Department of Informatics
Technical University of Munich

Master's Thesis in Informatics

A Neural Network-based Scenario Detection Framework for Road Perception

Ein auf Neuronalen Netzen Basierendes Szenario-Erkennungssystem für die Straßenwahrnehmung

Supervisor Prof. Dr.-Ing. habil. Alois C. Knoll

Advisor Walter Zimmer, M.Sc.
Frank Keidel

Author Ugurcan Polat

Date May 15, 2022 in Garching

Disclaimer

I confirm that this Master's Thesis is my own work and I have documented all sources and material used.

Garching, May 15, 2022

(Ugurcan Polat)

Abstract

Autonomous driving has been an active research area for many years to achieve safer and more comfortable transportation [26]. Road perception is at the core of autonomous driving since Advanced Driver-Assistant System functions rely on reliable sensing of the environment. One key issue in road perception is the detection of road lanes in different scenarios such as lane splits. For this purpose, a neural network-based scenario detection framework for detection lane split scenarios using time series data is developed. In the framework, three neural network models are experimented: Multi-Layer Perceptron, Convolutional Neural Network and self-attention based neural network. The solution also includes an end-to-end training pipeline for the developed models to provide flexibility in model development.

Another aspect in autonomous driving is the Intelligent Transportation Systems (ITS) which includes traffic scenario understanding on the road infrastructure and infrastructure-to-vehicle communication to improve the perception of the automated vehicles. One important topic in such systems is the detection of scenarios. Therefore, a head-to-tail collision detection method is proposed as part of this work. Furthermore, scenario detection for ITS is extended with near real-time capability.

Zusammenfassung

Das autonome Fahren ist seit vielen Jahren ein aktives Forschungsgebiet mit dem Ziel, den Verkehr sicherer und komfortabler zu machen [26]. Die Straßenwahrnehmung ist das Kernstück des autonomen Fahrens, da die Funktionen fortschrittlicher Fahrerassistenzsysteme auf einer zuverlässigen Erfassung der Umgebung beruhen. Ein zentrales Problem bei der Straßenwahrnehmung ist die Erkennung von Fahrspuren in verschiedenen Szenarien, wie z.B. bei Spurwechseln. Zu diesem Zweck wird ein auf neuronalen Netzen basierender Rahmen für die Erkennung von Fahrspurwechsel-Szenarien anhand von Zeitreihendaten entwickelt. In diesem Rahmen werden drei neuronale Netzwerkmodelle erprobt: Multi-Layer Perceptron, Convolutional Neural Network und ein auf Selbstbeobachtung basierendes neuronales Netz. Die Lösung umfasst auch eine End-to-End-Trainings-Pipeline für die entwickelten Modelle, um Flexibilität bei der Modellentwicklung zu gewährleisten.

Darüber hinaus gibt es im Bereich des autonomen Fahrens Intelligente Verkehrssysteme, die das Verstehen von Verkehrsszenarien auf der Straßeninfrastruktur und die Kommunikation zwischen Infrastruktur und Fahrzeug umfassen, um die Wahrnehmung der automatisierten Fahrzeuge zu verbessern. Ein wichtiges Thema in solchen Systemen ist die Erkennung von Szenarien. Daher wird im Rahmen dieser Arbeit eine Methode zur Erkennung von Kollisionen von vorne nach hinten vorgeschlagen. Darüber hinaus wird die Szenarienerkennung für intelligente Verkehrssysteme um die Fähigkeit erweitert, nahezu in Echtzeit zu fahren.

Acknowledgments

I am grateful to my academic advisor for the knowledge he shared and his invaluable patience and feedback.
I also would like to express my gratitude to my corporate advisor who generously provided knowledge and expertise.
I must express my very profound gratitude to my family. I could not have undertaken this journey without their belief in me which has kept my spirits and motivation high during this process.
I am also thankful to my classmates and friends for inspiring me and the moral support.
Additionally, I had the pleasure of working in collaboration with BMW Group AG which I am thankful for all the resources and knowledge support.
Finally, many thanks to all, who directly or indirectly, supported this work.

Contents

List of Abbreviations **xi**

1 Introduction **1**
1.1 BMW Group AG . . . 1
1.2 Providentia++ . . . 1
1.3 Problem Statement . . . 2
1.4 Contributions . . . 2
1.4.1 Contributions for Automated Vehicles . . . 3
1.4.2 Contributions for the Providentia++ project . . . 3

2 Theoretical Background **5**
2.1 Lane Boundary Polynomial Model . . . 5
2.2 Corner Case Detection . . . 5
2.3 Artificial Neural Network (ANN) . . . 6
2.4 Multi-Layer Perceptron (MLP) . . . 6
2.5 Convolutional Neural Network (CNN) . . . 7
2.6 Self-attention . . . 7
2.7 Robot Operating System (ROS) . . . 7
2.8 Time-series Data . . . 8
2.9 Trajectory Data . . . 8
2.10 Kubeflow . . . 9

3 Related Work **11**
3.1 Road Perception . . . 11
3.2 Corner Scenario Detection in Autonomous Driving . . . 12
3.3 Outliers in Time-Series Data . . . 12
3.3.1 Outlier Types . . . 13
3.3.1.1 Point Outliers . . . 13
3.3.1.2 Subsequence Outliers . . . 13
3.3.1.3 Outlier Time Series . . . 14
3.3.2 Outlier Detection Methods . . . 14
3.3.3 Neural Network-based Outlier Detection . . . 14
3.4 Providentia++ . . . 14
3.4.1 Scenario Dataset Creation Framework . . . 15
3.4.2 Maneuver Detection . . . 16
3.4.2.1 Maneuver Types . . . 17
3.4.2.2 Maneuver: Lane Change Detection . . . 17
3.4.2.3 Maneuver: Cut-in and Cut-out Detection . . . 17
3.4.2.4 Maneuver: Speeding and Standing Vehicle Detection . . . 18
3.4.2.5 Maneuver: Tailgate . . . 18

4 Solution Approach **19**
4.1 Neural Network Training Pipeline for Lane Split Detection . . . 19
4.1.1 Overview of the Pipeline . . . 19
4.1.1.1 Pipeline File Definition . . . 21
4.1.1.2 Pipeline Parameters . . . 21
4.1.1.3 Communication Between Components . . . 22
4.1.1.4 Kubernetes Persistent Volumes and Persistent Volume Claims 23
4.1.1.5 Kubernetes Secret . . . 23
4.1.1.6 Definition of Components . . . 23
4.1.2 Data Selection . . . 24
4.1.3 For Loop . . . 24
4.1.4 Signal Extraction . . . 25
4.1.5 Auto-Labeling . . . 25
4.1.5.1 Semantic Road Points . . . 26
4.1.5.2 Timestamp Labeling . . . 27
4.1.5.3 Output Label Description . . . 28
4.1.6 Dataset Preparation . . . 29
4.1.7 Training . . . 31
4.1.7.1 Training on Single Sample . . . 32
4.1.7.2 Multi-Layer Perceptron . . . 32
4.1.7.3 Training on Time Windows . . . 34
4.1.7.4 Generation of Time Windows . . . 34
4.1.7.5 Convolutional Neural Network . . . 34
4.1.7.6 Self-Attention based Neural Network . . . 35
4.1.7.7 Data Standardization . . . 36
4.1.7.8 Class Weights . . . 36
4.1.7.9 Event Files Balancing . . . 37
4.1.8 Training Run Tracking . . . 37
4.2 Providentia++ . . . 37
4.2.1 Scenario Description Format . . . 37
4.2.2 Accident Detection . . . 38
4.2.3 Scenario Detection on the Live System . . . 40
4.2.3.1 Logging to the Live System . . . 42

5 Evaluation & Analysis **45**
5.1 Metrics . . . 45
5.1.1 Confusion Matrix . . . 45
5.1.2 Accuracy . . . 46
5.1.3 Precision . . . 46
5.1.4 Recall . . . 46
5.1.5 Execution Time . . . 46
5.2 Lane Split Detection . . . 47
5.2.1 Dataset . . . 47
5.2.1.1 Simulation Data . . . 48
5.2.1.2 Real-world Data . . . 48
5.2.2 Results and Analysis . . . 48
5.2.2.1 Model Performance . . . 49
5.2.2.2 Inference Time Measurements . . . 51
5.3 Providentia++ . . . 51
5.3.1 Data . . . 51
5.3.1.1 Accident Recording . . . 52

5.3.1.2 Regular Traffic Recording 52
5.3.2 Results and Analysis 52
5.3.2.1 Accident Detection 52
5.3.2.2 Live System Support 52

6 Outlook **55**
6.1 Neural Network 55
6.2 End-to-end Training 55
6.3 Online Scenario Detection 56

7 Conclusion **57**

Bibliography **63**

List of Abbreviations

AD Autonomous Driving

ADAS Advanced Driver-Assistance System

AE Autoencoder

ANN Artificial Neural Network

AVs Automated Vehicles

BMVI German Federal Ministry of Transport and Digital Infrastructure

BMW Group AG Bayerische Motoren Werke Group Aktiengesellschaft

CNN Convolutional Neural Network

CSV Comma-Separated Values

DAG Directed Acyclic Graph

EU European Union

FN False Negative

FP False Positive

HTTP Hypertext Transfer Protocol

ITS Intelligent Transportation System

JSON JavaScript Object Notation

kph kilometers per hour

LiDAR Light Detection and Ranging

MLP Multi-Layer Perceptron

NN Neural Network

Providentia Proactive Video-Based Use of Telecommunication Technologies in Innovative Traffic Scenarios

PV Persistent Volume

PVC Persistent Volume Claim

radar Radio detection and ranging

ROS Robot Operating System

SDK Software Development Kit

TN True Negative

TP True Positive

TTC Time-To-Collision

TUM Technical University of Munich

URI Uniform Resource Identifier

V2I Vehicle-to-Infrastructure

YAML Yet Another Markup Language

Chapter 1

Introduction

This thesis is written in cooperation with one of the leading premium car manufacturers [55], BMW Group AG, and Autonomous Driving (AD) project Providentia++.

1.1 BMW Group AG

Bayerische Motoren Werke Group Aktiengesellschaft (BMW Group AG) is a corporation based in Germany that produces premium segment cars and motorcycles under the brands BMW, MINI, Rolls-Royce, and BMW Motorrad. The corporation is the leading luxury car manufacturer and it is investing heavily in autonomous driving hardware and software development in order to have advanced Automated Vehicles (AVs) in the increasingly competitive market [8]. Currently, its car portfolio equips range of Advanced Driver-Assistance System (ADAS) features including but not limited to lane-change assist, lane-keep assistant, collision avoidance systems, etc. that rely on the sensors mounted on the cars such as camera, Light Detection and Ranging (LiDAR), Radio detection and ranging (radar).

BMW Group AG is one of the partners in many European Union (EU) funded autonomous driving research projects such as EU-funded AdaptIVe, HI-DRIVE and L3Pilot [2, 28, 51] and it is a major contributor to the autonomous driving research area and tools [4, 14, 50] such as ROS.

1.2 Providentia++

The autonomous driving project funded by German Federal Ministry of Transport and Digital Infrastructure (BMVI), Providentia++ aims to improve the safety of the traffic participants and reduce traffic congestion by complementing the sensors on the automated vehicles on the road. Providentia++ builds on and extends the original Proactive Video-Based Use of Telecommunication Technologies in Innovative Traffic Scenarios (Providentia) project [47] that lasted from 2017 to 2019 where the cameras, radar and LiDAR sensors are stationed on the overheads of the highway signs to detect the traffic participants and create digital version of the world called "digital twin" based on these detections. The autonomous vehicles travelling on the road can use the digital twin information that can be obtained from the sensor systems on the road using 5G connection in order extend the perception where it can be limited due to sensor limitations of the car [68].

The Providentia++ project consortium is lead by the Chair of Robotics, Artificial Intelligence, and Real-time Systems at the Technical University of Munich (TUM) and includes

many partners from the industry such as fortiss, Valeo, Intel, Cognition Factory, Elektrobit, Siemens, Volkswagen, brighter AI, 3D Mapping Solutions, and Huawei [69].

1.3 Problem Statement

In the recent years, autonomous driving technology has been one of the key focus of the vehicle industry, and extensively studied by the researchers [26]. The automated driving systems must be able to understand their environment by detecting the driving space of the vehicle through detecting the lanes, objects or obstacles in the environment in order to replace human drivers in the future for safer and more comfortable transportation. Therefore, road perception or driving space detection is a crucial component of the automated vehicles. Road perception is usually achieved by using the map information, cameras mounted in the front of the vehicle and range of sensors such as LiDAR and radar [41], and as well as additional information such as Intelligent Transportation System (ITS). The automated car creates an internal representation of the environment based on the detected driving space.

The developments in the data-driven approaches such as neural networks and the rapid advances in the hardware have led to more advanced automated driving functions, for instance, lane keeping systems that rely on lane detection [46]. Even though the detection of lanes with traditional methods is highly accurate in the ideal conditions, scenarios such as merging or splitting lanes, and as well as lane changing are still challenging to detect, especially in the road perception domain where computing power is mostly limited to the CPU of the vehicles [62]. The road perception can be improved by detecting such scenarios and performing filtering on the lane detection.

The detection of lane split scenarios is challenging due to the variety of the situations and environment conditions in the real-world and it is crucial in the road perception area since it is demanded by automated features such as lane centering which helps keeping the car in the center of the lane [39]. In this work, a neural-network based lane split scenario detection method that run on the sensor data of a BMW prototype car is introduced.

Scenario detection for car accidents or traffic jams through the cross-vehicle ITS such as Providentia++ would be an important information for the automated vehicles on the road where it might not be possible to detect such scenarios solely based on the sensors of the car. Another aspect of the scenario detection field is scenario-based safety testing. In order to test the automated vehicles for as many different scenarios as possible, a variety of scenarios such as cars overtaking, standing cars on the road, cut-ins, etc. must be detected and the data that contains such scenarios would be test-cases in the testing of the autonomous vehicles to ensure the safety of such systems. It is also important to detect such scenarios in real-time to improve road safety through notification of the traffic participants in case of emergencies. Moreover, detection of different scenarios where there is a traffic congestion or many traffic participants do not obey the traffic rules could also contribute to take a step to improve those situations if there is a pattern. Therefore, real-time scenario detection support for various vehicle maneuvers such as lane changes, cut-in, cut-out and tailgate events is implemented in the Providentia++ project in this work alongside with the detection of accident detection.

1.4 Contributions

The contributions in this work contain a scenario detection framework in two different domains which are BMW prototype automated vehicle and Providentia++ where there are

static sensor stations on the highways.

1.4.1 Contributions for Automated Vehicles

The neural network-based scenario detection framework for automated vehicles implemented in this work consists of three parts: auto-labeling, neural-network models and a training pipeline.

1. Auto-labeling

   Training a neural network requires a labeled dataset since the introduced training methods are all supervised learning methods. Having a manual-labeling solution is not enough to achieve decent results since it requires extensive human labor to create a large dataset that covers variety of scenario situations.

2. Neural-network model

   In this work, three different neural-network models, namely Multi-Layer Perceptron (MLP), Convolutional Neural Network (CNN), and the self-attention based network, are implemented. Each neural network model type can be configured to be used in the training pipeline.

3. Training pipeline

   An end-to-end training pipeline that can be run periodically with a dataset getting larger in time is an important aspect to avoid having a neural network model that is unable to capture variety in the data.

   The pipeline includes each step that is required to generate a trained model. It first extracts sensor signal data for a list of data that is collected by the prototype cars, then performs auto-labeling on the extracted signals, generates training data, and finally trains a neural network model on the generated dataset. The pipeline runs can easily be configured to test different hyperparameters.

1.4.2 Contributions for the Providentia++ project

The neural network-based scenario detection framework for automated vehicles implemented in this work consists of two parts: live system support for the existing rule-based method and implementation of the detection of accident detection.

1. Live system support

   The scenario and maneuver detection system introduced in the Providentia++ project [44] does not support scenario detection in the live system when the detection relies on previous detections. The existing rule-based solution for the scenario and maneuver detection is extended to support running on live system with fixed-time detection caching.

2. Maneuver Detection

   The scenario detection support is extended with accident detection that uses rule-based solutions.

Chapter 2

Theoretical Background

The theoretical topics that found the pillars of the thesis work are described in this section.

2.1 Lane Boundary Polynomial Model

In the context of road perception, lane detection is an integral task to determine the driving space of an automated vehicle. The detected lane boundaries can be modeled in different ways such as a set of points or polynomials. The polynomial modelling of a lane boundary is shown in the Equation 2.1 where K is the order of the polynomial, $\mathcal{P} = \{a_0, ..., a_K\}$ is the set of polynomial coefficients [76].

$$p(y) = \sum_{k=0}^{K} a_k y^k \tag{2.1}$$

Figure 2.1: Lane boundary polynomials projected on the road image [76]

In Figure 2.1, lane boundaries modeled as third-order polynomials are projected onto the road image as green lines.

2.2 Corner Case Detection

Corner case definition varies in the literature. As stated by Heidecker et. al [37], corner cases mean deviations from "normal" in broad terms which overlap with definitions of *outliers*

which are huge enough deviations in an observation from other observations so that it can be considered as generated by another system, *novelties* that appear as samples not seen yet, and *anomalies* which can be defined [21] as patterns not consistent with normal behavior.

In the automated driving area, detection of corner cases or scenarios is crucial for each phase of the data processing toolchain. It is been shown in [37] that corner case detection is a valuable information for the driving function since it is an addition to the environment representation generated through perception. The detected corner cases can also be used as test cases for the further development of the automated driving functions.

2.3 Artificial Neural Network (ANN)

Artificial Neural Network (ANN) is a type of machine learning model that is inspired by the neural networks in the brains of animals [43]. ANN is a very popular research field in many disciplines such as classification, clustering, pattern recognition and prediction [1]. Similar to biological neural networks in design, a neuron in an ANN receives input data, processes the data by multiplying with a *weight* and passes the aggregated output to the next neuron. The "learning" is achieved through processing an input and updating the *weight* of each characteristic *feature* in the input data that define the importance of that particular *feature* in making a prediction by the factor of the error in the predictions represented as a *loss* function. An *optimizer* algorithm updates the attributes such as *weights* to achieve a lower *loss* value; therefore, better "learning" [83]. "Learning" processes can be controlled by changing the parameters that are called *hyperparameters* such as number of *layers* that are a group of neurons operating together at the same depth and *learning rate* which is the rate of change that affect updating *weights* based on the *loss* value [49]. A neuron in the ANN multiplies *features* in the data with the *weights* but the result is aggregated through an *activation function* that decides whether or not the neuron should pass the value to the next neuron. If each neuron in a *layer* multiplies each input with a *weight*, the *layer* is called *fully-connected* [25].

There are many types of ANNs including but not limited to perceptron, feed-forward network, radial basis network, recurrent neural network (RNN), long-short term memory (LSTM), autoencoder (AE), convolutional neural network (CNN) and generative adversarial network (GAN).

2.4 Multi-Layer Perceptron (MLP)

Perceptron is a simple model used in binary classification tasks where the output can only have two values, 0 or 1. The perceptron model algorithm is shown in Equation 2.2 where x is the input vector, w is the weight vector, b is the scalar-valued bias term and τ is the threshold. If the dot product of w and x combined with the bias term b is larger than the threshold τ, the perceptron outputs 1, otherwise 0 [64].

$$f(x) = \begin{cases} 1 & \text{if } w \cdot x + b > \tau \\ 0 & \text{otherwise} \end{cases} \tag{2.2}$$

However, the perceptron model is only capable of learning patterns in linear data by nature. In an MLP, there are multiple layers of perceptrons or simply multiple *fully-connected layers* and non-linear *activation function* after some neurons that allow it to learn from non-

linear data. MLPs can work on multi-class problems by having a perceptron for each class in the last layer [71].

2.5 Convolutional Neural Network (CNN)

Convolutional neural networks or CNNs are similar to MLPs but layers are replaced by convolutional layers. It is developed for image classification tasks but it can also be used for simple classification tasks. It differs from MLP by having convolutional layers that receive input from a limited number of neurons of the previous layer that form the *receptive field* whereas in MLP with fully-connected layers each neuron takes the input from the all neurons in the previous layer [65]. This allows CNNs to consider larger area in the input data since convolution is performed on a single data value in the input multiple times [5]. Since spatial relations of features can be "learned" through convolutional layers, CNNs perform better than MLPs if the location of a value in the input data is an important factor to determine the output [48]. Thus, CNNs are suitable for time-series data classification where relation between the data points for different timestamps is useful.

2.6 Self-attention

Attention is a mapping from a query and set of key-value pairs to an output [81]. The general idea is to put emphasis on the areas in the input where the areas are important. Attention has become an essential part of the approaches [60, 75] on sequential data tasks since it can learn the dependencies regardless of the distance to input or output sequences [81] whereas CNNs can only consider a fixed distance.

Self-attention is an attention mechanism relating different positions of a single sequence in order to compute a representation of the sequence [81]. In other words, self-attention is an attention implementation where the key and the value are derived from the same sequence in the attention function.

2.7 Robot Operating System (ROS)

Robot Operating System (ROS) is an open-source framework that includes a set of tools to enable robotics software development through re-using code and extending the functionality without knowing how the hardware works. As opposed to what the name suggests, ROS is neither an operating system nor only limited to robotics development. It is a meta-operating system that provides services similar to what an operating system offer such as device drivers, hardware abstraction, communication protocol over multi-device systems and visualization systems [70]. The services that ROS provide do not only provide a groundwork for robotics software development but they also allow ROS to be used as a communication layer for a system with multiple peripheral hardware devices.

The basic concepts that define the pillars of ROS works can be listed as *packages*, *nodes*, *messages*, *topics*, and *bags* [70]. *Packages* are the main units in the file system for organizing software in ROS and they can contain multiple *nodes* which are the processes where the computation is performed, and each *package* can be considered as a software module [70]. *Nodes* communicate with each other by consuming or producing *messages* which are a strictly typed data structure [70]. As it can be seen in Figure 2.2, *Messages* are consumed through

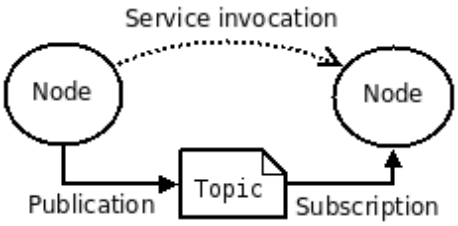


Figure 2.2: *Topic* based publish/subscribe communication model [72]

so-called *topic* subscriptions where a *node* that is needed to consume a certain type of data such as "odometry" or "map" subscribes to the certain topic to consume the data published by another *node* [70]. *Bags* are a data storing mechanism that allows saving and playing back *messages* [72].

