BERGISCHE
UNIVERSITÄT
WUPPERTAL

# Improving Neural Networks for Automated Driving using Corner Cases and Sensorfusion

**Approved dissertation**
**to obtain the doctoral degree**
**Doctor of Engineering (Dr.-Ing)**

at the
Faculty of Mechanical and Safety Engineering
at the University of Wuppertal

Submitted by
**Kamil Peter Kowol, M. Sc.**
from Rybnik (Poland)

| | |
|---|---|
| Supervisor: | Univ.-Prof. Dr.-Ing. Stefan Bracke |
| Co-Supervisor: | Univ.-Prof. Dr. rer. nat. Hanno Gottschalk |

| | |
|---|---|
| Dissertation submitted: | March 27, 2023 |
| Date of defense: | September 15, 2023 |

# Acknowledgments

First of all, I would like to thank Hanno Gottschalk and Matthias Rottmann for giving me the opportunity to work in the field of deep learning. Thank you for your support over the years as we have had productive discussions and produced some joint publications. I have enjoyed working with you very much and the knowledge gained through this collaboration will pave the way for my further career in this exciting and sought-after field.

In addition, I would like to thank Prof. Dr. Stefan Bracke for his support and expert guidance throughout my doctoral journey.

I would like to thank my student assistants, Natalie Grabowsky and Ben Hamscher, for all the corner cases they have generated, which were essential for my results concerning the driving simulator.

I would also like to thank all my colleagues and former colleagues in the stochastics group, without whose support this work would not have been possible. Furthermore, I particularly thank Svenja Uhlemeyer for the numerous scientific discussions, for proofreading this work but also for her wholehearted support away from the daily office routine.

Finally, I would like to thank my parents, sister and brother for their constant support over all the years.

# Foreword

The work presented in this thesis is in parts based on the following publications:

- K. Kowol, M. Rottmann, S. Bracke, and H. Gottschalk, *YOdar: Uncertainty-based Sensor Fusion for Vehicle Detection with Camera and Radar Sensors*, in Proceedings of the 13th International Conference on Agents and Artificial Intelligence - Volume 2: ICAART,, INSTICC, SciTePress, 2021, pp. 177–186 ©2021 SCITEPRESS
  - the paper was mostly written by myself and incorporated into Section 4.3 with minor adaptations
  - collaboration with co-authors, receiving their advice and incorporating their input to refine the paper
  - all experiments were implemented and conducted by myself
- K. Kowol., S. Bracke., and H. Gottschalk., *A-Eye: Driving with the Eyes of AI for Corner Case Generation*, in Proceedings of the 6th International Conference on Computer-Human Interaction Research and Applications - CHIRA, INSTICC, SciTePress, 2022, pp. 41–48 ©2022 SCITEPRESS
  - the paper was mostly written by myself and incorporated into Section 5.1 with minor adaptations
  - collaboration with co-authors, receiving their advice and incorporating their input to refine the paper
  - all experiments were implemented and conducted by myself
- K. Rösch, F. Heidecker, J. Truetsch, K. Kowol, C. Schicktanz, M. Bieshaar, B. Sick, and C. Stiller, *Space, time, and interaction: A taxonomy of corner cases in trajectory datasets for automated driving*, in IEEE Symposium on Computational Intelligence in Vehicles and Transportation Systems (IEEE CIVTS), IEEE SSCI, 2022 ©2022 IEEE

  - major contribution to section V-A (*Application on Simulation Data*)
  - major contribution in recording and classifying the synthetic data
  - partial contribution in writing of II (*Definitions*)
  - partial contribution to content and structure to the trajectory corner case taxonomy (*Table I*)
  - general editing of the text before submission

- K. Maag, R. Chan, S. Uhlemeyer, K. Kowol, and H. Gottschalk, *Two video data sets for tracking and retrieval of out of distribution objects*, in Computer Vision – ACCV 2022, L. Wang, J. Gall, T.-J. Chin, I. Sato, and R. Chellappa, eds., Cham, 2023, Springer Nature Switzerland, pp. 476–494
  - major contribution to the CARLA-WildLife dataset as I created it and put it into the form provided
  - partial contribution to the generation of the Wuppertal Obstacle Sequences dataset
  - major contribution in writing of 3.2 (*CARLA-WildLife*), 3.3 (*Wuppertal Obstacle Sequences*) and Appendix-A (CARLA-Wildlife part)
  - I mainly conducted the OoD segmentation and tracking experiments using the codes from [24, 112]
  - general editing of the text before submission

- D. Bogdoll, S. Uhlemeyer, K. Kowol, and J. M. Zöllner, *Perception datasets for anomaly detection in autonomous driving: A survey*, in 2023 IEEE Intelligent Vehicles Symposium (IV), 2023, pp. 1–8
  - major contribution to the description of chapter II-H (*Wuppertal OoD Tracking*)
  - major contribution to the anomaly source definitions in chapter II
  - partial contribution in writing of chapter I (*Introduction*)
  - partial contribution to content and structure to table I
  - general editing of the text before submission

- K. Kowol, S. Bracke, and H. Gottschalk, *survAIval: Survival Analysis with the Eyes of AI*, in Computer-Human Interaction Research and Applications, A. Holzinger, H. P. da Silva, J. Vanderdonckt, and L. Constantine, eds., Cham, 2023, Springer Nature Switzerland, pp. 153–170

– the paper was mostly written by myself and incorporated into Section 5.2 with adaptations

– collaboration with co-authors, receiving their advice and incorporating their input to refine the paper

– all experiments were implemented and conducted by myself

# Contents

Chapter 1

# Introduction

According to projections by the United Nations Department of Economic and Social Affairs, two-thirds of all people will live in a city by 2050 [167]. Higher traffic volumes in major cities are the result, leading to an increase in the probability of accidents for all road users [97] and thus inevitably to an increased need for safer components in and on the vehicle. To address this, research into better detection and decision-making methods using artificial intelligence (AI) is necessary. It can be considered a core technology specifically for autonomous driving.

The continuous enhancement of neural networks and the increasing amount of training data improved the recognition accuracy of real street scenes [22, 35, 151] which led to first tests with autonomous driving vehicles. This repeatedly resulted in fatal accidents [118, 119, 120]. A series of unfortunate circumstances led to a misinterpretation of the situation in each case and shows that the evaluation of rare situations can still be error-prone for AI algorithms. To counteract this, adding safety-critical driving situations, so-called corner cases, to the training data is essential to increase the performance of AI algorithms. These are currently only available to a limited extent, if at all, in publicly provided datasets. The targeted generation of corner cases is also challenging, as they do not constantly occur in road traffic, and if they do, they have to be recorded and labeled. This means, that to obtain a large amount of data with safety-critical driving situations, a high number of drives would also be required, which would consume costs and time. A much bigger problem, however, is that the targeted generation of corner cases can also be risky, as one is specifically looking for safety-critical situations and is forced to put oneself in immediate danger. Moreover, compliance with traffic regulations would only be possible in a secured area, which in turn can only reflect the real traffic situation to a limited extent, as other road users must also be included in order to maintain realism.

In contrast to real data generation, synthetic data generation does not incur additional costs for labeling or procuring expensive sensors, as these are immediately available in the virtual world. Only a computer with suitable software for autonomous driving is needed. Furthermore, synthetic data is easily changeable, allowing a high number of different data to be generated in a very short time. Changeable parameters are e.g. the environment, the number of objects, the weather or the time of day. The selective disregard of traffic rules of individual road users can also be implemented. The disadvantages of synthetic data are the unreal-looking world but also the unnatural movement patterns of road users, so that transferability to the real world (domain shift) is problematic.

It follows that generating corner cases in the real world is expensive and not easily feasible for security reasons. The goal of this work is therefore to generate synthetic corner cases that are problematic for AI algorithms as quickly as possible. This is done using simulation software, a specially designed test rig and two human drivers who are supposed to evaluate critical driving situations with human perception.

In addition to the targeted data acquisition of safety-critical corner cases, the redundant evaluation of a scene by means of a second sensor can also increase safety in road traffic. Therefore, another goal is to investigate whether the prediction of a fusion network of two sensors in a less frequent domain, namely night, can find more objects than the prediction of a sensor-specific network considered alone. For this, camera and radar data will be used from the real world, since synthesizing a physically accurate radar sensor is difficult.

This thesis is organized as follows. Chapter 2 provides an overview of the fundamentals in the field of artificial intelligence that are relevant to the remainder of this thesis. For this purpose, the concept of neural networks, with main reference to the textbooks [62, 117] will be discussed. In addition, a description is given of how autonomous driving can be optimized through the use of semantic segmentation networks. Chapter 3 presents the driving simulator that was developed in the scope of this thesis. First, the software used and the specially developed inference sensor are explained. Then the hardware components are presented that are necessary in interaction with the simulation software and the inference sensor in order to achieve the task of generating targeted corner cases. Since driving in real-time is essential for the accomplishment of the task, a detailed description of how this task was realized follows. In Chapter 4, the levels of autonomous driving are presented before the term corner case is described in more detail. Following this definition, three publications are presented that contribute scientifically in this field. One publication is not to record safety-critical data in order to increase reliability, but on a methodical basis, where a redundant evaluation of the scene is carried out by means of two sensors, using different technologies for data recording. The second publication provides new datasets containing Out-of-Distribution

objects (OoD), as well as a method for detecting unknown objects and tracking them in video sequences, in order to subsequently package similar unknown objects together as a new object class. Whereas the third publication is a listing of datasets containing OoD objects in the context of autonomous driving and is intended to serve as an overview for researchers in this field.

Chapter 5 presents the scientific work with the driving simulator. First, the experiments and the recording pipeline are described, followed by an investigation of whether and to what extent adding corner cases to the training leads to a longer survival time. Furthermore, survival analysis is used to investigate whether universal models can be replaced by expert models that are only trained on certain domains in order to save development time for the application. Next, a corner case trajectory taxonomy is presented where the corner cases created during the driving campaigns are classified. Finally, a conclusion and outlook are provided in Chapter 6.

Chapter 2

# State of the Art and Theoretical Foundation

This chapter provides an overview of the fundamentals in the field of deep learning (DL) that are relevant to the remainder of this thesis. The fundamentals from this chapter come mainly from two textbooks [62, 117] unless otherwise noted. First, the concept of *feedforward neural networks* will be explained followed by the *learning process*. Next, *convolution neural networks (CNNs)* are introduced as a special case of neural networks before *image segmentation* is described as a concept for recognizing objects on pixel level for autonomous driving. Finally, the *model development process* is described to illustrate the steps required to use neural networks in practice.

## 2.1 Feedforward Neural Networks

As for humans, neurons play an important role in pattern recognition in computers. Biological neural networks consist of multiple interconnected neurons, also called nerve cells, of which humans have between 10 and 100 billion in the brain. Following the biological model, artificial neural networks in computer science also include neurons - also known as perceptrons - that contain connections with numerical values, the weights $w$.

The perceptron denotes the smallest unit of a neural network and is itself a single-layer neural network for the task of binary classification. Figure 2.1 shows the schematic structure of a single perceptron. Each perceptron consists of several weighted inputs $w_i \cdot x_i$, a bias value $b$, a propagation function $\sigma$ and an activation function $\phi$ which produces an output value $a$. The propagation function returns

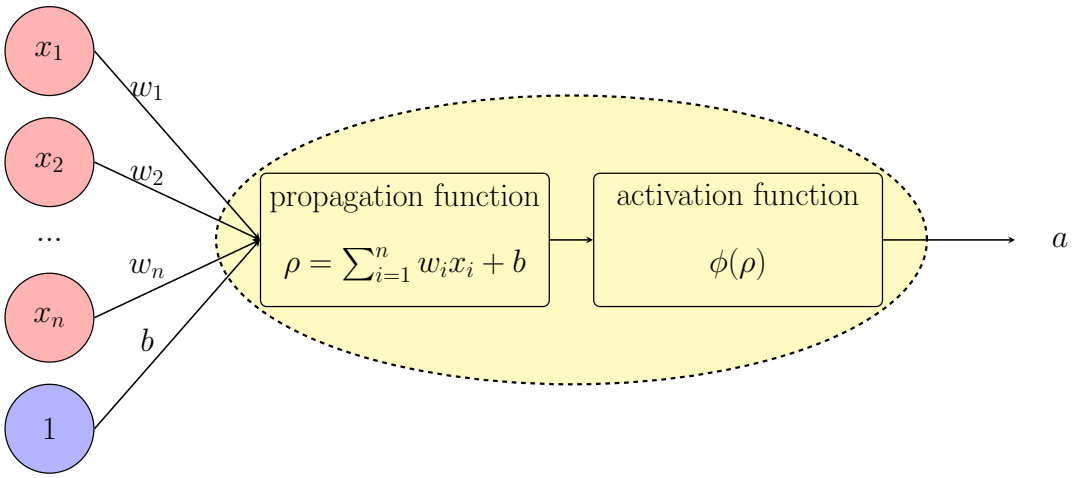



Figure 2.1: Perceptron.

the weighted sum of all inputs, while the activation function propagates a calculated value or not, which depends on the bias value as it shifts the threshold to fire on the x-axis. Together the functions are defined as

$$a = \phi(\rho) = \phi\Big(\sum_{i=1}^{n} w_i x_i + b\Big) \,. \tag{2.1}$$

Connecting multiple perceptrons so that their output provides the input to a downstream perceptron creates a neural network with multiple layers, as shown in Figure 2.2. In case they do not contain any loops or connections at the same layer, such networks are referred to as feedforward networks or multilayer perceptrons (MLP). If there are feedback connections the network is called recurrent neural network (RNN) [148]. In MLPs each edge in the graph connects the output of one perceptron to the inputs of all perceptrons in the following layer.

Typically, those neural networks are organized in three different layer types: input, hidden and output layer [154]. The input layer contains visible information about input data represented by numerical values in the form of an input vector. The number of neurons corresponds to its dimensionality. The input layer is followed by one or more hidden layers extracting complex information using function approximations. The term *hidden* refers to the fact that the information contained in these layers emerges indirectly from the input data and are not observed during training. In computer vision tasks, for instance, this is where contours or edges are detected. The output layer is the network's last layer and provides a final result. In classification tasks, for example, a probability value is calculated for each class, or in regression tasks a certain value is predicted. The

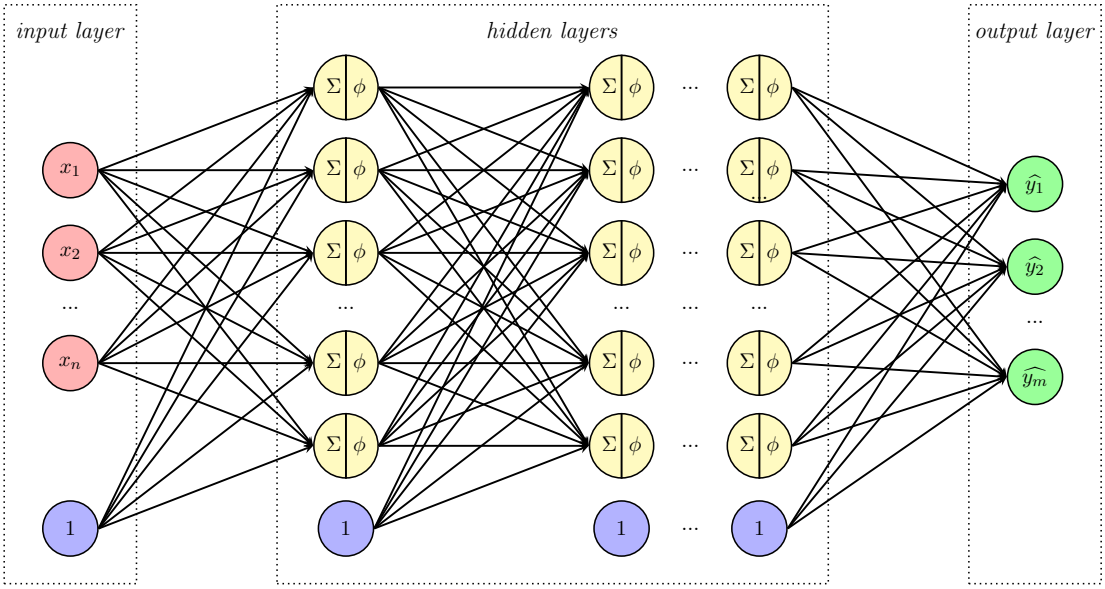



Figure 2.2: A feedforward neural network processes information through multiple hidden nodes in one direction only. Whenever a neural network possesses more than one hidden layer, it is termed deep neural network.

output of a feedforward neural network is determined by a chain of functions

$$\widehat{y} = f(x;\theta) = f^L(f^{L-1}(\ldots(f^1(x;\theta^1);\ldots);\theta^{L-1});\theta^L) \tag{2.2}$$

based on the order of layers, with parameters $\theta = (\theta^1, \theta^2, \ldots, \theta^L)$ and $\theta^l = (w^l, b^l)$, $l = 1, 2, \ldots, L$, where $w^l$ denotes the weight matrix and $b^l$ the bias term of a single layer $l$.

MLPs can model any logical function with enough hidden units, but this also results in many parameters and thus increased computation time. [38, 179]

## 2.2 Learning Process

First, a general overview of the training process in supervised learning is given with the help of Figure 2.3, before all individual parts are explained in more detail.

Solving a task with machine learning requires data from which patterns can be learned. In computer vision, this data consists of images whose quality, size, and variety significantly influence the performance of the task to be tackled. During the learning process, significant properties and features of the dataset

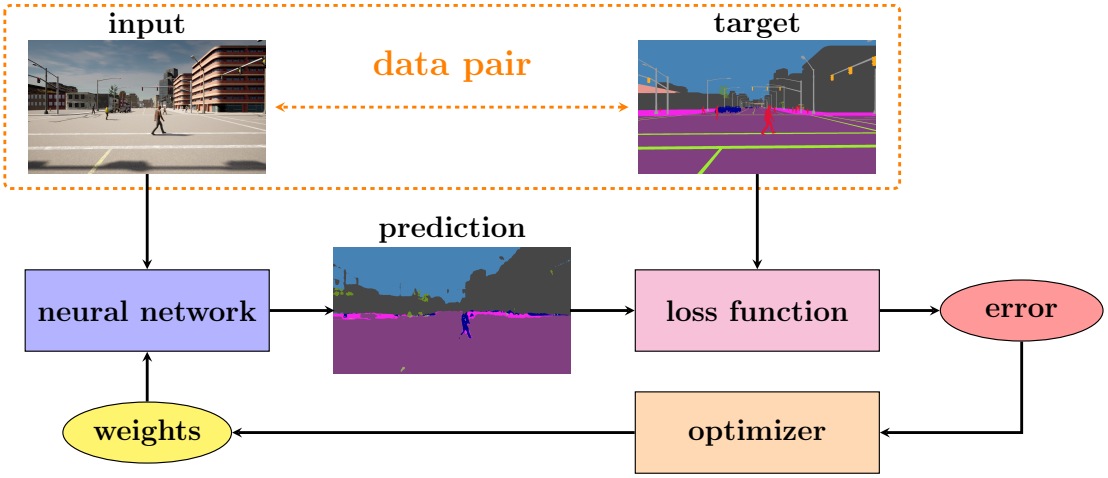



Figure 2.3: Learning process in supervised learning. An input image is processed through a neural network, including weights, to make a prediction. Subsequently, a loss function compares the prediction with the target, generating an error term. This error is then used by an optimization algorithm to adjust the weights to minimize the error in following iterations.

$\mathcal{S} = (s_1, s_2 \ldots, s_N)$ with $N$ being the dataset's full length, are to be observed from an unknown distribution $\mathcal{D}$ to obtain a predictor $h_s : \mathcal{X} \to (0,1)^Y$, with $\mathcal{Y} = \{1, \ldots, Y\}$, which fits the dataset. $\mathcal{X}$ is a set of objects which are represented as vectors and $\mathcal{Y}$ is a set of possible labels. For this, data pairs of $s_n = (x_n, y_n) \quad \forall n = 1, 2, \ldots, N$ are used to supervise the process, where $x \in \mathcal{X}$ is the input vector and $y \in \mathcal{Y}$ the target vector. [154]

So, input data is transformed into an input vector and forwarded into a neural network. Afterwards the network's output $\widehat{y}$ is compared to the target $y$ using a loss function which delivers an error. Afterwards an optimizer algorithm like AdaGrad [47], Adam [86], etc. back propagates the error through the network while adjusting the weights. The optimizer uses the gradient descent method for weight adjustment. The training data is divided into small packages, called batches, and passed through the network. After each batch, the weights are adjusted. An entire pass of all images occurring in training data is called an epoch and can take different amounts of time depending on the software (network, dataset, image size, etc.), but also hardware (number of available graphics cards and their memory size, hard disk speed, etc.). After the first epoch, performance measurement begins with a validation dataset designed to test performance on previously unseen images and reflect the generalization capabilities of the training success so far.

**Data** If enough data is available, a recommended approach is the hold-out method where the available dataset $\mathcal{S}$ is divided into three different parts - train, validation and test dataset - to promote generalization. A good thumb of rule is a split of 2:1:1, which is used for small datasets. The more data available, the higher the proportion of training data can be. [3] The training set is the biggest part which is needed to learn complex features to build the model. It serves as the knowledge foundation of a model and should be large, clean and diverse[1]. Any lack of knowledge in training data could, for example, lead to accidents in driving situations with autonomous driving vehicles when testing in the real world [118, 119, 120]. The main task of the validation set is model selection and parameter tuning [3]. It is used for performance computation during training and is thereby responsible for parameter adjustments, such as the learning rate. After selecting the best model, performance verification on unseen but similar data is done on a test set that serves as an estimator for the true error [154].

If insufficient data is available, the validation set would be small and result in a noisy estimate of predictive performance [11]. In such cases the k-fold cross validation (KFCV) or leave-one-out cross validation (LOOCV) method can be used to obtain accurate estimates of the true error during training.

The KFCV method divides the training set into $K_s$ equal subsets were one subset $k_s \in \{1, \ldots, K_s\}$ is used for testing and error estimation and to remain for training. This process is repeated $K_s$ times, each time using another subset for testing. The average performance of all test subsets is then stated. LOOCV, on the other hand, is a special case of KFCV that omits only one sample. Especially the LOOCV method is computationally expensive and should be only used for very small datasets. [3, 154]

**Activation Function** An important element in neural networks is the activation function $\phi$, which is responsible for efficient learning and thus for the quality of the predictions. They add a nonlinear property to the neural network allowing it to approximate more complex functions like the XOR-example, where the two classes $\{(0,0),(1,1)\}$ and $\{(0,1),(1,0)\}$ need to be separable. Activation functions are also differentiable, so gradients can be calculated through them which is needed for the backpropagation algorithm. The activation function plays two important roles in deep neural networks. First, within hidden layers they determine the range of values at which a perceptron is activated, which is also referred to as "firing". Second, as part of the output layer, they provide a probability value for each class. The softmax activation function is often used in this case, where a probability value is assigned to each class and the aggregation of all probabilities

[1] A. Karpathy, *CVPR 2021 Workshop on Autonomous Driving, Keynote.*, 2021

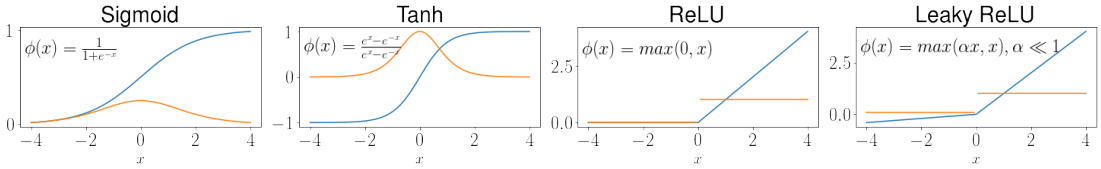



Figure 2.4: Different activation functions (blue) with its derivative (orange).

equals 1. The softmax activation function is often used in this context. Figure 2.4 illustrates frequently used activation functions with its derivatives.

**Loss Function** The goal of supervised learning is to reliably predict each label $y$ of the label set $\mathcal{Y}$ for any input $x$ of an input set $\mathcal{X}$. During training, the so-called loss or cost function $\ell$ compares the prediction of the model with its target and calculates a distance value, also referred to as loss. The loss is used as a feedback signal to adjust the networks weights in the direction of the negative gradient for minimization purposes. This is called the gradient descent algorithm.

A way to calculate the error is to count the number of misclassifications using the indicator function $\mathbb{I}$ as

$$\mathbb{I}(y, \widehat{y}) = \begin{cases} 1, & y = \text{argmax}(\widehat{y}) \\ 0, & y \neq \text{argmax}(\widehat{y}) \end{cases} , \tag{2.3}$$

which equals 1 if the prediction is correct or 0 when incorrect. When using $\mathbb{I}$, all errors are considered equal, which is not practical in reality since some errors are more costly than others. Therefore, asymmetric loss functions can be used, which penalize certain errors more [117]. Another problem using this function is that it is not differentiable and thus unsuitable for the backpropagation algorithm.

**Empirical Risk Minimization** The goal of the learning algorithm is to find the predictor $h_{\mathcal{D}} : \mathcal{X} \to (0,1)^Y$ that minimizes the error in relation to the unknown dataset $\mathcal{D}$ and the target function $f : \mathcal{X} \to (0,1)^Y$. The target function delivers the correct answer to the observed data, and we assume that the predictor does not predict the correct answer, from which follows that $h \neq f$. Since $\mathcal{D}$ is unknown, the true error cannot be computed straightforwardly. Therefore, an error $\mathcal{L}_{\mathcal{S}}(h)$, also known as the empirical risk or error, can be calculated on a subset $\mathcal{S} \in \{\mathcal{S}_{train}, \mathcal{S}_{val}, \mathcal{S}_{test}\}$ as follows:

$$\mathcal{L}_{\mathcal{S}}(h) = \frac{1}{N} \sum_{n=1}^{N} \ell(y_n, \widehat{y_n}) \, . \tag{2.4}$$

Equation (2.4) provides an averaged error, where $N$ is the length of all data pairs and $\ell$ the loss function, which gets a vector of class probabilities $\widehat{y}$ in addition to the correct class $y$. Since the goal during training is error minimization, the parameter search to minimize the empirical error on the training data is called *Empirical Risk Minimization (ERM)*. [117, 154]

A typical approach for minimization is the stochastic gradient descent (SGD) algorithm. It is an iterative method for optimizing a differentiable function to find the global minimum. Its derivation provides information about the direction to reach the minimum with respect to the model parameters which is done by small shifts in the direction of the negative gradient. During training, the data is divided into small sub-packages, called batches, which are randomly selected. Based on the error term of each batch, the model parameters $\theta = \{w, b\}$ are updated to reduce the error in the next iteration step $i_s$ by

$$\theta(i_s + 1) = \theta(i_s) - \eta \frac{\partial \mathcal{L}}{\partial \theta} \,, \tag{2.5}$$

with $\eta$ being the step size and $\ell$ the loss function.

The algorithm used to compute the gradient of a loss function applied to the networks output with respect to the weights in each layer is known as back-propagation [148] and can be subsequently used by a gradient-based optimization algorithm based on SGD for learning [62, 117]. For this purpose, the gradient of the error needs to be propagated back to calculate the next error. Figure 2.5 serves as an illustration to help to understand this process better.

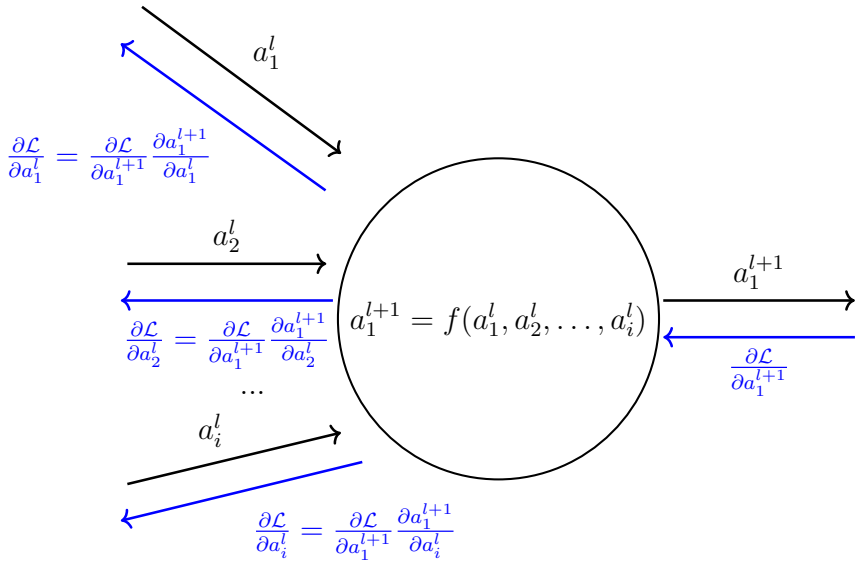



Figure 2.5: Gradient flow calculus.

Using a learning rate $\eta$ allows to scale the step size. If the step size is too large, the minimum could be skipped and thus never reached. A step size that is too small, on the other hand, could result in a global minimum never being reached because of being trapped in a local minimum. Therefore, a dynamic step size has been established in practice, which is large enough at the beginning to skip local minima, but decreases with time to ideally reach the global minimum with small steps.

Since this simple optimization algorithm is capable of computing gradients quickly and is therefore well suited for large datasets, both SGD and other optimizers based on it, such as AdaGrad [47] or Adam [86], find application in deep learning.

**Regularization** The ERM algorithm runs the risk of overfitting. This refers to a predictor that is too strongly oriented to the training data (analogy to memorization) and thus performs poorly on similar but unknown data. In contrast, the term underfitting refers to the case where training was too short and complex correlations were not learned. While training, the errors of training data and validation data are tracked with the number of iterations. As soon as both curves behave in opposite directions, this represents the perfect region where the model neither under- nor overfits and generalizes the best.

There are some regularization techniques that counteract or prevent overfitting. These include early stopping, weight decay, image augmentation, feature selection, L1 or L2 regularization or drop-out, in addition to using larger amounts of data.

- *Early stopping* [116]
  Overfitting can be prevented by stopping the training earlier than planned. To do this, the loss values for the training and validation datasets must be monitored during the training. Since the validation error as opposed to the training error worsens after a certain point, the training should be terminated early. A termination condition $E_{ES}$ is defined, which specifies how many epochs in a row the best validation loss must not drop below until the training is terminated.

- *Weight decay* [69]
  Weight decay, also called $L_2$-regularisation, is a technique for minimizing a loss function by adding the $L_2$ norm as a penalty term to the loss function, preventing the model from becoming too complex. In this case, the loss function changes as follows:

$$\mathcal{L}_{wd} = \mathcal{L}_{\mathcal{S}} + \lambda_{pen}\Omega(w) \tag{2.6}$$

$$= \mathcal{L}_{\mathcal{S}} + \lambda_{pen}\frac{1}{2}\left\|w\right\|_2^2 \ . \tag{2.7}$$

With $\Omega(w)$ describing the penalty parameter and $\lambda_{pen} \in \mathbb{R}^+$ the regulation parameter.

- *Image augmentation* [62]
  This is a process of generating new data by manipulating copies of the existing training images. This procedure improves generalization and is especially useful for small datasets as it enables an increased number of training images. Manipulation techniques are rotation, translation, scaling, cropping, hue and contrast, to name a few.

## 2.3 Convolutional Neural Networks

Convolutional neural networks (CNN) are specialized neural networks for processing data that have a well-known, grid-like topology like time-series or image data [95]. CNNs divide the input into overlapping 2D arrays, also referred to as *receptive fields* which represent parts of an object, and comparing each array to a series of small weight matrices, better known as *kernels* [117]. Instead of general matrix multiplications, the convolution operator $*$, which is the main core of this network, takes place in at least one layer.

Early CNN design patterns consisted of alternating convolutional layers with ReLU activation and max-pooling layers followed by one or more fully-connected layers at the end for classification purposes, such as Yann LeCun's LeNet [96]. As CNNs have grown larger and more complex over time, the convention of combining several repetitive layers into one block is often found. In the previous example, the sequence of convolution, ReLU activation and pooling would be combined into a convolution block.

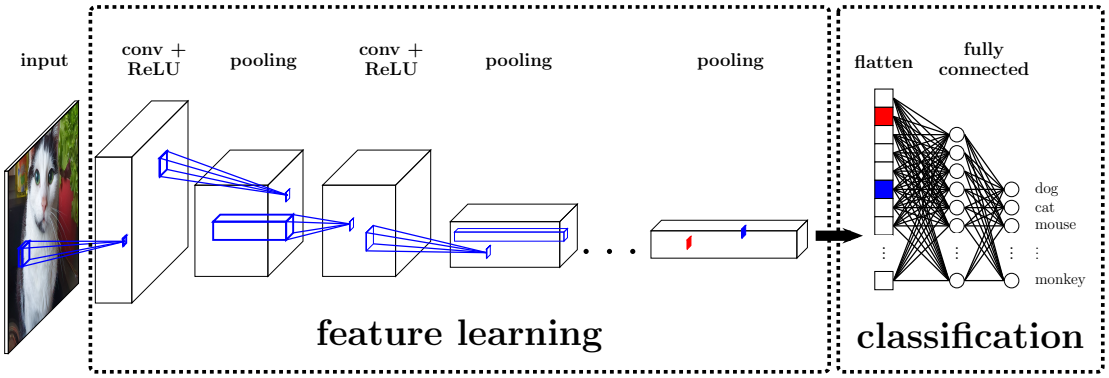



Figure 2.6: A simple CNN for image classification [117]. The input image passes through several blocks where features are detected and then a classification is provided as output.

Even though CNNs are also divided into three layers, the hidden layers can be further divided into two parts. First, there is the feature learning area, which consists mainly of convolutional blocks, and the classification area, which consists of the flattening, fully connected, and softmax layer (see Figure 2.6).

Compared to conventional neural networks, CNNs have several advantages, such as a lower memory requirement due to sparse connections between layers or weight sharing and thus fewer model parameters, faster processing of large amounts of data, additional storage of spatial information and robustness to altered lighting conditions, which is why they are increasingly used in image recognition since AlexNet [94] won the ImageNet challenge in 2015 [149].

**Convolution** In general, a convolution is a mathematical operation between two functions $f$ and $g$ that results in a third function $s$. Therefore, every value of $f$ is replaced by the weighted average of the surrounding values. One use case would be signal or image processing, where convolution can filter out background noise to some extent.

When applying convolution a fixed matrix, also known as a *kernel* $K$, is moved over the complete image $I$ and the convolution operator is applied to each considered subarea, also known as *receptive field*. The kernel constantly moves by a fixed pixel width $s_w$, referred to as *stride*. A convolution with $s_w > 1$ is also termed *strided convolution*, which reduces redundancies and speeds up computation time. The disadvantage is a drop in performance due to loss of information. All numbers in the receptive field are multiplied by the corresponding value from the kernel in order to sum up all factors according to

$$M(i,j) = (I \circledast K)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n) \ . \tag{2.8}$$

This equation is a special form of the convolution formula that requires no flipping of the kernel and is known as cross-correlation. The result is then offset with a bias value to create an entry in a new matrix called *feature map* $M$. A schematic representation of a convolution, including the ReLU activation function, is shown in Figure 2.7. This figure illustrates that convolution results in a dimensional reduction, which causes information to be lost at the image borders. Especially with small images this can lead to problems, so that a technique called *zero padding* can be applied. It appends artificial zeros to the borders of the image or feature map in order to preserve the dimensions.

The convolution operator, that contributes to maintain most of the spatial information, gives CNNs two important properties that are not present in classical dense layer neural networks. First, they are translationally invariant, so patterns can be detected anywhere in the image, and second, they learn spatial hierarchies

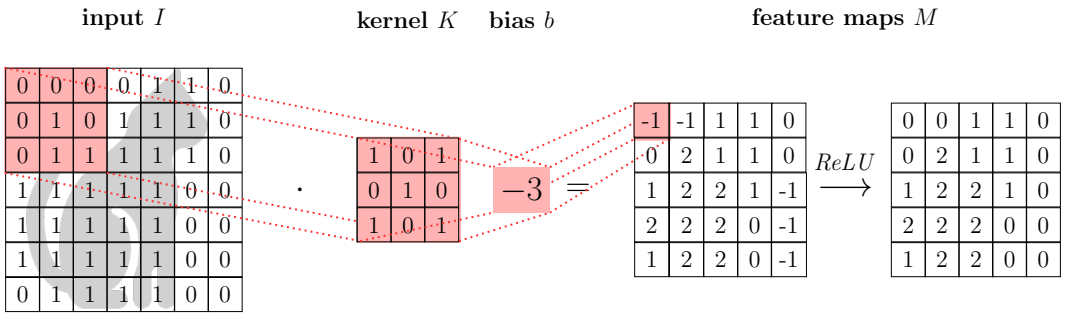



Figure 2.7: Schematic representation of a convolution including ReLU applied to an input image returning a so-called feature map. The convolution is first calculated on the first image segment (dimension $3 \times 3$) of the input with a kernel of the same dimension, subsequently the bias term is added to finally apply the activation function in order to save it as a numerical value in a feature map. Following this procedure, the kernel slides over the entire image and fills the feature map piece by piece.

of patterns. Only small local patterns are detected in the first layers, from which more complex patterns are detected in deeper layers that are still connected to the previous ones. [31]

**Pooling** With pooling, the feature maps created by convolution are reduced in size by combining neighboring pixels which reduces the dimensions of the feature maps. The most common form of pooling is max pooling and is shown in Figure 2.8. Initially, the pooling process is similar to convolution, where the input feature map is first divided into overlapping 2D image arrays. Then, only the maximum value of each array is written to a new feature map, which reduces the size of the subsequent feature map. Unlike convolution, pooling provides no learnable weights $w$.

