Former as a Bycicle class but when the anomalous objects were not included in training classes (trashcans and ball) the model is not secure in its predictions, labeling the objects as belonging to different classes.

5.2 Inference

5.2.1 Metrics

All the experiments carried out on CARLA-Anomaly have been evaluated using two Anomaly segmentation's pixel-wise metrics that evaluate each pixel independently by assigning a score from 0 to 1, the AuPRC and the FPR@95. The **Area under**

The Precision-Recall Curve (AuPRC) is a metric that estimates, usually by trapezoidal rule, the area under a Precision-Recall curve (PRC), a curve plotted by calculating Recall and Precision values for a series of different thresholds, and placing them respectively on the x and y-axes. This metric is much more informative when used in evaluating a task with a highly unbalanced dataset, like in Anomaly Segmentation, where much of the image pixels are usually normal.

In this context, Precision is defined as the fraction of correctly labeled anomalous pixels out of all the pixels labeled as such by the model. It was historically defined with the concepts of True Positives (TP) or the fraction of positively labeled sample that were really positives, False Positives (FP) or the fraction of positively labeled samples that weren't really positives, and their likewise counterparts True Negatives (TN) and False Negatives (FN). Considering an anomalous pixel as a Positive instance:

$$\begin{aligned} PRECISION(t) &= \frac{|Y_a \cap \hat{Y}_a(t)|}{|\hat{Y}_a(t)|} \\ &= \frac{TP}{TP + FP} \end{aligned}$$

Where t is the chosen threshold, Y_a is the set of anomalous pixels in a ground truth label $Y = \{Y_a, Y_n\}$ that contains anomalous and normal pixels, and $\hat{Y}_a$ is the set of pixels labeled as anomalous by the model. Recall is defined as the fraction of correctly labeled anomalous pixels out of all the anomalous pixels (in the ground truth, not the ones labeled by the model). Again considering anomalous pixels as positive instance, Recall is defined as:

$$\begin{aligned} RECALL(t) &= \frac{|Y_a \cap \hat{Y}_a(t)|}{|Y_a|} \\ &= \frac{TP}{TP + FN} \end{aligned}$$

with the same definitions as before.

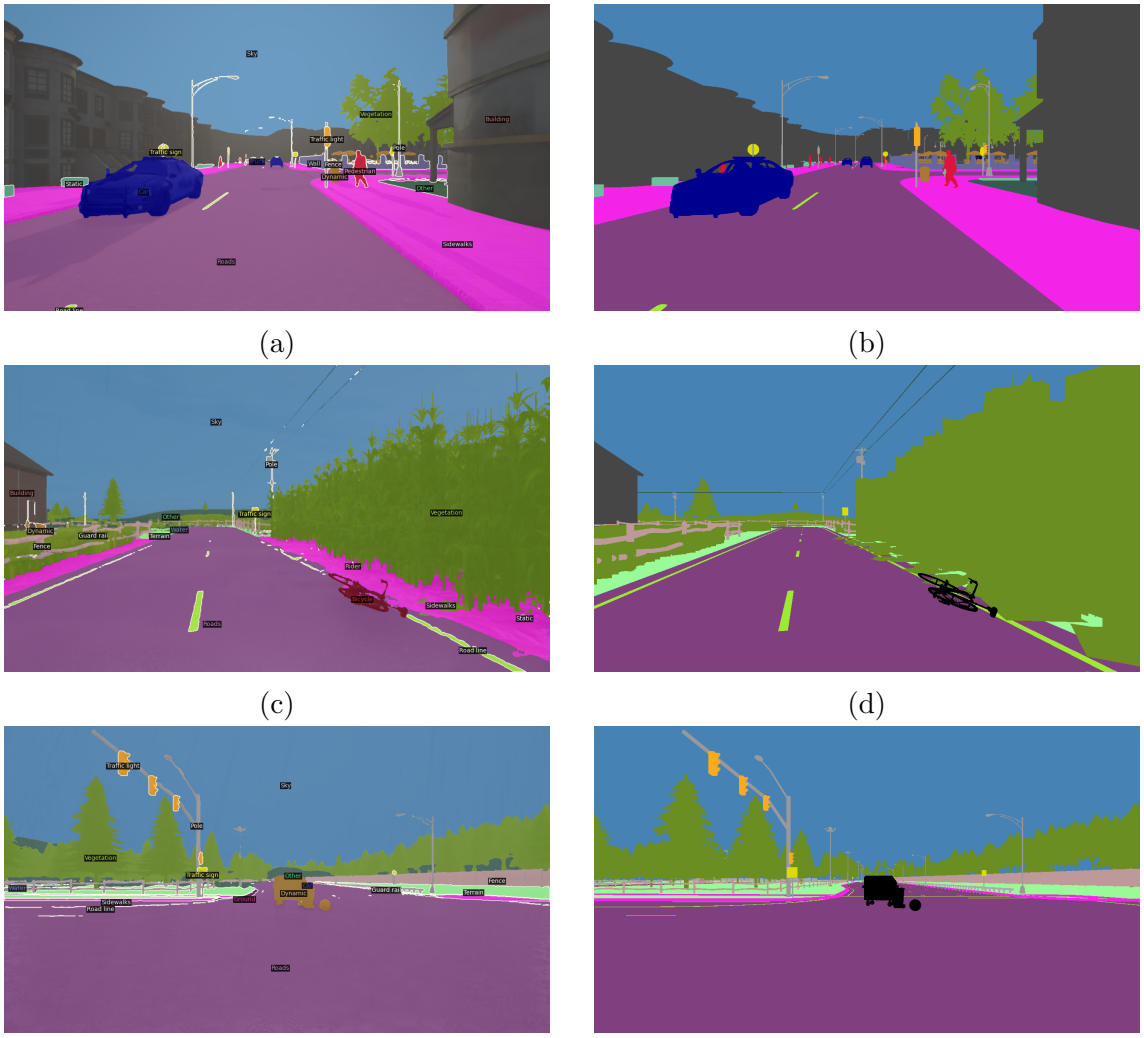
In safety-critical scenarios such as autonomous driving, another metric called the **False Positive Rate at a True Positive rate of 95%** (FPR_{95}), shortened as **FPR@95**, is used to evaluate how many pixels are labeled as anomalies while being really normal to achieve a correct labeling of 95% of anomalies that are really anomalies. It is formally defined as:

$$\begin{aligned} FPR_{95}(t^*) &= \frac{|Y_n \cap \hat{Y}_a(t^*)|}{|Y_n|} \\ &= \frac{FP}{FP + TN} \end{aligned}$$

where t^* is the threshold needed to achieve a True Positive rate of 95%.

5.2.2 Experiments

At first, the three Anomaly Segmentation models, AEM EAM and Mask2Anomaly where evaluated on the full CARLA-Anomaly dataset. Additionally, Mask2Anomaly



(a) (b) (c) (d) (e) (f)

Figure 5.2: Mask2Former predictions and ground truths for CARLA-Normal and CARLA-Anomaly

(a) Pre-finetuning (b) Post-finetuning

Figure 5.3: How predictions change with the finetuning of Mask2Anomaly

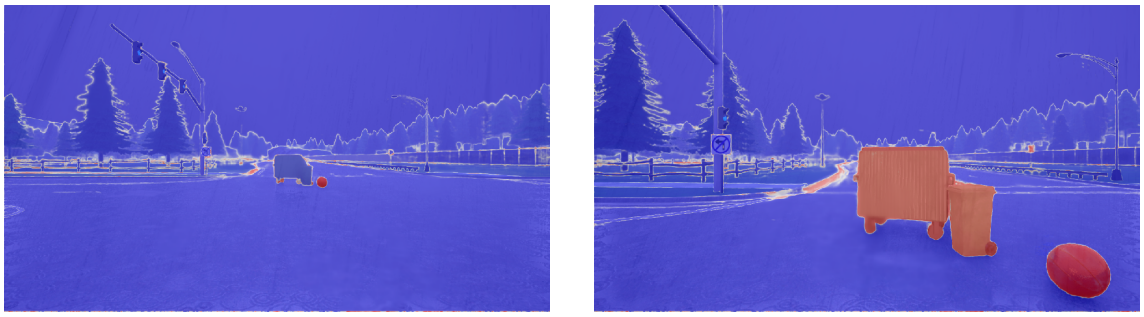
without the anomaly-mix finetuning procedure was included in the evaluation as a baseline and to confirm that the OOD-detection modules where working correctly. All results are shown in Table 5.2. By looking at metrics performances, it is clear that AEM and EAM are the worst performing methods out of all, even less performing than the un-finetuned Mask2Anomaly version. With the fine-tuning, Mask2Anomaly achieves the best performance out of all methods tested on CARLA-Anomaly; it does suffer, however, from large false positives. In Figure 5.4 are shown examples of Anomaly segmentation by Mask2Anomaly: in the first figure, Mask2Anomaly fails to recognize the anomalous trash container at a distance, while the ball is correctly labeled; in the second figure, Mask2Anomaly correctly labels all anomalies, this time nearer to the camera. This behavior is repeated in other episodes of the dataset. Going further, I decided to focus only on Mask2Anomaly (who gave the most promising results) and analyse more how the different characteristics of the dataset affect the model's abilities to correctly label anomalies. In a first analysis, CARLA-Anomaly was evaluated separately on the three differently placed RGB cameras on which the dataset was collected. Interestingly, the RGBRightFront camera split seems to be the easiest, and the model has very low number of false-positives. This could be partially explained by the fact that this split has the lowest number of anomaly pixels out of all, due to the anomaly placement that occurred during simulation:

- Center: 1.52% anomaly pixels
- Left: 1.38% anomaly pixels
- Right: 1.14% anomaly pixels

Another analysis divided the dataset into three splits based on light conditions of the weather that occurred during the CARLA simulation: Sunset, Noon and Night. Caused by the random extraction of the weather parameters, Noon and Night have double the frames of the Sunset split. Here, Mask2Anomaly performed best when evaluated on the Sunset split, achieving the best performance of all on CARLA-Anomaly; the Night split was also an easier one for the method, while again of the Noon split Mask2Anomaly suffers greatly from large false-positives. Finally, the last analysis was performed by splitting the data set into three subsets depending on the weather condition: Rain, Clear or Cloudy / Wet. Here, the most difficult split was the Clear, while the best performance was obtained in the Cloudy / Wet split. By examining the anomaly heatmaps visualizations, it is not clear why the Mask2Anomaly model presents these behaviors according to different characteristics of the dataset; further analysis is certainly necessary to understand it better.

	AuPRC ↑	FPR@95 ↓	Sensor	Weather	Frames
Mask2Former AEM	6.9	99.9	*	*	3819
Mask2Former EAM	7.1	99.8	*	*	3819
Mask2Anomaly (no fine-tuning)	10.0	98.5	*	*	3819
Mask2Anomaly	64.0	95.0	*	*	3819
Mask2Anomaly	62.0	97.1	RGBCenterFront	*	1273
Mask2Anomaly	83.2	12.3	RGBRightFront	*	1273
Mask2Anomaly	53.4	95.7	RGBLeftFront	*	1273
Mask2Anomaly	**84.6**	**9.1**	*	Sunset	726
Mask2Anomaly	58.3	97.2	*	Noon	1509
Mask2Anomaly	61.7	9.8	*	Night	1584
Mask2Anomaly	67.5	17.6	*	Rain	648
Mask2Anomaly	32.4	21.7	*	Clear	351
Mask2Anomaly	80.4	10.7	*	Cloudy, Wet	1581

Table 5.2: Anomaly Segmentation evaluation on CARLA-Anomaly and subsets



(a) (b)

Figure 5.4: Example of failure and success in Anomaly segmentation by Mask2Anomaly

Chapter 6

Conclusions

In conclusion, this thesis work was aimed at creating a dataset of road scenes that included various kinds of anomalies with the goal of benchmarking methods for Anomaly segmentation. To achieve this, I used CARLA, an open-source simulator that comes pre-packaged with maps and assets that lets users setup a realistic simulation to record data in. However, in order to insert anomalies into the environment, it was necessary to first search for free 3-D models of various kinds of object on the Internet and then integrating them into Unreal Engine and CARLA. To control the simulation in a predictable and customizable way, was also necessary the development of a script that, by utilizing the CARLA API, is able to spawn vehicles and pedestrians, attach sensors to the ego-vehicle and recording data using them, alternate different environments varying by map and weather condition in order to record data as diverse as possible. By using the script, I recorded two datasets, named CARLA-Normal, with no anomalies, and CARLA-Anomaly, with anomalies placed on the road. Then, I used these datasets to train and test Anomaly segmentation models all based on the Mask2Former architecture and mask-segmentation. The tests showed that all models suffer from a large percentage of false-positives, a characteristic that would make them unable to be deployed on autonomous vehicles because of safety risks. Further tests, where CARLA-Anomaly was partitioned based on certain sub-characteristic of the data, showed that the model is particularly suffering in certain scenarios while performing really well in others; further tests should be conducted in order to better understand these behaviors, maybe by constructing specialized datasets that focus on one particular condition at a time. In retrospective, I would say that is certainly possible to construct anomaly datasets by only using simulation, but the main problem lies in the difficulty of finding or custom-making 3-D assets in a good quantity and with varying appearances; as an example, in CARLA-Anomaly there is one kind of object per type, with a fixed texture combination and 3-D model, but in real life the same objects come in different shapes and varieties. However, using simulation has also great benefits: a great amount of diverse data can be recorded cheaply, while the same kind of real data would require more money and possibly regulatory permits. Regarding anomaly models, I believe that datasets built by leveraging the versatility of simulated environments, like CARLA-Anomaly, showed that they are far from perfect and that real datasets on which these models are tested, most of the times, are too small or