2.8 Time-series Data

Time-series data is a data type that consists of sequentially collected samples over the time domain. Each data point is distinguished by a timestamp; therefore, time-series data is immutable. Time-series data can be *univariate* if only one variable is observed at a time. An example to *univariate* time-series is data collected from the odometry sensor of a car. Moreover, time-series can be *multivariate* where multiple variables are recorded over time. For instance, three-axis accelerometer sensor of a car that collects information for *yaw*, *pitch* and *roll* over time. In Figure 2.3, an example to multivariate sensor data of speed, yaw rate and acceleration are visualized as a time graph can be seen.

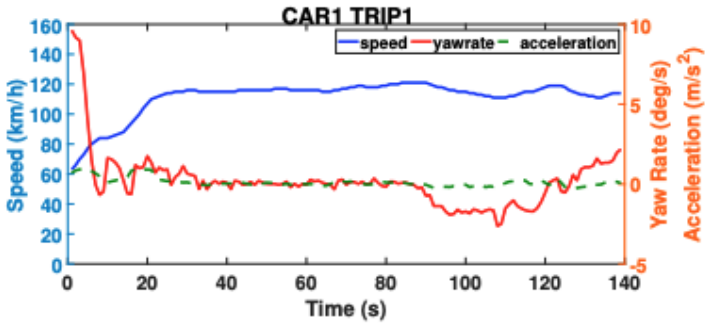


Figure 2.3: Time graph sensor data of a vehicle [38]

In the Equation 2.3, *univariate* time series T is modeled as an ordered sequence of n real-valued variables [30].

$$T = (t_1, ..., t_n), t_i \in \mathbb{R} \tag{2.3}$$

2.9 Trajectory Data

A trajectory is the path of a moving object over time. The trajectory data is a time-series data represented by a series of spatial coordinates and a timestamp such as $p = (x, y, t)$ in the

2D coordinate system where x and y are the coordinate values and t is the timestamp [87]. Trajectory data can be useful for urban planning to avoid congestion where recordings of the cars travelling across the city as well as motion planning of automated cars [84].

2.10 Kubeflow

Kubeflow [10] is a machine learning platform that enables developing, deploying and managing multi-stage machine learning workflows in a toolkit called Kubeflow Pipelines. Kubeflow is based on a container orchestration framework called Kubernetes [11] and can be run on any Kubernetes cluster regardless of the cluster type such as local single machine clusters or remote clusters with multiple servers for distributed workflows.

Kubeflow Pipelines provide a set of tools to enable automated and distributed model training defined as an end-to-end workflow divided into specific containerized stages such as data preparation, data augmentation, training and evaluation that communicate with each other through the file system or log output [34]. Moreover, Kubeflow has a hyper-parameter tuning tool with neural architecture search called Katib [33].

The end-to-end neural network model training pipeline implemented in this work is a Kubeflow pipeline. Kubeflow is selected based on the features that it provides such as its flexibility in development, scalability, availability, ease of use and versioning of the neural network models [35].

Chapter 3

Related Work

3.1 Road Perception

Perception is an integral part of achieving an automated driving system that is safe and reliable. In Figure 3.1, an overview of the process of automated vehicle control is shown. The process of a controlling vehicle in the environment starts with the perception module that perceives and monitors the environment through a range of sensors such as cameras, LiDAR and radar [57, 79]. The localization and mapping module determines the global and local position of the automated vehicle, and maps the environment of the vehicle from the sensor data and offline maps [58]. The path planning module generates the possible safe routes using the information generated from the previous modules, and then decision making module calculates the optimal route [58]. The vehicle control module is responsible for controlling the vehicle by calculating and using the appropriate vehicle commands to follow the optimal route found by the decision making module [79]. This shows that the information from the perception module should always be reliable for an automated vehicle to safely navigate.

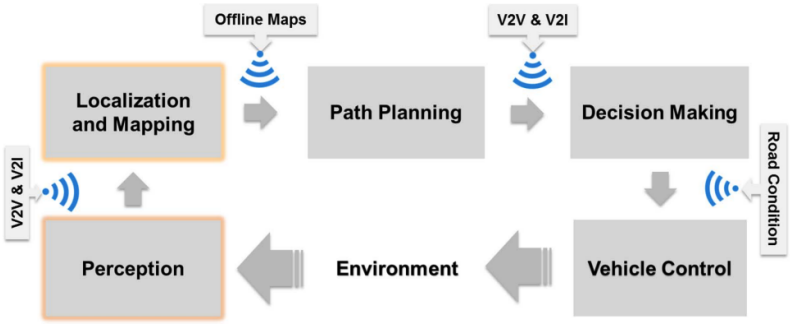
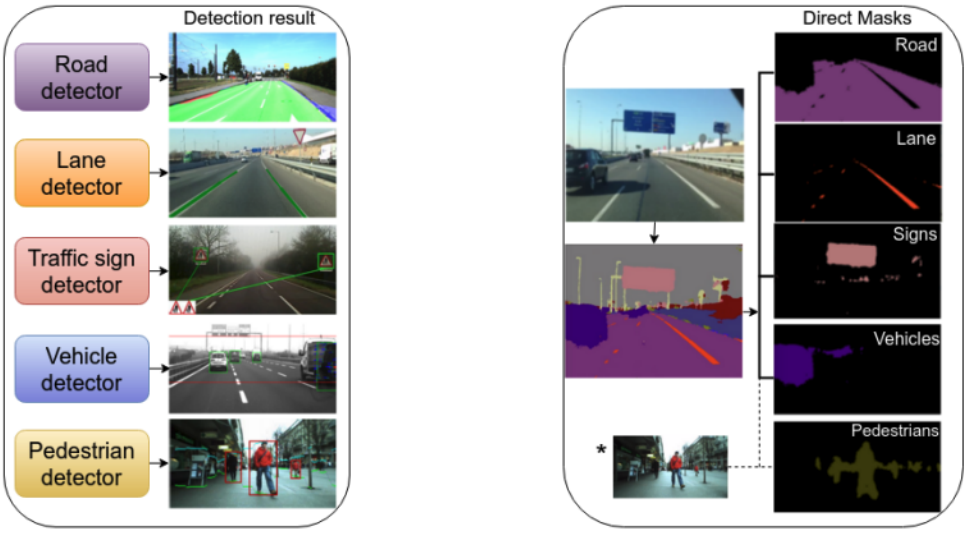


Figure 3.1: Overview of automated driving navigation architecture [79]

Perception module with separate detectors consists of sub-modules where each has a different purpose such as detecting roads, lanes boundaries, traffic signs, vehicles and pedestrians whereas a one-shot detector produces output for roads, lanes and signs, etc. in a single unified system [73]. An example perception module with separate detectors is shown in Figure 3.2a and a unified system is shown in Figure 3.2b.

Vehicle-to-Infrastructure (V2I) communication could provide a reliable information to the AVs regarding the road layout changes, accident situations, where the vehicle's sensor range may not be enough to "see" and detect in different scenarios [15, 36]. The ITS deployed on the roads such as Providentia [47] could be a useful information source for the perception modules of the automated vehicles.



(a) Perception module with multiple detectors

(b) Perception module with unified detector

Figure 3.2: Examples of perception module types [73]

3.2 Corner Scenario Detection in Autonomous Driving

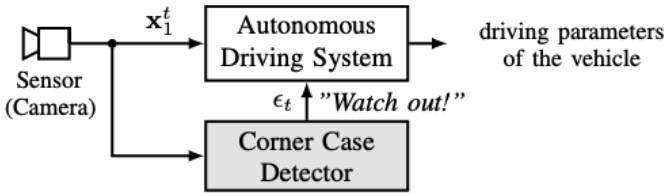


Figure 3.3: Corner case detection system [18]

Detection of corner cases that appear infrequently is critical in an automated driving system to ensure the safety of traffic participants. In Figure 3.3, integration of a corner case detection system to the autonomous driving system is illustrated. The system can support the autonomous driving system by providing self-awareness and criticality measures for the perception module so that the automated vehicle plans its motion accordingly [18]. The corner case detection task is similar to outlier detection tasks on the time-series data domain which this work covers.

For instance to the corner cases, lane merging and lane splitting scenarios are listed as corner cases of lane detection task of the perception modules of autonomous vehicles [46, 52, 54] since they can cause large deviations from the expected lane representation. In the lane merging scenario, the right or left-most lane merges into the lane next to it. Similarly, a new lane splits from the right or left most lane in lane splitting scenario.

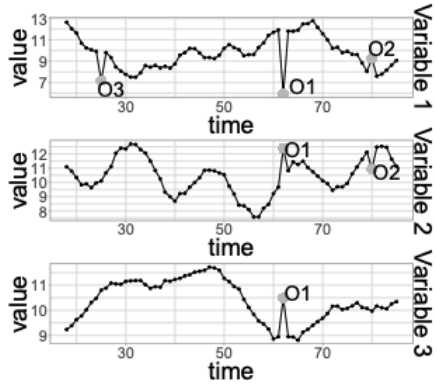
3.3 Outliers in Time-Series Data

Time-series data is a collection of data points ordered based on time and this type of data has been extensively collected, researched and used in various areas such as finance [7, 42, 56], medicine [6, 63, 78] to transportation [29, 82, 85]. Predicting future behaviors of a system and detecting outliers in the data relies on analyzing time series data [45]. In the domain

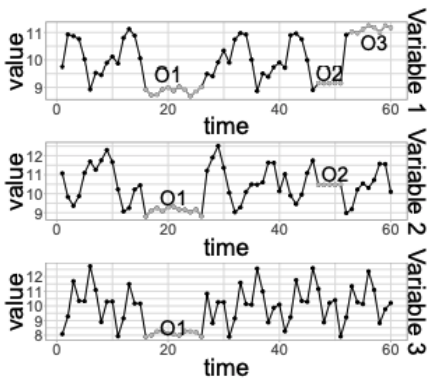
of AD, multivariate time series data is collected from the sensors of the AVs; for instance, speed information from the accelerometer sensor and lane detection polynomial coefficients used in the perception module of the AVs. Detecting anomalies or outliers in those systems is an important research area [17, 19]. There are different types of outliers in the time-series data that depend on the data points affected by the outlier; thus, there are different types of outlier detection methods based on the time series and outlier types [16]. The outlier types and detection methods are briefly reviewed in the following section, Section 3.3.1.

3.3.1 Outlier Types

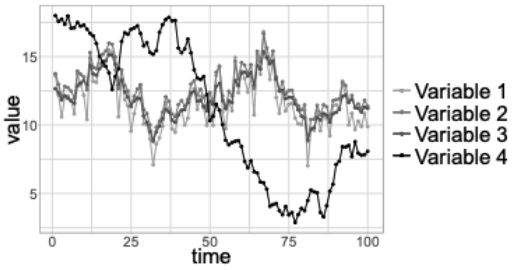
As stated in [16], outliers can be divided into three categories: *point outliers*, *subsequence outliers* and *outlier time series*.



(a) Multivariate time series with point outliers



(b) Multivariate time series with subsequence outliers



(c) Multivariate time series with time series outlier

Figure 3.4: Examples of outlier types [16]

3.3.1.1 Point Outliers

Point outlier is an outlier type that affects a specific time step in the data [16]. A point outlier can be univariate if it affects a single variable in the time series or multivariate if it affects multiple variables in the time series data. A point outlier is considered a local outlier if it deviates from the neighboring data points whereas it is considered a global outlier if it deviates from the other values in the time series [16]. In Figure 3.4a, *O1* and *O2* are multivariate point outliers whereas *O3* is a univariate point outlier.

3.3.1.2 Subsequence Outliers

Subsequence outlier is an outlier type that affects subsequent time points in a time series; however, a specific time point in the subsequence outlier may not be a point outlier [16].

Similar to point outliers described in Section 3.3.1.1, a subsequence outlier can be a univariate or multivariate outlier depending on the affected variable. In Figure 3.4b, *O1* and *O2* are multivariate point outliers whereas *O3* is a univariate subsequence outlier.

3.3.1.3 Outlier Time Series

An outlier time series occurs when the entire time series of a variable in the multivariate time series input is an outlier [16]. In Figure 3.4c, *Variable 4* is an outlier time series.

3.3.2 Outlier Detection Methods

Detection of outliers depends on the outlier type and characteristics. Most of the studies that focus on the detection of point outliers are traditional methods such as Bayesian filtering such as Kalman filters [77] or k-means clustering [86]. Those methods do not cover or perform well on the subsequence outliers or outlier time series [16].

3.3.3 Neural Network-based Outlier Detection

The diverse types of outliers, the infrequency of outliers in a time series and unknownness in the nature of the outliers make the detection of outliers with traditional methods difficult [66]. Therefore, the ability of generalization of the data puts neural networks in focus for outlier detection tasks [66].

According to Pang et al. in [66], neural network-based outlier or anomaly detection methods can be categorized into three types end-to-end anomaly score learning where the model predicts if a data point at a time step is an anomaly or not based on a score; learning feature representations of normality so that the model can predict the next "normal" value and compare it to the current data point to determine if there is an anomaly.

One method introduced as an example to the learning feature representations of normality type of outlier detection is the usage of Autoencoder (AE) [45]. The idea behind the usage of AE is based on the assumption that outliers cannot be reconstructed when the dense representation generated by the first part of the AE is used to reconstruct input data, therefore, outliers can be detected by comparing the reconstructed and the original input data [45]. Another neural network based method is based on MLP which is a end-to-end anomaly score learning type where the MLP model is trained on the anomaly samples and used to predict the noise value [22].

3.4 Providentia++

Providentia++ is the successor project to the original Providentia ITS system project [47] with an aim to improve the perception of the automated vehicles which have sensors with limited range, therefore the safety of the roads. There are 7 measurement points on the German A9 autobahn and the highway B471 near Munich that cover 3.5 km total length [24]. As it can be seen in Figure 3.5, there are multiple camera, LiDAR and radar sensors stationed on the measurement points of the German A9 highway.

The data coming from the multiple stationary sensors are fused together to generate a unified digital representation of the highway environment called the "digital twin" [47]. The traffic participants such as cars, buses, trucks and pedestrians are then detected with a

Figure 3.5: Providentia sensors stationed on one of the measurement points on the A9 autobahn [23]

machine learning-based object detection algorithm [47]. Recently, an ITS dataset consists of the traffic participants detected on the A9 autobahn is released [24].

3.4.1 Scenario Dataset Creation Framework

Kaeefer introduced a scenario mining, detection, classification and generation framework [44] for the Providentia++ project in his Master's thesis. The stages of the scenario dataset generation are shown in Figure 3.6.

These stages can be summarized as [44]:

1. Scenario Catalog and Analysis of Recorded Data

   Scenario catalog formed of traffic scenario situations that are relevant to and occur on the Providentia++ projects are determined after collecting scenario types from various sources.

   The recorded data is then analyzed based on the selected traffic scenario types to see how frequently a scenario in the scenario catalog occurs on the data.

2. Scenario Generation

   The scenarios that do not occur on the recorded data is artificially generated in the simulation environment

3. Scenario Mining

   In order to automate the scenario labeling, a scenario mining tool is introduced. The tool consists of 5 stages: *Data Extraction* where the raw data is extracted; *Data Preprocessing* where the raw data is processed to extract features that can be used in the maneuver detection; *Data Augmentation* where multiple variations of the extracted data are generated to increase the number of scenarios; *Maneuver Detection* where the vehicle maneuvers are automatically detected based on the trajectory information and the preprocessed data; *Scenario Statistics* where the detected maneuvers are aggregated to record the statistics.

4. Scenario Simulation / Visualization

The driving scenarios are written in a format that is compatible with simulation tools such as CARLA Simulator [27] to be simulated and visualized.

5. Scenario Database

   The scenario files are recorded to generate a scenario database by combining the data recording with labels, scenario statistics, maneuver labels, and simulation compatible file format.

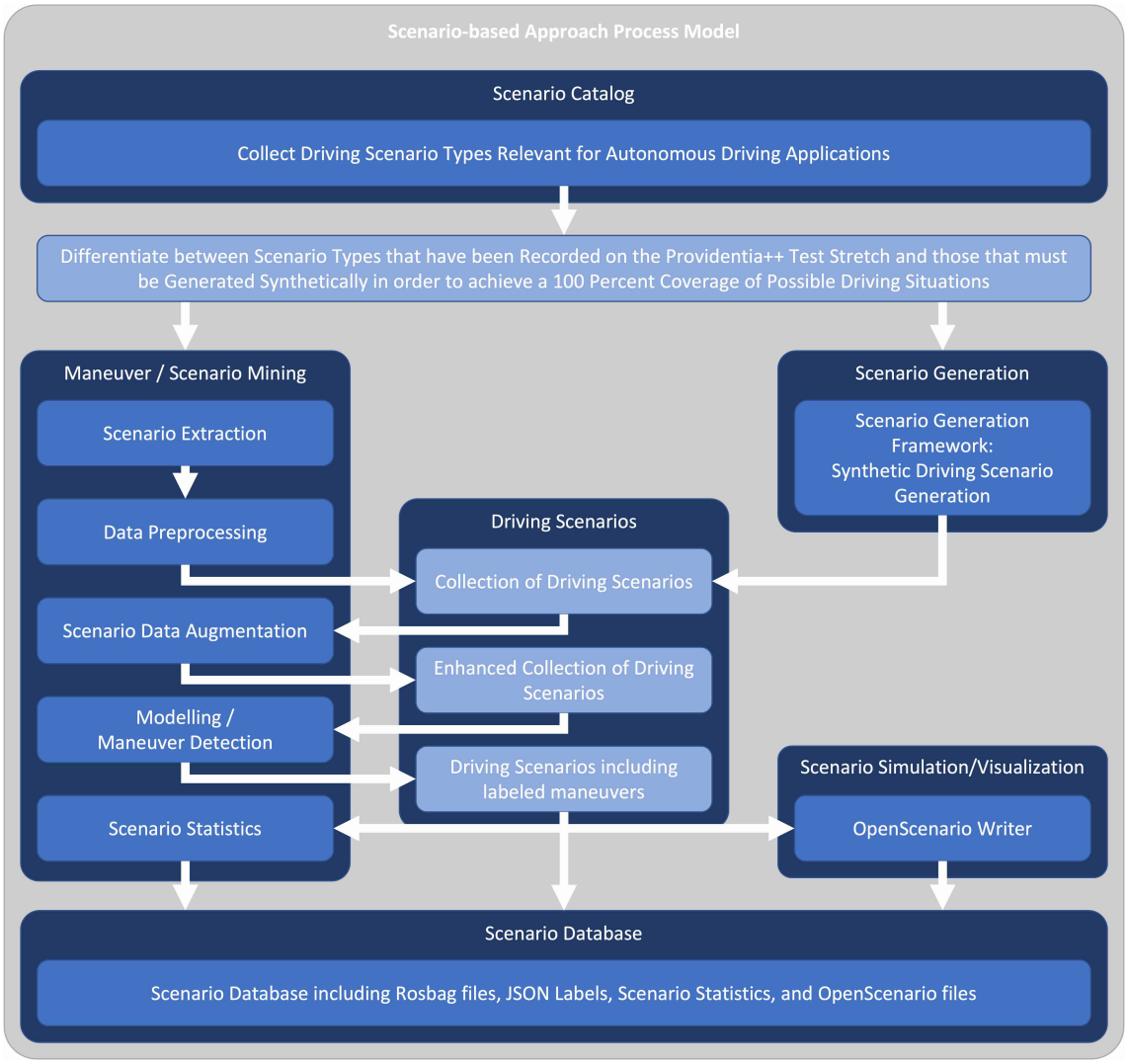


Figure 3.6: Stages of the scenario dataset generation [44]

The scenario dataset generation framework introduced by Kaefer [44] is compatible with pre-recording data; however, it is not possible to detect maneuvers that rely on past information in real-time since the real-time implementation only processes the current ROS *message*.

3.4.2 Maneuver Detection

The maneuvers of the vehicles are determined based on the trajectory information of the vehicle and preprocessed data. In the preprocessing step, several features are extracted from the trajectory information of each vehicle such as the vehicle's distance to the lane center,

the ID of the lane vehicle is on, distance to the following and leading vehicles, Time-To-Collision (TTC) information for the following and leading vehicles.

3.4.2.1 Maneuver Types

The maneuvers that could be detected by the *Maneuver Detection* component of the scenario dataset generation framework of the Providentia++ project are shown in Table 3.1. The algorithm behind the detection of selection of maneuvers is summarized in Sections from 3.4.2.2 to 3.4.2.5.

Road Type	Detected Driving Maneuvers
Highway	Enter / Exit Highway, Lane Change Left / Right, Cut-In Left / Right, Cut-Out Left / Right Speeding / Standing Vehicle, Tailgate
Urban / Rural	Turn Left / Right at Crossing, Straight / U-Turn at Crossing, Lane Change Left / Right, Cut-In Left / Right, Cut-Out Left / Right, Speeding / Standing Vehicle, Tailgate

Table 3.1: The overview of the detected maneuvers [44]

3.4.2.2 Maneuver: Lane Change Detection

According to [3] and [40], unsafe lane changes are responsible for a considerable amount of accidents that happen in the United States where the ratio of these types of accidents is %5 of all accident events. Therefore, it is important to detect lane change maneuver of the vehicles to analyze the accident situations or traffic congestion [53].

The maneuver detection tool in the Providentia++ project detects lane changes based on the ID of the lane vehicle is on and the distance to the lane center information [44]. Firstly, the time points when a vehicle crosses a lane marking are determined based on the lane ID information by finding the time points where the lane ID information of the vehicle changes. In order to determine the start and end of the lane change behavior, the distance to the lane center metric is used. When a driver decides to change the lane he/she is driving on, the vehicle starts to move away from the center of the lane; therefore, the distance to the center of the lane increases. The start of the lane change behavior is the time when increase in the distance to the lane center occurs. Similarly, the distance to the lane center decreases when the vehicle closes to the end of the lane change behavior. This marks the end time point of the lane change maneuver.

3.4.2.3 Maneuver: Cut-in and Cut-out Detection

The cut-in and cut-out maneuvers are among the most challenging situations in traffic and the vehicles with ADAS are tested against the scenarios including such maneuvers [61]. Cut-in and cut-out maneuvers are types of lane change maneuvers but it also includes another vehicle into the consideration. Cut-in is a maneuver type where a vehicle changes lane to an adjacent lane just in front of another vehicle. On the other hand, cut-out is a maneuver type where a vehicle changes its lane when it is very close to the car in front of it, in other words when it keeps a distance that is less than the required safe following distance.