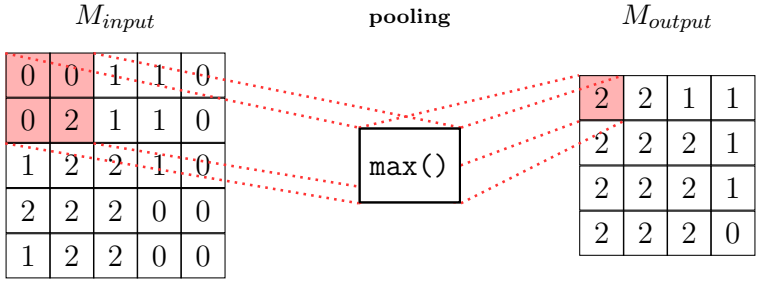



Figure 2.8: Schematic representation of max pooling. The input is first divided into overlapping 2D arrays (here dimension $2 \times 2$) in order to compute the maximum value of each array afterwards. We use a stride value of 1 to subdivide the arrays in this example.

**Fully-Connected** Conventional neural networks consist mainly of fully-connected layers. Their characteristic is that each neuron interacts with each neuron in the next layer. Even though this type of interconnections can be used to resolve tasks, redundant computations commonly occur, resulting in increased computation time. Therefore, they are more commonly used in CNNs in the classification area, where classification is subsequently performed using the probability output of, for example, a softmax layer.

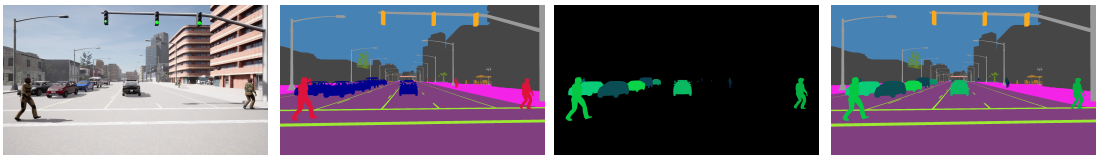
**Other building blocks** Over time, additional blocks for CNNs found application in scientific and commercial works, which could achieve more efficient and better performance for solving image classification tasks. Some of them are briefly explained from here, while detailed information can be found in the sources provided.

- *Deconvolution and transposed convolution* [117, 181]
  While a deconvolution only inverts the standard convolution with a known filter, the transposed convolution upsamples the feature map using learnable parameters.

- *Dropout* [62, 158]
  An algorithm for randomly removing neurons to prevent overfitting. During this process, connections that have already been learned are deliberately deleted so that other neurons contribute to the solution of a problem. Even if better performance can be achieved with dropout, one disadvantage to be mentioned is the longer computing time.

- *Batch normalization* [79, 117]
  Batch normalization is a normalization layer that increases the stability of a deep neural network. It ensures that the activations within a layer have a mean of zero and a unit variance when averaged over the samples in a batch.

- *Pyramid pooling* [182]
  A pyramid pooling module (PPM) was introduced in 2017 which fuses features under different pyramid scales to accumulate contextual information based on different regions. For this purpose pyramid pooling is done after a convolution layer, where different pooling steps are performed in parallel, generating feature maps of different sizes in different branches. This is followed by a convolution step in each branch with a subsequent upsampling step so that all feature maps can be concatenated with the original feature map.



original semantic instance panoptic

Figure 2.9: Examples of image segmentation tasks.

## 2.4 Image Segmentation

In computer vision, a common task is image segmentation, where the image is divided at the pixel level into coherent parts or segments that conceptually belong together [55, 161] and create a new image, also known as a mask. An image usually consists of three channels (red, green, blue) with values stored with an accuracy of 8 bits. Coloring contiguous segments creates a clustering of the entire image, where closed contours of each segment and the color allow for conclusions about the type of segment. Image segmentation on segment level exists in three different levels of detail: semantic segmentation, instance segmentation and panoptic segmentation. In addition, the content of an image is usually divided into the two categories *things*, which are countable objects such as humans or vehicles, and *stuff*, which are formless regions with similar texture or material like sky or road [2, 70, 87].

**Semantic Segmentation** An important application for autonomous driving is semantic segmentation, where each pixel of an image is assigned to a specific class. In this process, the original images are reduced in advance to the most important features and contiguous areas are assigned to the same class. For human visualization, each class is assigned a color in RGB space, resulting in color blob-like images which is shown in Figure 2.9.

The image shows some vehicles and pedestrians on the road. When learning this scene, it is more important to learn that there are vehicles and pedestrians than to classify each specific feature of the vehicle, such as the vehicle class, window panes or of the pedestrians like the cloth color. Even though the different features can help distinguish a vehicle from humans, feature reduction is essential for solving more complex tasks such as autonomous driving.

Semantic segmentation networks build upon the CNN architecture, with the difference that they do not classify the entire image, but each individual pixel. Therefore, they usually have nearly symmetric encoder-decoder structure, where each downsampling step in the encoder part is compensated by an upsampling step in the decoder part. This ensures that the original size of the input image is

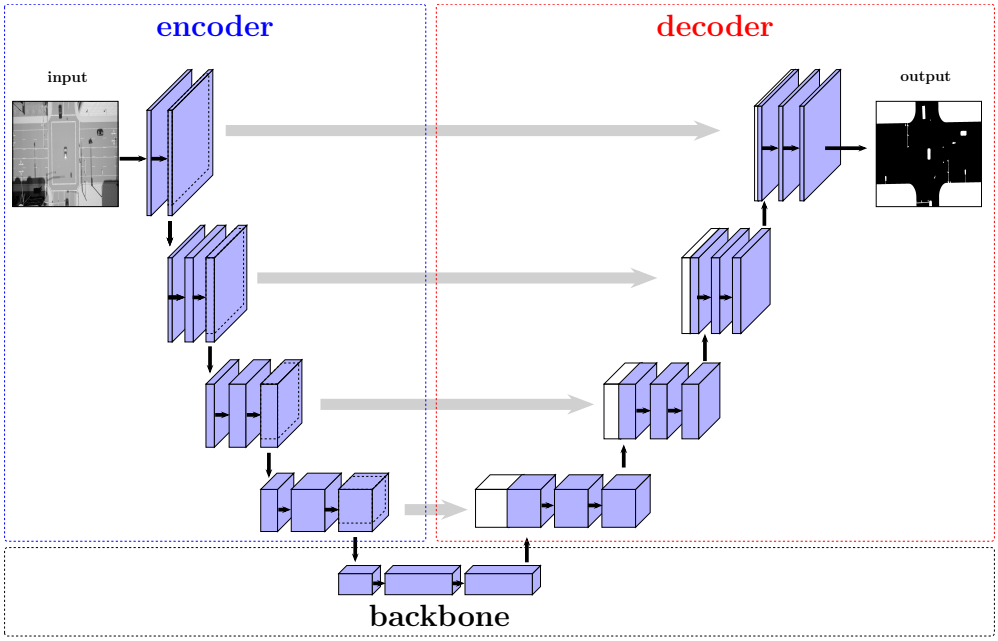



Figure 2.10: U-Net architecture according to [145]. An input image passes through an encoder-decoder structure to produce a pixel-by-pixel binary prediction. In this example, the task is to segment the road.

almost preserved. The encoder part extracts features and downsamples the input image, while the decoder part converts the encoder's low-resolution representations back into the original format. Downsampling - done with convolution blocks including stride and pooling - captures high level properties while losing spatial information whereas upsampling - done with transposed convolution blocks - partially restores this spatial information.

An evolution of this structure is U-Net, introduced in 2015, which uses skip connections to better reconstruct spatial information during upsampling [145]. Each - except the last - convolutional block (3 convolutional layers with ReLU) is followed by a copy of the feature map, resulting in two branches. The first branch is pooled using a $2 \times 2$ kernel and the second is cropped and concatenated in the upsampling process, which leads to more precise information reconstruction, see Figure 2.10. Since the U-Net does not use padding layers, the output layer is a scaled down version of the input image.

Last but not least, there is the output layer for pixel classification. However, semantic segmentation does not focus on differentiation between multiple instances, so overlapping instances of the same class will result in a merge. If distinction between instances is desired, instance or panoptic segmentation should be used.

**Instance Segmentation** The overall goal of instance segmentation is to distinguish between instances of the same semantic class. In addition, dynamic *things*, i.e., objects of high interest in road traffic like pedestrians or vehicles are considered, so that *stuff* is not taken into account. With instance segmentation, the task of object detection takes place at the pixel level. R-CNN [59] can be counted as a pioneer in this field, where the network first extracts many bottom-up proposals before localization and segmentation. Based on this, further networks were developed that aim to identify *things* such as MNC [39], Mask R-CNN [71] or FCIS [102].

**Panoptic Segmentation** Panoptic segmentation combines the tasks of semantic segmentation with instance segmentation. This image segmentation method identifies both *things* by class labels and additionally distinguishes between different instances, and *stuff*, marking related areas with similar structures. [87] This is typically done by using two branches of segmentation models, semantic and instance, resulting in a merge. Some well known panoptic networks are OANet [106], UPSNet [176], AUNet [101] as well as the real-time network PanoNet [29].

## 2.5 Object Detection

Object detection is a computer vision task with the aim to detect instances of objects of specific classes. Therefore, a bounding box is drawn around a predicted object, see Figure 2.11. A common object detection algorithm is YOLO *(You Only Look Once)*, which uses $1 \times 1$ convolutions, thereby obtaining the size of the input feature map. There are several versions, with YOLOv3 [142] performing well on any given input resolution. Furthermore, due to its multiscale architecture it finds objects of different sizes in images. It is also popular for embedded systems due to its real-time capability and provides good accuracy [114].


Figure 2.11: Bounding box examples from CARLA.

## 2.6 Evaluation Metrics

Several evaluation metrics for neural networks are available. This subsection focuses on the main metrics used in this work.

**Intersection over Union** The Intersection over Union (IoU), also known as the Jaccard index [81, 82], represents a similarity measure for quantities in the range from 0 (no intersection) to 1 (perfect match). Defined as

$$IoU(A, B) = \frac{|A \cap B|}{|A \cup B|}, \qquad A, B \subseteq \mathbb{R}^n \ , \tag{2.9}$$

this metric is calculated from the overlap of two quantities divided by the merged quantity.

In computer vision the IoU is used to describe the overlap of the ground truth and the predicted object. It can be used for bounding boxes, but also for semantic segmentation, see Figure 4.10. In addition to the IoU, there is also the mIoU, which takes into account the mean IoU value of all classes considered.


Figure 2.12: By means of the IoU for semantic segmentation, ground truth and prediction are compared with each other in such a way that the intersection of both is divided by the union.

**Accuracy** The pixel accuracy describes the number of correctly predicted pixels divided by the total number of pixels

$$\text{pixel accuracy} = \frac{\#\text{ correct pixel classifications}}{\#\text{ all pixel classifications}} . \tag{2.10}$$

The accuracy in object detection can be expressed in finding a class or not. For this purpose, the confusion matrix, which visualizes the performance of a model, can be used to compare the prediction with the ground truth. For this we use a condition to mark an object as found, e.g. IoU $\geq 0.5$. In case we found the ground truth object we call this a *true positive (TP)*, otherwise a *false negative (FN)*. Furthermore, if the model predicts an object but no ground truth data is available to this object, this is referred to *false positive (FP)*. If there is no object and we do not recognize something there, this is called a *true negative (TN)*. Then we can compute the accuracy as

$$\text{accuracy} = \frac{TP + TN}{TP + TN + FP + FN} . \tag{2.11}$$

**Mean Average Precision** The mean average precision (mAP) is a popular metric used to measure the performance of object detection models. It describes the reliability of a network's prediction for an object. The IoU value serves as a threshold to determine whether an object is found or not. Typically, a threshold of at least IoU $\geq 0.5$ is used for this purpose.

## 2.7 Model Development Process

In order to use a neural network in the real world, several process steps are required that are iteratively repeated to solve an underlying problem. The model development process, according to [159] can be divided into 5 steps starting with *Data Acquisition*. The focus here is on obtaining data. These represent a subset of the real world that is needed to solve a specific problem. Consideration should be given at the *Preprocessing* step to ensure that data are standardized and accurate, i.e., that images and labels match, and that they are anonymous so that no person, license plates or other metadata can be used to derive personal conclusions. In addition, the data should be made available in such a way that neural networks can work with them. This usually involves the development of a suitable data loader that also separates the data into training, validation and test data. Next *Modeling* starts, where the right algorithm needs to be chosen. For this purpose, self-built, as well as publicly available models can be used. Depending on the underlying model, hyperparameters can be adjusted to achieve

the best performance in combination with the training and validation data used. Often this requires a targeted parameter tuning, as the best parameters depend on the architecture and data. *Evaluation* starts after the final model was chosen. It will be tested on unseen data (test dataset) to measure the real performance of the model.

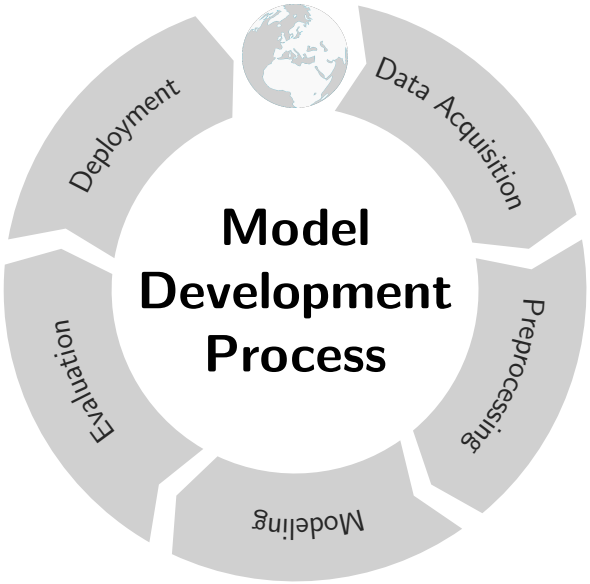



Figure 2.13: The model development process[2].

In addition to the test dataset, the model can also be tested on other available datasets, called benchmark datasets, to demonstrate its robustness and generalization abilities and compare it to other models using performance metrics such as accuracy, IoU (Intersection over Union), mAP (mean Average Precision) and others. In the *Deployment* step, the model is tested in the real world. This involves integrating it into a closed system so that sensors and preprocessing methods are used to provide input data to the model, which is then processed in real time. Subsequent algorithms can be used to handle the output signal, for example, to control actuators or to forward images to a screen. After the model has been field-tested and tried out in the real world, an optimization step can be started by restarting the entire process. This includes, among other things, targeted data generation in order to eliminate the weaknesses that have been occurred.

[2]Created with code from: `https://texample.net/tikz/examples/pdca-cycle/`, accessed: 2023-03-25

Chapter 3

# Driving Simulator

This chapter describes the steps taken to create a custom driving simulator that allows control of a vehicle in a virtual world using only the output of a segmentation network. For this purpose, the simulator setup is described before the software used, called CARLA [44], is introduced. This is followed by a description of the inference sensor implemented in CARLA before presenting some optimization steps necessary to achieve a smoother driving experience.

## 3.1 Introduction

If one or more entities, human or machine, are capable of controlling a vehicle in the synthetic world by means of a control unit, this is called driving simulation. There are several possible applications for driving simulators for human drivers. Besides entertainment and driving school or safety training, they are increasingly used in the scientific field like in [45, 66, 67, 173]. Such simulation systems are particularly suitable for observing and evaluating human behavior in critical driving situations without endangering anyone seriously. The driving behavior should be as authentic as possible so that the driver experiences situations like in a real vehicle. On the hardware side, this includes a control unit (steering wheel and pedals), a powerful computer with a graphics and sound card, one or more displays and a driver's seat including vehicle body. In addition, realistic software is required that not only provides graphical accuracy, including realistic lighting conditions, but also simulates the movement of various road users in compliance with traffic regulations.

By building a custom driving simulator, we would like to cover two use cases. One is to generate data quickly and safely, which increasingly consists of safety-critical

driving situations, and the other is to use it as a demonstration object to show the world how AI works and to enable people to assess AI more realistically.

## 3.2 Software

Game engines such as Unreal Engine [160], Unity [168], CryEngine [37], etc., provide state-of-the-art graphics and sophisticated physics modeling to create realistic environments. In turn, increasingly realistic and lifelike scenes are enabling the use of AI to improve autonomous driving. Based on a game engine, various driving simulations have been developed in recent years that can be used by scientists and companies to emulate the real world as closely as possible in the virtual world. Even though driving simulations for automated driving like NVIDIA DRIVE Sim™ [127], Apollo Simulation [6], Autoware [162] or SVL Simulator, formerly LGSVL Simulator [144], have their raison d'être, the open source driving simulation software CARLA [44] gains great popularity with its unrestricted changeability.

Since CARLA is open source and free, has a large and active community, receives regular updates and new features and the number of publications using CARLA is steadily increasing, this software serves as the basis for the driving simulator.

Most of the upcoming information about CARLA and the Unreal Engine is taken from the documentation of both[3][4] and from own user experience. If other sources are used, they will be mentioned separately.

### 3.2.1 CARLA

CARLA, which stands for *CAR Learning to Act*, is an open source driving simulator that can be used for data generation and/or testing of AI algorithms [44]. It uses the gaming engine Unreal Engine 4 [160], which calculates and displays the behavior of various road users while taking physics into account, thus enabling realistic driving. CARLA is mostly written in C++ but can be handled via Python API (*Application Programming Interface*). The installation of CARLA is simple and the documentation well written, so that the included scripts run without major problems. Furthermore, the CARLA platform can be modified and adapted to one's own use case, which is essential for the planned experiments.

[3] https://carla.readthedocs.io/en/0.9.13/ accessed: 2022-11-16
[4] https://docs.unrealengine.com/4.26/en-US/ accessed: 2022-11-16

**Unreal Editor** CARLA uses a modified fork of the Unreal Engine 4, which needs to be first cloned from the GitHub's page. This requires a GitHub account and an Epic Games account to be linked together. Afterwards, the engine has to be compiled from the cloned repository to create an executable program. This process is called *software build* which is a general term in the world of software development. This installs the so-called Unreal Editor, which can be used to edit the levels or, in the case of CARLA, the individual maps.

When using the Unreal Editor in CARLA, two different licenses - Unreal®Engine End User License Agreement (EULA)- were selectable: EULA for Publishing and EULA for Creators. Using the Unreal Engine for research purposes, the EULA for Creators was the right choice. In 2022 the two licenses were replaced by a single Unreal Engine EULA that covers all use cases [51, 52].

**Installation and Start** Several versions of CARLA have been released during the work on this dissertation, with version 0.9.14 being the most recent at the time of writing, whereas most of the research has been conducted using modified code from version 0.9.10. Figure 3.1 shows a timeline of the most important changes between the versions.

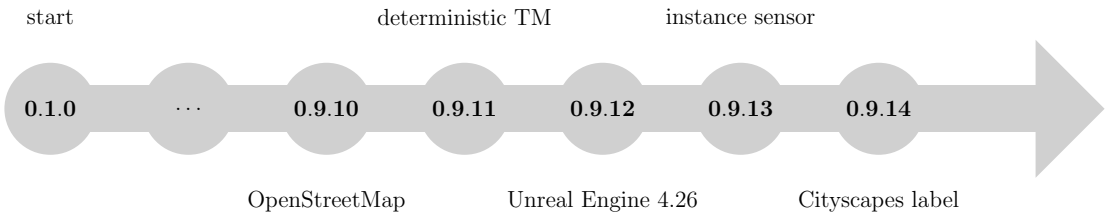



Figure 3.1: CARLA timeline with the biggest changes from version 0.9.10, where the OpenStreetMap integration occurred, allowing real streets to be integrated into a custom CARLA map. Version 0.9.11 brought a fully deterministic traffic manager (TM) that produces the same results and behaviors under the same conditions. In version 0.9.12, the graphics became more realistic due to a newer Unreal Engine version. Version 0.9.13 brought an instance segmentation sensor and the latest version modified the semantic segmentation class labels to be the same as the Cityscapes ones.

CARLA can be installed on various operating systems, however for licensing and customization reasons the Linux distribution Ubuntu LTS 18.04 was chosen. CARLA can be installed in three ways: building it on your own, downloading it as a prepackaged version or running it in a Docker container. The software uses

a client-server architecture, where the server manages everything related to the simulation, such as rendering objects and sensors, physics computation or world state updates, see Figure 3.2.

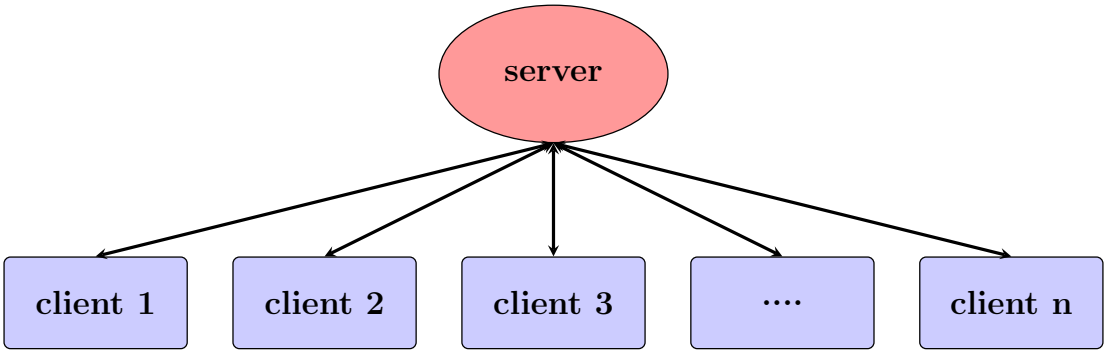



Figure 3.2: Server-client architecture in CARLA.

The CARLA server can be started either directly via the executable file or via the Unreal Editor. After that a window opens, which shows a virtual city. Using the keyboard and mouse pointing, the user can look around the whole world from the point of view of an all-seeing observer, called *spectator*. This is the active server that clients can connect to via Python API to interact with the world. The Python API is either build, already precompiled from the download file of CARLA or by using CARLA's pip package from `pypi.org`. The latter is only possible from version 0.9.12 and higher.

**Clients** The user is able to execute various scripts that interact as a client with the CARLA world via the Python API. In addition to custom scripts, the user can use ready-made clients such as the traffic manager (TM), which takes the control of all non-player characters (NPCs). The TM is a module, written in the programming language C++, that can be connected to CARLA as a client and populates as well as controls actors in CARLA to simulate realistic urban traffic conditions. To avoid unexpected and erroneous results it is important that both, the server and the TM are running in synchronous mode. The TM works as follows: First the current state of each traffic participant including position, velocity and additional information is stored. Furthermore, all entries of removed or defective traffic participants are cleaned up. Second, the movement of each actor is calculated to apply all the calculated positions of the actors as the third and last step.

Since traffic control is computationally intensive for each traffic participant, a hybrid physics mode was implemented in CARLA that removes the physics bottleneck. This is done by elimination of physical movements of road users outside a

certain radius around the ego vehicle. The road user movements are done by teleportation in each simulation step and are only changed back when they reappear within the radius of the ego vehicle.

Additional traffic simulators such SUMO (Simulation of Urban MObility) [110] or PTV Vissim [54] can be connected to CARLA to generate specific movements or behaviors of the road users.

Furthermore, the CARLA documentation provides a tutorial that introduces the main features and allows to create the first custom script. In addition to the tutorial, the software provides ready-made scripts intended for interaction with the virtual environment. For example, there is a script to control an ego-vehicle with a steering wheel that enables discovering the different maps or a recording script that stores spatial information about each road user so that previously driven drives can be reloaded which allows the subsequent storing of data with different sensors.

Combining the two scripts and adding a custom sensor called inference sensor were part of this thesis. This sensor produces an output using a pre-trained semantic segmentation network such that it is possible to interact with the networks output in real-time. **"Driving with the eyes of the AI"**, so to speak. Figure 3.3 shows two camera-based (RGB and semantic segmentation), one point cloud based (LiDAR) and the self-designed inference sensor.

**Simulation Time** In CARLA it is necessary to distinguish between the real time and the simulated time, since the latter has its own clock and is controlled by the server. The elapsed time between two simulation moments can be configured between a few milliseconds up to several seconds and is called time step. Depending on the task, a fixed or variable time step can be chosen. The fixed time step was mainly used in this thesis, which is important to mention since the variable time step is the default value in CARLA.

The client is in total control over the simulation in synchronous mode with a fixed time step. For an optimal physical substepping the constraints

$$t_\delta \leq t_{\text{substep}} n_{\text{substeps}} \ , \tag{3.1}$$

$$t_{\text{substep}} \overset{!}{\leq} 0.01666 \ , \tag{3.2}$$

need to be fulfilled, where $t_\delta$ is the fixed delta in seconds, $t_{\text{substep}}$ the maximal substep delta time and $n_{\text{substeps}}$ the maximal substeps.

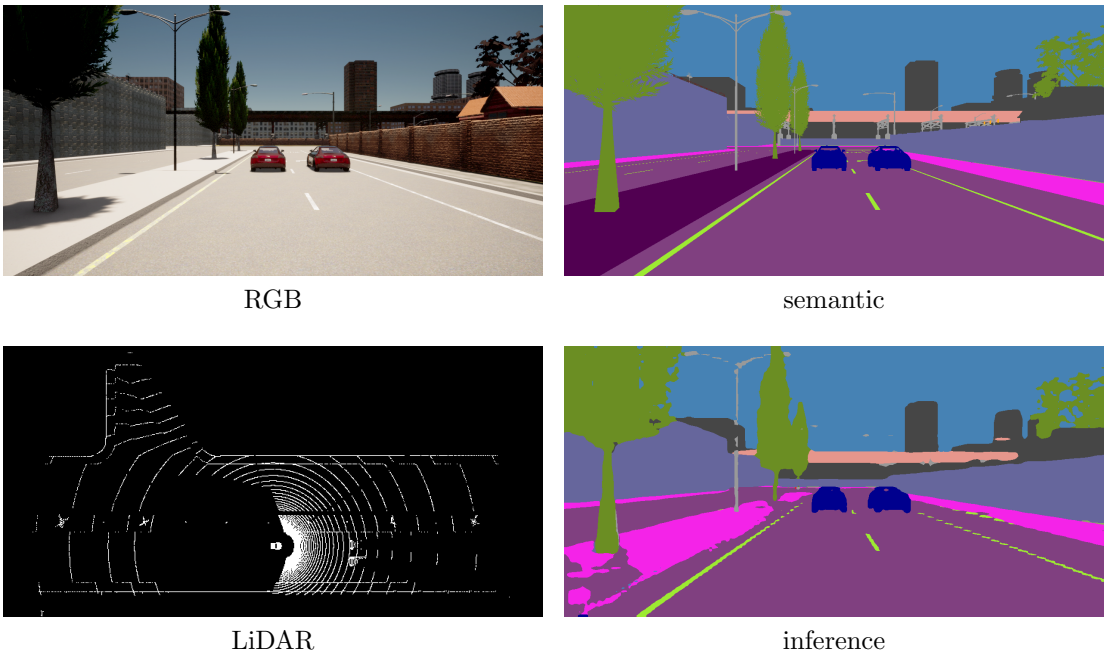



Figure 3.3: Examples of CARLA sensors including the self-designed inference sensor showing the exact same situation.

**Actors and Blueprints** Actors are all elements that interact directly or indirectly with other elements in the simulation and can be placed into or removed from a certain map. They contain all information about the model packed into so-called assets, consisting of all the methods, attributes and rendering information like meshes, textures and surface materials.

Blueprints in CARLA are predefined and ready-made actors which the user can incorporate into the simulation. They can include animations or attributes which can be partially modified, depending on the type of blueprint. All available blueprints in CARLA are listed in the blueprint library and can be *spawned* into the world, a term more commonly used in simulations or games than the word placed. Every blueprint gets a unique ID which can be used to change attributes or to receive information. There are fixed and changeable attributes. For example vehicles have fixed attributes like the number of wheels - depending on the blueprint model - and changeable attributes like color or speed.

Blueprints can be placed via PythonAPI in the CARLA world, this can be done with a modification of the Unreal Editor coordinates. The visibility of blueprints depend on their surface, which is determined by their geometry (created through meshes or brush surfaces), and their material (defined by texture and material parameters). Meshes define the actual geometry of an object/actor which creates the surface. The Unreal Engine distinguishes between 3 different mesh types:

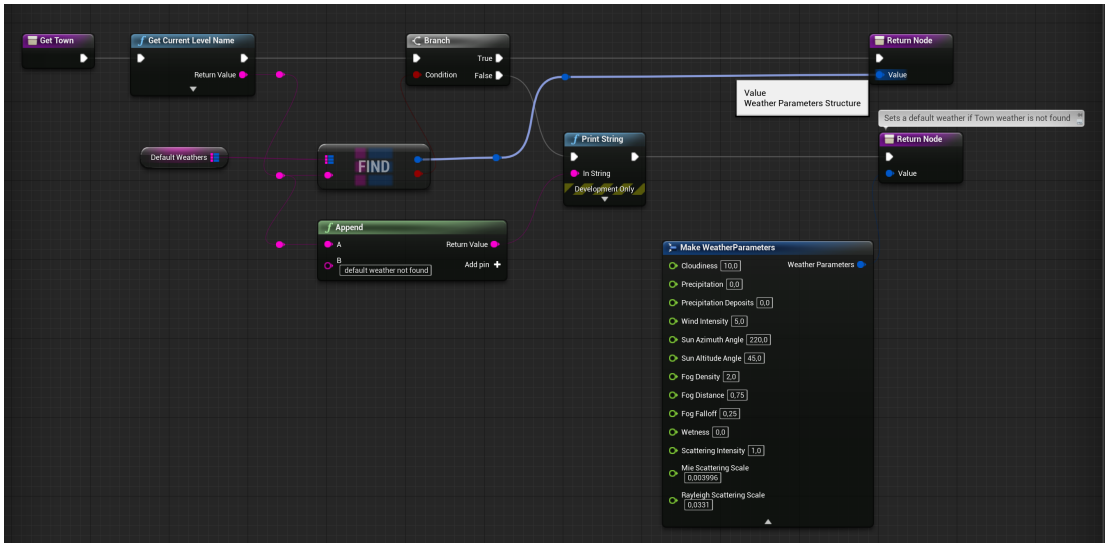



Figure 3.4: Weather blueprint graph for CARLA in Unreal Engine (UE). The gameplay scripting language is called *Blueprint Visual Scripting* based on a node graph where developers can use the programming language C++ to make blueprint nodes. Since the graphs in UE can be displayed and edited visually, no C++ programming skills are required to make logical connections, which is helpful for game designers or artists. Blueprint nodes imply a self-contained functionality performing something unique and exist in different types which are listed in Table 3.1. [140]

static, dynamic and skeletal. Materials define surface properties of objects such as color, reflectivity, unevenness, transparency, etc. They dictate the engine exactly how a surface should interact with light. Components of material are textures and several material parameters. If the light behavior of the real world needs to be simulated, physically based materials are required. Therefore, material attributes such as base color, roughness, metallic and specular have to be used. Textures are image files that are mapped to surfaces which the material is already applied to. They provide pixel-wise information about color, gloss, transparency and some other aspects.

However, there are also functional blueprints like sensors that have no visible surfaces and can only be attached to other blueprints.

**Sensors** Sensors in the CARLA world are actors, a common term in the Blueprint Visual Scripting language and not to be mistaken with the mechanical term, that retrieve data from their environment. They are attached to a vehicle and collect data either at each simulation step or at a specific event. In CARLA there are three types of sensors: cameras, detectors and others which are listed in Table 3.2. *Camera sensors* capture a special viewpoint of the CARLA world and return an

| Type | Description | CARLA examples |
|---|---|---|
| Class | Most common blueprint type. Functionalities can be created visually using a graph and stored in a content package called asset. Blueprint classes can be placed as instances into maps and interact with other blueprints. | Vehicles, walkers, lights, propfactory, weather, sensors |
| Level | Level-wide global event graph. It is the gameplay area that contains everything a player can see and interact with. | Maps |
| Interface | Interacting functions that connect blueprints together so that different types of objects can share particular functionality. | The *BoundingBox()*-function places a 3D frame around an actor or element to describe the geometry. |
| Data-only | Blueprint classes without node graphs. They inherit from its parent class and are used to make small changes. No new elements can be added. | All walkers are data-only blueprints except of the parent walker. |
| Macro | Set of nodes placed as an instance for reusing and time saving purposes. Macros are collapsed graphs to hide complexity and are collected in the Blueprint Macro Library. | - |

Table 3.1: Overview of all blueprint types in Blueprint Visual Scripting.

image. *Detectors* work differently. They retrieve data when a specific event occurs. This can be, for example, a lane crossing or a collision returning a special value instead of an image. *Other sensors* include all other sensors that do not fall into these two categories. For example, there are the two LiDAR sensors or the radar sensor, which record a point cloud in space.

**Non-Player-Characters** The term *Non-Player-Characters (NPC)* is used to describe dynamic objects in the simulation that move according to a certain pattern controlled by the traffic manager. In CARLA these objects are vehicles and walkers. The number is variable and limited by the size of the map.

In CARLA there are two - including bicycles and motorcycles - and four wheeled 3D modelled vehicles which are similar looking to their role models. All vehicles possess an autopilot mode, which is not based on any machine learning algorithm. The gear shift and each single wheel of a vehicle can be controlled by physical controls. Also, most vehicles have lights, which can be turned on or off by the user. Although there are many vehicles free to use, it is possible to integrate self created vehicle models into CARLA. Because of the big and active community of this project almost every upgrade of the software provides new models.

Walkers move on the sidewalk with predefined moving patterns, where all bones

| sensor name | type | description |
|---|---|---|
| RGB | camera | photorealisitc image |
| semantic segmentation | camera | subdivision into classes at pixel level |
| instance segmentation | camera | subdivision into classes at pixel level and a unique object ID |
| depth | camera | gray-scale map with depth information |
| optical flow | camera | motion of each pixel of the RGB camera |
| DVS (Dynamic Vision Sensor) | camera | perception of local changes in brightness |
| collision | detector | perception of collisions between actors |
| lane invasion | detector | checks whether the lane line is crossed |
| obstacle | detector | registers obstacles ahead of the ego vehicle |
| LiDAR | other | 4D point cloud with space dimensions and an intensity loss value |
| semantic LiDAR | other | space dimensions, angle of incidence, object index and semantic class of each point |
| radar | other | polar coordinates, distance and velocity of each point |
| IMU | other | retrieve values for accelerometer, gyroscope and compass |
| GNSS | other | for transmission of position and time |
| RSS | other | a mathematical model to ensure safety |

Table 3.2: Available sensors in CARLA 0.9.14

are moving, from one random point to another. In most cases, the streets are crossed at traffic lights or crosswalks. Even if they primarily choose the shortest walking route, a risk factor can be set to control the disregard of traffic rules. The higher this value, the greater the likelihood that a pedestrian will cross the street on the red light or suddenly step into the roadway to create dangerous situations.

**Maps** The map represents the simulated world. By default, six maps (Town01-05 and Town10) are available with the possibility to include four additional ones (Town06, Town07, Town11, Town12). All of them use the OpenDRIVE 1.4 standard [48] to describe for example roads, lanes or junctions. The provided maps are characterized by a great variability. The maps differ in size and shape, as well as in complexity. One can choose between single-lane and multi-lane roads, urban or rural areas. In addition, some maps contain special features such as traffic circles, tunnels, rails, hills, etc. Table 3.3 provides an overview of all available maps (default and the additional ones) in CARLA 0.9.14 and Figure 3.5 presents some examples. Moreover, the user may create his or her own maps or export real maps with OpenStreetMap and include them into CARLA. For this purpose, the map only has to be converted to the OpenDRIVE format. In this way, the real street shape is adopted, but static objects such as buildings or trees have to be added manually.

| map | description |
|---|---|
| Town01 | single-lane normal-sized city |
| Town02 | single-lane small sized city |
| Town03 | complex city with tunnel, traffic circles and unevenness |
| Town04 | highway with a small city |
| Town05 | multi-lane large city |
| Town06 | long highways |
| Town07 | rural environment |
| Town10 | a detailed metropolis |
| Town11 | the biggest map with 400 $km^2$ and long straight road |
| Town12 | business district with skyscrapers, residential areas, highways and rural |

Table 3.3: Overview of available maps in CARLA 0.9.14

Figure 3.5: Overview of the map variety in CARLA. Due to the large number of maps that are provided, several situations can be created.

