Technische Universität München

TUM School of Computation, Information and Technology

Validating Machine Learning-based Highly Automated Driving Functions by Diversity

Oliver Thomas De Candido

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften (Dr.-Ing.)

genehmigten Dissertation.

Vorsitz:		Prof. Dr.-Ing. habil. Erwin Biebl
Prüfer der Dissertation:	1.	Prof. Dr.-Ing. Wolfgang Utschick
	2.	Prof. Dr.-Ing. Michael Botsch
	3.	Assoc. Prof. Paul E. Hand

Die Dissertation wurde am 11.01.2023 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 24.08.2023 angenommen.

Acknowledgements

First and foremost, I would like to thank Prof. Dr.-Ing. W. Utschick for his continuous support throughout my years at the chair for Methods of Signal Processing (MSV) at the Technical University of Munich (TUM). Thank you for giving me the scientific freedom to explore the ideas presented in this dissertation and for supporting me in all of my research, teaching, and travel opportunities. I am grateful to Prof. Dr.-Ing. M. Botsch and Prof. Dr. P. Hand for reviewing my dissertation, agreeing to join my doctoral examination committee, and for their help throughout this endeavour. Thank you, Prof. Dr.-Ing. E. Biebl, for organising and chairing my doctoral examination. Further, thanks go to my colleague and friend, Dr.-Ing. M. Koller, for our discussions throughout our time at MSV and for proofreading this dissertation.

I would like to express my gratitude to my colleagues and collaborators at MSV, TUM, Technische Hochschule Ingolstadt (THI), Northeastern University (NU), Heinrich-Heine-Universität (HHU), and AUDI who helped shape the researcher I am with insightful discussions, helpful feedback, and a collaborative research environment that allowed us to crystallise our ideas into scientific results. Thanks to you all, I have many fond memories of my time as a doctoral student, and I am grateful I can now call many of you good friends. Furthermore, I would like to thank all the students who directly (or indirectly) worked with me—you helped me immensely, taught me how to lead research projects, and showed me my passion for teaching.

Finally, I would like to thank my family and friends all over the world for the love and support you have given me over the years. This work is dedicated to you.

Abstract

As more Highly Automated Driving (HAD) functions are implemented using Machine Learning (ML) methods, the challenge of validating them becomes ever more important. These challenges arise since classical software validation methods were not developed to validated ML-based HAD functions.

In this dissertation, we present methods to develop validation safety arguments considering various aspects of ML algorithms: From the datasets used to train them, through the feature embeddings within them, to the overall design of the algorithms. On the one hand, we introduce methods to compare the feature embeddings of various Deep Neural Network (DNN) architectures, allowing an engineer to choose an architecture based on more than just the classification performance. We compare the architectures based on their clustering ability in the feature embedding space and visualisations thereof. We argue that a compact clustering of the feature embeddings is a desirable property of a DNN architecture. Thus, we extend this safety argument to create k-means friendly feature embedding spaces. We additionally introduce a method to reject spurious classification outputs.

On the other hand, we discuss and analyse the distributional shifts present in two public highway driving datasets (the highD and the NGSIM datasets). We demonstrate that these negatively affect the performance of ML-based HAD functions by reducing the classification accuracy by up to 70%. On top of that, we introduce a fine-tuning algorithm to trade-off the performance on the source and the target distributions.

Finally, we propose an interpretable Lane Change Detector (LCD) algorithm based on the anomaly detection capabilities of Deep Autoencoders (DAEs). We demonstrate the interpretability of the algorithm, and show how an engineer can choose the parameters according to the desired HAD function performance. Throughout this dissertation, we use a lane change prediction task as a safety-critical HAD function use-case to demonstrate the proposed methods.

Ultimately, the techniques presented in this dissertation can be used individually or in combination to validate various aspects of ML-based HAD functions by diversity. They could, for example, also be used as evidence in a safety case for validation.

Contents

List of Figures

List of Tables

Acronyms

ADS Automated Driving System
ARI Adjusted Rand Index
AV Autonomous Vehicle

C2ST Classifier Two-Sample Test
CNN Convolutional Neural Network

DAE Deep Autoencoder
DNN Deep Neural Network
DT Decision Tree

FC Fully Connected
FIR Finite Impulse Response

GAP Global Average Pooling
GRU Gated Recurrent Unit

HAD Highly Automated Driving

iid independent and identically distributed

k-NN k-Nearest Neighbours

LCD Lane Change Detector
LDA Linear Discriminant Analysis
LSTM Long Term Short Term Memory

MC Multi-Channel
ML Machine Learning
MMD Maximum Mean Discrepancy
MSE Mean Squared Error

NGSIM Next Generation SIMulation

ODD Operational Design Domain

ReLU Rectified Linear Unit
RKHS Reproducing Kernel Hilbert Space
RNN Recurrent Neural Network

SAE Society of Automotive Engineers
SGD Stochastic Gradient Descent

TPR True Positive Rate
TTLC Time to Lane Change

UMAP Uniform Manifold Approximation and Projection

V&V Verification and Validation

XAI Explainable Artificial Intelligence

1 Introduction

The vision of developing and deploying Automated Driving Systems (ADSs) and Autonomous Vehicles (AVs) has been around for decades. It comes with the promise of making our streets safer; there was an average of $32{,}189$ motor vehicle related fatalities over the past decade (2010-2020) in the US alone [1]. However, until recently, developing ADSs was confined to academic areas of research, e.g., [2]. The Society of Automotive Engineers (SAE) defines six levels of automation of road vehicles, ranging from no driving automation and driver assistance functions (levels zero and one), all the way through conditional to full driving automation (levels three through five) [3]. An example of a level three ADS could be an AV which can chauffeur its passengers for sustained amounts of time on the highway; A level five ADS can drive in any environment without constraints. As defined by the SAE, an ADS or Highly Automated Driving (HAD) function[1] are "design-specific functionalities" which can perform part or all of the driving automation at SAE automation levels three through five [3].

One approach to implement the ADS task of observing the environment and outputting the controls of the AV is to split the task into modules [4], e.g., with a sensing, a perception, a prediction, and/or a planning module (green boxes in Fig. 1.1). Each module can contain specific HAD functions, e.g., the perception module can contain a function to detect lane markings on a highway or the prediction module can contain a function to predict the actions of surrounding vehicles. This modular pipeline design gives engineers the ability to design each module according to the engineering requirements, and to verify and validate it accordingly. Another option is to design an ADS in an end-to-end fashion (purple box in Fig. 1.1), i.e., taking the raw sensor inputs and outputting the controls of the vehicle. An end-to-end ADS can be more challenging to design, as an engineer has to take all possible inputs into account when designing the whole system.

To alleviate the burden on engineers to model every possible scenario which AVs

[1]Note, in the SAE definition, the authors define these as driving automation system *features*. However, as not to confuse these *features* with the methods presented in this dissertation, we refer to them as HAD or ADS functions.

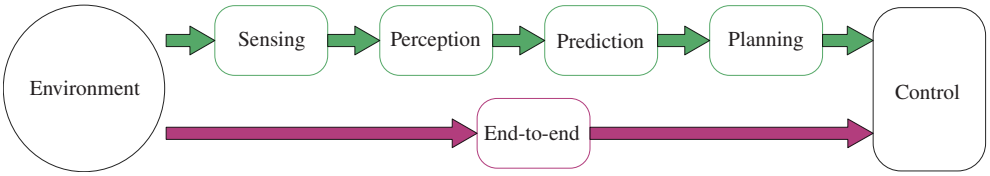


Figure 1.1: A modular design vs. an end-to-end design of an ADS.

might encounter—either in a modular or an end-to-end design—, Machine Learning (ML)-based HAD functions have become popular in recent years. These ML-based HAD functions are able to learn representations from a limited number of training scenarios. Thus, they belong to the class of data-driven functions in contrast to model-driven functions. ML-based algorithms have been implemented in a myriad of different HAD functions, see, e.g., [5] for an overview.

Despite ML-based algorithms' ability to achieve superhuman performance on various tasks, the challenge of verifying and validating the HAD functions incorporating them remains unsolved. The lack of explicit function specifications makes it challenging to apply the current functional safety standards like, e.g., ISO 26262 [6]. Moreover, they require representative datasets to train on, since the quality of the HAD function directly depends on the quality of the data used to train it. An advantage of ML-based HAD functions is their ability to generalise from (relatively few) training scenarios. However, this comes at the price of engineers not always fully understanding the black-box nature of these methods. We explore these aspects (and more) of ML-based HAD functions in Chapter 2.

In this work, we consider ML-based HAD functions in the prediction module of an ADS. To this end, we assume that the perception module, which typically precedes the prediction module, outputs lists of the objects surrounding the ego vehicle with their corresponding attributes. These objects are usually tracked over time such that the input to the ML-based HAD functions are multi-variate time-series data. There are various, publicly available datasets containing such data, e.g., the highD dataset [7] or the Next Generation SIMulation (NGSIM) datasets [8; 9]. On top of that, car manufacturers also collect and manage (proprietary) datasets. Throughout this work, we consider the safety-critical HAD function of predicting when the vehicles surrounding the ego vehicle are going to change lanes. We refer to this as the Time to Lane Change (TTLC) classification task; More details are provided in Section 3.5. An early prediction cannot only increase the comfort of the passengers in the ego vehicle, but it can also ensure that a potential evasive manoeuvre is carried out safely and correctly.

For example, the authors of [10] implement a Recurrent Neural Network (RNN)-based algorithm with Long Term Short Term Memory (LSTM) cells [11] to classify and predict when the vehicle is going to change lanes. The authors of [12] extend this RNN architecture to use Gated Recurrent Unit (GRU) cells [13] and extend the function to also predict the time until the lane change is completed. Recently, the authors of [14] use a similar RNN-based architecture with LSTM cells to predict the time to left lane changes and the time to right lane changes; Based on these time estimates, the driving manoeuvre can be classified.[2] Despite these algorithms displaying state-of-the-art performance in the TTLC problem with multi-variate time-series inputs, the challenge of validating the proposed algorithms is not addressed.

1.1 Contributions

We introduce various methods in this dissertation to address the challenge of validating ML-based HAD functions. These methods focus on different aspects of the ML algorithms, and provide an analysis and evidence for the safety of the HAD function where it is implemented. The scientific publications containing the results related to this dissertation are listed below.

Journal Publication:

1. O. De Candido, M. Koller, and W. Utschick, "Encouraging Validatable Features in Machine Learning-based Highly Automated Driving Functions," *IEEE Trans. on Intell. Vehicles*, vol. 8, no. 2, pp. 1837–1851, 2023. doi: 10.1109/TIV.2022.3171215

Conference Publications:

2. O. De Candido, X. Li, and W. Utschick, "An Analysis of Distributional Shifts in Automated Driving Functions in Highway Scenarios," in *Proc. IEEE 95th Veh. Technol. Conf. (VTC2022-Spring)*. IEEE, 2022. doi: 10.1109/VTC2022-Spring54318.2022.9860453

3. O. De Candido, M. Binder, and W. Utschick, "An Interpretable Lane Change Detector Algorithm based on Deep Autoencoder Anomaly Detection," in *Proc. IEEE Intell. Vehicles Symp. (IV)*. IEEE, 2021, pp. 516–523. doi: 10.1109/IV48863.2021.9575599

[2] Note, details about the aforementioned ML algorithms will be explained in Chapter 3.

4. O. De Candido, M. Koller, O. Gallitz, R. Melz, M. Botsch, and W. Utschick, "Towards Feature Validation in Time to Lane Change Classification using Deep Neural Networks," in *Proc. IEEE 23rd Intell. Transp. Syst. Conf. (ITSC)*. IEEE, 2020, pp. 1697–1704. doi: 10.1109/ITSC45102.2020.9294555

1.2 Outline

This dissertation is organised as follows: In Chapter 2, we introduce research related to guaranteeing or arguing for the safety of ML-based algorithms. We focus on HAD applications wherever applicable, and discuss the challenges related to validating ML-based HAD functions. We end the chapter with a discussion of the overall validation philosophy introduced in this dissertation: *Validation by Diversity*.

In Chapter 3, we discuss the preliminaries related to ML algorithms. We begin by motivating the use of ML algorithms and then introduce the multi-variate time-series data. Following this, we introduce the Deep Neural Network (DNN) architectures and optimisation strategies we investigate in this dissertation. Finally, we introduce the TTLC classification problem as a use-case and the multi-variate time-series datasets used in this work.

In Chapter 4, we introduce the methods to validate ML-based HAD functions by investigating the feature embeddings of various DNN architectures. We first introduce a passive analysis method, and then introduce a method to encourage more meaningful feature embeddings. Furthermore, we introduce a method to reject spurious outputs of an ML algorithm.

From there, we introduce the notion of distributional shifts between highway driving datasets in Chapter 5. We qualitatively and quantitatively analyse the two datasets we use, and show how the distributional shifts between them affect ML-based methods trained on them. Additionally, we introduce a fine-tuning method taking the distributional shifts into account—we are able to trade-off the forgetting of the source distribution and the learning of the target distribution.

In Chapter 6, we introduce an interpretable ML-based Lane Change Detector (LCD) algorithm. We show how an engineer can choose the parameters for this method depending on the performance they require, and investigate the interpretability of the proposed LCD algorithm.

Finally, in Chapter 7, we summarise the methods introduced in this dissertation and an outlook onto further work.

2 Safety of Machine Learning Algorithms

In this chapter, we discuss the safety of Machine Learning (ML) algorithms used in safety-critical applications. We primarily focus on methods pertaining to ML-based Highly Automated Driving (HAD) functions, but we discuss general applications and methods where appropriate. We begin by exploring the various techniques to make ML algorithms interpretable in Section 2.1. Following this, we discuss methods to create safety envelopes in Section 2.2. In Section 2.3 and Section 2.4, we introduce methods to verify and to validate ML-based HAD functions, respectively. Ultimately, in Section 2.5, we discuss the methods introduced in this dissertation and how they could be used as evidence to support a safety case to validate ML-based HAD functions.

For an overview of where ML is implemented in HAD functions, the reader can refer to [5]. For a more in-depth review of ML safety topics (and possible solutions), the reader can refer to [19; 20; 21].

2.1 Machine Learning Interpretability

In recent years, the field of Explainable Artificial Intelligence (XAI) has become popular within the ML research community as a way to better interpret and explain why ML algorithms make their decisions, see, e.g., [22; 23; 24]. Fundamentally, these explanations should be human interpretable. In general, there are two sub-goals within XAI: (i) to make the model intrinsically transparent; or (ii) to provide post-hoc explanations of the decisions an ML algorithm makes [25].[1] Researchers have further proposed interpretability methods focusing on the inner working of an ML algorithm, see, e.g., [27] and references therein.

An example of how to make an ML algorithm intrinsically transparent is to use simple ML techniques such as linear models or Decision Trees (DTs) (as long as they are not extremely deep). Furthermore, an engineer could compose an ML algorithm

[1] These categorisations of the XAI sub-goals are also sometimes referred to as the interpretability and the explainability of ML algorithms, respectively. For further details on this topic, see, e.g., [26, Ch. 2].

out of interpretable parts which might include more complicated techniques, e.g., Deep Neural Networks (DNNs). Intrinsically interpretable methods allow an engineer to fully comprehend the whole model, and to directly understand why a decision was made. Methods to provide post-hoc explanations of ML algorithms can be based on visualisation tools, e.g., heat-mapping methods [28; 29], or on providing textual explanations for a given output, e.g., [30; 31]. These methods not only help an engineer understand why an ML algorithm makes its decision but also to extract information or insights from the learned algorithms.

Within the Autonomous Vehicle (AV) community, researchers have proposed XAI method in various HAD functions. For example, in [32], the authors introduce a steering angle control algorithm based on driving video data. They provide visual interpretability using attention heat-maps, highlighting regions in the 2D-image that might influence the algorithm's decision. This form of XAI is an example of a post-hoc explanation of the regions in the input space influencing the ML algorithm. In [33], the authors propose the use of interpretable representations in an end-to-end ML-based motion planner. These are used to visualize the motion forecasting and to quantify the corresponding object detection performance. This form of XAI is an example of making the ML algorithm intrinsically interpretable.

2.2 Safety Envelopes

When cyber-physical systems such as AVs employ ML-based algorithms, one line of research to maintain their safety is to consider the system as a whole, i.e., abstract everything to a system level including all of the software and all of the hardware. This allows engineers to ensure the safety of the whole system irrespective of what methods are implemented in its subparts. For example, we can ensure that the planned actions of the AV remain safe whilst simultaneously providing a positive utility to the end-user, even if the Automated Driving System (ADS) modules contains ML-based functions. Generally, methods which enable safety on a system level are referred to as safety envelope or safety bag approaches [34].

In [35], the authors propose creating a safety envelope around the AV by encoding the rules of safe driving manoeuvres, e.g., safe distances or the rules of right-of-way, and checking whether they are infringed upon. They call this safety envelope algorithm "responsibility-sensitive safety." These rules can be checked by a computer before the AV tries to take an action to ensure that the system remains safe. For example, the safety envelope algorithm will check whether the minimum distance to the surrounding vehicles is maintained before performing the action; if this is not the case, the action

will not be carried out. This work has been extended and used in a myriad of work within the AV research community, e.g., [36; 37; 38]. A similar approach to building a safety envelope was pursued in [39] to define safe control policies of AVs—they call their approach the "safety force field." These safety envelope approaches fall under the doer/checker approach to safety [34]. On the one hand, we have a (possibly) ML-based HAD function intending to carry out an action, and on the other hand, we have a system to check that these intended actions are safe. The checker systems, are usually implemented using traditional software techniques, thus, it is possible to validate them using traditional validation methods.

By abstracting the AV to a system level, we can ensure that the AV only takes safe and useful actions by defining a safety envelope around it. For example, we know that right-of-way will not be taken from other vehicles and that the AV will maintain a safe distance to all surrounding vehicles. These rules are checked before the AV attempts to carry out an action. However, a safety envelope argument does not address the safety of the ML-based HAD functions themselves. This is a limiting factor of safety envelope approaches when considering ML-based HAD functions.

2.3 Machine Learning Verification

According to the IEEE standards vocabulary, *verification* is defined as "confirmation, through the provision of objective evidence, that specified requirements have been fulfilled" [40]. This was informally defined by software engineer Barry Boehm via the question "*Am I building the product right?*" [41]. In this section, we consider verification as methods to ensure that the original requirements of a system, function, or product are properly fulfilled. Therefore, in order to verify an ML-based HAD function, specific requirements for the function must first be defined and their fulfilment subsequently checked.

In terms of the verification of ML-based methods, most researchers focus on abstracting the specific requirements into a set of constraints, e.g., the requirement that the output of the ML algorithm always lies within a certain range. Solvers such as mixed integer linear programming solvers or satisfiability modulo theory solvers, can then be used to verify whether these constraints are fulfilled. For DNNs with Rectified Linear Unit (ReLU) activation functions, the authors of [42] transform the neurons of the network into Boolean combinations of constraints and use a satisfiability modulo theory solver to verify if the constraints are satisfiable or not. This method was successfully implemented to verify the output properties of a DNN in an airborne collision avoidance system, e.g., whether the output of the DNN controller would lead

to a collision for certain initial conditions. This work was extended in [43] to allow for the verification of larger DNNs.

An approach to verify the adversarial robustness of Convolutional Neural Networks (CNNs) with ReLU activation functions is presented in [44]. In their work, the authors introduce a framework they call AI^2 which represents CNNs using an abstract interpretation (see [45]) allowing them to approximate the program's behaviour and verify its outputs. They applied their verification method to an image classifier where the input images are perturbed in an attempt to fool the DNN. They are able to verify the robustness of the classifier against these perturbations on multiple datasets. In [46], the authors introduce a mixed integer linear program formulation of a DNN with ReLU activation functions. Given this formulation, the outputs of the DNN can be verified—they experiment with various classification tasks and show that their method can verify the outputs of shallow DNNs.

In summary, we observe that most verification methods of ML algorithms focus on DNNs with ReLU activation functions due to their favourable properties. Since we focus on validating ML-based HAD functions in this dissertation, we do not explore the verification of ML-based methods in more detail.[2] Moreover, to the best of our knowledge, no ML verification methods have explicitly been evaluated on ML-based HAD functions.

2.4 Machine Learning Validation

The definition of *validation* according to the IEEE standards vocabulary is "confirmation, through the provision of objective evidence, that the requirements for a specific intended use or application have been fulfilled" [40]. In contrast to verification, Boehm informally defined this via the question "*Am I building the right product?*" [41]. Thus, the difference between verification and validation is whether we are checking that the system fulfils the specified requirements (verification) or whether it fulfils the intended use-case (validation). Therefore, in order to validate ML-based HAD functions, we consider methods which provide evidence that the ML algorithm produced the intended HAD function.

As defined in [48, Sec. 2.4], the validation of ML-based HAD functions generally falls into three categories: (i) data validation; (ii) qualitative validation; or (iii) quantitative validation. Data validation mainly focuses on analysing the training and test data used to train and evaluate the ML algorithms, respectively. If the

[2] For an overview of verification methods of ML algorithms, we refer the interested reader to [19, Sec. 4] or [47, Sec. 6].

data are not representative enough, the missing scenarios must be added to ensure that the data coverage is improved upon. For example, if the training data only has right-hand-drive scenarios, any ML algorithm trained on these data would likely fail when exposed to left-hand-drive scenarios. Qualitative and quantitative validation methods focus on the ML algorithms themselves. To qualitatively validate an ML algorithm, methods should be implemented to help an engineer understand the inner representations or the reasons for a certain output (similar to ML interpretability, *cf.* Section 2.1). Quantitative validation focusses on extracting rules or concepts from the learned ML algorithms. Overall, qualitative and quantitative validation methods can be used to better understand decision processes within the ML algorithms. In terms of ML-based methods for HAD, most research focuses on the data validation and qualitative validation [48].

In terms of data validation, various researchers are working on methods to generate simulated test samples to test ML-based algorithms. A recent research project called PEGASUS [49], promoted by the German Federal Ministry for Economic Affairs and Energy, focused on creating a scenario database to test AVs and validate HAD functions. These scenarios are categorised based on their abstraction level, ranging from functional scenarios through logical to concrete scenarios. This allows an engineer to test ML-based HAD functions at these different abstraction levels. In [50], the authors train two generative ML-based models on the highD dataset [7], which allows them to generate new, highway driving trajectories. This work was extended in [51] to generate more realistic driving trajectories. These trajectories can then be used to test other ML-based driving functions as part of a simulated dataset validation. The authors of [52], introduce another scenario generation method based on dynamic programming estimating the distribution of failures of an AV. They use this distribution to create specific scenarios which cause the AV to fail. Using data validation methods, an engineer is able to test an already trained ML-based HAD function to find scenarios where the function fails.

Quantitative validation aims to extract rules or concepts from DNNs, see, e.g., [53; 54]. An understanding of these rules or concepts can be used in a safety argument, e.g., by extracting the rules within a DNN, an engineer can more easily use expert knowledge to validate the learned function. These methods are similar to the inner interpretability methods introduced in [27] (*cf.* Section 2.1). Since we primarily focus on dataset validation and qualitative validation methods, we do not elaborate further on the quantitative validation methods. Moreover, to the best of our knowledge, these methods have not been explicitly used in HAD applications.

A majority of the work to qualitatively validate ML-based methods use inter-

pretability methods, e.g., visualisation techniques or textual explanations, to aid in explaining why, or interpreting how, a certain decision was made. Thus, these methods are closely related to work in ML interpretability (*cf.* Section 2.1). In the following, we explore some qualitative validation methods leveraging ML interpretability.

In [55], the authors propose training an ADS controller end-to-end using DNNs, however, they additionally train a heat-mapping method and textual explanations to interpret the current outputs of their model. This allows them to both visually and textually validate that the AV makes correct driving manoeuvres based on two interpretability techniques. In [56], the authors introduce a method to extract interpretable features from multi-variate time-series data using heat-mapping techniques—ultimately, producing an enhanced dataset which can be used in downstream tasks. This dataset was used in [57], where a lane change prediction architecture is proposed which is interpretable by design. The architecture is based on a mixture-of-experts design, splitting the time-series input domain into distinct segments and training an interpretable classifier on each time segment. Since the latter method primarily focuses on building an interpretable architecture by design, it has the advantage that the ML-based HAD function can be directly validated.

We believe that a direct comparison of validation methods is challenging since each method investigates, or attempts to improve, a different aspect of the ML algorithm. Therefore, an engineer should have a toolbox of validation methods at hand to appropriately validate the ML-based HAD function in a given situation.

2.4.1 Validation via Safety Cases

Since ML-based HAD functions lack explicit function specifications, they are challenging to validate using the current functional safety standards. The challenge of applying the Verification and Validation (V&V) methods from ISO 26262 [58] given ML-based functions is explored in [6]. The lack of specific software requirements when designing ML-based algorithms makes applying the V-model[3] (*cf.* [58, Part 6])

[3]The ISO 26262 standard is based on V-models as reference processes for the V&V of different phases of the product design. Within the standard, a V-model is proposed for the V&V of the software development. These methods and processes provide a strategy to develop error-free software. The left arm of the V-model describes the methods and processes to design the software at different abstraction levels, e.g., the specification of software safety requirements or the software architectural design, culminating in the technical implementation of the software. The right arm of the V-model describes the methods and processes to verify and to validate, i.e., to test and to integrate, the implemented software units. This is where the V&V of the implemented software is performed—the software is verified and validated against the software design requirements from the left arm of the V-model. All stages of software development must be well documented and checked against the related technical safety requirements.

for V&V challenging. The safety of the intended functionality standard, ISO/PAS 21448 [59], aimed to overcome these challenges by abstracting the requirements away from hardware and software (including ML-based driving functions), to scenarios where the function is intended and designed to be used. These scenarios are either known or unknown and are either hazardous or non-hazardous. It requires that driving functions operate in the absence of reasonable risk, i.e., engineers should reduce the reasonable risk in hazardous scenarios (both known or unknown). However, the ISO/PAS 21448 mainly focusses on driver assistance functions instead of HAD functions [60].

AV researchers have proposed generating safety cases to validate AVs incorporating ML-based algorithms, see, e.g., [61; 34; 48]. Safety cases—also known as safety assurance cases—have been used in safety engineering for decades. Their purpose is to "communicate a clear, comprehensive and defensible argument that a system is acceptably safe to operate in a particular context" [62]. They are built of three elements: (i) safety requirements; (ii) safety arguments; and (iii) supporting evidence. The safety arguments should provide enough evidence to argue that the system is "safe enough" in a given context.[4] The *goal structuring notation* [62; 63] is one example of how to document a safety case. Fundamentally, a safety case is built on the supporting evidence, so the methods to provide this evidence are crucial.

In [64; 65; 66; 67], the authors take the first steps of proposing safety cases to argue for the safety of ML-based HAD functions. They do not, however, propose methods to be used as supporting evidence for their safety claims, as this is out of the scope of the papers. The methods presented in this work could be used as evidence for the safety goals in the proposed safety cases. On the other hand, these methods can be used independent to a safety case.

2.5 Validation Safety Arguments by Diversity: Overview

In this dissertation, we propose safety argumentation methods by considering different aspects of an ML-based HAD function. Building upon the previously mentioned work, we propose both qualitative validation methods and dataset validation methods (*cf.* Section 2.4). The safety argument methods also build upon interpretability research (*cf.* Section 2.1).

An overview of the proposed validation methods can be seen in Fig. 2.1. In

[4]Note, both the ISO 26262 and the ISO/PAS 21448 standards build a safety case using safety arguments, however, the V&V methods proposed in these standards do not consider ML-based HAD functions due to the difficulty of defining ML-based function specifications and the related safety requirements.

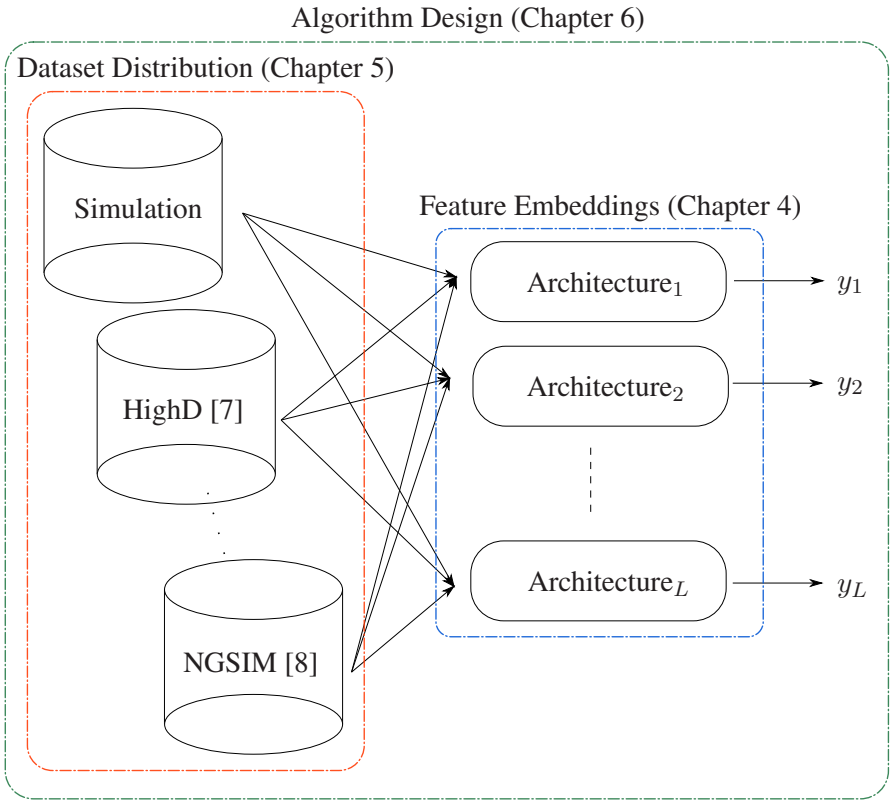


Figure 2.1: Overall philosophy of *Validation by Diversity*: Various methods to provide safety arguments for an ML-based HAD function. These methods consider various aspects of the ML algorithms—from the datasets used to train them to the features extracted by different architectures.

Chapter 4, we argue that comparing the classification performance of different DNN architectures is insufficient in safety-critical applications. Therefore, we develop a safety argument using inner and post-hoc interpretability methods (*cf.* [25; 27]). We propose comparing the architectures based on how well the feature embeddings cluster and how meaningful they are. Expanding on this validation safety argument, we use an algorithm to create k-means friendly spaces to improve the clustering of the feature embeddings. We additionally propose a method to reject spurious classification outputs.

On the other hand, the datasets used to train the ML-based HAD functions are also important to argue for safety. In Chapter 5, we investigate the distributions of various highway driving datasets. We observe a shift in their distributions and document how

this affects ML-based classifiers trained on one dataset and tested on another. This problem can arise when an ML-based HAD function is trained on a dataset from one country, but deployed in another country. To address the arising issues, we propose a method to fine-tune an ML algorithm and trade-off the forgetting and the learning between the two distributions. Finally, in Chapter 6, we consider the design of the overall ML-based HAD function. We propose an intrinsically interpretable Lane Change Detector (LCD) algorithm using ML-based anomaly detection algorithms. We demonstrate how an engineer can choose the algorithm's parameters in an interpretable manner.

On the one hand, the methods presented in this dissertation can be used as evidence for safety, e.g., in an overarching safety case (*cf.* Section 2.4.1). On the other hand, they can be directly used by an engineer, e.g., to decide which DNN architecture is most suitable for the given HAD function using the available data or to investigate the dataset distributions. Since the proposed methods consider various aspects of ML algorithms, we call the overall safety philosophy: *Validation by Diversity*.

3 Machine Learning Preliminaries

In this chapter, we introduce the Machine Learning (ML) fundamentals used throughout this dissertation. Since this topic has been thoroughly studied in various scientific papers and textbooks, see, e.g., [68; 69; 70], we focus on the parts of ML algorithms we investigate in this work. First, we briefly motivate the use of ML algorithms as data-driven functions in Section 3.1; Section 3.2 describes the time-series data we consider. Then, Section 3.3 explores the various Deep Neural Network (DNN) architectures employed in multi-variate time-series classification tasks. We subsequently discuss the procedure of optimising the tunable parameters in these ML algorithms in Section 3.4. Finally, Section 3.5 concludes this chapter by introducing the Highly Automated Driving (HAD) function and the datasets used in this dissertation.

3.1 Motivation

The rise in popularity of ML[1] algorithms in recent years has come hand-in-hand with the feedforward DNN, see, e.g., [68]. These DNNs take an input and transform it using multiple non-linear transformations to an output. This output is subsequently used for the task at hand, e.g., classification or regression. For classification problems, such a DNN aims to learn the mapping

$$f \ : \mathbb{X} \to \mathbb{Y}, \boldsymbol{x} \mapsto f(\boldsymbol{x}; \mathcal{P}) \tag{3.1}$$

with the DNN's tunable parameters collected in $\mathcal{P}$, an arbitrary input $\boldsymbol{x} \in \mathbb{X}$ from the domain $\mathbb{X}$, and the set of class labels $\mathbb{Y} = \{y_1, y_2, \ldots, y_K\}$ (assuming K classes in total). In the case of ML-based algorithms, the underlying statistical model is unknown, and hence, the derivation of the DNN model's parameters relies on a given dataset of M input/output pairs

$$\mathcal{D} = \left\{(\boldsymbol{x}^1, y^1), (\boldsymbol{x}^2, y^2), \ldots, (\boldsymbol{x}^M, y^M)\right\} \subset \mathbb{X} \times \mathbb{Y}. \tag{3.2}$$

[1]Note, throughout this dissertation we use the term ML to refer to both classical ML methods, e.g., Decision Trees (DTs) or k-Nearest Neighbours (k-NN) classifiers, and modern deep learning methods. A description of the history of these fields can be found in, e.g., [68, Ch. 1].

This dataset can be further split into a *training*, a *validation*, and a *test* dataset.

The reliance on datasets is the reason why ML algorithms are referred to as data-driven algorithms compared to model-driven algorithms. In the latter, a model of the given application is derived from physical rules. In contrast, the model in a data-driven algorithm is derived directly from the data. ML methods' ability to learn representations from a limited number of (training) examples alleviates the burden on engineers to explicitly model every scenario or environment the function might encounter. However, this comes with the trade-off that the engineer will have to put extra effort into validating the data-driven model using, e.g., expert knowledge.

Engineers implement ML-based methods in ever more parts of the Autonomous Vehicle (AV) pipeline. Many computer vision tasks in HAD functions are already solved using ML-based methods, some of these tasks include: semantic segmentation, where each pixel of an image is classified to a predefined category [71; 72]; detecting objects in an image, where a bounding box is created around each object [73; 74]; or detecting objects from 3-D point cloud data [75]. ML-based algorithms are also implemented in various other HAD functions, see, e.g., [5] for an overview.

The key ingredients of any ML method are:

1. The choice of function, i.e., the DNN architecture;

2. The choice of learning framework, i.e., the loss function, the optimiser, and/or the (hyper)-parameters;

3. The dataset from which the function's parameters should be derived.

In the rest of this chapter, we introduce the relevant ingredients for the ML algorithms used throughout the dissertation.

3.2 Time-Series Data

We use multi-variate time-series data as input to the ML-based functions. First, we formally introduce some definitions for ML based on time-series signals.

Definition 1. *A univariate time-series signal is an ordered vector of N real values,*

$$\boldsymbol{x}^T = \begin{bmatrix} x[1] & x[2] & \dots & x[N] \end{bmatrix} \in \mathbb{R}^{1\times N}. \tag{3.3}$$

Definition 2. *A multi-variate time-series signal is collection of* Γ *univariate time-series signals,*

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x}_1^T \\ \vdots \\ \boldsymbol{x}_\Gamma^T \end{bmatrix} \in \mathbb{R}^{\Gamma \times N}. \tag{3.4}$$

A multi-variate time-series signal can also be viewed per time-step by summarising the values of the different univariate time-series signals at that time-stamp in a vector, i.e.,

$$\boldsymbol{X} = \begin{bmatrix} \bar{\boldsymbol{x}}[1] & \dots & \bar{\boldsymbol{x}}[N] \end{bmatrix} \in \mathbb{R}^{\Gamma \times N}, \tag{3.5}$$

where $\bar{\boldsymbol{x}}[n] = [x_1[n], \dots, x_\Gamma[n]]^\mathrm{T} \in \mathbb{R}^\Gamma$ at each time-stamp $n = 1, \dots, N$.

We are also interested in short sequences within the time-series signals.

Definition 3. *Given a time-series signal* $\boldsymbol{x}^T \in \mathbb{R}^{1 \times N}$*, a sub-sequence* $\boldsymbol{x}_p^T$ *is a sampling of length* $W < N$ *of contiguous time-stamps from* $\boldsymbol{x}^T$*, i.e.,*

$$\boldsymbol{x}_p^T = \begin{bmatrix} x[p] & x[p+1] & \dots & x[p+W-1] \end{bmatrix} \in \mathbb{R}^{1 \times W}, \tag{3.6}$$

for $1 \leq p \leq N - W + 1$.

Finally, we introduce the collection of all possible sub-sequences generated by passing a sliding window over the time-series signal with a constant window size W [76].

Definition 4. *Given a time-series signal* $\boldsymbol{x}^T \in \mathbb{R}^{1 \times N}$*, the collection of sub-sequences (sliding windows)* $\boldsymbol{x}_p^T$ *for* $p = 1, \dots, N - W + 1$ *of a constant window size* $W < N$ *can be generated by the following*

$$\begin{aligned} \mathrm{win}(\boldsymbol{x}; W) = \Big\{ \begin{bmatrix} x[p] & x[p+1] & \dots & x[p+W-1] \end{bmatrix} \in \mathbb{R}^{1 \times W} : \\ \textit{for } p = 1, \dots, N - W + 1 \Big\} \end{aligned} \tag{3.7}$$

Definitions 3 and 4 can be extended to extract sub-sequences and sliding windows of multi-variate time-series signals using the definition of multi-variate time-series signals from Definition 2.