Detecting cut-in and cut-out maneuvers uses the output of lane change detection and TTC metric. In Equation 3.1 [74], TTC is defined where Δv is the velocity difference between the leading and following vehicles and Δd is the absolute distance between the vehicles.

$$TTC = \Delta v / \Delta d \tag{3.1}$$

In the cut-in detection algorithm, cut-ins are extracting the checking the lane changes maneuver points and checking if the TTC value after the lane change is less than 2 seconds [44]. Similarly, cut-outs are detected when the TTC value before the lane change is less than 2 seconds [44].

3.4.2.4 Maneuver: Speeding and Standing Vehicle Detection

Speeding and standing vehicles are detected solely based on the velocity information of each detected vehicle on the road [44]. Speeding vehicles are detected when the velocity of the vehicle exceeds the speed limit on the road which is 130 kilometers per hour (kph) on the A9 highway. Similarly, standing vehicles are detected when the velocity of the vehicle is very close to or equal to 0 kph [44].

3.4.2.5 Maneuver: Tailgate

Tailgate behavior can be described as driving a vehicle closer than the required safe distance and not increasing the distance to the required safe distance in a required timely manner. The required safe distance and time to correct distance vary based on the road type and the country-specific regulations. In German traffic regulations [20], the safe driving distance for a vehicle is the distance travelled in a second and required time to keeping the distance is 3 seconds in urban areas, and safe driving distance is vehicle's velocity divided by 2 in meters and required time to keeping the distance is 3 seconds if the speed is less than 160 kph, 1 seconds if it is more than that in areas other than urban road.

The detection of tailgates is performed using the distance to the leading vehicle and the velocity of the vehicle [44]. The algorithm performs the rules described in the German regulations described above. The tailgate events are divided into three categories based on the risk of the behavior causing an accident: minor, moderate and severe tailgate [44]. A tailgate event is considered as severe if the vehicle's velocity is over 100 kph and it has the distance to the leading car less than the 30 percent of the required minimum distance [44]. A moderate tailgate event is detected if a vehicle's velocity is over 100 kph and it is holding a distance to the leading car less than the 50 percent of the required minimum distance or a vehicle's velocity is between 80 and 100 kph and distance to the leading vehicle is 50 percent of the required safe distance [44]. Finally, any detected tailgate event that does not fall in moderate or severe tailgate behavior is labeled as minor tailgate [44].

Chapter 4

Solution Approach

The contributions of this thesis work are categorized into two main areas as discussed in Section 1.4. The first main contribution is the implementation of a neural network training pipeline for the detection of lane split situations. The solution approach for the pipeline is described in Section 4.1. The second main contribution is to the Providentia++ project. The solution approach for the detection of an accident detection and support for the live system maneuver detection is described in 4.2.

4.1 Neural Network Training Pipeline for Lane Split Detection

In addition to the neural network-based approach for the detection of lane split scenarios in the context of road perception, a training pipeline for the introduced neural network model based on Kubeflow Pipelines is implemented in this work.

Kubeflow is selected as the platform for the development of a training pipeline since it offers ease of use with a web interface to run a training job, flexibility in the development of tasks in the machine learning workflow, ability to run training tasks in a distributed system and an interface for automated hyper-parameter optimization. Moreover, the components developed for a pipeline can be re-used in another pipeline with a different purpose than the original one; thus, developing pipelines for Kubeflow reduces the development time and cost.

The pipeline is an end-to-end machine learning pipeline approach as it consists of components that define the entire model training workflow from data preparation to saving trained models and their parameters. The details of the pipeline and its components are discussed in Section 4.1.1 and the following sections.

4.1.1 Overview of the Pipeline

The training pipeline has 5 components in total; *data selection* component discussed in Section 4.1.2, *for loop* component that discussed in 4.1.3, *auto-labeling* component discussed in 4.1.5, *dataset preparation* component discussed in 4.1.6 and *training* component discussed in 4.1.7, respectively.

The overview of the pipeline is shown in Figure 4.1 as a DAG. The reason that *for loop* is represented as a component is the pipelines are DAG; thus, there is not a closed loop in the graph where a component is its own descendent. Therefore, the component that runs in the loop is shown on the left side of the for loop, and the components that are run after the for loop ends are shown on the right side of the for loop. The descriptions of the components are summarized below.

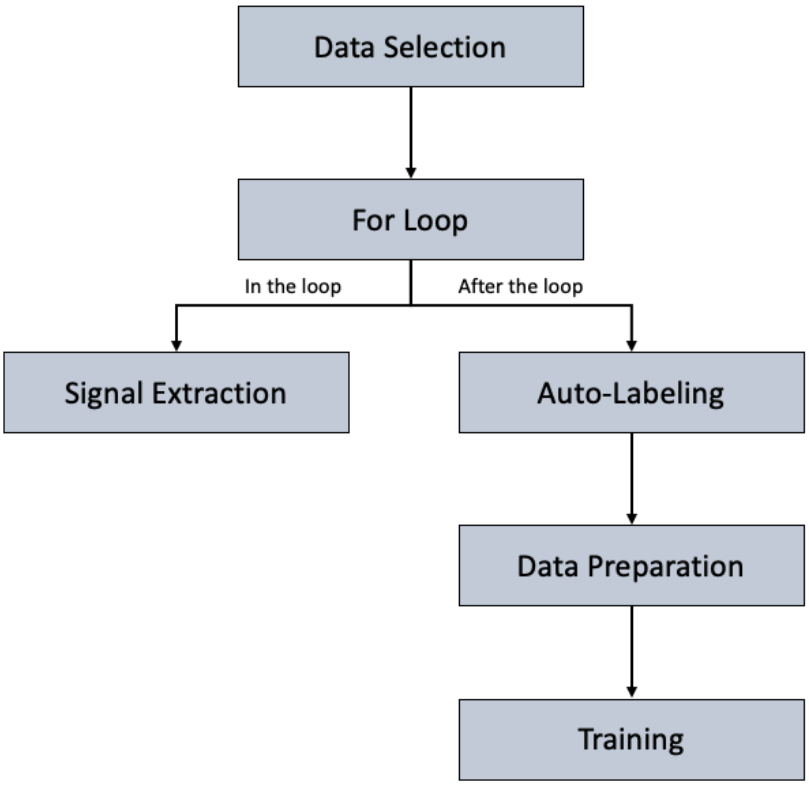


Figure 4.1: Overview of the Kubeflow pipeline as Directed Acyclic Graph (DAG)

1. Data Selection

   The selected data traces that will be used for training are filtered out based on the signals already extracted in the previous pipeline runs. The output of this component is the list of data traces that needs signal extraction. The for loop iterates over this output.

2. For Loop

   The sensor signals semantic road points and lane boundary polynomials that will be used in both training and auto-labeling are needed to be extracted. The signal extraction component is needed to be run for each data trace in the output of the previous component.

3. Auto Labeling

   After the loop finishes, the auto-labeling component starts labeling the data traces based on the extracted signals.

4. Dataset Preparation

   The signals are extracted and labels are generated for all the data traces in the previous components. In this component, the dataset is generated by combining the necessary signal data and the generated labels.

5. Training

The generated dataset is split into three subset datasets *training*, *validation* and *test* datasets. A selected neural network model which can be MLP, CNN or self-attention based is trained and evaluated on the generated datasets.

4.1.1.1 Pipeline File Definition

Kubeflow Pipelines are defined as Yet Another Markup Language (YAML) files and the pipeline definition files can be generated with a Python script in an automated fashion. Kubeflow provides a Python SDK [9] with a set of tools to define components with its name, input arguments, and resources that it might use such as Persistent Volume Claim (PVC) or a *Secret*. An example pipeline YAML file definition can be seen in Figure 4.2 with a component named *example_component* which is defined as a container image pointing to a Docker Uniform Resource Identifier (URI). The pipeline has a single parameter named as *example_param*.

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
name: Example Pipeline
spec:
  entrypoint: example_component
  templates:
  - name: example_component
    container:
      image: example/docker/image:latest
      args: [
        python3,
        program.py,
        --input,
        {{inputs.parameters.example_param}},
      ]
    inputs:
      parameters:
      - name: example_param
  arguments:
    parameters:
      - name: example_param
        value: 'default_str
```

Figure 4.2: An example Kubeflow pipeline definition

The pipeline YAML file can be uploaded to the Kubeflow platform via the web interface. The uploaded pipelines are versioned; therefore, it is possible to use another version of the same pipeline for testing purposes.

4.1.1.2 Pipeline Parameters

As discussed in Section 4.1.1.1, pipeline runs can be configured with parameters. The pipeline takes all the arguments defined in the file definition and each defined parameter can be used as arguments of components. The parameters of the pipeline implemented in this work and the components that use them are listed in Table 4.1. Each parameter is described in the sections that detail the components that use them.

Parameter	Used in Component(s)
selected_data_traces	Data Selection Dataset Preparation Auto-Labeling
run_tag	Dataset Preparation Training
time_offset_ns	Dataset Preparation
road_point_relevance_min_distance *road_point_relevance_max_distance*	Auto-Labeling
road_point_existence_probability_threshold	Auto-Labeling
validation_split *test_split*	Training
num_epochs	Training
batch_size	Training
kernel_regularizer	Training
bias_regularizer	Training
learning_rate	Training
label_names	Training
balance_event_files	Training
use_class_weight	Training
apply_normalization	Training
fill_nan_values	Training
time_window_size	Training
model_type	Training
use_batch_norm_1 *use_batch_norm_2*	Training
use_dropout_1 *use_dropout_2*	Training
use_pooling	Training
dense_layer_1_size *dense_layer_2_size*	Training
dense_layer_1_size *dense_layer_2_size*	Training
dropout_p_1 *dropout_p_2*	Training
convolution_kernel_size *convolution_filters*	Training
use_mlflow_logging *mlflow_tracking_uri* *mlflow_experiment_id*	Training

Table 4.1: Pipeline parameters and their descriptions

4.1.1.3 Communication Between Components

Kubeflow components communicate with each other either using the files in the file system or container outputs. If a component uses a container output as one of its inputs, the return value of the container is used. If components will "communicate" or share results of the operations they are performing using the filesystem, containers must be using volumes which are files or directories that point to a file storage in a local or remote system.

4.1.1.4 Kubernetes Persistent Volumes and Persistent Volume Claims

It is discussed in Section 2.10 that Kubeflow components are run on top of a Kubernetes cluster. A Kubernetes cluster is a set of server machines called *Nodes* that run applications and each running application is encapsulated in a *Pod* that runs single or multiple containers [11]. Since containers are run with the isolation principle, file storage on the cluster can be accessed from the containers by "binding" or attaching a volume that points to the file or directory on the file storage [13]. Accessing the file storage from a Kubernetes *Pod* is performed by mounting a resource called *PVC* to the *Pod* which is bound to a *PV* that points to the location on the file storage. In order to run the pipeline, required Persistent Volume (PV)s and PVCs must be created and this operation is performed only once since they can be reused with each run of the pipeline.

PV - PVC name	Description
pv-scenario-detection-common pvc-scenario-detection-common	The common pipeline directory in the file system to be used to store all the file output generated by the pipeline components. This volume has read and write access to the storage.
pv-signal-data pvc-signal-data	The directory that stores raw signal data for the traces. This volume is read-only.

Table 4.2: Overview of the PVs and PVCs used in the pipeline

In this work, there are 2 PVs and PVCs required to run the pipeline. These resources are listed and their purposes are described in Table 4.2. The files written to or read from the file storage by the components will be discussed in the sections that describe the components.

4.1.1.5 Kubernetes Secret

The sensitive information such as credentials should be used safely when they are needed by a *Pod* in a Kubernetes *cluster*. In order to ensure the safety of such sensitive information, Kubernetes has a resource type named *Secret* that stores the credentials as a dictionary that maps a credential name to credential value [12]. This ensures that Kubernetes or Kubeflow code does not include any sensitive information.

Name	Key - Value pair	Description
data-service-access-token	*ACCESS_TOKEN*: access token	The token that is needed to authorize requests sent to the web service that serves the signal data paths.

Table 4.3: Kubernetes *Secret* used in the pipeline

There is only one *Secret* used in this work and the information it stores is described in Table 4.3. The usage of this *Secret* is described in Section 4.1.4.

4.1.1.6 Definition of Components

Each component in a Kubeflow pipeline must be containerized. Hence all the required packages are included in the container and inputs and outputs are well-defined, a containerized application should perform the job it is designed for when its inputs are given correctly no matter the environment. However, some components in Kubeflow might be a simple Python

script that does not rely on any additional packages. In that case, the component can simply be defined as a Python script and Kubeflow automatically uses base Python container to run the script in the component.

In this work, all the components except the data selection component described in Section 4.1.2 are containerized applications.

4.1.2 Data Selection

The previous subsections in Section 4.1 lay down the foundations of how the implemented Kubeflow training pipeline works and the Kubernetes resources required to run the pipeline. The pipeline components are detailed in subsections starting from this subsection to Subsection 4.1.7.

Data selection component is the starting point of the pipeline as it can be seen in Figure 4.1 and it is defined as a simple Python script.

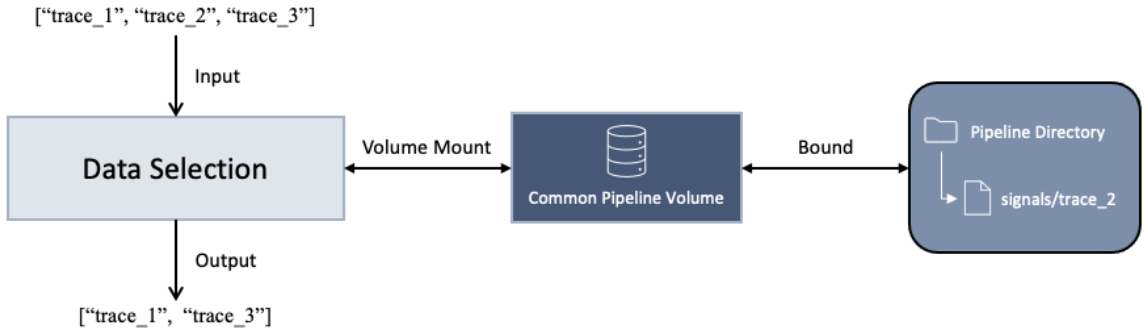


Figure 4.3: Illustration of the data selection component

The component takes a single input parameter which is a list of selected data trace ids to be used in training for the pipeline run. However, signal data required for training of some of the data traces might already been extracted in previous pipeline runs. Extracting the signal data for all of the selected data traces would make training runs slower and inefficient; therefore, data selection component checks the common pipeline storage to filter out the ids of the data traces with extracted signals. The component mounts the common pipeline volume described in Section 4.1.1.4 for the described operation. An example component run is illustrated in Figure 4.3 that shows the data trace id *trace_2* being filtered out since it exists in the directory that contains extracted signals.

4.1.3 For Loop

The for loop in the pipeline loops over the data traces whose signal data has not been extracted by the signal extraction component defined in Section 4.1.4 in the previous training runs. The component does not use any volumes and it only takes the list of data trace ids which is the output of the predecessor data selection component detailed in the previous section 4.1.2.

It is been discussed in the Section 4.1.1 describing the pipeline DAG shown in Figure 4.1 that the signal extraction component run inside the loop is shown as the successor of the for loop component while the components that are run when the for loop finishes succeeds the for loop component on the right side. This illustration is designed to conform how Kubeflow draws pipeline graphs with for loops.

The signal extraction component inside the for loop is invoked with a data trace id parameter in the input list of the for loop. The for loop component can run the signal extraction jobs in parallel for up to 10 jobs running concurrently.

4.1.4 Signal Extraction

The data traces are the real-world driven car data of the BMW Group AG prototype cars recorded to the filesystem. Moreover, paths to each trace can be fetched via a Hypertext Transfer Protocol (HTTP) request to an internal web service. The required signal data must be extracted from the raw trace recordings in the filesystem.

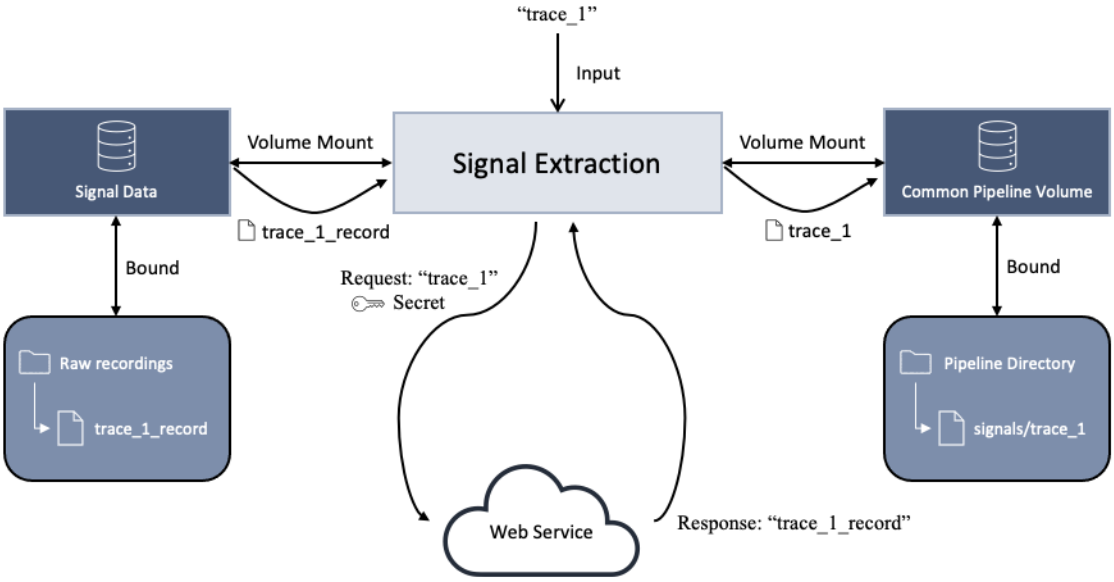


Figure 4.4: Illustration of the signal extraction component

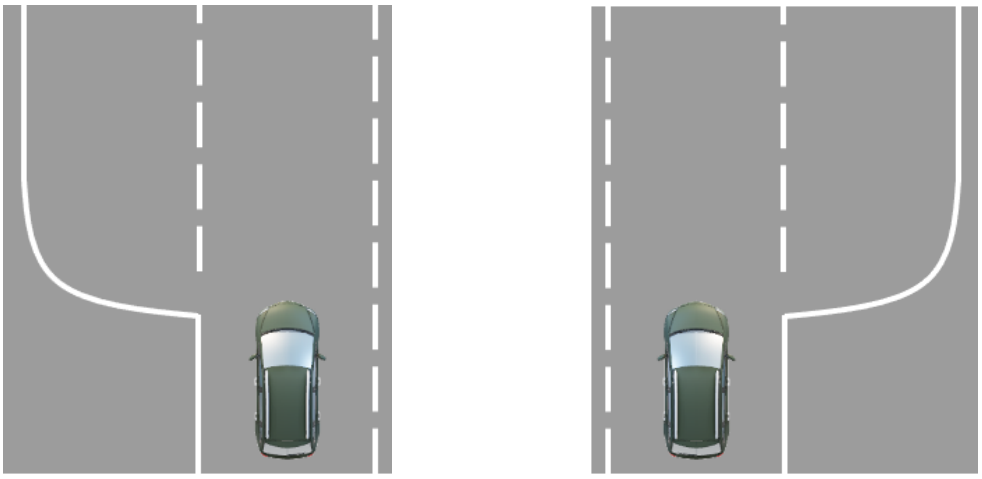
The signal extraction component extracts the desired signal data by fetching the path for the raw recordings via a HTTP request, and performing the extraction on the raw recordings. In order to fetch the file system paths from the web service, the HTTP requests must contain a token so that the requests are authenticated. Thus, the Kubernetes *Secret* described in the Section 4.1.1.5 is attached to the component container as environment variable. The environment variable is then included in the HTTP requests to get authenticated. The signal extraction component and its interactions with other resources are illustrated in Figure 4.4. In the illustration, the component extracts signal for the trace with id *trace_1*.

The signals are only required by the auto-labeling and dataset preparation components. The extracted signals are semantic road points signal which is detailed in Section 4.1.5.1 and lane boundary polynomials which is detailed in Section 4.1.6.

4.1.5 Auto-Labeling

Auto-labeling is an important tool in a neural network training workflow. There would be a large amount of data which is impractical to be manually labeled. Therefore, an automated labeling solution should be implemented. In this work, the aim is to detect outlier scenarios in the lane boundary polynomials when there is a lane split on the road. The scenarios of lane split to left and lane split to right are illustrated in Figures 4.5a and 4.5b, respectively.

The lane the vehicle is driving on splits to left or right, and a new lane on the left or right starts from the point split starts.



(a) Illustration of the lane split to left scenario **(b)** Illustration of the lane split to right scenario

Figure 4.5: Illustrations of lane split scenarios

In order to label such scenarios, a rule-based algorithm that uses the extracted signal which contains semantic road points is implemented. The details of the semantic road points signal are explained in Section 4.1.5.1. The idea behind the labeling algorithm is discussed in Section 4.1.5.2.

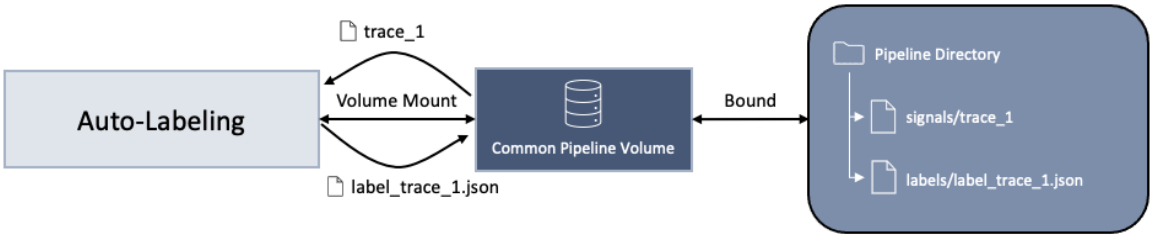


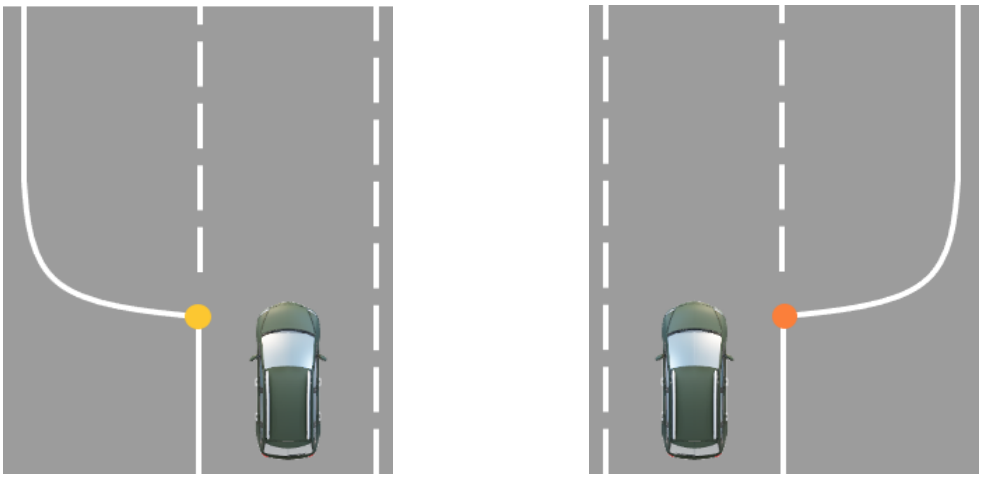
Figure 4.6: Illustration of the auto-labeling interactions with pipeline resources

The auto-labeling component is the first component that follows the for loop component as shown in Figure 4.1. Thus, the auto-labeling component and the components that follow it are run after the for loop finishes. The interactions of the auto-labeling component in the pipeline are shown in Figure 4.6. The component mounts the common pipeline volume to get the extracted signals from the volume and to write the generated labels to the volume.

4.1.5.1 Semantic Road Points

Semantic road points are the points that define the road events and they are detected by the BMW prototype camera mounted to the automated vehicle. A road event indicates the change on the detected road such as a new lane to the left or right, merging lanes from left or right and widening lanes. The detected points are positioned at the starting point of the road event.

The semantic road points that define the lane splitting to left and right road events are *NewLaneLeft* and *NewLaneRight*, respectively. The *NewLaneLeft* semantic road point is marked as a yellow point in the illustration of lane split to left scenario in Figure 4.7a. Similarly, *NewLaneRight* semantic road point is marked as orange point in the illustration of lane split



(a) Semantic road point for lane split to left scenario

(b) Semantic road point for lane split to right scenario

Figure 4.7: Illustrations of lane split scenarios with semantic road points

to right scenario in Figure 4.7b. These points denote the position of the starting point on the left or right lane where the splitting event starts.

The semantic road points also hold a useful information that indicates the distance to the lane that car is driving on in order to determine if it is a relevant point. The prototype camera might detect a semantic road point for a lane split scenario; however, this event might be occurring on a lane other than the lane the car is driving on. Therefore, the detected semantic road point would be irrelevant in detecting a lane split road event. The reason for this is the lane splits occurring on a lane that is not adjacent to the lane the car is driving on would not affect the detected lane boundaries that the car relies on for ADAS features.

In this work, the only semantic road point types used to detect lane split road events are the described *NewLaneLeft* and *NewLaneRight*.

4.1.5.2 Timestamp Labeling

The auto-labeling implemented in this work is based on time interval labeling. Since the data is time series and contains signal messages with respective timestamps, the message timestamps can be labeled with the desired label.

It has been described in Section 4.1.5.1 that the rule-based labeling is performed based on the existence of the relevant semantic road points, namely *NewLaneLeft* and *NewLaneRight*.

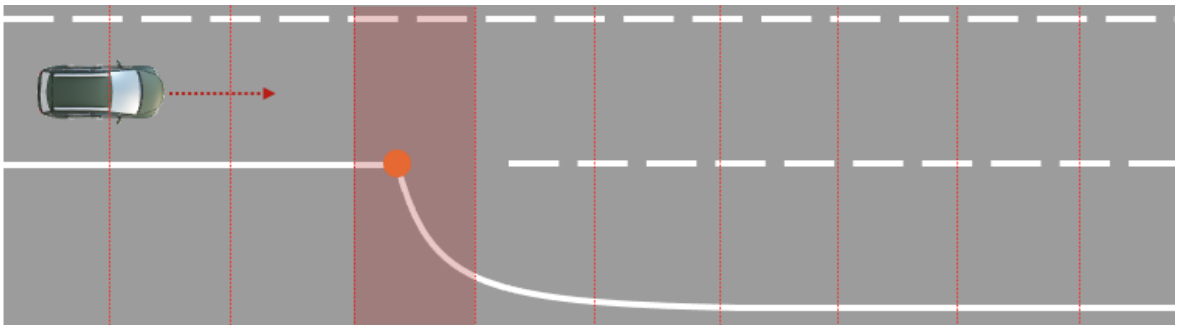


Figure 4.8: Illustration of the timestamp labeling for lane split to right scenario

A time interval is labeled as lane split to left or right if the signal messages in the interval contain *NewLaneLeft* or *NewLaneRight* and the road point in the messages is close to the

lane that the car is driving on. Hence, the semantic road points are first filtered based on the distance to the lane of the vehicle and existence probability. The semantic road point detection by the prototype camera might not be accurate; therefore, the existence probability attribute is used to filter uncertain detections. Then, the messages are labeled as lane split or not based on the existence of the semantic road points. In Figure 4.8, the semantic road point *NewLaneRight* is marked as a orange dot similar to the illustration in Figure 4.7b. The red-dotted vertical lines indicate the signal messages or the timestamps where a message arrives. The red rectangle containing the semantic road point mark is the interval labeled as lane split to right. The red-dotted vertical lines of the labeled interval are the start and end timestamps of the label.

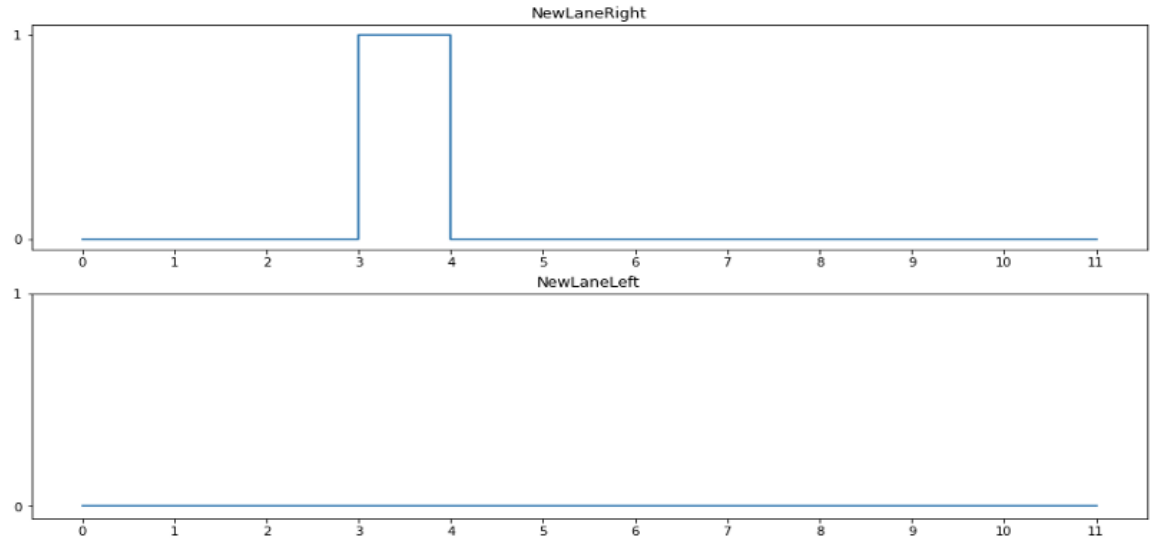


Figure 4.9: Plot of the semantic road point signals for scenario in Figure 4.8

The existence plot of the semantic road point signals *NewLaneLeft* or *NewLaneRight* for the example in Figure 4.8 is show in Figure 4.9. The plot at the top is the *NewLaneLeft* signal and the plot at the bottom is the *LaneSplitRight* signal where the x-axis is the timestamps. The y-axis value *1* means that the semantic road point exists and *0* means that the semantic road point does not exist at the timestamp t.

4.1.5.3 Output Label Description

As discussed in Section 4.1.5.2, the auto-labeling algorithm labels each signal message; therefore, each timestamp is labeled. Since the data is time series and the messages are ordered, storing the start and end timestamps of the label event in the output file is sufficient. Each data trace has its own label file that contains a list of dictionaries where each dictionary is a label. The format of the label files is JavaScript Object Notation (JSON) and an example label file is shown in Figure 4.10.

The lane boundary events, i.e labels, are enumerated integers. For example, the *0* value for the label means there is no lane split, the *2* value means lane split to right and the value *3* means lane split to left. The label names that map human-readable names to enumerated label values are given by the pipeline parameter *label_names*.