### 3.2.2 Additional Software

In order to realize the planned project, further software is required for the implementation of the project in addition to the operating system, CARLA and the Unreal Engine. The most important ones are briefly explained in this subsection.

**Git** Git [164] is a distributed version control software, which saves and tracks changes of computer files. The software is free and open source and particularly helpful in programming, so that changes to the code do not immediately lead

to complete failure, but are first tested by using branches, so that the original code is preserved. Even though Git doesn't have to be used compulsorily, it definitely helps when developing algorithms and in conjunction with a service such as GitHub [60] or GitLab [61], these are stored securely and can be made available to other developers.

**CUDA** Compute Unified Device Architecture (CUDA) is a parallel computing architecture from NVIDIA that enables a significant increase in computing power by leveraging the graphics processing unit (GPU). For this purpose, NVIDIA provides a toolkit that includes GPU-accelerated libraries, debugging and optimization tools, a C/C++ compiler and a runtime library. The use of the CUDA toolkit is possible under the most commonly used operating systems (Windows, Linux and Mac OS), as long as a CUDA-capable NVIDIA graphics card is used. [34] CUDA is essential for training and testing neural networks because it enables parallel computation of matrices and thus offers a significant advantage over the CPU.

**TensorRT** NVIDIA's TensorRT [125] is a deep learning software development kit for high-performance inference based on NVIDIA's parallel computing platform CUDA. It generates individually optimized runtime engines for pre-trained models, related to the user's own hardware configuration. There are three installation options for TensorRT, namely the installation from an installation file, via a pip-package and as a Docker container. Additionally, TensorRT needs the package PyCuda to control CUDA via Python.

TensorRT aims at fast processing of input data and uses several optimization techniques with little sacrifice of accuracy. For example, FP16 or INT8 is used instead of FP32, the floating-point format commonly used in deep learning, which speeds up processing. FP16 and INT8 have less memory but also lower accuracy than FP32, but this does not seem to have a major impact on the accuracy of the system when focusing on inference [121, 124]. Furthermore, GPU memory requirements can be minimized by reusing memory and merging layers and tensors by fusing nodes in a kernel optimizes the use of GPU memory. Selecting appropriate data layers and algorithms based on the particular GPU platform and processing data in parallel accelerate inference time, as does the use of dynamic kernels for recurrent neural networks. [125]

The workflow of TensorRT can be divided into two phases, build and deployment, see Figure 3.6. In the build phase, a pretrained model is transformed into a TRT module and several optimization steps are performed. The transformation into a TRT module happens only once and works as follows: First, a model must be defined and trained with an arbitrary machine learning framework. Then,

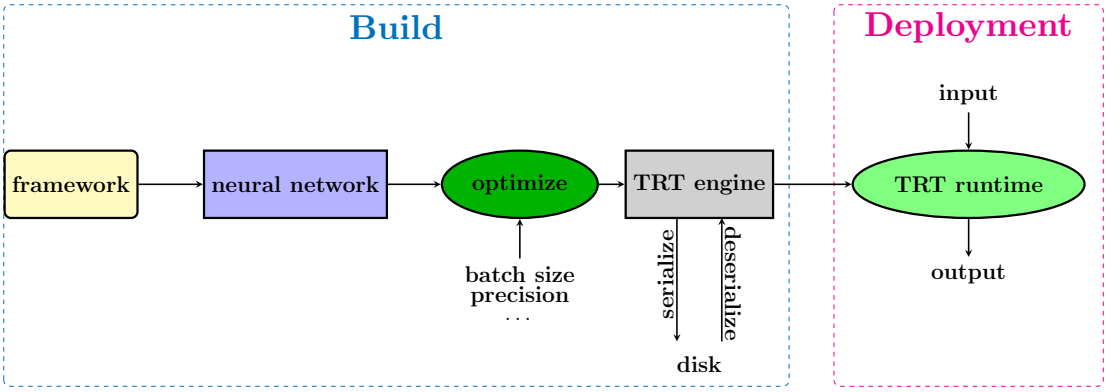



Figure 3.6: The TensorRT workflow is divided into a build and a deployment phase [126].

the model must be converted into the *Open Neural Network eXchange (ONNX)* format [163]. Since the ONNX format is an open format for representing deep learning models, it does not matter which framework is used. The most popular ML frameworks such as PyTorch, TensorFlow, Caffe2, MXNet, Matlab, etc. support the ONNX format. Then the following 6 optimization steps are performed, if possible, to save the TRT module afterwards:

1. precision reduction
2. layer and tensor fusion by merging nodes
3. kernel auto-tuning for selecting the best configuration related to the graphics card/hardware in use
4. dynamic tensor memory through reuse of tensors
5. multi-stream design for parallel processing
6. time fusion for recurrent networks

The deployment phase runs on an embedded device where the TensorRT software needs to be installed. Afterwards the converted model from the building phase can be imported and a batch of input data can be processed. In contrast to training, only the forward path takes place during inference.

## 3.3 Hardware Components

To create a realistic driving experience, hardware components are needed to interact with CARLA. In addition to a powerful computer, monitors including mounts, control units such as steering wheels and pedals, and driving seats are required, which are to be arranged as shown in Figure 3.7. The computer should meet the

system requirements of common driving simulation software while having one or more powerful graphics cards capable of using advanced machine learning models to compute real-time outputs of synthetic images.

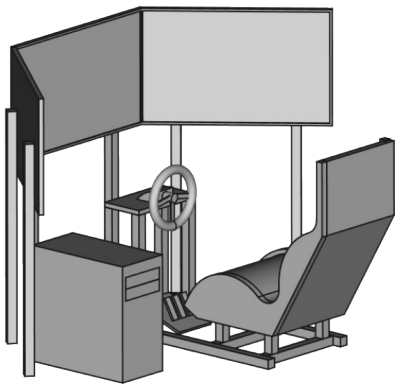


Figure 3.7: Schematic drawing of the planned driving simulator containing the most important components: workstation, seat, control unit, rack and monitors. Created with FreeCAD [143].

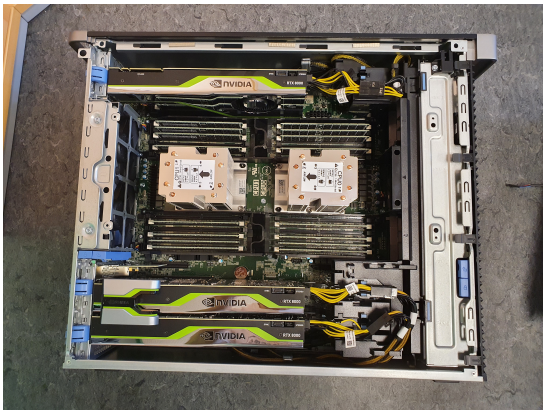


Figure 3.8: Insides of the ordered workstation. Next to 3 graphics cards, the 16 RAM bars and the two cores (covered by the cooling unit) are shown.

**Workstation** To meet the requirements for CARLA - version 0.9.10 was current at the time of procurement - and the planned data collection pipeline with two human drivers, one of whom requires real-time output from the neural network, the following hardware specification was created: A workstation from Dell (Precision T7920) ideally suited for compute-intense applications, see Figure 3.8. Two Intel Xeon Gold 6258R with a base clock speed of 3.0 GHz and turbo boost speed of 4.00 GHz with 24 cores serve as the central processing unit (CPU). A 1 TB ($16 \times 64$ GB) RAM offers high access speeds as well as enough memory to swap computations. For data production, the use of fast hard disk drive storage is advisable, which is why the operating system as well as the simulation and storage of the data is first done on one 2 TB solid state drive (SSD) with the M2 form factor and 3rd generation PCIe (Peripheral Component Interconnect express). The data is subsequently copied to a RAID 5 system equipped with 4 hard disk drives (HDD) containing 12 TB each for further processing. Although SSDs with PCIe interfaces are faster - write speeds are about 11.6 times faster and read speeds are 14.4 times faster than an HDD with a SATA interface - they are more expensive and have a shorter lifespan than HDDs, which are still used for storing large amounts of data. [122, 172]

To have redundant data storage, it is recommended to use a RAID (Redundant Arrays of Inexpensive Disks) system, where several HDDs are virtually connected

to one another so that no data is lost in the event of a hard disk failure. Depending on space, cost and application, different RAID systems can be arranged, with RAID 5 being used for the simulator. [135] It uses parity for data redundancy instead of mirroring, which requires less storage space. Because mirroring keeps multiple copies of data, RAID 5 can restore a failed drive using the parity data that is not stored on a single hard drive. This configuration allows the contents of 3 hard drives to be backed up redundantly using 4 hard disks.

In addition, a powerful graphics processor must perform real-time inference as quickly as possible to enable driving on the prediction. Table 3.4 gives an overview of the common graphics cards for workstations at the time of ordering. Since the RTX A6000 was still quite new on the market, it was not possible to get one. For price/performance reasons, we opted 3 NVIDIA Quadro RTX 8000 with 48 GB storage. Opting for multiple graphics cards has 2 advantages: flexibility in implementing the task and dual use. First, it wasn't clear whether using a single graphics card would be enough. Multiple graphics cards mean more flexibility to move multiple processes to different cards if needed. Furthermore, it is possible to connect 2 identical graphics cards via SLI (*Scalable Link Interface*) - a technology from NVIDIA - to increase performance. On the other hand, the goal was to be able to use the graphics cards for training neural networks as well, so that $3 \times 48$ GB are available for training.

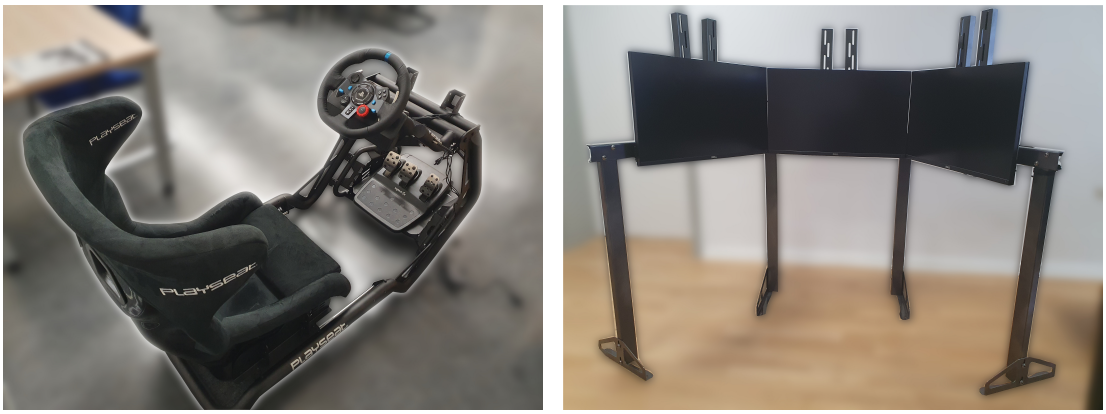
| **Specs** | **Quadro P6000** | **Quadro GV100** | **Quadro RTX 8000** | **Quadro RTX A6000** | **H100 (SXM socket)** |
|---|---|---|---|---|---|
| Launch | Oct. 2016* | Mar. 2018* | Aug. 2018* | Oct. 2020* | Mar. 2022* |
| Architecture | Pascal | Volta | Turing | Ampere | Hopper |
| Transistors [Mrd] | 11.8* | 21.1* | 18.6* | 28.3* | 80.0* |
| GPU memory [GB] | 24 | 32 | 48 | 48 | 80 |
| Memory bandwidth [GB/s] | 432 | 870 | 672 | 768 | 3350 |
| Memory type | GDDR5x | HBM2 | GDDR6 | GDDR6 | HBM3* |
| Base/Boost clock [MHz] | 1506*/1645* | 1132*/1627* | 1395*/1770* | 1410*/1800* | 1065*/1780* |
| Memory clock [MHz] | 1127* | 848* | 1750* | 2000* | 1500* |
| effective [Gbps] | 9* | 1.7* | 14* | 16* | 3* |
| CUDA cores | 3840 | 5120 | 4608 | 10752 | 8448* |
| Tensor cores | - | 640 | 576 | 336 | 528* |
| RT cores | - | - | 72 | 84 | - |
| Single-Precision performance [TFlops] | 12.63* | 14.8 | 16.3 | 38.7 | 30.07* |
| TMUs | 240* | 320* | 288* | 336* | 528* |
| ROPs | 96* | 128* | 96* | 112* | 24* |

Table 3.4: Specifications for workstation graphics cards available late 2020. Informations from NVIDIA's official data sheets. *Missing items from: `www.techpowerup.com/gpu-specs/`. The last column shows the latest graphics card at the time of writing.

**Control Unit** The control unit represents the interface between human and machine. It enables the human to control a vehicle in CARLA freely via the steering wheel and the brake or throttle pedals. The device of choice was the Logitech G29 [108], which is also pre-implemented in CARLA's control script and can therefore be used as a controller almost without any problems. Only the use of the force feedback, which imitates vibrations, caused problems under the Linux distribution used. However, thanks to the open source software *Oversteer - Steering Wheel Manager for Linux* [5], it was possible to set at least a fixed resistance value so that the user gets a more realistic steering experience as without. In addition, the steering wheel has a steering range of 900 degrees.

**Monitors** Since the terms monitor, display, and screen are sometimes used as synonyms, we will introduce the following definitions during this thesis to avoid misunderstandings. Monitors are stand-alone devices that connect to another device, usually a computer, to display information. It possesses a display in the front, which consists of a glass and a panel underneath. Nowadays, LCD (*Liquid Crystal Display*) screens are usually equipped with TN (*Twisted Nematics*), VA (*Vertical Alignment*) or IPS (*In Plane Switching*) panels, the latter being characterized by purer and more natural colors as well as viewing angle stability. All-in-one devices such as mobile phones or laptops have built-in displays, which is why they are not referred to as monitors. A screen is the entire visual output, which can be spread across multiple monitors. For the driving simulator 4 monitors are needed, 3 for the graphical output, which together represent a large screen, and one for the control station, which is used to start the CARLA world and to make all settings. The monitors are attached to a TV mount and placed in front of the driver's seat. It is important to ensure that the driver's eyes are able to see all 3 displays at the same time. All monitors - labeled Dell UP2716D - have an IPS panel, a display diagonal of 68.47 cm (27 inches) and a display resolution of $2560 \times 1440$ (WQHD). With a refresh rate of 60 Hz, a response time of 6 ms and a maximum brightness of 300 $cd/m^2$, the displays are well suited for the project.

**Rack and Seat** The rack is the steel frame of the driving simulator that carries seat, steering wheel and pedals, see Figure 3.9a. Since the simulator is also supposed to serve as a demonstration object, it was necessary during procurement to ensure that the rack is height-adjustable and at the same time stands firmly on the ground so that it can be used by different human body types. In addition, the simulator should be easy to assemble and disassemble so that it can be set up in other locations.



(a) Rack with seat and control unit. (b) TV mount with 3 monitors.

Figure 3.9: Rack, control unit and tv mount.

Figure 3.10: The fully assembled simulator with control center including peripheral devices and a monitor. The simulator can be used for demonstration purposes by driving freely in the virtual world of CARLA with support of a control unit and 3 monitors.

## 3.4 Code Adaptation and Acceleration

For the realization of the research project, we used the python API to modify the script for manual control from the official CARLA 0.9.10 repository. In doing so, we added another sensor - the inference sensor - which evaluates the CARLA RGB images in real-time and outputs the neural network's prediction on the screen. This involved training a semantic segmentation network according to the *Model Development Process* from Section 2.7 and providing it to the inference sensor.

By connecting a control unit including a steering wheel, pedals, a seat and a screen to CARLA, we enable a human driver to experience perception from the network's point of view. We are, so to speak, driving with the eyes of the AI,

which is why we call our approach:

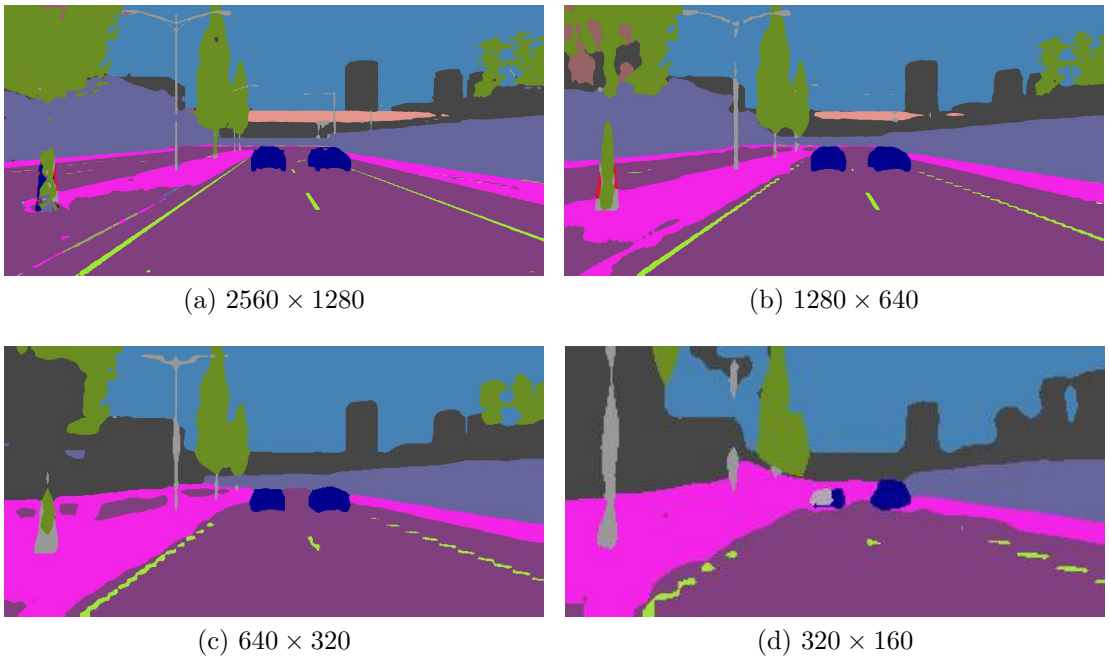
***A-Eye: Driving with the eyes of AI***.

The first attempts to implement a segmentation network - DeepLabV3 [27] with two different backbones (ResNet50 and ResNet101 [72]) - in CARLA using the inference function showed that it is possible to drive on the output of the segmentation network. However, it also became clear that smooth driving is not possible at 0.99 fps respectively 1.27 fps with a resolution of $2560 \times 1280$, which is why some optimization steps were still to be made. Real-time performance for semantic segmentation is described with about 24 fps [180]. This corresponds to the standard frame rate for cinema and television and ensures fluid looking movements with low enough latency.

**Resolution** The resolution should be inversely proportional to the frame rate, since segmentation networks have to predict fewer pixels for smaller images. Consequently, a suitable resolution should be found at which an appropriate image resolution with good recognizability of the driving scenes should be found. For this purpose, the time for executing the inference function up to display visualization was measured for 4 different resolutions, each for 2 networks and is shown in Table 3.5. Although the reduction in frame rate may increase as the resolution is reduced, Figure 3.11 also shows that the output quality falls below a reasonable level. Based on these findings, $1280 \times 640$ was chosen as the appropriate resolution, which means 2.46 million fewer pixels to predict than with a $2560 \times 1280$ resolution. A lower image resolution would create a black border on a WQHD resolution screen, so the screen resolution must also be reduced to completely fill the screen. Since only a handful can be selected, $1280 \times 720$ was chosen as the most suitable screen resolution.

| **Network** | | **Resolution** | | | |
|---|---|---|---|---|---|
| | | **2560x1280** | **1280x640** | **640x320** | **320x160** |
| DeeplabV3-ResNet101 | speed/image [ms] | 1.012 | 0.247 | 0.073 | 0.033 |
| | framerate [1/s] | 0.988 | 4.053 | 13.79 | 30.25 |
| DeeplabV3-ResNet50 | speed/image [ms] | 0.788 | 0.202 | 0.059 | 0.022 |
| | framerate [1/s] | 1.269 | 4.959 | 16.86 | 45.82 |

Table 3.5: Overview of the different framerates according to resolution size for DeeplabV3 with two different backbones. It becomes clear that the resolution has a big influence on the frame rate.



(a) 2560 × 1280 (b) 1280 × 640

(c) 640 × 320 (d) 320 × 160

Figure 3.11: Resolution study with DeepLabV3-ResNet-101 to determine the best cost-benefit ratio. Although reducing the resolution from (a) to (b) barely affects perception, further reduction is not recommended because it makes perception much less accurate.

**Segmentation Model** Although the reduction of the resolution led to an improved frame rate, smooth driving with 4–5 fps was still not possible. Therefore, the next step was to try out different segmentation networks. With nearly 61 million parameters DeepLabv3 with ResNet-101 backbone does not seem to be applicable for real-time processing (42 million parameters for DeepLabV3-ResNet50), so more suitable networks were sought. Therefore, a search for suitable real-time segmentation networks was an initial step. The following Table 3.6 lists the fastest five segmentation networks according to the authors of [133] on real-time inference for the Cityscapes test dataset in 2021.

We see that all models work with a smaller number of parameters than the first two used. Also, almost all values are created with NVIDIA's deployment tool TensorRT, except for the ones from Fast-SCNN. This was one reason why we initially focused on Fast-SCNN, as it was fast enough with 8.1 ms per frame at $2048 \times 1024$ resolution to run initial tests without additional software. Looking at the fastest segmentation networks on the official Cityscape's leaderboard page[5], there are 2 Fast-SCNN models in the top 3, as long as the input resolution is halved or quartered, see Table 3.7. With a speed of 3.5 ms per frame and a mIoU

[5]https://www.cityscapes-dataset.com/benchmarks/#scene-labeling-task and listed by the best runtime, accessed: 2023-02-24

| network | year | speed [ms] | fps [1/s] | mIoU [%] | $n_p$ [mio] | $\gamma$ | GPU | TensorRT |
|---|---|---|---|---|---|---|---|---|
| STDC1-50 [53] | 2021 | **4.0** | **250.4** | 71.9 | 8.44 | 0.5 | GTX 1080Ti | v5.0.1.5 |
| FasterSeg [28] | 2019 | 6.1 | 163.9 | 71.5 | 4.4 | 1.0 | GTX 1080Ti | v5.1.5 |
| BiSeNetV2 [180] | 2020 | 6.4 | 156.0 | 72.6 | 5.23* | 0.5 | GTX 1080Ti | v5.1.5 |
| Fast-SCNN [139] | 2019 | 8.1 | 123.5 | 68.0 | **1.11** | 1.0 | Titan XP | - |
| DDRNet-23-slim [132] | 2021 | 9.8 | 101.6 | **77.4** | 5.7 | 1.0 | GTX 2080Ti | v6.0.1 |

Table 3.6: Overview of the fastest five real-time segmentation models evaluated on the Cityscapes test dataset in 2021 according to [133]. $\gamma$ represents the downsampling ratio corresponding to the Cityscapes resolution of $2048 \times 1024$ for inference and $n_p$ the number of model parameters. Each entry was taken from the corresponding publication, if mentioned. If not, they were determined from another source and marked with *.

of still 62.8 %, the network still seems reasonable with half resolution, so upscaling the network's output is an additional optimization option in the future.

| resolution | $\gamma$ | mIoU [%] | drop [%] | speed [ms] | speedup [%] |
|---|---|---|---|---|---|
| $2048 \times 1024$ | 1.00 | 68.0 | - | 8.10 | - |
| $1024 \times 512$ | 0.50 | 62.8 | 7.647 | 3.50 | 56.79 |
| $512 \times 256$ | 0.25 | 51.9 | 23.68 | 2.06 | 74.57 |

Table 3.7: Speedup possibilities for Fast-SCNN through resolution reduction according to Cityscapes leaderboard[5].

Fast-SCNN, which stands for *Fast Segmentation Convolutional Neural Network*, uses two encoder branches which are added before classification. As shown in Figure 3.12 the networks architecture is composed in four modules: learning to downsample, global feature extractor, feature fusion and classifier.

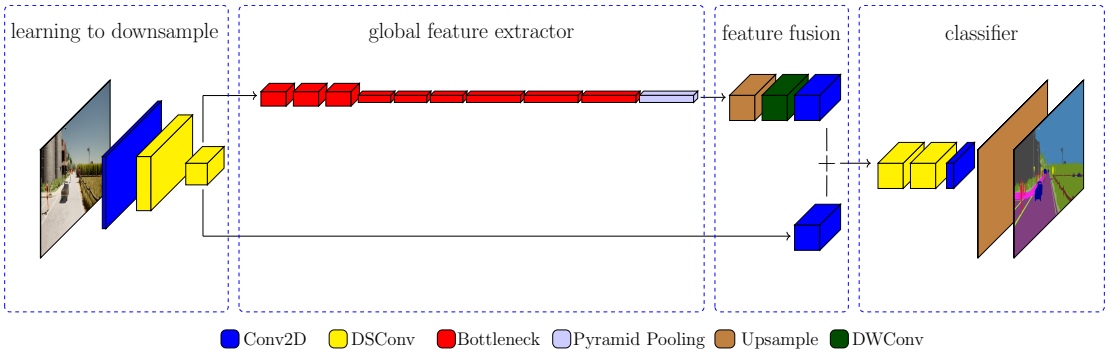



Figure 3.12: Fast-SCNN architecture. [139].

1. *Learning to downsample*
   This module consists of 3 layers, the first consisting of a typical 2D convolution followed by 2 depth-wise separable convolutions (DSConv). The

latter, as shown when Francois Chollet introduced the Xception architecture [32], are more computationally efficient than the standard ones and reduce overfitting. DSConvs consists of 2 parts: first, depth-wise convolution is performed to merge the outputs in the second step. Unlike standard convolution, where the convolution operator is applied to all input channels, depth-wise convolution (DWConv) is performed only for a single one separately. Afterwards all channels are combined, resulting in a single-channel output tensor. A kernel of size $3 \times 3$ and a stride of 2 is used for all layers, followed by batch normalization and ReLU as activation function.

2. *Global feature extractor*
   Working with nine inverted residual bottleneck blocks helps to detect global features due to the relatively large input resolution (1/8 of the original image resolution). Also, all bottleneck blocks use DWConvs. Last, pyramid pooling is added to accumulate contextual information based on different regions.

3. *Feature fusion*
   For efficiency, the features of the two branches are added. This is done by upsampling the output of the global feature extractor and applying a DWConv before adding the output of a standard convolution with the output of another standard convolution, which receives the output of the learning-to-downsample module as a skip connection.

4. *Classifier*
   Adding few layers after feature fusion increases accuracy, so that two DSConv and one standard convolution are used before the final softmax activation during training or the less computationally intensive argmax function during inference, respectively.

**Inference Pipeline** Both the resolution reduction and the use of a faster segmentation network were able to improve the frame rate. The next step was to optimize the inference pipeline where the inference sensor plays a central role of the A-Eye approach. This sensor is designed to convert the CARLA image into the form required by the network as quickly as possible. The network then provides a probability value per pixel for each class, with the most likely value being selected and stored in an output array. Since its entries consist of class IDs, these must then be converted to RGB space before being sent to the users screen. This conversion is referred to as mapping. Afterwards, those RGB images are displayed using Pygame [156], a python gaming library which converts NumPy [129] arrays to so-called *surface objects* to represent images to the user. In order to optimize the inference pipeline, it first had to be divided into the following contiguous blocks to measure how long each block takes to compute, subsequently

improving them when possible. Reasonable blocks are preprocessing, inference, mapping and displaying. Table 3.8 provides an overview of the inference pipeline duration, broken down by each individual block, at the beginning and at the end after all modifications have been made.

| **block** | **baseline [ms]** | **modification impact [ms]** | | | | **current state [ms]** |
|---|---|---|---|---|---|---|
| | | a | b | c | d | |
| pre-processing | 37.19 | -18.25 | 0 | 0 | 0 | 18.94 |
| inference | 13.24 | 0 | 0 | 0 | 0 | 13.24 |
| mapping | 65.64 | 0 | -21.70 | -27.95 | -11.77 | 4.22 |
| displaying | 2.77 | 0 | +5.77 | 0 | -3.62 | 4.92 |
| **sum** | 118.84 | 100.59 | 84.66 | 56.71 | 41.32 | 41.32 |

Table 3.8: Overview of all modifications made that affect the clients frame rate. Tests performed with Fast-SCNN on Town03 in CARLA 0.9.10 at a resolution of $1280 \times 640$ with a server framerate of 30 fps. The numerical values are averaged values over a period of at least 10 seconds.

*Modifications made to speed up image processing:*

a. Preventing transformation to another format:
Preprocessing involves converting the raw image signal from CARLA into the suitable form for the network. The signal, a NumPy array, must be sent through the same normalization process as used for training. This was not possible in a direct way due to the following error message: *ValueError: At least one stride in the given numpy array is negative, and tensors with negative strides are not currently supported.* Therefore, a workaround via the image library Pillow [33] was necessary. Although Pillow is intended for fast access and processing of images, time was lost during the transformation. The underlying NumPy error message could be bypassed by copying the array, so no conversion to Pillow format was needed anymore.

b. Mapping outsourcing to GPU:
Unlike the CPU, which processes operations sequentially, GPU units can split large processes into many parts and execute them in parallel. This is especially efficient for operations with many repetitions and provides faster output. Accordingly, outsourcing the mapping of class IDs to the appropriate color values in RGB space to the GPU speeds up as parallel operations can be performed. For this reason, mapping was done with PyTorch arrays instead of NumPy arrays, which was more time-saving. However, after mapping, the array must be loaded on the CPU because Pygame [156], which is essential for CARLA, converts NumPy arrays to surface objects. This is why the displaying part becomes slower.

c. Mapping boost using *torch.where* function from torch library:
The mapping function iterates through a list of all classes found by the network, with each class entry creating a boolean mask array and transferring this to a new array with the corresponding RGB value. Boolean mask arrays are arrays which include only *True* or *False* values. By wrapping the *torch.where* function around a boolean masked array this speeds up the mapping process.

d. Vectorization:
Despite optimized mask generation by PyTorch functions, for loops were still part of the inference pipeline. A technique that works without loops is called vectorization, which thereby speeds up Python code. This was implemented with a Pillow function called *putpallete*. An additional advantage is that a Pillow image can be efficiently converted to a NumPy array, which is needed for Pygame and therefore leads to a speedup in displaying.

After those modifications, the processing time is about 41.32 ms per frame, which corresponds to 24.20 fps. This means that the goal of achieving a real-time performance of more than 24 fps has been achieved on one monitor. Although this is already an acceptable working speed, further optimization steps could allow faster processing with more pixels, so that eventually 3 monitors can be used. The table offers that preprocessing now takes up the largest portion of the processing time, followed by the inference part. Various attempts to speed up the preprocessing failed, so the next step was to try to improve the inference time using a deployment tool, which is a very common way in research and industry.

**Deployment** Training a neural network takes much longer than deployment. During the training phase, the network is allowed to iteratively learn contextual information that should be processed as quickly as possible in the deployment phase. While a large memory size plays an important role during training phase to take advantage of the large number of parameters, efficiency plays a larger role in the second phase so that data can be processed in nearly real-time. Various tricks can be used to achieve faster processing without major sacrifices in accuracy. For example, the authors of [43] were able to show that numerous weights in a neural network are redundant and can therefore be omitted. [3]

Therefore, deployment tools can identify the savings potential and apply it to the hardware configuration used. A commonly used deployment tool is NVIDIA's TensorRT [125], which optimizes computation to the hardware used, parallelizes computations and reuses memory.

For this, the fully trained model needs to be converted to a TensorRT model, a step, that takes only few minutes. During deployment, the models run on a graphics card of the workstation where the TensorRT runtime was installed in

the version 8.2.3.0. Afterwards the converted model can be imported and a batch of input data can be processed. Table 3.9 shows the time measurement of two segmentation networks, Fast-SCNN and BiSeNetV2, as standard and TensorRT models, respectively. The pre-/post-comparison of both networks illustrates that the use of TensorRT models lead to a significant speedup, 2.57× for the Fast-SCNN and 2.75× for the BiSeNetV2. As a result, the total duration of the inference pipeline for the Fast-SCNN network is now 33.23 ms, which corresponds to 30.09 fps.

| **model** | **inference time [ms]** | |
|---|---|---|
| | **standard model** | **TensorRT model** |
| Fast-SCNN | 13.24 | 5.15 |
| BiSeNetV2 | 36.95 | 13.45 |

Table 3.9: Differences in the inference time of 2 models after they were converted to a TensorRT model at a resolution of $1280 \times 640$. The speedup results, for instance, from parallelizing calculations, reusing memory, and reducing accuracy to an acceptable level.

**Field of View** A larger field of view provides a more realistic driving experience because, on one hand, the driver perceives more of the simulation environment and thus the occurrence of perceptual errors is emphasized more. On the other hand, the driver is less influenced by the real environment because he or she can only notice the screen and no other movements in the corner of his eye. For this reason, the driving experience should be presented on three monitors instead of one. As the findings from the resolution study show, an increase in resolution goes hand in hand with a lower frame rate. This is also shown in Table 3.10, which shows that real-time capability could not be achieved at 11.02 fps on a $3840 \times 640$ resolution with Fast-SCNN. Even though the setup with 3 monitors does not seem suitable for the further research project, the setup with 3 monitors can be used for demonstration purposes to show visitors or event participants what semantic segmentation networks perceive and on what basis autonomous vehicles make decisions. Additionally, the lower fps are barely noticed in public, but this setup leaves a remaining impression, which almost every person could attest us so far.

| **model** | **time [ms]** | |
|---|---|---|
| | **3 monitors** | **1 monitor** |
| Fast-SCNN | 90.79 | 33.23 |
| BiSeNetV2 | 98.18 | 41.77 |

Table 3.10: Inference pipeline time measurement for 3 and 1 monitors.

## 3.5 Error Occurrence and Correction

This is a list of errors that occurred when setting up the test rig and editing the CARLA code.

**RuntimeError: rpc::rpc_error during call in function version** While working with CARLA versions older than 0.9.12 the client library is loaded via **.egg-file* corresponding to the python version used by the computer. The library provides an interface to control CARLA using python. This type of error occurs when the **.egg-file* is corrupted so that a client is no longer able to connect to the server. Possible troubleshooting options are:

1. rebuilt the **.egg-file*
2. the wrong PythonPath is given
3. there are multiple **.egg-files* and the wrong one is loaded.

In the present case, possibly the hasty unplugging of the power supply led to this error, as the computer had not yet completely shut down and the **.egg-file* became corrupted. Newer CARLA versions also allow loading the client library via a **.whl-file* or as a pip package. The latter can be downloaded and installed via `pypi.org`. Especially when working with the pip package this error should no longer occur.

**Low FPS rate when starting server via the Unreal Editor** Unreal Engine 4 lowers the performance of the server when running in background. This is the default mode and can be disabled. For this purpose, one has to uncheck the box *use less CPU in the background* in the editor preferences (Editor Preferences ▷ General ▷ Performance).

**Serialization error using TensorRT** If the error message *Error Code 1: Serialization (Serialization assertion stdVersionRead == serializationVersion failed. Version tag does not match)* appears, it is caused by the fact that the TensorRT version on the machine does not match the one used to convert the neural network into a TensorRT model. It is necessary to use the identical versions for the build and for the deployment to avoid this error message. This happens quickly when working on different machines, so it is recommended to run build and deployment on the same machine if possible.

**Retrospective saving of rides not possible** The use of the driving simulator aims at targeted data generation. This requires free driving without knowing when corner cases occur. To save resources, the recording and playback functions implemented in CARLA are used. The recording function saves the most important information of all road users per frame, which includes e.g. coordinate information and object parameters. These can later be loaded by the replay function and CARLA reconstructs the original rides from this information. The rides can be repeated as often as desired and saved from different angles and with different sensors.

In the beginning there were some problems with the self-designed reloading script, because the ego-vehicle gets its own ID, which is used during the subsequent recording from the ego-perspective. However, spawning takes longer, which resulted in replaying without catching the ego-vehicle and thus ending in the zero point of the respective map. A longer pause before starting the recording solved the problem.

Chapter 4

# Corner Cases in Autonomous Driving

This chapter first introduces the levels of autonomous driving before the term corner case and its various types are described in more detail. Following this definition, three publications are presented that contribute scientifically in this field. The first one works on a methodical basis, where a redundant evaluation of the scene is carried out by means of two sensors, using different technologies to absorb sensor driven corner cases. The second publication describes the creation of new datasets with so-called Out-of-Distribution (OoD) objects, which represent corner cases at the object level. In addition, methods for detecting, tracking and clustering OoD objects in video sequences are presented. Whereas the third publication is a listing of datasets containing OoD objects in the context of autonomous driving and is intended to serve as an overview for researchers in this field.