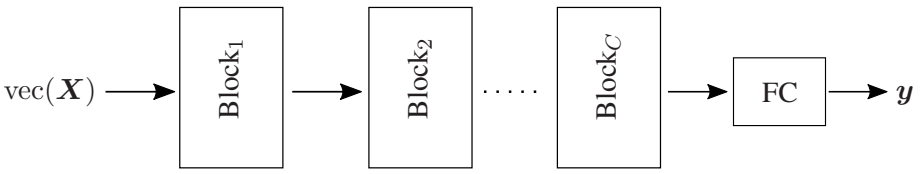


Figure 3.1: Abstract FC-DNN architecture.

3.3 Deep Neural Network Architectures

Over the years, various DNN architectures have been introduced for different applications. Each architecture creates feature embeddings in a different manner which we will investigate in later chapters. In this section, we introduce the considered DNN architectures in detail.

The general structure of a DNN is depicted in Figure 3.1. The input signal $\boldsymbol{X}$ is processed by C blocks before being linearly transformed to the output via Fully Connected (FC) linear layer. Each processing block can either be a single linear transformation followed by a non-linear activation function or it can be multiple mappings combined in a unique way (*cf.* Transformer-based methods in Section 3.3.4). We assume there are Λ layers in total—including the layer(s) in the C processing blocks and the final FC layer. In the following, we discuss the different possible mapping functions investigated throughout this work.

3.3.1 Fully Connected Deep Neural Networks

We first introduce the FC DNN. In this architecture, the non-linear functions in each layer i of the DNN—where $i = 1, \ldots, \Lambda$—can be expressed as

$$\boldsymbol{x}^{(i)} = g\Big(\underbrace{\boldsymbol{W}^{(i)}\boldsymbol{x}^{(i-1)} + \boldsymbol{b}^{(i)}}_{\boldsymbol{z}^{(i)}} \Big). \tag{3.8}$$

The learnable parameters of the DNN are collected in

$$\mathcal{P} = \{\boldsymbol{W}^{(i)} \in \mathbb{R}^{N_{(i)} \times N_{(i-1)}}, \boldsymbol{b}^{(i)} \in \mathbb{R}^{N_{(i)}} : 1 \leq i \leq \Lambda\}. \tag{3.9}$$

To this end, we consider a processing block (*cf.* Figure 3.1) in a FC-DNN to perform the mapping defined in (3.8). This implies $C = \Lambda - 1$. The signal $\boldsymbol{z}^{(i)}$ is the representation in layer i before the non-linear activation function. The non-linear activation function g is applied element-wise. Throughout this dissertation, if not otherwise stated, we use

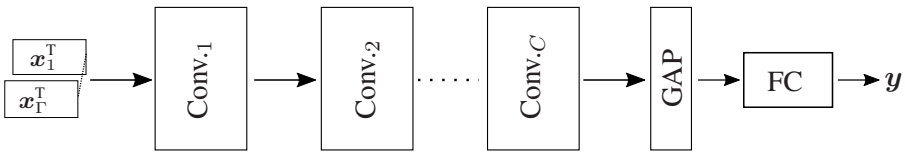


Figure 3.2: Abstract Convolutional Neural Network (CNN) architecture for multi-variate time-series data considered in this dissertation.

the hyperbolic tangent function as the non-linear activation function, i.e., $g = \tanh$. The output activation function depends on the application at hand.[2]

3.3.2 Convolutional Neural Networks

The general CNN architecture we consider is depicted in Figure 3.2. Since we consider multi-variate time-series data, we interpret the input as a multi-channel input—each input attribute of the time-series signal is therefore considered a channel. This is passed through C consecutive convolutional blocks depicted as Conv.$_1, \ldots,$ Conv.$_C$. These blocks are built of different convolutional layers, which will be discussed in the following sub-sections. Note, we refer to the input time-series samples as having multiple input channels, but after Conv.$_1$, we refer to the filtered signals as feature maps. After the convolutional blocks we employ a Global Average Pooling (GAP) layer, which takes the average of each filtered signal per feature map (see Section 3.3.2.6).

In this work, we consider convolutional blocks built up of two components:[3] first, a convolutional layer is applied to generate feature maps, and second, the feature maps are passed through a non-linear activation function. The feature maps are time-series signals, i.e., vectors, after the convolutional layer. For high-dimensional input data, one can additionally incorporate pooling layers, e.g., average pooling or max pooling, in the convolutional blocks to reduce the dimensions. The non-linear functions can be described as

$$\boldsymbol{x}_\gamma = g\left(\boldsymbol{z}_\gamma\right), \tag{3.10}$$

where both $\boldsymbol{x}_\gamma \in \mathbb{R}^N$ and $\boldsymbol{z}_\gamma \in \mathbb{R}^N$ represent time-series signals. The index $\gamma \in \{1, 2, \ldots, \Gamma^{(i)}\}$ represents the feature map γ; $\Gamma^{(0)} = \Gamma$ describe the input

[2]Note, for multi-class classification tasks, the softmax activation function is used at the output, which is not applied element-wise.

[3]For simplicity, we further assume the following: (i) no padding is added to the time-series signals, i.e., extra zeros to keep the signal length the same before and after convolution; (ii) a stride of one; (iii) and no time dilation.

channels. For the sake of simplicity, we will drop the layer index i in the feature maps wherever possible in the following. Note, the feature maps $\boldsymbol{z}_\gamma$ in layer i are a function of the signals $\boldsymbol{x}_\gamma$ in the previous layer $i-1$.

We can stack the feature maps in a layer, such that we can write the output using the 1-D CNNs as

$$\boldsymbol{z} = \begin{bmatrix} \boldsymbol{z}_1 \\ \vdots \\ \boldsymbol{z}_\gamma \\ \vdots \\ \boldsymbol{z}_{\Gamma^{(i)}} \end{bmatrix} = \boldsymbol{W} \begin{bmatrix} \boldsymbol{x}_1 \\ \vdots \\ \boldsymbol{x}_\gamma \\ \vdots \\ \boldsymbol{x}_{\Gamma^{(i-1)}} \end{bmatrix} + \boldsymbol{b}, \tag{3.11}$$

where $\boldsymbol{z} \in \mathbb{R}^{\Gamma^{(i)}(N-L+1)}$, $\boldsymbol{x}_\gamma \in \mathbb{R}^N$, and $\boldsymbol{b} \in \mathbb{R}^{\Gamma^{(i)}(N-L+1)}$ is a vector of bias terms. The total weight matrix $\boldsymbol{W} \in \mathbb{R}^{\Gamma^{(i)}(N-L+1)\times\Gamma^{(i-1)}N}$ has a different structure depending on the type of 1-D CNN architecture used. We will introduce the different structures in the following sub-sections.

We define an operator to create a matrix with a Toeplitz structure of the vector $\boldsymbol{w} = \begin{bmatrix} w[1] & w[2] & \dots & w[L] \end{bmatrix}^{\mathrm{T}} \in \mathbb{R}^L$ as

$$\operatorname{toep}(\boldsymbol{w}; N) = \begin{bmatrix} \boldsymbol{L}_0\boldsymbol{w}, & \boldsymbol{L}_1\boldsymbol{w}, & \dots, & \boldsymbol{L}_{N-L}\boldsymbol{w} \end{bmatrix}^{\mathrm{T}} \tag{3.12a}$$

$$= \begin{bmatrix} w[1] & w[2] & \dots & w[L] & 0 & \dots & 0 \\ 0 & w[1] & w[2] & \dots & w[L] & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & w[1] & w[2] & \dots & w[L] \end{bmatrix} \in \mathbb{R}^{N-L+1\times N}, \tag{3.12b}$$

with the selection matrix defined as

$$\boldsymbol{L}_l = \begin{bmatrix} \boldsymbol{0}_{L\times l}, & \boldsymbol{I}_{L\times L}, & \boldsymbol{0}_{L\times(N-L-l)} \end{bmatrix}^{\mathrm{T}} \in \mathbb{R}^{N\times L}, \tag{3.13}$$

where the parameter N is the length of the signal which the filter will be convolved with, and $l = 0, \dots, N-L$.

With this Toepliz matrix, we can define the discrete-time convolution as

$$\boldsymbol{z} = \boldsymbol{w} * \boldsymbol{x} = \operatorname{toep}(\boldsymbol{w}; N)\, \boldsymbol{x} \tag{3.14}$$

where $*$ represents the discrete-time convolution between the signal $\boldsymbol{x} \in \mathbb{R}^N$ and the vector (1-D filter) $\boldsymbol{w} \in \mathbb{R}^L$.

3.3.2.1 Standard Convolutional Layers

The first type of 1-D convolutional layer we consider is the standard convolution. It was originally introduced for 2-D CNNs in [77] to work with image data. These 1-D filters are applied to each feature map of the previous layer and summed up to generate a feature map in the current layer. This can be interpreted as learning a Finite Impulse Response (FIR) filter for each input feature map and summing over the filtered signals.

The total weight matrix (see (3.11)) for the standard convolution layer has the structure

$$\boldsymbol{W}_{\text{conv.}} = \begin{bmatrix} \text{toep}(\boldsymbol{w}_{1,1}; N) & \dots & \text{toep}(\boldsymbol{w}_{1,\Gamma^{(i-1)}}; N) \\ \vdots & \ddots & \vdots \\ \text{toep}(\boldsymbol{w}_{\Gamma^{(i)},1}; N) & \dots & \text{toep}(\boldsymbol{w}_{\Gamma^{(i)},\Gamma^{(i-1)}}; N) \end{bmatrix}, \tag{3.15}$$

with $\boldsymbol{w}_{\gamma,c} \in \mathbb{R}^L$ for $\gamma = 1, \dots, \Gamma^{(i)}$ and $c = 1, \dots, \Gamma^{(i-1)}$. We use the Toeplitz operator from (3.12a). Since each row of (3.15) represents a feature-map at the output ($\boldsymbol{z}_\gamma$), we see that we sum over all of the input signals from the previous layer.

As the input signals we consider are multi-variate time-series data, the filters in the first convolutional block (Conv.$_1$) can be interpreted as FIR filters on the raw input data. These not only extract temporal correlations, but also cross-channel correlations within the input signals. However, we should note that each feature map $\boldsymbol{z}^{(1)}$ in the first layer (and in the subsequent layers) is a summation of the filtered input signals. Thus, it is difficult to give a physical interpretation of the feature maps when employing a standard 1-D convolution. For example, in the first layer, filtered acceleration signals will be summed together with filtered distance signals, losing their physical interpretation. We return to this observation in Section 3.3.2.5.

3.3.2.2 Depth-wise Separable Convolutional Layers

The second type of convolutional layer we consider is a depth-wise separable convolution, popularised in recent years for image classification, see, e.g., [78; 79]. The main idea behind the 1-D depth-wise separable convolution is to split the convolutional task into two parts: first, the temporal correlations per feature map are extracted; and second, the cross-channel correlations are extracted using a 1×1 convolution operation.[4] By splitting the convolutional operation into two stages, the number of required parameters is reduced with similar performance (*cf.* [82]).

[4] A 1×1 convolution operation is a multiplication of each input channel with a scalar, allowing us to reduce the number of output channels. This is sometimes used before a 2-D convolutional operation to reduce the overall number of parameters of the network without reducing the performance [80; 81].

We can write the total weight matrix (see (3.11)) for the depth-wise separable convolution layer as

$$\boldsymbol{W}_{\text{sep.}} = \begin{bmatrix} w_{1,1}\text{toep}(\boldsymbol{w}_1; N) & \dots & w_{1,\Gamma^{(i-1)}}\text{toep}(\boldsymbol{w}_{\Gamma^{(i-1)}}; N) \\ \vdots & \ddots & \vdots \\ w_{\Gamma^{(i)},1}\text{toep}(\boldsymbol{w}_1; N) & \dots & w_{\Gamma^{(i)},\Gamma^{(i-1)}}\text{toep}(\boldsymbol{w}_{\Gamma^{(i-1)}}; N) \end{bmatrix}, \tag{3.16}$$

with $w_{\gamma,c} \in \mathbb{R}$ and $\boldsymbol{w}_c \in \mathbb{R}^L$ for $\gamma = 1, \dots, \Gamma^{(i)}$ and $c = 1, \dots, \Gamma^{(i-1)}$. We use the Toeplitz operator from (3.12a). We see the two parameters of the convolutional operation in (3.16): a FIR filter $\boldsymbol{w}_c \in \mathbb{R}^L$ is learned for each input signal from the previous layer, and then these are weighted using the scalars $w_{\gamma,c} \in \mathbb{R}$.

When applying the depth-wise separable convolution to the multi-variate time-series data, we note that for each convolutional block in the CNN, only one FIR filter $\boldsymbol{w}_c$ is learned per feature map. Therefore, only the most discriminative temporal correlations can be extracted from the input feature map, however, due to the multiplication with the weights $w_{\gamma,c}$, the influence of each input feature map for each output feature map can be adjusted. If we look at the signals in the first layer of the CNN, we see that this means, the influence of the cross-channel correlation between, e.g., a filtered acceleration signal and a filtered distance signal, can be adjusted using these weights. Again, similar to the standard 1-D convolution, a physical interpretation of the filtered signals is lost after the first layer, due to the mixing of input feature maps with different physical units.

3.3.2.3 Local Convolutional Layers

The third type of 1-D convolutional layer we consider is the locally connected 1-D convolutional layer. It is similar to the standard 1-D convolution; however, the weights are no longer shared over the whole input signal, i.e., a new set of L weight parameters is learned for each part of each feature map of the input sequence.

We can write the total weight matrix (see (3.11)) for the local convolutional layers as

$$\boldsymbol{W}_{\text{local}} = \begin{bmatrix} \text{toep}\left(\boldsymbol{w}_{1,1,0}, \dots, \boldsymbol{w}_{1,1,N-L}; N\right) & \dots \\ \vdots & \ddots \\ \text{toep}\left(\boldsymbol{w}_{\Gamma^{(i)},1,0}, \dots, \boldsymbol{w}_{\Gamma^{(i)},1,N-L}; N\right) & \dots \end{bmatrix}, \tag{3.17}$$

with $\boldsymbol{w}_{\gamma,c,k} \in \mathbb{R}^L$ for $\gamma = 1, \dots, \Gamma^{(i)}$, $c = 1, \dots, \Gamma^{(i-1)}$, and $k = 1, \dots, N - L$. We slightly abuse the notation of the toep operator where each filter vector $\boldsymbol{w}_{\gamma,c,k}$ is

multiplied with the selection matrix $\boldsymbol{L}_k$, as defined in (3.12a). Note, we assume the input signals $\boldsymbol{x}_\gamma$ are always the same length, i.e., N.

By relaxing the weight sharing condition in the standard convolutional layer, we allow the CNN to learn discriminative temporal correlations at each possible sequence of the input signals. This comes at the price of more parameters per feature map, since the weights at each sequence of the input signal are learned independently. Therefore, discriminative reoccurring temporal correlations might not be learned by the network.

3.3.2.4 Set of Learnable Weights in CNNs

In this sub-section, we discuss the set of learnable weights which the previously introduced 1-D CNN architectures are able to learn. Due to the Toeplitz structure of the 1-D convolutional layers, the set of possible learnable weights is different, depending on which 1-D convolutional layer is used.

To this end, we assume the linear transformation between layers i and $i-1$ have $\Gamma^{(i)}$ and $\Gamma^{(i-1)}$ feature maps, respectively. Without loss of generality, we assume there is no bias vector.

Proposition 1. *The sets of learnable weights between two layers of a CNN with $\Gamma^{(i)}$ and $\Gamma^{(i-1)}$ feature maps, using different 1-D convolutional layers are related as follows*

$$\mathbb{W}_{sep.} \subseteq \mathbb{W}_{conv.} \subseteq \mathbb{W}_{local}, \tag{3.18}$$

where the set of possible weights are defined as

$$\mathbb{W}_{conv.} = \{\boldsymbol{W} \in \mathbb{R}^{\Gamma^{(i)}(N-L+1)\times\Gamma^{(i-1)}N} : \boldsymbol{W} \textit{ has the form } (3.15)\}, \tag{3.19a}$$

$$\mathbb{W}_{sep.} = \{\boldsymbol{W} \in \mathbb{R}^{\Gamma^{(i)}(N-L+1)\times\Gamma^{(i-1)}N} : \boldsymbol{W} \textit{ has the form } (3.16)\}, \tag{3.19b}$$

$$\mathbb{W}_{local} = \{\boldsymbol{W} \in \mathbb{R}^{\Gamma^{(i)}(N-L+1)\times\Gamma^{(i-1)}N} : \boldsymbol{W} \textit{ has the form } (3.17)\}. \tag{3.19c}$$

Proof. First, we show that $\mathbb{W}_{\text{sep.}} \subseteq \mathbb{W}_{\text{conv.}}$ and $\mathbb{W}_{\text{sep.}} \subseteq \mathbb{W}_{\text{local}}$. Suppose $\boldsymbol{W}_1 \in \mathbb{W}_{\text{sep.}}$. Comparing the structure in (3.16) to (3.15), we observe that

$$\boldsymbol{W}_1 \in \mathbb{W}_{\text{conv.}} \;\Rightarrow\; \mathbb{W}_{\text{sep.}} \subseteq \mathbb{W}_{\text{conv.}}, \tag{3.20}$$

with $\boldsymbol{w}_{a,b} = w_{a,b}\boldsymbol{w}_a$ for $a = 1, \ldots, \Gamma^{(i)}$ and $b = 1, \ldots, \Gamma^{(i-1)}$. Moreover, comparing (3.16) to (3.17), we observe that

$$\boldsymbol{W}_1 \in \mathbb{W}_{\text{local}} \;\Rightarrow\; \mathbb{W}_{\text{sep.}} \subseteq \mathbb{W}_{\text{local}}, \tag{3.21}$$

with $\boldsymbol{w}_{a,b,k} = w_{a,b}\boldsymbol{w}_a$ for $a = 1, \ldots, \Gamma^{(i)}$, $b = 1, \ldots, \Gamma^{(i-1)}$, and $k = 1, \ldots, N-L$.

Next, we show that $\mathbb{W}_{\text{conv.}} \nsubseteq \mathbb{W}_{\text{sep.}}$ and $\mathbb{W}_{\text{conv.}} \subseteq \mathbb{W}_{\text{local}}$. Suppose $\boldsymbol{W}_2 \in \mathbb{W}_{\text{conv.}}$ where the relationship between two filters is $\boldsymbol{w}_{a,b} \neq \alpha \boldsymbol{w}_{c,b}$ for some index $a \neq c$ and a scalar $\alpha \in \mathbb{R}$. Therefore, comparing (3.15) and (3.16), we see that

$$\boldsymbol{W}_2 \notin \mathbb{W}_{\text{sep.}} \;\Rightarrow\; \mathbb{W}_{\text{conv.}} \nsubseteq \mathbb{W}_{\text{sep.}}. \tag{3.22}$$

On the other hand, comparing (3.15) and (3.17), we see that

$$\boldsymbol{W}_2 \in \mathbb{W}_{\text{local}} \;\Rightarrow\; \mathbb{W}_{\text{conv.}} \subseteq \mathbb{W}_{\text{local}}, \tag{3.23}$$

with $\boldsymbol{w}_{a,b,k} = \boldsymbol{w}_{a,b}$ for $a = 1, \ldots, \Gamma^{(i)}$, $b = 1, \ldots, \Gamma^{(i-1)}$, and $k = 1, \ldots, N - L$.

Finally, we show that $\mathbb{W}_{\text{local}} \nsubseteq \mathbb{W}_{\text{conv.}}$ and $\mathbb{W}_{\text{local}} \nsubseteq \mathbb{W}_{\text{sep.}}$. Suppose $\boldsymbol{W}_3 \in \mathbb{W}_{\text{local}}$ where the relationship between two filters is $\boldsymbol{w}_{a,b,k} \neq \boldsymbol{w}_{a,b,m}$ for some index $k \neq m$. Thus, by inspection, we see that

$$\boldsymbol{W}_3 \notin \mathbb{W}_{\text{conv.}} \;\Rightarrow\; \mathbb{W}_{\text{local}} \nsubseteq \mathbb{W}_{\text{conv.}}, \tag{3.24}$$

and

$$\boldsymbol{W}_3 \notin \mathbb{W}_{\text{sep.}} \;\Rightarrow\; \mathbb{W}_{\text{local}} \nsubseteq \mathbb{W}_{\text{sep.}}. \tag{3.25}$$

□

3.3.2.5 Multi-Channel Convolutional Neural Networks

Up to this point, we consider CNN architectures where all input channels—or input signals—are processed through the same CNN blocks. However, this leads to the summation of (filtered) signals with different physical units, which may be undesirable. Therefore, we consider further CNN architectures where each input channel is filtered individually; similar to the CNN design in [83]. In this design, instead of processing the multi-variate input signal as one signal, i.e., processing all of the input channels at once, each input channel is filtered individually. This architecture is visualised in Figure 3.3. Each of the input channels $1, \ldots, \Gamma$ is separately processed by convolutional blocks until they are concatenated before the output.

As discussed in the previous sub-sections, the multi-variate time-series signals we consider have different physical units, e.g., m for distances or m/s for velocities. By filtering these signals and summing them together, they lose their physical interpretation after the first layer of 1-D CNN filters. The architecture depicted in Fig. 3.3 can be interpreted as a trade-off between interpretability and feature extraction: on the one hand, we raise the overall interpretability of the CNN by using an Multi-Channel

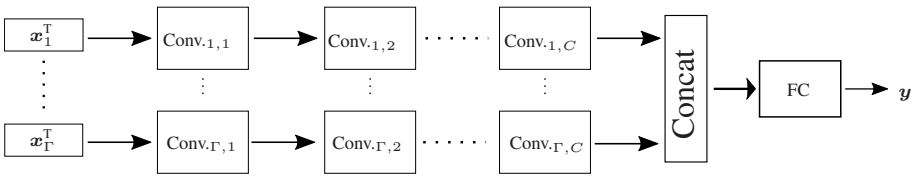


Figure 3.3: Abstract MC-CNN architecture for multi-variate time-series data, where each channel is separately transformed.

(MC) architecture since we can investigate each channel separately and the physical interpretation of the signals is maintained throughout the network. On the other hand, by separating the input channels, only temporal correlations can be extracted by the CNN, i.e., the cross-channel correlations are only taken into consideration by any FC linear layers at the output of the CNN.

A result of using an MC architecture is that in the first layer, each convolutional block creates one feature map for each filter, i.e.,

$$\boldsymbol{z}_c^{(1)} = \boldsymbol{W}_{c,\cdot}\boldsymbol{x}^{(0)} + \boldsymbol{b}_c \; \forall c \in \{1, \ldots, \Gamma^{(0)}\}, \tag{3.26}$$

where $\boldsymbol{W}_{c,\cdot}$ is the c-th column of the total weight matrix which can have the form (3.15), (3.16), or (3.17). We consider three MC-CNN architectures which use the various convolutional layers defined in Sub-Sections 3.3.2.1, 3.3.2.2, and 3.3.2.3. Proposition 1 still holds in the case of an MC-CNN architecture.

3.3.2.6 Global Average Pooling

The GAP layer was first proposed in [84] to prevent overfitting in the final, fully connected layers of a CNN. Moreover, the authors also argue that it introduces a robustness against spatial translation of input images. In this case, the robustness would be against the specific temporal location of the signal of interest.

In the GAP layer, the signal of each feature map after the Cth convolutional block is averaged over the remaining time dimension, i.e., each filter map is replaced by its average signal. This can be expressed as

$$z_\gamma^{(\Lambda-1)} = \frac{1}{N_{(\Lambda-1)}} \sum_{n=1}^{N_{(\Lambda-1)}} x_\gamma^{(\Lambda-1)}[n] \in \mathbb{R}. \tag{3.27}$$

Therefore, an average signal is taken for each feature map $\boldsymbol{x}_\gamma^{(\Lambda-1)}$ in the penultimate layer where $\gamma \in \{1, \ldots, \Gamma^{(\Lambda-1)}\}$. Note, we can also consider the GAP layer as part of the final convolutional block.

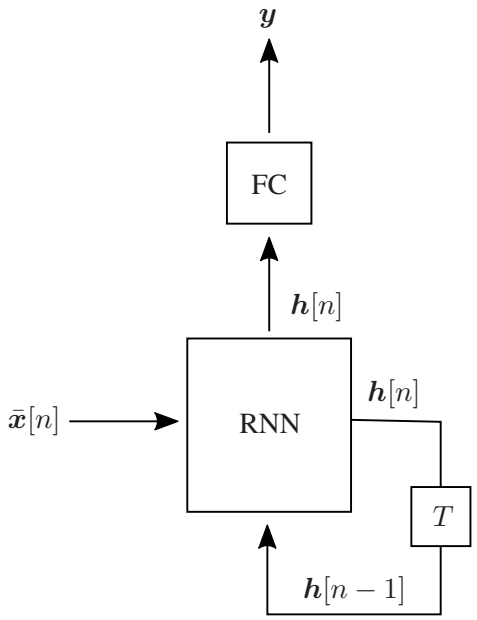


Figure 3.4: Abstract RNN architecture for multi-variate time-series data.

3.3.3 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a popular architecture choice when working with time-series or sequential data, see, e.g., [68, Ch. 10]. As the name suggests, RNNs allow for information to be passed recurrently through time which helps the network learn time dependences.

The general architecture is depicted in Figure 3.4. We see that at each time-stamp, the inputs $\bar{\boldsymbol{x}}[n]$ are passed to the RNN block, which outputs a hidden state representation $\boldsymbol{h}[n]$. The block T represents a one time-stamp delay of the hidden state representation $\boldsymbol{h}[n]$ which is subsequently fed back into the RNN block. The hidden state representation $\boldsymbol{h}[n]$ is also transformed by a linear layer to the output labels $\boldsymbol{y}$.[5] Ultimately, the RNN block computes the following:

$$\boldsymbol{z}[n] = \boldsymbol{W_h}\boldsymbol{h}[n-1] + \boldsymbol{W_x}\boldsymbol{x}[n] + \boldsymbol{b}, \tag{3.28}$$

with the linear transformation matrices $\boldsymbol{W_h}$ and $\boldsymbol{W_x}$ used to transform the hidden state representations from the previous time-stamp and the inputs from the current

[5]Note, the output labels can also have a time-stamp dependency, i.e., $y[n]$. However, as we only consider one label for each time-series input, we omit this notation for consistency.

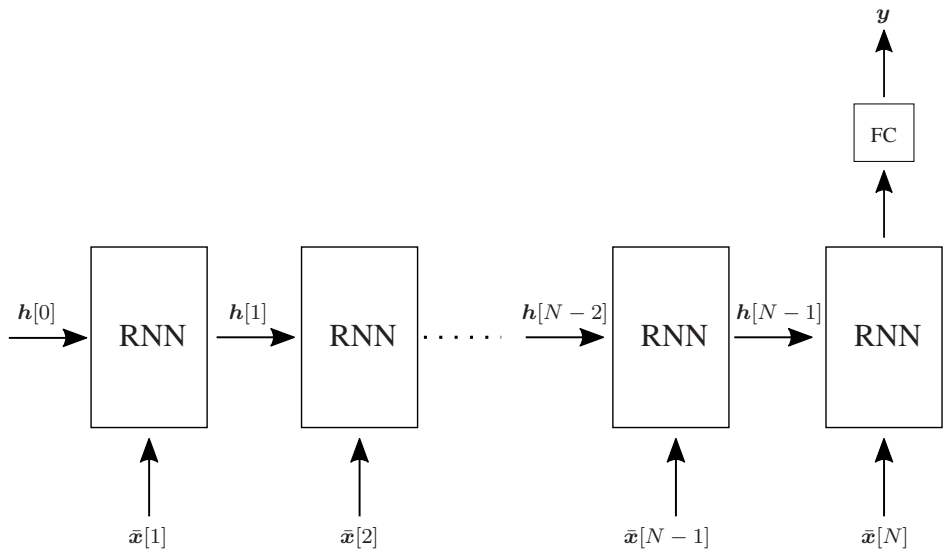


Figure 3.5: The RNN architecture unrolled over time for multi-variate time-series data.

time-stamp, respectively. The vector $\boldsymbol{b}$ is the bias term. The hidden state representation of the current time-stamp is subsequently calculated as

$$\boldsymbol{h}[n] = g(\boldsymbol{z}[n]), \tag{3.29}$$

with a non-linear activation function g applied element-wise. The output labels are a linear transformation of the hidden state representations

$$\boldsymbol{y} = \mathrm{softmax}\left(\boldsymbol{W}^{(\Lambda)}\boldsymbol{h}[n] + \boldsymbol{b}^{(\Lambda)}\right) \in (0,1)^{|\mathbb{Y}|}, \tag{3.30}$$

with the softmax activation function (*cf.* (3.41)).

Another possible representation of a RNN is depicted in Figure 3.5. Here, we see the RNN architecture unrolled over time, i.e., we assume an input time-series datum of N time-stamps and we see that the RNN block processes each input and hidden state representation and passes the hidden state representation on to the next time-stamp. After the Nth time-stamp of the input is passed to the RNN block, the output label $\boldsymbol{y}$ is calculated as described in (3.30). This unrolled representation of an RNN is useful for the methods presented in Chapter 4 and Chapter 5.

3.3.3.1 Gated Recurrent Neural Networks

Due to the special recurrent structure of RNNs, they cannot be trained using the standard backpropagation algorithm (see Section 3.4). Instead, they need to be trained

using special algorithms, e.g., backpropagation through time [85]. These algorithms need to take into consideration that the loss at a certain time-stamp not only depends on the hidden state representation at the current time-stamp, but it also influences the hidden state representation at the next time-stamp. This can lead to the hidden state representations either *exploding* or *vanishing*—this problem when training RNNs is referred to as the *exploding or vanishing gradient problem* [86; 87].

To overcome the vanishing gradient problem, and to help learn long-term dependences in the time-series data, researchers introduced gating mechanisms. This is achieved by scaling the hidden state representations in the RNN block using gates. These gates are generated by applying the sigmoid activation function to a weighted sum of hidden state representations and inputs. The sigmoid activation allows for a data-dependent scaling of the current hidden state representations. These additional hidden state representations improve the gradient flow, and thereby help mitigate the *exploding/vanishing gradient problem*.

In recent years, two RNN architectures have become popular: the Long Term Short Term Memory (LSTM) block [11] and the Gated Recurrent Unit (GRU) block [88; 13]. These RNN blocks have the same input and hidden state representations as a standard RNN block (*cf.* Figure 3.4 or Figure 3.5), however, they further introduce a state unit representation $\boldsymbol{s}[n]$ which is contained within the RNN block and has a self-loop similar to the hidden state representations. This state unit representation can control the long-term dependencies the RNN can learn. The LSTM block includes three gates: (i) an input gate to scale the inputs; (ii) an output gate to scale the outputs; and (iii) a forget gate to scale the state unit representation inside the RNN block. The GRU block simplifies this structure and uses a single gate to control the forgetting factor and the state unit representation. For more details about these RNN architectures, the interested reader can look into the original publications or refer to [68, Ch. 10.10], [89, Ch. 4].

3.3.4 Transformer Networks

In recent years, transformer-based methods, see, e.g., [90], have surpassed RNN and other feed-forward DNN architectures in popularity. They show state-of-the-art performance in many ML applications, e.g., natural language processing [90; 91; 92], image generation [93], image classification [94], or time-series classification [95]. They combine the efficient parallelisation benefits of training feed-forward DNNs with the ability to learn the importance between different parts of the input signal, i.e., they can learn long-term dependencies in the data similar to the hidden state representations of

an RNN without the challenge of the *exploding/vanishing gradient problem*. Moreover, they scale better to larger network sizes than RNNs, see, e.g., [96; 90, Tab. 1].

3.3.4.1 (Self-)Attention Mechanism

The attention mechanism is at the core of the transformer-based methods [90]. Fundamentally, it is the process of allowing the transformer-based model to learn which parts of the input signal it should "attend" to for the current task. It is inspired by database retrieval problems where one has a database of keys and values; given a new input query, which key does it align best to? To this end, we define the query, the keys, and the values as vectors

$$\boldsymbol{q}, \boldsymbol{k}[n], \boldsymbol{v}[n] \in \mathbb{R}^d, \tag{3.31}$$

for $n = 1, \ldots, N$ key-value pairs of items in the database. Note, we can assume that the key-value pairs are time-stamps of a time-series signal. Next, we define a notion of similarity between the query and the keys in the database. Commonly, the (scaled) inner product is used

$$s[n] = \langle \boldsymbol{q}, \boldsymbol{k}[n] \rangle = \boldsymbol{q}^{\mathrm{T}} \boldsymbol{k}[n] \in \mathbb{R}, \ \forall n = 1, \ldots, N. \tag{3.32}$$

Once a similarity score (attention score) is calculated for every query key pair, we normalise the scores to get the normalised attention score

$$a[n] = \mathrm{softmax}(s[n]) = \frac{\exp(s[n])}{\sum_{m=1}^{N} \exp(s[m])}, \ \forall n = 1, \ldots, N. \tag{3.33}$$

Finally, the output is calculated as the weighted sum of the values

$$\boldsymbol{o} = \sum_{n=1}^{N} a[n] \boldsymbol{v}[n]. \tag{3.34}$$

If there was one key which was most similar, i.e., $a[n] \to 1$ for the key at time-stamp n, to the query, then the output in (3.34) would be the value corresponding to that key. In general, this is not the case, and the output is a weighted sum of the value vectors, where the weighting values indicate how much influence each key value has on the output.

To generate queries at each time-stamp, $\boldsymbol{q}[n]$ for $n = 1, \ldots, N$, we transform the time-series input at each time-stamp into a d-dimensional space. We can collect all

time-stamps into the matrices

$$\boldsymbol{Q} = \Big[\boldsymbol{q}[1], \quad \ldots, \quad \boldsymbol{q}[N]\Big] = \boldsymbol{W}_q \boldsymbol{X} \in \mathbb{R}^{d \times N}, \tag{3.35a}$$

$$\boldsymbol{K} = \Big[\boldsymbol{k}[1], \quad \ldots, \quad \boldsymbol{k}[N]\Big] = \boldsymbol{W}_k \boldsymbol{X} \in \mathbb{R}^{d \times N}, \tag{3.35b}$$

$$\boldsymbol{V} = \Big[\boldsymbol{v}[1], \quad \ldots, \quad \boldsymbol{v}[N]\Big] = \boldsymbol{W}_v \boldsymbol{X} \in \mathbb{R}^{d \times N}, \tag{3.35c}$$

with the multi-variate time-series input $\boldsymbol{X} \in \mathbb{R}^{\Gamma \times N}$ (*cf.* Definition 2). The matrices $\boldsymbol{W}_q \in \mathbb{R}^{d \times \Gamma}$, $\boldsymbol{W}_k \in \mathbb{R}^{d \times \Gamma}$, $\boldsymbol{W}_v \in \mathbb{R}^{d \times \Gamma}$ are tunable parameters of the transformer network. The same steps highlighted in (3.32), (3.33), and (3.34) are performed on these queries, keys, and values. Moreover, we introduce the notion of *self-attention* here, since the queries, keys, and values are all linear transformations of the input $\boldsymbol{X}$. This allows the attention mechanism to relate different parts of the same input signal.

Applying (3.32), (3.33), and (3.34) to the collected time-stamps, we arrive at the output of the (self)-attention mechanism

$$\boldsymbol{O} = \boldsymbol{V}\boldsymbol{A}^{\mathrm{T}} = \boldsymbol{V}\mathrm{softmax}(\boldsymbol{Q}^{\mathrm{T}}\boldsymbol{K})^{\mathrm{T}}, \tag{3.36}$$

with the attention matrix $\boldsymbol{A} \in \mathbb{R}^{N \times N}$, which summaries how similar each query ($\boldsymbol{Q}$) is to the keys ($\boldsymbol{K}$). Note, the inner product in (3.32) is now performed after the transformation by $\boldsymbol{W}_q^{\mathrm{T}}\boldsymbol{W}_k$ of the input signals. Therefore, the self-attention mechanism learns a linear transformation into a space where the similarities are most useful for the task at hand.

Next, we introduce a non-linearity such that a transformer-based model built of self-attention mechanisms can learn useful representations. Thus, the matrix $\boldsymbol{O}$ is transformed by means of a FC linear layer with a non-linear activation function. Moreover, if we consider the multi-variate time-series sample as introduced in (3.35), the self-attention pipeline is missing a notion of sequential order, see, e.g., [90]. To this end, we can add a positional encoding[6] to the query, key, and value embeddings, e.g., by adding a sinusoidal position representation

$$\boldsymbol{e}[n] = \begin{bmatrix} \sin(n/10000^{2/d}), \\ \cos(n/10000^{2/d}), \\ \vdots \\ \sin(n/10000), \\ \cos(n/10000), \end{bmatrix} \in \mathbb{R}^d \tag{3.37}$$

[6]Note, when stacking multiple self-attention blocks, the positional encoding is only added to the input signal to the first self-attention block.