```
[
  {
    "startTimestamp": 120,
    "endTimestamp": 620,
    "laneBoundaryEvent": 2,
  },
  {
    "startTimestamp": 1240,
    "endTimestamp": 1990,
    "laneBoundaryEvent": 3,
  }
]
```

Figure 4.10: An example label file for a data trace generated by the auto-labeling component. *laneBoundaryEvent* denotes the label for the time interval.

4.1.6 Dataset Preparation

The dataset preparation component combines the labels generated by the auto-labeling component described in Section 4.1.5 and the lane boundary polynomial signal data that will be used as features of the neural network. The output files of this component are the input for the neural network training component where they are used as a dataset.

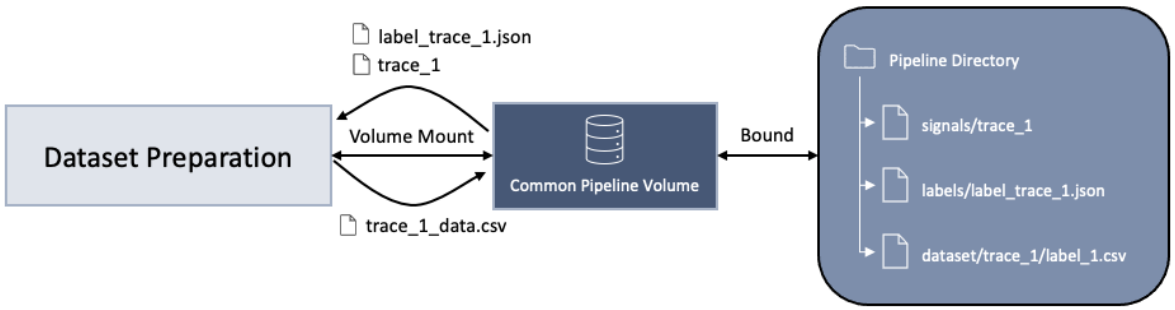


Figure 4.11: Illustration of the dataset preparation component interactions with pipeline resources

The interactions of the dataset preparation component are illustrated in Figure 4.11. The component uses the common pipeline volume to store the generated dataset files and read the extracted signals.

Each label generated by the auto-labeling component is written as a separate data file in Comma-Separated Values (CSV) file format. The reason for using CSV format over the JSON file is that the CSV file format takes less size on the disk than JSON since JSON requires syntax related characters for formatting whereas CSV only uses comma characters for formatting.

In order to generate labeled data files to be used in the training component that will be discussed in 4.1.7, the JSON label files for each selected data trace and the lane boundary signal data for the corresponding data traces are read and combined. As it has been discussed in Section 4.1.5.2 and shown in Figure 4.8, the labels contain start, end timestamps and the lane boundary event which is the enumerated label. However, the label files generated by the auto-labeling content only contain labels for the classes *LaneSplitLeft* and *LaneSplitRight*; thus, the default label *NoLaneSplit* event is missing from the label files. Moreover, in order to capture the change in the signal messages when there is a lane split scenario, it is important

to feed the neural network also the messages that come before and after the lane split event. This will help the model to capture what makes *LaneSplitLeft* and *LaneSplitRight* scenarios different than the *NoLaneSplit* label. Therefore, the data files that will be used as dataset in the training component should be generated with a larger time interval instead of the actual label.

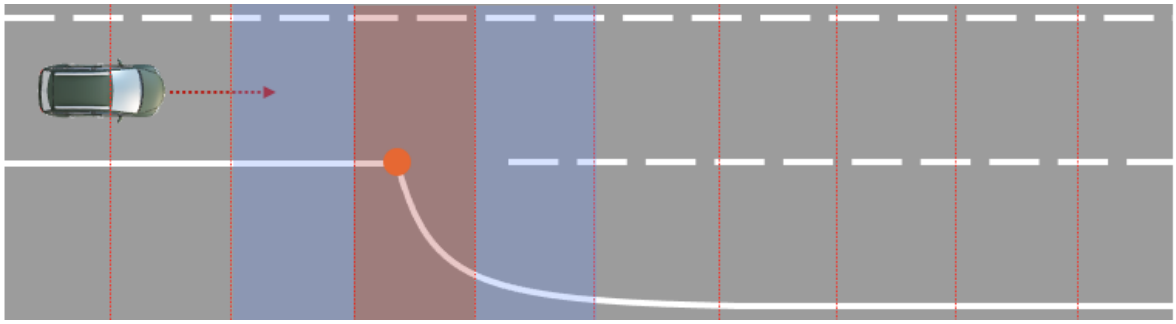


Figure 4.12: Illustration of the training data generation for lane split to right scenario

The dataset files are generated by using an offset value. The offset value is used to expand the label time interval from $(start\ timestamp, end\ timestamp)$ to $(start\ timestamp - offset, end\ timestamp + offset)$. The timestamps that are outside of the original label time interval; therefore, the timestamps in the range of $(start\ timestamp - offset, start\ timestamp)$ and $(end\ timestamp, end\ timestamp + offset)$, are labeled as *NoLaneSplit*.

The possible labels and corresponding enumerated labels for a timestamp in the data are shown in Table 4.4.

Label	Enumerated Label
NoLaneSplit	0
LaneSplitRight	2
LaneSplitLeft	3

Table 4.4: The label names and the corresponding enumerated labels

The process of labeling the extended timestamp is illustrated in Figure 4.12 which shows the data generation from the label created by the auto-labeling component. The original label time interval shown as the red rectangle that contains the orange semantic road point similar to the labeled interval in Figure 4.8 is extended with the offset shown as blue rectangles that come before and after the labeled interval rectangle.

After the extended time interval is determined, the labels are combined together with the matching lane boundary polynomial messages. Each timestamp contains three lane boundary polynomial information; left, center and right lanes. The information for the lanes is in the extracted signal information. The data preparation component matches the label timestamps that correspond to the semantic road point message timestamps with the lane boundary polynomial messages. The lane boundary polynomials in this work are three-order polynomials shown in the Equation 4.1 where $p(y)$ is the polynomial value; therefore, the $(y, p(y)$ point on the lane. The a_0, a_1, a_2 and a_3 are the polynomial coefficient values.

$$p(y) = a_3 y^3 + a_2 y^2 + a_1 y + a_0 \tag{4.1}$$

Alongside the polynomial coefficients, the view range start and end values for the lane boundary are also included in the data as features. The view range start and end values define the value interval for the y value in the polynomial equation shown in Equation 4.1. This is a useful information since the end of the view range might change when there is a

vehicle or an object in front of the car that occludes the lanes on the road or shortened lane boundary detection because of a lane split event.

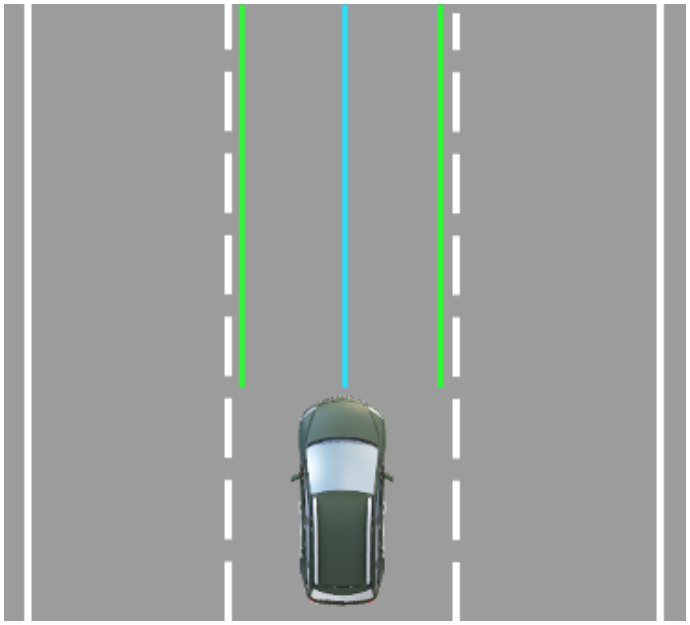


Figure 4.13: Illustration of the lane boundaries detected by the car camera

In Figure 4.13, the three lane boundaries included in the training data are illustrated; green lines as the left and right lane boundaries and turquoise line as the center lane boundary.

To summarize, each data file is a CSV file containing a timestamp, left, right and center lane polynomial coefficients and view range end-start values, speed of the vehicle and the enumerated label in every row.

4.1.7 Training

The training component performs the neural network model training using the data generated by the dataset preparation component. There are 2 different approaches for the problem; training and making predictions based on a single message which is discussed in Section 4.1.7.1 and based on time windows which is discussed in Section 4.1.7.3. There are also 3 different model types, namely MLP, CNN and self-attention based neural network. The details of each model and how they correspond to different training and prediction approaches are discussed in the respective sections; 4.1.7.2 for MLP model, 4.1.7.5 for CNN model and 4.1.7.6 for self-attention based model.

The interactions of the training component are illustrated in Figure 4.14. The component uses the common pipeline volume to read the generated dataset files and write the trained model and the confusion matrix as an image. Moreover, the component saves the trained model and the confusion matrix image alongside with the training parameters and metrics to a machine learning tracking service called *MLflow*. The logged information and *MLflow* are detailed in the dedicated Section 4.1.8.

As shown in Figure 4.15, the workflow of the training component consists of balancing the dataset, reading and standardizing the data, training and evaluation steps. The file-based balancing described in Section 4.1.7.9 is optional and can be replaced by the usage of class weights for data balancing which is described in Section 4.1.7.8. The data is read and split

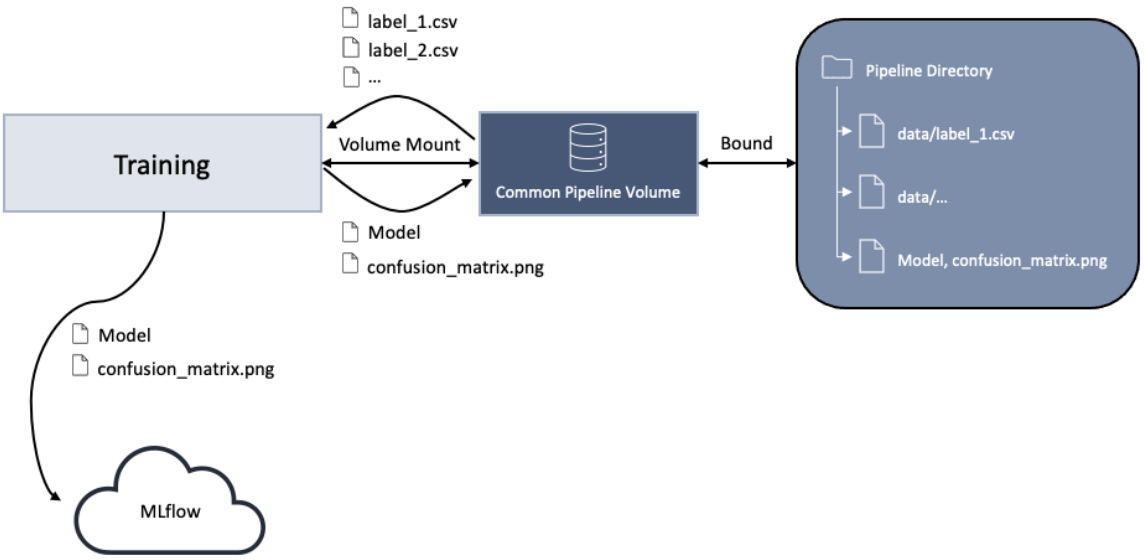


Figure 4.14: Illustration of the training component's interactions

into training, validation and test datasets based on the pipeline parameters that determine the ratio of each dataset. Then, each dataset is standardized as described in Section 4.1.7.7. The training and validation datasets are used during training and the data balancing is achieved through class weights if the option is enabled with the respective parameter *use_class_weight*. The trained model is also evaluated and the confusion matrix is generated using test dataset.

Before going into the details of the training approaches, it is worth noting that all 3 models are simple models that include number of layers not more than 3. The reason behind this is that the trained model can be used on car hardware to be tested for a real-world use-case and the hardware on the vehicles that would do the computation for the real-time inference is limited in terms of computing resources for most cars on the market. Therefore, the models are designed to be lightweight and simple.

4.1.7.1 Training on Single Sample

The simple baseline approach to the lane split detection problem is to train a neural network model that takes a single data point during training, and makes a prediction on a single data point during inference. Thus, the position of a data point in the training data is not an information interpreted by the neural network. The advantages or the disadvantages of the approach are discussed in the 5.

The MLP model implemented in this work is an example to the single sample training approach.

4.1.7.2 Multi-Layer Perceptron

The MLP model has 2 dense layers, i.e. fully-connected layers, optional batch normalization and dropout layers. The model can be configured to have batch normalization or dropout layer after each dense layer. Furthermore, the sizes of the dense layers, i.e number of neurons in the layer, can also be configured. The output layer has 3 neurons that output softmax values for each class shown in the Table 4.4.

The model is illustrated in Figure 4.16. The activation function for each dense layer is the ReLU function. The inputs are the data features for the single data point in the time series and they are the lane polynomial coefficients and view range start-end values for the left,

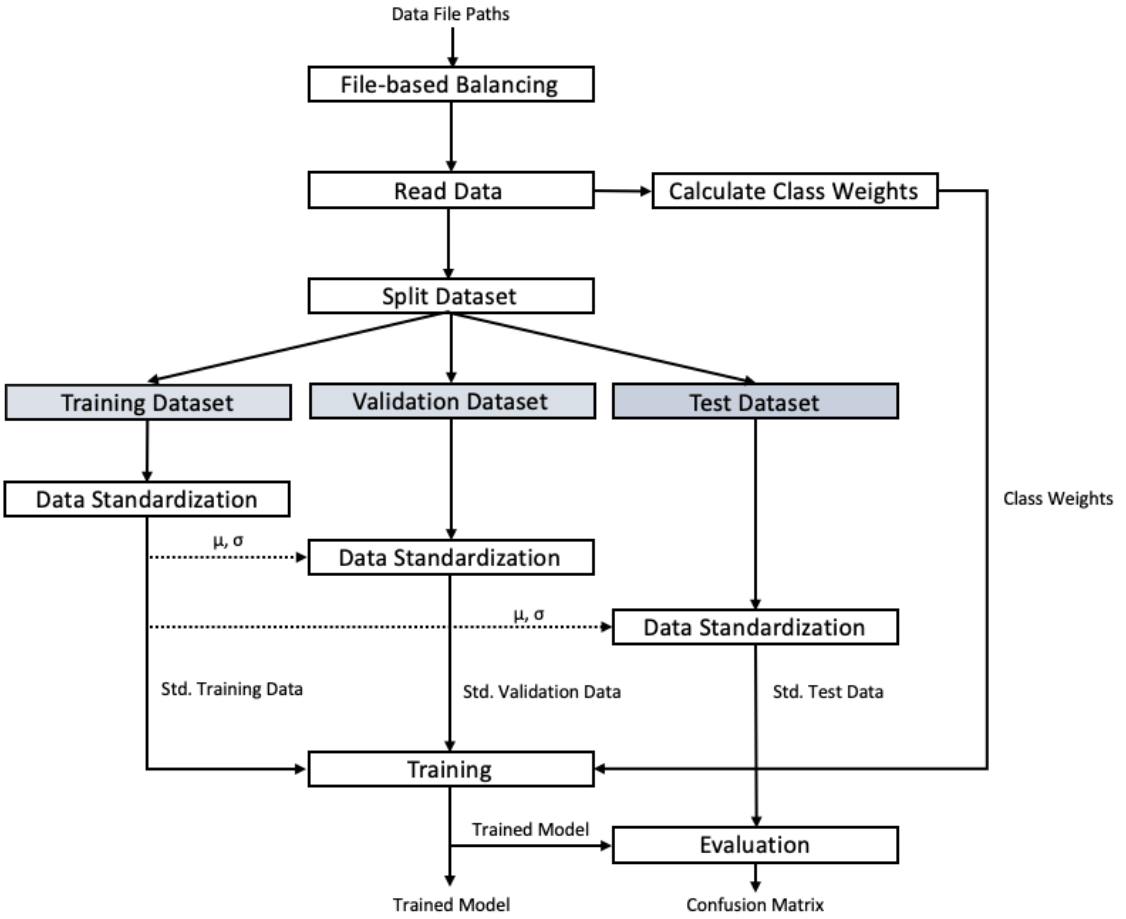


Figure 4.15: Illustration of the training component's workflow

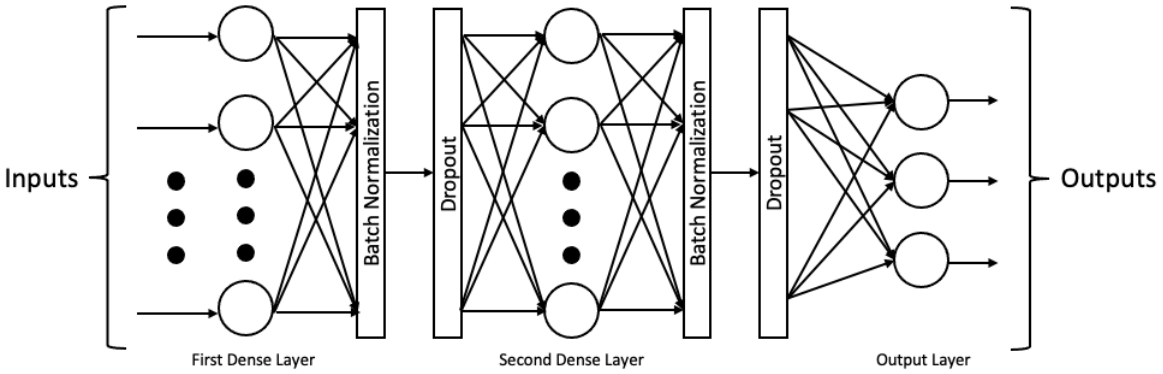


Figure 4.16: Illustration of the MLP model with all optional layers enabled

right and center lanes and the speed information. The size of the input layer is equal to the number of features and the size of the output layer is equal to the number of classes which is 3 in this work.

The batch normalization layers can be added to the model in order to have a faster training since they allow using higher learning rates that reduces the number of steps required to converge by normalizing the input to the layer comes after [31] batch normalization. Furthermore, dropout layers can also be added to the model with the aim to reduce the risk of overfitting on the training data by changing the network architecture by randomly blocking

some of the neurons based on a probability threshold so that learned weights would not be highly customized to the training data [31].

4.1.7.3 Training on Time Windows

The second approach to the lane split detection problem is to train a neural network model that takes a time window consisting of a fixed number of data points instead of a single data point. The advantage of using a time window is the ability of the model to learn sequence patterns in the data since data points in a time series are generally not independent from each other. This technique is called *sliding windows*.

The CNN and self-attention-based neural network models implemented in this work use sliding windows as their inputs.

4.1.7.4 Generation of Time Windows

The time windows are generated for each data point in the input by combining $w-1$ number of samples up to the data point the time window is generated for and the data point itself where w is the window size.

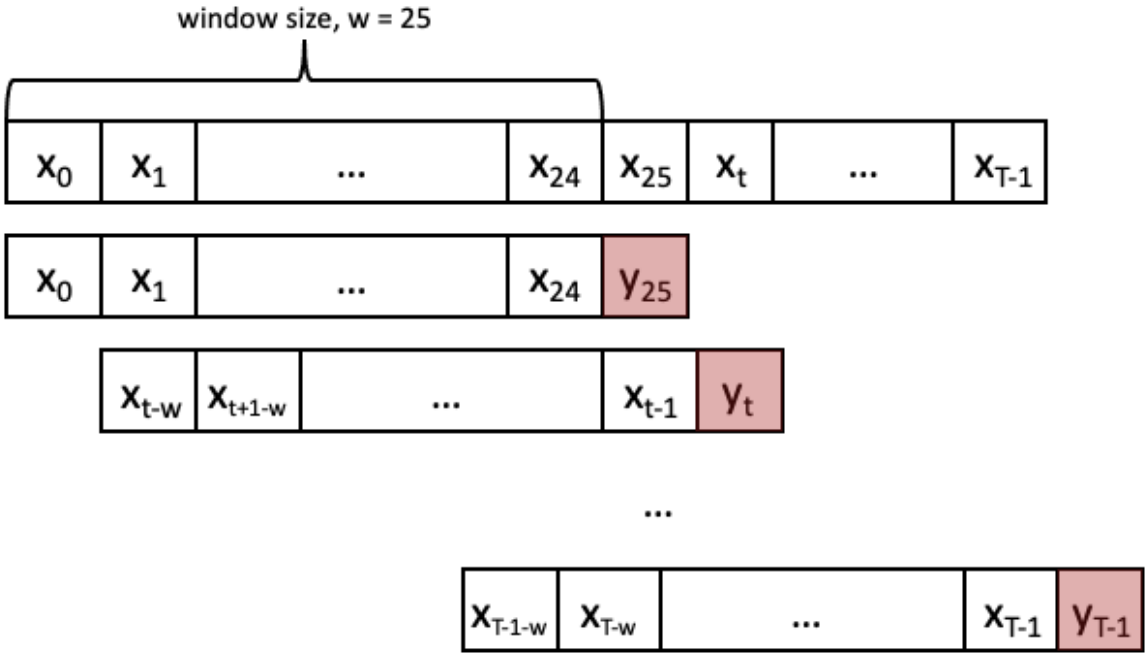


Figure 4.17: Illustration of the input generation with *sliding window*, own illustration (source: [67])

The generation of the time windows for each data point is illustrated in Figure 4.17 where the window size is 25 and the number of data points in the input is $T-1$. The x_k denotes a data sample in the input and y_k is the label of the kth data point. The first $w-1=24$ data points are not used since it is not possible to generate a time window for those data points. The reason for that is that there are not $w-1=24$ data points preceding them. The solution for this problem might be to fill in the missing data points with simply 0 value or the mean of the whole input.

4.1.7.5 Convolutional Neural Network

The CNN model has a single 1D convolutional layer followed by optional batch normalization, average pooling and dropout layers. Then, the output is flattened to have a 1D input to the

dense layer that outputs softmax for each class used in this work.

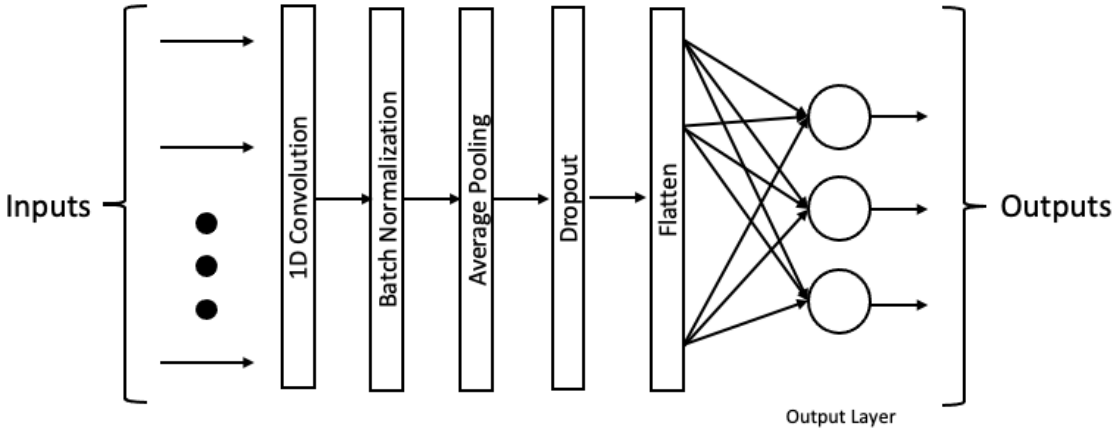


Figure 4.18: Illustration of the CNN model with all optional layers enabled

In Figure 4.18, the CNN model is illustrated. The time window input is fed to the 1D convolutional layer that creates a feature map of the input. For an input with a window size of $w = 25$ and a number of convolution filters that are used to extract features is f the output size of the first layer is $(23, f)$ for the convolution stride value of 1, kernel size 3 and no padding. The average pooling layer between batch normalization and dropout is used to reduce the number of parameters in the network and the output of the layer is of size $(11, f)$ with pool size p and stride s since the output size is calculated as $(input - p + 1)/s$. The flatten layer concatenates the input to have 1D output; therefore, the output size is $11xf$. The dense layer produces the softmax output from the 1D input.

4.1.7.6 Self-Attention based Neural Network

CNNs are able to capture local dependencies in the data; however, they cannot capture the global meaning of the data which might be a useful information to detect a long lane split event. CNN models need a large receptive field; therefore, large kernels or deeper model architecture to detect long-range dependencies. This makes CNN models inefficient when capturing such dependencies in the data; hence, it is not suitable to deploy a CNN model to the hardware with limited computing power. On the other hand, self-attention can overcome the shortcomings of CNN since it "sees" the input data as whole instead of a receptive field approach as in CNN.

In Figure 4.18, the self-attention based neural network model is illustrated. The time window input is fed to two separate 1D convolutional layers to create feature maps of the input called *query* and *value*. Both of the feature maps are fed into an attention layer that creates the *query-value encoding*. One of the initial feature maps created by one of the 1D convolutional layer called *query* is also fed into an average pooling layer whereas the *query-value encoding* is also fed into a separate average pooling layer. The outputs of the pooling layers are concatenated and fed into the dense layer that outputs softmax values for the 3 classes defined in 4.4.

The network is named as self-attention-based neural network since the attention layer gets its *query* and *value* encoding inputs from the same original neural network input which is a time window generated as described in Section 4.1.7.4.

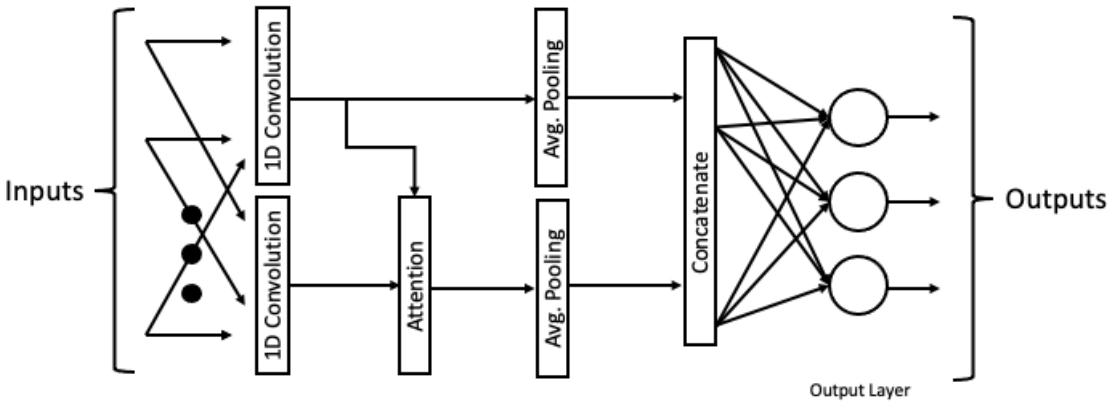


Figure 4.19: Illustration of the self-attention based model with all optional layers enabled

4.1.7.7 Data Standardization

Data standardization is applied to each feature in the input data so that each feature is centered around 0. This avoids features with a large scale to dominate the weights during training; therefore, it provides more stable and faster training. The equation for the operation is shown in Equation 4.2 where X is the input, μ is the mean of the feature and σ is the standard deviation.

$$X' = \frac{X - \mu}{\sigma} \tag{4.2}$$

The μ and σ values are calculated from the training dataset and the same parameters are used to standardize the validation and the test datasets. The model trains on the training dataset and learns the parameters based on the scales of the features in the training dataset. In order to have validation and test dataset features on the same scale that the model is trained on, these datasets are standardized using the same parameters.

4.1.7.8 Class Weights