## 4.1 Levels of Driving Automation

The Society of Automotive Engineers (SAE) has introduced a taxonomy for levels of driving automation [80] which is based on [130]. Arising from this is a classification of automation into 6 levels of motor vehicles, ranging from no automation to full automation, see Table 4.1. *Level 0* describes a state of no automation, where the human driver is in full control of the driving situation. In *Level 1*, the driver retains permanent control of the vehicle and is supported by single assistance systems such as cruise control or lane departure warning. In *Level 2*, the driver remains in control of the vehicle, with some tasks being performed without human interaction. To achieve this, single systems are combined in such a way that individual tasks can be performed autonomously, e.g. automatic parking or lane keeping by steering coupled with braking and accelerating the vehicle. Unlike

in *Level 1*, the driver is allowed to take his hands off the steering wheel for short time periods, while all assistance systems must be constantly monitored by the driver, as he will be held liable for accidents that occur due to malfunctions. *Level 3* describes a situation in which the driver is allowed to take his eyes off the road and hands off the steering wheel since the vehicle temporarily operates on its own. The driver must be able to control the vehicle at any time, as the vehicle should raise a signal in case of problems or uncertainty. In those cases, the algorithm offers proposals or solutions, which should be confirmed or rejected by the driver. In *Level 4*, the vehicle controls autonomously in predefined situations, so that the human takes over the observer role and assumes control when the situation changes. For example, a vehicle could take control when traffic conditions are reasonable, such as good weather and little traffic, and then hand it back to the human when conditions change like heavy rain or increased traffic. *Level 5* marks the target state in which the human takes over the passenger role and the vehicle is controlled fully autonomously, regardless of the complexity of the situation.

| level | name | description |
|---|---|---|
| 0 | no driving automation | driver in full control |
| 1 | driver assistance | driver supported by single systems |
| 2 | partial driving automation | single tasks performed automatically |
| 3 | conditional driving automation | autonomous driving with human supervision |
| 4 | high driving automation | autonomous driving under predefined conditions without human supervision |
| 5 | full driving automation | no human driver needed |

Table 4.1: Levels of driving automation according to ISO/SAE PAS 22736:2021.

These levels describe the states that autonomous driving can assume. The legal situation in Germany currently permits high driving automation, but only under certain conditions[6]. Namely, only in pre-approved areas under regular supervision by a technical supervisor. Even though today's algorithms work well, they are prone to errors, especially when it comes to special cases that rarely occur. Therefore, it is necessary for science to make progress in detecting safety-critical driving situations, so-called corner cases, so that policy will allow higher than *Level 3* without restrictions.

## 4.2 Corner Case

When thinking about autonomous vehicles that move safely through traffic, it is necessary to perceive the environment correctly in order to provide safe driving.

[6] Autonome-Fahrzeuge-Genehmigungs-und-Betriebs-Verordnung from 24. June 2022 (BGBl. I p. 986)

Especially the detection of atypical and dangerous situations is crucial for the safety of all road users. In order to improve the ability of today's models to handle such critical situations, datasets are required that allow for targeted training and, more importantly, testing with such critical situations.

Even though there is no uniform definition for the term *corner case* in the context of autonomous driving, most definitions in the literature mean the same thing, namely a rare but untypical safety-critical driving situation. This can include objects on or near the road that were not part of the training data, but also objects that are part of the training data that come together in a complex constellation, resulting in a safety-critical driving situation.

According to [18], a corner case for camera-based systems in the field of autonomous driving describes a *"non-predictable relevant object/class in relevant location"*. This means that the unpredictable happens to moving objects (relevant class) interacting with each other on the road (crossing trajectories). Based on this definition, a corner case detection framework was presented to calculate a corner case score based on video sequences. The authors of [19] subsequently developed a systematization of corner cases, in which they divide corner cases into different levels and according to the degree of complexity. In addition, examples were given for each corner case level. This was also the basis for a subsequent publication with additional examples [20]. Since the approach in these references is camera-based, a categorization of corner cases at sensor level was adapted in [73], where radar and LiDAR sensors were also considered. The authors define the 4 superordinate layers sensor, content, temporal and method, which also take the previously defined levels into account. Since this is a scientifically consistent definition that also considers different sensor modalities, we would like to adopt it. Table 4.2 provides a general overview of the different layers.

**Method Layer Corner Case** While *Sensor*, *Content* and *Temporal Layer* describe corner cases from the perspective of the human driver, the *Method Layer* specifies corner cases in machine learning models due to lack of knowledge. Accordingly, epistemic uncertainty comes into play, which can be addressed by targeted data generation. Therefore, our focus is on this type of layer to increase security. In the following, we explain corner cases of the other layers by providing example images, where we additionally infer each image using a semantic segmentation network to simultaneously show errors of the *Method Layer* when the network has not been trained on such examples.

**Sensor Layer Corner Case** The *Sensor Layer* takes into account all cases where the sensor's hardware or software has led to unclean data and thus produces outliers. For example, misaligned or damaged components provide different data

| corner case layer | description | example |
|---|---|---|
| sensor | unclean data due to hardware or physical problems of the sensor | broken lense, overexposure |
| content - domain level | constant change in visual appearances | weather conditions, left and right-hand traffic |
| content - object level | anomalies on or beside the road | animals, novelty objects, flooded streets |
| content - scene level | known objects in unusual quantity or location | traffic jam, demonstrations, fallen trees lying on the street |
| temporal | consideration of the trajectories of road users in video sequences | pedestrian appears behind a bus, child runs after a ball |
| method | errors/uncertainties in prediction due to lack of knowledge | training in sunny weather, but tested in fog |

Table 4.2: Overview of corner case layers. While sensor, content and time layer denote a corner case according to human understanding, the method layer refers to prediction errors due to epistemic uncertainty.

than the standard, which is safety-critical like shown in Figure 4.1. Common measurement errors occur with the LiDAR sensor, for example, due to intensity values that are too high or too low, or with the radar sensor due to interference phenomena. Furthermore, too small object surfaces can lead to misperception due to the coarse resolution of both sensors. To avoid *Sensor Layer* corner cases, but also to safeguard traffic situations, sensor fusion approaches can be used, which combine data from at least two sensors. These data can either be sent together through a fusion network, or first processed by separate networks in order to fuse both predictions. A way to apply the latter in object detection with the use of image and radar data is described in Section 4.3.

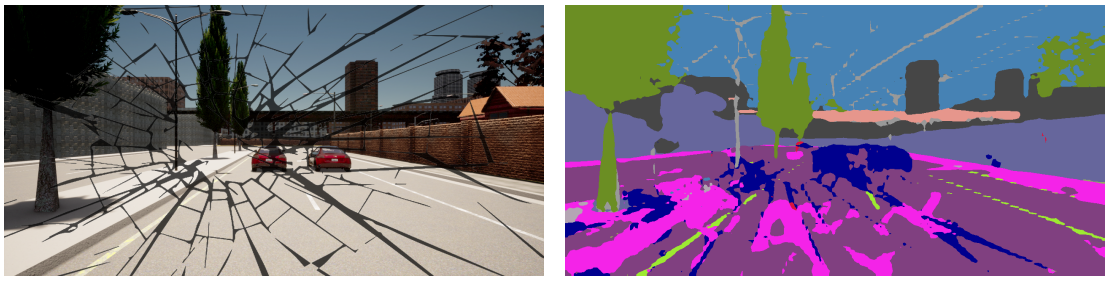


Figure 4.1: A broken lens represents a corner case at the *Sensor Layer*.

**Content Layer Corner Case** This layer is divided into domain, object and scene level. Corner cases at the *Domain Level* describe a constant change of visual appearance, such as driving on roads in other countries as road signs or even traffic routing (left-hand traffic) differs. Especially this kind of corner case is

challenging for machine learning algorithms as they see similar data that consists of one or a few different domains during training. If these are subsequently tested in an unknown domain, problems may arise in the prediction due to a lack of generalizability. This is shown in Figure 4.2, where neither fog nor tunnel images were part of the training data. A typical domain shift also occurs when models are trained on synthetic data but tested on real images.

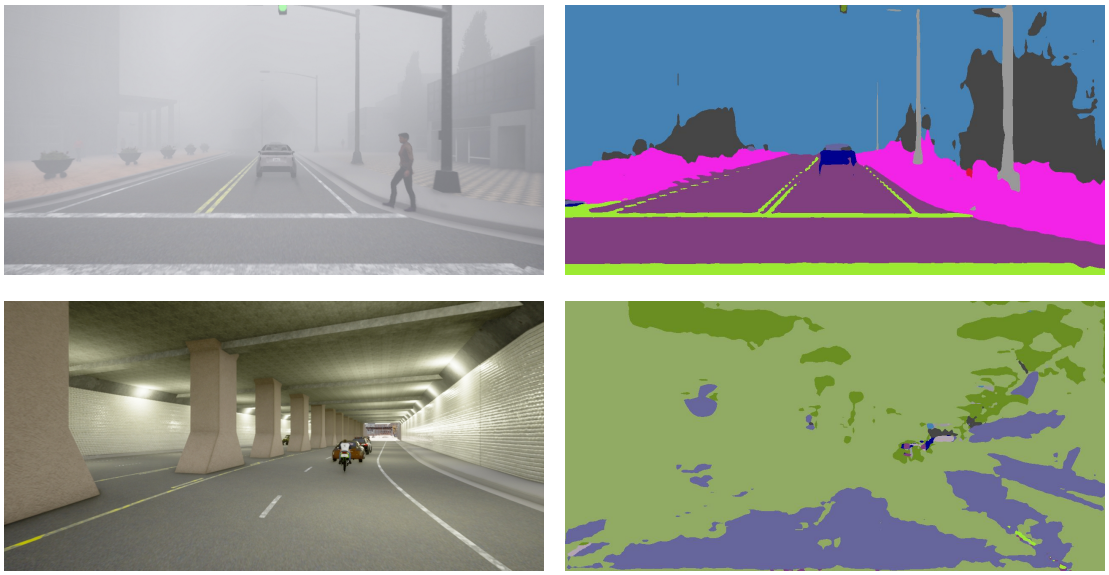


Figure 4.2: Examples of *Domain Level* corner cases. If neither fog (above) nor tunnel drives (below) are part of the training, neural networks have problems dealing with such situations, which can lead to safety-critical situations.

An *Object Level* corner case describes anomalies on or beside the road that are rare but very relevant for safety reasons. For example, when objects that are not naturally found along the road are located there. These include animals, unknown or undefinable objects, among others. Figure 4.3 shows an example of zoo animals on the street, which were not part of the training set. In the context of anomalies, terms such as outlier, Out-of-Distribution (OoD), and novelty are also used, for which there is no clear differentiation. In the *Method Layer* context, we therefore follow the authors of [166], who describe outliers and OoD objects as subcategories of anomalies and refer to them as noise or samples drawn from a different distribution than the one on which the model was trained. Furthermore, the authors define novelties as previously unseen objects that represent a new concept and occur in higher quantities. These can be new items as well as objects that only occur in certain localities like skis and snowmobiles in snowy regions, surfboards on trailers in surfing regions or vehicles that are almost unique to Asia, such as Tuk Tuks.

Science is currently focusing on finding such anomalies, therefore addressing un-

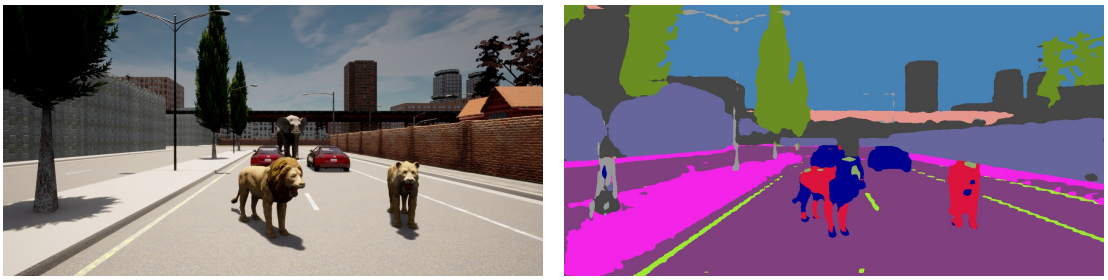


Figure 4.3: Example of an *Object Level* corner case with some zoo animals on the street.

certainty in this area is an important focus to be made, as the authors of [85] have shown that semantic segmentation performance improves when uncertainty is taken into account. That is why some recent scientific work is concerned with the segmentation of OoD [8, 9, 10, 24].

Distinguishing between important and unimportant anomalies is a major difficulty, as anomalies such as leaves on the road do not have to lead to emergency braking. For this reason, more research needs to be done in this area to enable *Level 4 and 5* for autonomous driving. The different levels are described in more detail in Section 4.1.

If known objects occur in an unusually high quantity or if the place where known objects occur is untypical, then we speak of *Scene Level* corner cases. Such examples are demonstrations where many people can be seen at the same time (unusual quantity, see Figure 4.4), or fallen trees lying on the road (unusual location).

Figure 4.4: Example of a *Scene Level* corner case with many pedestrians on the street. A scene that was not present in this form during training and still seems not to be a big problem for the segmentation network. A sign of good generalizability for the human class.

**Temporal Layer Corner Case** The *Temporal Layer* deals with corner cases in video sequences where the trajectories of road users become relevant. Thereby,

contextual information must also be taken into account for the distinction between dangerous and harmless. For example the trajectories of a vehicle and a pedestrian may cross at a traffic light. If the driver identifies the red phase and reduces his speed, the pedestrian can cross the road unharmed, otherwise a safety-critical situation occurs. A more detailed description of corner cases in this layer was done in a joint work [150], which is discussed in more detail in Section 5.3.

Based on this preliminary work, we define corner cases in the field of autonomous driving as follows: Safety-critical driving situations that are challenging for AI algorithms because they were not represented in the training data. These can include anomalies and/or several known objects that interact in an untypical way with each other.

## 4.3 YOdar: Uncertainty-based Sensor Fusion for Vehicle Detection with Camera and Radar Sensors [93]

The prediction of neural networks also strongly depends on the sensor quality. Even data from another sensor manufacturer can lead to misinterpretations. If noise is added to the data due to damage or wear, this can lead to so-called *Sensor Layer* corner cases. But also *Domain Level* corner cases can be irritating for the prediction of camera-based networks if they were not included in the training data. One way to become more robust against such corner cases is to use at least two redundant systems with different sensors. In this way, sensor-specific problems can be intercepted by the other system, leading to increased security. Therefore, in this work we focus on the night domain, which is difficult for camera sensors but leaves the radar sensor unaffected.

Previous studies show that the use of more than one sensor, the so-called sensor fusion, leads to an improvement in object detection accuracy, e.g. when combining camera and LiDAR sensors [65, 68, 107, 157, 174]. Up to now, datasets containing real street scenes using radar data and another sensor are rather the exception, although synthetic data with different sensors, such as camera and radar sensors, can be generated using simulators like CARLA [44] or LSGVL [144].

With the publication of the nuScenes dataset [22], the scientific community obtained access to real street scenes recorded with different sensors including radar. In total there are 5 radar sensors distributed around the car that generate the data. Ever since, the number of published papers dealing with sensor fusion combining radar with other sensors with the help of the nuScenes dataset has increased, see e.g. [83, 123, 141]. We now briefly review these approaches. The authors

of [83] propose an object detection convolutional neural network (CNN) named RVNet which is equipped with two input branches and two output branches. One input branch processes image data, the other one radar data. Similarly to YOLOv3 [142], the network utilizes two output branches to provide bounding box predictions, i.e., one branch is supposed to detect smaller obstacles, the other one larger obstacles. The authors conclude that radar features are useful for detecting on-road obstacles in a binary classification framework. On the other hand the features extracted from radar data seem not to be useful in a multiclass classification framework due to the sparsity of the data.

Another deep-learning-based radar and camera sensor fusion for object detection is the CRF-Net (CameraRadarFusionNet) [123], which automatically learns at which level the fusion of both sensor data is most beneficial for object detection. The CRF-Net uses a so-called BlackIn training strategy and combines a RetinaNet (VGG backbone), a custom-designed radar network and a Feature Pyramid Network (FPN) for classification and regression problems. The main branch is composed of five VGG-Blocks, every block receives pre-processed radar and image data for further processing which is forwarded to the FPN-Blocks. The network is tested on the nuScenes dataset and a self-build one. The authors provide evidence that the BackIn training strategy leverages the detection score of a state-of-the-art object detection network.

Furthermore, a fusion approach for LiDAR and radar is introduced in [141]. This approach is designed for multi-class object detection of pedestrian, cyclist, car and noise (empty region of interest) classes. To this end, LiDAR and radar data are first processed individually. The LiDAR branch detects objects and tracks them over time. On the other hand, the radar branch provides the object classification, where three independent fast Fourier transforms (FFTs) are applied on the range-Doppler-angle spectrum. After time synchronization the two branches are merged, resulting in regions of interest. These regions are fed to the CNN, based on the VGGNet architecture, which computes the classes probabilities. Since this approach works well for vehicle and noise classification but has problems with pedestrians and cyclist classes the network was improved by applying a tracking filter on top of the classifier. They used a Bayes filter which improved the classification performance for the two challenging classes.

In summary, the works presented [83, 123] aim at simultaneously fusing and interpreting image and radar data within a CNN. In [141], LiDAR and radar data are first fused and afterwards a CNN processes the fused input. While these approaches are beneficial with respect to maximizing performance, they require additional fallback solutions in case that a sensor drops out. Also in contrast to other sensor fusion solutions, our approach preserves the option to use both networks redundantly. In this way, indications of only one of the networks can be

used for scenario constructions that are alternative to the main scenario provided by the fusion approach.

It also seems inevitable that sensor fusion approaches require additional uncertainty measures to verify the quality of the developed methods and networks. A tool for semantic segmentation called MetaSeg that estimates prediction quality on segment level was introduced in [146] and extended in [112, 147, 153]. It learns to predict whether predicted components intersects with the ground truth or not, which can be viewed as meta classifying between two classes ($IoU = 0$ and $IoU > 0$). To this end, metrics are derived from the CNN's output and pass them on to another meta-classifier. This work of false positive detection was extended in [25] where the number of overlooked objects was reduced by only paying with a few additional false positives. The overproduction of false positives is suppressed by MetaSeg. Following these approaches for uncertainty quantification, we use metrics from the output of two CNNs. We pass them through to a gradient boosting classifier, which reduces the number of false positive predictions. In addition, by reducing the score threshold for object detection, we are able to improve over the performance of the respective single sensor networks.

In our tests, we utilize a YOLOv3 [142] as a state-of-the-art object detection network to process the camera data and complement this with a custom-designed CNN that performs a 1D binary segmentation which is supposed to detect obstacles. Further downstream of our computer vision pipeline we introduce a very general uncertainty-based fusion algorithm. Based on the predictions of both CNNs and their uncertainties as well as other geometrical meta-information, the fusion algorithm learns to provide a prediction by means of a structured dataset. In our experiments we demonstrate for the case of street scenes recorded at night, that this approach significantly improves the object detection accuracy. Furthermore, both networks only show moderate correlation which further supports our safety argument.

### 4.3.1 Characteristics

Today's vehicles are equipped with sensors for recording driving dynamics, which register movements of the vehicle in three axes, as well as sensors for detecting the environment. The latter try to map the vehicle environment as accurately as possible to promote automated driving. This section briefly describes the characteristics of the three most used sensors for the perception of the environment for automated driving, i.e., camera, radar and LiDAR, including their advantages and disadvantages. Camera sensors take two-dimensional images of light by electrical means. They are accurate in measuring edges, contour, texture and coloring. Furthermore, they are easily integrated into the design of a modern

vehicle. However, 3D localization from images is challenging and weather-related visual impairment can lead to higher uncertainties in object detections [170].

Radar sensors use radio waves to determine the range, angle and relative velocity of objects. Long-range radar sensors have a high range capability up to 200 − 250 m [41, 83, 152] and are cheaper than LiDAR sensors [4]. Compared to cameras, radar sensors are less affected by environmental conditions and pollution [4, 41, 57]. On the other hand, radar data is sparse and does not delineate the shape of the obstacles [57, 83]. LiDAR sensors use a light beam, emitted from a laser, to determine the distances and shapes of objects.

LiDAR sensors are highly accurate in 3D localization and surface measurements as well as a long-range view up to 300m [137]. They are expensive to buy and bad weather conditions like rain, fog or dust reduce the performance [4, 41].

Each sensor has its advantages and disadvantages (summarized in Table 4.3), so that a sensor fusion with at least two sensors makes sense in order to provide a better safety standard.

| specifications | | camera | radar | LiDAR |
|---|---|---|---|---|
| **distance** | range | ++ | +++ | +++ |
| | resolution | ++ | +++ | ++ |
| **angle** | range | +++ | ++ | +++ |
| | resolution | +++ | + | ++ |
| **classification** | velocity resolution | + | +++ | ++ |
| | object categorization | +++ | + | ++ |
| **environment** | night time | + | +++ | +++ |
| | rainy/cloudy weather | + | +++ | ++ |

+ = good, ++ = better, +++ = best

Table 4.3: Overview of the advantages and disadvantages of the most common sensors for autonomous driving [136].

### 4.3.2 Object Detection via Radar

In this section we introduce the 1D segmentation network that we equip for detecting vehicles. First we explain the pre-processing method, then we describe the network architecture, the loss function and the network output.

**Preprocessing** In a global 3D coordinate system, the radar data is situated in a 2D horizontal plane. Hence, before training a neural network, we pre-process the radar data for two reasons. First, in order to simplify the fusion after processing each sensor with a neural network separately, we project the given radar data into the same 2D perspective as given by the front view camera. Secondly, one

can observe that after this projection, the remaining section of the radar sensor modality is close to 1D. Figure 4.5 depicts radar points projected into the front view camera image. Darker colors indicate closer objects and brighter colors indicate more distant objects. Consequently, we build and train a neural network to perform a 1D segmentation.

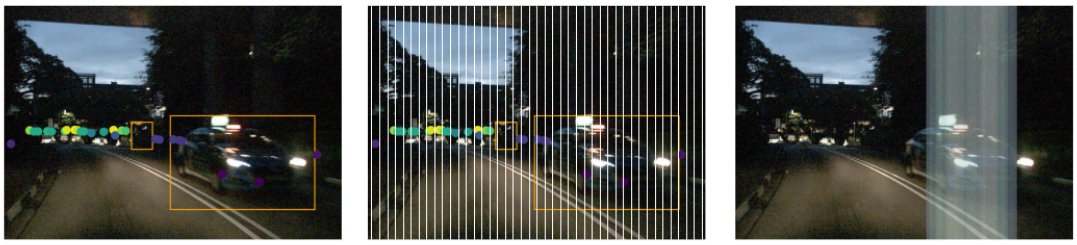


Figure 4.5: Preprocessing and prediction of the radar network. On the left we see the radar points projected to the front view camera image, the brighter the color value, the further away the points are. The center image is divided into a certain number of slices $N_s$ from which we generate the input matrix for the training of the CNN. The right-hand image corresponds to the output of the radar network.

To be more specific, we pre-process the ground truth for training the radar network as illustrated in Figure 4.6. That is, we divide the given front view image into a chosen number $N_s$ of slices and generate an occupancy array of length $N_s$. The $i$th entry of this array is equal to 1 if there is a ground truth object intersecting with the $i$th slice of the image and 0 else. This ground truth construction defines the desired prediction for the radar network.

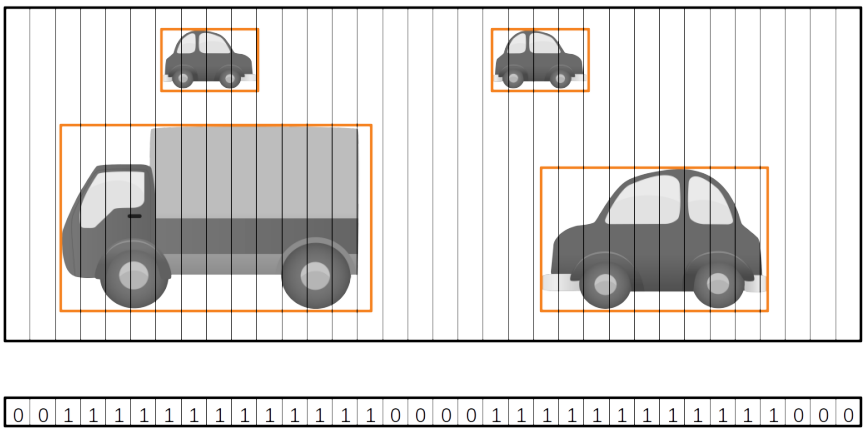



Figure 4.6: Ground truth vector for radar data. Every slice that overlaps with a bounding box in the front view camera image obtains the value 1, otherwise 0.

The radar data is pre-processed similarly to the ground truth as we aim at providing the neural network with an input tensor of size $N_s \times N_t \times N_f$ where $N_t$ denotes

the number of considered time steps and $N_f$ denotes the number of features. By assigning radar points to image slices we can drop the $x$-coordinate (which is implied by the array index up to an quantization error). For each slice $i = 1, \ldots, N_s$ which contains at least one radar point we store the following $N_f$ features in the input matrix: $y$-coordinates (indicating the distance of the reflection point), the height coordinate with respect to the front view image that the radar point obtains by projection into the image plane as well as the relative lateral and the longitudinal velocity.

**Network Architecture** The network architecture for the radar network is based on a FCN-8-network [109] and is depicted in Figure 4.7. The hidden layers consist of three convolution blocks, three deconvolution blocks followed by a concatenate layer, a fourth convolution block, a flatten layer and one fully-connected block. Finally, the network contains a sigmoid layer from which we get in $[0, 1]$. Each convolution and deconvolution block includes a (de-)convolution layer, a batch normalization and a leaky ReLU as activation function, respectively. The convolution blocks capture context information while losing spatial information whereas the deconvolution blocks restore this spatial information. Using bypasses, context information can be linked with spatial information. Furthermore, each fully connected block consists of a dense layer followed by a leaky ReLU activation function (except for the final layer where we use a sigmoid activation function).

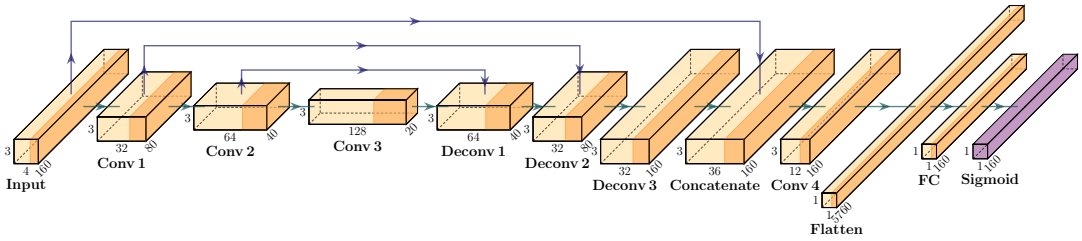



Figure 4.7: CNN architecture of our custom FCN-8 inspired radar network.

**Loss function** Let $D = \{(r^{(i)}, q^{(i)}) : i = 1, \ldots, n\}$ denote a dataset of tuples containing radar data $r^{(i)} \in \mathbb{R}^{N_s \times N_t \times N_f}$ and ground truth $q^{(i)} \in \{0, 1\}^{N_s}$. The radar network $g$ provides an array of estimated probabilities indicating whether a given slice $s$ is occupied or not. We denote $\varepsilon^{(i)} = g(r^{(i)})$. For training the neural network we use the binary cross-entropy for each array entry $s = 1, \ldots, N_s$, i.e., for a single data sample $(r^{(i)}, q^{(i)})$ we have

$$\ell(q_s^{(i)}, \varepsilon_s^{(i)}) = -\alpha q_s^{(i)} \log\left(\varepsilon_s^{(i)}\right) - (1 - q_s^{(i)}) \log\left(1 - \varepsilon_s^{(i)}\right) , \tag{4.1}$$

where $\alpha$ is a tunable parameter. We introduced this parameter in order to account for the imbalance of zeros and ones in the ground truth. When train-

ing with stochastic batch gradient descent, the loss function is summed over all $s = 1, \ldots, N_s$ and then the mean is computed over all indices $i$ in the batch.

**Output** The predictions of the radar network result in a vector consisting of values in $[0, 1]$ that estimate the probability of occupancy. Neighboring slices whose predicted probabilities are above a certain threshold $T_g$ are recognized as one coherent object, also called slice bundle. Figure 4.8 shows for example an image with four slice bundles. The more a slice bundle fills in a bounding box, the higher the 1D IoU gets.

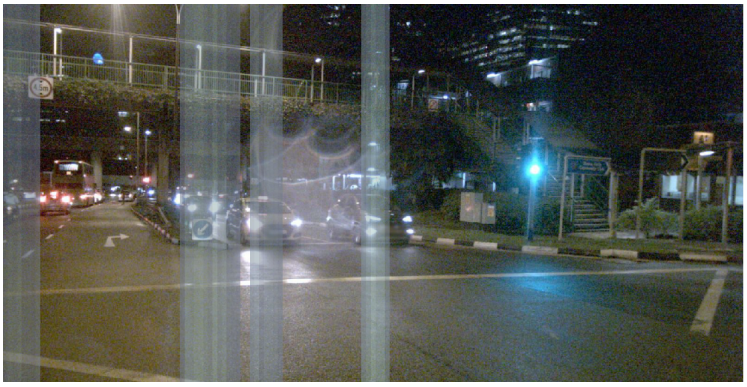


Figure 4.8: Image of a radar detection example with four predicted slice bundles.

### 4.3.3 Object Detection via Sensor Fusion

After describing the object detection method using radar sensor data in the previous section, this section deals with the image detection method and the fusion of both methods. Various object detection networks have been developed in recent years, whereby the YOLOv3 network has become a very good choice when fast and accurate real-time detection is desired [7, 142]. It has been observed that YOLOv3 works very well under good weather and visibility conditions but has problems with object recognition in bad visibility like hazy weather [99, 165] or darkness, like Xiao et al. have investigated for object detection with RFB-Net [175]. In our experiments, we focus on object detection by camera and radar at night. To this end, we first use each method separately in order to connect both outputs with gradient boosting [56], see Figure 4.9. Similarly to [146] we derive metrics from each CNN output and pass them through a gradient boosting classifier to increase the number of detected vehicles. The data and metrics used for the classifier are explained in Section 4.3.4. The YOLOv3 threshold $T_f$ for vehicle detection is set to a low value such that we get a higher number of bounding box predictions. From the many predicted bounding boxes, gradient

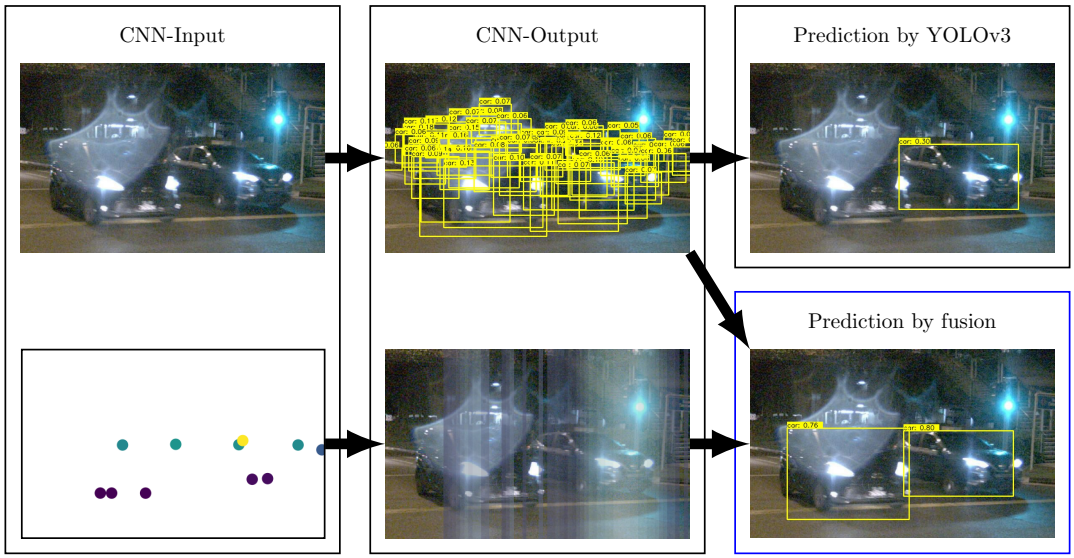



Figure 4.9: Illustration of our YOdar method. In the top branch, the YOLOv3 produces many candidate bounding boxes, but after score thresholding and non-maximum suppression, only one of both cars is detected. By lowering the object detection threshold and fusing the obtained boxes with the radar prediction, both cars are detected by YOdar as shown in the bottom right panel.

boosting select those boxes that are likely to contain an object according to the output of both networks. On the one hand, the radar sensor should detect vehicles not recognized by the YOLOv3 network. On the other hand, gradient boosting should support the decision-making process by additional information in case the YOLOv3 network is uncertain.

### 4.3.4 Fusion Metrics and Methods

The fusion method that we introduce in this section is of generic nature. Therefore, we denote by $f$ the arbitrary camera network and by $g$ a 1D segmentation radar network. Given an input scene $(x, r)$, we obtain two network outputs, one for the image input $x$, one for the radar input $r$. Each prediction obtained by $f$ consists of a set $B = \{b_1, \ldots, b_\kappa\}$ containing $\kappa$ boxes where $\kappa$ depends on $x$. Each box $b_i$ is identified with a tuple $\xi_i$ that contains an objectness score value $z_i$, a probability $f(v|x, b_i)$ that the box $b_i$ contains a vehicle $v$, a center point with its x-coordinate $c_i^x \in \mathbb{R}$ and y-coordinate $c_i^y \in \mathbb{R}$ as well as width $\varsigma_i \in \mathbb{R}$ and height $\vartheta_i \in \mathbb{R}$ of the box, i.e.,

$$\xi_i = (z_i, f(v|x, b_i), c_i^x, c_i^y, \varsigma_i, \vartheta_i)\,. \tag{4.2}$$

For the radar network $g$ we obtain a 1D output of probabilities, $g_s(v|r)$ for each of the slices $s = 1, \ldots, n$, that this slice $s$ belongs to a vehicle $v$, recall Figure 4.6. As depicted in Figure 4.5 we identify slices $s$ and bounding boxes $b_i$. In order to aggregate slices $s$ from the radar network over bounding boxes $b_i$ obtained by the camera network, let $S_i$ denote the set of all slices $s$ that intersect with the box $b_i$. We denote by

$$\mu_i = \frac{1}{|S_i|} \sum_{s \in S_i} g_s(v|r) \tag{4.3}$$

the average probability of observing a vehicle in the box $b_i$ according to the radar network's probabilities. The standard deviation corresponding to Equation (4.3) is termed $\sigma_i$. As a set of metrics, by which we compute a fused prediction, we consider

$$M_i(x, r) = (\xi_i, A_i, \mu_i, \sigma_i) \ , \tag{4.4}$$

where $A_i = \varsigma_i \cdot \vartheta_i$ denotes the size of the box $b_i$. In summary, we use these nine metrics $M_i(x, r)$ for all scenes $(x, r)$ and boxes $b_i$ that are visible with respect to the front view camera.

To perform the fusion of the camera based network prediction and the radar based network prediction we proceed in two steps. First we compute the ground truth which states for each box predicted by the camera network whether it is a true positive (TP) or a false positive (FP). Afterwards we train a model to discriminate by means of $M_i$ whether $b_i$ is a TP or an FP.

More precisely, for the sake of computing ground truth, we define TP and FP in the given context as follows: For a predicted box $b_i$ and a ground truth box $q_a$, which has the biggest intersection $|q_a \cap b_i|$ of all ground truth boxes of the same class, the intersection over union is defined as follows:

$$IoU(b_i) = \max_{q_a} \frac{|q_a \cap b_i|}{|q_a \cup b_i|} \, . \tag{4.5}$$

Oftentimes we omit the argument $b_i$ if it is clear from the context. Given a chosen threshold $\Gamma \in [0, 1)$ we define that $b_i$ is a TP if $IoU(b_i) > \Gamma$ and an FP if $IoU(b_i) \leq \Gamma$. For the sake of completeness, we define that a false negative is a ground truth box $a$ that fulfills $\max_{b_i} \frac{|q_a \cap b_i|}{|q_a \cup b_i|} \leq \Gamma$. Note that in the latter definition, the ground truth element is fixed while the left-hand side of this expression is maximized over all predicted boxes. After computing the ground truths, i.e., whether a box $b_i$ predicted by the camera network yields a TP or an FP, we train a model. The gathered metrics $M_i$ yield a structured dataset where the columns are given by the different metrics and the rows are given by all predicted boxes $b_i$ for each input scene $(x, r)$. By means of this dataset and the corresponding TP/FP annotation, we train a classifier to predict whether a box predicted by the camera network is a TP or an FP.

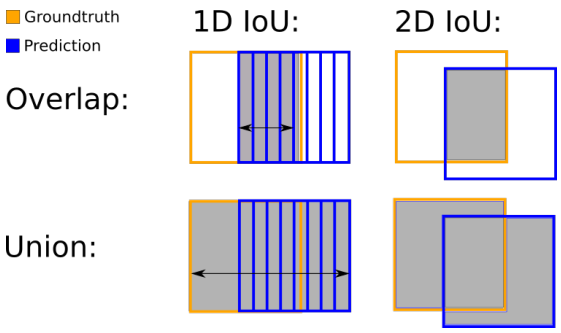



Figure 4.10: IoU calculation for 1D and 2D bounding boxes.