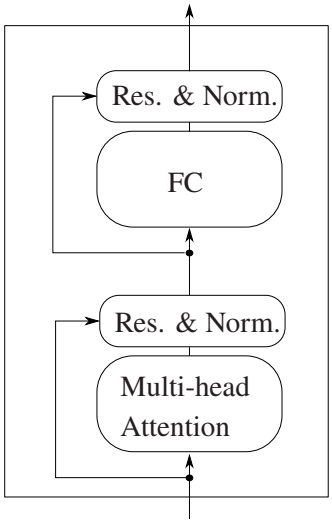


Figure 3.6: The inner workings of a multi-headed self-attention block.

where the new embeddings are

$$\tilde{\boldsymbol{q}}[n] = \boldsymbol{q}[n] + \boldsymbol{e}[n] \in \mathbb{R}^d, \tag{3.38a}$$

$$\tilde{\boldsymbol{k}}[n] = \boldsymbol{k}[n] + \boldsymbol{e}[n] \in \mathbb{R}^d, \tag{3.38b}$$

$$\tilde{\boldsymbol{v}}[n] = \boldsymbol{v}[n] + \boldsymbol{e}[n] \in \mathbb{R}^d. \tag{3.38c}$$

Furthermore, we can include multiple self-attention heads in one attention block, i.e., instead of learning one embedding matrix for the queries, keys, and values, we split this into multiple embeddings and apply the self-attention to each of the embeddings. Thus, we learn three sets of matrices $\{\boldsymbol{W}_{q,1}, \ldots, \boldsymbol{W}_{q,h}\}$, $\{\boldsymbol{W}_{k,1}, \ldots, \boldsymbol{W}_{k,h}\}$, and $\{\boldsymbol{W}_{v,1}, \ldots, \boldsymbol{W}_{v,h}\}$, where $\boldsymbol{W}_{a,b} \in \mathbb{R}^{d/h \times \Gamma}$ for all $a = q, k, v$ and $b = 1, \ldots, h$. Here, h is the number of attention heads. This gives us h output matrices $\boldsymbol{O}_h$ (*cf.* (3.36)), which are subsequently combined with a FC linear layer.

Finally, the original authors of the (self-)attention mechanism [90] introduce a few components to the attention mechanism to help with training: (i) they use residual layers [97] (Res.) around the self-attention mechanism and the FC linear layer; (ii) they introduce layer normalisation [98] after the self-attention mechanism and the FC linear layer; (iii) and, they scale the inner product (*cf.* (3.32)) by $\sqrt{d}$, with the dimension d of the embedding space. Figure 3.6 depicts the complete self-attention processing block. The main self-attention mechanism is summarised in the "Multi-head Attention" sub-block. The "Res. & Norm." blocks represent the residual connections and layer normalisation.

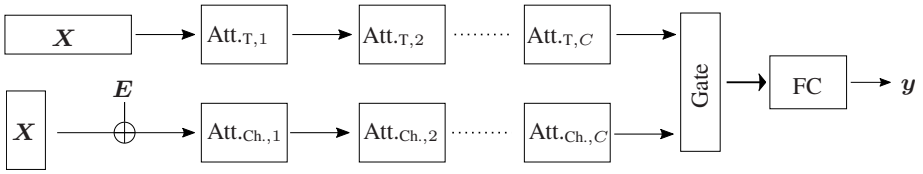


Figure 3.7: Abstract gated transformer architecture for multi-variate time-series data.

3.3.4.2 Gated Transformer Network

The original transformer network (see [90]) was designed to perform language translation tasks, where the authors introduced an encoder-decoder architecture. The encoder part is built by stacking multiple multi-head self-attention blocks (as introduced in Section 3.3.4.1) on top of each other and then feeding the corresponding output embedding to the transformer decoder blocks. The decoder blocks are used in sequence-to-sequence tasks as introduced in [90], e.g., when a sentence is translated from one language to another. Since we are interested in time-series classification tasks, i.e., we directly classify the inputs using the representations after the self-attention heads, we do not introduce the decoder part here.

To classify time-series data, the authors of [95] introduce the gated-transformer. As depicted in Figure 3.7, this architecture was designed to take the structure of time-series data into account by considering the multi-variate time-series data over the time-stamps (the "Att.$_{\mathrm{T}}$" blocks) and over the channels (the "Att.$_{\mathrm{C}}$" blocks) in parallel. We observe that the multi-variate time-series signal processed over the channels requires the positional encoding ($\boldsymbol{E}$, *cf.* (3.38)) whereas when it is processed over the time-stamps, this is not required. Processing the multi-variate time-series signal in this way is similar to the two perspectives introduced in Definition 2. After processing the time-series signals through $2C$ self-attention blocks, they introduce a gating mechanism (similar to the gating in an RNN) to combine the representations depending on their importance and finally pass the signal through a FC linear layer.

3.3.5 Deep Autoencoders

A final DNN architecture we use is a Deep Autoencoder (DAE) with 1-D convolutional layers.[7] DAEs with convolutional layers were first introduced in [99]. We use 1-D convolutional layers since they show good representation power and they can extract discriminative features in multi-variate time-series data, see, e.g., [100].

[7]We can use any of the previously introduced 1-D convolutional layers in the DAE (*cf.* Section 3.3.2).

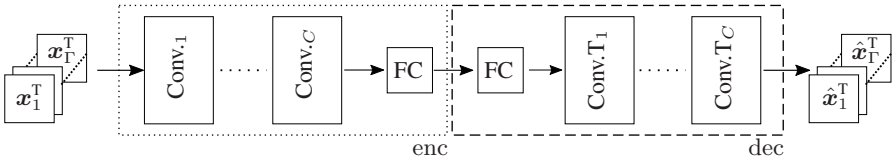


Figure 3.8: Abstract DAE architecture for multi-variate time-series data.

The general architecture of a DAE is depicted in Figure 3.8. It is built of two functions: the encoder and the decoder. On the left side, we see the Γ input channels of a multi-variate time-series signal which are first passed through the encoder function $\mathrm{enc} : \mathbb{R}^{\Gamma \times N} \to \mathbb{R}^p$ (dotted box), and then the signal is passed through the decoder function $\mathrm{dec} : \mathbb{R}^p \to \mathbb{R}^{\Gamma \times N}$ (dashed box). The encoder function attempts to learn a lower dimensional representation, i.e., we assume $p \ll \Gamma \times N$, of the input signal, whereas the decoder function attempts to reconstruct the original input signal given this representation, i.e., $\hat{\boldsymbol{x}}_\gamma^\mathrm{T} \approx \boldsymbol{x}_\gamma^\mathrm{T}$. Since the latent space dimension is smaller than the input dimension ($p \ll \Gamma \times N$), this creates a bottleneck where the DAE must compress the information in the input signal in the latent space representations. Generally, the encoder and the decoder networks are designed symmetrically, with the same number of convolutional blocks in each. Since we are going from a low to a high dimensional signal in the decoder function, we need to use transposed convolutional layers or convolutional layers with upsampling so that the dimensions at the output match. The last layer of the encoder function and the first layer of the decoder function are both FC linear layers. Therefore, the DAE function can be summarised as

$$f_\mathrm{DAE}(\boldsymbol{X}) = \mathrm{dec}(\mathrm{enc}(\boldsymbol{X}; \mathcal{P}_E); \mathcal{P}_D), \tag{3.39}$$

with the encoder parameters $\mathcal{P}_E$ and the decoder parameters $\mathcal{P}_D$.

3.4 Optimisation

In this section, we discuss the loss functions and optimisation techniques used to train the previously introduced DNN architectures. To this end, we assume we have chosen an DNN architecture and a training dataset of labelled input/output samples $\mathcal{D}_\text{train} = \{(\boldsymbol{X}^1, y^1), (\boldsymbol{X}^2, y^2), \ldots, (\boldsymbol{X}^{M_\text{train}}, y^{M_\text{train}})\}$ with inputs $\boldsymbol{x} \in \mathbb{X}$ and labels $y \in \mathbb{Y}$.[8]

[8]Note, the techniques to train a model in an unsupervised or self-supervised manner, i.e., when the output labels are not used to calculate the loss, are not discussed here. The interested reader can refer to [69, Ch. 14].

3.4.1 Cross Entropy Loss

When considering a classification task, i.e., the case where the set of labels $\mathbb{Y}$ is discrete with K different labels, we use the cross entropy loss. To employ this loss, we employ the softmax function after the final FC linear transformation of the chosen DNN architecture. This approximates the *posterior* probability of each class. The output of the DNN architecture can thus be expressed as

$$\boldsymbol{x}^{(\Lambda)} = \mathrm{softmax}\left(\boldsymbol{W}^{(\Lambda)}\boldsymbol{x}^{(\Lambda-1)} + \boldsymbol{b}^{(\Lambda)}\right) \in (0,1)^K, \tag{3.40}$$

with the weight matrix $\boldsymbol{W}^{(\Lambda)}$ and the bias vector $\boldsymbol{b}^{(\Lambda)}$ of the final layer. The $\mathrm{softmax}$ function is defined as

$$\mathrm{softmax}(\boldsymbol{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}, \tag{3.41}$$

for a vector $\boldsymbol{z} \in \mathbb{R}^K$.

In the context of a classification problem, we can tune the parameters $\mathcal{P}$ of the DNN architecture using the cross entropy loss defined as

$$\mathcal{L}_{\text{ce}}(\mathcal{D}_{\text{train}}; \mathcal{P}) = -\frac{1}{M_{\text{train}}} \sum_{m=1}^{M_{\text{train}}} \sum_{c=1}^{K} t_c^m \ln\left(x_c^{m,(\Lambda)}\right), \tag{3.42}$$

with the one-hot-encoded true labels $t_c^m \in \{0,1\}$ for all of the training data indexed by m, and the *posterior* probabilities defined in (3.40) for each input $\boldsymbol{x}^m$. We have M_{train} samples in the training dataset $\mathcal{D}_{\text{train}}$ and K class labels.

3.4.2 Mean Squared Error Loss

On the other hand, when considering a reconstruction task, i.e., the case where we attempt to reconstruct the input (*cf.* Section 3.3.5), we can use the Mean Squared Error (MSE) loss function. To employ this loss function, we generally use a linear activation function, i.e., no activation function, at the output. We can tune the parameters of the DNN architecture $\mathcal{P}$ using the MSE loss defined as

$$\mathcal{L}_{\text{MSE}}(\mathcal{D}_{\text{train}}; \mathcal{P}) = \frac{1}{M_{\text{train}} N^2} \sum_{m=1}^{M_{\text{train}}} \|\boldsymbol{X}^m - f(\boldsymbol{X}^m; \mathcal{P})\|_F^2, \tag{3.43}$$

with the Frobenius norm $\|\cdot\|_F$ and ML algorithm represented as the function $f : \mathbb{R}^{N\times N} \to \mathbb{R}^{N\times N}$, assuming the inputs are $\boldsymbol{X} \in \mathbb{R}^{N\times N}$. Again, assuming we have M_{train} samples in the training dataset $\mathcal{D}_{\text{train}}$.

3.4.3 Gradient Descent Algorithm

Once we have chosen an architecture and defined the loss function for the given problem, we need to tune the parameters of the DNN architecture. Since the loss function is a non-convex, non-linear function of the parameters $\mathcal{P}$, no closed form or global optimal solutions exist (*cf.* [68, Ch. 8]). To optimise the parameters, we rely on an iterative gradient-descent optimisation algorithm, i.e., the parameters are updated iteratively according to the gradient at the current inputs and a given step-size (also referred to as the learning rate), i.e.,

$$\boldsymbol{P}^{(t+1)} \leftarrow \boldsymbol{P}^{(t)} - \eta \nabla_{\boldsymbol{P}} \mathcal{L}(\mathcal{D}_{\text{train}}; \mathcal{P}), \quad \forall \boldsymbol{P} \in \mathcal{P}, \tag{3.44}$$

with the step-size η, iteration index t, and the gradients $\nabla_{\boldsymbol{P}}$ of the loss function w.r.t. to the parameters of the DNN architecture. This gradient-based optimisation is efficiently implemented by the backpropagation algorithm [101].

To improve the optimisation both in terms of speed of convergence and avoiding getting stuck in local optima, one usually performs gradient-descent on mini-batches (random sub-sets of the training data). This is known as Stochastic Gradient Descent (SGD). More advanced methods to improve the convergence of the gradient-descent algorithm have been introduced in the literature, e.g., the ADAM update rule [102] or the RMSprop update rule, which both introduce momentum terms and additionally change the step-size η depending on the current mini-batch and past gradient information.

Since this method of optimisation has no guarantees on the optimality of the found parameters, we can attempt to avoid overfitting the training dataset by:[9] using random initialisations of the parameters; training various architectures on different k-folds of the training dataset [69, Ch. 7.2]; early stopping by tracking the loss on the training set and a validation set [69, Ch. 11.5]; regularising the parameters of the network; or, introducing a batch normalisation [103] or layer normalisation [98] (as seen in the attention mechanism *cf.* Section 3.3.4.1).

3.5 Use-Case: Time to Lane Change Prediction

We consider a Time to Lane Change (TTLC) prediction task, i.e., the task of predicting when tracked vehicles are going to change lanes in a highway scenario, as a use-case. Designing an ML algorithm to perform well on this task not only increases the safety of the AV, but it can also increase the comfort of the passengers riding in the AV.

[9] Note, this is by no means an exhaustive list of methods to improve training ML algorithms; these are merely the methods used in this work to help improve the training.

Attribute	Unit	Description
$v_{\text{lat.}}$	$\mathrm{m/s}$	Lateral velocity
$v_{\text{long.}}$	$\mathrm{m/s}$	Longitudinal velocity
$a_{\text{lat.}}$	$\mathrm{m/s^2}$	Lateral acceleration
$a_{\text{long.}}$	$\mathrm{m/s^2}$	Longitudinal acceleration
d_{left}	m	Distance to left lane marking
d_{right}	m	Distance to right lane marking
$d_{\text{pre.}}$	m	Distance to preceding vehicle in same lane
$d_{\text{foll.}}$	m	Distance to following vehicle in same lane
$d_{\text{l, pre.}}$	m	Distance to preceding vehicle in left lane
$d_{\text{l, foll.}}$	m	Distance to following vehicle in left lane
$d_{\text{r, pre.}}$	m	Distance to preceding vehicle in right lane
$d_{\text{r, foll.}}$	m	Distance to following vehicle in right lane

Table 3.1: Twelve of the possible input attributes.

In this section, we first introduce the highway driving datasets used to train the ML algorithms on, and then we formally define the TTLC prediction task.

3.5.1 Highway Driving Datasets

As input data for the ML models, we use two publicly available and widely used highway driving datasets—the highD dataset [7] and the Next Generation SIMulation (NGSIM) [8; 9] datasets. These datasets all summarise the lane change manoeuvres of vehicles as multi-variate time-series samples.

The highD dataset [7] was collected by flying a drone above six German highway segments in various locations. In total, more than 16 hours of highway driving data were collected containing thousands of lane changes. The recordings from the different locations are combined into a single dataset.

The NGSIM dataset was collected by the NGSIM program on US highways by filming the vehicles from multiple cameras fixed on top of skyscrapers close to the highways. The researchers recorded the trajectories of vehicles for a period of time on both the US highway 101 [8] and the I-80 Freeway [9]. In total, 1.5 hours of driving data were collected; they include a few hundred lane changes each. Since both datasets were collected within the scope of the same project, i.e., using the same pre- and post-processing techniques, we combine them into a single NGSIM dataset. This is

equivalent to the highD dataset, which is also a collection of driving scenarios from various locations.

From the video data, the authors use an image detection algorithm and a tracking algorithm to extract the trajectories of the vehicles. Each vehicle trajectory contains various attributes. We assume that the vehicles are tracked for at least N time-stamps, defining a scenario. We summarise these in a multi-variate time-series signal (*cf.* Definition 2) as

$$\boldsymbol{X} = \Big[\bar{\boldsymbol{x}}[1], \quad \bar{\boldsymbol{x}}[2], \quad \ldots, \quad \bar{\boldsymbol{x}}[N]\Big] \in \mathbb{R}^{\Gamma \times N}, \tag{3.45}$$

where at each time-stamp $n = 1, \ldots, N$ the signal has Γ attributes.

We consider subsets of the input attributes listed in Table 3.1. The distances to the surrounding vehicles is measured from the centre of the tracked vehicle to the centre of the other vehicles. A snapshot from an exemplary scenario is illustrated in Figure 3.9. We see the longitudinal and lateral velocities and accelerations of the tracked vehicle, the distances to the left and right lane markings, and the distances to the surrounding vehicles. In Figure 3.9, there is no preceding vehicle in the same lane or following vehicle on the left-hand side of the tracked vehicle. Therefore, these distances $d_{\text{pre.}}$ and $d_{\text{l, foll.}}$ are set to a maximum value, e.g., $d_{\text{max}} = 200\ [\text{m}]$. Furthermore, we see that there are two categories of input attributes: those which describe the tracked vehicle's trajectory (in the red boxes), and those which describe the interaction between the other vehicles in the scenario (in the blue box). The interaction features define the social dynamics between road vehicle users. These have been considered in previous ML research, see, e.g., [104; 105]. It can be interesting to observe which category of features an ML-based algorithm focusses on given different driving tasks.

3.5.2 Time to Lane Change Classification Problem

To define the TTLC prediction problem, we first extract all scenarios where the vehicle was in view for at least N time-stamps. We assume that the driving manoeuvre to be predicted occurs at time-stamp N. There are three possible driving manoeuvres to be predicted: the vehicle either kept in its lane for all N time-stamps, the vehicle changed lanes to the left or it changed lanes to the right.

To define the TTLC prediction problem as a sub-sequence classification problem, we first take the multi-variate time-series signals (*cf.* (3.45)), and divide each into $P = N/N_{\text{sub.}} \in \mathbb{N}$ sub-sequences of equal length $N_{\text{sub.}}$. Henceforth, every sub-sequence is considered an input datum $\boldsymbol{X} \in \mathbb{R}^{\Gamma \times N_{\text{sub.}}}$.

Thus, every datum either belongs to a scenario where a vehicle kept in its lane or changed lanes; left lane changes are labelled L, right lane changes are labelled R, and

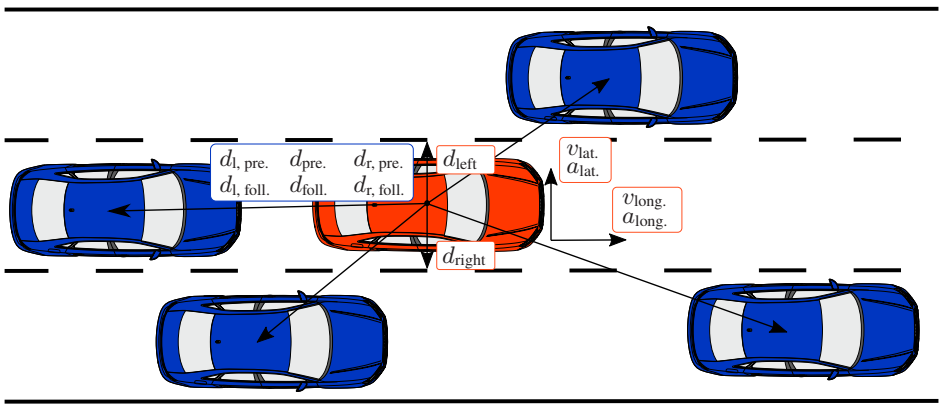


Figure 3.9: Highway scenario depicting the possible input attributes (see Table 3.1).

lane keeping manoeuvres are labelled K. We further index the labels of the lane change sub-sequences as L_ρ and R_ρ, for $\rho = 1, \ldots, P$, depending on how many sub-sequences in the future the lane change occurs. For example, the label L_1 corresponds to a sub-sequence where a left lane change occurs in $1 \cdot N_{\text{sub.}}$ time-stamps, the label L_2 corresponds to a sub-sequence where the left lane change occurs in $2 \cdot N_{\text{sub.}}$ time-stamps, and so on.

Ultimately, the TTLC sub-sequence classification problem is to predict the label of each sub-sequence $\boldsymbol{X} \in \mathbb{R}^{\Gamma \times N_{\text{sub.}}}$. The set of possible class labels is defined as $\mathbb{Y} = \{\mathrm{L}_1, \mathrm{L}_2, \ldots, \mathrm{L}_P, \mathrm{K}, \mathrm{R}_1, \mathrm{R}_2, \ldots, \mathrm{R}_P\}$. The dataset of driving manoeuvres is summarised as

$$\mathcal{D} = \left\{ (\boldsymbol{X}^1, y^1), (\boldsymbol{X}^2, y^2), \ldots, (\boldsymbol{X}^M, y^M) \right\}, \tag{3.46}$$

where each input is a multi-variate time-series datum $\boldsymbol{X} \in \mathbb{R}^{\Gamma \times N_{\text{sub.}}}$ and has the corresponding class labels $y \in \mathbb{Y}$. In total, the whole dataset contains M samples.

4 Validation Safety Arguments based on Feature Embeddings

In this chapter, we introduce methods to generate safety arguments to validate Deep Neural Network (DNN) architectures by investigating their feature embeddings (*cf.* Section 4.2). At first, we might believe that comparing DNN architectures based on their classification performance, e.g., their accuracies, is sufficient to choose one for a safety-critical function (*cf.* Section 4.4). However, as we will see, the quality of their feature embeddings can differ significantly when passively studied. To this end, we introduce a feature validation method (*cf.* Section 4.3) where we cluster and visualise the feature embeddings to compare DNN architectures on another performance measure. We argue that this analysis should be performed before choosing a DNN architecture for a safety-critical Machine Learning (ML)-based Highly Automated Driving (HAD) function. Additionally, we introduce methods to actively encourage better clustering of the feature embeddings (*cf.* Section 4.5), thus, strengthening the safety argument based on feature embeddings. Finally, we propose methods to reject inconsistent classification outputs of a DNN architecture (*cf.* Section 4.6) inspired by the feature validation method. The results presented in this chapter are based on those found in our work [15; 18].

4.1 Feature Embedding Motivation

In this section, we motivate developing a safety argument based on the feature embeddings of DNNs. It is well known that modern DNN architectures show super-human performance on a myriad of tasks—from image classification problems to protein folding predictions. This ability comes from the fact that DNNs learn useful representations (or features), i.e., the activations in the hidden layers of a DNN, by transforming the input data into a space where the task at hand can be performed well. The features in the initial layers of a DNN have been shown to be more general, whereas the output features are specific to the task at hand [106]. Researchers often take advantage of this fact by pre-training large DNNs extensively on large datasets,

and then fine-tuning the output parameters on a new task, which is contained in the field of transfer learning [68, Ch. 15] (*cf.* Section 5.1).

Other work has focused on using the features in various layers to robustify DNNs, increase confidence in the classification output, or detect adversarial examples [107]. They achieve this by quantifying the conformity of the features of a given input to those of the training dataset. On the other hand, researchers have used the features at the output layer of a DNN to identify out-of-distribution samples and enable open-set classification [108]. This is possible because the features at the output layer of a DNN show similar activation patterns for similar concepts, e.g., in image classification problems, the feature activation patterns of sharks are similar to scuba divers but different from baseball images [108, Fig. 1].

Another emerging field of research is visualising the features in various layers of a DNN [109; 110]. This enables researchers to take a closer look at the concepts learned by DNNs and gain an understanding of how human understandable concepts can be found in the feature activations of a DNN.

To this end, we propose feature validation methods (see [15; 18]) which allow an engineer to investigate the extracted feature embeddings of various DNN architectures and then justify the choice of an architecture for a given task. We argue that just because multiple DNN architectures show a similar performance on a given task, does not mean that the feature embeddings of those architectures are of the same quality or as understandable by a human user. We consider this a passive feature validation method since the feature embeddings of the various DNN architectures are only investigated and a safety argument is built around this analysis. Following from this passive feature validation method, we extend the proposed method to actively encourage a better clustering of the extracted feature embeddings.

4.2 Feature Embedding Architecture

In order to validate the features of different DNN architectures, we first abstract the feature embedding in the penultimate layer of a DNN as highlighted in Fig. 4.1. To achieve this, we pass the input samples through the DNN and extract the corresponding feature embeddings. This is a useful abstraction since the feature embeddings in the penultimate layer of a DNN must learn a meaningful representation of the data for the task at hand, e.g., for a classification task these feature embeddings must be linearly separable to the output classes. We observe in Fig. 4.1 that this abstraction of the first

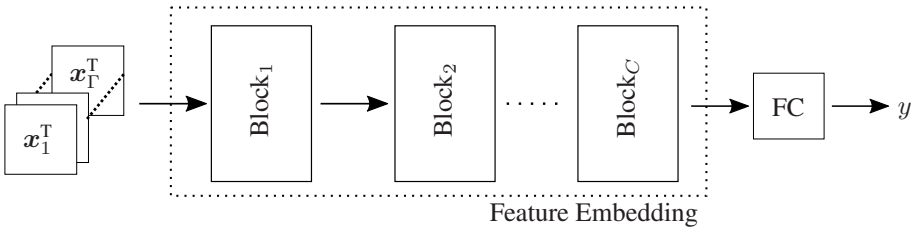


Figure 4.1: Abstract DNN architecture for multi-variate time-series data, highlighting the feature embedding.

C processing blocks can be written as a function

$$\begin{aligned} \mathrm{feat} : \mathbb{R}^{\Gamma \times N} &\rightarrow \mathbb{R}^{N_{(\Lambda-1)}} \\ \boldsymbol{X} = \begin{bmatrix} \boldsymbol{x}_1^{\mathrm{T}} \\ \vdots \\ \boldsymbol{x}_\Gamma^{\mathrm{T}} \end{bmatrix} &\mapsto \boldsymbol{x}^{(\Lambda-1)}, \end{aligned} \tag{4.1}$$

which maps a multi-variate time-series input datum ($\boldsymbol{X}$) to a feature embedding ($\mathrm{feat}(\boldsymbol{X})$) in the penultimate layer of the DNN (assuming there are Λ layers in the DNN). This feature embedding function will be used as the foundation of the feature validation methods presented here.

As introduced in Section 3.3, there are many different possible mappings of the C blocks of a DNN to transform the input signal to the feature embeddings, e.g., Fully Connected (FC) layers, convolutional layers, or attention-based layers. Depending on the application at hand, one architecture might appear to be advantageous compared to the others in terms of a performance criterium, e.g., the classification accuracy. However, this does not directly indicate which DNN architecture should be used for a safety-critical HAD function. As we see in this chapter, the feature embeddings of different DNN architectures can vary significantly—both in terms of their clustering and in terms of their interpretability. Therefore, an engineer should additionally investigate the DNN architectures in more detail in order to choose one for an ML-based HAD function.

4.2.1 Classification Architectures

In this chapter, we demonstrate the feature validation method by comparing the performance of various DNN architectures on the Time to Lane Change (TTLC) classification task (*cf.* Section 3.5.2). To this end, we train the DNN architectures

depicted in Table 4.1. The architecture description column gives a brief reminder of the different DNN architectures; for more details refer to Section 3.3. We see there are multiple 1-D Convolutional Neural Network (CNN)-based architectures (including Multi-Channel (MC) architectures), Recurrent Neural Network (RNN)-based architectures (using either Long Term Short Term Memory (LSTM) or Gated Recurrent Unit (GRU) cells), an attention-based architecture, and an FC-DNN. This is a wide range of different DNN architectures including multiple state-of-the-art designs from the Autonomous Vehicle (AV) and the time-series classification literature.

Name	Architecture description
FC-DNN	Fully-connected DNN (*cf.* Section 3.3.1)
CNN-I	Standard 1-D CNN (*cf.* Section 3.3.2.1)
CNN-II	Depth-wise Sep. 1-D CNN (*cf.* Section 3.3.2.2)
CNN-III	Local 1-D CNN (*cf.* Section 3.3.2.3)
CNN-I MC	Multi-channel Standard 1-D CNN (*cf.* Section 3.3.2.5)
CNN-II MC	Multi-channel Depth-wise Sep. 1-D CNN (*cf.* Section 3.3.2.5)
CNN-III MC	Multi-channel Local 1-D CNN (*cf.* Section 3.3.2.5)
LSTM-[10]	RNN with LSTM cells (*cf.* Section 3.3.3)
GRU-[12]	RNN with GRU cells (*cf.* Section 3.3.3)
Gated Tr.-[95]	Attention-based Transformer (*cf.* Section 3.3.4)

Table 4.1: The various DNN architectures trained on the TTLC classification problem and then validated using the feature validation method.

4.2.2 Simulation Setup

We consider the TTLC classification task introduced in Section 3.5.2. To generate a suitable dataset to train the DNNs, we first extract all scenarios from the highD dataset [7] (*cf.* Section 3.5.1) where the tracked vehicle is visible for $N = 150$ time-stamps prior to the driving manoeuvre. Since the data are recorded at 25 Hz, this is equivalent to 6 s. Furthermore, we consider sub-sequences of length 2 s, such that each lane change manoeuvre is split into sub-sequences of length $N_{\text{sub.}} = 50$ time-stamps. We have a total of three classes for left and right lane changes, and one class for the lane keeping manoeuvre, i.e., $\mathbb{Y} = \{\mathrm{L}_1, \mathrm{L}_2, \mathrm{L}_3, \mathrm{K}, \mathrm{R}_3, \mathrm{R}_2, \mathrm{R}_1\}$ (*cf.* Section 3.5.2). The

dataset can be summarised as

$$\mathcal{D} = \{(\boldsymbol{X}^1, y^1), (\boldsymbol{X}^2, y^2), \ldots, (\boldsymbol{X}^M, y^M)\}, \tag{4.2}$$

where each input multi-variate time-series $\boldsymbol{X} \in \mathbb{R}^{\Gamma \times N_{\text{sub.}}}$ and the class labels $y \in \mathbb{Y}$. We consider almost all of the input attributes introduced in Table 3.1 except for the distance to the left and the right lane markings; thus, in this chapter, we consider $\Gamma = 10$ attributes.

After splitting the scenarios into sub-sequences and balancing the classes, we have a total of $21{,}854$ samples; these are separated into two disjunct sets for training and for testing the DNNs. We use 80% of the data for the training set and 20% for the test set, i.e., $M_{\text{train}} = \lfloor 0.8 \cdot M \rfloor$ and $M_{\text{test}} = M - M_{\text{train}}$.

To train the DNNs, we use the cross-entropy loss function (*cf.* Section 3.4.1). We train each algorithm using the Adam optimiser [102] with an initial learning rate of $\eta = 0.0005$ and mini-batches of size 200. All of the network parameters are initialised using the Glorot initialisation [111]. We employ early stopping with a patience of 20 epochs to reduce overfitting the training set. Additionally, we average the classification performance using 10-fold cross validation (see [69]). The exact parameters and architecture designs of the models in Table 4.1 can be found in Appendix C.1.

4.3 Feature Embedding Safety Argument

The proposed method to qualitatively validate the extracted feature embeddings in DNNs can be summarised in two steps:

Step 1) After training the DNN, the feature embeddings of the test data, i.e., data which are unseen during training, are extracted using the feat function (*cf.* (4.1)). These feature embeddings are clustered using an unsupervised clustering algorithm; in the experiments we use k-means clustering. We vary the number of clusters and examine them. Since we have the true class labels, the quality of the clustering of the different DNN architectures can be quantified using an external cluster validation method. This is done by comparing the cluster labels, i.e., the outputs of the clustering algorithm, to the true class labels.

Step 2) The high-dimensional feature embeddings are projected into a lower dimension using a dimensionality reduction technique; in the experiments we use the Uniform Manifold Approximation and Projection (UMAP) [112]. These low-dimensional representations are visualised to allow the engineer to investigate the meaningfulness of the feature embeddings and the clusters from Step 1). A

feature embedding space is considered meaningful if the feature embeddings of similar classes are embedded close together and those of different classes are far apart. Moreover, if the feature embeddings of different classes do not overlap, the embedding space is considered meaningful. This is especially important for safety-critical driving manoeuvres. For example, samples belonging to the class right lane change in the next seconds (R_1) should be embedded far apart from those belonging to the class left lane change in the next seconds (L_1).

An engineer can use this feature validation method to compare different DNN architectures which might show a similar performance in terms of, e.g., the classification accuracy. If one of the DNN architectures has more meaningful feature embeddings, that architecture might be preferred in a given safety-critical application. Furthermore, the fact that the feature embeddings are meaningful and have been investigated can be incorporated into the overall validation safety argument.

4.4 Standard Feature Embedding Spaces

In this section, we discuss the results of applying the feature validation method directly to the DNNs introduced in Section 4.2. First, we discuss the classification results and demonstrate that the embedded features are suitable to develop a safety argument upon. Following this, we apply the feature validation method and discuss the elements in more detail.

4.4.1 Classification Results

First, we benchmark the DNN architectures introduced in Section 4.2. We train the algorithms using the simulation setup discussed in Section 4.2.2.

In Table 4.2, we depict the classification results of the various DNNs. As evaluation metrics, we use both the classification accuracy and the macro-averaged F_1 score [113]. The $F_{1,y}$ score for class y is calculated as the harmonic mean of the precision and recall scores

$$F_{1,y} = 2\frac{\text{Precision}_y \cdot \text{Recall}_y}{\text{Precision}_y + \text{Recall}_y} \tag{4.3}$$

where the precision and recall scores are defined as $\text{Precision}_y = (\text{TP}_y)/(\text{TP}_y + \text{FP}_y)$, $\text{Recall}_y = (\text{TP}_y)/(\text{TP}_y + \text{FN}_y)$ with the true positives (TP_y), false positives (FP_y), and false negatives (FN_y) of a given class y. The macro-averaged F_1 score is the arithmetic mean of the $F_{1,y}$ scores per class, i.e., $F_1 = \sum_y F_{1,y}/\left|\mathbb{Y}\right|$ with the total

Architecture	Standard Space	
	Accuracy [%]	F_1 [%]
FC-DNN	$88.68(\pm 1.01)$	$88.64(\pm 1.00)$
CNN-I	$90.41(\pm 0.37)$	$90.37(\pm 0.37)$
CNN-II	$87.29(\pm 0.47)$	$87.20(\pm 0.51)$
CNN-III	$90.15(\pm 0.38)$	$90.12(\pm 0.40)$
CNN-I MC	$91.30(\pm 0.54)$	$91.24(\pm 0.55)$
CNN-II MC	$90.54(\pm 0.24)$	$90.50(\pm 0.24)$
CNN-III MC	$89.78(\pm 1.74)$	$89.69(\pm 1.79)$
LSTM-[10]	$91.29(\pm 0.76)$	$91.24(\pm 0.77)$
GRU-[12]	$92.48(\pm 0.36)$	$92.49(\pm 0.37)$
Gated Tr.-[95]	$90.41(\pm 0.60)$	$90.41(\pm 0.60)$

Table 4.2: Classification performance of the various DNN architectures for the standard feature embedding space. The results are averaged over the 10-folds with the 95% confidence interval in brackets.

number of classes $|\mathbb{Y}|$. All of the values are averaged over the 10-folds and the 95% confidence intervals are shown in brackets.

We observe in Table 4.2 that the RNN-based architectures show a slightly better performance overall, which is not surprising as these architectures were designed specifically for the TTLC prediction task as taken from the literature. The attention-based and CNN architectures perform almost as well as the RNN-based architectures on this classification task. The CNN-II and FC-DNN architectures show the worst classification performance, but not by much. Overall, we see that these architectures all perform reasonably well on the given task, and no architecture outperforms the others by so much that we would immediately choose it.

To further support the claim that all of the DNN architectures show a similar classification performance, we train a k-Nearest Neighbours (k-NN) [114] and a Decision Tree (DT) [115] classifier. To achieve this, we first embed the training data, and train a k-NN and a DT on these feature embeddings. We use $k = 5$ neighbours for the k-NN, and allow the DT to train to minimum impurity on the training set.[1] In

[1]Note, training the DT to minimum impurity is default when using the `sklearn` [116] package. We attempted to tune the DT parameters, e.g., maximum depth of the tree or the minimum number of leaves per node, however, we did not find any significant improvement of the performance on the test dataset.

Architecture	k-NN		DT	
	Accuracy [%]	F_1 [%]	Accuracy [%]	F_1 [%]
FC-DNN	$89.22(\pm 0.25)$	$89.19(\pm 0.25)$	$86.48(\pm 0.22)$	$86.47(\pm 0.23)$
CNN-I	$89.54(\pm 0.21)$	$89.48(\pm 0.21)$	$87.54(\pm 0.19)$	$87.52(\pm 0.19)$
CNN-II	$86.08(\pm 0.31)$	$86.01(\pm 0.31)$	$82.58(\pm 0.40)$	$82.54(\pm 0.40)$
CNN-III	$89.49(\pm 0.35)$	$89.44(\pm 0.36)$	$86.42(\pm 0.51)$	$86.40(\pm 0.52)$
CNN-I MC	$87.51(\pm 0.30)$	$87.45(\pm 0.29)$	$87.46(\pm 0.26)$	$87.45(\pm 0.26)$
CNN-II MC	$85.43(\pm 0.49)$	$85.39(\pm 0.47)$	$86.07(\pm 0.34)$	$86.05(\pm 0.34)$
CNN-III MC	$85.96(\pm 1.59)$	$85.95(\pm 1.58)$	$85.66(\pm 1.97)$	$85.63(\pm 1.98)$
LSTM-[10]	$92.34(\pm 0.30)$	$92.33(\pm 0.29)$	$89.92(\pm 0.37)$	$89.89(\pm 0.38)$
GRU-[12]	$92.79(\pm 0.30)$	$92.79(\pm 0.30)$	$90.57(\pm 0.43)$	$90.56(\pm 0.43)$
Gated Tr.-[95]	$90.65(\pm 0.58)$	$90.63(\pm 0.59)$	$86.76(\pm 1.37)$	$86.77(\pm 1.37)$

Table 4.3: Classification results of the various DNN architectures classifying the feature embeddings in the standard space using a k-NN and a DT. The results are averaged over the 10-folds with the 95% confidence interval in brackets.

Table 4.3, we present the classification results on the feature embeddings of the test data. We observe that the classification performance—both in terms of the accuracy and the F_1 score—are almost the same as using the full DNN, i.e., classifying the test data using the trained DNNs. We observe that the DTs trained on the feature embeddings show a slightly worse performance, but this is likely due an overfitting of the training data. However, the k-NNs perform even better than the DNN for some architectures, e.g., the Gated Tr.-[95] or the GRU-[12]. Overall, these results indicate that the feature embeddings learned by the various DNN architectures are representative enough such that a simple classifier (here a k-NN or a DT) is able to perform just as well as the DNN model.