An issue in applying a neural network approach to the lane split detection problem is the class imbalance in the input data. The neural network approach in this work handles the problem as a multi-class classification task where there are 3 classes, namely *NoLaneSplit*, *NewLaneLeft* and *NewLaneRight*. The problem is that a class usually dominates the generated dataset in size. Therefore, the model tends to put importance on the dominant class to improve accuracy. For example, if a binary classification problem with classes "go to gym" and "stay at home" has a dataset where "stay at home" labels are the %99 of the samples, the accuracy of the model would be %99 percent if it always predicts "stay at home" class.

$$w_c = \frac{N}{n_c * C} \tag{4.3}$$

In order to avoid the problem described above, classes are weighted based on their sizes and inversely proportional to the importance that the model should put on a class. The equation for the class weight of class c is shown in Equation 4.3 where w_c is the class weight, n_c is the number of samples belonging to class c, N is the total number of samples in the dataset and C is the number of classes.

4.1.7.9 Event Files Balancing

Another option to handle the class imbalance problem is to filter the data files of the dominant class to have a balanced dataset. The labeled data files can be filtered by selecting the number of class files as the size of the minority class.

However, filtering out some data files in the dataset also means that some information is completely lost and the model misses the opportunity to learn them from.

4.1.8 Training Run Tracking

In order to track each training run and easily compare the results, the training component logs the trained model file, the confusion matrix of the validation dataset and the training metrics that include the loss history, accuracy, precision and recall values for each training epoch to a web service platform named MLflow Tracking.

The MLflow Python Software Development Kit (SDK) automatically logs some information during training such as resulting metrics and loss history. However, the parameters used in the training run, trained model files and the confusion matrix image logging are implemented with the SDK methods.

The training run tracking is essential for hyperparameter tuning. Each training run has logged metrics and loss history so that MLflow can plot and even compare them with the web dashboard. The logged trained models can be downloaded to be used later for inference or resuming training from the last epoch.

4.2 Providentia++

The two main contributions to the Providentia++ project in this work are the implementation of rule-based accident scenario detection and scenario detection on the live system described in Sections 4.2.2 and 4.2.3, respectively.

The groundwork for the scenario detection framework is implemented by Kaefer in his work [44] that includes extraction of data used for maneuver detection such as distance between cars, TTC values, IDs of the lanes that the traffic actors are in, and maneuver detection based on the extracted features. This work extends the existing implementation with accident event detection, logging of metadata on the live system that uses ROS for communication, maneuver and event scenario detection on the live system.

Before going into the details of the approach to the contributions, the data format used in the project called *Scenario Description Format* to describe the scenarios in the data trace is described in Section 4.2.1.

4.2.1 Scenario Description Format

The *Scenario Description Format* is a data format used to fully represent the describe the traffic scenario of a given input data. The data format is illustrated in Figure 4.20.

The dictionary *Scenario* holds data nodes named *Meta information* and *Actors*. The *Meta information* data node is used to store the metadata information about the scenario such as the number of frames, frame rate or weather condition when the data is recorded. The *Actors* node holds the list of detected traffic participants or actors such as cars, trucks, buses or pedestrians. Each actor has basic information such as the tracking ID of the object, color, length, width and height in addition to object's driving behavior such as trajectory or path,

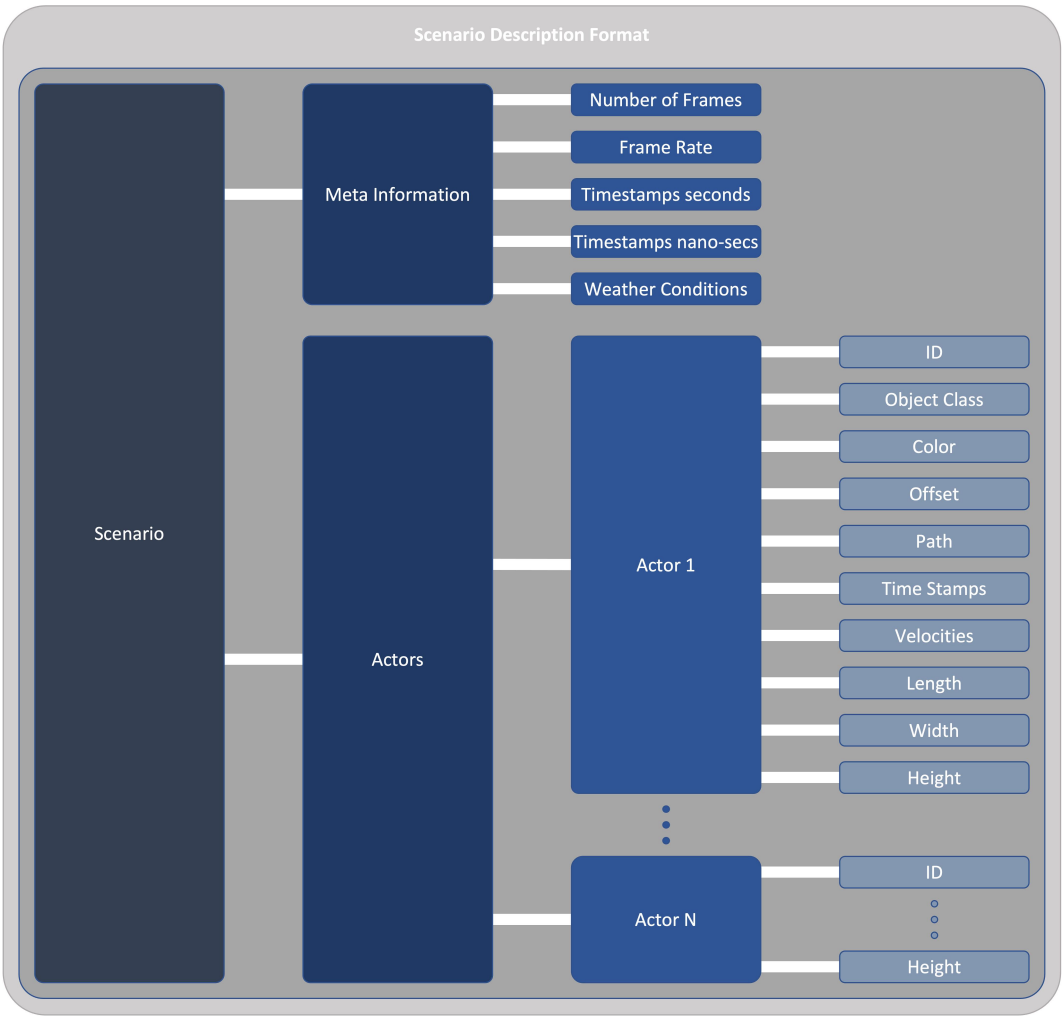


Figure 4.20: The illustration of the scenario description before this work [44]

lane ID, velocities, offset. Moreover, detected vehicle maneuvers are also recorded to actor dictionary such as speeding or standing, cut-in or cut-out and lane change behaviors. Therefore, the data format is the extended and used to describe the scene both offline and online, i.e. live system, scenario detection.

The scenario is first constructed from the ROS messages in *Scenario Description Format*. Then, the feature extractor extracts the features such as distance to the leading and following vehicle, TTC to the leading vehicle and the lane id for maneuver detection. These extracted features are also recorded to the actor itself. The maneuver detection is applied to the scenario and the extracted maneuvers are recorded to the actor in a similar fashion to what feature extractor does. This process is shown in Figure 4.21.

4.2.2 Accident Detection

Accidents are defined as the collision of a vehicle with an obstruction such as other traffic participants, vehicles, pedestrians or animals for instance, road barriers or trees [59].

Some of the possible accident scenarios on the German A9 highway are illustrated in Figure 4.22. The figure at the top shows, Figure 4.22a, the scenario where the vehicle runs into the road barriers. The center figure, Figure 4.22b shows the scenario where the following vehicle likely hit the leading vehicle since TTC is 0.5 seconds. The bottom figure, Figure

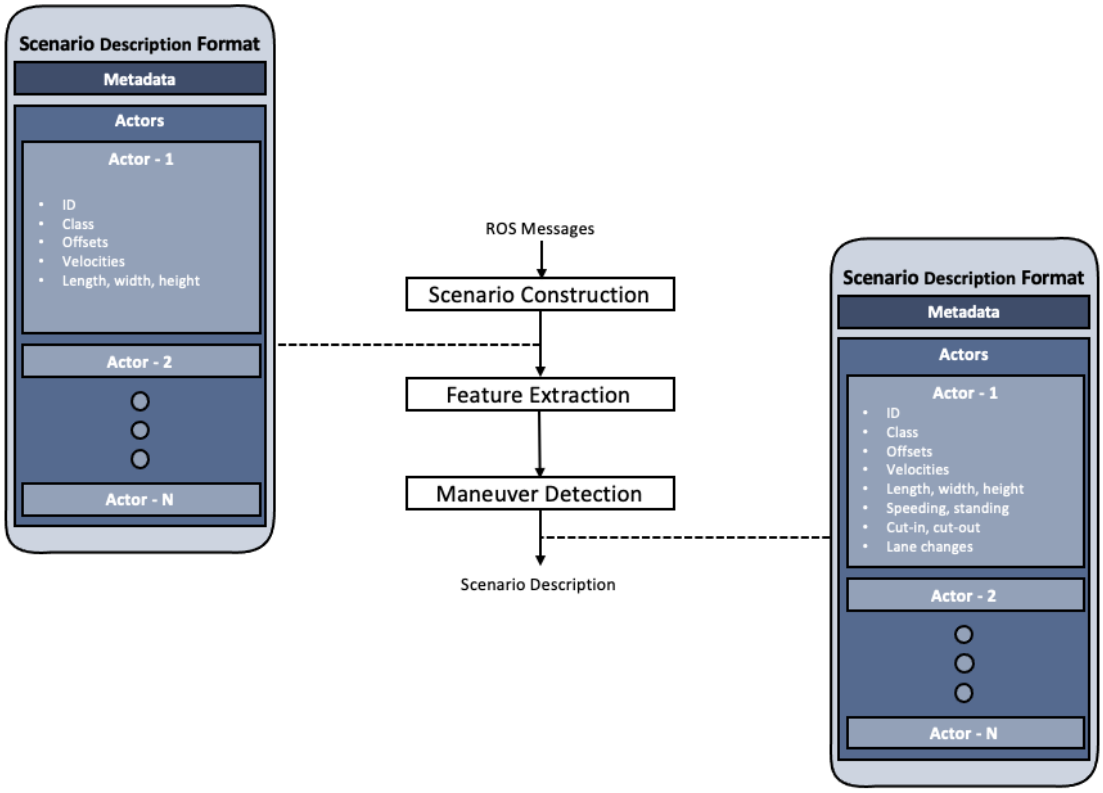


Figure 4.21: The illustration of maneuver detection and construction of the scenario

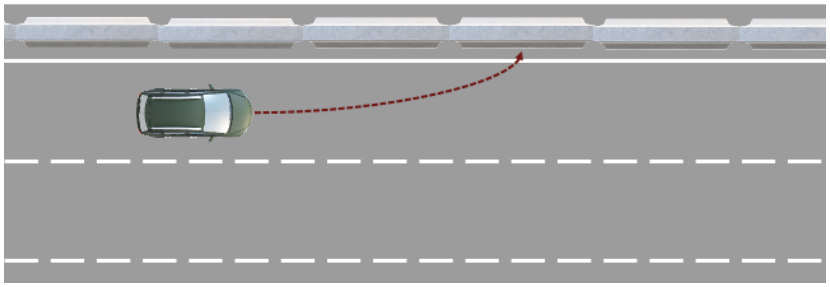
4.22c, shows the scenario where a vehicle changes lane where there is another vehicle on the trajectory. In this work, only accident scenario detected is shown in the center figure which is the head-to-tail collision.

In order to detect a head-to-tail collision, the accident detection algorithm checks for the TTC and the distance between the vehicles. If the TTC value is less than or equal to 1 second the actor is labeled as *collision likely* since 1 second is the limit for humans to react in a near collision situation and 1.1 seconds as TTC is the threshold for collision [80]. However, the actors are only marked as *in accident* if TTC is approaching to 0, distance to the leading car is less than 50 centimeters and the actor is already been marked as *collision likely* before. The accident labeling is shown in Equation 4.4 where $L_{accident}$ is the label of the accident scenario.

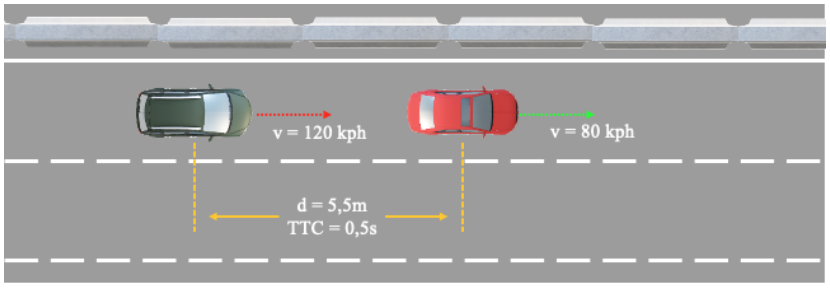
$$L_{accident} = \begin{cases} \text{collision likely}, & 0.5 < TTC < 1 \\ \text{in accident}, & TTC < 0.5 \text{ and } distance_{leading} < 50cm \\ \text{none}, & \text{otherwise} \end{cases} \tag{4.4}$$

If the following actor is marked as *in accident*, then the leading actor is marked with the same label since that actor will be hit by the following vehicle.

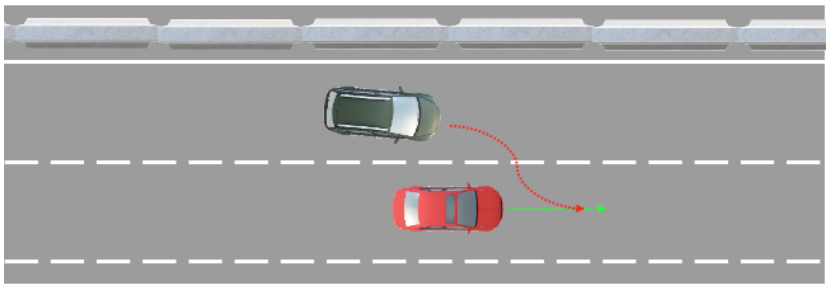
The accidents are also categorized based on the severity of the collision. If the speed difference between two vehicles is less than 50 kph, the accident is marked as *minor* accident. If the speed difference is in the range of 50 to 70 kph, the accident is marked as *moderate*. Similarly, if the speed difference is more than 70 kph, the accident is marked as *severe*.



(a) Green vehicle collides with the road barrier



(b) Green vehicle collides with the following vehicle due to speed difference



(c) Green vehicle will collide with another vehicle driving on the adjacent lane

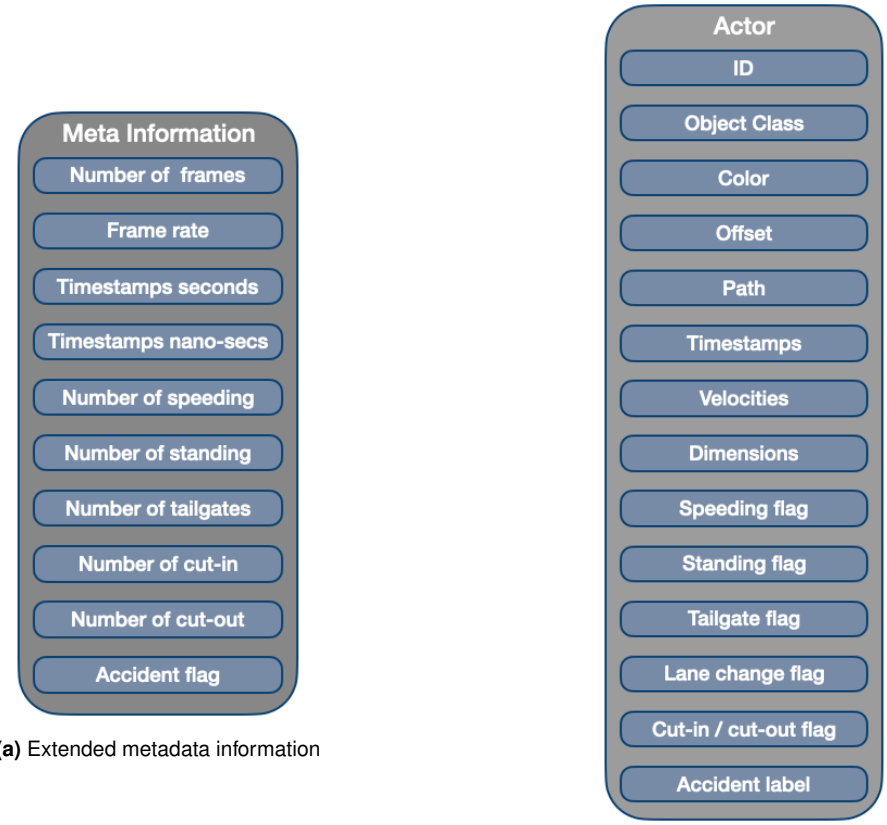
Figure 4.22: Illustration of some possible accident types on highway

4.2.3 Scenario Detection on the Live System

It has been discussed in Section 4.2.1 that the detected scenarios are stored in a data format named *Scenario Description Format*. However, the data format introduced in [44] can only be used to analyze single ROS message which causes accident detection and maneuver detection for lane change, cut-in, cut-out events to be left out on the live system. Therefore, the data format is extended in this work to keep history of the messages up to 200 hundred messages which is the equivalent of 8 seconds if the message frequency is 25Hz.

The extended data format illustrated in Figure 4.24 is similar to the original format except for the addition of *message_history* to the scenario dictionary and the extended metadata field that includes the number of lane changes, tailgate events, cut-in, cut-out and existence of an accident detected from the *message_history* field. The extended metadata field is shown in Figure 4.23a. Moreover, actor's path field is also extended to include the detected scenarios and yaw data calculated by using the current and previous actor positions. The extended actor field is shown in Figure 4.23b. The calculation of actor yaw value is shown in Equation 4.5 where x and y are the x-axis and y-axis coordinates and $atan2$ is the 2-argument arctangent calculation.

$$yaw_t = atan2(y_t - y_{t-1},\ x_t - x_{t-1}) \tag{4.5}$$



(a) Extended metadata information

(b) Extended actor field

Figure 4.23: Illustration of extended metadata and actor fields of *Scenario Description Format*

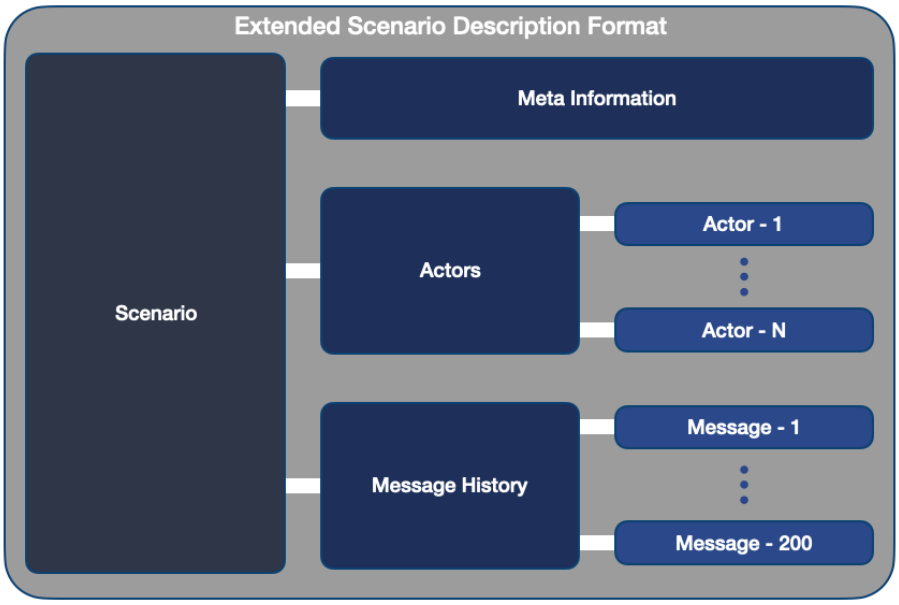


Figure 4.24: The illustration of the extended scenario description

The message history has a size of 200 messages and is initialized as empty. As the messages arrive to the system they are recorded to the history. If the history is full, the oldest entry in the history is removed and the new message is added to the history. The last element of the message history is always the latest message. Once a message arrives, the scenario

detection *node* updates the message history, and the run the feature extraction and scenario extraction on the updated message history.

The metadata information for the detected scenario object in format extended *Scenario Description Format* is generated for the 200 messages in the message history. Therefore, it includes scenarios detected for the last 8 seconds if the message frequency is 25Hz.

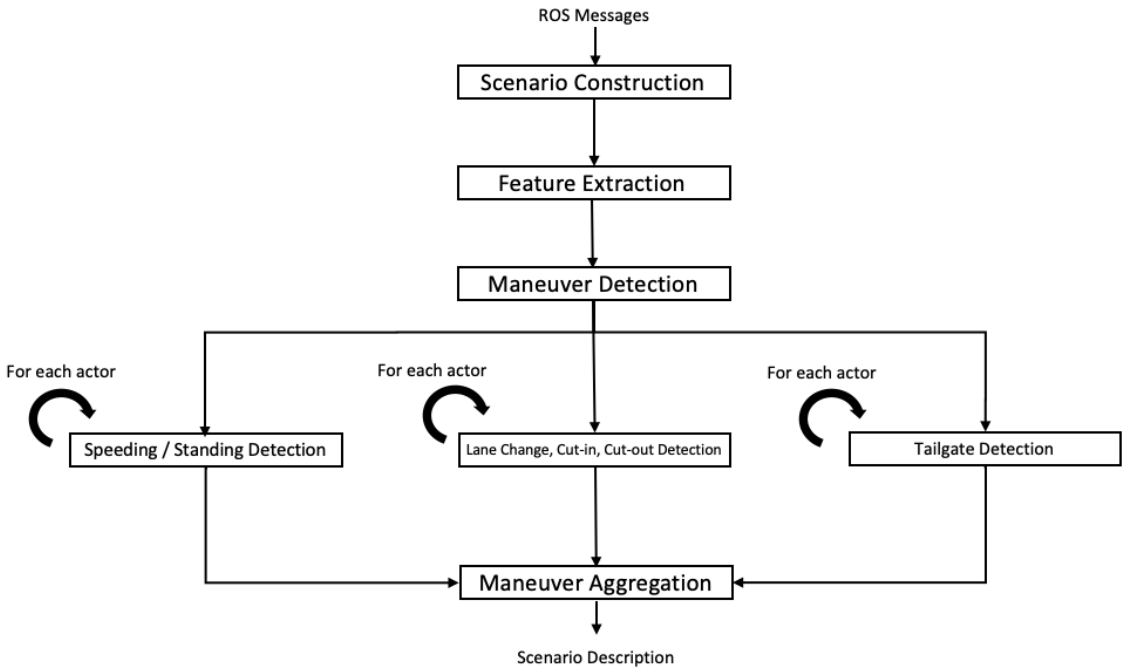


Figure 4.25: The illustration of the live maneuver detection workflow before this work [44], own illustration

Furthermore, the scenario detection algorithm processes ROS messages to detect scenarios. The frequency of the ROS messages is 25Hz; therefore, 25 messages per second. In order to detect scenarios on the live system, the message processing time should be in an acceptable range, i.e. less than 0.04seconds per message which is the time difference between messages; thus, each message can be processed and scenarios be detected in a timely manner.

Kaefer showed in his work [44] that the scenario detection tool is able to detect lane change, cut-in, cut-out, tailgate, speeding and standing maneuvers and events in offline mode which means extracting data from *Rosbags*. However, there is no time constraint for offline runs. Running scenario detection in real-time time requires improvements to the scenario detection workflow. The maneuver and event detection workflow in Kaefer's work [44] is illustrated in Figure 4.25. It can be seen from the figure that the maneuver detection jobs for each actor are run in separate loops.

In order to improve the workflow, the maneuver and event detection jobs are combined in a module which can be mentioned as scenario detection. This not only makes the workflow simpler but also significantly reduces the repeated operations when compared to the previous workflow. The new workflow is shown in Figure 4.26. Moreover, a lookup dictionary for actor/object IDs to message history indices is used in order to have instant access to the message data from the history. This eliminates the search time required to find an object in the message history when it is needed during the maneuver or event detection task.

4.2.3.1 Logging to the Live System

The input ROS messages are obtained by the scenario detector by subscribing to the object detection topic published by the object detector. The messages are in *BackendOutput* format

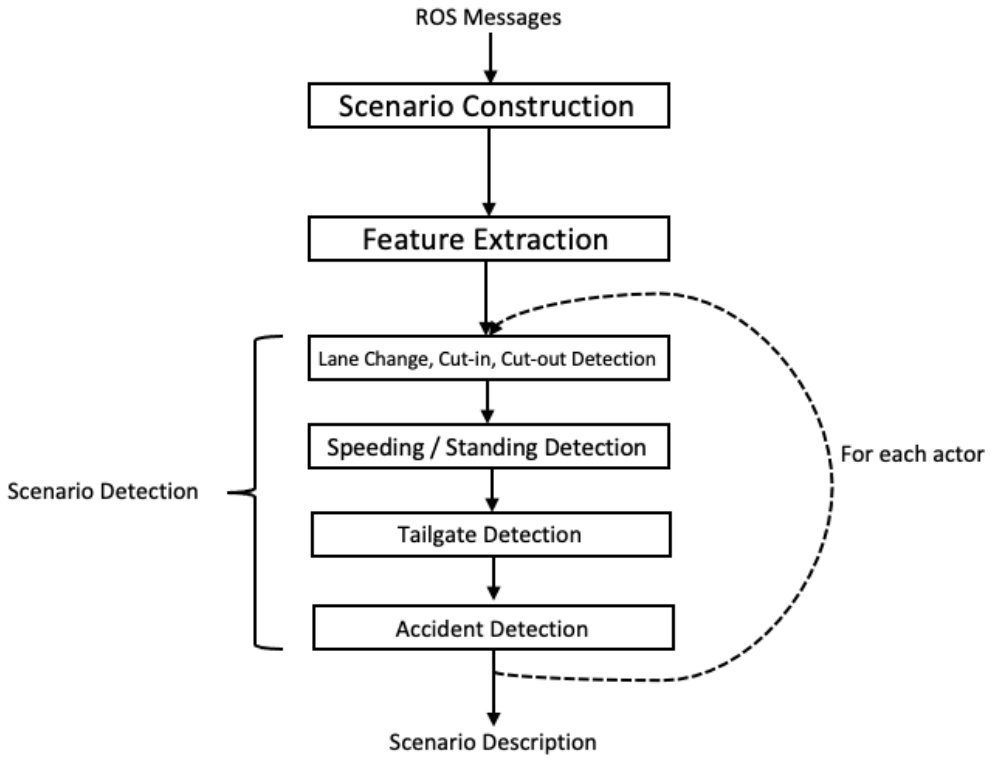


Figure 4.26: The illustration of the live maneuver detection workflow in this work

which stores information such as timestamp and the list of objects. The objects in the list contain velocity, path, ID, object class and dimension information.

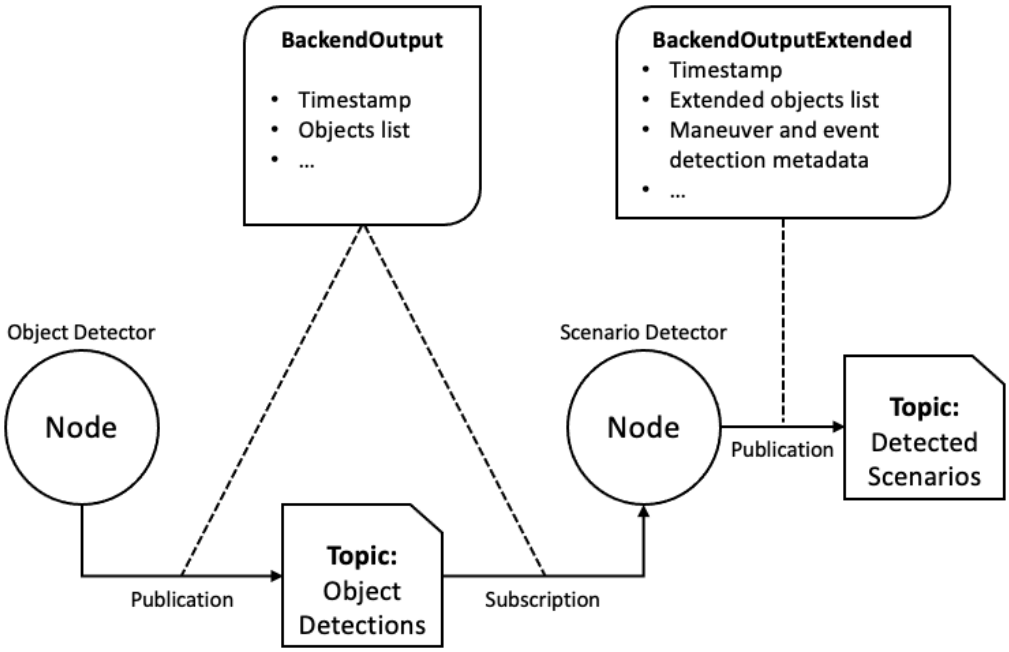


Figure 4.27: Providentia++ Live System ROS communication illustration for scenario detection

The scenario extractor processes the data obtained from the ROS messages as it has been shown in Figure 4.26. The detected maneuvers for each actor alongside the metadata infor-

mation for the ROS messages are stored in the extended *Scenario Description Format* object.

The generated *Scenario Description Format* object is translated to *BackendOutputExtended* object which is a custom ROS message type. This message type is the extended version of the message type received from the subscribed topic so that it stores the detected scenarios and metadata information for the message such as the number of detected lane changes, cut-in, cut-out, accident and tailgate events. Finally, this ROS message is published to a new topic that serves purpose for publishing the detected scenarios. The ROS communication is shown in Figure 4.27.

Chapter 5

Evaluation & Analysis

There are 4 main accomplishments in 2 categories; implementation of a neural network end-to-end training pipeline and lightweight neural network-based lane split detection solution for vehicle domain; scenario detection support on the live system and accident detection for the V2I domain, Providentia++ project. The evaluation of the accomplishments in this work and the analysis of the results are discussed in the Sections 5.2 and 5.3. The metrics used to evaluate the results are discussed in the Section 5.1.

5.1 Metrics

In the evaluation and analysis of this work, there are 4 metrics used which are accuracy, precision, recall, and execution time. The confusion matrix is one of the outputs of the neural network training pipeline implemented in this work; therefore, it is discussed in Section 5.1.1.

5.1.1 Confusion Matrix

It has been discussed in Section 4.1.7 where the workflow of the training component of the neural network pipeline is described and illustrated in Figure 4.14 that the trained model is evaluated over a test dataset after the model is trained based on the parameters. The output of this evaluation is a confusion matrix.

A confusion matrix is one of the metrics used to evaluate the results of a classification task and it is a matrix that gives information about the actual and prediction labels. The rows of the confusion matrix are the actual labels whereas the columns are the predicted labels.

		Predicted		
		Class 1	Class 2	Class 3
Actual	Class 1	a	b	c
	Class 2	d	e	f
	Class 3	g	h	i

Table 5.1: Confusion matrix example for 3 classes

An example confusion matrix for 3 classes is given in Table 5.1. Each entry shows information for the number of elements in the class it belong to for both actual and predicted labels. For example, the cell with value f means that the number of sample predicted as *Class 3* but actually belongs to *Class 2* is f. This shows that the cells on the main diagonal from the top left corner to bottom right corner contain the information regarding the number