The IoU can be calculated in different dimensions $D = 1, 2$. In this work the 1D IoU is used for the radar network. As soon as a low threshold $T_g$ has been reached, a predicted object is considered as TP, otherwise as FP. The 2D IoU is used for the YOLOv3 network, analogously we speak of TP and FP according to a threshold $T_f$. An illustration is given in Figure 4.10. The mean average precision (mAP) is a popular metric used to measure the performance of models. The mAP is calculated by taking the average precision (area under precision as a function of recall, a.k.a. precision recall curve) over one class.

### 4.3.5 Numerical Experiments

As explained in detail in the previous section, we use a custom FCN-8-like network for processing the radar data from the nuScenes dataset [22]. It contains urban driving situations in Boston and Singapore. The dataset has a high variability of scenes, i.e., different locations, weather conditions, daytime, recorded with left- or right-hand traffic. In total, the dataset contains 1,000 scenes, each of 20 seconds duration and each frame is fully annotated with 3D bounding boxes. The vehicle used, a Renault Zoe, was set up with 6 cameras at 12 Hz capture frequency, 5 long-range radar sensors (FMCW) with 13 Hz capture frequency, 1 spinning lidar with 20 Hz capture frequency, 1 global positioning system module (GPS) and 1 inertial measurement unit (IMU). Each scene is divided into several time frames for which each sensor provides a suitable signal.

For our experiments, the YOLOv3 network was pretrained with day images from the COCO dataset [103] and afterwards with 249 randomly selected scenes from the nuScenes dataset containing 10,000 images with different weather conditions and times of day. Furthermore, we have trained a CNN with radar data from the nuScenes dataset to detect vehicles. 743 of the scenes (29,853 frames) were used as training data, 82 scenes (3,289 frames) as validation data and 25 scenes (1,006 frames) as test data, see Table 4.4. For the test set, we consider exclusively all frames recorded at night that are not part of the training data in order to create

a perception-wise challenging test situation. For the training and validation sets we used a natural split of day and night scenes pre-defined by the frequencies in the nuScenes dataset. Due to the resolution of the radar data, we focus on the category vehicle in our evaluation. This includes the semantic categories car, bus, truck, bicycle, motorcycle and construction vehicle.

| **splitting** | **number of images/frames** | | **night images/frames [%]** | |
|---|---|---|---|---|
| | YOLOv3 | radar | YOLOv3 | radar |
| train | 10,000 | 29,853 | 7.08 | 9.04 |
| val | 3,289 | 3,289 | 8.57 | 8.57 |
| test | 1,006 | 1,006 | 100 | 100 |

Table 4.4: Data used for training, validation and testing of the radar and YOLOv3 model.

**Training** As input, the radar network obtains a tensor with the dimensions $160 \times 3 \times 4$ that contains for each of the 160 considered slices the current frame (i.e., the current time step) and two previous frames. Each frame contains four features, i.e., x-, and y-coordinates, lateral and longitudinal velocity. From the radar data we removed all ground truth bounding boxes that do not contain any radar points with valid velocity vectors.

The radar network is implemented in Keras [30] with TensorFlow [1] backend. Training on one NVIDIA Quatro GPU P6000 takes 229 seconds training time. The network structure is shown in Figure 4.7 and the training parameters are shown in Table 4.5. We have trained the neural network three times with three different learning rates, i.e., the first 20 epochs with a learning rate of $10^{-3}$, 10 epochs with $10^{-4}$ and 10 epochs with $10^{-5}$. The networks output vector has the same dimension ($160 \times 1$) as the ground truth vector, where each entry contains a probability value, whether there is a vehicle in the respective area or not. If the probability value of a single slice is equal or higher than the threshold $T_g = 0.5$, then the network predicts a vehicle. The higher the probability value, the brighter the slice is displayed in Figure 4.5.

| **parameter** | **radar** | | | **YOLOv3** |
|---|---|---|---|---|
| batchsize | 128 | | | 6 |
| learning rate | $10^{-3}$ | $10^{-4}$ | $10^{-5}$ | $10^{-4} - 10^{-6}$ |
| epochs | 20 | 10 | 10 | 100 |
| weight decay | $3 \times 10^{-4}$ | | | variable |
| loss function | modified binary-crossentropy | | | binary-crossentropy |
| optimizer | Adam | | | Adam |

Table 4.5: Training parameters.

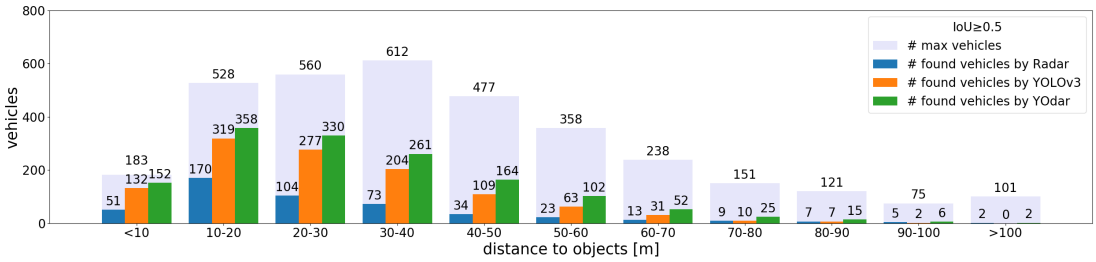



Figure 4.11: Vehicles recognized at night with consideration of the respective distance from the ego car. An object is considered as detected when it has an $IoU \geq 0.5$ with the ground truth. These are average values from three test runs.

The YOLOv3 network was trained with 10,000 images consisting of different scenes. Therefore, we converted the 3D bounding boxes in the nuScenes dataset into 2D bounding ones. To this end, we chose the smallest 2D bounding box that contains the front and rear surfaces of the 3D bounding box in the given ego car view. YOLOv3 is implemented in the Python framework TensorFlow [1]. Training with the same GPU as used for the custom radar network takes 50.32 hours training time. The training parameters are stated in Table 4.5.

**Evaluation** All results in this section are averaged over 3 experiments to obtain a better statistical validity. Figure 4.11 shows absolute numbers of recognized objects ($IoU \geq 0.5$ with the ground truth) at night for each of the networks (radar and YOLOv3) standalone as well as for our uncertainty-based fusion approach (YOdar). The numbers are broken down corresponding to distance intervals along the horizontal axis. The lavender bar (in the background) displays the numbers of vehicles in the ground truth for the given distance interval. The blue bar states the numbers of objects recognized by the radar network. The performance of the radar network is low, only a small percentage of objects are found. After 20 meters, the performance decreases with growing distance. The poor recognition of objects from radar can be explained by the small number of points provided for each frame. In addition, relative velocities are used for training, which means that mainly moving objects can be recognized and stationary or parked vehicles remain undetected. The orange bar shows a significant increase of objects recognized by the YOLOv3 network compared to the radar network. Although mainly closer objects are recognized, there remain difficulties in object recognition with more distant objects. The green bar shows the objects recognized by YOdar. Compared to the YOLOv3 network, more vehicles are recognized for each of the given distances. In total, compared to the YOLOv3 network, the sensor fusion approach recognizes 313 vehicles more (which amounts to an increase of 9.20 percent points).

So far, we have seen that we recognize more objects with the YOdar approach

than with YOLOv3 or our radar network separately. However, the increased sensitivity also yields some additional FPs. In order to compare the number of FPs for YOLOv3 and YOdar, we adjust the sensitivity of the YOLOv3 network by lowering the threshold $T_f$ such that the TP level for YOLOv3 is roughly equal to the TP level of YOdar. The resulting number of FPs is given in Table 4.6. Indeed, YOdar generates 575 less FP predictions than YOLOv3 for a common TP level, on which we let YOdar operate in our tests.

| **network** | **unchanged output** | | **TP level adjustment** | |
|---|---|---|---|---|
| | TP | FP | TP | FP |
| YOLOv3 | 1,154 | 98 | 1,478 | 1,024 |
| YOdar | 1,467 | 449 | 1,467 | 449 |

Table 4.6: Comparison of the number of false positives for YOLOv3 and YOdar at a common level of false positives.

Digging deeper into the discussed results, we now break down the distance intervals along the distance radii. Figure 4.12 states the absolute numbers of objects broken down by distance (vertical axis) and the pixel intervals of width 100 of the input image with a total width of 1600 pixels (horizontal axis). More precisely, each interval denoted by $i$ on the horizontal axis represents the pixels $(row, column)$ with $column \in [i - 99, i]$. A ground truth object is a member of such an interval, if the center of the box is contained in the respective interval and has the respective distance from the ego car. Thus, this can be viewed as a spatial distribution of the ground truth where the center of the bottom row is the area closest to the ego car. The majority of the objects is located in the intervals given by $i = 500, \ldots, 1200$.

Figure 4.13 shows the relative amount of objects recognized by the YOLOv3 network. It shows that mainly objects closer to the ego car and straight ahead are recognized, while objects farther away or located on the very left or very right end of the image often remain unrecognized. Figure 4.14 states in absolute numbers how many additional objects are recognized in each particular area when using YOdar instead of YOLOv3. The increase is clear and also mostly in the relevant areas close to the ego car and straight ahead. This is in line with the idea of focusing with the radar on objects in motion (by considering objects that carry velocities). These results show that an uncertainty-based fusion approach like YOdar is indeed able to increase the performance significantly. This finding is also confirmed by the mAP and accuracy values stated in Table 4.7. While YOLOv3 achieves 31.36% mAP, YOdar achieves 39.40% which is also close to state of the art deep learning based fusion results for the nuScenes dataset with the natural split of day and night scenes as reported in [123].

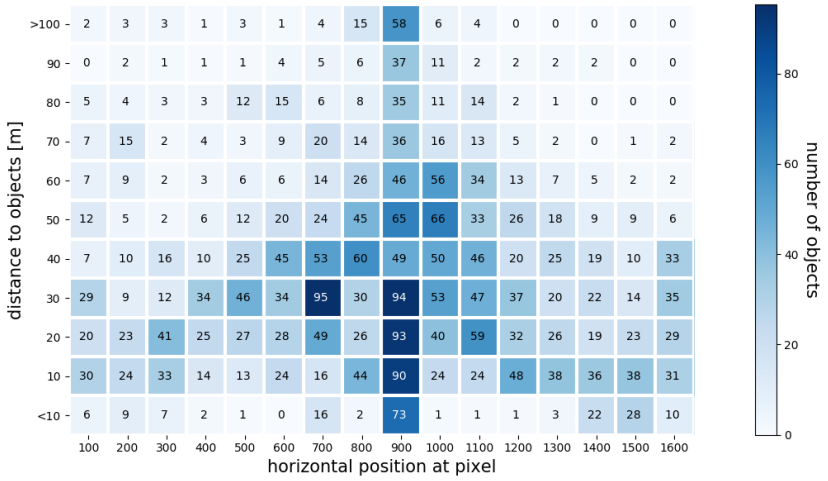



Figure 4.12: Ground truth heatmap displaying the spatial distribution of the test data, broken down by distance (vertical axis) and the pixel intervals of width 100 of the front view input image with a total width of 1600 pixels (horizontal axis). More precisely, each interval denoted by $i$ on the horizontal axis represents the pixels $(row, column)$ with $column \in [i - 99, i]$.

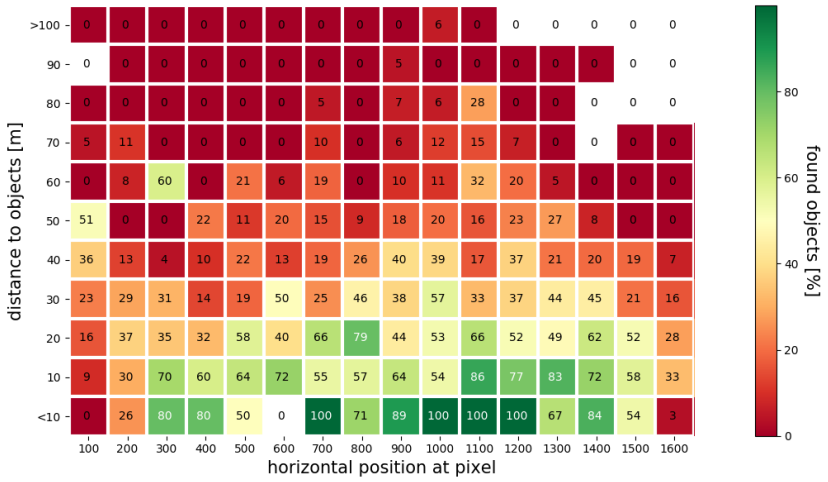



Figure 4.13: Relative amount of objects recognized by the YOLOv3 network evaluated on the test data. The underlying geometry is the same as in Figure 4.12.

| | **radar** | **YOLOv3** | **YOdar** |
|---|---|---|---|
| accuracy [%] | 14.42 | 33.90 | 43.10 |
| mAP [%] | 7.93 | 31.36 | 39.40 |

Table 4.7: Accuracies and mAP scores of all three networks.

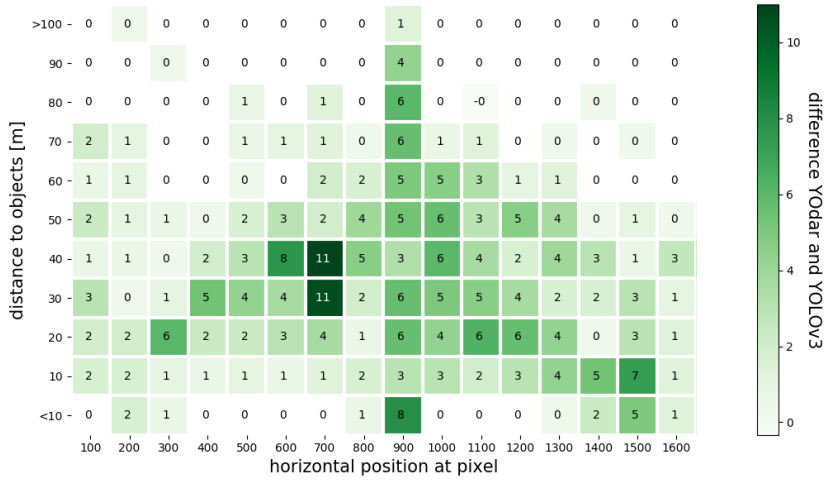



Figure 4.14: Number of objects recognized by YOdar minus the number of objects recognized YOLOv3. The underlying geometry is the same as in Figure 4.12.

### 4.3.6 Conclusion

We introduced the uncertainty-based sensor fusion approach YOdar, which first processes camera and radar data individually before post-processing their results through a gradient boosting method to provide a joint prediction for both networks. Each branch detects objects, the camera branch uses a YOLOv3 network trained with day and night scenes and the radar branch uses a custom-based radar network. The outputs of every branch are aggregated and then passed through a post-processing classifier that again learns the same vehicle detection task. Compared to the YOLOv3 network, the YOdar fusion method detects at night a significant additional amount of vehicles in total. While YOLOv3 achieves 31.36% mAP, YOdar achieves 39.40% mAP which is also close to state of the art deep learning based fusion results for the nuScenes dataset with the natural split of day and night scenes.

With the YOdar approach we could show that the use of two different sensors, camera and radar, provides an improvement in object detection and thus is also more robust against different corner case types. As soon as more radar data is available, further investigations can be performed to support these results. In addition, it should be investigated whether the use of a third sensor, for example LiDAR, makes sense and/or whether two sensors are sufficient to increase safety to such an extent that the cost-benefit calculation does not work out with three sensors.

## 4.4 Datasets for Tracking and Retrieval of Out-of-Distribution Objects [111]

In autonomous driving, it is very important to detect objects that are outside the semantic space of the network, so-called Out-of-Distribution objects (OoD). As described in the taxonomy for corner cases, OoD objects are corner cases at the object level, which need to be identified. This can be performed at pixel-level which is commonly known as OoD segmentation, see also [10, 13, 21, 23, 24, 63, 64, 104, 105].

If the task of OoD segmentation is to be modeled in reality, algorithms should be available that detect OoD objects in video sequences and track them over time to forecast critical situations more accurately. Therefore, we present the novel task of OoD tracking as a combined hybrid computer vision task consisting of OoD detection, OoD segmentation, and object tracking and additionally provide a way to use tracking for retrieving OoD objects of the same class. This is first done by identifying an OoD object from the first frame in which it occurs and assigning a unique ID across multiple frames. In order to be able to achieve the task of tracking, suitable datasets are needed that contain video sequences with OoD objects. For this reason, we provide two annotated datasets, one recorded in the real world, named *Street Obstacle Sequences (SOS)*, and one in a synthetic environment, named *CARLA-WildLife (CWL)*. In addition, we provide a third unannotated dataset, named *Wuppertal Obstacle Sequences (WOS)*, to serve as an application. Some examples of the datasets are shown in Figure 4.15.

Figure 4.15: Some examples of the SOS (top), CWL (middle) and WOS (bottom) datasets.

Figure 4.16: Collection of objects in CARLA-WildLife dataset.

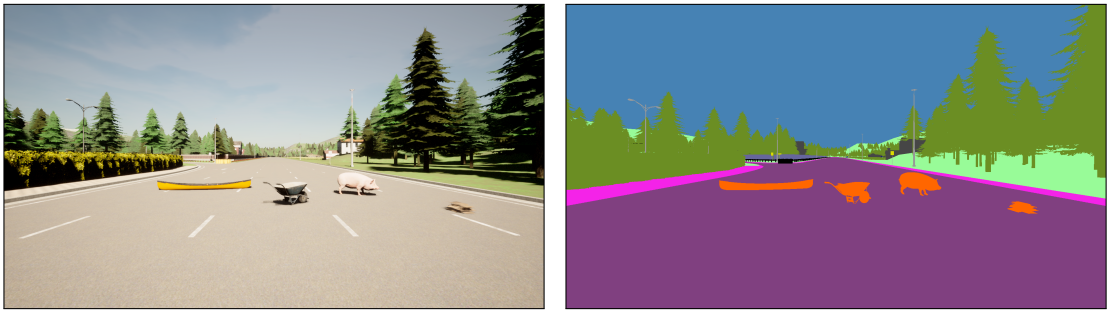
### 4.4.1 Generation of the CARLA-WildLife Dataset

Since the recording and labeling of real data is time-consuming, as witnessed for the SOS dataset, and the selection of diverse real-world OoD objects is limited in practice, we additionally introduce a synthetic dataset for OoD detection offering a large variety of OoD object types. The main advantage of synthetic data is that they can be produced inexpensively with accurate pixel-wise labels of full scenes, besides being able to manipulate the scenes as desired. For OoD segmentation we define the street as the region of interest, while everything outside this area is defined as *void* and does not appear in the evaluation part. The area of the street, on the other hand, is classified binary between street and no street, i.e., OoD.

By adding free of cost available assets from Unreal Engine marketplace [160] to the driving simulation software CARLA [44], we generate 26 synthetic video sequences recorded at a rate of 10 frames per second with 18 different object types placed on the streets of CARLA. During the sequences, the ego vehicle moves towards these objects, including, for example, dogs, wolves, balls, canoes, pylons, or bags. See Figure 4.16 for an overview of all objects placed in CWL. The objects were selected based on whether they could cause hazardous street scenarios. Since these objects are not included in the standard set of semantic labels provided by CARLA, each object type is added as extra class retroactively.

In addition to the semantic segmentation based on the Cityscapes labeling policy

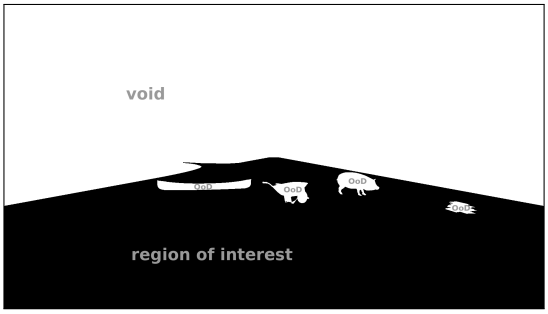
(and including the OoD class), CWL further provides instance segmentation, i.e., individual OoD objects of the same class can be distinguished within each frame, and tracking information, i.e., the same object instance can be identified over the course of video frames. Moreover, we provide pixel-wise distance information for each frame of entire sequences as well as aggregated depth information per OoD object depicting the shortest distance to the ego-vehicle, see Figure 4.17.



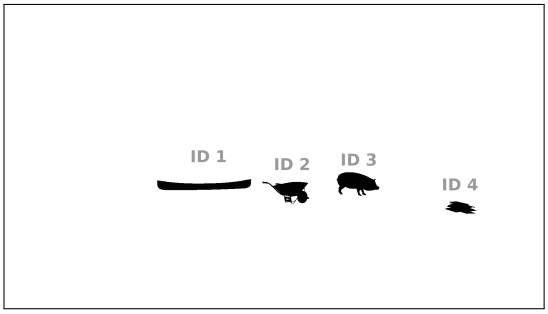
(a) original



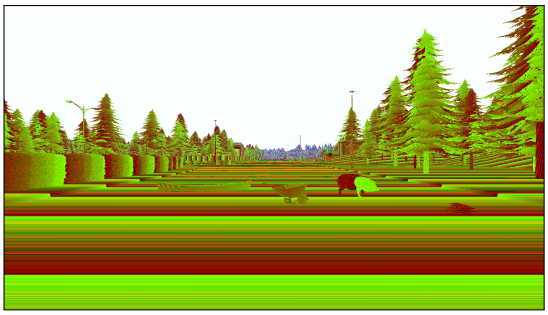
(b) semantic




(c) semantic OoD




(d) instance OoD



(e) depth raw

Figure 4.17: Available data for each sequence in CWL. Besides the original rgb image, we provide the semantic mask according to the Cityscapes classes and add new ones to them corresponding to the object type class. Furthermore, we provide OoD masks for semantic, instance and depth. (e) reveals the raw depth mask we extract from CARLA. Each pixel has a distance value on the logarithmic scale and must first be converted. This conversion is also done for CWL on 16 bit and is also provided.

**Adding assets into CARLA** The visibility of objects in Unreal Engine depend on their surface, which is determined by their geometry (created through meshes or brush surfaces), and their material (defined by texture and material parameters). Meshes define the actual geometry of an object/actor which creates the surface. UE distinguishes between 3 different mesh types: static, dynamic and skeletal.

Materials define surface properties of objects such as color, reflectivity, unevenness, transparency, etc. They dictate the engine exactly how a surface should interact with light. Components of material are textures and several material parameters. If the light behavior of the real world needs to be simulated, physically based materials are required. Therefore, material attributes such as base color, roughness, metallic and specular have to be used. Textures are image files that are mapped to surfaces which the material is already applied to. They provide pixel-wise information about color, gloss, transparency and some other aspects.

A static mesh is a geometry that consists of a bunch of polygons which are cached and rendered by the graphics card. They are 3D models created in third-party modeling applications like Blender[7] or Autodesk 3ds Max[8] and can be imported into the Unreal Editor via the content browser. They are stored in packages as **.fbx*-file and can be used in several ways to create renderable elements.

In order to add new assets to the Unreal Engine and thus to the CARLA world, it is necessary to have ready-made **.fbx*-files that contain geometry, textures as well as material data. Once an object's **.fbx*-file has been imported into the Unreal Engine, it is available as an object and can be dragged and dropped into a CARLA map. Besides the location coordinates, the size, color and some other parameters of an object can also be changed, like shown in Figure 4.18.

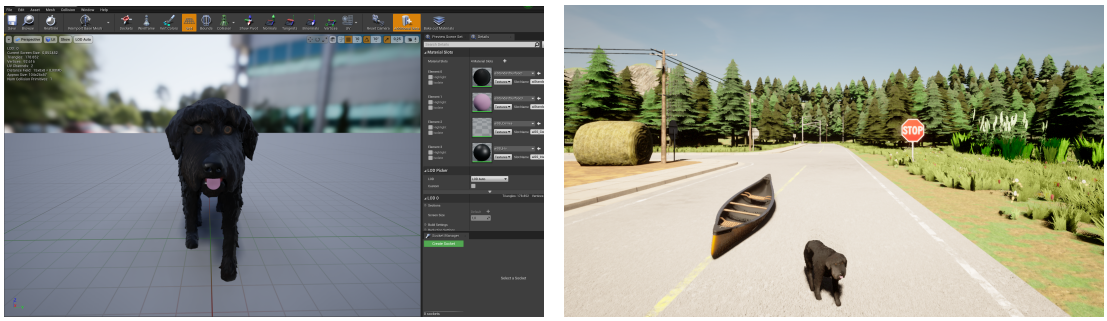


Figure 4.18: The model view of an asset in Unreal Editor (left), placed in the CARLA world and appearing in the CWL dataset (right).

[7]https://www.blender.org/ accessed 2022.11.17
[8]https://www.autodesk.com/products/3ds-max/ accessed 2022.11.17

CARLA labels all newly added assets as unknown, which is why the ID for unknown classes (0) in CARLA is stored for the semantic segmentation ID. Several attempts to import those objects into CARLA as a new class failed, so we had to spend more time in post-processing to give each asset a unique class ID. Some assets have additional animations that were exhibited when the dataset was created. To do this, each object placed on the road had to be unchecked under *Play* on the *Animation* tab. If, in addition to the new objects, the user wants to place other traffic participants, such as vehicles, that can move freely, the *PhysicsActor* preset under the *Collision* tab must be set to *NoCollision*. Otherwise, these objects will represent a static object that cannot be overtaken, and the traffic behind it will jam.

### 4.4.2 Method

An overview of our method can be found in Figure 4.19. First, we create the region of interest and an entropy heatmap from an input image. From these two pieces of information, we obtain the OoD segmentation after the method introduced in [24], which is also based on an evaluation of the prediction quality using meta classification [147]. This is done for each frame, so that the tracking algorithm, introduced in [112], then assigns a unique ID to a newly found object in a sequence that ideally spans multiple frames. An identical object is found under the condition that either center points are close to each other or if the overlapping of segmentation in successive frames is sufficiently large. In parallel, the retrieval algorithm, adapted from [128, 166], clusters similar objects from the OoD segmentation into the 2D embedding space, which thus form new classes. False predicted OoD objects are sorted out based on the tracking information by requiring a minimum number of frames with the same ID.

## 4.5 Datasets for Anomaly Detection [17]

As described in the previous subsection, corner cases can be divided into 4 layers, where the content layer can be further subdivided into 3 levels. In addition to the domain and scene level, there is also the object level, which contains anomalies and unknown objects on and next to the road. Especially for object detection or semantic segmentation algorithms it is challenging to deal with such objects, as they have to gather their experience from the data provided during training and have to map new, unknown classes onto previously defined classes. This often leads to the networks becoming uncertain in the areas of anomaly and dividing them into different areas, using semantic segmentation networks as an example, resulting in undefinable color blobs. For this reason, anomaly or unknown object

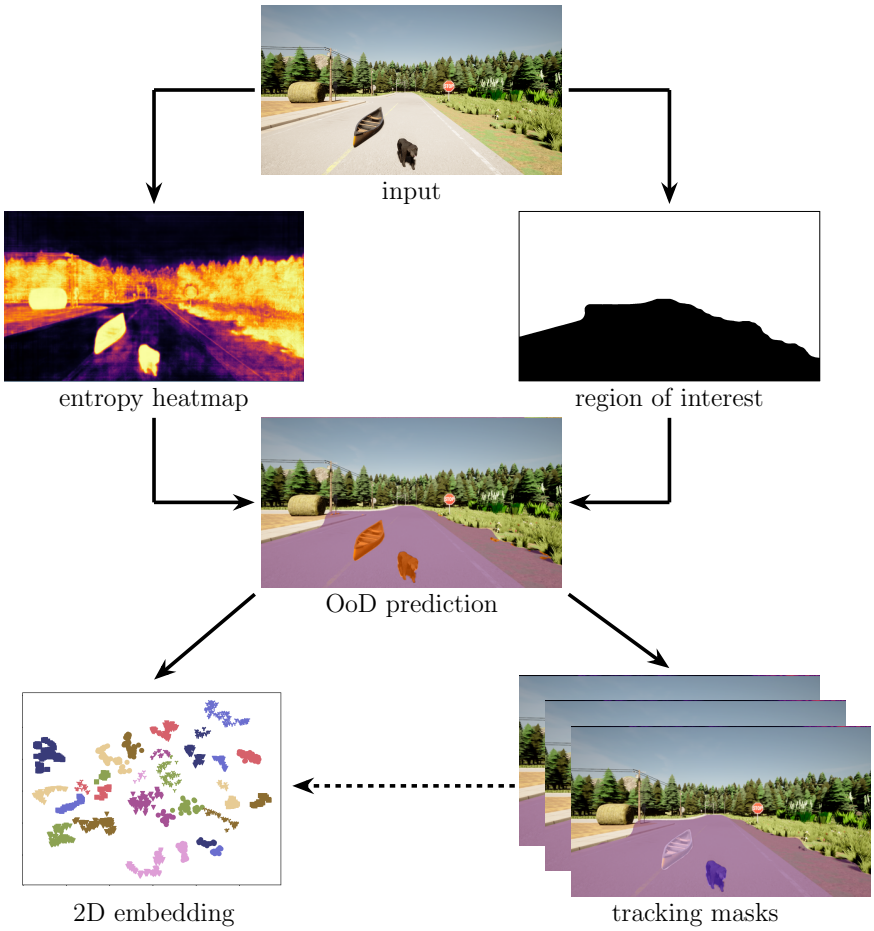



Figure 4.19: Overview of our OoD tracking and retrieval method. The OoD segmentation is done using the region of interest and an entropy heatmap. Afterwards, each newly found OoD object is assigned its own ID, which can be used to track it across multiple frames. Additionally, similar objects in the 2D embedding space are grouped as separate classes, so that the same objects from the sequences of each frame end up in the same area.

detection is a very active research area [14, 16, 26, 46, 166] to move one step closer to autonomous driving. So a joint work to an anomaly survey paper was done, that lists all the publicly available datasets (as the end of January 2023) that will hopefully bring humanity closer to the future vision of safe autonomous driving.

Our criteria to be included in the list were as follows. The dataset must be publicly available, contain sensor data from the ego-perspective and provide pixel- or point-wise anomaly labels in the form of a validation set. After screening a lot of papers, we found a total of 16 that met these criteria which are clustered by their benchmark, see Table 4.8.

In addition to publication date and sensor type, we provide information about the

| dataset | year | sensors | size (test/val) | resolution | anomaly source | temporal | #OoD classes | ground truth |
|---|---|---|---|---|---|---|---|---|
| **Fishyscapes** [12, 13] | | | | | | | | |
| FS Lost and Found | 2019 | camera | 275 / 100 | 2048 × 1024 | recording | no | 1 | semantic mask |
| FS Static | 2019 | camera | 1,000 / 30 | 2048 × 1024 | data augmentation | no | 1 | semantic mask |
| **CAOS** [74] | | | | | | | | |
| StreetHazards | 2019 | camera | 1,500 | 1280 × 720 | simulation | yes | 1 | semantic mask |
| BDD-Anomaly | 2019 | camera | 810 | 1280 × 720 | class exclusion | no | 3 | semantic mask |
| **SegmentMeIfYouCan** [23] | | | | | | | | |
| RoadAnomaly21 | 2021 | camera | 100 / 10 | 2048 × 1024<br>1280 × 720 | web sourcing | no | 1 | semantic mask |
| RoadObstacle21 | 2021 | camera | 327 (+55) / 30 | 1920 × 1080 | recording | yes | 1 | semantic mask |
| **CODA** [100] | | | | | | | | |
| CODA-KITTI | 2022 | camera, LiDAR | 309 | 1242 × 1376 | void classes | no | 6 | bounding boxes |
| CODA-nuScenes | 2022 | camera, LiDAR | 134 | 1600 × 900 | void classes | no | 17 | bounding boxes |
| CODA-ONCE | 2022 | camera, LiDAR | 1,057 | 1920 × 1020 | aut. OoD proposal | no | 32 | bounding boxes |
| CODA2022-ONCE | 2022 | camera, LiDAR | 717 | 1355 × 720 | aut. OoD proposal | no | 29 | bounding boxes |
| CODA2022-SODA10M | 2022 | camera | 4,167 | 1280 × 720<br>958 × 720 | aut. OoD proposal | no | 29 | bounding boxes |
| **Wuppertal OoD Tracking** [111] | | | | | | | | |
| Street Obstacle Sequences (SOS) | 2022 | camera, depth | 1,129 | 1920 × 1080 | recording | yes | 13 | instance mask |
| CARLA-WildLife (CWL) | 2022 | camera, depth | 1,210 | 1920 × 1080 | simulation | yes | 18 | instance mask |
| **Misc** | | | | | | | | |
| Lost and Found [138] | 2016 | stereo cameras | 2,104 | 2048 × 1024 | recording | yes | 42 | semantic mask |
| WD-Pascal [9] | 2019 | camera | 70 | 1920 × 1080 | data augmentation | no | 1 | semantic mask |
| Vistas-NP [64] | 2020 | camera | 11,167 | varying | class exclusion | no | 4 | semantic mask |

Table 4.8: Overview over all analyzed datasets, clustered by the benchmark in which they were presented.

size of the dataset, including resolution, whether the images are video sequences or incoherent scenes, how many Out-of-Distribution (OoD) objects are included, and in what form the labels are available. In addition, we provide each dataset sorted by one of the following anomaly sources:

- **Automated OoD Proposal**
  This approach allows the use of large unlabeled datasets. An automated proposal method is used to generate initial anomaly proposals. This can be done using various anomaly detection approaches, such as uncertainty, intermediate detections, geometric priorities, or model inconsistencies. Subsequently, human experts take care of false positives and refine the proposal.

- **Data Augmentation**
  For this technique, any dataset can be used as a baseline. By synthetic manipulation of scenes [58, 90], anomalies are pasted onto the original image and can be labeled accordingly. Anomalies are typically not included in the Cityscapes classes.

- **Recording and Simulation**
  Here, anomalies are recorded through data collection by driving in the real world [111, 138] or in the synthetic world [15, 74, 91]. Currently, these are available as static objects, although dynamic OoD objects are also available with WOS. Often, anomalies are also not included in the Cityscapes classes.

- **Class Exclusion**
  This approach is based on a labeled dataset. Hypothetical anomalies are

created by excluding frames with known classes from the train and validation splits. A novel test split is created with these, treating the selected classes as anomalies.

- **Web Sourcing**
  In this approach, human experts actively search for images that include atypical classes. As a reference list for known classes, often Cityscapes classes are used.

- **Misc Classes**
  Based on a labeled dataset, all regions which are either labeled with *void* or *misc* can be examined further. These terms are often used interchangeably and mostly refer to uncommon objects or irrelevant areas. Human experts then relabel those classes as anomalies, if appropriate.

Chapter 5

# Applications for Driving Simulator

This chapter presents some applications of the self-designed driving simulator in a research environment. Firstly, an approach is described that uses two human drivers to find corner cases in real time during a driving campaign, in order to save them for further training. Subsequently, in another driving campaign, it is investigated whether driving with a model trained with corner cases lasts longer than driving with models without corner cases. Based on this, a survival analysis is carried out to investigate, in a third driving campaign, the probability of survival under different weather conditions. Expert models of one domain will be compared with a universal one. Finally, the corner cases generated during all driving campaigns will be classified into a corner case trajectory taxonomy.

## 5.1 A-Eye: Driving with the Eyes of AI for Corner Case Generation [91]

Despite AI systems achieve impressive performance in solving specific tasks, e.g. in automated driving, they lack understanding of the context of safety in traffic. In contrast, while humans are often described as lousy drivers, as they tend to be diverted or feel fatigue, humans have a fine understanding, when a traffic scene could lead to a situation, where humans are at risk.

It has been observed previously, that to increase robustness and performance of AI algorithms many clean and diverse scenes are needed [84]. However, a large amount of annotated data per se might not imply safe operation in those rare situations, where road users are exposed to a substantial risk. This is why we aim to present an accelerated testing strategy that leverages human risk perception to capture corner cases of the *Method Layer* and thereby achieve performance

improvement in safety-critical scenes. In order to obtain many safety-critical corner cases in a short time, we stop training at an early stage so that the network is sufficiently well-trained. Nevertheless, the scenes generated in this way are still useful to improve fully trained networks.

For this purpose, a semantic network is trained with synthetic images from the open source driving simulation software CARLA [44]. In addition, the driving simulator described in Chapter 3 is modified to create a test rig with two control units to steer the ego-vehicle with two human drivers. In this process, the semantic segmentation network Fast-SCNN is integrated into CARLA in such a way that first the original CARLA image is sent through the network and the prediction is displayed on the screen of one driver. The second driver, in turn, sees the real CARLA image and is supposed to intervene as a safety driver only if he or she feels that a situation is being wrongly assessed by the other driver. We aim to consider situations in which the AI algorithms lead to incorrect evaluations of the scene which we refer to as safety-relevant corner cases (*Method Layer*), in order to improve performance through targeted data enrichment. This is done by exchanging images from the original dataset with the safety-critical corner cases, thus keeping the total amount of data fixed. We show that the semantic segmentation network that contains safety-critical corner cases in the training data performs better on similar critical situations than the network that does not contain any safety-critical situations.