Thus, we conclude that it is meaningful to investigate the feature embeddings in the penultimate layer of various DNN architectures, as the representations in this layer are expressive enough to classify the data. Moreover, the achievable classification performance is independent of the final linear transformation.

4.4.2 Feature Validation in Standard Spaces

As we observe in Table 4.2 and Table 4.3, all of the DNN architectures show similar classification performances on the TTLC classification task. However, since this is a safety-critical HAD function, an engineer would want to rely on more than just the classification performance to decide which architecture to use. To this end, we employ the passive feature validation method (*cf.* Section 4.3) to validate the feature embeddings extracted by the different DNNs. By employing this method, we are able to investigate the various architectures and compare them using a measure other than just the classification accuracy. We will refer to the feature embeddings of the DNNs trained with the standard cross-entropy loss to be in the *standard feature embedding space*.

4.4.2.1 Feature Clustering

In the first step of the proposed feature validation method, we cluster the extracted feature embeddings of the test data using an unsupervised clustering algorithm: k-means clustering with $k = K'$ clusters.[2] After clustering, we obtain a set of cluster labels $\{c'_1, \ldots, c'_{K'}\}$. We vary the number of clusters K'. Moreover, we use the fact that we have the true labels of each embedded feature, such that we can use an external cluster validation method to quantify the clustering results.

We use the Adjusted Rand Index (ARI) [117; 118] to validate the cluster labels.[3] The ARI gives the percentage of samples where the cluster label and the true class label are the same. The rand index lies within the interval $[0, 1]$, however, the ARI can become slightly negative if the rand index is larger than its expected value. The ARI is adjusted for randomness in the clustering labels, e.g., a permutation of the clustering labels would produce a different rand index, but this is corrected for when using the ARI. A low ARI indicates that the cluster labels do not correspond to the true class labels, and a high ARI indicates that they do. The ARI is calculated based on the contingency matrix $\boldsymbol{D} \in \mathbb{N}^{K \times K'}$, where every row corresponds to one of the true labels and every column corresponds to one of the possible cluster labels (assuming there are K' cluster labels). The matrix element $[\boldsymbol{D}]_{i,j} \in \mathbb{N}$ counts the number of

[2]Note, during the DNN training, the test dataset $\mathcal{D}_{\text{test}}$ remained unseen.

[3]Note, the cluster labels $\{c'_1, \ldots, c'_{K'}\}$ do not necessarily have an obvious relationship with the true class labels $y \in \mathbb{Y}$.

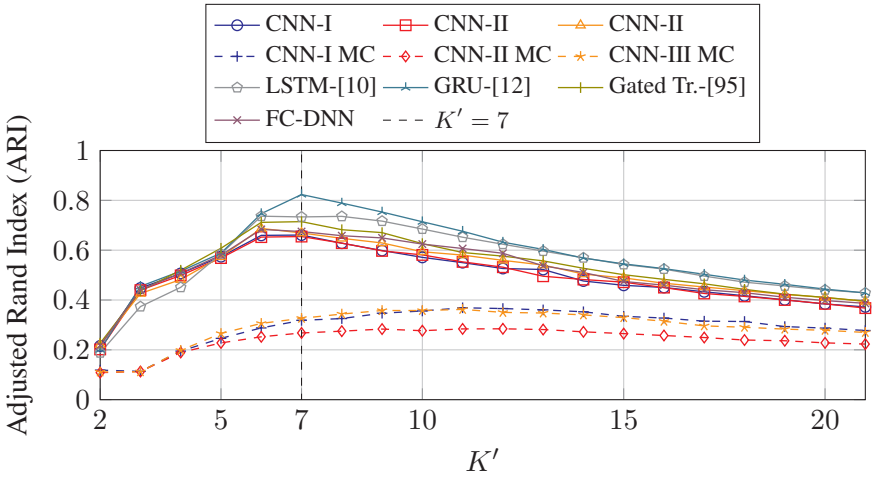


Figure 4.2: The average ARI values vs. the number of clusters K' in the standard feature embedding space.

samples in cluster j with the true class label i. The ARI is calculated as [118]

$$\text{ARI} = \frac{\sum_{i=1}^{K}\sum_{j=1}^{K'}\binom{[\boldsymbol{D}]_{i,j}}{2} - \left(\sum_{i=1}^{K}\binom{[\boldsymbol{D}]_{i,\cdot}}{2}\sum_{j=1}^{K'}\binom{[\boldsymbol{D}]_{\cdot,j}}{2}\right)/\binom{M_{\text{test}}}{2}}{\frac{1}{2}\left(\sum_{i=1}^{K}\binom{[\boldsymbol{D}]_{i,\cdot}}{2} + \sum_{j=1}^{K'}\binom{[\boldsymbol{D}]_{\cdot,j}}{2}\right) - \left(\sum_{i=1}^{K}\binom{[\boldsymbol{D}]_{i,\cdot}}{2}\sum_{j=1}^{K'}\binom{[\boldsymbol{D}]_{\cdot,j}}{2}\right)/\binom{M_{\text{test}}}{2}}, \tag{4.4}$$

where $\binom{a}{2}$ is a binomial coefficient defined as 0 if $a = 0$ or 1, and the notation $[\boldsymbol{D}]_{i,\cdot}$ and $[\boldsymbol{D}]_{\cdot,j}$ represents a sum over row i and a sum over column j, respectively. The total number of samples is M_{test}.

We run the k-means clustering algorithm with 100 random initialisations and take the clustering with the lowest inertia value, i.e., the smallest sum of squared distances between the samples and their closest cluster centre, for each number of clusters K'. We vary the total number of clusters within $K' \in \{2, \ldots, 3 \cdot K\}$, with the total number of true classes $K = |\mathbb{Y}|$; in the simulations $K = 7$.

The results of the clustering in the standard embedding space are depicted in Fig. 4.2. The ARI values are averaged over the 10-folds for each DNN architecture. Furthermore, we indicate $K' = 7$ on the horizontal-axis, since this is the total number of classes in the TTLC classification problem. Most of the DNN architectures show a peak at around $K' = 7$ which is unsurprising. However, the MC CNN architectures show a different behaviour—they have lower ARI values over the whole range of K' and their maximum ARI value is not at $K' = 7$, but they are still able to classify

the data reasonably well. This could be due to the fact that k-means clustering is not suitable in the feature embedding space generated by these architectures. The RNN-based architectures show the best ARI value at $K' = 7$, closely followed by the attention-based method. The CNN-based and the FC-DNN architectures all perform equally well, just slightly worse than the state-of-the-art architectures from the literature.

These results give us an interesting insight into the feature embedding spaces generated by the different DNN architectures: despite their similar classification performances, a clustering of the feature embeddings compared to the true test labels can vary depending on the chosen architecture. Therefore, if an engineer has the choice between different architectures, they might be more inclined to choose one whose feature embeddings cluster well. Additionally, we have another quality measure to evaluate and compare classification algorithms by. This could be added as a statement to the overall safety argument of the given ML algorithm.

4.4.2.2 Feature Visualisation

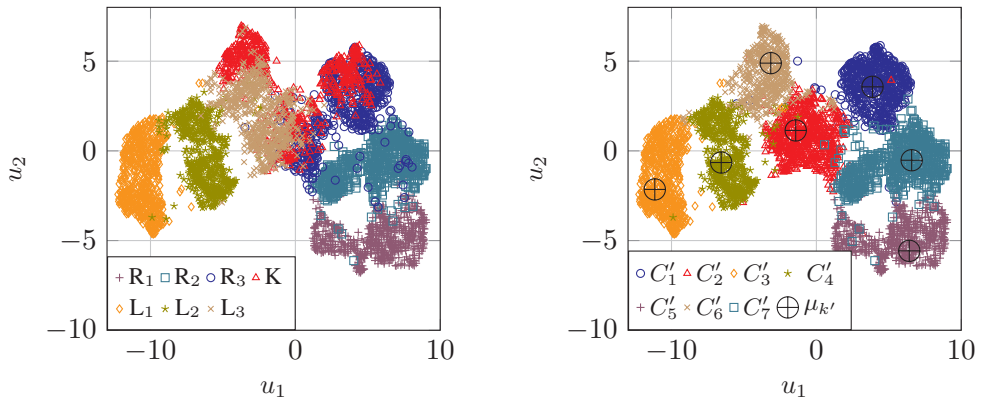
In the second step of the feature validation method, we visualise the high-dimensional feature embeddings. We do this in order to gain an understanding of the relationship between the feature embeddings of the different classes and to validate the k-means clustering algorithm performed in the first step. To this end, we visualise the extracted feature embeddings of the test data using a state-of-the-art dimensionality reduction technique, UMAP [112]. In the following, we discuss the UMAP results using the CNN-I architecture, however, a similar analysis can be performed on the feature embeddings of any of the other DNN architectures.[4]

The UMAP results for the CNN-I architecture are depicted in Fig. 4.3. We perform the UMAP algorithm using the following parameters: 15 neighbours in the high dimensional space[5]; and a minimum distance of 0.5 in the lower dimensional space. These parameters were chosen as they show a good representation of both the local and the global structures within the test dataset and within the different clusters.

In Fig. 4.3a, we see the UMAP embeddings of the test dataset in the standard feature embedding space. The samples are labelled with their corresponding label from the test dataset. We observe that those samples belonging to left and to right lane changes shortly before the lane change (labels: L_2, L_1, R_2, R_1, respectively) are embedded apart from each other. Moreover, the samples with these labels are clustered

[4] In Appendix A.1, we plot the UMAP representations of the feature embeddings of two further architectures: the GRU-[12] and the CNN-II MC.

[5] Note, the high dimension space at the input of the UMAP is the feature embedding space.



(a) The standard feature embedding space with true test data class labels.

(b) The standard feature embedding space with cluster labels.

Figure 4.3: The UMAP representation for the CNN-I architecture with different labels in the standard feature embedding space.

reasonably compactly together. On the other hand, the samples which are far away from the lane change (labels: L_3, R_3) are embedded close together and overlapping with the samples from the lane keeping scenarios (label: K). Intuitively, this can be explained by the fact that the input attributes which describe a lane change long before the event happens are similar to those which describe a lane keeping scenario, e.g., the tracked vehicle usually remains in the centre of their lane and does not accelerate in a lateral direction. The main difference between these three classes will be found in the attributes describing the social interactions in the scenario (*cf.* Section 3.5).

In Fig. 4.3b, we see the same UMAP samples, now labelled with the cluster labels from the k-means clustering performed in the first step of the feature validation method. This plot corroborates the ARI results we observe in Fig. 4.2, i.e., the cluster labels found via k-means clustering in the feature embedding space of the CNN-I architecture corresponds reasonably well to the true sample labels. We see that the samples with cluster labels C'_3, C'_4, C'_5, and C'_7 almost perfectly correspond to the class labels L_1, L_2, R_1, and R_2, respectively. We also see in Fig. 4.3b that the three overlapping classes L_3, R_3, and K are labelled as three clusters (C'_1, C'_2, and C'_6) by the k-means clustering. This would explain why the ARI was strictly smaller than 1 for this architecture. Moreover, we see that the cluster centres $\mu_{k'}$ are also reasonable. Overall, we observe in Fig. 4.3 that the feature embedding space generated by the CNN-I architecture is meaningful, and that the k-means clustering results are also

interpretable using the UMAP visualisations.

4.5 k-means Friendly Feature Embedding Spaces

Motivated by the interesting observation that DNN architectures showing equally good classification performance (in terms of their accuracies and their F_1 scores), can generate feature embeddings of varying quality. We now introduce a method to actively encourage meaningful feature embeddings of DNNs.

Thus, we build upon the passive analysis by actively creating a feature embedding space which shows—both quantitatively and qualitatively—better clustering. This not only strengthens the safety argument built from the feature validation method, but also improves the interpretability of the embedded features. Moreover, an engineer can strengthen the safety argument to an already validated DNN architecture which does not yet demonstrate promising clustering abilities by additionally employing k-means friendly training.

4.5.1 Generating k-means Friendly Spaces

We introduce a k-means friendly feature embedding space at the output of the feature embedding function (*cf.* (4.1)). This is inspired by the k-means friendly space used in the latent space of a Deep Autoencoder (DAE), see, e.g., [119; 120]. The latent space is located between the encoder and the decoder of a DAE. These papers motivate a k-means friendly space such that the embeddings in the latent space are clustered depending on the intrinsic properties in the data. For example, if you were to train a DAE on a dataset with different classes, the latent space of the DAE should cluster into these depending on their common attributes. In contrast, we are motivated to ensure a k-means friendly feature embedding space in the penultimate layer of a DNN classifier to strengthen the overall safety argument.

To achieve this, we introduce an additional loss term (similar to [119]) to the standard cross-entropy loss function used to train classifiers (*cf.* Section 3.4.1)

$$\mathcal{L}_{\text{ce + }k\text{-means}}(\mathcal{D}_{\text{train}};\mathcal{P}) = \mathcal{L}_{\text{ce}}(\mathcal{D}_{\text{train}};\mathcal{P}) + \alpha \cdot \frac{1}{2}\sum_{m=1}^{M_{\text{train}}} \|\mathrm{feat}(\boldsymbol{X}^m) - \boldsymbol{H}\boldsymbol{\pi}^m\|_2^2\,, \tag{4.5}$$

with the tunable parameters $\mathcal{P}$ of the DNN architecture, a trade-off parameter $\alpha > 0$, a matrix of cluster centres $\boldsymbol{H} \in \mathbb{R}^{N_{(\Lambda-1)}\times\Omega}$, and a selection vector $\boldsymbol{\pi}^m \in \{0,1\}^\Omega$ with $\|\boldsymbol{\pi}^m\|_1 = 1$. Ultimately, $\boldsymbol{H}\boldsymbol{\pi}^m$ is the cluster centre closest to $\mathrm{feat}(\boldsymbol{X}^m)$. The

Algorithm 1 k-means Friendly Training

Inputs: DNN including parameters $\mathcal{P}$, number of epochs E, number of clusters Ω in feature embedding space
Initialisation: Initialise the cluster centre matrix $[\boldsymbol{H}]_{i,j} \sim \mathcal{U}(h_{\text{min}}, h_{\text{max}})$ and randomly assign each training sample to a cluster via $\boldsymbol{\pi}^m$.
for $\text{epoch} = 1, \ldots, E$ **do**
  $G \leftarrow \nabla_{\mathcal{P}} \mathcal{L}_{\text{ce + } k\text{-means}}(\mathcal{D}_{\text{train}}; \mathcal{P})$
  Update the parameters of the DNN to minimise (4.5) using the gradients G and an optimiser, e.g., Stochastic Gradient Descent (SGD) or ADAM
  for $m = 1, \ldots, M_{\text{train}}$ **do**
    $\boldsymbol{\pi}^m \leftarrow \arg\min_{\boldsymbol{e}_\omega \in \{\boldsymbol{e}_1, \ldots, \boldsymbol{e}_\Omega\}} \|\text{feat}(\boldsymbol{X}^m) - \boldsymbol{H}\boldsymbol{e}_\omega\|_2^2$
  end for
  for $\omega = 1, \ldots, \Omega$ **do**
    $[\boldsymbol{H}]_{\cdot,\omega} \leftarrow \sum_{m \in \mathcal{C}_\omega} \text{feat}(\boldsymbol{X}^m) / |\mathcal{C}_\omega|$ {Compute the mean of all feature embeddings corresponding to cluster ω as the new cluster centre.}
  end for
end for

cross-entropy loss and the feature embedding function are defined in (3.42) and (4.1), respectively. We assume there are $\Omega \in \mathbb{N}$ clusters in the k-means friendly feature embedding space. Since we aim to create a feature embedding space which encourages a clustering of the labelled data, we only consider the case where $\Omega = K$, where K is the number of classes. This parameter must be set before training. Since this loss function should be minimised, the additional term in (4.5) ensures that the extracted feature embeddings lie close, in terms of the Euclidean distance, to the selected cluster centre. These cluster centres are randomly initialised and each sample is assigned to a cluster centre.

To train a DNN with the loss function defined in (4.5), we alternate between optimizing the parameters of the DNN ($\mathcal{P}$), assigning the cluster selection vectors $\boldsymbol{\pi}$, and updating the cluster centres stored in $\boldsymbol{H}$, similar to [119]. The training algorithm is summarised in Algorithm 1.

To initialise the k-means friendly training algorithm, we first randomly sample the cluster centres to lie within the range of the activation function of the feature embedding space, e.g., for the $\tanh$ activation function $h_{\text{min}} = -1$ and $h_{\text{max}} = +1$. Next, each sample is randomly assigned to a cluster. For each epoch of training, we first update the DNN parameters $\mathcal{P}$ according to the k-means loss function (4.5) in

Lines 4 and 5. In Lines 6–8, we iterate over all samples in the training set and assign each to the closest cluster centre—the vector $\boldsymbol{e}_\omega \in \{0,1\}^\Omega$ has the value 1 only at the ωth index. Finally, we update the Ω cluster centres according to the extracted feature embeddings in Lines 9–11. We define the set of training sample indices which belong to a specific cluster ω as $\mathcal{C}_\omega = \{m : \boldsymbol{\pi}^m = \boldsymbol{e}_\omega, \forall m\}$.

4.5.1.1 Computational Complexity of k-means Friendly Training

The computational cost of calculating the gradient in Line 4 and updating the parameters in Line 5 is a constant multiple of evaluating the additional k-means loss term in (4.5) [121, Ch. 3]. The exact computational complexity of this step depends on the implemented DNN architecture (*cf.* Section 3.3). Furthermore, the computational complexity of finding the closest cluster centre (Lines 6–8) and updating the cluster centres (Lines 9–11) has a computational cost of $\mathcal{O}(\Omega N_{(\Lambda-1)} M_{\text{train}})$, where $N_{(\Lambda-1)}$ is the dimension of the feature embedding space and M_{train} is the number of training samples. The feature embeddings only have to be calculated once per epoch after Line 5 by performing a forward pass through the DNN. These can then be stored for the subsequent calculations. We assume the k-means friendly training is done offline—as most DNNs are trained offline before deployment—, so the additional computational complexity should not pose a problem. The online complexity is not affected by the proposed approach.

4.5.2 Classification Results

Now, we report the classification performance after training the DNNs with Algorithm 1. Similar to Section 4.4, we use the DNN architectures introduced in Section 4.2 and train them using the simulation setup discussed in Section 4.2.2. We use a trade-off parameter $\alpha = 0.1$ in (4.5) for the following architectures: CNN-I, CNN-I MC, and FC-DNN; for the other architectures, we use a trade-off parameter of $\alpha = 1$.

The classification performances of the k-means friendly trained DNNs are reported in Table 4.4. Again, we report the classification accuracies and the macro-averaged F_1 scores. If we compare the classification results in Table 4.4 with those in Table 4.2, we observe that the classification performances are decreased due to the k-means friendly training, e.g., the FC-DNN loses 2.82% and 3.06% in terms of classification accuracy and the F_1 score, respectively. For some of the DNN architectures, e.g., the LSTM-[10], CNN-II, or attention-based models, the k-means friendly training maintains or marginally increases the classification performance. By investigating the classification performances, we can observe that some DNN architectures are more robust to the

Architecture	k-means Friendly Space	
	Accuracy [%]	F_1 [%]
FC-DNN	$85.86(\pm 1.43)$	$85.58(\pm 1.58)$
CNN-I	$84.64(\pm 1.42)$	$84.43(\pm 1.46)$
CNN-II	$90.86(\pm 0.39)$	$90.80(\pm 0.40)$
CNN-III	$91.12(\pm 0.65)$	$91.10(\pm 0.66)$
CNN-I MC	$84.60(\pm 2.96)$	$84.35(\pm 3.09)$
CNN-II MC	$85.71(\pm 2.60)$	$85.29(\pm 3.03)$
CNN-III MC	$89.68(\pm 0.61)$	$89.63(\pm 0.61)$
LSTM-[10]	$91.05(\pm 2.77)$	$91.03(\pm 2.81)$
GRU-[12]	$88.96(\pm 2.20)$	$88.97(\pm 2.21)$
Gated Tr.-[95]	$90.38(\pm 1.75)$	$90.31(\pm 1.79)$

Table 4.4: Classification performance of the various DNN architectures in the k-means friendly feature embedding space. The results are averaged over the 10-folds with the 95% confidence interval in brackets.

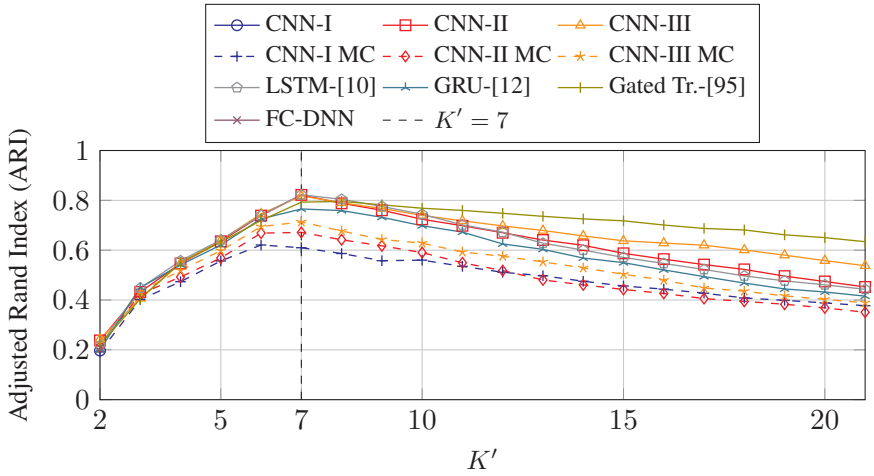
k-means friendly training than others. This could be attributed to the fact that some architectures already clustered well before k-means friendly training or that some k-folds diverged during training. The performances remain relatively similar between the different architectures, so that they seem equally well-suited for the TTLC task even after k-means friendly training.

4.5.3 Feature Validation in k-means Friendly Space

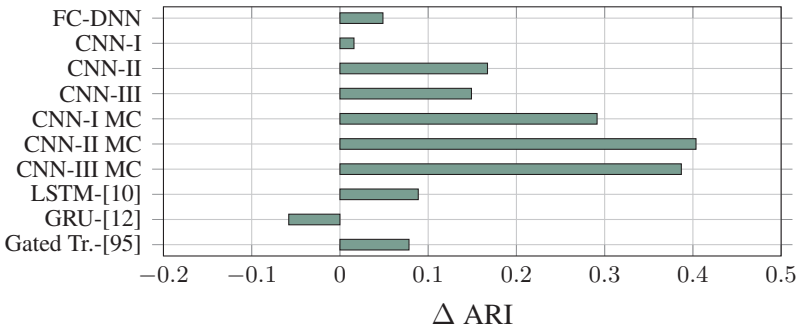
After observing the effect of using k-means friendly training, we want to investigate the extracted feature embeddings using the feature validation method. We saw that the k-means friendly feature embedding space slightly reduces the classification performance of the DNNs. However, in this section, we show that the k-means friendly training achieved our main goal: to create a feature embedding space which both qualitatively and quantitatively shows an improved clustering of the data.

4.5.3.1 Feature Clustering

The first step in the feature validation method is to quantify the clustering of the extracted feature embeddings. To this end, we use the ARI measure. In Fig. 4.4a, we plot the



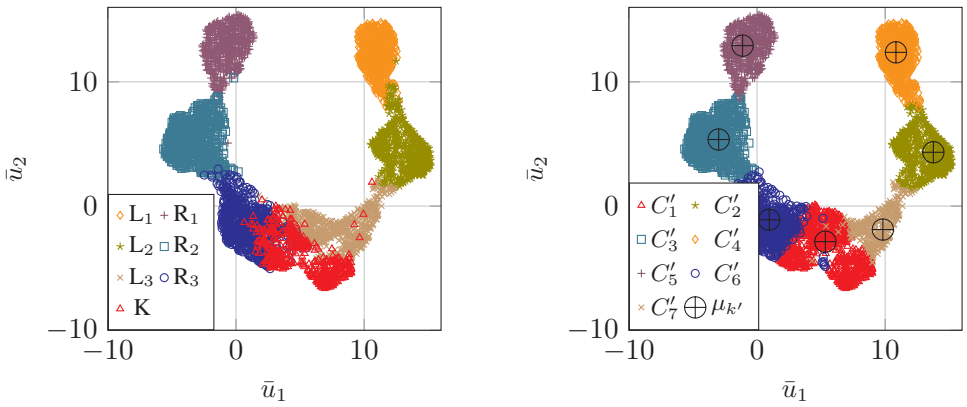
(a) Average ARI in the k-means friendly feature embedding space.



(b) Gain in ARI at $K' = 7$ between the k-means friendly feature embedding space and the standard feature embedding space ($\text{ARI}_{k\text{-means}}$ - $\text{ARI}_{\text{standard}}$). Compare these results with Fig. 4.2.

Figure 4.4: Subplot (a): Average ARI values vs. the number of clusters K' in the k-means feature embedding space. Subplot (b): Gain in ARI at $K' = 7$.

average ARI value in the k-means friendly feature embedding space. By encouraging a k-means friendly feature embedding space, almost all of the DNN architectures now have a maximum ARI value at $K' = 7$. This indicates that the k-means friendly training works as expected by clustering the feature embeddings according to the class they belong to since we set $\Omega = K$. In Fig. 4.4a, we also observe that the MC CNN architectures—which had poor clustering in the standard feature embedding space—now show the best clustering for K' around K clusters. This result highlights the



(a) The k-means friendly feature embedding space with true test data class labels.

(b) The k-means friendly feature embedding space with cluster labels.

Figure 4.5: The UMAP representation for the CNN-I architecture with different labels in the k-means friendly feature embedding space. The qualitative feature embeddings in these plots should be compared with the standard feature embedding space in Fig. 4.3.

potential gain in clustering which the k-means friendly training introduces. In Fig. 4.4b, we calculate the gain in ARI at $K' = 7$, i.e., $\Delta\text{ARI} = \text{ARI}_{k\text{-means}} - \text{ARI}_{\text{standard}}$. We observe that the average ARI is improved for all of the architectures besides the GRU-[12], where the average ARI slightly decreased. This could be due to the fact that the average ARI for the GRU-[12] is already high in the standard feature embedding space and that on certain k-folds the k-means friendly training diverged (see the discussion in Section 4.5.4). However, the average ARI value of the GRU-[12] is still relatively good compared with the other DNN architectures.

As clustering the feature embeddings is the first step of the feature validation method, the ARI results show the promise of using the k-means friendly training. As mentioned earlier, the k-means friendly training can be used in conjunction with any DNN architecture such that if an architecture is already validated using another method, we can strengthen the safety argument by additionally employing k-means friendly training. Next, we explore the qualitative results of feature validation by visualising the feature embedding space to see the improvement.

4.5.3.2 Feature Visualisation

In Fig. 4.5, we plot the UMAP embeddings of the CNN-I architecture in the k-means friendly feature embedding space. Note, since we have two different feature embedding spaces, we have two different UMAP functions, indicated by the axis $\{u_1, u_2\}$ for the standard space, and $\{\bar{u}_1, \bar{u}_2\}$ for the k-means friendly feature embedding space (compare Fig. 4.3 and Fig. 4.5).

We observe a direct qualitative improvement of the clustering in the feature embedding space when we compare Fig. 4.5a to the UMAP embeddings in Fig. 4.3a. Each cluster corresponding to a class label is more compactly clustered, e.g., looking at the cluster of samples with label R_1, we see in Fig. 4.5a that the samples are compactly clustered together, whereas in Fig. 4.3a these samples are more spread out; we could almost argue that there are two sub-clusters of the class R_1 in Fig. 4.3a. The same observation can be made when looking at the representations of the other driving manoeuvres. Moreover, the shapes of the true class labels and cluster labels are similar in Fig. 4.5a and Fig. 4.5b. This is reflected in the larger ARI since the ARI compares the cluster labels to the true class labels, so the more samples with the same labels, the larger the ARI value.

This result indicates that we achieved the goal of creating a k-means friendly feature embedding space where the feature embeddings of the different classes are clustered compactly together. Not only are the quantitative clustering results better than for the standard feature embedding space, but the qualitative UMAP results also show an improvement in clustering. These aspects help when interpreting the extracted feature embeddings for feature validation. Moreover, this directly improves the interpretability of the extracted feature embeddings. Thus, a safety argument using the feature validation method is improved upon due to the k-means friendly training introduced in this section.

4.5.4 Discussion on k-means Friendly Spaces

The k-means friendly training algorithm actively creates a feature embedding space showing both quantitatively and qualitatively better clustering of the extracted feature embeddings. By improving the clustering of the feature embeddings, we contend that a stronger safety argument can be attached to the DNN trained in this manner, i.e., the feature embeddings of the safety relevant classes are more compactly clustered leading to better interpretability. Furthermore, if an already validated DNN previously did not show good clustering, e.g., the MC CNN architectures, this can be addressed using the proposed method.

Architecture	Early Stopping Epoch	
	Standard Space	k-means Friendly Space
FC-DNN	$72(\pm 7)$	$22(\pm 0)$
CNN-I	$111(\pm 4)$	$22(\pm 0)$
CNN-II	$90(\pm 15)$	$226(\pm 25)$
CNN-III	$126(\pm 16)$	$342(\pm 28)$
CNN-I MC	$279(\pm 28)$	$105(\pm 83)$
CNN-II MC	$290(\pm 24)$	$283(\pm 98)$
CNN-III MC	$290(\pm 48)$	$698(\pm 64)$
LSTM-[10]	$130(\pm 6)$	$241(\pm 77)$
GRU-[12]	$152(\pm 21)$	$97(\pm 67)$
Gated Tr.-[95]	$74(\pm 9)$	$175(\pm 55)$

Table 4.5: Average early stopping epoch during the training of the DNNs. The results are averaged over the 10-folds with the 95% confidence interval in brackets.

These benefits also increase the interpretability of the DNN architectures as we not only rely on the classification performances to compare them, but can evaluate them on their feature embedding clustering as well. This increase in interpretability, however, comes at a price in classification performance—this can be interpreted as the cost of interpretability, see, e.g., [22; 122]. In a practical application, the engineer must make a trade-off between increasing the interpretability of their ML-based method and the performance of that method. We argue that the k-means friendly training in combination with the feature validation method, can give the engineer a stronger safety argument for the validation of the chosen ML-based method. Moreover, in the following section, we propose a method to regain some of the lost classification performance.

On the other hand, the proposed k-means friendly training algorithm is sensitive to the initialisation of the DNN parameters and the cluster centre matrix $\boldsymbol{H}$ (*cf.* Algorithm 1). Moreover, the hyper-parameter choices, e.g., initial step-size or model parameters, also affect the k-means friendly training. This can be seen in the average early stopping epoch during training of the various DNNs, depicted in Table 4.5. Compared to training the DNN using the standard cross-entropy loss, the k-means friendly training, in general, stops earlier (since we employ an early stopping criterium when training the DNNs). For the architectures which stop quickly, the valida-

Algorithm 2 Distance-based Rejection Rule

Inputs: Input datum $\boldsymbol{X}$, classification function f_{DNN}
Parameters: Cluster centres $\{\boldsymbol{\mu}_c\}_{c=1}^{K'}$, distance measure d, labelling function λ, the distance r.
$y_{\text{DNN}} \leftarrow f_{\text{DNN}}(\boldsymbol{X})$
$k' \leftarrow \arg\min_{c=1,\dots,K'} d(\boldsymbol{\mu}_c, \text{feat}(\boldsymbol{X}))$
$y_{\text{cluster}} \leftarrow \lambda(k')$
if $y_{\text{cluster}} = y_{\text{DNN}}$ **and** $d(\text{feat}(\boldsymbol{X}), \boldsymbol{\mu}_{k'}) \leq r$ **then**
    Accept y_{DNN}
else
    Reject y_{DNN}
end if

tion loss increases indicating that the k-means friendly training was diverging. For the architectures which trained longer during the k-means friendly training, e.g., CNN-II, the classification performance was better than for the standard training. Moreover, the early stopping epoch varies between 10-folds more during k-means friendly training, further indicating that a good initialisation is important. Therefore, k-means friendly training should be combined with a fine-tuning of the hyper-parameters.

4.6 Classification Rejection Rules

In the preceding sections, we presented the feature validation method, where we investigate the ability of various DNN architectures to extract feature embeddings which cluster according to their class labels. However, we observe a trend that the classification performance is negatively affected by the k-means friendly training. As a counter measure, we propose a method to take advantage of the clustering of the feature embeddings according to their classes by rejecting classification outputs which are not consistent with the cluster they belong to. This allows us to regain the classification performance lost through k-means friendly training.

We introduce an algorithm to reject spurious DNN classification outputs by computing the distance of the feature embedding of a given input to the closest cluster centre. We check whether the corresponding cluster label matches the classification label and whether the feature embedding lies close to the cluster centre. If the classification label differs from the label of the closest cluster centre, we reject the classification. The algorithm is summarised in Algorithm 2.

Algorithm 2 depends on the following parameters: pre-defined cluster centres

$\{\boldsymbol{\mu}_c\}_{c=1}^{K'}$ in the feature embedding space (e.g., those obtained from k-means clustering); a distance measure $d : \mathbb{R}^{N_{(\Lambda-1)}} \times \mathbb{R}^{N_{(\Lambda-1)}} \rightarrow \mathbb{R}$ to compare the feature embedding with the cluster centres; and, a label mapping function $\lambda : \{1, \ldots, K'\} \rightarrow \mathbb{Y}$, mapping the cluster centre index to a class label. Note, since there can be a different number of cluster centres than classes, $K' \neq K$, the labelling function is necessary to map the cluster index to an output class label. In Line 3, we classify the input sample using the DNN. In Line 4, we find the cluster centre closest (in terms of the pre-defined distance measure d) to the feature embedding $\mathrm{feat}(\boldsymbol{X})$ of the sample. Then, a cluster class label is calculated using the label mapping function in Line 5. Finally, in Lines 6–10, the output classification is accepted if two conditions are met: (i) the cluster class label and the DNN class label are the same; and (ii) the feature embedding is within a distance r of the chosen cluster centre. Otherwise, the classification output is rejected.

Intuitively, we can understand Algorithm 2 as follows: As we observed in the previous sections, the feature embeddings in the penultimate layer of a DNN are generally clustered according to their class labels. Therefore, we can use the training dataset to calculate the cluster centres. We can subsequently compare the distance of the feature embedding of an input sample to the closest cluster centre. If the label at the output of the DNN matches the label of that cluster centre and the feature embedding lies within the distance r of that cluster centre, then we accept the DNN classification; otherwise, we reject it.

In the following, we introduce two possible methods to obtain the cluster centres and two possible distance measures. We refer to the classification rejection methods according to the distance measures they are based on.

4.6.1 Euclidean-based Rejection

In Fig. 4.2 and Fig. 4.4a, we observe that most DNN architectures show a high ARI for the number of clusters equal to the number of classes ($K' = K$). Since the k-means clustering used to calculate the cluster labels is based on the Euclidean distance metric, the first rejection rule is based on this metric, too.

To this end, we first use the cluster centres $\{\boldsymbol{\mu}_c\}_{c=1}^{K'}$ we obtain from clustering the training data. As a distance measure, we use the Euclidean distance from the cluster centres to the feature embedding, i.e., $d_E(\boldsymbol{\mu}_c, \boldsymbol{a}) = \|\boldsymbol{a} - \boldsymbol{\mu}_c\|_2^2$. Each cluster centre is labelled with the majority class label in that cluster.

4.6.2 Mahalanobis-based Rejection

The authors of [123] postulate that employing the softmax activation function at the output of a DNN implies that the feature embeddings in the penultimate layer are fit to a class-conditional Gaussian distribution. If we assume a multivariate Gaussian distribution for the class-conditional probabilities in the feature embedding space, i.e., we assume the feature embeddings are distributed according to $\mathcal{N}(\boldsymbol{\mu}_c, \boldsymbol{\Sigma})$ with a mean $\boldsymbol{\mu}_c$ and tied covariance matrix $\boldsymbol{\Sigma}$, we see that the Linear Discriminant Analysis (LDA) classifier has a similar structure to the softmax output of a DNN, see, e.g., [124, Ch. 4.2]. We can write the *posterior* distribution of the LDA as

$$p(y = c|x) = \frac{\exp\left(\boldsymbol{\mu}_c^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}x - \frac{1}{2}\boldsymbol{\mu}_c^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_c + \log(\xi_c)\right)}{\sum_{c'}\exp\left(\boldsymbol{\mu}_{c'}^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}x - \frac{1}{2}\boldsymbol{\mu}_{c'}^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_{c'} + \log(\xi_{c'})\right)}, \tag{4.6}$$

with the tied covariance matrix $\boldsymbol{\Sigma}$, the class mean $\boldsymbol{\mu}_c$, and the class *prior* probability ξ_c. If we compare (4.6) with the softmax (3.41), we see that the *posterior* estimate at the output of the DNN is similar to the LDA if we set the weight vector and bias term equal to

$$\boldsymbol{w}_c^{(\Lambda),\mathrm{T}} = \boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_c, \tag{4.7}$$

$$b_c^{(\Lambda)} = -\frac{1}{2}\boldsymbol{\mu}_c^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_c + \log(\xi_c), \tag{4.8}$$

for $c = 1, \ldots, K$.