of samples predicted correctly. These samples are known as True Positive (TP). If a sample does not belong to a certain class C and the prediction is a class other than the class C, the prediction is known as True Negative (TN). On the other hand, if the sample indeed belongs to the class C but the prediction is other than the class C, the prediction is know as False Negative (FN). If the predictions is class C but the actual label is a class other than C, the prediction is a False Positive (FP).

5.1.2 Accuracy

One of the most used metrics in the evaluation of classification solutions is the accuracy metric. Accuracy shows how often predictions are correct. In other words, it is the ratio of TP to the total number of samples.

However, accuracy is not a good metric if the dataset is imbalanced. For instance, if majority of the samples belong to some class C and a model labels all the samples as class C without considering the input samples, the accuracy of the model would be equal to the ratio of the dominant class in the whole dataset which would be high accuracy score. Therefore, precision and recall metrics are also used to measure the model performance.

5.1.3 Precision

Precision is a metric that gives information about how correct the predictions are among all predictions. Precision is calculated for each class. High precision score for class C means that high ratio of the predicted class C labels are correct. However, this does not mean that most of the samples belong to class C are correctly predicted as class C. The formula for precision is given in Equation 5.1.

$$Precision = \frac{TP}{TP + FP} \tag{5.1}$$

5.1.4 Recall

The recall is a metric that gives information about how correct the predictions for a class are among all the samples belonging to that class. A high recall score for class C means that a high ratio of the samples belonging to class C is correctly predicted as class C. However, this does not mean that all of the predictions for class C belong to class C.

$$Recall = \frac{TP}{TP + FN} \tag{5.2}$$

5.1.5 Execution Time

The execution time metric is used to measure the computational performance of the implementations. This is a useful metric to have an intuition about the usability of the implementations in the work for production systems.

The execution time metric is simply calculated by getting the time difference between end and start of the job executed.

5.2 Lane Split Detection

In this work, a neural network-based lane split detection framework is implemented. In order to train the model in an end-to-end approach, a training pipeline built for the Kubeflow platform is also a contribution in this work.

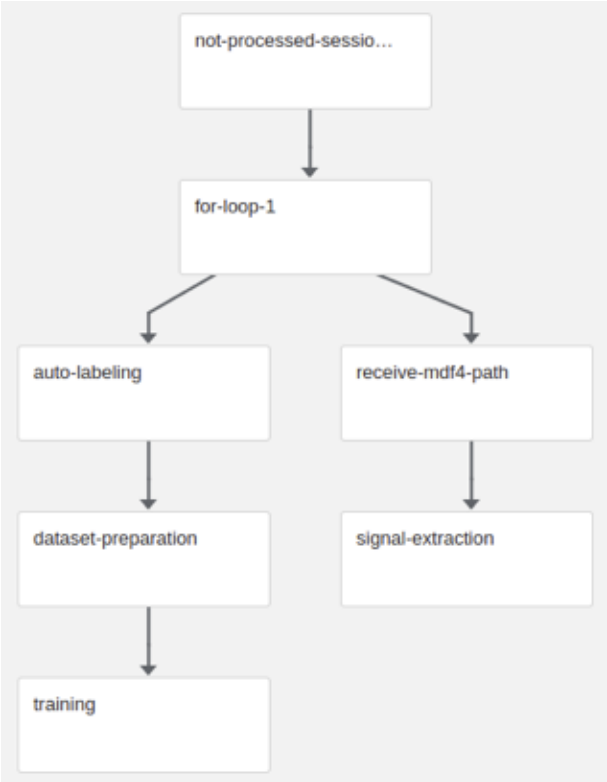


Figure 5.1: Deployed Kubeflow pipeline DAG on the platform

The deployed Kubeflow pipeline can be easily run on any system with the Kubeflow platform whether it is a local or remote deployment. The pipeline runs can be configured with the available parameters described in 4.1 and a model can be trained with a few parameter selections and clicking a button since it is an end-to-end approach. This reduces the time cost that would be spent on training a model with different parameters such as in hyper-parameter tuning task. A screenshot of DAG of the deployed pipeline on Kubeflow platform is shown in Figure 5.1. The names of the components shown in the screenshot differs from the simplified names described in Section 4.1. The first component on the platform is the session selection component and the *receive-mdf4-path* component on the platform is described in the signal extraction section 4.1.4.

The neural network approach to lane split detection problem includes three different models, MLP that trains on a single sample, CNN and a self-attention-based neural network that trains on time windows. Each model is trained on both simulation data and real-world recordings. The results of the trained models and the details of the datasets are described in the following sections.

5.2.1 Dataset

There are two types of datasets used to train neural network models in this work; a dataset generated by a script and a dataset consisting of real-world recordings. Both dataset files are

in CSV format and share the same feature columns.

5.2.1.1 Simulation Data

The simulation dataset is generated by a script that simulates lane splits, s-curved lanes, and lane widening events. The reason for having different event types is to have a trained model that can distinguish similar scenarios since the changes on lanes might be similar in scenarios of both lane split events and a non-split event. For instance, lane widens in both lane split and lane widening events but the rate of change on the lanes might differ. Similarly, left or right lane would be curved in a scenario of lane split similar to a s-curved lane scenario. The difference is both left and right lanes would be curved in a s-curved lane event. There is also added random noise to the simulation data to have a more realistic dataset.

As it has been discussed in Section 4.1.6, each file in the data file corresponds to a single label instance and contain 21 columns; *timestamp*, the label under name *lane boundary event*, speed, 6 columns for each left, center and right lanes; coefficients for three-degree lane boundary polynomials and view range start, end values.

The number of label files in the dataset is 1968. The left and right lane split label files correspond to the %6 of the dataset each; therefore, %88 of the files correspond to *NoLaneSplit* class. If the number of messages is considered; 692k messages in total are broken down as 15k for left lane split and 15k for right lane split.

5.2.1.2 Real-world Data

In order to evaluate the performance of the model on real-world data, recordings from the test drives of BMW prototype cars are used. Some of the sessions are labeled manually with hindsight and some of the sessions are labeled by the auto-labeling component in the pipeline. The feature columns in the real-world dataset are the same as the columns in the simulation dataset.

The manually labeled sessions include the label files for outliers in the lane boundary detection which will be discussed in the results section 5.2.2. There are 12 manually labeled event files in total which include 5 lane split to the left and 7 lane split to the right scenario.

There are 6 driving sessions labeled by the auto-labeling algorithm. There are 3 label files for lane split left and 5 label files for lane split right that result in 131 messages in total.

5.2.2 Results and Analysis

Each model is evaluated in different settings: training and evaluating the simulation data, training on the simulation data but evaluating on manually labeled data, and finally training and evaluating on the real-world recordings with the labels generated by the auto-labeling script. Each evaluation setting shows different aspect of the model performance. Moreover, time to make an inference on 100 sample is also measured for 3 different neural network models.

The model architecture parameters are the same for all evaluation settings. The first and second dense layer sizes are 256 and 128, respectively for the MLP model. The number of filters and kernel sizes are 3 for convolutional layers for both CNN and self-attention-based models. The dropout and batch normalization layers are enabled for all models with a dropout probability of 0.5. Each model is trained for 20 epochs with learning rate 0.001 and batch size 128.

5.2.2.1 Model Performance

The datasets are imbalanced datasets where *NoLaneSplit* class dominates. Therefore, the two mentioned class balancing methods, class weights and event file-based balancing, must be compared first to decide which method to use. In Table 5.2, accuracy scores obtained by training and evaluating MLP model on the simulation dataset are shown. The accuracy is the highest when no balancing method is used since model tends to predict each sample as *NoLaneSplit* which is the dominant class. Therefore, the recall metric is used to determine the get an insight on models ability to generalize. It has been seen that using event file-based balancing significantly improves the recall score for lane split labels. Hence, event file-based balancing is used to balance the dataset.

	Without Balancing	Class Weights	Event File-Based
Accuracy	**0.990**	0.914	0.962
Recall - Left Split	0.262	0.591	**0.824**
Recall - Right Split	0.234	0.621	**0.874**

Table 5.2: Comparison of class balancing methods

The first evaluation setting is the usage of simulation data for both training and evaluation. This setting shows the model's ability to learn from the proposed data and generate decent results. The results are presented in Table 5.3 and the CNN model performs the best in all metrics. The usage of the time windows produces better results as the self-attention model also outperforms MLP model. The reason behind the lower performance score of the self-attention model when it is compared to CNN model is the missing positional encoding implementation. As discussed in Section 2.6, attention network would detect long-term dependencies better than CNN architecture. However, it lacks a useful information to consider which is the relative positions of samples in the whole data. Therefore, positional encoding that represents the position of a sample in the data would be applied to attention-based solutions. However, this encoding implementation is missing in this work which describes the lower performance of self-attention based model.

	MLP	CNN	Self-attention
Accuracy	0.990	**0.994**	0.958
Precision - Left Split	0.972	**0.992**	0.982
Precision - Right Split	0.979	**0.991**	0.977
Recall - Left Split	0.894	**0.944**	0.901
Recall - Right Split	0.924	**0.961**	0.932

Table 5.3: Comparison of model performances solely on simulation data

The second evaluation setting is using manually labeled real-world recording during evaluation on the models trained on simulation data. The results are presented in Table 5.4.

	MLP	CNN	Self-attention
Accuracy	**0.523**	0.496	0.421

Table 5.4: Comparison of model performances for simulation data training, manually labeled data evaluation

The accuracy metric for each model is significantly lower than the achieved values on the evaluation of simulation data. This is due to the difference in the simulation and real-world data. The lane boundary polynomials in the simulation are the representation of the actual lane shapes on the road. However, in the real-world recordings, the lane boundary

polynomials would be outliers due to incorrect lane detection by the prototype camera. An example to outlier lane polynomials are shown in Figure 5.2 where center lane deviates from the supposed position due to the lane split. The model trained on simulation data cannot capture these outliers in the data; therefore, the performance of the models are reduced. The best accuracy is achieved by MLP model. The reason behind that is the single sample based evaluation method of the MLP model since *NoLaneSplit* is still the dominant class and deviations of the center lane might be interpreted by the model as a s-curve lane. This shows that a model trained on simulation cannot be reliably used for real-world use cases.

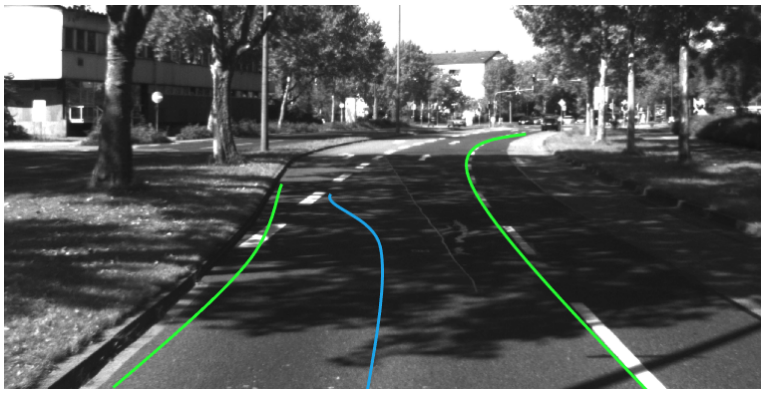


Figure 5.2: Outlier lane boundaries in lane split left scenario, own illustration, image source: [32]

The third and final evaluation setting is using real-world recordings with labels generated by the auto-labeling script. The results are presented in Table 5.4. The results presented in Table 5.5 show significantly lower accuracy scores for all the model types. Though, the CNN model that uses time windows produces the best results among them except for the precision metric of right lane split scenarios.

(a) Lane boundaries before semantic road point detection

(b) Lane boundaries after semantic road point detection

Figure 5.3: Change in lane boundaries in lane split left scenario, own illustration, images source: [32]

The explanation for the low accuracy scores when using labels generated by the auto-labeling component is the failure of the labeling logic as a result of the assumption that semantic road point always exists when there is a lane split scenario. However, this assumption is simply not correct for lane boundary outliers shown in Figure 5.2 since outliers usually occur when the *NewLaneLeft* or *NewLaneRight* road points are not detected. The lane boundaries are usually correct and reflects the simulation data when the semantic road points are correctly detected. In Figure 5.3a, the left and center lane boundaries are deviated to the left due to lane split to left scenario. Few frames after, the semantic road point *NewLaneLeft* shown as orange dot in the figure is detected and the lane boundaries are corrected as seen in Figure 5.3b. Therefore, labels generated by the auto-labeling component only records the messages that come after the semantic road point is detection and label the scenario as *LaneSplitLeft*. This causes a confusion in the data since the lane boundaries in Figure 5.3b are the

lane boundaries that could be detected in regular driving without a lane split scenario.

	MLP	CNN	Self-attention
Accuracy	0.641	**0.748**	0.726
Precision - Left Split	0.341	**0.633**	0.629
Precision - Right Split	**0.689**	0.592	0.576
Recall - Left Split	0.750	**0.950**	0.850
Recall - Right Split	0.886	**0.914**	0.857

Table 5.5: Comparison of model performances for real-world auto-labeling data evaluation

5.2.2.2 Inference Time Measurements

The 3 model types MLP, CNN and self-attention based Neural Network (NN) are compared in terms of the time it takes to make predictions on data. This is an important metric if the model is to be used in real-time where the processing of signal messages promptly is important.

	Model Type		
	MLP	CNN	Self-Attention NN
Time (ms)	58.234	**52.721**	56.521

Table 5.6: Inference time measurements for 3 model types, inference on 100 samples

The inference measurements are recorded on a *fifth-generation dual-core Intel Core i5 CPU* after averaging the 100 inference repetitions with trained the same model types discussed in the model performance discussion section. The inference time measurements are similar for the three models while CNN model is the fastest even though the pooling layers significantly reduce the number of operations for middle layers in the network. The reason might be that the input fed to the first layer of MLP is smaller than other models that use time windows, ($Nx19$) vs ($NxWx19$) for N number of samples and W windows size. The reason CNN is faster than the self-attention based model is that there are two convolution operations in the first layer of the self-attention model which increases the number of operations. Though there is not a significant difference in inference time measurements of the models, CNN would be selected as the model if time constraint is important.

5.3 Providentia++

The contributions to the Providentia++ project in this work are the accident detection implementation and live system support for the existing maneuver detection solution. Therefore, the accident detection performance and the live system support are evaluated to determine if it matches the performance of offline maneuver detection.

5.3.1 Data

There are two Providentia++ recordings stored in *Rosbag* format. One of the recordings contains an accident event and the other file is a regular driving recording.

Both of the recordings contain detected objects and their positions, dimensions, speeds, classes and estimated trajectory information.

5.3.1.1 Accident Recording

The accident recording contains an accident event that involves 3 cars. The type of the accident is combination of head-to-tail collision and lane change collision which are illustrated in Figures 4.22b and 4.22c, respectively. The length of the recording is 1 minute and 59 seconds.

5.3.1.2 Regular Traffic Recording