Our approach somehow follows the idea of active learning, where we get feedback on the quality of the prediction by interactively querying the scene. However, unlike in standard active learning we do not leave the query strategy to the learning algorithm, but make use of the human's fine-tuned sense of risk to query safety-relevant scenes from a large amount of street scenes, leading to enhanced performance in safety-critical situations.

The contributions of this work can be summarized as follows:

- An experimental setup, that could also be implemented in the real world, permits testing the safety of the AI perception separately from the full system safety including the driving policy of an automated vehicle.
- A proof of concept for the retrieval of training data for automated driving with a human-in-the-loop approach that is safety-relevant.
- A proof that training on safety-relevant situations generated during poor network performance is beneficial for the recognition of street hazards.

**Human-Centered AI** According to the Defense Advanced Research Project Agency (DARPA), the development of AI systems is divided into 3 waves [42, 75]. While in the first wave patterns were recognized by humans and linked into logical

relationships (crafted knowledge), statistical learning could be used in the second wave due to improved computing power and increased memory capacity.

We are currently in the third wave, which relates to the explainability and contextual understanding of AI. Here, the black-box approaches, which emerged in the second wave, are to be understood so that AI decision-making becomes comprehensible to humans. This last point is therefore also an important one in the HCAI initiative, which aims to connect different domains with human-centered AI [98].

On this basis, the authors of [177, 178] developed an extended HCAI framework that defines the following three different design goals for the human-centered AI: *Ethically Aligned Design* that avoid biases or unfairness of AI algorithms such that these algorithms make decisions according to human criteria and rules. *Technology Design* which considers human and machine intelligence to exploit synergies. *Human Factors Design* to make AI-solutions explainable. The aim is to give humans an insight into the decision-making process of AI algorithms, so that trust in the current technology can be increased. For this purpose, we have built a specially developed test rig to incorporate human behavior into the further development of AI in the field of autonomous driving. At the same time, we would like to use the test rig to provide a demonstration object to illustrate AI algorithms to society. The focus will be on allowing humans to visually perceive and interact with the decision-making of AI algorithms. In the context of autonomous driving this would mean:

***Driving with the eyes of AI.***

Human behaviors can also be analyzed using driving simulators. For example, in [45] a realistic test rig including a steering wheel and pedals for data collection was developed. Therefore, thirteen subjects were recruited to drive on different routes while being distracted by static or dynamic objects or by answering messages on their cell phones. By adding nonlinear human behaviors and using realistic driving data, the authors have been able to predict human driving behavior more accurately in testing. Another driving simulator was presented in [66] to develop and evaluate safety and emergency systems. The control units are connected to a generic simulator for academic robotics which uses the Modular Open Robots Simulation Engine *MORSE* [50]. They used an experiment with four road users, one human driver and three vehicles driving in pilot mode and forcing two out of 36 collision situations (*a lead vehicle stopped* and *a vehicle changing lanes*) defined by the National Traffic Safety Administration (NHTSA). The impact of a driver assistance system on the driver was one of the factors studied.

### 5.1.1 Experimental Setup

**Driving Simulator** Targeted enrichment of training data with safety-critical driving situations is essential to increase the performance of AI algorithms. Since the generation of corner cases in the real world is not an option for safety reasons, generation remains in the synthetic world, where specific critical driving situations can be simulated and recorded. For this purpose, the autonomous driving simulator CARLA [44] is used. It is open source software for data generation and/or testing of AI algorithms. It provides various sensors to describe the scenes such as camera, LiDAR and radar and delivers ground truth data. CARLA is based on the Unreal Engine game engine [160], which calculates and displays the behavior of various road users while taking physics into account, thus enabling realistic driving. Furthermore, the world of CARLA can be modified and adapted to one's own use case with the help of a Python API.

For our work, we used the API to modify the script for manual control from the CARLA repository. In doing so, we added another sensor, the inference sensor, which evaluates the CARLA RGB images in real-time and outputs the neural network semantic prediction on the screen. An example is shown in Figure 5.1. By connecting a control unit including a steering wheel, pedals and a screen, to CARLA, we make it possible to control a vehicle with **the eyes of the AI** in the synthetic world of CARLA. We also connected a second control unit with the same components to the simulator, so that it is possible to control the same vehicle with two different control units, see Figure 5.2. The second control unit is thus operated on the basis of CARLA's clear image and can intervene at any time. It always has priority and triggers the last three seconds of driving, which are buffered, to be written to the hard disk. The pseudocode for corner case triggering is shown in Algorithm 5.1. In order for the semantic driver to follow the traffic rules in CARLA, the script had to be modified additionally. The code has been modified to display the current traffic light phase in the upper right corner and the speed in the upper center. In addition, the record and replay function from CARLA was used to create a replay script, which can be used to reconstruct the driven scenes retrospectively and thus save the desired sensors subsequently.

**Test Rig** The test rig consists of the following components: a workstation with CPU, 3x GPU Quadro RTX 8000, 2 driving seats, 2 control units (steering wheel with pedals), one monitor for each control unit and two monitors for the control center. The driving simulator software used is the open source software CARLA version 0.9.10.

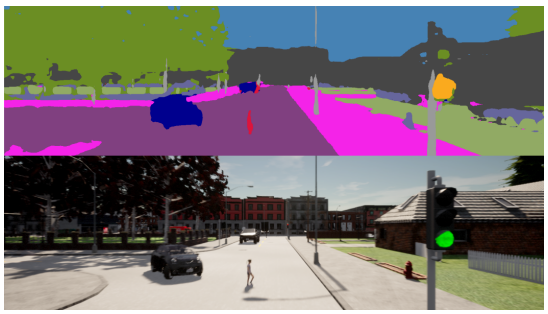


Figure 5.1: View of the semantic driver (top) and the safety driver (bottom).

Figure 5.2: Test rig including steering wheels, pedals, seats and screens.

**Algorithm 5.1:** Corner case detection with $(i, I_{CC}) \in \mathbb{N}_0$, $\delta_t \in \mathbb{R}^{\geq 0}$, $\nu \in [0, 1]$, $\varphi \in [-450, 450]$. The semantic driver is in full control of the vehicle using the `ManualControl()` function as long as the safety driver does not intervene. As soon as this happens, a counter $i$ goes up until a certain value $I_{CC}$ is exceeded and the safety driver is temporary in control over the vehicle. This was necessary to prevent very small movements of the steering wheel from being interpreted as intervention, which initially led to no smooth driving. The threshold value for $I_{CC}$ is kept low so that triggering in the millisecond range remains possible. The variable $\nu$ describes the pedal position of brake and throttle while $\varphi$ specifies the steering angle.

**Input:** $i$, $I_{CC}$, $d$, $t$, $\delta_t$
**Output:** `CornerCaseDetection()`
$t \leftarrow t_0$
**while** $i < I$ **do**
  **if** $\nu^{safe}_{brake}(t) = \nu^{safe}_{throttle}(t) = 0 \wedge \varphi^{safe}_{steer}(t) == \varphi^{safe}_{steer}(t - \delta_t)$ **then**
    $i \leftarrow 0$;
    `ManualControl`$(\nu^{sem}_{brake}, \nu^{sem}_{throttle}, \varphi^{sem}_{steer})$
  **else**
    $i \leftarrow i + 1$
    `ManualControl`$(\nu^{safe}_{brake}, \nu^{safe}_{throttle}, \varphi^{safe}_{steer})$
  $t \leftarrow t + \delta_t$

**Dataset for Initial Training and Testing** For training, a custom dataset was generated using CARLA 0.9.10, consisting of 85 scenes with 60 frames each. In

addition, there is a validation dataset with 20 scenes. The dataset was generated on seven maps with one fps and contains the corresponding semantic segmentation image in addition to the rendered synthetic image. The maps include the five standard maps in CARLA and two additional maps that offer a mix of city, highway and rural driving. Various parameters can be set in CARLA, we focused on the number of Non-Player-Characters (NPCs), including cars, motorcycles, bicycles and pedestrians, and on environment parameters such as sun position, wind and clouds. Depending on the size of the map, the number of NPCs ranged from 50 to 150.

The clouds and wind parameters can be set in the range between 0 and 100, with 100 being the highest value. The wind parameter is responsible for the movement of tree limbs and passing clouds and was in the range of 0 and 50. The cloud parameter describes the cloudiness, where 0 means that there are no clouds at all and 100 that the sky is completely covered with clouds. We have chosen values between 0 and 30. The altitude describes the angle of the sun in relation to the horizon of the CARLA world, with values between $-90$ (midnight) and 90 (midday). Values between 20 and 90 were used for our purpose. The other environmental parameters like rain, wetness, puddles or fog are set to zero. The parameters are chosen so that the scenes reflect everyday situations with a natural scattering of NPCs and in similar good weather. During data generation, the movement of all NPCs was controlled by CARLA.

Furthermore, 21 corner case scenes were used as test data, each containing 30 frames. Another test dataset containing 21 standard scenes without corner cases serves as a comparison, each containing 30 frames.

**Training** To drive on the predicted semantic mask, a real-time capable network architecture is needed. For these purposes, the Fast Segmentation Convolutional Neural Network (Fast-SCNN) model was used [139]. It uses two branches to combine spatial details at high resolution and deep feature extraction at lower resolution achieving a mean Intersection over Union (mIoU) of 0.68 at 123.5 fps on the Cityscapes dataset [35]. The network was implemented in the python package PyTorch [134] and training was done on a NVIDIA Quadro RTX 8000 graphics card. Sixteen of the 23 classes available in CARLA were used for training. Cross entropy was used as the loss function and ADAM as the optimization algorithm. A polynomial decay scheduler was also used to gradually reduce the learning rate.

We intentionally stopped the training after 5 epochs to increase the frequency of perception errors for the network. The resulting network is sufficiently well-trained to recognize the road and all road users, although objects further away are poorly recognized. An example is shown at the top of Figure 5.1.

**Experimental Design** Two operators conducted the driving campaign and the duration of driving as a safety driver or semantic driver was set at 50:50. Both participants had time to familiarize themselves with the hardware and the CARLA world before the start of the first driving campaign so that driving errors could be minimized. Two driving campaigns are planned; the first campaign will generate targeted corner cases and the second will test whether adding the corner cases included in campaign 1 leads to an improvement in the perception of safety-critical situations.

### 5.1.2 Retrieval of Corner Cases

For the generation of corner cases, we consider the following experimental setup. Two test operators record scenes in our specially constructed test rig (see Figure 5.2), where one subject (safety driver) gets to see the original virtual image and the other (semantic driver) the output of the semantic segmentation network (see Figure 5.1). The test rig is equipped with controls such as steering wheels, pedals and car seats and connected to CARLA to simulate realistic traffic participation.

The corner cases were generated as shown in Figure 5.3, using the real-time semantic segmentation network Fast-SCNN where visual perception was limited by intentionally stopping training early. This is sufficient to move in the virtual streets, but is poor enough to enhance corner cases of the *Method Layer*. We note that, according to [169], there were 128 accidents involving autonomous vehicles on the road during test operations in 2014-2018, at least 6% of which can be directly linked to misbehavior by the autonomous vehicle. It follows that at least every 775,335 km driven, a wrongful behavior of the autonomous vehicle occurs. Using a poorly trained network as a part of our accelerated testing strategy, we were able to generate corner cases after 3.34 km in average between interventions of the safety driver. We note, however, that the efficiency of the corner cases was evaluated using a fully trained network. Figure 5.4 shows two safety-critical corner cases where the safety driver had to intervene to prevent a collision.

If the safety driver triggered the recording of a corner case, the test operators label the corner case with one of four options available (overlooking a walker or a vehicle, disregarding traffic rules, intervening out of boredom) and may leave a comment. Furthermore, the kilometers driven and the duration of the ride are notated. The operators were told to obey the traffic rules and not to drive faster than 50 km/h during the test drives. After a certain familiarization period, driving errors decreased and sudden braking by the semantic driver was also reduced. The reason for this is that the network partially represents areas as vehicles or pedestrians with fewer pixels. Over time, the drivers learned to ignore

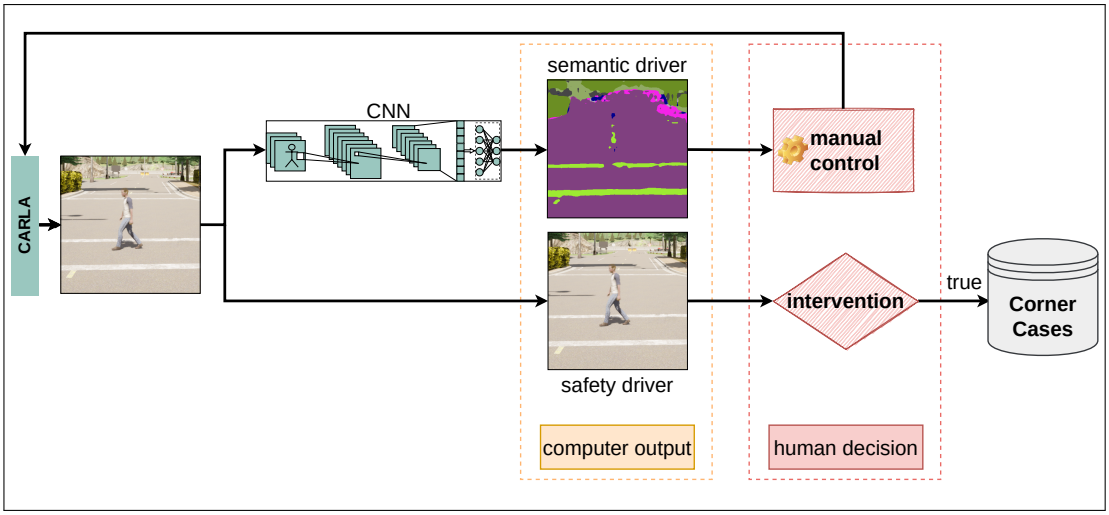



Figure 5.3: Two human subjects can control the ego vehicle. The semantic driver moves the vehicle in compliance with traffic rules in the virtual world and sees only the output of the semantic segmentation network. The safety driver, who sees only the original image, assumes the role of a driving instructor and intervenes in the situation by braking or changing the steering angle as soon as a hazardous situation occurs. Intervening in the current situation indicates poor situation recognition of the segmentation network and represents a corner case. Triggering a corner case ends the acquisition process and a new run can be started.

such situations because experience showed that there was no object there based on the previous frames.

The rides are tracked and by the intervention of the safety driver the last 3 seconds of the scene are saved. Subsequently, the scenes can be loaded and images saved from the ego vehicle's perspective using the camera and the semantic segmentation sensor. We collect 50 corner cases before retraining from scratch with a mixture of original and corner case images. For each corner case, the last 3 seconds are saved at 10 fps before the intervention by the safety driver. In total, we get 1500 new frames. When using this corner case data for retraining, we delete the same number of frames from the original training dataset.

We selected 50 corner cases in connection with pedestrians. Therefore, the inclusion of corner case scenes into the training dataset significantly increases the average number of pixels with the pedestrian class in the training data. To establish a fair comparison of the efficiency of corner cases as compared to a simple upsampling of the pedestrian class, we created a third dataset that contains approximately the same number of pixels per scene for the pedestrian class as the dataset with the corner cases, see Table 5.1.

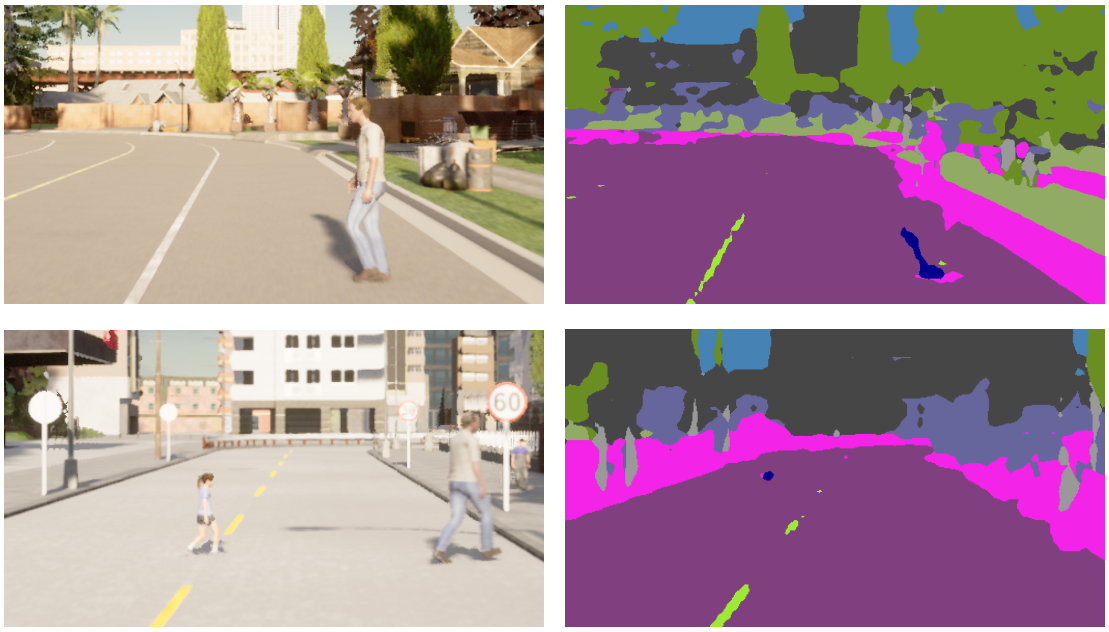


Figure 5.4: Two examples of a corner case where the safety driver had to intervene to avoid a collision due to the poor prediction of the semantic segmentation network (both images on the right).

| **train data** | | **test data** | | | |
|---|---|---|---|---|---|
| | | safety-critical | | natural distribution | |
| *name* | $pixel_{ped}/scene$ | $IoU_{ped}$ | $mIoU$ | $IoU_{ped}$ | $mIoU$ |
| natural distribution | 3583.6 | 0.4600 | 0.6954 | 0.4937 | **0.7610** |
| pedestrian enriched | 6101.1 | 0.5399 | 0.6911 | **0.5586** | 0.7554 |
| corner case enriched | 6215.7 | **0.5683** | **0.7173** | 0.5384 | 0.7517 |

Table 5.1: Performance measurement on two test datasets. The comparison shows that the addition of safety-critical scenes in training also improves performance in testing with safety-critical scenes.

### 5.1.3 Evaluation and Results

All results in this section are averaged over 5 experiments to obtain a better statistical validity. For testing purposes, we generated 21 additional corner cases for validation. With the same setup as before, we train the Fast-SCNN for 200 epochs on all three datasets and thereby obtain three networks. Table 5.1 shows the evaluation of all three models on the class pedestrian for the 21 safety-critical test corner cases. We see that adding corner cases to the training data leads to an improvement in pedestrian detection in safety-critical situations, which can also be shown by an example in Figure 5.5. There we see a situation with a pedestrian crossing the road, with a slope directly behind him that seems to end the road at

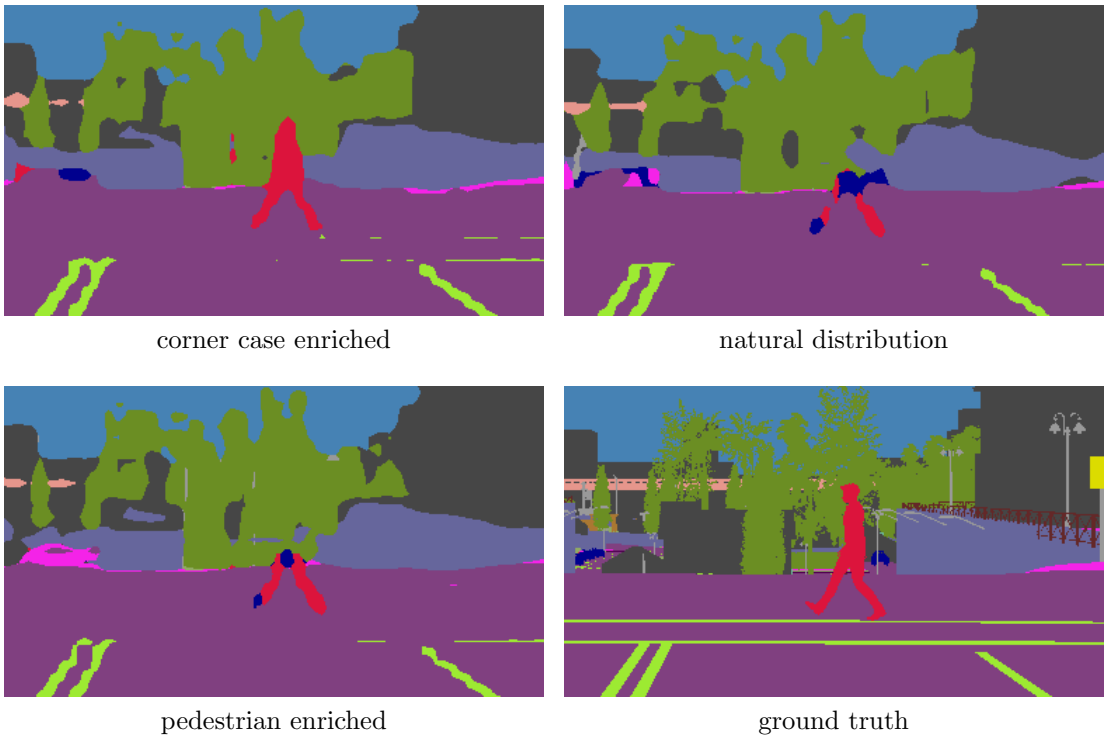



Figure 5.5: Evaluation on corner case test data shows that the model using corner case data in training recognizes pedestrians better than the model trained with the natural distributed dataset or the dataset which contains more pedestrians.

the level of the horizon. Therefore, the networks that did not have corner cases in the training data seem to have problems with this situation, while the model with corner cases detects the humans much better.

While training the network using naive upsampling of pedestrians does not have any positive effect on the classe's Intersection over Union (IoU) as compared with the original training data, we achieve a gain in the IoU by 2.19% when using the dataset containing corner cases. In addition, the 3 models were tested on a dataset with a natural distribution of pedestrians. Here it can be seen that the model trained with corner cases does not perform as well as the model with the same number of pedestrians. It follows that the model performs better in critical situations, while the models without corner cases perform less well.

Since our method for generating corner cases in safety-critical situations provides an improvement in detecting pedestrians in safety-critical situations, we launched a second campaign to verify how long it takes driving with the 3 trained networks to identify a corner case. When conducting the second driving campaign, the same operating parameters were set as in the first campaign and the duration until a corner case occurred was recorded. This includes the same maps and

weather conditions as well as the same two drivers. However, the two drivers were not aware of the network's underlying training data. Table 5.2 shows the duration and kilometers driven for the different datasets, as well as the occurrence of corner cases during these rides. We see that adding corner cases during training also reduces the frequency until new corner cases reappear.

| **dataset** | **distance** $d$ [km] | **time** $t$ [min] | **#CC** [-] | **mean**$_{d_{CC}}$ [km/CC] | **std**$_{d_{CC}}$ [km/CC] | **mean**$_{t_{CC}}$ [min/CC] | **std**$_{t_{CC}}$ [min/CC] |
|---|---|---|---|---|---|---|---|
| natural disritbution | 121.32 | 411 | 13 | 7.73 | 14.25 | 25.93 | 39.60 |
| pedestrian enriched | 163.09 | 500 | 21 | 7.52 | 10.47 | 23.25 | 28.72 |
| corner case enriched | 153.38 | 528 | 11 | **13.84** | 8.68 | **47.47** | 31.87 |

Table 5.2: Corner case appearances on Fast-SCNN trained with 3 different datasets.

We therefore demonstrated the benefits of our method to generate corner cases, especially for safety-critical situations. We were also able to show that adding safety-critical corner cases recorded by intentional perceptional distortions improves performance, so future datasets should include such situations.

### 5.1.4 Conclusion

Due to the lack of explanation and transparency in the decision-making of today's AI algorithms, we developed an experimental setup that allows to visualize these decisions and thus to allow a human driver to evaluate the driving situations while **driving with the eyes of AI**, and from this to extract data that includes safety-critical driving situations. Our self-developed test rig provides two human drivers to control the ego vehicle in the virtual world of CARLA. The semantic driver receives the output of a semantic segmentation network in real-time, based on which she or he is supposed to navigate in the virtual world. The second driver takes the role of the driving instructor and intervenes in dangerous driving situations caused by misjudgments of the AI. We consider driver interventions by the safety driver as safety-critical corner cases which subsequently replaced part of the initial training data. We were able to show that targeted data enrichment with corner cases created with limited perception leads to improved pedestrian detection in critical situations.

In addition, we continue the further development of AI by means of human risk perception to identify situations that are particularly important to humans and thus train the AI precisely where it is particularly challenged by a human perspective. Therefore we next investigate whether a single network is required to overcome the so-called domain gap, which describes the difference in data during training and deployment, or whether, for cost and performance reasons, different

networks should be used depending on the task. This will be investigated in the next chapter using different weather conditions and survival analysis.

## 5.2 survAIval: Survival Analysis with the Eyes of AI [92]

If automotive manufacturers want to put autonomous vehicles higher than level 2 on the road, they should ensure that safety-critical driving situations are registered and that a safe solution for all road users is found as quickly as possible. One way to achieve this is to provide a large amount of diverse data to the network during training. Especially closing the gap of domain shifts requires targeted data generation from multiple domains to achieve a good performance. Even if using more data and the best models leads to overcoming the domain gap, the question is whether this is the most efficient way from the manufacturer's point of view. In this regard, we investigate whether overcoming the domain gap in different weather conditions with specialized networks works as well or even better than a general model in the sense that all weather modalities are covered during training. Here, we make use of the method presented in Section 5.1 for finding synthetic corner cases with two human drivers and evaluate them using survival analysis to make statements about the influencing variables on upcoming corner cases.

### 5.2.1 Survival Analysis

Survival analysis is the study of lifespans, also survival times, and their influencing factors [115]. It uses statistical methods to investigate time intervals between sequential events. Groups, but also individuals can be considered as the unit of study when an expected event happens during a considered time period like the time from birth until death, the time from entry a clinical trial until death, the time from buying a vehicle until an accident happens, or other use cases. The basic goals of survival analysis are [89]:

- estimation and interpretation of survivor or hazard functions
- comparing survivor and/or hazard functions
- relationship determination of explanatory variables to lifespans

First, some typical terms of survival analysis are introduced with an overview in Table 5.3.

The observation time period is described by a beginning point $t_{start} = 0$ and an end point $t_{end} > 0$ defined by a failure condition due to a special event [77]. An event

| term | explaination |
|---|---|
| observation time | observation period for which start and end points are known |
| entity | single object or individual of the observed study |
| event | change in status (e.g. life to death, accident-free to accident) |
| entry | starting state (e.g. birth, date of vehicle purchase) |
| failure time T | exit time of a subject |
| risk set | all test objects in the study |
| censoring | incomplete information about either entry before or/and event after the observation time |
| truncation | non-observable data that either does not exist or whose entry and exit state have not been observed |
| lifespan | duration until an event occurs |
| hazard | probability that an observed entity has a certain event at time $t$ |

Table 5.3: Terms in survival analysis.

implies a change in status, e.g., from alive to dead, from healthy to sick, or from accident-free to accident and is usually easy to find. However, defining the exact failure event is a more difficult task in some cases [113]. Although it is desirable to know each the beginning and end point of an individual observed in the study, one or both are not always observed which is known as censoring. Figure 5.6 provides an overview of some typical observation types, where white circles describe the entry state. Using our experiments with the driving simulator, the beginning point of pedal pressing may describe the entry state. A cross represents a change of state, such as the occurrence of a corner case due to an impaired perception, while black circles refer to a change of state that was triggered by unexpected reasons like an intervention out of boredom rather than a corner case as cause of impaired perception. Observations 1 and 9 describe a *truncated state*, which is non-observable data that either does not exist or whose entry and exit state have not been observed. Observations 2, 7, 8 characterize *left-censored* data as their starting points are not identifiable as they occurred prior the observation start. In addition, observations 5 to 8 escape the observation time unchanged, so they are referred to as *right-censored* as their exit event could not be observed. In addition, the events of observations 2–4 are observed during the observation time, with only 3 being *uncensored* since both start and end times are known. Although an event was detected at observation 4, the expected event did not occur and/or there were other causes for this condition.

Parts of the theory of survival analysis are taken from [89], unless otherwise stated. The continuous random variable $T$ describes the time of occurrence of an event, which denotes the time of death of a subject, the time of failure of a machine, start of a disease or similar. $t$ denotes a particular time of interest, which can be used to describe the probability that $T$ has not yet occurred at time $t$, i.e., that the entity has survived. Accordingly, the survival function $S(t)$ represents

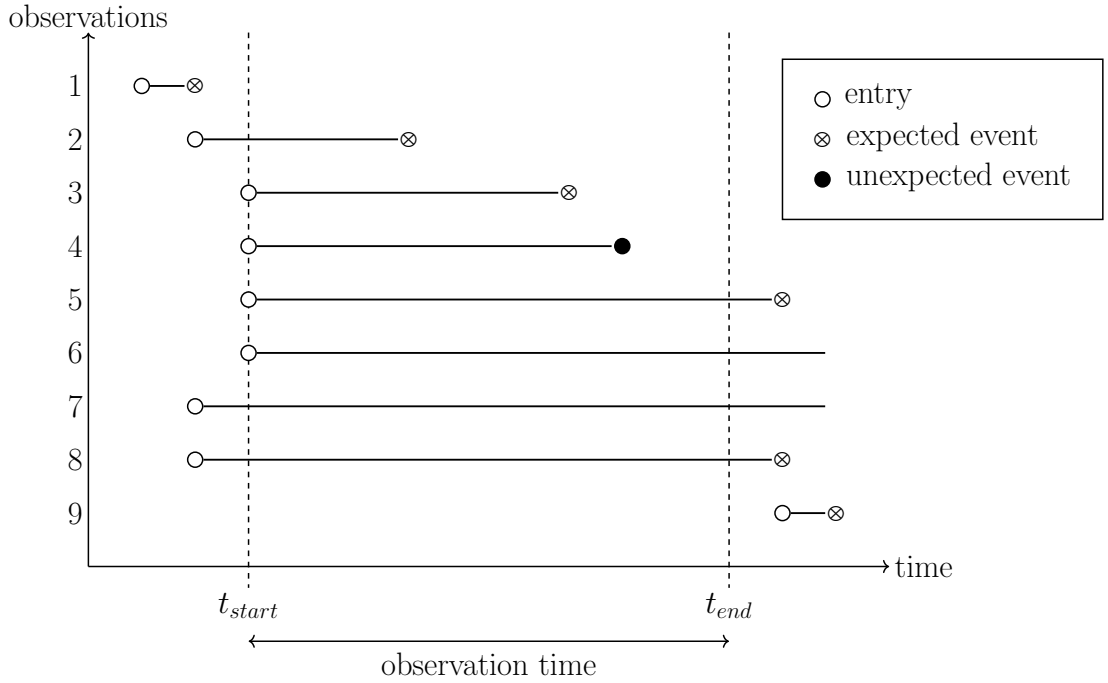



Figure 5.6: Examples of different observation types. Circles mark the beginning of an observation, while crosses or black circles mark an event. When there is no information about either the entry and/or the event state, this is referred to as censoring.

the probability that the event of an entity at time $t$ did not occur in the observed time period, and can be formulated as follows:

$$S(t) = Pr(T > t) \ . \tag{5.1}$$

Two ways to describe a survival distribution are survival and hazard functions. As a survival function, the so-called Kaplan-Meier [49] estimator is often used, which estimates the probability that an event for an entity does not occur within a certain time interval. It is defined as follows:

$$\hat{S}(t_j) = \prod_{i=0}^{j} \frac{n_i - d_i}{n_i} \ . \tag{5.2}$$

The observation time $t_j$ is therefore divided into $j$-parts, each of which considers a time interval $\Delta t = (t_i, t_{i+1}]$. With $n$ being denoted by the number of entities which are alive at $\Delta t$ and $d$ the number of entities which already left the observation at $\Delta t$.

Since $T$ is a continuous random variable, it is necessary to work with the probability density function $f(t)$, which describes the probability, that an event occurs

in a time interval. The cumulative density function $F(t)$, which is the area under the density function up to the value $t$, describes the probability, that the event occurs at time $T \leq t$:

$$F(t) = \int_{-\infty}^{t} f(u)\, du \; . \tag{5.3}$$

On the other hand, if we consider the probability that an event will not occur until a given time, which is what the survival function means, we can also write the following:

$$S(t) = 1 - F(t) \; . \tag{5.4}$$

In many situations, it is crucial to know how an individual risk for a particular outcome changes over time due to other events. For example, weather conditions can negatively affect the lifespan of a semantic driver when the model was not trained with such data. In addition, the use of multiple unknown weather variables can lead to interactions, which in turn can alter a semantic driver's lifespan. For those cases the hazard rate $h(t)$ indicates the probability that an observed entity experiences a failure event the next short time interval $\Delta t$ [88]. It describes the risk of the actual failure rate as a function of time.

The hazard rate is defined as

$$h(t) = \lim_{\Delta t \to 0+} \frac{Pr(t \leq T < t + \Delta t | t \leq T)}{\Delta t} = \frac{f(t)}{S(t)} \; . \tag{5.5}$$

The cumulative hazard $H(t)$ is used to estimate the hazard probability which is defined as follows:

$$H(t) = -\log(S(t)) = \int_{0}^{t} h(s)\, ds \; . \tag{5.6}$$

The hazard ratio (HR) is a measure of the relative survival experience of two groups ($A$ or $B$) and is defined as follows:

$$HR = \frac{O_A / E_A}{O_B / E_B} \; . \tag{5.7}$$

The ratio $O/E$ describes the relative death rate of a group, where $O$ is the observed number of deaths and $E$ the expected number of deaths [113]. The HR is useful to compare two individuals or groups.

The Cox proportional hazards model, introduced in 1972 [36], uses the hazard

function as a function of the influencing variables and is defined as

$$h(t, \mathbf{Z}) = h_0(t) \exp\left(\sum_{i=1}^{p} \beta_i Z_i\right), \quad \mathbf{Z} = (Z_1, Z_2, \ldots, Z_p), \tag{5.8}$$

where $h_0$ describes the baseline hazard, which depends only on time and is therefore equivalent to the Kaplan-Meier estimator. $\mathbf{Z}$ denotes the influence variables, which are time-independent and $\beta$ the regression coefficients of the influence variables to be estimated.

The Cox model is often called proportional hazards model since the ratio of the risk for 2 entities with covariates $\mathbf{Z}$ and $\mathbf{Z}^*$ is proportional. The relative risk, also known as the hazard ratio (HR), describes that an individual with risk factor $\mathbf{Z}$ will experience an event proportional to an individual with risk factor $\mathbf{Z}^*$. The relative risk is defined as follows: [88]

$$HR = \frac{h(t, \mathbf{Z})}{h(t, \mathbf{Z}^*)} = \frac{h_0(t) \exp(\sum_{i=1}^{p} \beta_i Z_i)}{h_0(t) \exp(\sum_{i=1}^{p} \beta_i Z_i^*)} \tag{5.9}$$

$$= \exp\left[\sum_{i=1}^{p} \beta_i (Z_i - Z_i^*)\right] . \tag{5.10}$$

It becomes noticeable that HR is independent of time.

Additionally, probabilities about the occurrence of an event can be calculated with the hazard function so that the influence of different parameters can be taken into account. Furthermore, events that have already occurred are included in the calculation so that the probability of an event occurring in the next time step can be predicted. This can be done with the partial likelihood, including a risk set $R(t_d)$ and an index set of death times $D$:

$$L(\beta) = \prod_{d=1}^{D} \frac{\exp(\sum_{i=1}^{p} \beta_i Z_{di})}{\sum_{j \in R(t_d)} \exp(\sum_{i=1}^{p} \beta_i Z_{ji})} . \tag{5.11}$$

To optimize the regression coefficients we can maximize the log-likelihood, i.e.,

$$\beta^* = \operatorname{argmax} \log(L(\beta)) . \tag{5.12}$$

This is done by computing:

$$\nabla_\beta \log(L(\beta)) = 0 , \tag{5.13}$$

which can be solved numerically.

### 5.2.2 Experimental Design

After learning the basics of survival analysis, we will use it to find factors that affect survival while driving in the driving simulator. We will use the setup presented in Section 5.1 and observe how long it takes for a corner case to occur under different weather conditions. For this study, the previously used semantic segmentation network Fast-SCNN [139] is trained on good weather data, which we refer to *clear*, and serves as a baseline before being fine-tuned with different weather conditions, namely *rain*, *fog* and *night*. Figure 5.7 gives an overview of the different weather conditions. In addition, a further model is re-trained on all 3 weather conditions, referred to as *mix*, resulting in a total of 5 models available for the experiments. For post-training, 2100 additional images per weather setting (300 per map) are provided for training and 420 for testing. In the following, we

rain fog night

Figure 5.7: Overview of the used weather conditions. The grayish sky, falling water drops as well as water puddles on the road are characteristic for *rain*. In the case of *fog*, fine water droplets cover the image, and it is especially tough to see in depth. *Night* images are characterized by many dark areas, with streetlights and vehicle lights illuminating the scenes.

refer to each of the weather conditions *rain*, *fog* and *night* as expert models, since they are specifically trained on one domain. In contrast, all 3 weather settings are available to the *mix* model during training, which we refer to universal model. The baseline and universal models are tested on all five test datasets, whereas the expert models are tested on the respective trained conditions as well as on the *clear* ones. Table 5.4 gives an overview of the performance of all models on the particular test data.