Since the LDA classifier uses the Mahalanobis distance to optimally classify data under the aforementioned assumptions, we are motivated to use this as the distance measure in Algorithm 2. The Mahalanobis distance is calculated as

$$d_M(\boldsymbol{\mu}_c, \boldsymbol{a}; \boldsymbol{\Sigma}) = (\boldsymbol{a} - \boldsymbol{\mu}_c)^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\boldsymbol{a} - \boldsymbol{\mu}_c), \tag{4.9}$$

with a class-conditional mean vector $\boldsymbol{\mu}_c$ and the tied covariance matrix $\boldsymbol{\Sigma}$. We estimate the class-conditional sample means and sample covariance matrix on the training dataset as

$$\boldsymbol{\mu}_c = \frac{1}{|\mathcal{M}_{\text{train},c}|}\sum_{m\in\mathcal{M}_{\text{train},c}} \text{feat}(\boldsymbol{X}^m), \tag{4.10}$$

$$\boldsymbol{\Sigma} = \frac{1}{M_{\text{train}}}\sum_{m=1}^{M_{\text{train}}}(\text{feat}(\boldsymbol{X}^m) - \boldsymbol{\mu})(\text{feat}(\boldsymbol{X}^m) - \boldsymbol{\mu})^{\mathrm{T}}, \tag{4.11}$$

with the set of indices $\mathcal{M}_{\text{train},c} = \{m : y^m = c, \forall y^m \in \mathcal{D}_{\text{train}}\}$. The tied covariance matrix is calculated using the mean embedding of all of the training samples, i.e., $\boldsymbol{\mu} = 1/M_{\text{train}}\sum_{m=1}^{M_{\text{train}}}\text{feat}(\boldsymbol{X}^m)$.

In the end, we use the following parameters for the distance-based rejection rule in Algorithm 2: the sample class means are used as cluster centres, i.e., $\{\boldsymbol{\mu}_c\}_{c=1}^{K}$ from (4.10); the Mahalanobis distance as defined in (4.9) is used as a distance measure; and, the labelling function simply takes the label of $\boldsymbol{\mu}_c$ as the cluster label.

4.6.3 Classification Rejection Rules: Results ($r = \infty$)

Using the two rejection rules defined in the previous sections, we can perform inference on the test dataset again and reject spurious classifications. Note, the DNNs do not need to be retrained to apply the rejection rules—they are merely applied during inference. In this section, we take $r = \infty$ around the cluster centres, i.e., we only check whether the cluster label is equal to the classification label in Line 6 of Algorithm 2.

Table 4.6 summarises the classification performances of the various DNN architectures using the previously introduced rejection rules. The column "Rejections [%]" shows the percentage of test data samples which were rejected. As a reference, we could imagine uniform randomly rejecting test data samples. However, if the data are independent,[6] we would expect this neither to affect the classification accuracy nor to affect the F_1 score, e.g., if we have a classification accuracy of 90% and then randomly reject $p\%$ of the samples, we would still have a classification accuracy of 90% on the remaining $(100 - p)\%$ of the samples. Therefore, any improvement in performance after applying the rejection rules indicates that we are rejecting incorrect classifications.

In general, we can regain the classification performance lost using the k-means friendly feature embedding space by applying either rejection rule. In Fig. 4.6, we observe the difference in classification performance in the k-means friendly feature embedding space before and after applying the rejection rules, i.e., comparing the difference between Table 4.4 with the respective columns in Table 4.6. The average classification accuracies of all the DNN architectures increase after applying the rejection rules. However, for certain architectures, the Euclidean rejection rule can lead to a reduction in the average F_1 score. This can occur when the clusters are not completely pure, i.e., the samples from some classes are split between multiple clusters leading to too many samples being rejected by Algorithm 2.

These performance gains, however, come at the price of not classifying all test data. Looking at Table 4.6, we see in the standard feature embedding space, the average percentage of rejected samples is as high as 52.97% for the Euclidean rejection rule.

[6] The training and test data are generally assumed to be independent and identically distributed (iid) in most ML applications.

Architecture	Rejection Method	Standard Space			k-means Friendly Space		
		Accuracy [%]	F_1 [%]	Rejections [%]	Accuracy [%]	F_1 [%]	Rejections [%]
FC-DNN	Euclidean	$92.73(\pm0.44)$	$90.80(\pm0.44)$	18.75	$91.82(\pm0.50)$	$90.81(\pm0.45)$	13.20
	Mahalanobis	$91.72(\pm0.29)$	$91.41(\pm0.33)$	6.50	$90.94(\pm0.28)$	$90.23(\pm0.32)$	9.10
CNN-I	Euclidean	$93.95(\pm0.32)$	$92.21(\pm0.37)$	20.65	$89.09(\pm0.91)$	$87.62(\pm1.07)$	13.35
	Mahalanobis	$93.31(\pm0.18)$	$93.08(\pm0.17)$	6.27	$89.62(\pm0.33)$	$88.86(\pm0.28)$	9.08
CNN-II	Euclidean	$91.18(\pm0.76)$	$87.14(\pm2.69)$	19.26	$93.15(\pm0.27)$	$92.95(\pm0.24)$	4.05
	Mahalanobis	$89.82(\pm0.31)$	$89.37(\pm0.38)$	7.03	$93.28(\pm0.17)$	$93.03(\pm0.16)$	5.46
CNN-III	Euclidean	$93.33(\pm0.50)$	$89.67(\pm2.84)$	18.40	$92.52(\pm0.29)$	$92.25(\pm0.30)$	5.90
	Mahalanobis	$92.44(\pm0.54)$	$92.30(\pm0.58)$	2.67	$92.72(\pm0.44)$	$92.58(\pm0.48)$	3.29
CNN-I MC	Euclidean	$94.41(\pm0.89)$	$71.03(\pm4.38)$	49.55	$88.79(\pm2.37)$	$79.59(\pm3.46)$	19.26
	Mahalanobis	$93.86(\pm0.28)$	$93.54(\pm0.32)$	9.56	$89.71(\pm1.59)$	$88.73(\pm1.86)$	10.52
CNN-II MC	Euclidean	$93.28(\pm0.76)$	$67.34(\pm3.00)$	52.97	$88.87(\pm2.77)$	$86.16(\pm4.14)$	12.12
	Mahalanobis	$92.88(\pm0.30)$	$92.48(\pm0.30)$	9.68	$90.70(\pm1.02)$	$89.56(\pm1.71)$	10.10
CNN-III MC	Euclidean	$92.23(\pm1.92)$	$64.22(\pm7.61)$	47.90	$92.42(\pm1.44)$	$91.90(\pm1.64)$	10.13
	Mahalanobis	$91.27(\pm0.64)$	$90.74(\pm0.71)$	9.09	$91.79(\pm0.52)$	$91.49(\pm0.55)$	6.58
LSTM-[10]	Euclidean	$93.44(\pm0.58)$	$88.16(\pm3.42)$	11.14	$92.78(\pm1.24)$	$92.57(\pm1.52)$	3.55
	Mahalanobis	$93.87(\pm0.19)$	$93.68(\pm0.20)$	4.66	$93.82(\pm0.66)$	$93.62(\pm0.88)$	4.64
GRU-[12]	Euclidean	$93.96(\pm0.22)$	$93.88(\pm0.22)$	3.03	$91.68(\pm1.34)$	$91.13(\pm1.58)$	7.63
	Mahalanobis	$93.82(\pm0.21)$	$93.75(\pm0.23)$	3.78	$92.24(\pm1.13)$	$91.99(\pm1.21)$	5.59
Gated Tr.-[95]	Euclidean	$93.42(\pm0.46)$	$92.26(\pm0.45)$	14.05	$92.41(\pm1.64)$	$86.16(\pm8.50)$	11.97
	Mahalanobis	$92.83(\pm0.41)$	$92.64(\pm0.43)$	5.10	$92.78(\pm0.52)$	$92.58(\pm0.62)$	3.99

Table 4.6: Classification performance of the various DNN architectures for both the standard feature embedding space and the k-means friendly feature embedding space after applying the rejection rules with $r = \infty$. The results are averaged over the 10-folds with the 95% confidence interval in brackets.

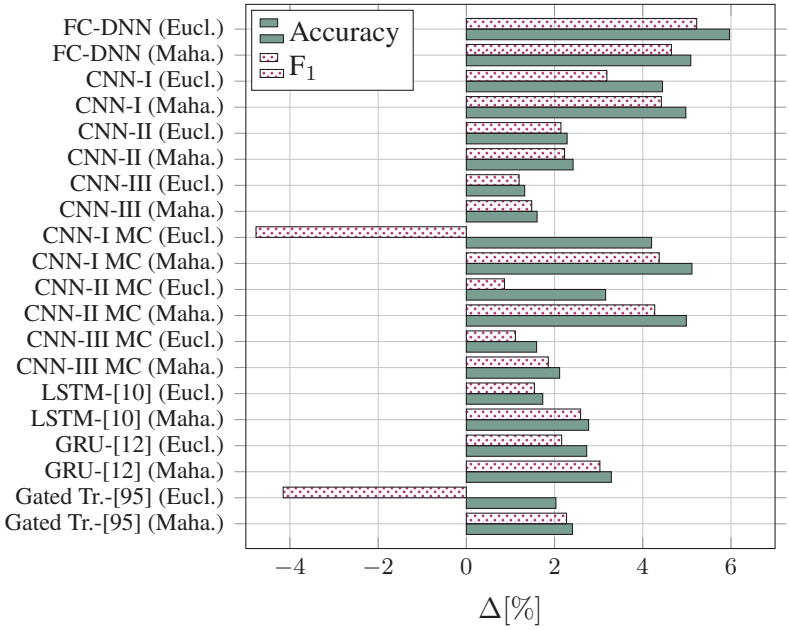


Figure 4.6: Classification performance change in the k-means friendly embedding space for each DNN architecture after using either the Euclidean rejection rule (Eucl.) or the Mahalanobis rejection rule (Maha.).

On the other hand, the k-means friendly training encourages better clustering of the feature embeddings and reduces the average number of samples which are rejected (especially for the Euclidean rejection rule). Even in the standard feature embedding space, the rejection rules can improve the accuracy of the classifier (especially when applying the Mahalanobis rejection rule).

4.6.4 Classification Rejection Rules: Results (variable r)

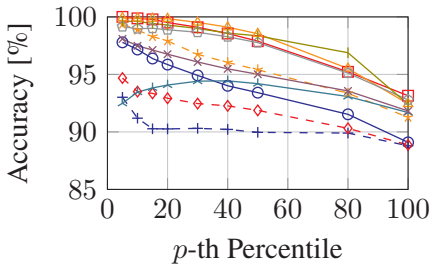
To further investigate the classification rejection rules, we can vary the distance around the cluster centres to influence how many samples are rejected. As we saw, with $r = \infty$ the average number of rejected samples varies depending on the architecture and whether a k-means friendly space was created (*cf.* Table 4.6). Now, we analyse the classification rejection rules by varying the distance r. Since each k-fold of each architecture creates a different feature embedding space where the distances between embedded samples can vary on orders of magnitude (this also depends on the distance measure used), we need to define what is *close* and what is *far* in the feature embedding space.

To define suitable distances we can use the feature embeddings of the training dataset. First we map all of the training data into the feature embedding space. Then, we calculate all of the distances of these feature embeddings to the closest cluster centre. With these distances, we can define r in two ways: (i) we take the p-th percentile distance as r; or (ii) we take a distance as a fixed percentage of the maximum distance (r_{max}) in the feature embedding space. Choosing r to be equal to the p-th percentile means that p percent of all of the training data distances are smaller than r; alternatively, r is equal to a percentage of r_{max}. Using option (i), we are able to compare the classification rejection rules where the percentage of rejections remains constant between the architectures. When employing option (ii), we are able to better investigate how compactly clustered the feature embeddings are for the different architectures and feature embedding spaces (standard vs. k-means friendly); we are able to investigate the classification rejection rules in both cases as well.

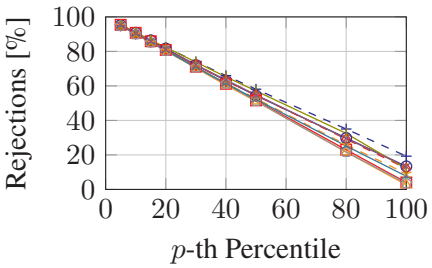
First, we investigate the Euclidean- and the Mahalanobis-based classification rejection rules in the k-means friendly feature embedding space.[7] In the top row of Fig. 4.7, we see the effect of the Euclidean-based classification rejection rule using the p-th percentile distance as r. We vary the percentile between $p \in [5, 100]$. We see in Fig. 4.7b that the average number of rejections is almost the same for each architecture at each percentile distance. This is as expected, since r is directly taken from the percentiles of the distances of the training dataset. However, looking at Fig. 4.7a, we observe that the average accuracy for each of the architectures differs. When we reject almost all of the samples (for $p = 5$), some of the architectures are able to achieve an average accuracy of almost 100%. Most of the architectures show a monotonic decrease in the average accuracy as the distance is increased. The average accuracy achieved at $p = 100$ is the same as when we use $r = \infty$ (*cf.* Table 4.6). However, we see that the GRU-[12] architecture's accuracy first increases and then decreases. This implies that the rejected samples are not misclassifications for small r values. Further, this could imply that this classification rejection rule does not work well with certain feature embedding spaces.

Similar conclusions can be drawn when using the Mahalanobis-based classification rejection rule, as depicted in the bottom row of Fig. 4.7. We notice that the Mahalanobis-based classification rejection rule is unable to achieve 100% accuracy despite rejecting almost all of the samples. Furthermore, additional architectures, e.g., CNN-I or CNN-II, show a non-monotonically decreasing performance as the distance is increased. This would imply that the Mahalanobis-based classification rejection rule is more

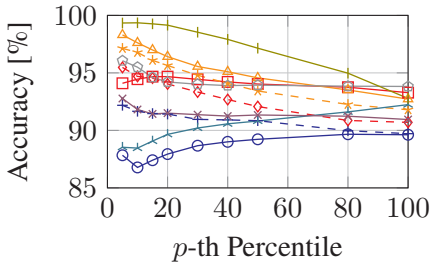
[7] An analysis of these classification rejection rules in the standard feature embedding space can be found in Appendix A.2.



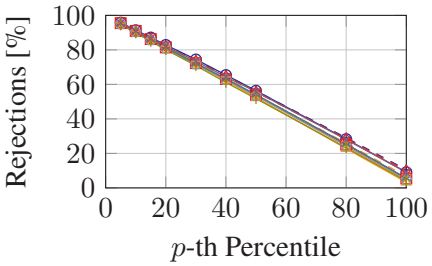
(a) Accuracy: Euclidean-based rejection.



(b) Rejections: Euclidean-based rejection.



(c) Accuracy: Mahalanobis-based rejection.



(d) Rejections: Mahalanobis-based rejection.

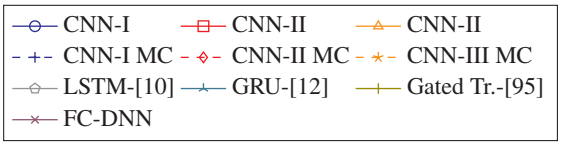
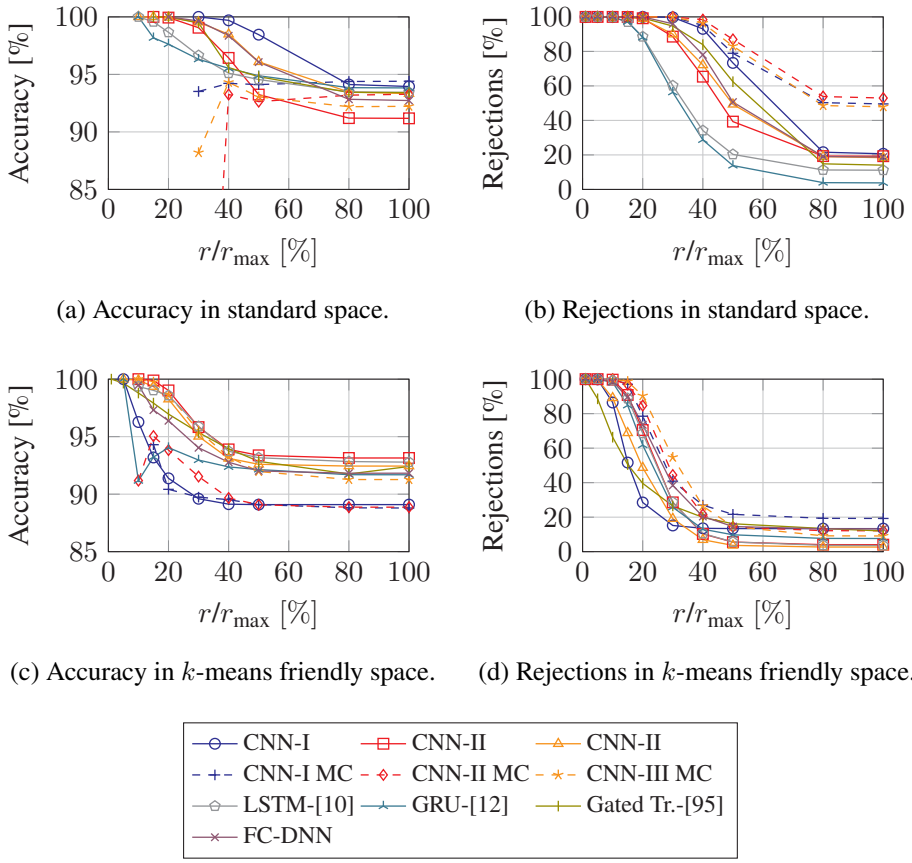


Figure 4.7: Classification rejection in the k-means friendly space using the p-th percentile distance as the distance r. The top row of plots (Fig. 4.7a and Fig. 4.7b) uses the Euclidean-based classification rejection rule; the bottom row of plots (Fig. 4.7c and Fig. 4.7d) uses the Mahalanobis-based classification rejection rule.

sensitive to the clustering of the samples; the tied-covariance matrix also affects the distance calculations which influences which samples are rejected. We also observe that the decrease in average accuracy is smaller for most architectures compared with the Euclidean-based rejection rule. Again, the average accuracies at $p = 100$ are the same as when we use $r = \infty$.

In Fig. 4.8, we plot the average accuracy and the average number of rejections against the distance which is set relative to the maximum distance (r_{max}). All of these plots are generated using the Euclidean-based classification rejection rule. Compared to the number of rejection plots in Fig. 4.7, where the different architectures show a



(a) Accuracy in standard space.

(b) Rejections in standard space.

(c) Accuracy in k-means friendly space.

(d) Rejections in k-means friendly space.

Figure 4.8: Classification rejection using Euclidean rejection rule and percentage of max distance as the distance r. The top row of plots (Fig. 4.8a and Fig. 4.8b) is in the standard feature embedding space; the bottom row of plots (Fig. 4.8c and Fig. 4.8d) is in the k-means friendly feature embedding space.

similar number of classification rejections for each p-th percentile, we now observe in Fig. 4.8b and Fig. 4.8d that there are differences between the architectures. Moreover, we see that there is a large difference between the standard feature embedding space and the k-means friendly embedding space. In the former, the architectures reject all samples until $r/r_{\max} = 15\%$; some architectures, e.g., the MC CNN architectures, sill reject all samples until almost $r/r_{\max} = 40\%$. In the k-means friendly space, the number of rejections decreases much faster. The architectures show an improvement in accuracy or number of rejection after $r/r_{\max} = 50\%$ in the k-means friendly space. In Fig. 4.8c, we see a similar performance between the architectures in the

k-means friendly space. The GRU-[12] architecture shows a jump in the accuracy for $r/r_{\text{max}} = 10$ and 15%. In Fig. 4.8a, the MC CNNs show a bad classification performance for a small distance; this indicates that the feature embeddings are poorly clustered in the standard space (as we observed in Section 4.4), and thus, the Euclidean-based rejection rule rejects too many correct classifications.

Overall, we see that by tuning the distance r around the cluster centres, we are able to trade off the classification accuracy with the number of rejected samples. In a safety-critical application this trade-off can be made by the engineer, where the smaller a distance they take, the more confident they can be with the classification outputs of the chosen DNN architecture. We observe that some architectures produce a feature embedding space more suitable for classification rejections than others. Moreover, the k-means friendly training leads to feature embedding spaces which tend to harmonise with the rejection rules.

4.7 Feature Embedding based Safety Argument: Summary

In this chapter, we introduce a feature validation method allowing an engineer to not only rely on the classification performance of a DNN architecture to choose a model for a safety-critical function. We start with a passive analysis of various DNN architectures, and introduce a training method which can help encourage validatable features. We can use k-means friendly training in combination with other validation methods to strengthen the overall safety argument. Additionally, we introduce a method to reject spurious classification outputs using different distance measures. This method can be used to improve the performance of ML-based classifiers, or to increase the confidence in the classifications.

We can use the methods from this chapter to argue for safety during the validation of ML-based algorithms implemented in safety-critical HAD functions by considering the feature embeddings of various DNN architectures. In the end, an engineer can choose the DNN architecture showing the most meaningful feature embeddings with the best clustering (in terms of a clustering quality measure, e.g., the ARI). Additionally, they can encourage validatable features of an DNN architecture using k-means friendly training.

5 Validation Safety Arguments based on Dataset Distributions

In this chapter, we discuss the challenge of using Machine Learning (ML)-based driving functions in the presence of similar datasets with different distributions—so called distributional shifts (*cf.* Section 5.2). We find that there are distributional shifts between two publicly available highway driving datasets which can significantly affect the classification performance of an ML-based Highly Automated Driving (HAD) function (*cf.* Section 5.3). One possibility to overcome these distributional shifts is to transfer the knowledge from one distribution to the other. We introduce a method to trade off the forgetting of the source domain (the original training dataset) and learning of the target domain (the other dataset) when fine-tuning an ML-based HAD function in the presence of distributional shifts (*cf.* Section 5.4). We see that this method allows an engineer to influence the desired performance of the ML-based driving function depending on the situation. The results presented in this chapter are based on those found in [16; 125; 126; 127].

5.1 Dataset Distributions Motivation

In this section, we motivate developing a validation safety argument based on an analysis of the possible discrepancies between dataset distributions. Since the Society of Automotive Engineers (SAE) introduced the various levels of driving automation in 2014 [3], the Autonomous Vehicle (AV) community has accepted that every driving automation system, or function thereof, must be engineered to work within a pre-defined Operational Design Domain (ODD). The ODD of a given driving function can be specified to be, e.g., the geographical location, the time-of-day, or specific environmental circumstances [3]. For example, a HAD function could be engineered to work within a geographical zone like an airport or a city during the day time when the street is well lit. However, within a given ODD, there can still be differences in the distributions of datasets used to train ML-based HAD functions (as we will see in this chapter). Moreover, since ML-based HAD functions lack explicit function

specification, it is challenging to ensure that they will always work as expected within the given ODD. This is closely related to the reasons why ML-based HAD functions are difficult to validate using the existing methods and processes defined in the ISO/PAS standards (*cf.* Section 2.4).

Therefore, we believe that only relying on the pre-defined ODD which the HAD function should operate in is insufficient to argue for its safety when ML algorithms are used. At most, we can collect data within an ODD and train an ML algorithm on them. However, as we see in this chapter, this can lead to HAD functions which do not generalise across distributions. For example, an ODD might describe a highway scenario with clear lane markings and driving during the day-time. Two datasets, e.g., the highD dataset and the Next Generation SIMulation (NGSIM) dataset, can fulfil this ODD description but still have different distributions of their attributes. For example, the distribution of longitudinal velocities or the distribution of the distances to other vehicles can differ. Since ML algorithms rely on the distribution of the datasets, a difference in the distribution of the data will lead to ML-based functions which may not generalise as expected (or as required). A mismatch between distributions of the data is known as a distributional shift. Examples of the different types of distributional shifts which can occur in highway driving datasets will be introduced in Section 5.2.

Since no training dataset will fully cover all possible inputs, i.e., there will be a difference in the distributions of most datasets, there are two general approaches to tackle this challenge: (i) detecting when samples come from a different distribution than the one the ML model was trained on; or (ii) transferring the knowledge from one distribution or task to another. Approach (i) is useful since ML algorithms can struggle to generalise across distributions, so we want to know when we are able to trust the ML-based HAD functions especially in safety-critical applications and when we should exercise caution. The other approach is useful to leverage large datasets to (hopefully) learn generalisable ML functions. It has been (empirically) shown that larger datasets and larger models learn more generalisable features [128]. These two approaches are usually referred to as out-of-distribution detection and transfer learning, respectively.

In terms of out-of-distribution detection, we assume that the data used to train the ML algorithm is in-distribution and all other test data are out-of-distribution. There are two general classes of out-of-distribution data: those with a covariate shift, i.e., a shift in the inputs to the ML model, or those with a prior probability shift, i.e., a shift in the labels of the data. We discuss these types of distributional shifts in highway driving data in Section 5.2. An overview of different methods to detect out-of-distribution data can be found in [129]. Generally speaking, detecting out-of-distribution data is

closely related to the problem of quantifying the confidence of an ML algorithm. If the model is input a sample from a different distribution, it should be able to output a low confidence or out-right reject that sample. For example, if the longitudinal velocity of the tracked vehicle lies outside the range of the training distribution, we would reduce the confidence of the ML algorithm's output.

On the other hand, we might have a large, representative dataset of highway driving data from one country which we want to transfer to another country. For example, we might not be legally allowed to export data collected in a specific country, however, we could export the ML models trained on these data. When we want to transfer the knowledge learned on the training distribution (the source domain) to another distribution (the target domain), we refer to transfer learning. This can be done to take advantage of large pre-trained ML models by fine-tuning the (output) layers to the specific task at hand. The assumption is that the features extracted in the initial layers of an ML model learn general concepts, e.g., in image processing, these could be edge features or simple shapes. Different transfer learning tasks are categorised depending on the problem setting, i.e., whether the domains are similar or if we have labelled data in the source and/or the target domain, see, e.g., [130; 131]. Although transfer learning methods consider cases where there is a distributional shift, e.g., when the source and the target domains are assumed to be different, they do not usually consider the implications of training under these conditions. It might be the case that we have a distributional shift between two datasets, but we want to ensure that we do not forget the source domain. This is the challenge of avoiding (catastrophic) forgetting which is related to the topic of continual learning, see, e.g., [132].

The challenges of successfully dealing with out-of-distribution samples or transferring knowledge between distributions are especially important when ML algorithms are deployed in the real-world. In this chapter, we shed light on the possible distributional shifts which can occur in highway driving data, we analyse how these can affect trained ML models, and we introduce a method to transfer knowledge between datasets keeping in mind that there is a distributional shift between them.

5.2 Distributional Shifts in Highway Driving Datasets

A *distributional shift* or *dataset shift* describes the phenomenon when the distribution of the data which were used to train the ML algorithms does not match that of the test data [133]. Moreover, in general, the data are assumed to be independent and identically distributed (iid), which generally does not hold in the real-world.

In supervised ML tasks, we train the algorithms on a training dataset $\mathcal{D} =$

$\{(\boldsymbol{x}^m, y^m)\}_{m=1}^M$, where the inputs $\boldsymbol{x} \in \mathbb{X}$ and labels $y \in \mathbb{Y}$ are iid samples of an unknown joint distribution $p(\boldsymbol{x}, y)$. Then, we test on a test dataset. In general, a distributional shift describes the case where $p_A(\boldsymbol{x}, y) \neq p_B(\boldsymbol{x}, y)$, where p_A and p_B are the unknown joint distributions of two datasets $\mathcal{D}_A$ and $\mathcal{D}_B$, e.g., A and B could be the training and the test datasets.

In the following, we introduce the types of distributional shifts and how they can arise. We aim to give an intuition how these could arise in highway driving datasets, and how they might be affected by the ODD in which the ML algorithm should function.

5.2.1 Covariate Shift

This form of shift occurs when the covariate[1] distribution $p(\boldsymbol{x})$, e.g., the distribution of the velocities, differs between datasets. However, the posterior distribution $p(y|\boldsymbol{x})$ remains the same [134]. This implies that the samples in two datasets have the same label distribution but the covariate distribution is different.

In highway driving data, this can be observed when recording data at different locations, or in countries where the driving rules and drivers' behaviour might differ. For example, the average driving velocity on a German highway will be higher than on a US highway; or the traffic density on different highway sections might differ which would lead to different distributions of the relative distances between vehicles. A covariate shift can lead the ML algorithm to fit the training data well, but the model can be misspecified on a different dataset. For example, training an ML algorithm on data collected in Germany can lead to it not generalising to US highways because the velocities are lower and the traffic density is higher in the US. One can use importance sampling to reweigh the samples' contribution to the estimation error and compensate for a covariate shift [134]. A covariate shift between two datasets might not violate the ODD definition unless it is specific enough to eliminate a mismatch between recorded attributes. For example, if the ODD was only highways during the day-time, two datasets can have a covariate shift but still lie within the same ODD.

5.2.2 Prior Probability Shift

A prior probability shift, also known as a semantic or label shift [129], occurs when the prior distribution $p(y)$, e.g., the probability of a lane change, differs but the likelihood, $p(\boldsymbol{x}|y)$, remains the same [133]. This can also occur when the a specific class label lies in the support of one distribution but not in another, e.g., $p_A(y) > 0$ but $p_B(y) = 0$

[1]Note, the covariates are the input attributes or input features of the ML algorithm.

for some realisation(s) of $y \in \mathbb{Y}$. Moreover, since the prior probability is usually estimated on one dataset, this estimate might not be valid for other datasets.

For example, when extracting driving manoeuvres from a highway dataset, one dataset might have more samples with the label *lane keeping* than with the label *lane change* compared to another dataset, indicating a prior probability shift. Moreover, the label *emergency stop* could exist in one highway driving dataset and not exist in another. Furthermore, a prior probability shift can occur in highway driving datasets as implicit driving rules are baked into them, e.g., on German highways, vehicles should not overtake on the right-hand side, making the label for this manoeuvre very unlikely in a dataset recorded in Germany. However, in a country with left-hand traffic this is highly likely. It is possible to use *a priori* knowledge of the problem at hand to estimate which prior probabilities are possibly correct. Since a supervised ML algorithm tries to predict the label y, it is unsurprising that this form of distributional shift will affect the performance of an ML algorithm [133]. As the ODD is concerned defining the meta-level attributes of scenarios, it does not necessarily consider whether certain classes will be present or absent from the dataset.

5.2.3 Concept Shift

A concept shift, or concept drift [135], can be defined as the case where either the *posterior* distribution $p(y|\boldsymbol{x})$, e.g., the probability of a lane change for a given scenario, or the *likelihood* $p(\boldsymbol{x}|y)$, e.g., the probability that a given scenario describes a lane change, differs between datasets [136]. Such a shift can result from a change in the causal relationship between the input data and the corresponding labels. It can also occur when the concepts contained in the datasets change over time.

In highway driving data, this form of distributional shift can easily occur. For example, the input features which describe a left lane change on a highway with right-hand traffic will differ from those on a highway with left-hand traffic. Moreover, as more AVs enter the highways, they will not drive exactly the same as human drivers, which would lead to a concept drift between datasets containing only human drivers and those containing a mix of humans and AVs. As we can expect, this is the most difficult form of distributional shift for ML algorithms to deal with [137]. The ODD of a given HAD may or may not be robust to concept shifts depending on how specific it was defined. For example, on a closed geographical location, e.g., an airport, the concepts within this ODD can be specified such that any ML algorithm trained on data collected at a specific airport can be expected to work properly within that ODD.

5.2.4 Possible Sources of Distributional Shifts

There can be many causes for distributional shifts, the four most common sources are briefly described below. For more details, see, e.g., [133; 138].

5.2.4.1 Sample Selection Bias

A sample selection bias can occur whenever data are collected in the real-world under self-imposed constraints. Thus, the data which are collected might not accurately represent the true underlying distribution [133]. In highway driving datasets, a sample selection bias could occur due to, e.g., the time of collection (during rush-hour or not), the location on the highway (just before or after an on-ramp), or the weather (recording in bad weather conditions). Additionally, since datasets are usually collected by different teams, the pre- and post-processing techniques can differ. A properly defined ODD should help combat the sample selection bias constraints should already be considered in the ODD definition.

5.2.4.2 Imbalanced Dataset

In multi-class classification tasks, one class could be rarer than others, leading to imbalanced datasets. To overcome this, researchers can randomly sub-sample the over-represented classes and artificially *balance* the dataset such that each class is represented by the same number of samples. In highway driving, lane changes are much rarer than lane keeping manoeuvres, so researchers generally balance datasets before training ML algorithms on them. By balancing the dataset, we introduce a sample selection bias with a known selection bias. It is difficult for the ODD to consider imbalanced datasets since it considers higher level attributes of the scenario. Furthermore, the ODD does not consider what samples are contained in highway driving datasets.

5.2.4.3 Domain Shift

As defined in [133], a domain shift occurs when the interpretation of the measurements, e.g., the measurement units, varies for different datasets. For example, highway driving datasets can be collected by teams in countries which use the metric measurement units, e.g., kilometres, whereas other teams might use the imperial measurement units, e.g., miles. This can be compensated for by mapping the representations from one measurement unit to another. Another example could be recording a highway scenario from a bird's-eye view or from a vehicle driving in traffic—depending on the vantage

point the same scenario can have various interpretations, e.g., due to occlusions. This source of distributional shift is also difficult to capture within an ODD.

5.2.4.4 Source Component Shift

A further potential cause of distributional shifts can be the source component shift. This occurs when the data are sampled from a number of different sources, e.g., sensors, or if different sub-populations are measured. For example, the type of camera used to record highway driving data can affect the dataset distribution. Here, the source of measurement directly causes different input features and targets to be captured in the datasets [133]. As the ODD is concerned with the scenarios where AVs should function, they might not consider a source component shift.

5.3 Analysing Distributional Shifts in Highway Driving Datasets

The authors of [139] were the first to summarize the challenge of robustifying real-world ML systems to distributional shifts. In the context of ML algorithms in AVs, the authors of [64; 140] discuss the challenge of distributional shifts when deploying AVs in the real-world. However, in these publications, the authors only state the (postulated) challenge of dealing with distributional shifts [64]. In this section, we take this one step further, and explicitly analyse the types of distributional shifts which can occur in highway driving scenarios.

Due to the abundance of image data and the relative ease with which researchers can collect new datasets, distributional shifts between image datasets have been more thoroughly studied. In [141], the authors study the implicit biases which image datasets contain, which is a closely related problem to distributional shifts. They show that a classifier can easily classify which dataset images with the same class label belong to.[2] Moreover, they show that when training an ML algorithm on one dataset and testing on another, the performance drops by roughly 48%. This happens despite the fact that each of the standard image datasets claims to be representative.

The authors of [142] show that ML algorithms trained on a popular benchmark image dataset cannot generalize onto a novel test dataset. The authors create a new dataset by following the same pre-processing steps used to collect the original dataset. They show that all of the state-of-the-art ML-based classification algorithms perform 10% worse in terms of accuracy on this new dataset. This indicates that there is a distributional shift between the original and the newly created datasets.

[2]Note, this is similar to the out-of-distribution detection discussed in Section 5.1.

Recently, seven new datasets with real-world distributional shifts to train and to test ML algorithms on were introduced in [143]. Additionally, the *Shifts* dataset [144] was introduced with the same motivation; it also contains vehicle motion prediction data. These datasets were created with domain experts to represent true distributional shifts which can occur during deployment.

In the following, we analyse the distributional shifts which can occur in realistic highway driving data. We analyse two publicly available highway driving datasets: the highD dataset [7] and the NGSIM dataset [8; 9] (*cf.* Section 3.5.1). Numerous publications propose ML-based driving algorithms and demonstrate them using these datasets, see, e.g., [145] and references therein. We identify and quantify the distributional shifts between these datasets, and show the effect they have on ML-based safety-critical HAD functions.

5.3.1 Highway Dataset Preparation

To analyse the datasets, we first extract the attributes introduced in Section 3.5.1, which were common between both datasets. These were all of the attributes besides the distance to the left and to the right lane markings. We first only extract driving trajectories belonging to the three driving manoeuvres—these will later be split into sub-sequences for the Time to Lane Change (TTLC) classification problem. If a tracked vehicle is in the outer most left (or right) lane, the lateral distance to vehicles to the left (or right) of this vehicle is set to zero since there are no vehicles further left (or right) than it. Since the NGSIM dataset is recorded in imperial units, we convert all measurements into metric units for proper comparison. This can be noted as an example of a *domain shift* (*cf.* Section 5.2.4.3). Furthermore, the NGSIM dataset is recorded at a sampling rate of 10 Hz, whereas the highD dataset is recorded at a sampling rate of 25 Hz. Thus, we sub-sample the highD dataset to have comparable datasets. This is an example of a *source component shift* (*cf.* Section 5.2.4.4).

In the end, we summarise the tracked attributes in a multi-variate time-series signal (*cf.* Definition 2)

$$\boldsymbol{X} = [\bar{\boldsymbol{x}}[1], \bar{\boldsymbol{x}}[2], \ldots, \bar{\boldsymbol{x}}[N]] \in \mathbb{R}^{\Gamma \times N}, \tag{5.1}$$

where at each time-stamp $n \in \{1, \ldots, N\}$, we summarise the Γ attributes in a vector $\bar{\boldsymbol{x}}[n] = [x_1[n], x_2[n], \ldots, x_\Gamma[n]]^T \in \mathbb{R}^\Gamma$. We take scenarios where each vehicle is tracked for at least N time-stamps, and the event, i.e., the lane change, we want to classify occurs within this time duration. In the experiments, we consider 6 s prior to the lane change, i.e., $N = 60$.

	Left	Right	Keep	Total
highD (unbalanced)	2274 (5.88%)	2587 (6.69%)	33822 (87.43%)	38683 (100%)
highD (balanced)	2274	2274	2274	6822
NGSIM (unbalanced)	1555 (2.53%)	500 (0.81%)	59382 (96.66%)	61437 (100%)
NGSIM (balanced)	500	500	500	1500

Table 5.1: The number of lane change (left and right) and lane keeping (keep) scenarios in each dataset, before and after dataset balancing.

Since both datasets have a different number of lane change and lane keeping scenarios, we perform dataset balancing before training the ML-based classifiers (*cf.* Section 5.2.4.2). We uniformly random sample scenarios from each class to ensure that the number of samples per class is equal. The total number of scenarios before and after dataset balancing can be seen in Table 5.1. There are many more lane keeping scenarios than lane changes before re-balancing the datasets. This imbalance is unsurprising since lane changes are relatively rare driving manoeuvres compared to lane keeping. Moreover, we see that the *prior* probability of the lane change manoeuvres are different in the different datasets: in the highD dataset, left lane changes are rarer than right lane changes (5.88% vs. 6.69%, respectively), however, in the NGSIM dataset, these probabilities are reversed (2.53% vs. 0.81%, respectively). This is an example of a *prior probability shift* (*cf.* Section 5.2.2). Moreover, this already indicates that there are differences in the types of driving manoeuvres found in both datasets.