This recording file is the same file used in the evaluation of Kaefer's work [44]. The length of the recording is 60 seconds. The distribution of the events in the recording can be seen in the results table 5.8.

5.3.2 Results and Analysis

The contributions to the Providentia++ projects are evaluated in two areas: accident detection and the live system support that uses ROS for communication.

5.3.2.1 Accident Detection

There is a single accident instance to be used in the evaluation of the accident detection method. The accident occurs on the highway where a vehicle tries to perform a cut-out but collides with the vehicle in the adjacent lane that it intends to switch to, and then it hits the leading vehicle in its initial lane.

	# Accident Instances	# Cars Involved	Severity
Ground Truth	1	3	Moderate
Detected	2	2	Minor

Table 5.7: Accident detection evaluation results

The accident detection results are shown in Table 5.7. The method is able to detect the accident event described above; however, it only labels the car trying to cut-out and the following car it hit after hitting the adjacent vehicle as in the accident scenario. It misses the vehicle in the adjacent lane. The reason for this missing detection is that the method only checks distance to the following vehicle and completely ignore the traffic participants in adjacent lanes. Moreover, the severity of the accident is incorrectly determined by the method since number of cars involved is a value below the moderate accident threshold. Furthermore, the method detects another accident that does not exist in the ground truth. The reason behind this might be outliers in the data since velocity values jump due to incorrect detection from object detector; therefore, TTC values are not calculated correctly. This leads to detect some sever tailgate events as accidents.

5.3.2.2 Live System Support

The live system support is evaluated in a simulated environment. The scenario detector subscribes to a ROS *topic* to get the object detection messages and detect scenarios. In order to test this workflow, the regular driving recording stored as *Rosbag* is played and the messages are published to the object detection *topic*. Therefore, the scenario detector receives messages from *Rosbag* file.

The scenario detector exactly matches the performance of the offline detection performance.

		Offline	Live-System		
Maneuver Type	**Ground Truth**	**Detected**	**Detected**	**Precision**	**Recall**
Lane Change Left	14	16	16	0.875	1.00
Lane Change Right	55	77	77	0.714	1.00
Cut-in Left	6	13	13	0.462	1.00
Cut-in Right	2	5	5	0.400	1.00
Cut-out Left	1	1	1	1.00	1.00
Cut-out Right	2	5	5	0.400	1.00
Minor Tailgate	139	229	229	0.607	1.00
Moderate Tailgate	26	64	64	0.406	1.00
Severe Tailgate	10	45	45	0.222	1.00
Speeding Vehicle	103	144	144	0.715	1.00
Standing Vehicle	1	1	1	1.00	1.00
Accident	0	1	1	0.00	0.00

Table 5.8: Offline and online scenario detection results

The results show that the live system support does match the performance of the offline scenario detection implementation. Therefore, the live system results are reliable and show that it can be used online. The maneuver detection performance will be determining factor in using the system online. However, the maneuver detection performance is not in the scope of this work.

Another important metric to evaluate the live system support is to measure the scenario detection time for a ROS message. In this measurement, the average detection time for a ROS message is calculated from 1000 consecutive messages.

	Before this work	**In this work**
Average Detection Time (s)	28.2	0.72

Table 5.9: Comparison of scenario detection time

The Table 5.9 shows scenario detection time measurements that are recorded on a virtual machine running on a computer with *fifth-generation dual-core Intel Core i5 CPU*. The detection time has been significantly reduced. Even though the result obtained after the improvements explained in this work is not ideal for the scenario detection in real-time where there are 25 ROS messages need to be processed every second, this time would be reduced when the system runs on a faster computer without a virtual machine setup.

Chapter 6

Outlook

There are 4 major accomplishments in this work; implementation of a neural network-based lane split scenario detection method to improve road perception, an end-to-end model training pipeline, an accident detection method for the Providentia++ stationary perception system, and support for the detection scenarios in the live system.

In this chapter, an outlook is given on the possible use cases for the accomplishments in general and how they can be extended for new use cases.

6.1 Neural Network

There are 3 different neural network models introduced in this work for the detection of lane split scenarios on the road. The aim of using neural networks for the defined task is to have a solution that performs better than the existing rule-based methods thanks to the ability of data-driven approaches to generalize.

Even though the network models introduced in this work are introduced for the purpose of the detection of lane split scenarios, they can be easily extended for the detection of other perception related scenarios such as merging lanes or in general any task that rely on time series data as long as there is a dataset for the task. The model architectures are flexible and agnostic to the data and task.

6.2 End-to-end Training

An end-to-end training approach is adopted with the implementation of a training pipeline to train the neural network models introduced in this work. This approach not only provides convenience and simplicity for training a model but also provides flexibility to adapt the pipeline for the development of other data-driven tasks.

The pipeline designed by applying end-to-end training approach include every step used in machine learning workflows from data extraction to auto-labeling and evaluation of the training model. Since each step is designed separately and combined together in the pipeline, individual steps can be modified for different training use cases. This reduces the time needed to develop a data-driven approach; therefore, more effort can be put into model development for achieving better results.

6.3 Online Scenario Detection

The offline scenario detection system that runs on recordings has been extended to support performing scenario detection on the fly. This has two main use cases: the first is narrowing down the search intervals on the recordings to find a scenario as they happen and providing real-time or near real-time statistics about the current or recent driving conditions on the road.

In the future, ITS systems such as Providentia++ can be used to notify the traffic participants on the road or the authorities in case of emergencies as the scenario detection becomes faster and more robust.

Chapter 7

Conclusion

The main goal of this work is to improve perception which is one of the core modules in the AVs, and through that the safety of the traffic participants. The perception sensing in the AVs starts with the sensors installed on the vehicle but it does not necessarily rely on the vehicle itself but it might also be supported by the V2I systems such as ITS. Hence ADAS features depend on a reliable perception solution, it has been focused in this work to contribute improvements for perception in both vehicles and ITS. Thereby, the listed tasks are accomplished:

1. Implementation of neural network-based approach for lane split scenario detection
2. Development of an auto-labeling method
3. Creation of an end-to-end model training pipeline
4. Detection of traffic accidents on the stationary sensor systems
5. Online scenario detection for ITS

For each of the tasks listed above, the values added by the solutions and the possible improvements for the future are highlighted.

1. Implementation of neural network-based approach for lane split scenario detection

   Using rule-based approaches for the road perception module of AVs is a common practice. However, rule-based models fail to generalize the underlying features of data; thus, the system might not be robust to scenario changes. Hence, a neural network-based approach including MLP, CNN and self-attention is adopted for scenario detection of lane splits. The implemented models, especially time window-based models, show that a neural network is able to detect lane split scenarios if high quality data is provided. The model performances can be improved by using deeper models such as a multi-layer CNN or the Transformer architecture [81]. The implemented self-attention method can also be extended with the addition of positional encoding which should improve the performance.

   Moreover, the performance of the models should also be tested on the road. This can be achieved by implementing an efficient forward-pass implementation of the neural network model that could make predictions in real-time. If the model produces reliable results, then the implemented inference logic can be used in the production vehicles by the auto makers.

2. Development of an auto-labeling method

   The size and quality of the data is one of the most important aspects of development of a successful data-driven approach. Moreover, data should be labeled for supervised learning approaches. Since time burden for manually labeling data is high, the labeling processes should be automated. For this reason, a rule-based auto-labeling method that relies on the detection of new lanes is developed for the splitting lane scenario detection problem. This automatize the labeling process; therefore, reduces the time cost for the development of a data-driven approach.

   The generated labels are based on the existence of semantic road points for new lanes which is not realistic in general. Therefore, the auto-labeling tool can be improved by extending the rule-based method with lane geometry logic such as checking the distance between left and right lanes. Another improvement might be to use an active learning approach to generate labels with a deep neural network model.

3. Creation of an end-to-end model training pipeline

   An end-to-end neural network training pipeline is created as part of this work to provide flexibility and ease of use for model development. Currently, training pipeline runs are initiated manually but this can be improved by the addition of support of recurring runs or automated runs that would train a model as the dataset gets larger.

4. Detection of traffic accidents on stationary sensor systems

   The head-to-tail type traffic accidents are detected by a logic-based method that uses TTC and distance to leading vehicle information. However, other accident types such as road barrier collisions that involve single vehicle or collisions occurring during lane changes that involve vehicles on adjacent lanes are not covered. The rule-based method can be extended by using additional information such as the path of the vehicles and yaw information . However, all these features rely on the detection made by another node in the system. The rule-based accident detection performance heavily relies on the quality of the data provided by the object detection system. In the future, a neural network based approach that uses images instead of time series data can combine the object and scenario detectors.

5. Online scenario detection for ITS

   The offline scenario detection system that detects vehicle maneuvers and events such as lane changes, cut-in, cut-out, tailgate, speeding or standing vehicles is extended to support as part of this work. The detection performance of online system matches the performance of the offline one. Furthermore, the scenario detection time is significantly improved since the detection is 40 times faster. Further improvements to the detection time is possible by switching to a more efficient programming language such as C++ or combining the feature extraction and scenario detection modules together to additionally reduce the number of operations per scenario detection for a message.

List of Figures

2.1 Lane boundary polynomials projected on the road image [76] 5
2.2 *Topic* based publish/subscribe communication model [72] 8
2.3 Time graph sensor data of a vehicle [38] 8

3.1 Overview of automated driving navigation architecture [79] 11
3.2 Examples of perception module types [73] 12
3.3 Corner case detection system [18] 12
3.4 Examples of outlier types [16] 13
3.5 Providentia sensors stationed on one of the measurement points on the A9 autobahn [23] 15
3.6 Stages of the scenario dataset generation [44] 16

4.1 Overview of the Kubeflow pipeline as DAG 20
4.2 An example Kubeflow pipeline definition 21
4.3 Illustration of the data selection component 24
4.4 Illustration of the signal extraction component 25
4.5 Illustrations of lane split scenarios 26
4.6 Illustration of the auto-labeling interactions with pipeline resources 26
4.7 Illustrations of lane split scenarios with semantic road points 27
4.8 Illustration of the timestamp labeling for lane split to right scenario 27
4.9 Plot of the semantic road point signals for scenario in Figure 4.8 28
4.10 An example label file for a data trace generated by the auto-labeling component. *laneBoundaryEvent* denotes the label for the time interval. 29
4.11 Illustration of the dataset preparation component interactions with pipeline resources 29
4.12 Illustration of the training data generation for lane split to right scenario . . . 30
4.13 Illustration of the lane boundaries detected by the car camera 31
4.14 Illustration of the training component's interactions 32
4.15 Illustration of the training component's workflow 33
4.16 Illustration of the MLP model with all optional layers enabled 33
4.17 Illustration of the input generation with *sliding window*, own illustration (source: [67]) 34
4.18 Illustration of the CNN model with all optional layers enabled 35
4.19 Illustration of the self-attention based model with all optional layers enabled . 36
4.20 The illustration of the scenario description before this work [44] 38
4.21 The illustration of maneuver detection and construction of the scenario 39
4.22 Illustration of some possible accident types on highway 40
4.23 Illustration of extended metadata and actor fields of *Scenario Description Format* 41
4.24 The illustration of the extended scenario description 41
4.25 The illustration of the live maneuver detection workflow before this work [44], own illustration 42

4.26 The illustration of the live maneuver detection workflow in this work 43
4.27 Providentia++ Live System ROS communication illustration for scenario detection 43

5.1 Deployed Kubeflow pipeline DAG on the platform 47
5.2 Outlier lane boundaries in lane split left scenario, own illustration, image source: [32] 50
5.3 Change in lane boundaries in lane split left scenario, own illustration, images source: [32] 50

List of Tables

3.1 The overview of the detected maneuvers [44] 17

4.1 Pipeline parameters and their descriptions 22
4.2 Overview of the PVs and PVCs used in the pipeline 23
4.3 Kubernetes *Secret* used in the pipeline 23
4.4 The label names and the corresponding enumerated labels 30

5.1 Confusion matrix example for 3 classes 45
5.2 Comparison of class balancing methods 49
5.3 Comparison of model performances solely on simulation data 49
5.4 Comparison of model performances for simulation data training, manually labeled data evaluation 49
5.5 Comparison of model performances for real-world auto-labeling data evaluation 51
5.6 Inference time measurements for 3 model types, inference on 100 samples . . 51
5.7 Accident detection evaluation results 52
5.8 Offline and online scenario detection results 53
5.9 Comparison of scenario detection time 53

Bibliography

[1] Abiodun, O. I., Jantan, A., Omolara, A. E., Dada, K. V., Mohamed, N. A., and Arshad, H. "State-of-the-art in artificial neural network applications: A survey". In: *Heliyon* 4.11 (2018), e00938.