The evaluation of the initial model shows a significant decrease of all IoU values in any weather conditions, with the safety-critical class human below 0.1 for fog and night being awful. In contrast, the performance of the universal model remains largely the same. Additionally, compared to the mix model, the expert models perform better in rain and night and worse in fog for the human class. In the mIoU, the universal model always outperforms the experts. This comparison has already shown the tendency for the expert models to perform at least as well or even slightly better than the universal model in the human class, while the overall

| model | test data | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | **clear** | | **rain** | | **fog** | | **night** | | **mix** | |
| | $IoU_{ped}$ | $mIoU$ | $IoU_{ped}$ | $mIoU$ | $IoU_{ped}$ | $mIoU$ | $IoU_{ped}$ | $mIoU$ | $IoU_{ped}$ | $mIoU$ |
| clear | **0.487** | **0.759** | 0.368 | 0.586 | 0.024 | 0.207 | 0.063 | 0.191 | 0.123 | 0.321 |
| rain | 0.379 | 0.606 | **0.485** | 0.718 | - | - | - | - | - | - |
| fog | 0.074 | 0.130 | - | - | 0.301 | 0.596 | - | - | - | - |
| night | 0.292 | 0.302 | - | - | - | - | **0.402** | 0.655 | - | - |
| mix | 0.451 | 0.657 | 0.471 | **0.734** | **0.326** | **0.644** | 0.369 | **0.694** | **0.402** | **0.682** |

Table 5.4: Test data model performance in all weather conditions.

performance in the mIoU is best for the universal model in all weather conditions. The next step is to conduct the weather driving campaign, where each model is also tested under these weather conditions in order to obtain a reliable statement about its performance in test.

The experiments are conducted as described in Section 5.1, so that two drivers drive freely on the roads of CARLA. During the rides, the semantic driver has full control over the vehicle, while the safety driver observes the rides and should intervene in the scene only in safety-critical driving situations using the brake pedal or the steering wheel. Intervention indicates incorrect assessment of the scene, which is a corner case of the *Method Layer*. Differences from the previous driving campaigns include the number of maps and the duration of the rides. This time, the focus is only on Town01 and Town03, since they have a high variability and due to their moderate size the number of vehicles and pedestrians does not need to be set excessively high in order to consistently see some, which relieves the traffic manager and thus computations on the CPU. In addition to the reduced number of maps, the drives will be limited to 600 seconds. If no corner case occurs during this time, the drive is stopped, which corresponds to a right-censored observation. In addition, the drivers didn't know what data the network had been trained on during the experiments as well as what weather condition they were driving in. The baseline and universal models are tested for 120 minutes on each weather setting (*clear*, *rain*, *fog*, *night*). In addition, the expert models are tested on the respective weather condition, also for 120 minutes each. In total, this results in 1320 minutes with 11 different combinations.

### 5.2.3 Results

A total of 160 drives with a maximum length of 600 seconds were performed. If no corner case occurs in this time, the drives are aborted so that we have a right-censored data point. Therefore, the number of rides per combination varies, as models in which a corner case appears more quickly can also be driven more frequently. The software used for survival analysis is lifelines [40]. Figure 5.8

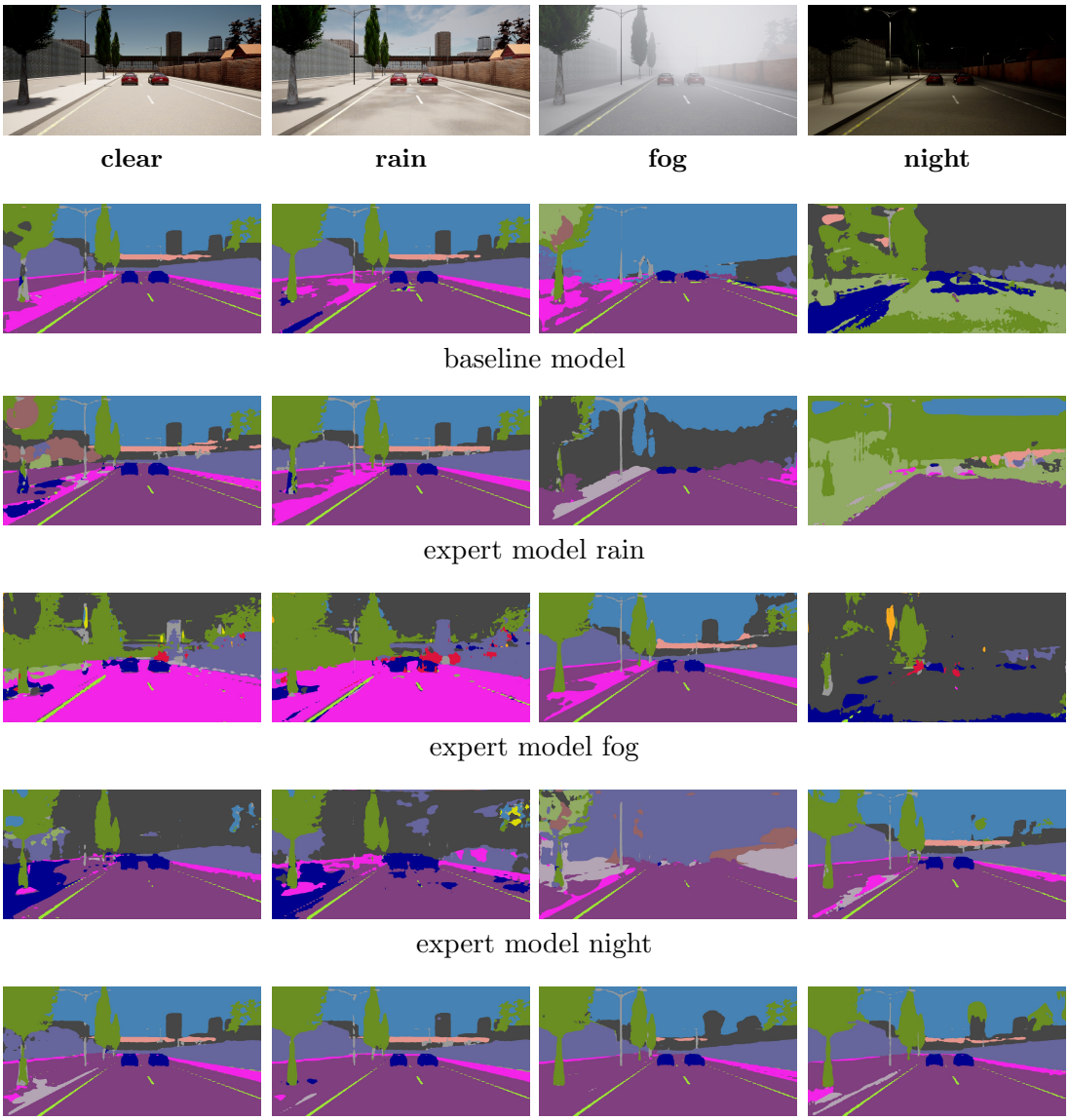



Figure 5.8: Model outputs on each weather setup. Under the baseline model, it would be still possible to drive in rain, whereas fog and night would become a risk. The expert models perform well in their domain but quite worse in the other ones. On the other hand, the universal model performs sufficiently well in all weather conditions.

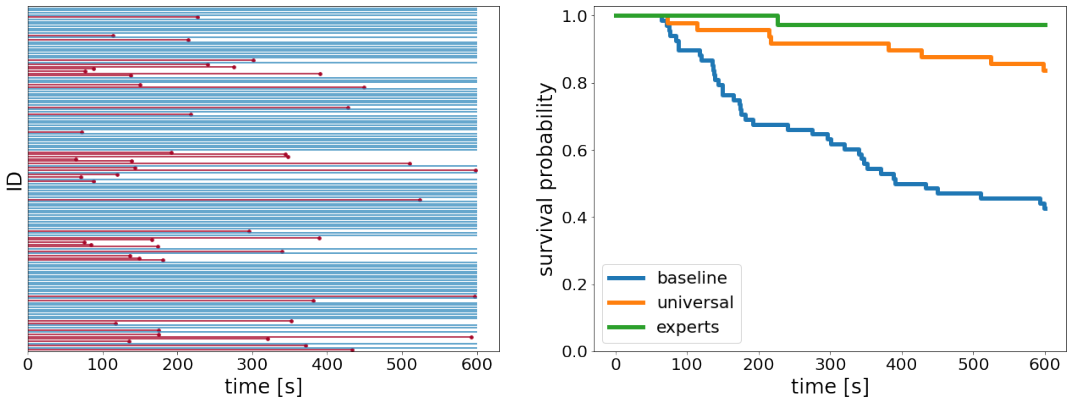
shows the predictions of the models in each weather condition, whereas Table 5.5 presents the total number of corner cases registered with respect to the trained model and the driven weather conditions. As we can see, there are barely corner cases in the expert models, which is why we group them together in their own

model type, the experts type. All observations during the study are visualized in Figure 5.9(a). In total, we have 48 observations of corner cases that can be used for survival analysis. Furthermore, the two students drove 406.838 km on the virtual streets of CARLA.

| **model type** | **trained** | **tested** | | | |
|---|---|---|---|---|---|
| | | clear | rain | fog | night |
| **baseline** | clear | 4 | 5 | 13 | 17 |
| **experts** | rain | - | 0 | - | - |
| | fog | - | - | 0 | - |
| | night | - | - | - | 1 |
| **universal** | mix | 1 | 3 | 1 | 3 |

Table 5.5: List of all observed corner cases during weather campaign by model and tested weather condition.

As a first step, we consider the plot for the Kaplan-Meier estimation in Figure 5.9(b) for the 3 model types *baseline, universal* and *experts*, which shows that the probability of a corner case occurring is lowest for the expert model, closely followed by the universal model. The baseline model seems to be very sensitive to different weather conditions, which is why there is only a survival probability of 63.24% after 300 seconds and at the end of the observation period only 42.65%.




(a) lifespans

(b) Kaplan-Meier estimation

Figure 5.9: (a) Lifespans of all observations during the study. Red lines show the occurrence of a corner case, whereas blue lines are right-censored. The majority of the drives, approx. 70%, did not lead to a corner case. (b) Kaplan-Meier estimation for all model types. The probability that no corner case occurs is highest in the expert models, followed by the universal model. The poor generalizability in bad weather provides that the survival probability in the base model decreases significantly over time.

We then use the Cox model to obtain the regression coefficients. For this, the input variables must first be preprocessed. For the weather parameter *rain* the

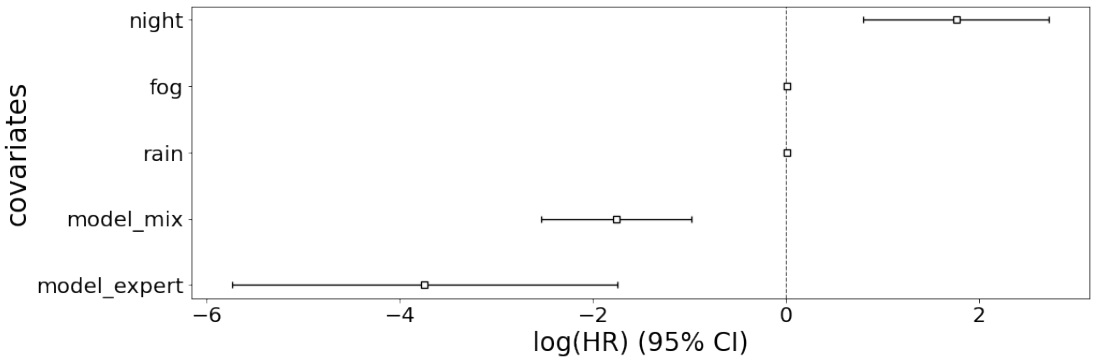



Figure 5.10: The comparison of the hazard ratios shows that the night ensures that a corner case is more likely to occur. If an expert or universal model is used instead, a corner case occurs less frequently, which is also evident from the Kaplan-Meier estimate.

values can range from 70 to 100 and for *fog* from 50 to 100. The parameter night is assigned to a Boolean variable and the value 1 is set as soon as the sun position parameter ($\in [-90, 90]$) is $< 0$. Additionally, we distinguish on which model we are driving, for this we use also a Boolean variable and set a 1 for either the expert model or the universal model.

Table 5.6 shows the evaluations of the Cox model. The analysis demonstrates that 3 covariates can be classified as significant, as their confidence interval is below 0.05. Fog is significant with 92% and rain even only with 58%.

The hazard rate is calculated using the expert model as an example. Since this value is a boolean variable, it can be calculated as follows:

$$HR_{expert} = \frac{h_{expert=1}(t)}{h_{expert=0}(t)} = 0.02 \; . \tag{5.14}$$

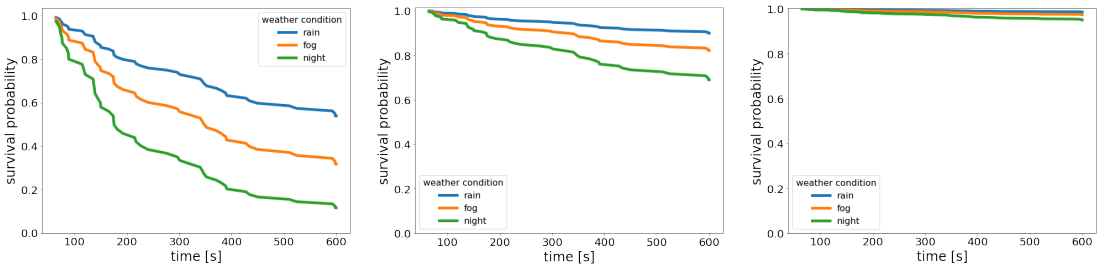
Driving with an expert model reduces the hazard rate by 98% with a low ranging confidence interval.

| **covariate** | **hazard ratio HR** | **95% confidence interval for the hazard ratio** | **confidence level p** |
|---|---|---|---|
| rain | 1.01 | 0.99 - 1.02 | 0.42 |
| fog | 1.01 | 1.00 - 1.02 | 0.08 |
| night | 5.83 | 2.23 - 15.22 | $< 0.005$ |
| experts | 0.02 | 0.00 - 0.17 | $< 0.005$ |
| universal | 0.17 | 0.08 - 0.38 | $< 0.005$ |

Table 5.6: Cox proportional hazards model.

Next we have a closer look to the probabilities for all models in different weather conditions. Figure 5.11 shows the performance for all models over time and for the

weather conditions *rain, fog, night*. The baseline model has the biggest problems when driving on unseen weather conditions, with the highest probability of a corner case occurring at *night*. It also appears to be the most problematic for the universal and expert models, with significantly higher survival probabilities. The comparison between the universal and the expert models indicates that the latter perform noticeably better on their trained domains than the universal models.




(a) baseline (b) universal (c) experts

Figure 5.11: The survival probabilities for rain, fog and night clearly show that the baseline model struggles with all bad weather settings. The universal model seems to be more robust, but the probability of survival at night also drops to 69% at the end of the study, whereas the expert model assures a survival of 95%.

### 5.2.4 Conclusion

Although the validity of such a few data points must be treated with caution, a trend does seem to emerge, namely that the use of expert models indeed seems to be more appropriate, as an omniscient model has to find a balance to perform well in each domain. Therefore, it may be useful to focus on some basic data and add other models for special cases that are temporarily responsible for prediction. Examples of use would be driving in left-hand traffic or in snowy winter regions, so that an appropriately trained model could be used. It would also be conceivable to have a separate trained model for each country that may be used when crossing borders. This solution might be based on the vehicle's GPS coordinates and would not require an additional upstream classification network.

## 5.3 A Taxonomy of Corner Cases in Trajectory Datasets for Automated Driving [150]

As a next step, corner cases from the *Temporal Layer* will be considered, in which several consecutive frames can describe a safety-critical driving situation.

Especially for autonomous driving vehicles, the focus is on safe navigation in road traffic, which requires knowledge about future movement and behavior patterns of all road users. This is why recent studies focus on the task of motion planning for autonomous vehicles finding a path or trajectory to avoid collisions [131]. The so-called trajectory of each road-user needs to be predicted in order to avoid harm. Especially unusual but critical and rare trajectories can be challenging for AI algorithms and need to be considered in training and validation for machine learning methods. This is why we introduced a definition of trajectory corner cases as follows:

*A corner case in a trajectory dataset is a highly relevant but mostly very rare and anomalous trajectory. The relevance of a trajectory is mainly determined by the interaction with other agents (e.g., road users), the surrounding environment, norms and (traffic) rules, and most importantly, by the task at hand.*

Based on this definition, a taxonomy was developed to help understand critical driving situations and, if applicable, provide reasons for their occurrence. Here, corner cases are categorized according to their origin in the driver's information processing system, whether human or machine, and the corner case class. For this purpose, we developed a unified model of the information processing pipeline for automated vehicles, human drivers, and vulnerable road users (VRUs) with different stages, see Figure 5.12. We used Wickens' human information process pipeline as a role model, which is widely used and generally accepted in traffic psychology and traffic safety research [155, 171].

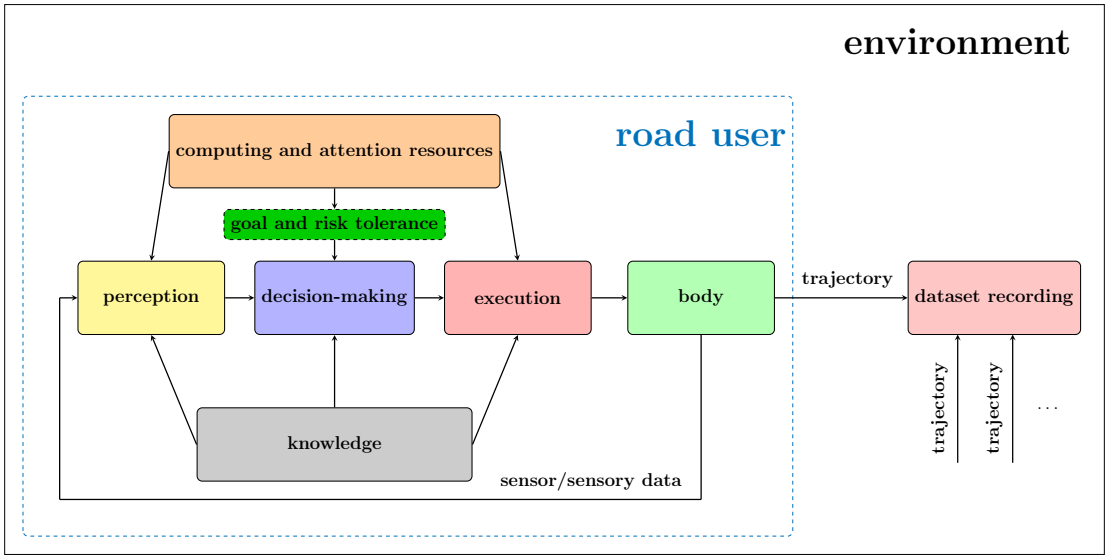



Figure 5.12: Schematic of unified processing pipeline for all kinds of road users (human, automated vehicle or VRU). The figure has been slightly modified for simplification.

The driver information processing system produces data that can be divided into

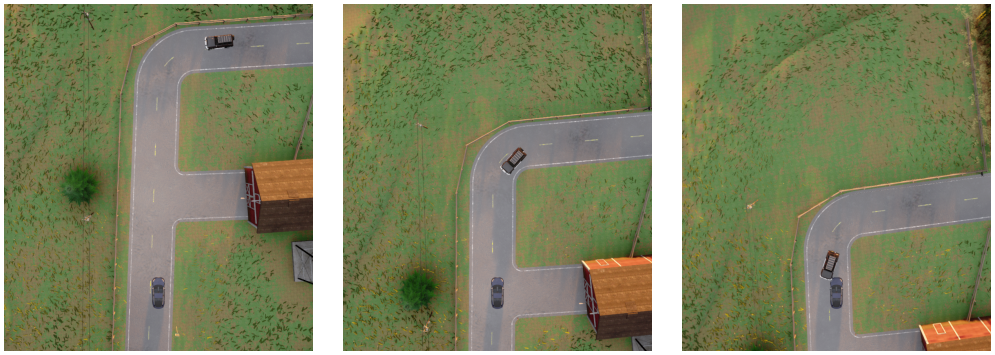
different classes of trajectory corner cases, differentiating between single and multiple trajectories with or without environmental influences. The final taxonomy with examples can be found in Table 5.7.

| stage | ego trajectory | ego trajectory & other road users | ego trajectory & environment | ego trajectory & other road users & environment |
|---|---|---|---|---|
| **perception** | emergency braking due to inattention | near collision due to missed junction sign | wrongly driving in one-way street due to overlooked sign | crash due to overlooked sign |
| **decision making** | high velocity in sharp corner | near collision in lane changing due to time pressure | planned disregard for a puddle that covers a large hole | failure to understand priority at complex crossing |
| **goal and risk tolerance** | kick-down start | tail gating | cutting corners | willingly taken right of way |
| **execution** | stall engine | rear-end collision because of insufficient braking | scrape obstacle when parking | rolling too far into an intersection due to insufficient braking |
| **body** | tire burst | rear-end collision due to brake failure | collision due to failing headlights | taken right of way due to broken windshield or camera optics |
| **computing and attention resources** | any of the examples from stages 1 – 5 | | | |
| **knowledge** | high velocity in surprisingly sharp corner | pedestrian at crosswalk looking away is interpreted as yielding | U-turn in no passing area | slow approach to intersection interpreted as giving right of way |
| **environment** | crash with fallen tree | collision due skidding on ice | leaving the road due to aquaplaning | taken right of way due to impaired visibility |
| **trajectory recording** | noisy IMU measurements or failed tracking of ego road user in camera image | | | |

Table 5.7: Taxonomy with example situations for trajectory corner cases. The rows describe the system component where the corner case occurs. The columns identify the trajectory corner case class, which describes the type of data required to detect the corner case. Ego refers to the road user causing the corner case. The colors represent the responsible area in the processing pipeline. The table has been slightly modified for simplification.©2023 IEEE

During the experiments from Section 5.1, where two human drivers were driving in the virtual streets of CARLA, various corner case situations were recorded from which the trajectory data could subsequently be obtained and classified into the developed taxonomy. In the following there will be shown some examples that contributed to the structure of the table, among others.

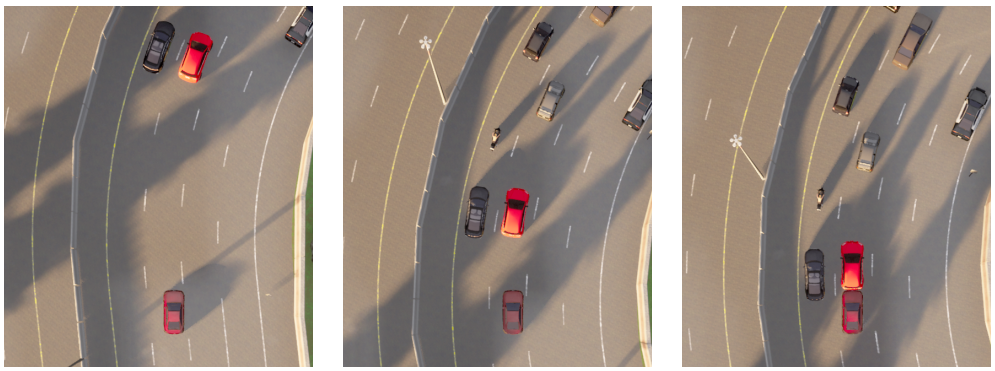
In Figure 5.13 an accident occurs even though the ego vehicle is obeying the traffic rules due to the fact that the road user in the opposite lane is driving too fast around the curve and crosses the lane of the ego vehicle. Due to the increased speed, the road user crosses the oncoming lane, so this situation would be classified as *goal and risk tolerance* related to the environmental trajectory.



(a) (b) (c)

Figure 5.13: While driving too fast in a tight curve, one road user crossed the oncoming lane, resulting in a head-on collision between two road users. The consequence of a high *goal and risk tolerance*.

A rear-end collision into an end of a traffic jam can be caused by too timid braking in combination with driving too fast, see Figure 5.14. The road user seems to have underestimated his own speed or the low speed of the road users ahead. Such a situation would be classified in the *execution* stage including *ego trajectory & other road users*.



(a) (b) (c)

Figure 5.14: Driving into the end of a traffic jam due to insufficient braking by the road user resulted in a collision.

Figure 5.15 shows a vehicle driving towards a two-lane traffic circle. Next to this vehicle is a bus, which somewhat obscures the view of the road. While the

bus driver allows a cyclist to pass, the vehicle owner collides with him due to impaired visibility. This situation is *not* classified as a *perception* error, because the cyclist was not perceivable behind the bus. Instead, as the *decision-making* should account for the possibility of occluded road users with priority, the error is attributed to this category.

(a) (b) (c)

Figure 5.15: Due to overlapping phenomena in the traffic cycle, the ego road user could not see the oncoming cyclist at the right time, which led to a near collision. A *perception* error with impaired *decision-making*.

Figure 5.16 describes the illegal turn into a one-way street due to a perceptual error, namely overlooking the one-way street sign. Looking only at the ego trajectory, nothing has gone wrong since the vehicle is still on the asphalted road and does not make any untypical movements. If, on the other hand, we look at the context, it becomes obvious that overlooking the one-way street places this *perception* error in the *ego trajectory & environment* category.

(a) (b) (c)

Figure 5.16: The unauthorized turn of the ego vehicle into a one-way street leads to a perception error of the category *environment*.

Chapter 6

# Conclusion & Outlook

When thinking about autonomous driving vehicles that move safely through traffic, it is necessary to perceive the environment correctly in order to provide safe driving. To ensure this, deep learning algorithms must be extensively trained and tested with data required to solve the task. This data should cover numerous driving situations and ideally also include safety-critical scenes. Scenes can become safety critical if they were not present in the training data for deep learning algorithms and lead to uncertainty and incorrect predictions when confronted with them. Therefore, in this work, we have addressed the detection and generation of such safety-critical driving situations. To this end, the theoretical foundations of deep learning were first presented, describing the learning process and introducing semantic segmentation as a computer vision task for autonomous driving. A driving simulator was then constructed to display the output of a semantic segmentation network to the screen in real-time, allowing a human driver to move a vehicle in a simulated environment using a control unit, including a steering wheel and pedals, a seat and one or three monitors. The construction of the simulator as well as the open source software CARLA, which serves as the simulation environment for the scientific work, were described in detail. Furthermore, the inference sensor implemented in CARLA was presented, which enables driving on the output of a semantic segmentation network in the first place. Additionally, all improvement steps, including code acceleration techniques, that were necessary to enable driving with the inference sensor in real-time (at least 24 fps) were explained.

Since this driving simulator is planned to be used for finding safety-critical situations in road traffic, the term corner case and its different levels needed first to be introduced. Since the main focus of this work is on the so-called *Method Layer* corner case, which is based on epistemic uncertainty, the consideration of the other layers was partially investigated in side projects, see [17, 111, 150].

With a well-functioning driving simulator providing a real-time output of a semantic segmentation network and the knowledge about corner cases, the detecting of the latter was tackled. For this purpose, the driving simulator was equipped with a second control unit along with a seat and monitor, so that two human drivers are able to control the same vehicle in the simulation environment. Even though both drivers can control the same vehicle at the same time, the display output is different. One of the drivers gets to see the output of the semantic segmentation network, whereas the other driver sees the original output of CARLA. The latter is given the task of a supervisor and should only take control of the vehicle as soon as a safety-critical driving situation arises according to human understanding. With this setup, we were able to recognize and record safety-critical driving situations that are challenging for the AI, so-called methodical corner cases, in a first driving campaign. With these recordings, we were then able to train a model that was less susceptible to methodical corner cases, resulting in longer, damage-free drivings within a second driving campaign.

Following the example of the first two driving campaigns, a third campaign was launched. In predefined observation times, it was measured whether and how long it takes for a corner case to appear. The survival time was examined on models that were trained on special weather domains (rain, fog, night) and subsequently tested in these domains. In comparison, a universal model was trained to face all domains during training, which was also tested on these weather domains. These runs were evaluated using the Cox proportional hazard model to determine influencing variables on corner case occurrence frequency. Even though the small number of data points is not yet significant enough, there seems to be a tendency that expert models are an option for time and cost reasons, which can be added according to the application at hand. This would be possible, for example, through an upstream domain classification model that first classifies the current scene into a domain and then reads in the suitably trained model.

Even though the driving simulator was primarily designed to find corner cases quickly and safely, it can also be used as a demonstration object to make deep learning algorithms available to society in the form of visual output. In doing so, each and every person could form his/her own opinion whether to trust the "unknown" machine. In addition, this driving simulator should contribute to the acceptance of a new technology, since autonomous driving under *Level 5*, according to scientists and automotive producers, is only a matter of time.

As a demonstration object, it is easy to show the limitations of deep learning systems and how important the selection of data is for training. Simple change of e.g. weather parameters is possible by pressing a button to demonstrate difficulties in for the model unknown domains. Students can also be introduced to the field of deep learning in a demonstrative way, which can support the university teaching

mission. In this context, the driving simulator can help to achieve rapid success in this area playfully, making research fun.

Although a major part of this work has to do with data generation, there is also another way to increase safety when corner cases occur. This is important because no matter how much corner case data is produced, there are always unknown objects hard to deal with or possible combinations of known parameters that can lead to uncertainty, in the worst case, to traffic accidents. One way to be more robust against such corner cases is to use at least two redundant systems with different sensors. In this way, sensor-specific problems can be intercepted by the other system, leading to increased safety. This is why we introduced the YOdar approach, which demonstrates that a fusion model using data from two different sensors, namely camera and radar, provides an improvement in object detection in a critical domain, if barely known, such as nighttime compared to the results of each model individually.

Let us now turn to the outlook to conduct further research with the results presented in this work. Even though we are able to record safety-critical driving situations with the driving simulator, these are only available in the synthetic world. There are clear discrepancies between the image quality and the natural movements of vehicles and pedestrians compared to reality. This domain gap must be addressed in future research. One way that has become increasingly popular in recent years is the use of so-called *Generative Adversarial Networks (GAN)*, which could be used to transfer synthetic images into the real world. The authors of [76, 78, 183] are already showing promising results. Therefore, we are optimistic that in the future research will be able to generate real-looking images from the corner cases created in this work, as these can subsequently be used to train neural networks with a mixture of real and synthetic images. It could also save time and money by eliminating the need for expensive labeling of data, as this information is already present.

It would also be conceivable to carry out the experiments from Chapter 5 in the real world. Therefore, for example, a driving school vehicle could be used that can be controlled with two control units and is equipped with a camera that captures real-time images and transmits them to a computer that also feeds the image through a semantic segmentation network. This output could be visualized to the semantic driver through VR (*Virtual Reality*) glasses. In this case, the safety driver would take over the task of the driving instructor, who ensures that safe driving is guaranteed, as known in driving schools. In the process, driving situations similar to the setup in Chapter 5 can be recreated with the eyes of the AI and safety-critical driving situations can be found in reality. Of course, these experiments should first be conducted in a closed area and at low speeds.

The promising results from the YOdar approach could be investigated in more

detail as soon as more radar data is available. Talking of which, as the YOdar approach uses real data, this could also be outsourced to the synthetic world once a physics-based sensor is implemented in a simulated environment. This might also save time and costs, although the domain shift between the real and synthetic world would also need to be investigated. An interesting research question would also be whether the use of a third sensor, e.g. LiDAR, makes sense and/or whether two sensors are sufficient to increase safety to such an extent that the cost-benefit calculation using three sensors does not add up. The incorporation and use of two sensors is from a vehicle manufacturer's point of view preferable to three, as long as the level of safety remains reasonably the same. Real-time processing of all three sensor data can also be difficult and requires additional research.

In summary, the addition of corner cases for deep learning algorithms in automotive is relevant for safety reasons and a contribution to this can be made by the driving simulator presented in this thesis, which makes the following possible:

***Driving with the eyes of AI for corner case generation.***

# List of Figures

# List of Tables

# List of Notations and Abbreviations

In this thesis, the following abbreviations and notations are used:

| | |
|---|---|
| $\mathcal{X}$ | set of input data |
| $\mathcal{Y}$ | set of labels |
| $\mathcal{S}$ | dataset |
| $N$ | datasets full length |
| $\mathcal{D}$ | unknown distribution |
| $\mathbb{I}$ | indicator function |
| $\widehat{y}$ | predicted class probability vector |
| $y$ | class label |
| $x$ | input data |
| $s$ | data pairs |
| $h_s$ | predictor |
| $\theta$ | network parameters |
| $w$ | weight |
| $b$ | bias |
| $h$ | predictor |
| $\phi$ | activation function |
| $\rho$ | propagation function |
| $a$ | output of a perceptron |
| $i_s$ | iteration step |
| $\lambda_{pen}$ | penalty parameter |
| $\mathcal{L}, \ell$ | error/loss function |
| $IoU$ | intersection over union |
| $mIoU$ | mean intersection over union |
| $mAP$ | mean average precision |
| $TP$ | number of true positives |
| $FP$ | number of false positives |
| $TN$ | number of true negatives |
| $FN$ | number of false negatives |
| $\eta$ | stepsize |
| $E_{ES}$ | termination condition for early stopping |
| $K_s$, $k_s$ | subsets/subset for KFCV method |

| | |
|---|---|
| $M$ | feature map |
| $K$ | kernel |
| $I$ | image |
| $L, l$ | number of layers/layer |
| $s_w$ | stride |
| $t_\delta$ | fixed delta time in CARLA |
| $\gamma$ | downsampling ratio |
| $n_p$ | number model parameters |
| $N_s$ | number of slices |
| $N_t$ | number of time steps |
| $N_f$ | number of features |
| $r$ | radar data |
| $q$ | ground truth |
| $q_a$ | ground truth box |
| $s$ | slice |
| $B, b$ | set of boxes, box |
| $\xi$ | radar parameters |
| $v$ | vehicle |
| $c$ | center point |
| $z$ | objectness score value |
| $\varsigma$ | box width |
| $\vartheta$ | box height |
| $\kappa$ | number of boxes |
| $\sigma$ | standard deviation |
| $\Gamma$ | threshold |
| $D$ | dimensions |
| $\nu$ | pedal position |
| $\varphi$ | steering angle |
| $I_{CC}$ | threshold for corner case |
| $\delta_t$ | time step |
| $t$ | time |
| $T$ | failure/exit time |
| $\Delta t$ | time interval |
| $S(t)$ | survival function |
| $F(t)$ | cumulative density function |
| $f(t)$ | probability density function |
| $h(t)$ | hazard function |
| $H(t)$ | cumulative hazard |
| $HR$ | hazard ratio |
| $O$ | observed number of deaths |
| $E$ | number of expected deaths |
| $\beta$ | regression coefficients |
| $p$ | confidence level |
| $Z$ | influencing variables on hazard function |

| | |
|---|---|
| AI | Artificial Intelligent |
| API | Application Programming Interface |
| CC | Corner Case |
| CNN | Convolutional Neural Networks |
| CPU | Central Processing Unit |
| CUDA | Compute Unified Device Architecture |
| CWL | CARLA WildLife dataset |
| DL | Deep Learning |
| DSConv | Depth-wise Separable Convolutions |
| DWConv | Depth-wise Convolutions |
| ERM | Empirical Risk Minimization |
| EULA | End User License Agreement |
| FN | False Negative |
| FP | False Positive |
| fps | frames per second |
| GAN | Generative Adversarial Network |
| GPS | Global Positioning System |
| GPU | Graphics Processing Unit |
| HDD | Hard Disk Drive |
| HR | Hazard Ratio |
| IMU | Inertial Measurement Unit |
| IoU | Intersection Over Union |
| IPS | In Plane Switching |
| KFCV | K-Fold Cross Validation |
| LCD | Liquid Crystal Display |
| LOOCV | Leave-One-Out Cross Validation |
| mAP | mean Average Precision |
| mIoU | mean Intersection over Union |
| MLP | Multilayer Perceptron |
| NHTSA | National Traffic Safety Administration |
| NPC | Non-Player Character |
| ONNX | Open Neural Network eXchange |
| OoD | Out-of-Distribution |
| OS | Operating System |
| PCIe | Peripheral Component Interconnect express |
| PPM | Pyramid Pooling Module |
| RAID | Redundant Arrays of Inexpensive Disks |
| ReLU | Rectified Linear Unit |
| RNN | Recurrent Neural Network |
| ROP | Raster Operations Pipeline |
| RT | Ray Tracing |

| | |
|---|---|
| SAE | Society of Automotive Engineers |
| SGD | Stochastic Gradient Descent |
| SOS | Street Obstacle Sequences |
| SSD | Solid State Drive |
| TM | Traffic Manager |
| TMU | Texture Mapping Unit |
| TN | True Negative |
| TP | True Positive |
| TRT | TensorRT |
| UE | Unreal Engine |
| VR | Virtual Reality |
| VRU | Vulnerable Road User |
| WOS | Wuppertal Obstacle Sequences |
| WQHD | Wide Quad High Definition |
| YOLO | You Only Look Once |

# Bibliography


[1] M. Abadi, A. Agarwal, P. Barham, et al., *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*, CoRR, abs/1603.04467 (2016).