As we have seen in the preparation of the data, these two publicly available datasets contain various types of distributional shifts. Although, we can overcome some of them, e.g., by sub-sampling the highD dataset or (re-)balancing the datasets, we see in the following analyses, that some distributional shifts between the datasets still remain. Furthermore, we see that these remaining distributional shifts negatively affect ML-based HAD functions trained on these datasets.

5.3.2 Qualitative Analysis

In Fig. 5.1, we plot the average attribute value at each time-stamp for all driving manoeuvres—this represents the average attribute distribution for each attribute at each time-stamp. The shaded area represents one standard deviation away from the

mean. We observe that the average longitudinal velocity $v_{\text{long.}}$ (upper right sub-plot) is higher in the highD dataset. This can be expected, as this dataset was collected on a German highway. Moreover, we observe that the distances (lower six sub-plots) to other vehicles is much smaller in the NGSIM dataset, which implies that the traffic density is higher. This could also explain why the average longitudinal velocities are lower in NGSIM. The distributions of the accelerations in both the longitudinal and lateral directions ($a_{\text{long.}}$ and $a_{\text{lat.}}$) are different between the two datasets. At roughly 20 time-stamps before the event occurs ($N \approx 40$) we observe that the lateral acceleration $a_{\text{lat.}}$ in the highD data diverges from the mean. This can be explained by the vehicles accelerating or decelerating shortly before changing lanes to stay in flow with the traffic.

The average trend of some of the signals is also different between the two datasets, e.g., the attribute $d_{\text{r, ahead}}$ grows on average in the highD dataset and stays relatively constant in the NGSIM dataset. This could be explained by different driving styles between the two countries. The standard deviation of the attributes in the highD dataset is also larger than in the NGSIM dataset. This could affect an ML algorithm trained on these datasets, since ML algorithms update their parameters based on the distribution of the training data. Thus, we can qualitatively conclude that there is a *covariate shift* (*cf.* Section 5.2.1) between the highD and the NGSIM dataset.

Another qualitative analysis is to visualise the *likelihood* distribution for the different driving manoeuvres. To this end, we plot the average value of the attributes corresponding to a left lane change from both datasets in Fig. 5.2, i.e., the mean of $p(\boldsymbol{X}|\{y = \text{left}\})$. We observe that the average lateral velocity for left lane changes is similar for both datasets. The distribution of the longitudinal accelerations ($a_{\text{long.}}$) are also similar, but the NGSIM data has a higher standard deviation. The means of the other attributes differ significantly, especially the longitudinal velocity ($v_{\text{long.}}$). We can see that the trend of the distance attributes is different between the datasets. This qualitative analysis indicates that there is also a *concept shift* (*cf.* Section 5.2.3) between the two datasets. An analysis of the distribution of the other driving manoeuvre can be found in Appendix B.[3]

5.3.3 Quantitative Analysis

Following the qualitative analysis of the distributional shifts, we continue with a quantitative analysis. To this end, we employ a two-sample statistical hypothesis test

[3]Note, similar qualitative conclusions about the distributions of the various driving manoeuvres can be made if we use normalised input attributes, e.g., if we normalise each attribute from each dataset to lie within the range of $[0, 1]$.

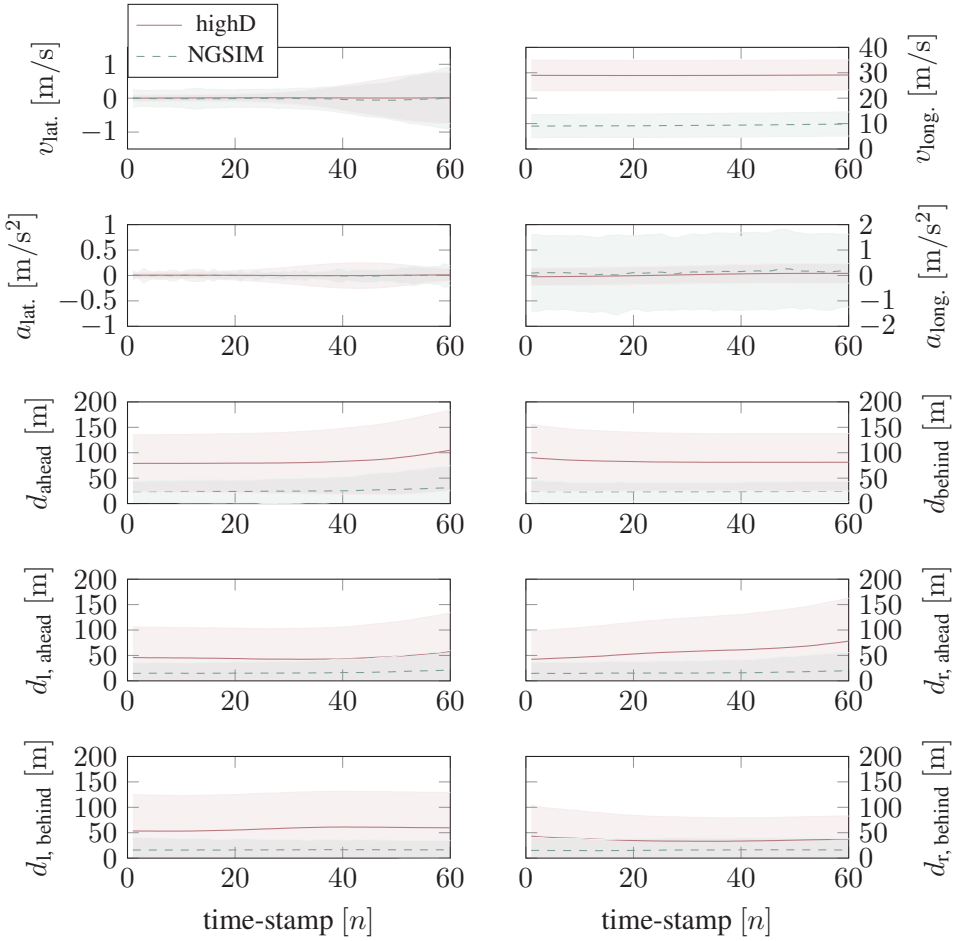


Figure 5.1: Qualitative analysis of the datasets by visualising the average attribute values at each time-stamp for all driving manoeuvres. The shaded areas represent one standard deviation away from the mean.

to determine whether the samples in two datasets originate from the same underlying probability distribution.

5.3.3.1 Statistical Hypothesis Tests

Suppose we have two datasets, $\mathcal{D}_A = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_M\}$ and $\mathcal{D}_B = \{\boldsymbol{x}'_1, \ldots, \boldsymbol{x}'_{M'}\}$, where the samples $\boldsymbol{x} \sim P$ and $\boldsymbol{x}' \sim Q$ are iid sampled from the (unknown) probability distributions P and Q, respectively. We assume $\boldsymbol{x} \in \mathbb{X}$ and $\boldsymbol{x}' \in \mathbb{X}$. We are interested

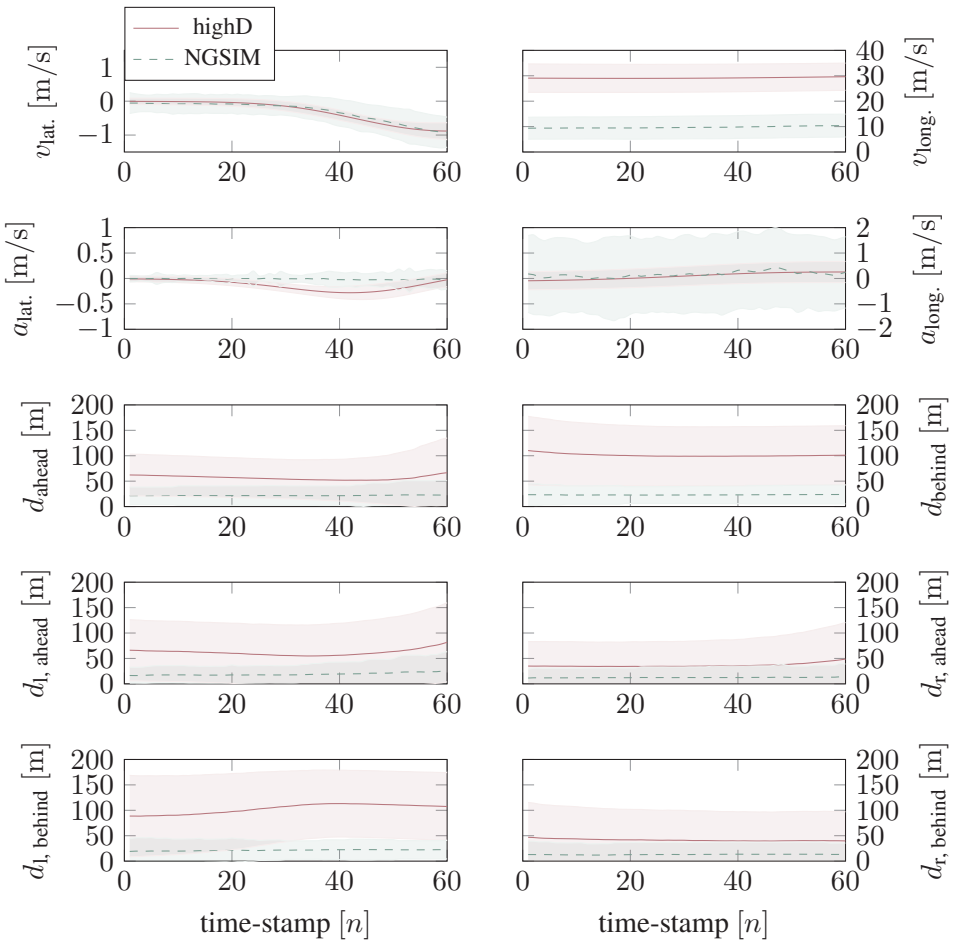


Figure 5.2: Qualitative analysis of the datasets by visualising the average attribute values at each time-stamp for left lane changes. The shaded areas represent one standard deviation away from the mean.

in distinguishing between two hypotheses: the *null hypothesis*,

$$H_0 : P = Q, \tag{5.2}$$

and the *alternative hypothesis*,

$$H_1 : P \neq Q. \tag{5.3}$$

To this end, a test statistic $T : \mathbb{X}^M \times \mathbb{X}^{M'} \to \mathbb{R}$ is constructed based on the samples in both datasets, where M and M' are the number of samples in dataset $\mathcal{D}_A$ and $\mathcal{D}_B$,

respectively. To deal with the situation where the distribution of the test statistic is unknown, we use the bootstrapping method [146], i.e., we uniformly resample the union of the datasets $\mathcal{D}_A \cup \mathcal{D}_B$ without replacement, to estimate the empirical distribution of the test statistic under the null hypothesis H_0. To estimate the empirical distribution of the test statistic under the alternative hypothesis H_1 we follow a similar sampling method as [147, Alg. 1]. We sub-sample the two datasets without replacement to emulate having multiple original datasets. We can subsequently calculate the test statistic T on these sub-sampled datasets under the alternative hypothesis H_1.

Once we derive the test statistic under the null hypothesis H_0, we can calculate a p-value of the test statistic to estimate the statistical significance of rejecting the alternative hypothesis. The p-value is the probability of the two-sample test returning a test statistic at least as large as the test statistic calculated on the datasets $\mathcal{D}_A$ and $\mathcal{D}_B$ (before bootstrap resampling) when H_0 is true. Thus, the null hypothesis is rejected if the p-value lies under a pre-defined significance value α and accepted otherwise; α is usually set to 0.01 or 0.05. This represents a 1% or 5% false alarm rate for H_0 being true. Furthermore, we calculate the True Positive Rate (TPR) of rejecting the null hypothesis when H_1 is true at a 5% false positive rate using [147, Alg. 1].

5.3.3.2 Kernel Test Statistic

A popular choice of a non-parametric test statistic used for two-sample tests is based on the Maximum Mean Discrepancy (MMD) measure, see, e.g., [148] for more details. To define the MMD, we first define a positive definite kernel $k : \mathbb{X} \times \mathbb{X} \to \mathbb{R}$ in a Reproducing Kernel Hilbert Space (RKHS) $\mathcal{H}$, where $k(\boldsymbol{x}, \boldsymbol{x}') = \langle \phi(\boldsymbol{x}), \phi(\boldsymbol{x}') \rangle_{\mathcal{H}}$ with a corresponding feature mapping $\phi \in \mathcal{H}$. The MMD measure between two probability distributions can be defined as [148, Lemma 4],

$$\mathrm{MMD}^2(P, Q) = \|\boldsymbol{\mu}_P - \boldsymbol{\mu}_Q\|^2_{\mathcal{H}}, \tag{5.4}$$

with the mean embedding of the distribution P or Q as $\boldsymbol{\mu}_P \in \mathcal{H}$ or $\boldsymbol{\mu}_Q \in \mathcal{H}$, respectively. The mean embeddings are defined such that $\mathbb{E}_P[\phi(\boldsymbol{x})] = \langle \phi(\boldsymbol{x}), \boldsymbol{\mu}_P \rangle_{\mathcal{H}}$ and $\mathbb{E}_Q[\phi(\boldsymbol{x})] = \langle \phi(\boldsymbol{x}), \boldsymbol{\mu}_Q \rangle_{\mathcal{H}}$ for all $\phi \in \mathcal{H}$. Thus, the MMD measure compares all higher-order moments of the distributions in the RKHS. The MMD measure is equal to 0 if and only if $P = Q$ [148, Lemma 5]; if the kernel is universal.

If we have two datasets $\mathcal{D}_A$ and $\mathcal{D}_B$ whose samples are assumed to be drawn from P and Q, respectively, we can state an unbiased empirical estimate of the squared

MMD measure as [148, Lemma 6]

$$\widehat{\mathrm{MMD}}^2(\mathcal{D}_A, \mathcal{D}_B) = \frac{1}{M(M-1)} \sum_{\substack{i=1 \\ j \neq i}}^{M} k(\boldsymbol{x}_i, \boldsymbol{x}_j) + \frac{1}{M'(M'-1)} \sum_{\substack{i=1 \\ j \neq i}}^{M'} k(\boldsymbol{x}'_i, \boldsymbol{x}'_j) - \frac{2}{MM'} \sum_{i=1}^{M} \sum_{j=1}^{M'} k(\boldsymbol{x}_i, \boldsymbol{x}'_j). \tag{5.5}$$

In the experiments, we employ the Gaussian kernel, $k(\boldsymbol{x}, \boldsymbol{x}') = \exp(-\gamma \|\boldsymbol{x} - \boldsymbol{x}'\|_2^2)$, where $\boldsymbol{x}, \boldsymbol{x}' \in \mathbb{X}$, and the bandwidth parameter $\gamma = 1/(D\sigma^2)$ with the input dimension D and where we use σ^2 as the variance of all samples in the datasets $\mathcal{D}_A$ and $\mathcal{D}_B$.

5.3.3.3 Classifier Two-Sample Test

Alternatively, we can train a Deep Neural Network (DNN) to detect whether the samples come from the same distribution—we use the Classifier Two-Sample Test (C2ST) [149]. First, we create a training dataset $\mathcal{S}$ by taking an equal number of samples from both datasets, i.e.,

$$\mathcal{S} = \{(\boldsymbol{x}_i, y_i = 1)\}_{i=1}^{M_\mathcal{S}} \cup \{(\boldsymbol{x}'_j, y_j = 0)\}_{j=1}^{M_\mathcal{S}}, \tag{5.6}$$

where the samples $\boldsymbol{x} \in \mathcal{D}_A$ and $\boldsymbol{x}' \in \mathcal{D}_B$, and we take $M_\mathcal{S} \leq \min(M, M')$. Furthermore, we split the dataset $\mathcal{S}$ into $\mathcal{S}_{\text{train}}$ and $\mathcal{S}_{\text{test}}$, with $M_{\mathcal{S},\text{train}}$ and $M_{\mathcal{S},\text{test}}$ samples, respectively

We train a classifier $g : \mathbb{X} \to [0, 1]$, which approximates the *posterior* distribution $p(y_k = 1|\boldsymbol{x})$, when employing a sigmoid activation function at the output of the DNN. A hypothesis test can now be performed by analysing the accuracy of the classifier on the test dataset, i.e.,

$$T_{\text{C2ST}} = \frac{1}{M_{\mathcal{S},\text{test}}} \sum_{k=1}^{M_{\mathcal{S},\text{test}}} \mathbb{I}(\text{round}(g(\boldsymbol{x}_k)) = y_k), \tag{5.7}$$

with the indicator function $\mathbb{I}$. The test statistic T_{C2ST} is Gaussian distributed with $T_{\text{C2ST}} \sim \mathcal{N}(1/2, 1/(4M_{\mathcal{S},\text{test}}))$ [149]. Thus, the null hypothesis H_0 is accepted when the test statistic T_{C2ST} (the accuracy of the classifier) is around 50%, and rejected when a pre-defined threshold is exceeded, e.g., $T_{\text{C2ST}} \geq 90\%$. Since we only have two classes as the samples are labelled according to which dataset they stem from, if the classifier has an accuracy of 50% it is the same as random guessing. Therefore, the classifier is unable to differentiate between the two classes and H_0 is accepted.

5.3.3.4 Quantitative Results

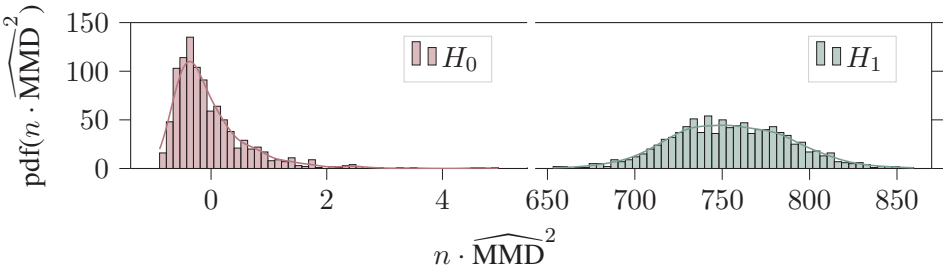
First, we estimate the distribution of the MMD measures under H_0 and H_1 as described in Section 5.3.3.2. We plot the estimated distributions in Fig. 5.3 for the different driving manoeuvres in the highD and the NGSIM datasets. We observe that the distribution of the MMD under H_0 and H_1 are very different, i.e., a statistical hypothesis test would reject H_0 indicating that there is a distributional shift between the two datasets. On top of this, we see that the distribution of the MMD measure for the different driving manoeuvres also indicate a distributional shift. In all cases, the p-values are smaller than $\alpha = 0.05$, indicating that the null hypothesis H_0 would be rejected. Looking at the TPR under H_1, we observe that a hypothesis test with a false alarm rate of 5% would always accept H_1. These results quantitatively indicate that there is a distributional shift between the two datasets and between the driving manoeuvres contained in the datasets (a *concept shift*, *cf.* Section 5.2.3)).

In order to verify these results, we estimate the distribution of the MMD measure under H_0 and H_1 on each dataset individually. To achieve this, we split each dataset into two disjoint sets and calculate the MMD measure on these sub-sets. The results are depicted in Fig. 5.4. We observe in Fig. 5.4a and Fig. 5.4b that the distributions are similar. For these results, we observe a p-value larger than $\alpha = 0.05$, indicating that the null hypothesis H_0 would be accepted. Moreover, we achieve a TPR under H_1 of around 5.0% for both tests, indicating that a hypothesis test using the MMD measure would more likely accept H_0. This indicates that there is no distribution shift within either the highD nor the NGSIM datasets.

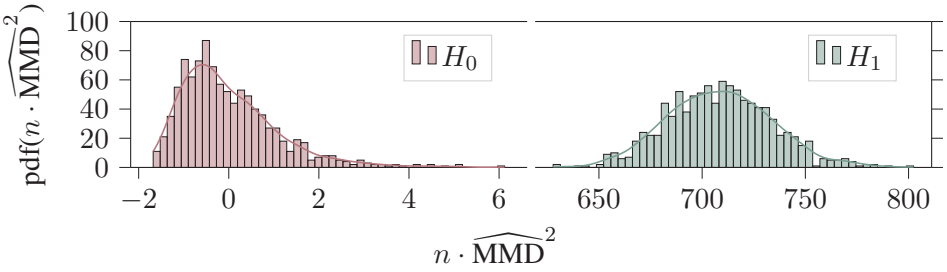
Following this, we analyse the results of using the C2ST to determine whether there is a distributional shift between the datasets. The C2ST results are depicted as the classification accuracy on $\mathcal{S}_{\text{test}}$. The C2ST network architecture is taken from [150, Sec. 2.2.2], with one output neuron. We train it for 50 epochs using the Adam optimiser [102] with mini-batches of size 16 and an initial learning rate of $\eta = 0.005$. To get reliable results, we average the accuracies of ten C2STs trained on different train/test splits of the various datasets.

We observe in Tab. 5.2a that the C2ST rejects the null hypothesis H_0 for the whole dataset since the average accuracy is larger than 90%. Furthermore, the bottom three rows indicate that there is a *concept shift* (*cf.* Section 5.2.3) between the two datasets for all driving manoeuvres; the C2ST rejects the null hypothesis H_0 for these as well.

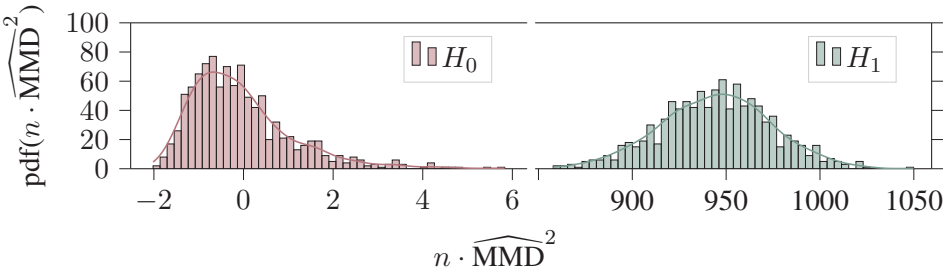
In Tab. 5.2b, we show the results of the C2ST with samples only from within one dataset, i.e., we split each dataset into two disjoint sets, and test whether the two sub-sets come from the same distribution. The C2ST indicates that samples from



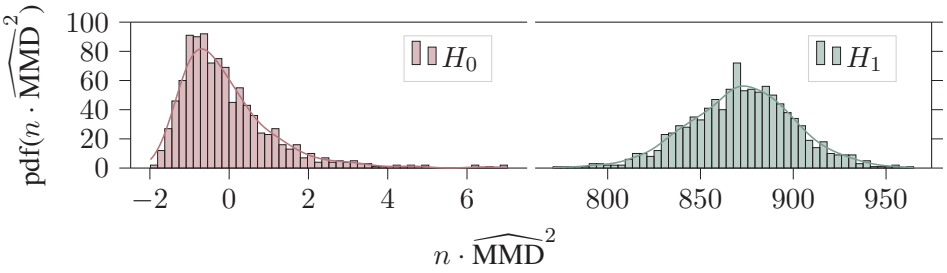
(a) $\mathcal{D}_{\text{highD}}$ vs. $\mathcal{D}_{\text{NGSIM}}$; p-value : 0.001; TPR H_1 : 100%.



(b) $\mathcal{D}^{\text{keep}}_{\text{highD}}$ vs. $\mathcal{D}^{\text{keep}}_{\text{NGSIM}}$; p-value : 0.001; TPR H_1 : 100%.

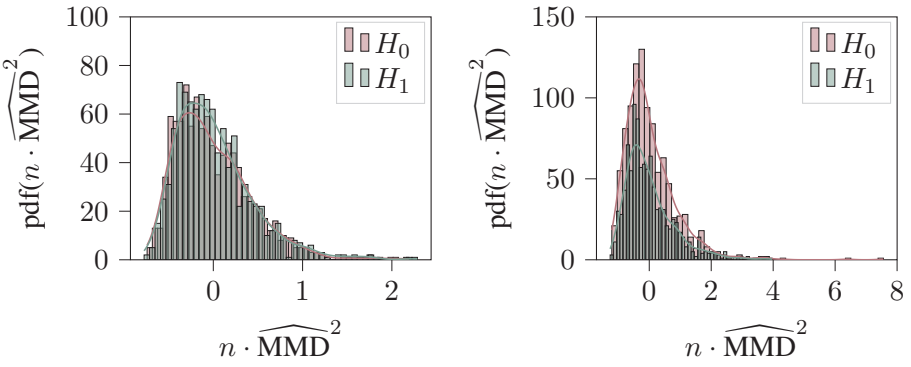


(c) $\mathcal{D}^{\text{left}}_{\text{highD}}$ vs. $\mathcal{D}^{\text{left}}_{\text{NGSIM}}$; p-value : 0.001; TPR H_1 : 100%.



(d) $\mathcal{D}^{\text{right}}_{\text{highD}}$ vs. $\mathcal{D}^{\text{right}}_{\text{NGSIM}}$; p-value : 0.001; TPR H_1 : 100%.

Figure 5.3: Estimates of the MMD test statistic distribution under H_0 and H_1 for different datasets and driving manoeuvres. In these cases, the hypothesis test would reject the null hypothesis, indicating there is a distributional shift between the datasets. (Note, the horizontal axes are discontinuous in these plots.)



(a) $\mathcal{D}_{\text{highD}}$ vs. $\mathcal{D}_{\text{highD}}$; p-value : 0.728; TPR H_1 : 5.0%

(b) $\mathcal{D}_{\text{NGSIM}}$ vs. $\mathcal{D}_{\text{NGSIM}}$; p-value : 0.452; TPR H_1 : 5.2%.

Figure 5.4: Estimates of the MMD test statistic distribution under H_0 and H_1 for the same dataset. In these cases, the hypothesis test would accept the null hypothesis, indicating there is no distributional shift within the datasets.

	T_{C2ST} [Acc. $\%$]
$\mathcal{D}_{\text{highD}}$ vs. $\mathcal{D}_{\text{NGSIM}}$	$97.25(\pm 0.27)$
$\mathcal{D}^{\text{left}}_{\text{highD}}$ vs. $\mathcal{D}^{\text{left}}_{\text{NGSIM}}$	$95.35(\pm 1.14)$
$\mathcal{D}^{\text{right}}_{\text{highD}}$ vs. $\mathcal{D}^{\text{right}}_{\text{NGSIM}}$	$94.25(\pm 2.01)$
$\mathcal{D}^{\text{keep}}_{\text{highD}}$ vs. $\mathcal{D}^{\text{keep}}_{\text{NGSIM}}$	$94.40(\pm 1.68)$

(a) Hypothesis testing between datasets.

	T_{C2ST} [Acc.$\%$]
$\mathcal{D}_{\text{highD}}$ vs. $\mathcal{D}_{\text{highD}}$	$49.68(\pm 0.84)$
$\mathcal{D}^{\text{left}}_{\text{highD}}$ vs. $\mathcal{D}^{\text{left}}_{\text{highD}}$	$48.51(\pm 1.01)$
$\mathcal{D}^{\text{right}}_{\text{highD}}$ vs. $\mathcal{D}^{\text{right}}_{\text{highD}}$	$48.67(\pm 1.63)$
$\mathcal{D}^{\text{keep}}_{\text{highD}}$ vs. $\mathcal{D}^{\text{keep}}_{\text{highD}}$	$49.16(\pm 0.97)$
$\mathcal{D}_{\text{NGSIM}}$ vs. $\mathcal{D}_{\text{NGSIM}}$	$48.63(\pm 1.10)$
$\mathcal{D}^{\text{left}}_{\text{NGSIM}}$ vs. $\mathcal{D}^{\text{left}}_{\text{NGSIM}}$	$51.50(\pm 3.00)$
$\mathcal{D}^{\text{right}}_{\text{NGSIM}}$ vs. $\mathcal{D}^{\text{right}}_{\text{NGSIM}}$	$46.30(\pm 2.08)$
$\mathcal{D}^{\text{keep}}_{\text{NGSIM}}$ vs. $\mathcal{D}^{\text{keep}}_{\text{NGSIM}}$	$45.20(\pm 1.26)$

(b) Hypothesis testing within datasets.

Table 5.2: Results indicating: (a) distributional shifts between the NGSIM dataset and the highD dataset (all T_{C2ST} average accuracies are $> 90\%$); (b) no distributional shifts were detected within either dataset (all T_{C2ST} average accuracies are around 50%). The results are averaged over 10 random restarts with the 95% confidence interval in brackets.

within each dataset stem from the same distribution (since the classification accuracy is around 50%). Moreover, the driving manoeuvres within each dataset also come from the same underlying distribution. On top of this, these results indicate that combining the two NGSIM datasets into one dataset, as done in this analysis, was meaningful.

5.3.4 Effects of Distributional Shifts on Learned Models

To evaluate the effect distributional shifts have on learned ML models, we train ML-based classifiers to perform the TTLC task (*cf.* Section 3.5.2). We train the same DNN architectures introduced in Chapter 4 (see Table 4.1). This was a collection of various 1-D Convolutional Neural Networks (CNNs) architectures (including Multi-Channel (MC) architectures), Recurrent Neural Network (RNN)-based architectures, an attention-based architecture, and a Fully Connected (FC)-DNN.

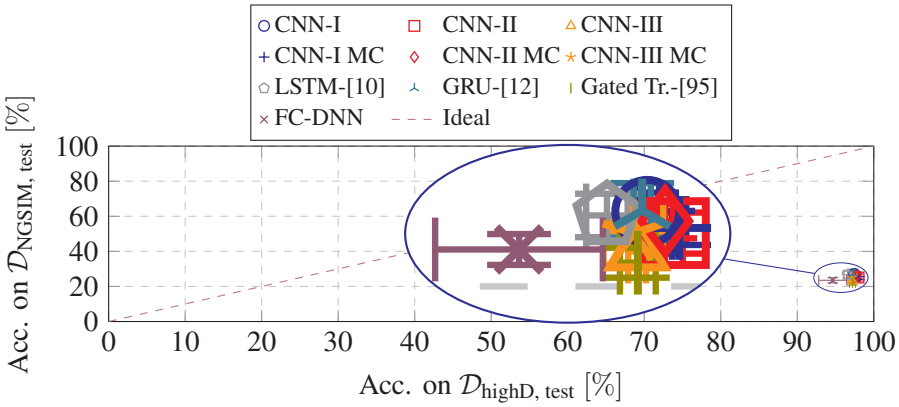
To generate a suitable dataset to train the DNNs, we first extract all scenarios from the highD [7] and the NGSIM [8; 9] datasets where the tracked vehicle is visible for 6 s time-stamps prior to the driving manoeuvre; equivalent to $N = 60$ time-stamps. In this section, we consider sub-sequences of length 3 s, such that each lane change manoeuvre is split into sub-sequences of length $N_{\text{sub.}} = 30$ time-stamps. Therefore, we have a total of two classes for left and right lane changes, and one class for the lane keeping manoeuvre, i.e., $\mathbb{Y} = \{\text{L}_1, \text{L}_2, \text{K}, \text{R}_2, \text{R}_1\}$ (*cf.* Section 3.5.2).[4]

We split both balanced datasets into disjoint sets. We use 80% of the data for the training set and 20% for the test set, i.e., $M_{\text{train}} = \lfloor 0.8 \cdot M \rfloor$ and $M_{\text{test}} = M - M_{\text{train}}$. Thus, we create the datasets $\mathcal{D}_{\text{highD, train}}$, $\mathcal{D}_{\text{highD, test}}$, and $\mathcal{D}_{\text{NGSIM, train}}$, $\mathcal{D}_{\text{NGSIM, test}}$, respectively.

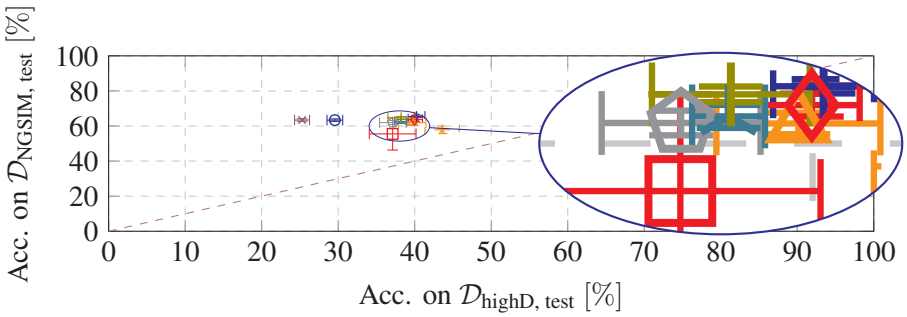
To train the DNNs, we use the cross-entropy loss function (*cf.* Section 3.4.1). We train each algorithm using the Adam optimiser [102] with an initial learning rate of $\eta = 0.0005$ and mini-batches of size 200. All of the network parameters are initialised using the Glorot initialisation [111]. We employ early stopping with a patience of 20 epochs to reduce overfitting the training set. Additionally, we average the classification performance using 10-fold cross validation (see [69]). The exact parameters and architecture designs of the models in Table 4.1 can be found in Appendix C.1. The algorithms are trained once on $\mathcal{D}_{\text{highD, train}}$ and once on $\mathcal{D}_{\text{NGSIM, train}}$.

To visualise the effect which the distributional shifts have on trained ML algorithms, we plot the average accuracy of each classifier architecture on the test sets from both datasets. This visualisation method was introduced in [142]. Fig. 5.5a and Fig. 5.5b

[4]Note, this is a slightly different TTLC task compared to Chapter 4 due to the different sampling rate of the NGSIM dataset used in this chapter.



(a) Classifiers trained on $\mathcal{D}_{\text{highD, train}}$.



(b) Classifiers trained on $\mathcal{D}_{\text{NGSIM, train}}$.

Figure 5.5: The average accuracy of the learned ML algorithms tested on both test datasets. The error bars represent the 95% confidence intervals. All models perform better on the distribution they were trained on.

depict the accuracies when the models are trained on either the highD or the NGSIM training datasets, respectively. Each plot depicts an accuracy pair of an ML-model trained on one distribution and tested on both distributions. The error bars represent the 95% confidence intervals. The lines labelled *Ideal* in Fig. 5.5 shows the ideal transferability between two datasets if there was no distributional shift, i.e., the test performance would be the same on both datasets.

We observe in Fig. 5.5a the accuracies of the models trained on $\mathcal{D}_{\text{highD, train}}$. We see that the accuracy on $\mathcal{D}_{\text{highD, test}}$ is much larger than on $\mathcal{D}_{\text{NGSIM, test}}$. Almost all ML models achieve an accuracy of over 95% on the $\mathcal{D}_{\text{highD, test}}$, which stems from the same distribution as the data they were trained on (*cf.* Section 5.3.3.4). However, on $\mathcal{D}_{\text{NGSIM, test}}$, the models only achieve an accuracy below 35%. We observe that the

performance of all of classifiers drops by an average of 72%. This could be seen as the classifiers overfitting the training distribution ($\mathcal{D}_{\text{highD, train}}$).

On the other hand, when we train the ML models on $\mathcal{D}_{\text{NGSIM, train}}$, we obtain the results in Fig. 5.5b. In this plot, we see that the distributional shifts do not affect the models as much when they are trained on $\mathcal{D}_{\text{NGSIM, train}}$ since the accuracy pairs lie closer to the *Ideal* line. However, we see that the performance on $\mathcal{D}_{\text{NGSIM, test}}$ is still better than on $\mathcal{D}_{\text{highD, test}}$ (all accuracy pairs lie above the *Ideal* line). Moreover, we observe that there is a greater spread of the classification performances of the different classifiers, e.g., the FC-DNN classifiers are affected the most by the distributional shift, whereas the CNN-III MC classifiers lie close to the *Ideal* line. In this case, the average decrease in accuracy is only 25%. However, the accuracies on the training dataset are not impressive. One reason for this could be that the NGSIM training dataset is much smaller than the highD dataset; it is known that DNN models require lots of data to train and generalise well.

5.4 Fine-Tuning under Distributional Shifts

Given the results from the previous section, we now consider how we can train an ML-based classifier under distributional shifts. A first idea would be to use fine-tuning to transfer the knowledge from one domain into the other, see, e.g., [130; 131]. However, as we see in the following, transferring the knowledge without any further considerations of the distributional shifts between the two datasets will leave us with a model which performs well on the new distribution (the target domain) but poorly on the original distribution (the source domain). If performed naively, fine-tuning will forget the source distribution and only fit the target distribution, i.e., we end up going from an accuracy-pair point in Fig. 5.5a to an accuracy-pair point in Fig. 5.5b or vice versa.

As we saw in Chapter 4, the feature embeddings in the penultimate layer of a trained DNN are important and sufficient to perform a classification task (*cf.* Section 4.4.1 and Table 4.3). Therefore, we want to use the feature embeddings from the source domain when fine-tuning on the target domain. To this end, we introduce an additional loss term to the fine-tuning classification loss function. This gives an engineer the ability to control the transfer of knowledge from the source to the target domain. They can trade off forgetting the source domain and learning the target domain. On top of that, methods from domain adaptation tasks (*cf.* [130]), e.g., deep domain confusion [151] or deep adaptation network [152], use the feature embeddings at various layers in a DNN in the loss function to adapt a model to a new domain.

In the following, we refer to the dataset from the source and from the target domain as $\mathcal{D}_S$ and $\mathcal{D}_T$, respectively. The DNN trained on the source domain is f_S with parameters $\mathcal{P}_S$; the DNN fine-tuned on the target domain is f_T with parameters $\mathcal{P}_T$. The DNN f_T is initialised with the parameters $\mathcal{P}_S$, i.e., the parameters learned on the source domain. Ultimately, we aim to train the DNN f_T on $\mathcal{D}_T$ without forgetting what was learned on $\mathcal{D}_S$.