[2] AdaptIVe. *AdaptIVe project*. 2017. URL: https://www.adaptive-ip.eu/ (visited on 03/20/2022).

[3] Administration, N. H. T. S. and Transportation, U. D. of. *Traffic Safety Facts 1998-a Compilation of Motor Vehicle Crash Data from the Fatal Accident Reporting System and the General Estimates System*. Createspace Independent Publishing Platform, 1999.

[4] Aeberhard, M., Kühbeck, T., Seidl, B., Friedl, M., Thomas, J., and Scheickl, O. "Automated Driving with ROS at BMW". In: *ROSCon 2015 Hamburg*. 2015.

[5] Albawi, S., Mohammed, T. A., and Al-Zawi, S. "Understanding of a convolutional neural network". In: *2017 international conference on engineering and technology (ICET)*. Ieee. 2017, pp. 1–6.

[6] Aljawarneh, S., Anguera, A., Atwood, J. W., Lara, J. A., and Lizcano, D. "Particularities of data mining in medicine: lessons learned from patient medical time series data analysis". In: *EURASIP Journal on Wireless Communications and Networking* 2019.1 (2019), pp. 1–29.

[7] Angadi, M. C. and Kulkarni, A. P. "Time Series Data Analysis for Stock Market Prediction using Data Mining Techniques with R." In: *International Journal of Advanced Research in Computer Science* 6.6 (2015).

[8] Arriaza, J. "A Study of Autonomous Vehicles: Background, Current Issues, & Outlook of Self-Driving Cars". In: (2021).

[9] Authors, T. K. *Introduction to the Pipelines SDK*. 2022. URL: https://www.kubeflow.org/docs/components/pipelines/sdk/sdk-overview/ (visited on 05/02/2022).

[10] Authors, T. K. *Kubeflow Documentation*. 2022. URL: https://www.kubeflow.org/docs/started/introduction/ (visited on 04/30/2022).

[11] Authors, T. K. *Kubernetes Documentation*. 2022. URL: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ (visited on 04/30/2022).

[12] Authors, T. K. *Secrets - Kubernetes*. 2022. URL: https://kubernetes.io/docs/concepts/configuration/secret/ (visited on 05/12/2022).

[13] Authors, T. K. *Volumes - Kubernetes*. 2022. URL: https://kubernetes.io/docs/concepts/storage/volumes/ (visited on 05/12/2022).

[14] Banerjee, K., Notz, D., Windelen, J., Gavarraju, S., and He, M. "Online camera lidar fusion and object detection on hybrid data for autonomous driving". In: *2018 IEEE Intelligent Vehicles Symposium (IV)*. IEEE. 2018, pp. 1632–1638.

[15] Barrachina, J., Sanguesa, J. A., Fogue, M., Garrido, P., Martinez, F. J., Cano, J.-C., Calafate, C. T., and Manzoni, P. "V2X-d: A vehicular density estimation system that combines V2V and V2I communications". In: *2013 IFIP Wireless Days (WD)*. IEEE. 2013, pp. 1–6.

[16] Blazquez-Garcia, A., Conde, A., Mori, U., and Lozano, J. A. "A review on outlier/anomaly detection in time series data". In: *ACM Computing Surveys (CSUR)* 54.3 (2021), pp. 1–33.

[17] Bogdoll, D., Nitsche, M., and Zöllner, J. M. "Anomaly Detection in Autonomous Driving: A Survey". In: *arXiv preprint arXiv:2204.07974* (2022).

[18] Bolte, J.-A., Bar, A., Lipinski, D., and Fingscheidt, T. "Towards corner case detection for autonomous driving". In: *2019 IEEE Intelligent vehicles symposium (IV)*. IEEE. 2019, pp. 438–445.

[19] Breitenstein, J., Termöhlen, J.-A., Lipinski, D., and Fingscheidt, T. "Systematization of Corner Cases for Visual Perception in Automated Driving". In: *2020 IEEE Intelligent Vehicles Symposium (IV)*. IEEE. 2020, pp. 1257–1264.

[20] bussgeldkatalog.org. *Sicherheitsabstand: Welchen Abstand schreibt die StVO vor? [Safety distance: What distance does the StVO stipulate?]* 2022. URL: https://www.bussgeldkatalog.org/sicherheitsabstand/ (visited on 05/09/2022).

[21] Chandola, V., Banerjee, A., and Kumar, V. "Anomaly Detection: A Survey. ACM Computing Surveys". In: *vol* 41 (2009), p. 15.

[22] Chien, Y.-R., Chen, J.-W., and Xu, S. S.-D. "A multilayer perceptron-based impulsive noise detector with application to power-line-based sensor networks". In: *IEEE Access* 6 (2018), pp. 21778–21787.

[23] Creß, C. and Knoll, A. C. "Intelligent Transportation Systems With The Use of External Infrastructure: A Literature Survey". In: *arXiv preprint arXiv:2112.05615* (2021).

[24] Creß, C., Zimmer, W., Strand, L., Lakshminarasimhan, V., Fortkord, M., Dai, S., and Knoll, A. "A9-Dataset: Multi-Sensor Infrastructure-Based Dataset for Mobility Research". In: *arXiv preprint arXiv:2204.06527* (2022).

[25] Da Silva, I. N., Spatti, D. H., Flauzino, R. A., Liboni, L. H. B., and Reis Alves, S. F. dos. "Artificial neural networks". In: *Cham: Springer International Publishing* 39 (2017).

[26] Dingyi, Y., Haiyan, W., and Kaiming, Y. "State-of-the-art and trends of autonomous driving technology". In: *2018 IEEE International Symposium on Innovation and Entrepreneurship (TEMS-ISIE)*. IEEE. 2018, pp. 1–8.

[27] Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., and Koltun, V. "CARLA: An open urban driving simulator". In: *Conference on robot learning*. PMLR. 2017, pp. 1–16.

[28] Hi-Drive. *Hi-Drive project*. 2020. URL: https://www.hi-drive.eu/ (visited on 03/23/2022).

[29] Durango-Cohen, P. L. "A time series analysis framework for transportation infrastructure management". In: *Transportation Research Part B: Methodological* 41.5 (2007), pp. 493–505.

[30] Esling, P. and Agon, C. "Time-series data mining". In: *ACM Computing Surveys (CSUR)* 45.1 (2012), pp. 1–34.

[31] Garbin, C., Zhu, X., and Marques, O. "Dropout vs. batch normalization: an empirical study of their impact to deep learning". In: *Multimedia Tools and Applications* 79.19 (2020), pp. 12777–12815.

[32] Geiger, A., Lenz, P., Stiller, C., and Urtasun, R. "Vision meets Robotics: The KITTI Dataset". In: *International Journal of Robotics Research (IJRR)* (2013).

[33] George, J., Gao, C., Liu, R., Liu, H. G., Tang, Y., Pydipaty, R., and Saha, A. K. "A scalable and cloud-native hyperparameter tuning system". In: *arXiv preprint arXiv:2006.02085* (2020).

[34] George, J. and Saha, A. "End-to-end Machine Learning using Kubeflow". In: *5th Joint International Conference on Data Science & Management of Data (9th ACM IKDD CODS and 27th COMAD)*. 2022, pp. 336–338.

[35] Golubovic, D. and Rocha, R. "Training and Serving ML workloads with Kubeflow at CERN". In: *EPJ Web of Conferences*. Vol. 251. EDP Sciences. 2021, p. 02067.

[36] Guler, S. I., Menendez, M., and Meier, L. "Using connected vehicle technology to improve the efficiency of intersections". In: *Transportation Research Part C: Emerging Technologies* 46 (2014), pp. 121–131.

[37] Heidecker, F., Breitenstein, J., Rösch, K., Löhdefink, J., Bieshaar, M., Stiller, C., Fingscheidt, T., and Sick, B. "An Application-Driven Conceptualization of Corner Cases for Perception in Highly Automated Driving". In: *2021 IEEE Intelligent Vehicles Symposium (IV)*. IEEE. 2021, pp. 644–651.

[38] Heyns, E., Uniyal, S., Dugundji, E., Tillema, F., and Huijboom, C. "Predicting traffic phases from car sensor data using machine learning". In: *Procedia computer science* 151 (2019), pp. 92–99.

[39] Hillel, A. B., Lerner, R., Levi, D., and Raz, G. "Recent progress in road and lane detection: a survey". In: *Machine vision and applications* 25.3 (2014), pp. 727–745.

[40] Hou, Y., Edara, P., and Sun, C. "Situation assessment and decision making for lane change assistance using ensemble learning methods". In: *Expert Systems with Applications* 42.8 (2015), pp. 3875–3882.

[41] Hsu, C.-M., Lian, F.-L., Huang, C.-M., and Chang, Y.-S. "Detecting drivable space in traffic scene understanding". In: *2012 International Conference on System Science and Engineering (ICSSE)*. IEEE. 2012, pp. 79–84.

[42] Idrees, S. M., Alam, M. A., and Agarwal, P. "A prediction approach for stock market volatility based on time series data". In: *IEEE Access* 7 (2019), pp. 17287–17298.

[43] Jain, A. K., Mao, J., and Mohiuddin, K. M. "Artificial neural networks: A tutorial". In: *Computer* 29.3 (1996), pp. 31–44.

[44] Kaefer, A. "Deep Traffic Scenario Mining, Detection, Classification and Generation on the Autonomous Driving Test Stretch using the CARLA Simulator". MA thesis. Technical University of Munich, 2022.

[45] Kieu, T., Yang, B., and Jensen, C. S. "Outlier detection for multidimensional time series using deep neural networks". In: *2018 19th IEEE International Conference on Mobile Data Management (MDM)*. IEEE. 2018, pp. 125–134.

[46] Kim, Z. "Robust lane detection and tracking in challenging scenarios". In: *IEEE Transactions on intelligent transportation systems* 9.1 (2008), pp. 16–26.

[47] Krämmer, A., Schöller, C., Gulati, D., Lakshminarasimhan, V., Kurz, F., Rosenbaum, D., Lenz, C., and Knoll, A. "Providentia-A Large-Scale Sensor System for the Assistance of Autonomous Vehicles and Its Evaluation". In: *arXiv preprint arXiv:1906.06789* (2019).

[48] Krenker, A., Bešter, J., and Kos, A. "Introduction to the artificial neural networks". In: *Artificial Neural Networks: Methodological Advances and Biomedical Applications. InTech* (2011), pp. 1–18.

[49] Krogh, A. "What are artificial neural networks?" In: *Nature biotechnology* 26.2 (2008), pp. 195–197.

[50] Kuhn, C. B., Hofbauer, M., Petrovic, G., and Steinbach, E. “Trajectory-based failure prediction for autonomous driving”. In: *2021 IEEE Intelligent Vehicles Symposium (IV)*. IEEE. 2021, pp. 980–986.

[51] L3Pilot. *L3pilot project*. 2021. URL: https://l3pilot.eu/ (visited on 03/20/2022).

[52] Lai, A. H. and Yung, N. H. “Lane detection by orientation and length discrimination”. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 30.4 (2000), pp. 539–548.

[53] Li, G., Lai, W., Sui, X., Li, X., Qu, X., Zhang, T., and Li, Y. “Influence of traffic congestion on driver behavior in post-congestion driving”. In: *Accident Analysis & Prevention* 141 (2020), p. 105508.

[54] Linarth, A. and Angelopoulou, E. “On feature templates for particle filter based lane detection”. In: *2011 14th International IEEE Conference on Intelligent Transportation Systems (ITSC)*. IEEE. 2011, pp. 1721–1726.

[55] Madić, V., Marković, D., and Mijušković, V. “COMPETITIVE STRATEGIES IN PREMIUM AUTOMOTIVE SEGMENT”. In: *TEME* (2021), pp. 639–659.

[56] Majumdar, S. and Laha, A. K. “Clustering and classification of time series using topological data analysis with applications to finance”. In: *Expert Systems with Applications* 162 (2020), p. 113868.

[57] Marti, E., De Miguel, M. A., Garcia, F., and Perez, J. “A review of sensor technologies for perception in automated driving”. In: *IEEE Intelligent Transportation Systems Magazine* 11.4 (2019), pp. 94–108.

[58] Maurer, M., Gerdes, J. C., Lenz, B., and Winner, H. *Autonomous driving: technical, legal and social aspects*. Springer Nature, 2016.

[59] Mohammed, A. A., Ambak, K., Mosa, A. M., and Syamsunur, D. “A review of traffic accidents and related practices worldwide”. In: *The Open Transportation Journal* 13.1 (2019).

[60] Mozaffari, S., Arnold, E., Dianati, M., and Fallah, S. “Early Lane Change Prediction for Automated Driving Systems Using Multi-Task Attention-based Convolutional Neural Networks”. In: *IEEE Transactions on Intelligent Vehicles* (2022).

[61] NCAP, E. *2018 Automated Driving Tests*. 2022. URL: https://www.euroncap.com/en/vehicle-safety/safety-campaigns/2018-automated-driving-tests/ (visited on 05/08/2022).

[62] Nieto, M., Garcia, L., Scnderos, O., and Otaegui, O. “Fast multi-lane detection and modeling for embedded platforms”. In: *2018 26th European Signal Processing Conference (EUSIPCO)*. IEEE. 2018, pp. 1032–1036.

[63] Nikolaidis, K., Kristiansen, S., Goebel, V., Plagemann, T., Liestøl, K., and Kankanhalli, M. “Augmenting physiological time series data: A case study for sleep apnea detection”. In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer. 2019, pp. 376–399.

[64] Noriega, L. “Multilayer perceptron tutorial”. In: *School of Computing. Staffordshire University* (2005).

[65] O’Shea, K. and Nash, R. “An introduction to convolutional neural networks”. In: *arXiv preprint arXiv:1511.08458* (2015).

[66] Pang, G., Shen, C., Cao, L., and Hengel, A. V. D. “Deep learning for anomaly detection: A review”. In: *ACM Computing Surveys (CSUR)* 54.2 (2021), pp. 1–38.

[67] Parera, C., Redondi, A. E., Cesana, M., Liao, Q., and Malanchini, I. "Transfer learning for channel quality prediction". In: *2019 IEEE International Symposium on Measurements & Networking (M&N)*. IEEE. 2019, pp. 1–6.

[68] Providentia++. *Providentia++ project*. 2020. URL: https://innovation-mobility.com/en/project-providentia/ (visited on 04/05/2022).

[69] Providentia++. *Providentia++ project*. 2020. URL: https://innovation-mobility.com/en/partners-providentia/ (visited on 03/05/2022).

[70] Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A. Y., et al. "ROS: an open-source Robot Operating System". In: *ICRA workshop on open source software*. Vol. 3. 3.2. Kobe, Japan. 2009, p. 5.

[71] Ramchoun, H., Ghanou, Y., Ettaouil, M., and Janati Idrissi, M. A. "Multilayer perceptron: Architecture optimization and training". In: (2016).

[72] Robotics, O. *ROS/Concepts*. 2014. URL: https://wiki.ros.org/ROS/Concepts (visited on 05/02/2022).

[73] Romera, E., Bergasa, L. M., and Arroyo, R. "Can we unify monocular detectors for autonomous driving by using the pixel-wise semantic segmentation of cnns?" In: *arXiv preprint arXiv:1607.00971* (2016).

[74] Saffarzadeh, M., Nadimi, N., Naseralavi, S., and Mamdoohi, A. R. "A general formulation for time-to-collision safety indicator". In: *Proceedings of the Institution of Civil Engineers-Transport*. Vol. 166. 5. Thomas Telford Ltd. 2013, pp. 294–304.

[75] Scheel, O., Nagaraja, N. S., Schwarz, L., Navab, N., and Tombari, F. "Attention-based lane change prediction". In: *2019 International Conference on Robotics and Automation (ICRA)*. IEEE. 2019, pp. 8655–8661.

[76] Tabelini, L., Berriel, R., Paixao, T. M., Badue, C., De Souza, A. F., and Oliveira-Santos, T. "Polylanenet: Lane estimation via deep polynomial regression". In: *2020 25th International Conference on Pattern Recognition (ICPR)*. IEEE. 2021, pp. 6150–6156.

[77] Ting, J.-A., Theodorou, E., and Schaal, S. "A Kalman filter for robust outlier detection". In: *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2007, pp. 1514–1519.

[78] Tormene, P., Giorgino, T., Quaglini, S., and Stefanelli, M. "Matching incomplete time series with dynamic time warping: an algorithm and an application to post-stroke rehabilitation". In: *Artificial intelligence in medicine* 45.1 (2009), pp. 11–34.

[79] Van Brummelen, J., O'Brien, M., Gruyer, D., and Najjaran, H. "Autonomous vehicle perception: The technology of today and tomorrow". In: *Transportation research part C: emerging technologies* 89 (2018), pp. 384–406.

[80] Van Der Horst, R. and Hogema, J. "Time-to-collision and collision avoidance systems". In: (1993).

[81] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. "Attention is all you need". In: *Advances in neural information processing systems*. 2017, pp. 5998–6008.

[82] Vlahogianni, E. I. and Karlaftis, M. G. "Testing and comparing neural network and statistical approaches for predicting transportation time series". In: *Transportation research record* 2399.1 (2013), pp. 9–22.

[83] Walczak, S. "Artificial neural networks". In: *Encyclopedia of Information Science and Technology, Fourth Edition*. IGI Global, 2018, pp. 120–131.

[84] Wang, D., Fan, J., Xiao, Z., Jiang, H., Chen, H., Zeng, F., and Li, K. “Stop-and-wait: Discover aggregation effect based on private car trajectory data”. In: *IEEE transactions on intelligent transportation systems* 20.10 (2018), pp. 3623–3633.

[85] Zhang, Y. and Fricker, J. D. “Quantifying the impact of COVID-19 on non-motorized transportation: A Bayesian structural time series model”. In: *Transport Policy* 103 (2021), pp. 11–20.

[86] Zhao, Z., Zhang, Y., Zhu, X., and Zuo, J. “Research on time series anomaly detection algorithm and application”. In: *2019 IEEE 4th Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*. Vol. 1. IEEE. 2019, pp. 16–20.

[87] Zheng, Y. “Trajectory data mining: an overview”. In: *ACM Transactions on Intelligent Systems and Technology (TIST)* 6.3 (2015), pp. 1–41.