[2] E. H. Adelson, *On seeing stuff: the perception of materials by humans and machines*, in Human Vision and Electronic Imaging VI, B. E. Rogowitz and T. N. Pappas, eds., vol. 4299, International Society for Optics and Photonics, SPIE, 2001, pp. 1 – 12.

[3] C. C. Aggarwal, *Neural Networks and Deep Learning - A Textbook*, Springer, 2018.

[4] R. Aldrich and T. Wickramarathne, *Low-cost radar for object tracking in autonomous driving: A data-fusion approach*, in 2018 IEEE 87th Vehicular Technology Conference (VTC Spring), 2018, pp. 1–5.

[5] B. Arlandis et al., *Oversteer - Steering Wheel Manager for Linux (Version 0.6.0)*. [Online] https://github.com/berarma/oversteer, 2019. Accessed: 2023-02-15.

[6] Baidu, Inc., *Apollo - An open autonomous driving platform*. [Online] https://github.com/ApolloAuto/apollo, 2017. Accessed: 2023-02-08.

[7] B. Benjdira, T. Khursheed, A. Koubaa, et al., *Car Detection using Unmanned Aerial Vehicles: Comparison between Faster R-CNN and YOLOv3*, in 2019 1st International Conference on Unmanned Vehicle Systems-Oman (UVS), feb 2019, pp. 1–6.

[8] V. Besnier, A. Bursuc, D. Picard, and A. Briot, *Triggering failures: Out-of-distribution detection by learning from local adversarial attacks in semantic segmentation*, in Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV), October 2021, pp. 15701–15710.

[9] P. Bevandić, I. Krešo, M. Oršić, and S. Šegvić, *Simultaneous Semantic Segmentation and Outlier Detection in Presence of Domain Shift*, in German Conference on Pattern Recognition (GCPR), 2019.

[10] G. D. Biase, H. Blum, R. Siegwart, and C. Cadena, *Pixel-Wise Anomaly Detection in Complex Driving Scenes*, in IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2021, virtual, June 19-25, 2021, Computer Vision Foundation / IEEE, 2021, pp. 16918–16927.

[11] C. M. Bishop, *Pattern recognition and machine learning, 8th Edition*, Information science and statistics, Springer, 2006.

[12] H. Blum, P.-E. Sarlin, J. Nieto, R. Siegwart, and C. Cadena, *Fishyscapes: A Benchmark for Safe Semantic Segmentation in Autonomous Driving*, in International Conference on Computer Vision (ICCV) Workshop, 2019.

[13] ——, *The Fishyscapes Benchmark: Measuring Blind Spots in Semantic Segmentation*, International Journal of Computer Vision (IJCV), 129 (2021).

[14] D. Bogdoll, E. Eisen, C. Scheib, M. Nitsche, and J. M. Zöllner, *Multimodal Detection of Unknown Objects on Roads for Autonomous Driving*, in International Conference on Systems, Man and Cybernetics (SMC), 2022.

[15] D. Bogdoll, S. Guneshka, and J. M. Zöllner, *One Ontology to Rule Them All: Corner Case Scenarios for Autonomous Driving*, in European Conference on Computer Vision (ECCV) Workshop, 2022.

[16] D. Bogdoll, M. Nitsche, and J. M. Zöllner, *Anomaly Detection in Autonomous Driving: A Survey*, in Conference on Computer Vision and Pattern Recognition (CVPR) Workshop, 2022.

[17] D. Bogdoll, S. Uhlemeyer, K. Kowol, and J. M. Zöllner, *Perception datasets for anomaly detection in autonomous driving: A survey*, in 2023 IEEE Intelligent Vehicles Symposium (IV), 2023, pp. 1–8.

[18] J.-A. Bolte, A. Bar, D. Lipinski, and T. Fingscheidt, *Towards Corner Case Detection for Autonomous Driving*, in 2019 IEEE Intelligent Vehicles Symposium, IV 2019, Paris, France, June 9-12, 2019, IEEE, 2019, pp. 438–445.

[19] J. Breitenstein, J.-A. Termöhlen, D. Lipinski, and T. Fingscheidt, *Systematization of Corner Cases for Visual Perception in Automated Driving*, in IEEE Intelligent Vehicles Symposium, IV 2020, Las Vegas, NV, USA, October 19 - November 13, 2020, IEEE, 2020, pp. 1257–1264.

[20] ——, *Corner Cases for Visual Perception in Automated Driving: Some Guidance on Detection Approaches*, CoRR, abs/2102.05897 (2021).

[21] D. Brüggemann, R. Chan, M. Rottmann, H. Gottschalk, and S. Bracke, *Detecting Out of Distribution Objects in Semantic Segmentation of Street Scenes*, in The 30th European Safety and Reliability Conference (ESREL), vol. 2, 2020.

[22] H. Caesar, V. Bankiti, A. H. Lang, et al., *nuScenes: A Multimodal Dataset for Autonomous Driving*, in 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020, IEEE, 2020, pp. 11618–11628.

[23] R. Chan, K. Lis, S. Uhlemeyer, H. Blum, S. Honari, R. Siegwart, P. Fua, M. Salzmann, and M. Rottmann, *SegmentMeIfYouCan: A Benchmark for Anomaly Segmentation*, in Conference on Neural Information Processing Systems (NeurIPS), 2021.

[24] R. Chan, M. Rottmann, and H. Gottschalk, *Entropy maximization and meta classification for out-of-distribution detection in semantic segmentation*, in Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV), Oct. 2021, pp. 5128–5137.

[25] R. Chan, M. Rottmann, F. Hüger, P. Schlicht, and H. Gottschalk, *Controlled False Negative Reduction of Minority Classes in Semantic Segmentation*, in 2020 IEEE International Joint Conference on Neural Networks (IJCNN), 2020.

[26] R. Chan, S. Uhlemeyer, M. Rottmann, and H. Gottschalk, *Deep Neural Networks and Data for Automated Driving*, Springer, 2022, ch. Detecting and Learning the Unknown in Semantic Segmentation.

[27] L.-C. Chen, Y. Zhu, G. Papandreou, F. Schroff, and H. Adam, *Encoder-decoder with atrous separable convolution for semantic image segmentation*, in European Conference on Computer Vision, 2018.

[28] W. Chen, X. Gong, X. Liu, Q. Zhang, Y. Li, and Z. Wang, *FasterSeg: Searching for Faster Real-time Semantic Segmentation*, in International Conference on Learning Representations, 2020.

[29] X. Chen, J. Wang, and M. Hebert, *Panonet: Real-time panoptic segmentation through position-sensitive feature embedding*, 2020.

[30] F. Chollet, *Keras - Deep learning library. Available: https://github.com/keras-team/keras*, 2015.

[31] F. Chollet, *Deep learning with python*, Manning Publications, New York, NY, Oct. 2017.

[32] ——, *Xception: Deep Learning With Depthwise Separable Convolutions*, in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Jul. 2017.

[33] J. A. Clark, *Python Imaging Library (Fork).* [Online] https://python-pillow.org/, 2010. Accessed: 2023-03-26.

[34] S. Cook, *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*, Applications of GPU Computing Series, Elsevier Science, Boston, 2013.

[35] M. Cordts, M. Omran, S. Ramos, et al., *The Cityscapes Dataset for Semantic Urban Scene Understanding*, in Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016.

[36] Cox, D. R., *Regression Models and Life-Tables*, Journal of the Royal Statistical Society: Series B (Methodological), 34 (1972), pp. 187–202.

[37] Crytek GmbH, *CryEngine.* [Online] https://www.cryengine.com/, 2002. Accessed: 2023-03-26.

[38] G. V. Cybenko, *Approximation by superpositions of a sigmoidal function*, Mathematics of Control, Signals and Systems, 2 (1989), pp. 303–314.

[39] J. Dai, K. He, and J. Sun, *Instance-Aware Semantic Segmentation via Multi-task Network Cascades*, in 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 3150–3158.

[40] C. Davidson-Pilon, *Lifelines: Survival Analysis in Python*, Journal of Open Source Software, 4 (2019), p. 1317.

[41] F. de Ponte Müller, *Survey on Ranging Sensors and Cooperative Techniques for Relative Positioning of Vehicles*, Sensors, 17 (2017).

[42] Defense Advanced Research Projects Agency, *AI Next Campaign. Available: https://www.darpa.mil/work-with-us/ai-next-campaign*, 2018. Accessed 07-Jun,-2022.

[43] M. Denil, B. Shakibi, L. Dinh, M. A. Ranzato, and N. de Freitas, *Predicting Parameters in Deep Learning*, in Advances in Neural Information Processing Systems, C.J. Burges and L. Bottou and M. Welling and Z. Ghahramani and K.Q. Weinberger, ed., vol. 26, Curran Associates, Inc., 2013.

[44] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, *CARLA: An Open Urban Driving Simulator*, in Proceedings of the 1st Annual Conference on Robot Learning, 2017, pp. 1–16.

[45] K. Driggs-Campbell, V. Shia, and R. Bajcsy, *Improved driver modeling for human-in-the-loop vehicular control*, in 2015 IEEE International Conference on Robotics and Automation (ICRA), 2015, pp. 1654–1661.

[46] X. Du, Z. Wang, M. Cai, and Y. Li, *VOS: Learning What You Don't Know by Virtual Outlier Synthesis*, in International Conference on Learning Representations (ICML), 2022.

[47] J. Duchi, E. Hazan, and Y. Singer, *Adaptive Subgradient Methods for Online Learning and Stochastic Optimization*, Journal of Machine Learning Research, 12 (2011), pp. 2121–2159.

[48] M. Dupuis and H. Grezlikowski, *OpenDRIVE – An Open Standard for the Description of Roads in Driving Simulations*, in Proceedings of the Driving Simulation Conference DSC Europe, 2006, pp. 25–35.

[49] E. L. Kaplan and Paul Meier , *Nonparametric estimation from incomplete observations*, Journal of the American Statistical Association, 53 (1958), pp. 457–481.

[50] G. Echeverria et al., *Modular open robots simulation engine: Morse*, in 2011 IEEE International Conference on Robotics and Automation, 2011, pp. 46–51.

[51] Epic Games, *Unreal® Engine End User License Agreement*. [Online] `https://www.unrealengine.com/de/eula-reference/content-de`, 1998. Accessed: 2023-02-12.

[52] ——, *Unreal® Engine EULA Change Log*. [Online] `https://www.unrealengine.com/de/eula-change-log/unreal`, 2023. Accessed: 2023-02-12.

[53] M. Fan, S. Lai, J. Huang, X. Wei, Z. Chai, J. Luo, and X. Wei, *Rethinking BiSeNet for Real-Time Semantic Segmentation*, in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Jun. 2021, pp. 9716–9725.

[54] M. Fellendorf, *VISSIM: A Microscopic Simulation Tool to Evaluate Actuated Signal Control including Bus Priority*, in Proceedings of the 64th ITE Annual Meeting, Oct. 1994.

[55] D. A. Forsyth and J. Ponce, *Computer Vision - A Modern Approach, Second Edition*, Pitman, 2012.

[56] J. H. Friedman, *Stochastic gradient boosting*, Computational Statistics & Data Analysis, 38 (2002), pp. 367–378.

[57] P. Fritsche, S. Kueppers, G. Briese, et al., *Radar and lidar sensorfusion in low visibility environments*, in Proceedings of the 13th International Conference on Informatics in Control, Automation and Robotics (ICINCO 2016) - Volume 2, Lisbon, Portugal, July 29-31, 2016, SciTePress, 2016, pp. 30–36.

[58] T. Genevois, J.-B. Horel, A. Renzaglia, and C. Laugier, *Augmented Reality on LiDAR data: Going beyond Vehicle-in-the-Loop for Automotive Software Validation*, in Intelligent Vehicles Symposium (IV), 2022.

[59] R. Girshick, J. Donahue, T. Darrell, and J. Malik, *Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation*, in 2014 IEEE Conference on Computer Vision and Pattern Recognition, 2014, pp. 580–587.

[60] GitHub Inc. (former Logical Awesome LLC), *GitHub - Build software better, together.* [Online] https://github.com/, 2008. Accessed: 2023-02-25.

[61] GitLab Inc., *GitLab - Create, review and deploy code together.* [Online] https://about.gitlab.com/, 2012. Accessed: 2023-02-25.

[62] I. J. Goodfellow, Y. Bengio, and A. C. Courville, *Deep Learning*, Adaptive computation and machine learning, MIT Press, 2016.

[63] M. Grcić, P. Bevandić, and S. Šegvić, *Dense anomaly detection by robust learning on synthetic negative data*, arXiv preprint arXiv:2112.12833, (2021).

[64] M. Grcić., P. Bevandić., and S. Segvić., *Dense Open-set Recognition with Synthetic Outliers Generated by Real NVP*, in International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISAPP), 2021.

[65] S. Gu, Y. Zhang, J. Tang, et al., *Road Detection through CRF based LiDAR-Camera Fusion*, in 2019 International Conference on Robotics and Automation (ICRA), 2019, pp. 3832–3838.

[66] A. E. Gómez et al., *Driving simulator platform for development and evaluation of safety and emergency systems*, Cornell University Library, (2018).

[67] P. Hang, Y. Zhang, and C. Lv, *Interacting with human drivers: Human-like driving and decision making for autonomous vehicles*, 2022.

[68] M. K. Hansen and J. P. Underwood, *Multi-modal obstacle detection in unstructured environments with conditional random fields*, CoRR, abs/1706.02908 (2017).

[69] S. Hanson and L. Pratt, *Comparing Biases for Minimal Network Construction with Back-Propagation*, in Advances in Neural Information Processing Systems, D. Touretzky, ed., vol. 1, Morgan-Kaufmann, 1988.

[70] B. Hariharan, P. Arbeláez, R. Girshick, and J. Malik, *Simultaneous detection and segmentation*, in Computer Vision – ECCV 2014, D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, eds., Cham, 2014, Springer International Publishing, pp. 297–312.

[71] K. He, G. Gkioxari, P. Dollár, and R. Girshick, *Mask R-CNN*, in 2017 IEEE International Conference on Computer Vision (ICCV), 2017, pp. 2980–2988.

[72] K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition*, in 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 770–778.

[73] F. Heidecker, J. Breitenstein, K. Rösch, et al., *An Application-Driven Conceptualization of Corner Cases for Perception in Highly Automated Driving*, in 2021 IEEE Intelligent Vehicles Symposium (IV), Nagoya, Japan, 2021.

[74] D. Hendrycks, S. Basart, M. Mazeika, A. Zou, J. Kwon, M. Mostajabi, J. Steinhardt, and D. Song, *Scaling Out-of-Distribution Detection for Real-World Settings*, in International Conference on Machine Learning (ICML), 2022.

[75] P. T. Highnam, *The defense advanced research projects agency's artificial intelligence vision*, AI Magazine, 41 (2020), pp. 83–85.

[76] J. Hoffman, E. Tzeng, T. Park, J.-Y. Zhu, et al., *CyCADA: Cycle Consistent Adversarial Domain Adaptation*, in International Conference on Machine Learning (ICML), 2018.

[77] D. Hosmer, S. Lemeshow, and S. May, *Applied Survival Analysis: Regression Modeling of Time-to-Event Data*, Wiley Series in Probability and Statistics, Wiley, 2008.

[78] X. Huang and S. Belongie, *Arbitrary Style Transfer in Real-Time with Adaptive Instance Normalization*, in 2017 IEEE International Conference on Computer Vision (ICCV), 2017, pp. 1510–1519.

[79] S. Ioffe and C. Szegedy, *Batch normalization: Accelerating deep network training by reducing internal covariate shift*, in International conference on machine learning, pmlr, 2015, pp. 448–456.

[80] ISO Central Secretary, *Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles*, Standard ISO/SAE PAS 22736:2021, International Organization for Standardization, Geneva, CH, 2021.

[81] Jaccard, Paul, *Lois de distribution florale dans la zone alpine*, Bulletin de la Société vaudoise des sciences naturelles, 38 (1902), pp. 69–130.

[82] ——, *THE DISTRIBUTION OF THE FLORA IN THE ALPINE ZONE.1*, New Phytologist, 11 (1912), pp. 37–50.

[83] V. John and S. Mita, *RVNet: Deep Sensor Fusion of Monocular Camera and Radar for Image-Based Obstacle Detection in Challenging Environments*, in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 11854 LNCS, 2019, pp. 351–364.

[84] A. Karpathy, *CVPR 2021 Workshop on Autonomous Driving, Keynote.*, 2021.

[85] A. Kendall, V. Badrinarayanan, and R. Cipolla, *Bayesian segnet: Model uncertainty in deep convolutional encoder-decoder architectures for scene understanding*, ArXiv, abs/1511.02680 (2015).

[86] D. P. Kingma and J. Ba, *Adam: A Method for Stochastic Optimization*, in 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings, Yoshua Bengio and Yann LeCun, ed., 2015.

[87] A. Kirillov, K. He, R. B. Girshick, C. Rother, and P. Dollár, *Panoptic Segmentation*, in IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019, Computer Vision Foundation / IEEE, 2019, pp. 9404–9413.

[88] J. P. Klein and M. L. Moeschberger, *Survival analysis*, Statistics for Biology and Health, Springer, New York, NY, 2 ed., Mar. 2003.

[89] D. G. Kleinbaum and M. Klein, *Survival Analysis*, Springer New York, 2012.

[90] T. Koduri, D. Bogdoll, S. Paudel, and G. Sholingar, *AUREATE: An Augmented Reality Test Environment for Realistic Simulations*, in World Congress Experience (WCX), SAE International, 2018.

[91] K. Kowol., S. Bracke., and H. Gottschalk., *A-Eye: Driving with the Eyes of AI for Corner Case Generation*, in Proceedings of the 6th International Conference on Computer-Human Interaction Research and Applications - CHIRA, INSTICC, SciTePress, 2022, pp. 41–48.

[92] K. Kowol, S. Bracke, and H. Gottschalk, *survAIval: Survival Analysis with the Eyes of AI*, in Computer-Human Interaction Research and Applications, A. Holzinger, H. P. da Silva, J. Vanderdonckt, and L. Constantine, eds., Cham, 2023, Springer Nature Switzerland, pp. 153–170.

[93] K. Kowol, M. Rottmann, S. Bracke, and H. Gottschalk, *YOdar: Uncertainty-based Sensor Fusion for Vehicle Detection with Camera and Radar Sensors*, in Proceedings of the 13th International Conference on Agents and Artificial Intelligence - Volume 2: ICAART,, INSTICC, SciTePress, 2021, pp. 177–186.

[94] A. Krizhevsky, I. Sutskever, and G. E. Hinton, *ImageNet Classification with Deep Convolutional Neural Networks*, in Advances in Neural Information Processing Systems, F. Pereira and C.J. Burges and L. Bottou and K.Q. Weinberger, ed., vol. 25, Curran Associates, Inc., 2012.

[95] Y. LeCun, *Generalization and network design strategies*, 1989.

[96] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, *Gradient-based learning applied to document recognition*, Proceedings of the IEEE, 86 (1998), pp. 2278–2324.

[97] K. Lemmer, *Neue autoMobilität / Automatisierter Straßenverkehr der Zukunft (acatech STUDIE)*, Herbert Utz Verlag, München, 2016.

[98] F.-F. Li and J. Etchemendy, *Introducing Stanford's Human-Centered AI Initiative. Available: https://hai.stanford.edu/news/introducing-stanfords-human-centered-ai-initiative*, 2018. Accessed 03-Jun,-2022.

[99] G. Li, Y. Yang, and X. Qu, *Deep Learning Approaches on Pedestrian Detection in Hazy Weather*, IEEE Transactions on Industrial Electronics, 67 (2020), pp. 8889–8899.

[100] K. Li, K. Chen, H. Wang, L. Hong, C. Ye, J. Han, Y. Chen, W. Zhang, C. Xu, D.-Y. Yeung, X. Liang, Z. Li, and H. Xu, *CODA: A Real-World Road Corner Case Dataset for Object Detection in Autonomous Driving*, in European Conference on Computer Vision (ECCV), 2022.

[101] Y. Li, X. Chen, Z. Zhu, L. Xie, G. Huang, D. Du, and X. Wang, *Attention-Guided Unified Network for Panoptic Segmentation*, in 2019

IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2019, pp. 7019–7028.

[102] Y. Li, H. Qi, J. Dai, X. Ji, and Y. Wei, *Fully Convolutional Instance-Aware Semantic Segmentation*, in 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017, pp. 4438–4446.

[103] T. Lin, M. Maire, S. J. Belongie, et al., *Microsoft COCO: common objects in context*, in Computer Vision - ECCV 2014 - 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V, vol. 8693 of Lecture Notes in Computer Science, Springer, 2014, pp. 740–755.

[104] K. Lis, S. Honari, P. Fua, and M. Salzmann, *Detecting road obstacles by erasing them*, arXiv preprint arXiv:2012.13633, (2020).

[105] K. Lis, K. Nakka, P. Fua, and M. Salzmann, *Detecting the unexpected via image resynthesis*, in Proceedings of the IEEE/CVF International Conference on Computer Vision, 2019, pp. 2152–2161.

[106] H. Liu, C. Peng, C. Yu, J. Wang, X. Liu, G. Yu, and W. Jiang, *An end-to-end network for panoptic segmentation*, in 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2019, pp. 6165–6174.

[107] Z. Liu, S. Yu, X. Wang, and N. Zheng, *Detecting drivable area for self-driving cars: An unsupervised approach*, CoRR, abs/1705.00451 (2017).

[108] Logitech International S.A., *G920/G29 - Racing wheel for Xbox, PlayStation and PC*. [Online] https://www.logitechg.com/en-us/products/driving/driving-force-racing-wheel.941-000110.html, 2015. Accessed: 2023-02-15.

[109] J. Long, E. Shelhamer, and T. Darrell, *Fully convolutional networks for semantic segmentation*, in IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015, IEEE Computer Society, 2015, pp. 3431–3440.

[110] P. A. Lopez, M. Behrisch, L. Bieker-Walz, et al., *Microscopic Traffic Simulation using SUMO*, in The 21st IEEE International Conference on Intelligent Transportation Systems, IEEE, 2018.

[111] K. Maag, R. Chan, S. Uhlemeyer, K. Kowol, and H. Gottschalk, *Two video data sets for tracking and retrieval of out of distribution objects*, in Computer Vision – ACCV 2022, L. Wang, J. Gall, T.-J. Chin, I. Sato, and R. Chellappa, eds., Cham, 2023, Springer Nature Switzerland, pp. 476–494.

[112] K. Maag, M. Rottmann, and H. Gottschalk, *Time-dynamic estimates of the reliability of deep semantic segmentation networks*, 2020 IEEE 32nd International Conference on Tools with Artificial Intelligence (ICTAI), (2020), pp. 502–509.

[113] D. Machin, Y. B. Cheung, and M. K. B. Parmar, *Survival Analysis - A Practical Approach*, John Wiley & Sons, Inc, 2 ed., 2006.

[114] Q.-C. Mao, H.-M. Sun, Y.-B. Liu, and R.-S. Jia, *Mini-yolov3: Real-time object detector for embedded applications*, IEEE Access, (2019), pp. 133529–133538.

[115] D. Moore, *Applied Survival Analysis Using R*, Use R!, Springer International Publishing, 2016.

[116] N. Morgan and H. Bourlard, *Generalization and Parameter Estimation in Feedforward Nets: Some Experiments*, in Advances in Neural Information Processing Systems, D. Touretzky, ed., vol. 2, Morgan-Kaufmann, 1989.

[117] K. P. Murphy, *Probabilistic Machine Learning: An Introduction*, MIT Press, 2022.

[118] National Transportation Safety Board, *Preliminary Report Highway HWY16FH018*, 2016.

[119] ——, *Preliminary Report Highway HWY18FH011*, 2018.

[120] ——, *Preliminary Report Highway HWY18MH010*, 2018.

[121] H.-H. Nguyen, D. N.-N. Tran, and J. W. Jeon, *Real-Time Semantic Segmentation on Edge Devices with Nvidia Jetson AGX Xavier*, in 2022 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia), 2022, pp. 1–4.

[122] T. Niggemeier, *Vergesslicher Speicher - Flash-Grundlagen, Teil 2: Lebensdauer der Daten (in german)*, c't, 11 (2021), pp. 116–121.

[123] F. Nobis, M. Geisslinger, M. Weber, et al., *A deep learning-based radar and camera sensor fusion architecture for object detection*, in 2019 Sensor Data Fusion: Trends, Solutions, Applications, SDF 2019, Bonn, Germany, October 15-17, 2019, IEEE, 2019, pp. 1–7.

[124] P. Nousi, E. Patsiouras, A. Tefas, and I. Pitas, *Convolutional neural networks for visual information analysis with limited computing resources*, in 2018 25th IEEE International Conference on Image Processing (ICIP), 2018, pp. 321–325.

[125] NVIDIA Corporation, *Nvidia TensorRT*. [Online] https://developer.nvidia.com/tensorrt, 2021. Accessed: 2022-03-03.

[126] ——, *NVIDIA Deep Learning TensorRT Documentation, update March 2023*. [Online] https://docs.nvidia.com/deeplearning/tensorrt/quick-start-guide/index.html, 2023. Accessed: 2023-03-26.

[127] ——, *NVIDIA DRIVE Sim$^{TM}$- Powered by Omniverse*. [Online] https://developer.nvidia.com/drive/simulatione, 2023. Accessed: 2023-02-08.

[128] P. Oberdiek, M. Rottmann, and G. A. Fink, *Detection and retrieval of out-of-distribution objects in semantic segmentation*, 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), (2020), pp. 1331–1340.

[129] T. Oliphant, *NumPy - The fundamental package for scientific computing with Python* . [Online] https://numpy.org/, 2005. Accessed: 2023-03-26.

[130] On-Road Automated Driving (ORAD) Committee, *Taxonomy and Definitions for Terms Related to On-Road Motor Vehicle Automated Driving Systems*, Jan. 2014.

[131] B. Paden, M. Čáp, S. Z. Yong, D. Yershov, and E. Frazzoli, *A Survey of Motion Planning and Control Techniques for Self-Driving Urban Vehicles*, IEEE Transactions on Intelligent Vehicles, 1 (2016).

[132] H. Pan, Y. Hong, W. Sun, and Y. Jia, *Deep dual-resolution networks for real-time and accurate semantic segmentation of traffic scenes*, IEEE Transactions on Intelligent Transportation Systems, (2022), pp. 1–13.

[133] I. Papadeas, L. Tsochatzidis, A. A. Amanatiadis, and I. Pratikakis, *Real-Time Semantic Image Segmentation with Deep Learning for Autonomous Driving: A Survey*, Applied Sciences, (2021).

[134] A. Paszke et al., *PyTorch: An Imperative Style, High-Performance Deep Learning Library*, in Advances in Neural Information Processing Systems 32, Curran Associates, Inc., 2019, pp. 8024–8035.

[135] D. Patterson, G. Gibson, and R. Katz, *A case for Redundant Arrays of Inexpensive Disks (RAID)*, ACM SIGMOD Record, 17 (1988).

[136] J. Phillips, *Achieving Safe Autonomous Driving. Available: https://www.ni.com/de-de/perspectives/imminent-trade-offs-for-achieving-safe-autonomous-driving.html*, 2020.

[137] A. Pidurkar, R. Sadakale, and A. K. Prakash, *Monocular camera based computer vision system for cost effective autonomous vehicle*, in 2019 10th International Conference on Computing, Communication and Networking Technologies (ICCCNT), Jul. 2019, pp. 1–5.

[138] P. Pinggera, S. Ramos, S. Gehrig, U. Franke, C. Rother, and R. Mester, *Lost and Found: Detecting Small Road Hazards for Self-Driving Vehicles*, in International Conference on Intelligent Robots and Systems (IROS), 2016.

[139] R. P. K. Poudel, S. Liwicki, and R. Cipolla, *Fast-SCNN: Fast Semantic Segmentation Network*, in 30th British Machine Vision Conference 2019, BMVC 2019, Cardiff, UK, September 9-12, 2019, BMVA Press, 2019, p. 289.

[140] S. Pv, *Beginning unreal engine 4 blueprints visual scripting: Using C++: From beginner to pro*, Apress Media, LLC, 2021.

[141] R. Pérez, F. Schubert, R. Rasshofer, and E. Biebl, *A machine learning joint lidar and radar classification system in urban automotive scenarios*, Advances in Radio Science, 17 (2019), pp. 129–136.

[142] J. Redmon and A. Farhadi, *Yolov3: An incremental improvement*, CoRR, abs/1804.02767 (2018).

[143] J. Riegel, W. Mayer, and Y. van Havre, *FreeCAD (Version 0.20.29603)*. [Online] https://www.freecad.org/, 2002. Accessed: 2023-03-26.

[144] G. Rong, B. H. Shin, H. Tabatabaee, et al., *LGSVL Simulator: A High Fidelity Simulator for Autonomous Driving*, in 23rd IEEE International Conference on Intelligent Transportation Systems, ITSC 2020, Rhodes, Greece, September 20-23, 2020, IEEE, 2020, pp. 1–6.

[145] O. Ronneberger, P. Fischer, and T. Brox, *U-Net: Convolutional Networks for Biomedical Image Segmentation*, in Medical Image Computing and Computer-Assisted Intervention - MICCAI 2015 - 18th International Conference Munich, Germany, October 5 - 9, 2015, Proceedings, Part III, N. Navab, J. Hornegger, W. M. W. III, and A. F. Frangi, eds., vol. 9351 of Lecture Notes in Computer Science, Springer, 2015, pp. 234–241.

[146] M. Rottmann, P. Colling, T. P. Hack, R. Chan, F. Hüger, P. Schlicht, and H. Gottschalk, *Prediction Error Meta Classification in Semantic Segmentation: Detection via Aggregated Dispersion Measures of Softmax Probabilities*, in 2020 IEEE International Joint Conference on Neural Networks (IJCNN), 2020.

[147] M. Rottmann and M. Schubert, *Uncertainty measures and prediction quality rating for the semantic segmentation of nested multi resolution street scene images*, 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), (2019), pp. 1361–1369.

[148] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Learning representations by back-propagating errors*, Nature, 323 (1986), pp. 533–536.

[149] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, *ImageNet Large Scale Visual Recognition Challenge*, International Journal of Computer Vision (IJCV), 115 (2015), pp. 211–252.

[150] K. Rösch, F. Heidecker, J. Truetsch, K. Kowol, C. Schicktanz, M. Bieshaar, B. Sick, and C. Stiller, *Space, time, and interaction: A taxonomy of corner cases in trajectory datasets for automated driving*, in IEEE Symposium on Computational Intelligence in Vehicles and Transportation Systems (IEEE CIVTS), IEEE SSCI, 2022.

[151] A. Savkin, T. Lapotre, K. Strauss, et al., *Adversarial Appearance Learning in Augmented Cityscapes for Pedestrian Recognition in Autonomous Driving*, in 2020 IEEE International Conference on Robotics and Automation, ICRA 2020, Paris, France, May 31 - August 31, 2020, IEEE, 2020, pp. 3305–3311.

[152] M. Schneider, *Automotive radar: Status and trends*, in In Proceedings of the German Microwave Conference GeMIC 2005, 2005, pp. 144–147.

[153] M. Schubert, K. Kahl, and M. Rottmann, *Metadetect: Uncertainty quantification and prediction quality estimates for object detection*, CoRR, abs/2010.01695v2 (2020).

[154] S. Shalev-Shwartz and S. Ben-David, *Understanding Machine Learning - From Theory to Algorithms*, Cambridge University Press, 2014.

[155] D. Shinar, *Traffic Safety and Human Behavior*, Emerald Group Publishing, 2017.

[156] Shinners, Pete, *Pygame*. [Online] https://www.pygame.org/news, 2000. Accessed: 2023-02-26.

[157] V. D. Silva, J. Roche, and A. M. Kondoz, *Robust fusion of lidar and wide-angle camera data for autonomous mobile robots*, Sensors, 18 (2018), p. 2730.

[158] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*, J. Mach. Learn. Res., 15 (2014), p. 1929–1958.

[159] Suresh, Harini and Guttag, John, *A framework for understanding sources of harm throughout the machine learning life cycle*, in Equity and Access in Algorithms, Mechanisms, and Optimization, EAAMO '21, New York, NY, USA, 2021, Association for Computing Machinery.

[160] T. D. Sweeney, *Unreal Engine*. [Online] https://www.unrealengine.com, 1998. Accessed: 2023-03-26.

[161] R. Szeliski, *Computer Vision - Algorithms and Applications*, Texts in Computer Science, Springer, 2011.

[162] The Autoware Foundation, *Autoware - the world's leading open-source software project for autonomous driving*. [Online] https://github.com/autowarefoundation/autoware, 2016. Accessed: 2023-02-08.

[163] The Linux Foundation, *Open Neural Network Exchange (ONNX) - Open standard for machine learning interoperability*. [Online] https://onnx.ai/, 2017. Accessed: 2023-02-21.

[164] Torvalds, Linus, *Git*. [Online] https://git-scm.com/, 2005. Accessed: 2023-02-25.

[165] P. Tumas, A. Nowosielski, and A. Serackis, *Pedestrian Detection in Severe Weather Conditions*, IEEE Access, 8 (2020), pp. 62775–62784.

[166] S. Uhlemeyer, M. Rottmann, and H. Gottschalk, *Towards Unsupervised Open World Semantic Segmentation*, in Conference on Uncertainty in Artificial Intelligence (UAI), 2022.

[167] United Nations, *World Urbanization Prospects - The 2018 Revision*, 2018.

[168] Unity Technologies, *Unity Game Engine*. [Online] https://www.unity.com, 2005. Accessed: 2023-03-26.

[169] J. Wang, L. Zhang, Y. Huang, J. Zhao, and F. Bella, *Safety of autonomous vehicles*, Journal of Advanced Transportation, 2020 (2020), pp. 1–13.

[170] Z. Wang, Y. Wu, and Q. Niu, *Multi-sensor fusion in automated driving: A survey*, IEEE Access, 8 (2020), pp. 2847–2868.

[171] C. D. Wickens, *Engineering Psychology and Human Performance*, Routledge, Taylor & Francis Group, London, UK, 2016.

[172] C. Windeck, *Der c't-Speicher-Guide - Der optimale Massenspeicher für Ihre Anwendung: ein Leitfaden (in german)*, c't, 7 (2022), pp. 16–19.

[173] J. Wu et al., *Human-in-the-loop deep reinforcement learning with application to autonomous driving*, CoRR, abs/2104.07246 (2021).

[174] T.-E. Wu, C.-C. Tsai, and J.-I. Guo, *LiDAR/camera sensor fusion technology for pedestrian detection*, in 2017 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC), 2017, pp. 1675–1678.

[175] Y. Xiao, A. Jiang, J. Ye, and M.-W. Wang, *Making of Night Vision: Object Detection Under Low-Illumination*, IEEE Access, 8 (2020), pp. 123075–123086.

[176] Y. Xiong, R. Liao, H. Zhao, R. Hu, M. Bai, E. Yumer, and R. Urtasun, *UPSNet: A Unified Panoptic Segmentation Network*, in 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2019, pp. 8810–8818.

[177] W. Xu, *User-centered design (iv): Human-centered artificial intelligence*, Journal of Applied Psychology, 25 (2019), pp. 291–305.

[178] W. Xu et al., *Transitioning to human interaction with ai systems: New challenges and opportunities for hci professionals to enable human-centered ai*, International Journal of Human-Computer Interaction, (2022), pp. 1–25.

[179] Yarotsky, Dmitry, *Error bounds for approximations with deep ReLU networks*, Neural Networks, 94 (2016).

[180] C. Yu, C. Gao, J. Wang, G. Yu, et al., *BiSeNet V2: Bilateral Network with Guided Aggregation for Real-Time Semantic Segmentation*, International Journal of Computer Vision, 129 (2021), pp. 3051–3068.

[181] M. D. Zeiler, D. Krishnan, G. W. Taylor, and R. Fergus, *Deconvolutional networks*, 2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, (2010), pp. 2528–2535.

[182] H. Zhao, J. Shi, X. Qi, X. Wang, and J. Jia, *Pyramid Scene Parsing Network*, in 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017, pp. 6230–6239.

[183] J.-Y. Zhu et al., *Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks*, in 2017 IEEE International Conference on Computer Vision (ICCV), 2017, pp. 2242–2251.