unrealistic to have an accurate portrayal of the model capabilities. I hope this thesis work would lead to further investigations on these models, and to the creation of better and more realistic simulated datasets that could be used to benchmark models, that, one day, would be deployed on real, completely autonomous, vehicles.

Bibliography

[1] Ekim Yurtsever et al. "A Survey of Autonomous Driving: Common Practices and Emerging Technologies". In: *CoRR* abs/1906.05113 (2019). arXiv: 1906.05113. URL: http://arxiv.org/abs/1906.05113.

[2] S. Singh. *Critical Reasons for Crashes Investigated in the National Motor Vehicle Crash Causation Survey*. Technical Report. National Highway Traffic Safety Administration, 2015.

[3] SAE International. *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*. Technical Report SAE J3016. SAE International, 2016.

[4] *Tesla Autopilot*. 2025. URL: https://en.wikipedia.org/wiki/Tesla_Autopilot.

[5] Mercedes-Benz Group. *Mercedes-Benz increases top speed of its Level 3 automated driving system to 95 km/h*. Accessed: 2025-03-05. 2024. URL: https://group.mercedes-benz.com/innovations/product-innovation/autonomous-driving/drive-pilot-95-kmh.html.

[6] *Robotaxi*. Accessed: 2025-03-05. 2025. URL: https://en.wikipedia.org/wiki/Robotaxi.

[7] E. Ackerman. "Toyota's Gill Pratt on Self-Driving Cars and the Reality of Full Autonomy". In: *IEEE Spectrum* (2017).

[8] Shervin Minaee et al. "Image Segmentation Using Deep Learning: A Survey". In: *CoRR* abs/2001.05566 (2020). arXiv: 2001.05566. URL: https://arxiv.org/abs/2001.05566.

[9] Marius Cordts et al. "The Cityscapes Dataset for Semantic Urban Scene Understanding". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016).

[10] Bowen Cheng et al. *Masked-attention Mask Transformer for Universal Image Segmentation*. 2022. arXiv: 2112.01527 [cs.CV]. URL: https://arxiv.org/abs/2112.01527.

[11] Carla Development Team. *CARLA: an Open-Source Simulator for Autonomous Driving*. Accessed: 2025-03-06. 2025. URL: https://carla.readthedocs.io/en/latest/.

[12] Hermann Blum et al. "The Fishyscapes Benchmark: Measuring Blind Spots in Semantic Segmentation". In: *arXiv preprint arXiv:1904.03215* (2019).

[13] Robin Chan et al. "SegmentMeIfYouCan: A Benchmark for Anomaly Segmentation". In: *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*. Ed. by J. Vanschoren and S. Yeung. Vol. 1. 2021. URL: https://datasets-benchmarks-proceedings.neurips.cc/paper_files/paper/2021/file/d67d8ab4f4c10bf22aa353e27879133c-Paper-round2.pdf.

[14] Dan Hendrycks et al. "Scaling Out-of-Distribution Detection for Real-World Settings". In: *ICML* (2022).

[15] Matej Grcic, Josip Šarić, and Siniša Šegvić. "On Advantages of Mask-level Recognition for Outlier-aware Segmentation". In: 2023.

[16] Shyam Nandan Rai et al. *Mask2Anomaly: Mask Transformer for Universal Open-set Segmentation*. 2023. arXiv: 2309.04573 [cs.CV]. URL: https://arxiv.org/abs/2309.04573.