We introduce a fine-tuning loss function aiming to keep the feature embeddings in the target domain close to those learned from the source domain. Thus, we use the source domain feature embeddings when fine-tuning the DNN's parameters on the target domain. This leads to the loss function

$$\mathcal{L}_{\text{ce + dist.}}(\mathcal{D}_T; \mathcal{P}_T) = (1-\zeta) \cdot \mathcal{L}_{\text{ce}}(\mathcal{D}_T; \mathcal{P}_T) + \zeta \cdot \mathcal{L}_{\text{dist.}}(\text{feat}_{f_S}, \text{feat}_{f_T}, \mathcal{D}_T; \mathcal{P}_T), \tag{5.8}$$

where $\mathcal{L}_{\text{ce}}$ is the cross-entropy loss on the target domain and $\mathcal{L}_{\text{dist.}}$ is a loss function keeping the distance between the feature embeddings from the source network (feat_{f_S}) close to those in the target domain (feat_{f_T}). The parameter $\zeta \in [0, 1)$ is the trade-off between how much weight we put on learning the classification task on the target dataset ($\mathcal{L}_{\text{ce}}$) and how much weight we put on keeping the feature embeddings close ($\mathcal{L}_{\text{dist.}}$). With a $\zeta = 0$, we perform fine-tuning without considering the feature embeddings in the source domain. When $\zeta \to 1$, the loss function de-emphasises the classification task in the target domain and primarily focuses on keeping the feature embeddings in the target domain close to those from the source domain. The distance loss function is defined as

$$\mathcal{L}_{\text{dist.}}(\text{feat}_{f_S}, \text{feat}_{f_T}, \mathcal{D}_T; \mathcal{P}_T) = \frac{1}{M_T} \sum_{m=1}^{M_T} \|\text{feat}_{f_S}(\boldsymbol{X}^m; \mathcal{P}_S) - \text{feat}_{f_T}(\boldsymbol{X}^m; \mathcal{P}_T)\|_2^2, \tag{5.9}$$

where the feature embedding functions feat_{f_S} and feat_{f_T} (*cf.* (4.1)[5]) depend on the parameters learned on the source domain $\mathcal{P}_S$ and the target domain $\mathcal{P}_T$, respectively. This loss function encourages the feature embeddings from the target domain to remain close to those calculated using the source network.

Algorithm 3 summarises how to fine-tune the parameters on the target distribution. As inputs to the algorithm, we require the DNN trained on the source domain (f_S) with parameters $\mathcal{P}_S$, a number of training epochs E, a training dataset from the target

[5]Note, to differentiate between the feature embedding function using the source DNN and the target DNN, we explicitly label the function with f_S and f_T, respectively.

Algorithm 3 Fine-Tuning under Distributional Shifts

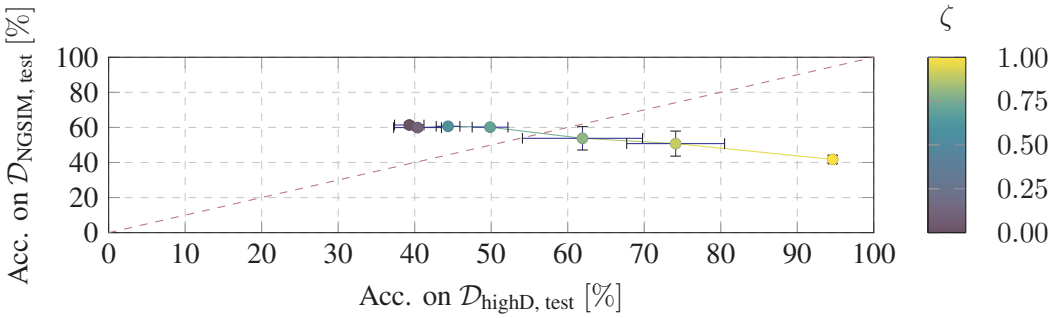
1: **Inputs:** DNN f_S trained on $\mathcal{D}_S$ with parameters $\mathcal{P}_S$, number of epochs E, target training dataset $\mathcal{D}_T$, trade-off parameter ζ
2: $\mathcal{P}_T \leftarrow \mathcal{P}_S$
3: **for** $\text{epoch} = 1, \ldots, E$ **do**
4: $\quad G \leftarrow \nabla_{\mathcal{P}_T} \left[(1-\zeta) \cdot \mathcal{L}_{\text{ce}}(\mathcal{D}_T; \mathcal{P}_T) + \zeta \cdot \mathcal{L}_{\text{dist.}}(\text{feat}_{f_S}, \text{feat}_{f_T}, \mathcal{D}_T; \mathcal{P}_T)\right]$
5: $\quad$ Update the parameters $\mathcal{P}_T$ of f_T to minimise (5.8) using the gradients G and an optimiser, e.g., ADAM
6: **end for**

domain $\mathcal{D}_T$, and the trade-off parameter ζ. In Line 2, we initialise the target DNN with the same parameters learned on the source domain. In Line 4, we take the gradient of the fine-tuning loss function with respect to the parameters $\mathcal{P}_T$. Then we update the parameters $\mathcal{P}_T$ to minimise (5.8) using the gradients G. Note, during this fine-tuning, the parameters $\mathcal{P}_S$ are fixed and not updated. The trade-off parameter ζ allows an engineer to either learn more of the target domain (by setting $\zeta \to 0$) or remember more from the source domain (by setting $\zeta \to 1$).

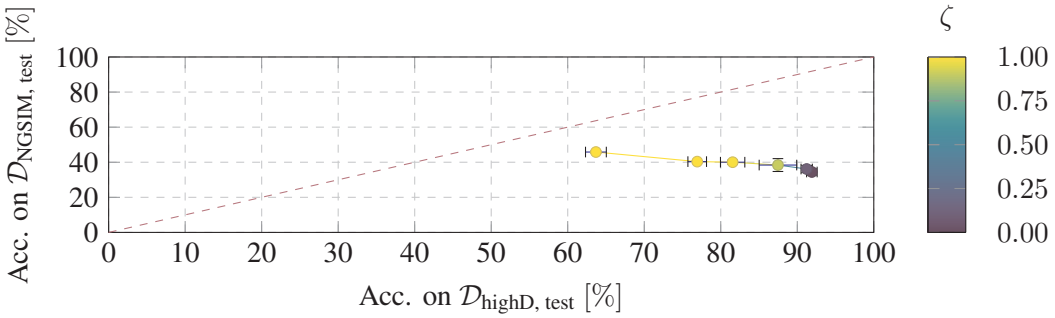
5.4.1 Fine-Tuning under Distributional Shifts: Results

With the proposed fine-tuning algorithm (Algorithm 3), we can now fine-tune the parameters of the networks introduced in Section 5.3.4. We use the same 80%/20% train/test split as introduced in the previous section for the source and the target datasets. To fine tune the DNNs, we use the Adam optimiser [102] with an initial learning rate of $\eta = 0.0001$ and mini-batches of size 64. We set the maximum number of epochs to $E = 100$ in Algorithm 3.

In Fig. 5.6, we highlight how the trade-off parameter ζ can be used to control forgetting of the source domain and the learning of the target domain. We fine-tune on the CNN-I architecture; however, similar results can be achieved when fine-tuning the other DNN architectures (see Fig. 5.7 or Appendix B.2). In Fig. 5.6a, we show the results of fine-tuning the ML algorithm on the NGSIM dataset after originally training on the highD dataset; we plot the converse results in Fig. 5.6b. We vary ζ between 0 and 1, where $\zeta = 0$ is equivalent to fine-tuning without taking the distance loss into account (*cf.* (5.8)). The different ζ values are represented by the colour of the points. We observe that by merely fine-tuning the CNN on the target distribution, we end up with a similar performance as when we do not consider the source distribution at all. We can compare the dark blue point in Fig. 5.6a with the corresponding point



(a) Classifiers trained on $\mathcal{D}_S = \mathcal{D}_{\text{highD, train}}$ and fine-tuned on $\mathcal{D}_T = \mathcal{D}_{\text{NGSIM, train}}$.



(b) Classifiers trained on $\mathcal{D}_S = \mathcal{D}_{\text{NGSIM, train}}$ and fine-tuned on $\mathcal{D}_T = \mathcal{D}_{\text{highD, train}}$.

Figure 5.6: The average accuracy of the CNN-I architecture, trained on a source dataset $\mathcal{D}_S$ and fine-tuned on the target dataset $\mathcal{D}_T$. The models are then tested on both test datasets. The error bars represent the 95% confidence intervals.

in Fig. 5.5b. In this case, the ML algorithm performs well on the target distribution but poorly on the source distribution likely due to the distributional shift between the two datasets. On the other hand, we observe that by increasing ζ, we can trade off the forgetting and learning to achieve (arbitrary) points along the curve in the accuracy space. For example, it might be important for the ML algorithm to perform equally well on both distributions or we might want to encourage the DNN not to forget the source domain. We see a similar ability to influence the trade-off with ζ in Fig. 5.6b.

To highlight that the proposed fine-tuning method works on the other DNN architectures, we plot the accuracy pairs before and after fine-tuning in Fig. 5.7. For each DNN architecture, we plot the average accuracy on $\mathcal{D}_{\text{highD, test}}$ and $\mathcal{D}_{\text{NGSIM, test}}$. We first plot the accuracies before fine-tuning (Pre), i.e., the accuracy pairs from Fig. 5.5a. Then we plot the fine-tuning with $\zeta = 0$ and with $\zeta \to 1$, to highlight the two extreme points of the proposed fine-tuning algorithm. We see, in general,

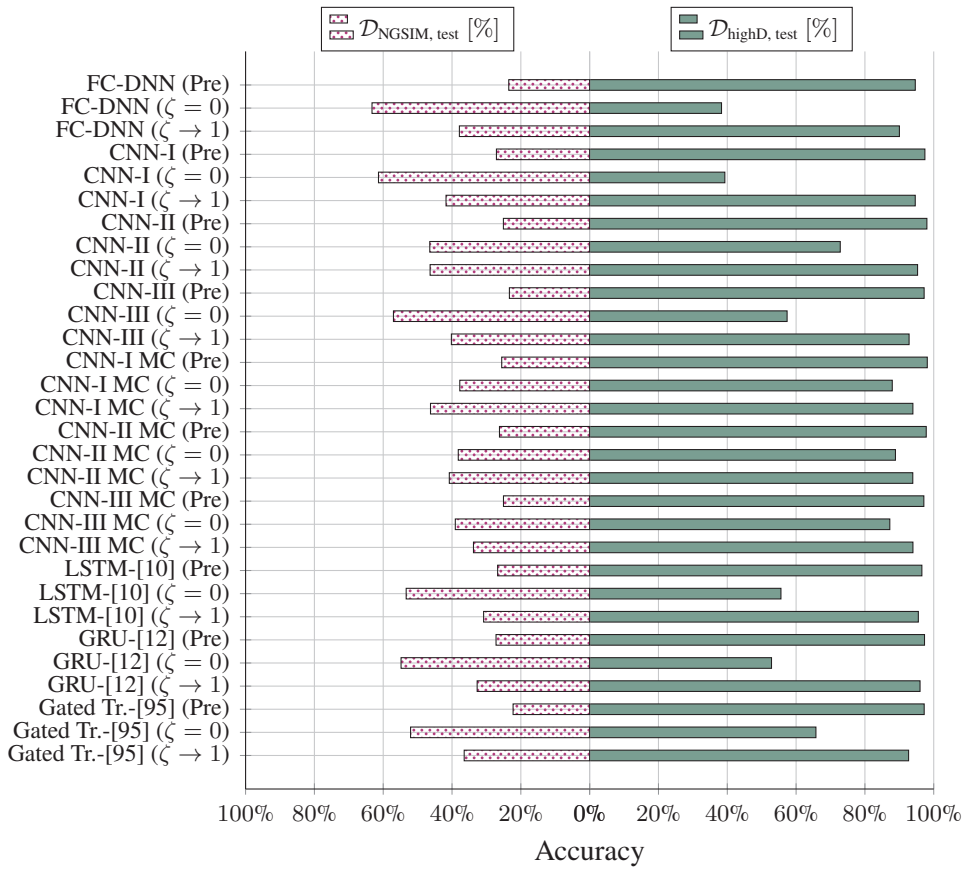


Figure 5.7: Classification performance before and after fine-tuning with Algorithm 3 with different ζ values. The classifiers are trained on $\mathcal{D}_S = \mathcal{D}_{\text{highD, train}}$ and fine-tuned on $\mathcal{D}_T = \mathcal{D}_{\text{NGSIM, train}}$.

that before fine-tuning there is a large discrepancy between the accuracy on the two test datasets (the rows with the label *Pre*). When we fine-tune without taking the source distribution into account ($\zeta = 0$), we observe that this discrepancy changes sign, i.e., the DNN performs better on the target distribution ($\mathcal{D}_T$) than on the source distribution ($\mathcal{D}_S$). However, if we set $\zeta \to 1$, we see that we do not forget the source distribution ($\mathcal{D}_S$) but gain performance on the target distribution ($\mathcal{D}_T$). By varying the trade-off parameter between 0 and 1, an engineer can choose a suitable parameter for the task at hand. Moreover, the proposed fine-tuning algorithm can influence the learning of a wide-range of DNN architectures under distributional shifts.

5.5 Dataset Distribution based Safety Argument: Summary

Motivated by the challenge of validating ML algorithms in safety-critical HAD functions, we turn our attention to the data which are used to train these driving functions in this chapter. First, we recapitulate the different types of distributional shifts which can occur in highway driving data, and we discuss when and how these shifts may occur. Subsequently, we demonstrate that a distributional shift exists between two public highway driving datasets. We provide both a qualitative and a quantitative analysis of the distributional shifts between the datasets.

We additionally show that these shifts impact the performance of an ML-based HAD function trained on the datasets. Ultimately, we introduce a fine-tuning algorithm allowing an engineer to trade off forgetting of the source distribution and learning of the target distribution. This method can be used in safety-critical HAD functions if a distributional shift has been detected between two datasets, e.g., when a vehicle drives in a new country or in novel environment conditions.

The methods we introduce in this chapter can be used to first analyse the distribution of the datasets where the ML-based HAD function might be deployed and to fine-tune this ML algorithm to the new distribution. Allowing an engineer to argue for safety by considering—and adapting an ML algorithm to—the distributional shifts between different datasets.

Validation Safety Arguments based on Algorithm Designs

6

In this chapter, we discuss methods to develop safety arguments for the validation of Highly Automated Driving (HAD) functions based on the design of the Machine Learning (ML) algorithm (*cf.* Section 6.1). We focus on designing an interpretable ML algorithm which directly aids in validating ML-based HAD functions. We introduce an interpretable Lane Change Detector (LCD) algorithm based on an explicit decision rule-set considering the reconstruction errors of Deep Autoencoders (DAEs) trained to reproduce specific driving manoeuvres (*cf.* Section 6.2). We further demonstrate the interpretability and tunability of this interpretable LCD algorithm (*cf.* Section 6.3). The results presented in this chapter are based on those found in [17; 153].

6.1 Algorithm Design Motivation

The overall design of the ML algorithm plays a major role in creating a validation safety argument in safety-critical applications. Whether an engineer uses a traditional ML algorithm, e.g., a Decision Tree (DT) or a k-Nearest Neighbours (k-NN) classifier, or if they choose to use an end-to-end learned ML algorithms[1] will affect the overall safety of the system. The former methods are directly human interpretable whereas an end-to-end learned ML system is a black-box. On top of that, data-driven ML-based functions are vulnerable in various ways, e.g., they can easily be attacked by perturbing the inputs or they can be overly confident on wrong outputs.

One possibility to tackle the vulnerabilities which ML-based functions display is to apply a function corrector model around an already trained black-box ML algorithm, see, e.g., [154; 155]. These function corrector models are motivated by stochastic separability theories ([156; 157]) which make use of properties in high dimensional spaces. Specifically, they take advantage of the measure concentration

[1]End-to-end learned HAD functions generally take the environment signal data as input to the ML algorithm, and output the Autonomous Vehicle (AV) controls directly. This algorithm design is in contrast to a modular design with, e.g., perception, planning, and acting modules (*cf.* Fig. 1.1).

in high dimensional space such that the higher the data dimension, the higher the probability that one can linearly separate finite sets of samples. These stochastic separability theories were proven for various (simple) distributions, e.g., uniform distribution on a ball or a unit cube [156; 157]. Moreover, when applied to the large Deep Neural Network (DNN) models, the authors of [154] show a linear corrector can separate incorrect classifications from correct classifications. Thus, a corrector can improve the overall performance of the ML-based function. We could design a corrector function around the original ML algorithm and correct its classifications.

Additionally, ML algorithms are vulnerable to adversarial attacks [158]—these are perturbed inputs (usually imperceivable to the human eye) but the DNN outputs an incorrect label with high confidence, e.g., the image of a *cat* can be perturbed such that it is classified as a *dog* by the DNN. These adversarial attacks have also been shown to work on time-series data, see, e.g., [159; 160; 161], and even in the real-world by manipulating road signs, see, e.g., [162]. As more HAD functions integrate ML-based pipelines, these adversarial attacks pose an ever greater challenge. A popular method to defend against adversarial attacks is to include the perturbed inputs into the training dataset and train a classifier robust to those attacks—this is known as adversarial training [158; 163]. Another defence against adversarial attacks is to detect which inputs are adversarial, see, e.g., [164; 165]. Therefore, when designing an ML algorithm for an application in the real-world or safety-critical functions, we should take defending against adversarial examples into consideration.

Another possible algorithm design choice is the extent to which the ML algorithm is human interpretable. Since end-to-end learned ML algorithms are opaque to the users and engineers, when they make a mistake, it is challenging to understand exactly why it happened. We discuss the field of Explainable Artificial Intelligence (XAI) in terms of ML safety and for the validation of ML-based safety-critical functions in Section 2.1 and Section 2.4, respectively. The methods discussed in those sections, show the possibility of designing interpretable ML algorithms which can, in turn, strengthen the overall safety argument.

Overall, these algorithm design choices come with various trade-offs, e.g., there is a trade-off between the performance of ML-based functions and their interpretability or their robustness to adversarial or to out-of-distributional samples. These trade-offs should be taken into consideration when comparing the performance of ML-based functions. In this chapter, we introduce an interpretable algorithm design and allows an engineer to choose the function parameters depending on what performance measure they prioritise.

6.2 An Interpretable Lane Change Detector Algorithm

The proposed interpretable LCD algorithm considers the Time to Lane Change (TTLC) prediction use-case introduced in Section 3.5, i.e., we track surrounding vehicles over time and detect when they are going to change lanes. Since the input data are multi-variate time-series data (see Definition 2), we design the interpretable LCD algorithm to take this into account. The algorithm detects a lane change depending on the reconstruction errors of three independent ML-based anomaly detection functions—each trained on a specific driving manoeuvre.

First, we discuss a possible method to detect anomalies—also known as outliers—in time-series data. An overview of other possible (ML-based) methods for anomaly detection can be found in [166; 167]. The DAE [168; 99] is a popular anomaly detection method (see, e.g., [166]) because it is trained in an unsupervised manner, i.e., it detects anomalies based on the intrinsic properties of the data without the need of labels. A DAE attempts to reconstruct the input signal whilst representing the essential information in a smaller dimensional space, i.e., it aims to minimise the reconstruction error between input and output signals with a smaller dimensional latent space between them (*cf.* Section 3.3.5). After a DAE is trained on a specific distribution, it should only be able to reconstruct samples from the same distribution. Samples from other distributions will be incorrectly reconstructed, and the reconstruction error will be relatively large (compared to in-distribution samples). This is the basis of a DAE anomaly detector—when the reconstruction error is small, the sample can be categorised to the training data distribution and when it is large, the sample can be flagged as anomalous.

In [169], a time-series anomaly detection algorithm is introduced using a Convolutional Neural Network (CNN) with an encoder/decoder architecture, similar to the structure of a DAE. The algorithm is able to detect and classify different types of anomalous signals in streaming time-series data. In [170], an anomaly detection algorithm is introduced to detect abnormal trajectories of road participants at various intersections. The authors train a DAE on sequential data recorded at these intersections. They then classify unseen time-series samples as being *normal* or *abnormal* depending on the reconstruction error at the output of the DAE, i.e., if the reconstruction error is larger than a pre-defined threshold, the sample is classified as an anomaly (*abnormal*).

Researchers have also employed DAE algorithms in the related tasks of: reconstructing, predicting, and generating vehicle trajectories in highway scenarios. The authors of [171] introduce an algorithm to predict the future trajectory of vehicles using Recurrent Neural Network (RNN)-based DAEs, which also takes the drivers'

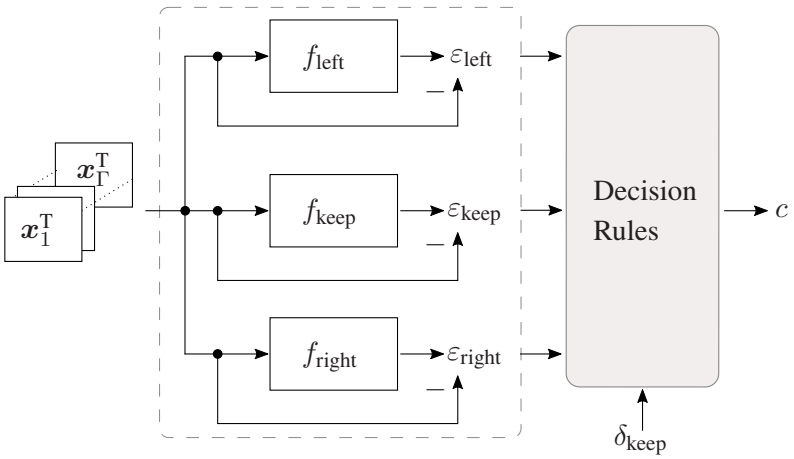


Figure 6.1: An overview of the interpretable LCD algorithm demonstrating a decoupling of the ML-based reconstruction (dashed box) and the explicit decision rules. Each function f is an independently trained ML method used to reconstruct the time-series signals $\boldsymbol{x}_1^{\text{T}}, \ldots, \boldsymbol{x}_\Gamma^{\text{T}}$. An interpretable decision rule-set outputs the decision c based on the reconstruction errors.

lane change intention into consideration. They show that this DAE architecture can predict the lateral and longitudinal position of vehicles up to 5 s into the future. In [50], the authors introduce multiple unsupervised ML methods to generate lane change trajectories. They show that a DAE algorithm is able to generate new realistic trajectories by varying the latent space representations. This work was extended in [51] using Bézier curves to generate smoother trajectories, but the underlying DAE architecture remains the same. Therefore, we see that a DAE is able to successfully learn the underlying distribution of multi-variate time-series data and even reconstruct driving trajectories.

With these results in mind, we propose an interpretable LCD algorithm, depicted in Fig. 6.1. It takes advantage of the performance of DAEs to reconstruct driving trajectories and detect anomalies. The algorithm classifies the lane changes based on the reconstruction errors at the output of three independently trained DAEs (depicted in the dashed box in Fig. 6.1): one to reconstruct left lane changes (f_{left}); one to reconstruct right lane changes (f_{right}); and one to reconstruct lane keeping (f_{keep}). Since these DAEs are trained on independent datasets, we argue that they should only be able to reconstruct samples from their respective distributions. The main idea is: if, e.g., we have a left lane change sample, we expect a small reconstruction error at the output of f_{left} (since this was the distribution f_{left} is trained on). At the same

time, we expect both f_{right} and f_{keep} to struggle to reconstruct this sample, and they would have larger reconstruction errors. Therefore, we can define a threshold for the reconstruction errors to determine when a sample belongs to the driving manoeuvres the DAE was trained on. For the previous example, the reconstruction error of f_{left} would be smaller than its corresponding threshold, and the errors of f_{right} and f_{keep} would be larger than their thresholds.

We can then define a rule-set based on these thresholds to detect upcoming driving manoeuvres. By pre-determining these thresholds, an engineer is able to trade off the classification performance of the detection algorithm with a reliable detection time, i.e., whether the algorithm can reliably detect a lane change earlier or later (*cf.* Definition 5). Furthermore, the proposed LCD algorithm is intrinsically interpretable (*cf.* Section 2.1). We investigate this in more detail in Section 6.3.4.

6.2.1 Dataset Generation

As training data for the DAEs, we use the realistic driving data summarised in the highD dataset [7] (*cf.* Section 3.5.1). First, we extract the driving manoeuvrers, i.e., left lane changes, right lane changes, and lane keeping, from the highD dataset; These contain the multi-variate time-series signals the DAEs should reconstruct. For the lane keeping data, a scenario is defined where the observed vehicle remains in its lane for all time-steps.

We create a training dataset for the DAEs as follows: first, a set of scenarios is created

$$\mathcal{D}'_j = \left\{ \tilde{\boldsymbol{X}}^1, \ldots, \tilde{\boldsymbol{X}}^{M_j} \right\}, \tag{6.1}$$

with the dataset label $j \in \{\text{left, right, keep}\}$ and each scenario is defined as

$$\tilde{\boldsymbol{X}} = \left[\bar{\boldsymbol{x}}[1], \quad \bar{\boldsymbol{x}}[2], \quad \ldots, \quad \bar{\boldsymbol{x}}[N_{\text{lc}}] \right] \in \mathbb{R}^{\Gamma \times N_{\text{lc}}}, \tag{6.2}$$

where the event, e.g., the centre of a vehicle crosses the right lane marking, occurs at time-step N_{lc}. At each time-stamp $n = 1, \ldots, N_{\text{lc}}$, we summarise the Γ attribute signals in a vector $\bar{\boldsymbol{x}}[n] = [x_1[n], x_2[n], \ldots, x_\Gamma[n]]^{\text{T}} \in \mathbb{R}^\Gamma$. We use the input attributes which describe the tracked vehicle's trajectory, since it was shown that these can be reconstructed by the DAE architecture (see [50]). These trajectory input attributes are summarised in Table 6.1.

Once we have a set of scenarios describing different driving manoeuvres—each with a different number of samples (M_j)—we further process the data by passing a

Attribute	Unit	Description
$v_{\text{lat.}}$	$\mathrm{m/s}$	Lateral velocity
$v_{\text{long.}}$	$\mathrm{m/s}$	Longitudinal velocity
$a_{\text{lat.}}$	$\mathrm{m/s^2}$	Lateral acceleration
$a_{\text{long.}}$	$\mathrm{m/s^2}$	Longitudinal acceleration
d_{left}	m	Distance to left lane marking
d_{right}	m	Distance to right marking

Table 6.1: The input attributes used when training the DAEs for the interpretable LCD algorithm (in this case, $\Gamma = 5$).

sliding window over them with a constant window size W [76] (*cf.* Definition 4). This creates a set of sub-sequences of the scenario, which the DAE should reconstruct.

Thus, the dataset for each driving manoeuvre is summarised as

$$\mathcal{D}_j = \left\{ \mathrm{win}\left(\tilde{\boldsymbol{X}}^1; W\right), \ldots, \mathrm{win}\left(\tilde{\boldsymbol{X}}^{M_j}; W\right) \right\}, \tag{6.3}$$

with the dataset label $j \in \{\text{left, right, keep}\}$, and each input sample, after windowing, has the dimension $\bar{\boldsymbol{X}} \in \mathbb{R}^{\Gamma \times W}$. The function win is defined in Definition 4. In total, we have $|\mathcal{D}_j| = M_j(N_{\text{lc}} - W + 1)$ samples for each driving manoeuvre since each scenario is split into $N_{\text{lc}} - W + 1$ sub-sequences.

With a dataset defined for each of the three manoeuvres, we split them into three disjunct sets for: (i) training the DAE ($\mathcal{D}_{\text{train},j}$); (ii) determining the threshold values ($\mathcal{D}_{\text{val},j}$); and (iii) testing the detection performance ($\mathcal{D}_{\text{test},j}$). We separate the dataset into a 70%, 10%, and 20% split of the total data $|\mathcal{D}_j|$, respectively.

An introduction of the DAE architecture is presented in Section 3.3.5. Since we aim to reconstruct the input signals, we use the Mean Squared Error (MSE) error function to train the DAEs (*cf.* Section 3.4.2).

6.2.1.1 Anomaly Detection of Driving Manoeuvres

A main benefit of training a DAE on multi-variate time-series data is the ability to detect anomalous signals (see, e.g., [166; 170]). Since the DAE can reconstruct data which are from the same distribution as the training set, when a signal from a different distribution is input, the reconstruction error at the output will be larger. Thus, by observing the reconstruction error, we can estimate whether the input signal is in-distribution or not.

6.2.2 Interpretable Lane Change Detector Algorithm

In the first step of the proposed LCD algorithm, three DAEs (f_{left}, f_{right}, and f_{keep}) should be trained on representative datasets. The three DAEs are trained on left lane changes, right lane changes, and lane keeping manoeuvres, contained in $\mathcal{D}_{\text{train, left}}$, $\mathcal{D}_{\text{train, right}}$, and $\mathcal{D}_{\text{train, keep}}$, respectively. Once the DAEs have been trained, the weights are frozen, and the DAEs are only used to reconstruct the input, i.e., they are only used for inference.

An overview of the LCD algorithm is depicted in Algorithm 4. For an arbitrary input $\bar{\boldsymbol{X}}$ we first calculate the reconstruction errors ($\varepsilon_{\text{left}}, \varepsilon_{\text{right}}$, and $\varepsilon_{\text{keep}}$) of each DAE (Lines 3, 4, and 5). Next, in Line 6, we calculate the difference δ_{keep} between the current reconstruction error $\varepsilon_{\text{keep}}$ and the reconstruction error from the previous time-stamp $\varepsilon'_{\text{keep}}$. By tracking the change in reconstruction error of f_{keep}, we track the derivative of the reconstruction error. With the given threshold values τ_{left}, τ_{right}, τ_{keep}, and τ_δ (*cf.* Section 6.2.3), the algorithm estimates which class the current window belongs to using explicit decision rules [24, Ch. 4.5]. This highlights the direct interpretability of the LCD algorithm. A left lane change is detected if f_{right} cannot reconstruct the current datum, and either f_{keep} cannot reconstruct it as well or the difference δ_{keep} is large (this is condition (a) in Lines 7 and 9). Simultaneously, f_{left} can reconstruct the datum (see Line 7). A similar rule-set is used to detect right lane changes (see Line 9). If the current datum is not a lane change, the label at the output is $y = \text{K}$ (lane keeping).

A slight modification to the LCD algorithm is to set the condition (a) to `TRUE` for all values in Lines 7 and 9. Thus, a lane change is detected only when the DAE corresponding to a lane change can reconstruct the current input and the DAE of the other lane change manoeuvre cannot reconstruct it. The rest of the LCD algorithm remains unchanged. By only considering the lane change DAEs, we lose some explainability of why the output label was chosen, however, we gain in classification performance. We refer to this version of the algorithm as LCD_{lcs}.

6.2.3 Performance Trade-off: Threshold Determination

An important step in the design of the LCD algorithm is to determine the threshold values (τ_{keep}, τ_{left}, τ_{right}, and τ_δ) to compare the reconstruction losses of different DAEs. These thresholds—as seen in Algorithm 4—not only determine how well the algorithm can classify each manoeuvre, but also how early a lane change is reliably detected. Thus, they can be chosen depending on the engineering requirements—a benefit of designing a fully interpretable ML algorithm. We use the macro-averaged

Algorithm 4 Interpretable Lane Change Detector (LCD) Algorithm

Inputs: Multi-variate time-series datum: $\bar{\boldsymbol{X}} \in \mathbb{R}^{\Gamma \times W}$, reconstruction error from previous time-stamp: $\varepsilon'_{\text{keep}}$
Parameters: Threshold values τ_{left},τ_{right}, τ_{keep}, and τ_{δ}
$\varepsilon_{\text{left}} \leftarrow \|\bar{\boldsymbol{X}} - f_{\text{left}}(\bar{\boldsymbol{X}})\|_F^2$
$\varepsilon_{\text{right}} \leftarrow \|\bar{\boldsymbol{X}} - f_{\text{right}}(\bar{\boldsymbol{X}})\|_F^2$
$\varepsilon_{\text{keep}} \leftarrow \|\bar{\boldsymbol{X}} - f_{\text{keep}}(\bar{\boldsymbol{X}})\|_F^2$
$\delta_{\text{keep}} \leftarrow \varepsilon_{\text{keep}} - \varepsilon'_{\text{keep}}$
if $\underbrace{(\varepsilon_{\text{keep}} \geq \tau_{\text{keep}} \textbf{ or } \delta_{\text{keep}} \geq \tau_{\delta})}_{(a)}$ **and** $(\varepsilon_{\text{left}} < \tau_{\text{left}})$ **and** $(\varepsilon_{\text{right}} \geq \tau_{\text{right}})$ **then**
    $y \leftarrow$ L {Left Lane Change}
else if $\underbrace{(\varepsilon_{\text{keep}} \geq \tau_{\text{keep}} \textbf{ or } \delta_{\text{keep}} \geq \tau_{\delta})}_{(a)}$ **and** $(\varepsilon_{\text{left}} \geq \tau_{\text{left}})$ **and** $(\varepsilon_{\text{right}} < \tau_{\text{right}})$ **then**
    $y \leftarrow$ R {Right Lane Change}
else
    $y \leftarrow$ K {Lane Keeping}
end if
return $y, \varepsilon_{\text{keep}}$

F_1 score [113] as the classification performance measure we aim to maximize (see definition in (4.3)).

To this end, we create an aggregated dataset out of the validation datasets introduced in Section 6.2.1. We label each sample with a label corresponding to the driving manoeuvre, e.g., all samples in $\mathcal{D}_{\text{val, left}}$ are given the label $y = L$, all samples in $\mathcal{D}_{\text{val, right}}$ are given the label $y = R$, and all samples in $\mathcal{D}_{\text{val, keep}}$ are given the label $y = K$. Thus, we define the validation dataset as

$$\mathcal{D}_{\text{val}} = \left\{(\boldsymbol{X}^i, y^i)\right\}_{i=1}^{M_{\text{val}}}, \tag{6.4}$$

where the samples are taken from $\mathcal{D}_{\text{val, left}}$, $\mathcal{D}_{\text{val, right}}$ and $\mathcal{D}_{\text{val, keep}}$. The total number of samples is $M_{\text{val}} = |\mathcal{D}_{\text{val, left}}| + \left|\mathcal{D}_{\text{val, right}}\right| + \left|\mathcal{D}_{\text{val, keep}}\right|$.

On the other hand, we want the LCD algorithm to reliably detect a lane change as early as possible before the lane change actually occurs. Thus, we define a reliable detection time as:

Definition 5. *A lane change detection is considered reliable if the prediction does not change in the time interval between the first prediction and the actual lane change.*

We search for the threshold values $\boldsymbol{\tau} = [\tau_{\text{left}}, \tau_{\text{right}}, \tau_{\text{keep}}, \tau_{\delta}]^{\mathrm{T}} \in \mathbb{R}^4$ which maximise the F_1 score whilst maintaining a reliable early detection on the validation dataset $\mathcal{D}_{\text{val}}$.

Furthermore, we define the set of reconstruction errors of each DAE on the validation dataset as

$$\mathcal{E}_j = \left\{ \|\boldsymbol{X} - f_j(\boldsymbol{X})\|_F^2 \in \mathbb{R} \,:\, \forall \boldsymbol{X} \in \mathcal{D}_{\text{val},j} \right\}, \tag{6.5}$$

with the trained DAEs f_j, the validation dataset for each manoeuvre $\mathcal{D}_{\text{val},j}$, and $j \in \{\text{left, right, keep}\}$. We can now exhaustively search for the optimal thresholds over the range $\tau_j \in [0, \max\{\mathcal{E}_j\}]$. We will explore this further in the following sections.

Alternatively, a method to estimate an appropriate threshold to define an anomalous signal is to estimate the mean and standard deviation of the reconstruction error of each DAE on the validation dataset. This is the method employed in [170]. To this end, we can also set each threshold as

$$\tau_j = \text{mean}\{\mathcal{E}_j\} + 3 \cdot \text{std}\{\mathcal{E}_j\}, \tag{6.6}$$

with the mean and the standard deviation of the reconstruction error values on each validation dataset from (6.5).

6.3 Interpretable Lane Change Detector Algorithm: Results

In this section, we discuss the simulation setup and results we achieved with the LCD algorithm introduced in Section 6.2.2 compared with some ML-based reference algorithms. We show how the LCD algorithm detects driving manoeuvres in an interpretable manner and how the threshold parameters can be chosen by an engineer depending on the requirements.

6.3.1 Simulation Setup

We train each DAE using the training data $\mathcal{D}_{\text{train},j}$ for the respective driving manoeuvre with the MSE loss (*cf.* (3.43) in Section 3.4.2). Each attribute is normalised over all samples and all time-stamps to lie within the range $[-1, +1]$. To train the DAEs, we assume the lane changes occur at time-stamp $N_{\text{lc}} = 100$, i.e., the scenarios before windowing are of dimension $\tilde{\boldsymbol{X}} \in [-1, +1]^{5 \times 100}$ (*cf.* (6.2)). This means the DAEs attempt to reconstruct 4 s before the lane change—this is motivated by the results from [171; 50]. Moreover, we use a window of length $W = 25$ time-stamps which

corresponds to sub-sequences of the time-series data of length 1 s. Thus, the inputs to the DAEs are of dimension $\bar{\boldsymbol{X}} \in [-1, +1]^{5\times 25}$ (*cf.* (6.3)).

We train each DAE for 200 epochs using the Adam optimiser [102] with mini-batches of size 200 and an initial learning rate of $\eta = 0.0005$. We use the same architecture for each DAE, with a symmetric encoder and decoder design. The exact parameters and architectures are summarised in Appendix C.2. The latent space has 10 dimensions.

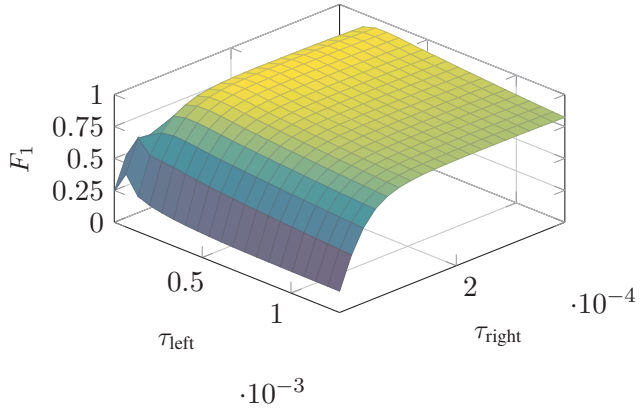
As reference algorithms, we use two RNN-based architectures from the literature: one with a Long Term Short Term Memory (LSTM) cell [10] and one with a Gated Recurrent Unit (GRU) cell [12]. To train the reference algorithms, we use the same training data which we trained the DAEs on, i.e., $\mathcal{D}_{\text{train, left}}$, $\mathcal{D}_{\text{train, right}}$ and $\mathcal{D}_{\text{train, keep}}$.[2] Furthermore, we label each sample with the corresponding driving manoeuvre. Given this training set, we can train the networks using the standard cross-entropy classification loss (*cf.* Section 3.4.1). The reference algorithms are trained with the same hyper-parameters as the DAEs, i.e., the same number of epochs, mini-batch size, and learning rate. The RNN architectures were taken from [10] and [12]. For the CNN architecture, we take the encoder design (*cf.* Appendix C.2) and change the output to 3 neurons—one for each class.

6.3.2 Lane Change Detector Algorithm Threshold Determination

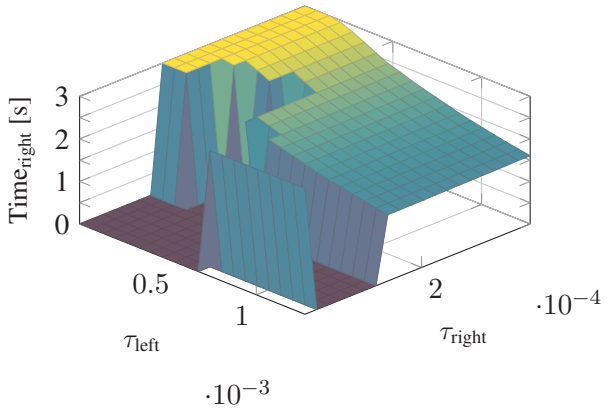
Since the LCD algorithm is designed to detect lane changes in an interpretable manner—only based on a simple rule-set—, we discuss that an engineer can choose the threshold values to achieve the desired performance. As introduced in Section 6.2.3, we can also calculate the threshold values based on the standard deviations on the validation datasets. This is the simplest method to determine the thresholds and they should give an initial indication of how the LCD algorithm works based on detecting anomalies. Using this method, we obtain the threshold values of $\tau_{\text{keep, std}} = 5.93{\cdot}10^{-5}$, $\tau_{\text{left, std}} = 3.25{\cdot}10^{-4}$, $\tau_{\text{right, std}} = 1.10{\cdot}10^{-4}$, and $\tau_{\delta,\text{std}} = -7.50{\cdot}10^{-5}$. Note, τ_δ is calculated on the differences of $\varepsilon_{\text{keep}}$ values for each sub-sequence. We denote the LCD algorithm with these threshold values as LCD_{std}.

We can also exhaustively search for the threshold values for the interpretable LCD algorithm which give us a desired functionality. To this end, we keep the thresholds relating to the keep DAE, such that $\tau_{\text{keep, exh}} = \tau_{\text{keep, std}}$ and $\tau_{\delta,\text{exh}} = \tau_{\delta,\text{std}}$. Next, we can search over the possible thresholds for the left and the right DAEs which give us

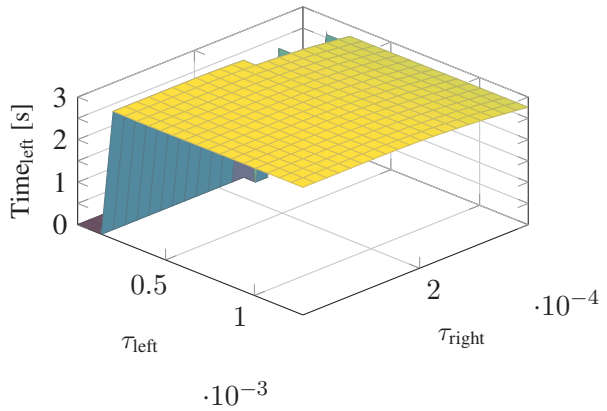
[2]Note, the classification task is defined with windowed scenarios, so the resulting classifiers are slightly different than those presented in the previous chapters.



(a) F_1 score.



(b) Average reliable detection time for right lane changes.



(c) Average reliable detection time for left lane changes.

Figure 6.2: Threshold choice for the LCD algorithm showing the trade-off between a high F_1 score and a reliable early detection.

a desired function performance. To this end, we plot the macro-averaged F_1 score (*cf.* (4.3)), the average reliable detection time for left and for right lane changes (with a reliable lane change detection from Definition 5) on the validation dataset in Fig. 6.2. In the plot, we consider the average reliable detection times only when at least 90% of the validation data are detected reliably. In Fig. 6.2a, we see that a threshold value $\tau_{\text{left}} = 2.72{\cdot}10^{-4}$ and $\tau_{\text{right}} \in (1.84{\cdot}10^{-4}, 2.81{\cdot}10^{-4})$ achieves an F_1 score of over 94%. For a reliable detection time of over 2.90 s for right lane changes we require $\tau_{\text{left}} \in (4.69{\cdot}10^{-6}, 2.05{\cdot}10^{-4})$ and $\tau_{\text{right}} \in (2.03{\cdot}10^{-4}, 3.79{\cdot}10^{-4})$ in Fig. 6.2b. An average reliable detection time of at least 2.90 s for left lane changes is achieved for $\tau_{\text{left}} \geq 2.05{\cdot}10^{-4}$ and $\tau_{\text{right}} \leq 2.23{\cdot}10^{-4}$ as seen in Fig. 6.2c. With these results in mind, we choose the left and the right threshold values to be $\tau_{\text{left, exh}} = 2.72{\cdot}10^{-4}$ and $\tau_{\text{right, exh}} = 2.23{\cdot}10^{-4}$, respectively. We denote the algorithm with these threshold values as LCD_{exh}. We determine the same values for τ_{left} and τ_{right} if we consider the LCD_{lcs} algorithm (only considering the lane change DAEs). Therefore, we use the same threshold values for that algorithm, too.

6.3.3 Lane Change Detection Results

Now, we discuss the reliable lane change detection achieved by the different LCD algorithms on the test dataset.[3] We observe in Table 6.2a that the purely ML-based methods achieve better classification performances than the interpretable LCD algorithms. Interestingly, the CNN-based classifier slightly outperforms the RNN-based algorithms on this lane change detection task. Since the encoders of the DAEs use the same CNN architecture, the classification results support the argument that a CNN with 1-D filters is able to extract discriminative features. The interpretable LCD algorithm achieves a classification performance to within 4.26% and 4.13% of the black-box ML-based methods in terms of classification accuracy and F_1 score, respectively. We observe that the LCD_{exh} and LCD_{lcs} algorithms achieve a better performance compared with the LCD_{std} algorithm in terms of the classification metrics. These results indicate the benefit of determining the threshold values manually. These classification results show that the proposed LCD algorithm is able to classify the different driving manoeuvres whilst maintaining interpretability.

In Table 6.2b, we observe the average reliable detection time results for the different algorithms. Again, the LCD algorithms almost achieve the same performance as the black-box ML-based classification algorithms; the CNN-based classifier achieves the

[3]Note, the test dataset is labelled similarly to the validation dataset such that we can calculate the classification performance metrics on unseen data. Furthermore, the test dataset remained unseen by all of the LCD and reference algorithms during training and threshold determination.

Alg.	Acc. [%]	F_1 [%]	Precision [%]	Recall [%]
LSTM [10]	99.62	99.62	99.59	99.66
GRU [12]	99.58	99.59	99.56	99.62
CNN	99.68	99.68	99.63	99.74
LCD_{std}	88.98	89.30	89.26	90.11
LCD_{exh}	94.87	95.02	95.26	94.81
LCD_{lcs}	95.32	95.46	95.57	95.37

(a) Lane change detection classification results.

Alg.	$\text{Time}_{\text{left}}$[s]	Rel. [%]	$\text{Time}_{\text{right}}$[s]	Rel. [%]
LSTM [10]	3.04	99.77	3.03	99.60
GRU [12]	3.04	99.32	3.03	99.60
CNN	3.04	100.00	3.04	100.00
LCD_{std}	3.00	96.38	2.75	80.08
LCD_{exh}	2.97	95.02	2.84	90.54
LCD_{lcs}	3.00	94.80	2.87	90.34

(b) The average reliable detection time results, including a percentage of reliable (Rel.) detections on the test dataset, for left and for right lane changes.

Table 6.2: Classification and reliable detection time results of the interpretable LCD algorithms and the reference algorithms on the test dataset with $N_{\text{lc}} = 100$.

best overall performance. On average, the LCD algorithms can reliably predict a lane change almost 3.00 s before left lane changes, only 0.04 s away from the black-box DNN architectures. Their performance of detecting right lane changes is slightly worse than left lane changes—both in terms of the time before the lane change and the reliability. We see that the LCD_{std} algorithm performs equally well at reliably detecting left lane changes; its performance of detecting right lane changes is slightly worse. Recall, we chose the threshold values τ_{left} and τ_{right} to achieve a reliability of at least 90% on the validation data. As we see in Section 6.3.4, we can explain why the LCD algorithms detect lane changes slightly later than the reference algorithms. This is an advantage of having an interpretable ML-based HAD function.

Overall, the simulation results indicate that the proposed LCD algorithm can reliably detect lane changes; however, it shows a (slight) performance degradation compared to black-box DNN architectures. It is known, see, e.g., [22; 122], that an

increase in interpretability of ML-based classifiers comes at the cost of classification performance. This is the price of the interpretability of the classification. Moreover, we see the direct trade-off between explainability and classification performance: The LCD_{lcs} algorithm ignores the lane keeping DAE and is missing a further reason for detecting a lane change, however, it also shows better classification performance compared with the LCD_{exh} algorithm with a lower reliability of the detected lane changes.

6.3.4 Interpretability of the Lane Change Detector Algorithm

In the previous sub-section, we saw that the LCD algorithm is able to classify different driving manoeuvres almost as well as state-of-the-art, black-box ML algorithms. In general, the reasons why ML algorithms make certain classifications is opaque and not directly interpretable.[4] Now, we demonstrate how the proposed LCD algorithm is interpretable. To this end, we take three scenarios—one for each driving manoeuvre—from the test dataset and assume $N_{\text{lc}} = 200$. We use this value of N_{lc} to test the generalisability of the ML-based HAD funcitons. Therefore, we pass the algorithms more windows than they were trained on, however, we see that all of the algorithms are able to generalise to these new samples. Then, we pass each scenario through the window function (see Definition 4), and classify each sample sequentially.

The results of passing each windowed scenario through the classifiers are depicted in Fig. 6.3. The first driving manoeuvre—a vehicle changing lanes to the right—is depicted between 1 s and 8 s. We see that the LCD_{exh} algorithm outputs the label $c_{\text{LCD}_{\text{exh}}} = \text{R}$, only once all conditions in Line 9 of Algorithm 4 are fulfilled, i.e., f_{left} detects an anomalous signal, the difference between reconstructions of the DAE for lane keeping is large, and f_{right} is able to reconstruct the signal. Since the LCD_{lcs} algorithm shares the same threshold values, it also detects a right lane change at this time-stamp. The classification occurs 4.2 s before the right lane change for the given threshold values. Moreover, the reconstruction error at the output of f_{right} remains small and the reconstruction errors of f_{left} and f_{keep} increase after the lane change is detected. We observe in the top plot that the GRU-based method initially predicts a right lane change before the LCD_{exh} algorithm, but then shows an erroneous label ($c_{\text{GRU-[12]}} = \text{K}$) after initially classifying the windows as a lane change to the right. This would not be considered a reliable detection as per Definition 5. The reason for this change in decision is not obvious nor directly explained by the RNN architecture.

[4] It is possible to use post-hoc methods to try and explain why an ML algorithm made a certain decision, e.g., using heatmapping methods [28; 29]. However, this is not an inherent part of the algorithm design.

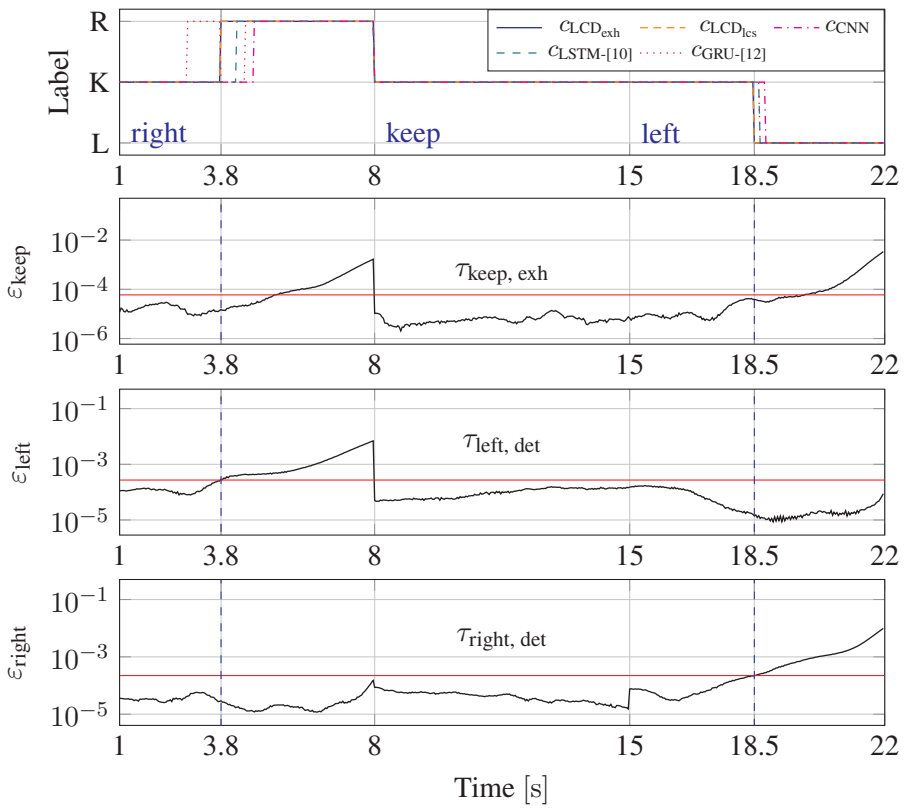


Figure 6.3: Classification output of the LCD and different ML algorithms for different driving manoeuvres, illustrating the interpretability of the LCD algorithm.

Moreover, we observe that the interpretable LCD algorithms detect the right lane change before both the LSTM- and CNN-based architectures.

Between 8 s and 15 s, all four classifiers correctly detect that the vehicle remains in its lane (c = K). From the reconstruction error plots, we observe that all three DAEs are able to reconstruct the samples, with a reconstruction error smaller than the pre-determined threshold values. This makes sense, because all DAEs are trained on windows far before the lane change, where the vehicle stays in its lane. Finally, a scenario where the vehicle changes lanes to the left is depicted between 15 s and 22 s. Again, by observing the reconstruction errors, we can directly explain why the LCD algorithm classified the sample as a left lane change. It is interesting to note that the DAE for right lane change manoeuvres, f_{right}, detected an anomalous signal earlier than the DAE for lane keeping. The lane change is correctly classified 3.5 s before it

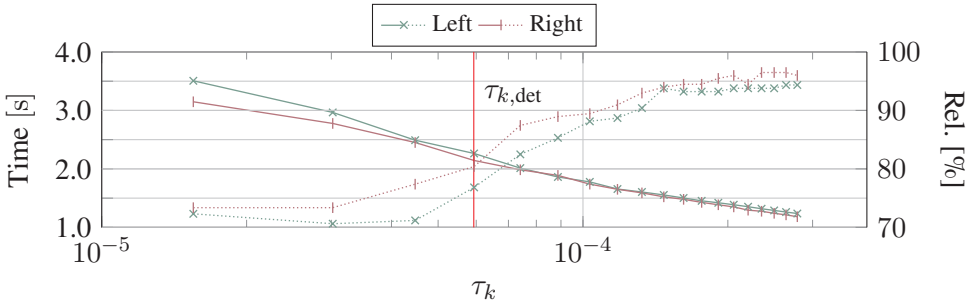


Figure 6.4: Average reliable detection time (solid lines) and percentage of reliable detections (dotted lines) plotted against the threshold τ_k. Left lane changes indicated by: $\times$; right lane changes indicated by: $|$.

occurred, which was earlier than the other black-box ML algorithms.

The results not only highlight the interpretability of the LCD algorithm, but they also show that the reconstruction error at the output of each DAE is independent for different inputs. Moreover, we demonstrate how one can understand and explain the detections made by the LCD algorithm. We also see why the LCD algorithms have a worse classification performance: the classification decision is solely based on the three reconstruction errors and since the errors are not large enough until close to the driving manoeuvre, the classification output is delayed.

6.3.5 Performance Trade-off: Early vs. Reliable Detections

The results thus far highlight the interpretability of the LCD algorithm and they show that the performance is comparable to purely ML-based methods. Now, we investigate the performance trade-off between early and reliable detections by varying the threshold τ_k for the LCD_{exh} algorithm. We use a test dataset with $N_{\text{lc}} = 200$ to investigate the generalisability of the LCD algorithm.

The LSTM [10] method achieves the best performance with an average detection time of $\text{Time}_{\text{left}}$ of 3.78 s and $\text{Time}_{\text{right}}$ of 3.53 s with a reliability of 97.06% and 96.58%, respectively. Although this performance is good, it cannot directly be improved or adapted to take the engineering requirements into account, unlike the proposed LCD algorithm.

In Fig. 6.4, we show the trade-off between an early average reliable detection time and the reliability estimate depending on the threshold value τ_k. We observe that with a small threshold value, we achieve an average reliable detection time of 3.51 s

and 3.15 s for left and right lane changes, respectively. However, this comes at the cost of less reliable detections of around 73.00%. On the other hand, if we make τ_k large enough, we observe a reliability estimation of 94.40% and 96.00%, which comes at the cost of a smaller average detection time. This smaller average detection time can be seen as the cost of a more reliable detection and vice versa. Furthermore, the value $\tau_{k,\text{exh}}$, which we use in the algorithms, is near the intersection point of the curves. These results not only emphasise the benefit of a fully interpretable LCD algorithm, but they also show how the threshold values can be chosen to achieve a desired performance.

6.4 Algorithm Design based Safety Argument: Summary

In this chapter, we consider the complete algorithm design of an ML-based HAD function. To this end, we introduce an interpretable LCD algorithm, based on the reconstruction error of three independent DAEs. The LCD algorithm is parametrised by threshold values which affect the performance. We demonstrate how an engineer can choose appropriate threshold values for the task at hand. The performance capability of the LCD algorithm is investigated on realistic highway driving data. We see that the interpretable algorithm performs almost as well as black-box ML methods. Moreover, we highlight the inherent interpretability of the proposed LCD algorithm.

The LCD algorithm we introduce in this chapter can directly be used in a safety-critical HAD function. Since it is interpretable, an engineer can directly argue for safety as the outputs and the reasoning for the classification can be understood; this allows for the direct validation of the LCD algorithm. On top of that, the LCD algorithm can use otherwise validated anomaly detectors as the DNN architectures. This would further strengthen a safety argument based on an interpretable algorithm design.

7 Conclusion and Outlook

7.1 Conclusion

In this dissertation, we address the challenge of validating Machine Learning (ML)-based Highly Automated Driving (HAD) functions. We start with an investigation of the current proposals on how to ensure the safety of ML algorithms. We see that many validation proposals are built upon ML interpretability methods or on abstracting rules and concepts from Deep Neural Networks (DNNs). Building a safety case is one approach to validate a safety-critical ML-based HAD function, whereby we require evidence to support such a case.

In Chapter 4, we introduce methods to investigate and encourage clustering of the feature embeddings of different DNN architectures. We observe that merely relying on the classification performance, e.g., the classification accuracy, of an ML method is insufficient to argue for safety as many methods can show similar results. However, when investigating the feature embeddings, we are able to identify differences between them. On top of that, we introduce a method to encourage a k-means friendly space in the penultimate layer of a DNN which—both qualitatively and quantitatively—improves the clustering of the feature embeddings. Finally, we also introduce a method to reject certain classifications at the output of the DNN, which are inconsistent with the feature embeddings of the training data.

As ML-based HAD functions are data-driven functions, the datasets used to train them are of utmost importance. In Chapter 5, we discuss the challenge of distributional shifts when considering highway driving datasets. We analyse two public datasets and show that the distributional shifts between them negatively affect ML algorithms trained on these data. To address these distributional shifts, we introduce a fine-tuning method taking the feature embeddings in the source and in the target domains into consideration to tackle the trade-off between forgetting the source distribution and learning the target distribution.

Finally, in Chapter 6, we discuss the challenge of designing a HAD function taking ML interpretability into consideration (*cf.* Section 2.1). To this end, we introduce an interpretable Lane Change Detector (LCD) algorithm which classifies lane changes

based on the reconstruction error of three independently trained Deep Autoencoders (DAEs). We demonstrate that the LCD algorithm is interpretable with thresholds that can be chosen by an engineer taking the engineering requirements into account.

In this dissertation, we introduce methods addressing different parts of ML algorithms: from the data used to train them to the overall algorithm design. These methods can be used individually or complementarily to validate various aspects of ML-based HAD functions or, alternatively, as evidence for a safety case. We argue that a single validation method is unlikely to cover all aspects of ML-based HAD functions, as they are data-driven functions without clear requirements. Therefore, multiple validation methods are required to satisfactorily validate these functions. The more aspects of the ML algorithm a validation safety argument considers, the stronger the overall safety argument can be. We call this approach: *Validation by Diversity*.

7.2 Future Research Directions

In the following, we give an outlook on future research directions building upon the contributions in Chapter 4, Chapter 5, and Chapter 6.

7.2.1 Feature Embeddings

Developing a validation safety argument based on feature embeddings can be extended in various ways. We can further investigate the clusters of feature embeddings from different DNN architectures to quantify which training samples are clustered together by which architectures. If the clusters between architectures are different, an ensemble of various architectures could be used to ensure a diversity of feature embeddings. The feature embeddings could additionally be investigated using self-supervised ML methods [172; 173]. In this case, the feature embeddings created by the different DNN architectures can be compared to those created using supervised learning. Methods to compare the feature embeddings (or representations) of different architectures have been studied in, e.g., [174; 175].

Moreover, the rejected samples from each DNN architecture could also give an insight into the feature embedding capabilities of the different DNN architectures. We could further investigate which samples are rejected to gain an understanding about why they were falsely classified—they could also be incorrectly labelled samples. Another challenge of using ML algorithms in safety-critical applications is confidence calibration [176]. The proposed classification rejection method could be adapted to estimate the confidence of the DNN. We can subsequently reject samples where the DNN's confidence lies below a certain threshold.

7.2.2 Dataset Distributions

We argue that an integral part in creating a validation safety argument for ML-based HAD functions is based on the distribution of the training data (as discussed in Section 2.4). An early detection of a distributional shifts is paramount. To extend the distributional shift analysis, we could further improve the distributional shift detection, e.g., using signature kernels in the Maximum Mean Discrepancy (MMD) [177]. Another approach to detect out-of-distribution samples would be to train an ML algorithm to learn the data distribution and estimate the likelihood that a given input scenario belongs to the training distribution. If this likelihood is below a certain threshold, we can update the confidence of the DNN or reject the sample all together.

On top of that, the challenge remains to robustify ML-based HADfunctions against distributional shifts. Thus, we can further investigate continual learning methods [132] or fine-tuning methods within the context of ML-based HAD functions. Since there is a heavy-tail of scenarios which can occur [178], it is important that these HAD functions do not forget previously learned distributions and generalise to *black swan* events.

7.2.3 Algorithm Designs

Finally, we argue that a validation safety argument can be based on the algorithm design itself. The interpretable LCD algorithm can be extended by further optimising the anomaly detection, e.g., not only using the output reconstruction error but also taking the activations in the hidden-layers into account when detecting anomalous signals (see, e.g., [179]). We can also use a mixture-of-experts design (as seen in [57]) which splits the input domain into sub-domains where an *expert* is trained on each sub-domain. For the LCD algorithm, this could mean creating training datasets containing scenarios at different times before the lane change, e.g., one second, two seconds, etc., and then training DAEs on each of these datasets. Then, we could predict the lane change depending on which of the *expert* DAEs detects anomalous signals.

Moreover, the interpretability of the LCD algorithm can be extended if we consider the reconstruction error of each signal individually. Thus, potentially enabling the LCD algorithm to also explain why a lane change is imminent, e.g., the longitudinal acceleration is typical for a left lane change and not for a right lane change. On top of that, the interpretability of the DAEs can be improved by using instance-based influence methods, see, e.g., [180; 181]. These methods allow us to understand which training samples most influence the reconstruction of the current input sample.

Feature Embeddings: Supplementary Results

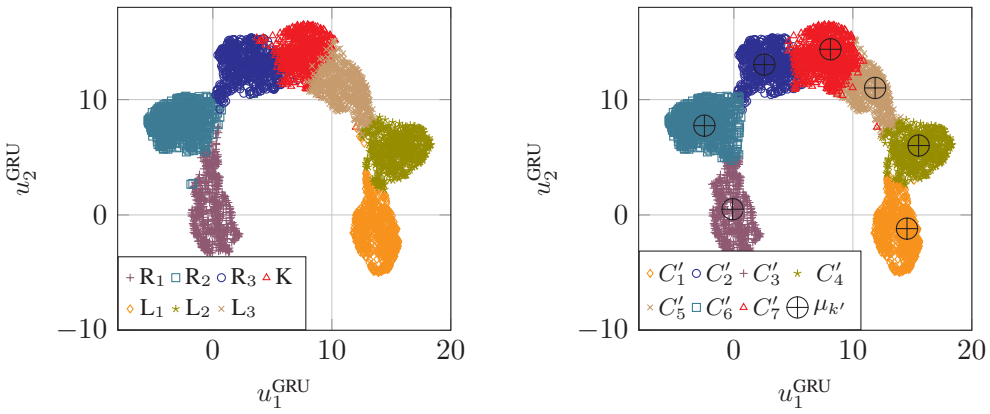
A

A.1 Additional Feature Embedding Visualisations

In this section, we investigate the Uniform Manifold Approximation and Projection (UMAP) representations of two further Deep Neural Network (DNN) architectures introduced in Chapter 4, as this is an important step in the proposed feature validation safety argumentation method. We investigate the feature embeddings of the Gated Recurrent Unit (GRU)-[12] and the Convolutional Neural Network (CNN)-II Multi-Channel (MC) architectures. We choose these architectures because the GRU-[12] architecture showed the highest Adjusted Rand Index (ARI) score in the standard feature embedding space and the CNN-II MC architecture showed the biggest improvement through k-means friendly training (*cf.* Algorithm 1). We take the trained networks from Chapter 4 and visualise the feature embeddings of the test dataset using UMAP.

A.1.1 Standard Feature Embedding Space

First, we plot the UMAP representations in the standard feature embedding space, i.e., the feature embedding space after training the DNN architectures with the standard cross entropy loss (*cf.* Section 3.4.1). In Fig. A.1, we see the UMAP representation of the GRU architecture; We see the embeddings with the true class labels in Fig. A.1a and with the cluster labels in Fig. A.1b. We see that the axes are labelled with u_i^{gru}, $i \in \{1, 2\}$ since these UMAP representation space is calculated especially for this architecture. From now on, the axes of each UMAP plot will be labelled with the corresponding architecture. We see that the safety-critical classes (L_1, L_2, R_1, and R_2) are clustered compactly within their classes and far apart from the opposite driving manoeuvres. The classes L_3, K, and R_3 are not overlapping as we saw for the CNN-I architecture in the standard feature embedding space (*cf.* Fig. 4.3). We see that the cluster labels and the true class labels agree for most classes, which explains the high ARI score. Moreover, the cluster centres in Fig. A.1b are meaningful. Interestingly,



(a) The standard feature embedding space with true test data class labels.

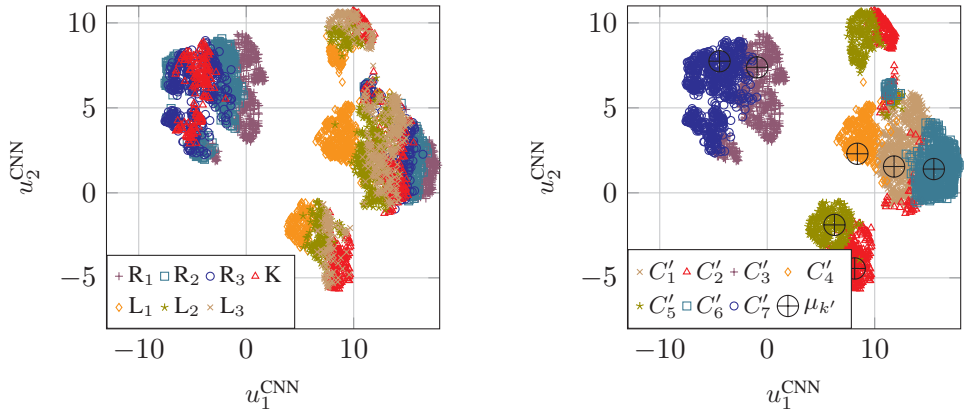
(b) The standard feature embedding space with cluster labels.

Figure A.1: The UMAP representation for the GRU architecture with different labels in the standard feature embedding space.

the feature embeddings look similar to the UMAP representations of the CNN-I architecture *after* k-means friendly training (*cf.* Fig. 4.5).

In Fig. A.2, we plot the UMAP representations of the feature embeddings from the CNN-II MC architecture. This architecture showed the worst ARI score in the standard feature embedding space, implying that the extracted feature embeddings are not well clustered according to the class labels. We see in Fig. A.2a that the classes are overlapping. Moreover, instead of clustering the data according to the true labels, this architecture has extracted the feature embeddings to form three distinct clusters: one of right lane changes (top left), one of left lane changes (top and bottom cluster on the right hand side), and one cluster of all classes (middle cluster on the right). The fact that the cluster of left lane changes is split into two smaller clusters could be an artefact of the UMAP representation—we see that the these have the cluster label C_2' and C_5' in Fig. A.2b. Due to this clustering, the safety-critical classes L_1, L_2, R_1 and R_2 are no longer clustered compactly together, but are spread across two distinct clusters. These qualitative results corroborate the low ARI score calculated in Section 4.4.2.1.

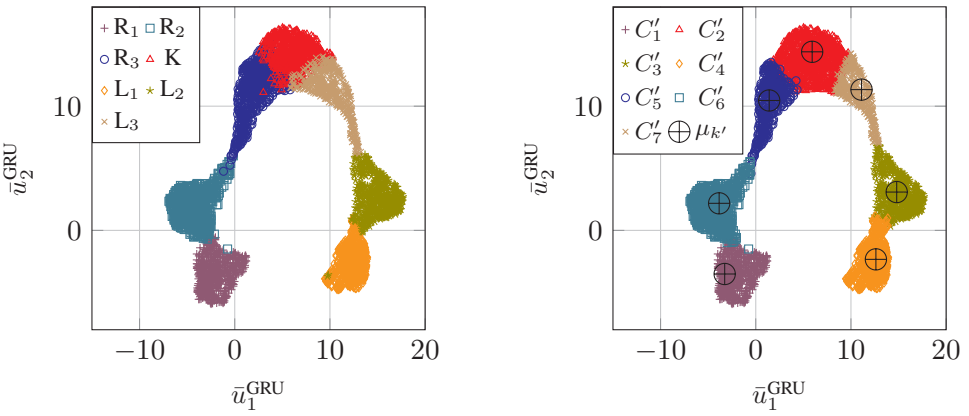
We see from these results that if an architecture shows a high ARI score, it is likely that the feature embeddings are meaningful; In contrast, a low ARI implies that the feature embeddings are not as interpretable—making this architecture less suitable for a safety-critical function.



(a) The standard feature embedding space with true test data class labels.

(b) The standard feature embedding space with cluster labels.

Figure A.2: The UMAP representation for the CNN-II MC architecture with different labels in the standard feature embedding space.



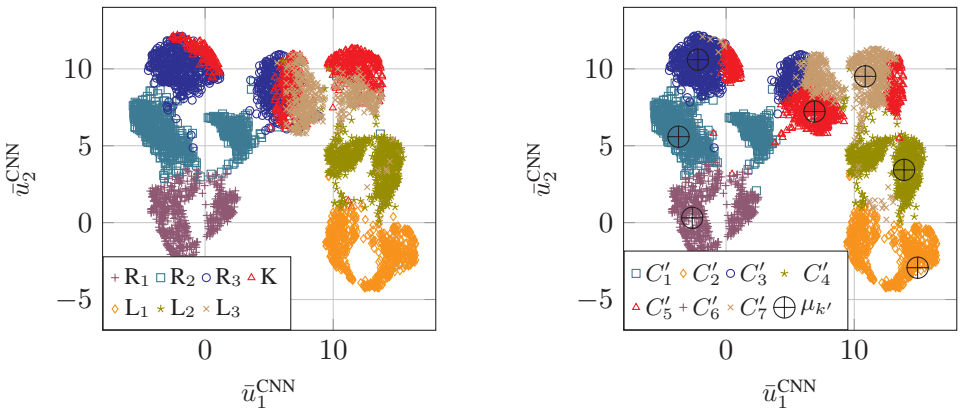
(a) The k-means friendly feature embedding space with true test data class labels.

(b) The k-means friendly feature embedding space with cluster labels.

Figure A.3: The UMAP representation for the GRU architecture with different labels in the k-means friendly feature embedding space.

A.1.2 k-means Friendly Feature Embedding Space

Next, we visualise the feature embeddings of the same two architectures after k-means friendly training (*cf.* Section 4.5). Since these architectures are trained again, we label the axes with a bar to highlight that this is a different UMAP representation. In



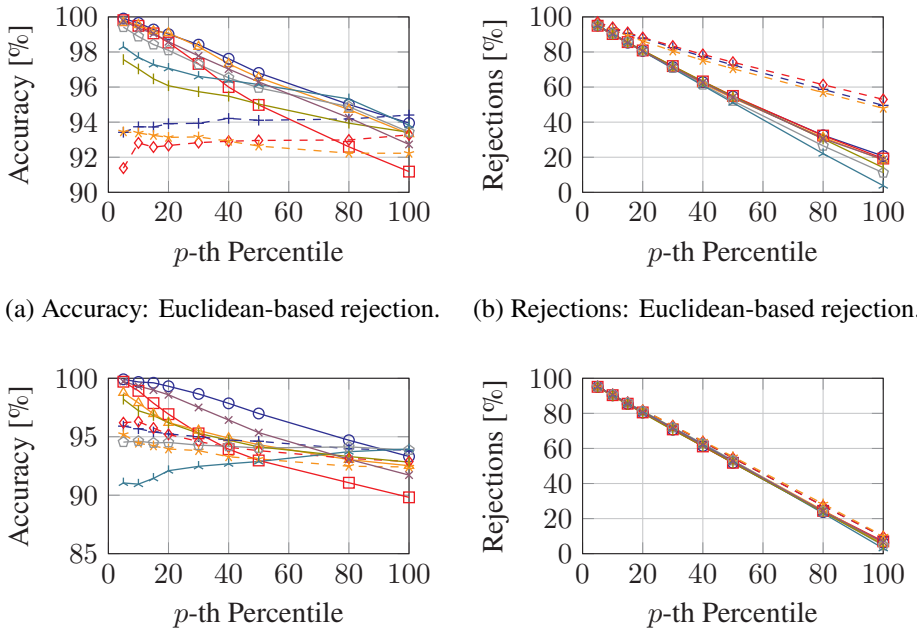
(a) The k-means friendly feature embedding space with true test data class labels.

(b) The k-means friendly feature embedding space with cluster labels.

Figure A.4: The UMAP representation for the CNN-II MC architecture with different labels in the k-means friendly feature embedding space.

Fig. A.3, we see the UMAP representations generated using the GRU-[12] architecture. Comparing Fig. A.3 with Fig. A.1, we see that they are similar, i.e., the classes are clustered compactly together and the safety-critical classes are embedded far apart from each other. The k-means friendly training appears to have made the feature embeddings cluster more compactly if we inspect Fig. A.3b. Moreover, there are fewer individual samples which belong to one class but are embedded within another class (see green L_2 point in the bottom left of the L_1 class). Overall, the feature embedding space generated using k-means friendly training is similar to the standard space for the GRU-[12] architecture. The ARI score after k-means friendly training was slightly lower than without, however, qualitatively, we see that the feature embeddings did not change significantly.

In Fig. A.4, we observe the UMAP representations generated by the CNN-II MC after k-means friendly training. In Section 4.5, we saw that the k-means friendly training helped improve the ARI score of this architecture. This is reflected in the UMAP representations in Fig. A.4. Fig. A.4a shows that the safety-critical classes are now better clustered together, and the right and left lane changes are embedded far apart from each other. Each of the safety-critical classes is clustered close together, however, we see that this architecture created two clusters per class (see class L_1 in Fig. A.4a). The classes L_3, K, and R_3 are still overlapping after k-means friendly training, however, they are overlapping less than in the standard feature embedding



(a) Accuracy: Euclidean-based rejection. (b) Rejections: Euclidean-based rejection.

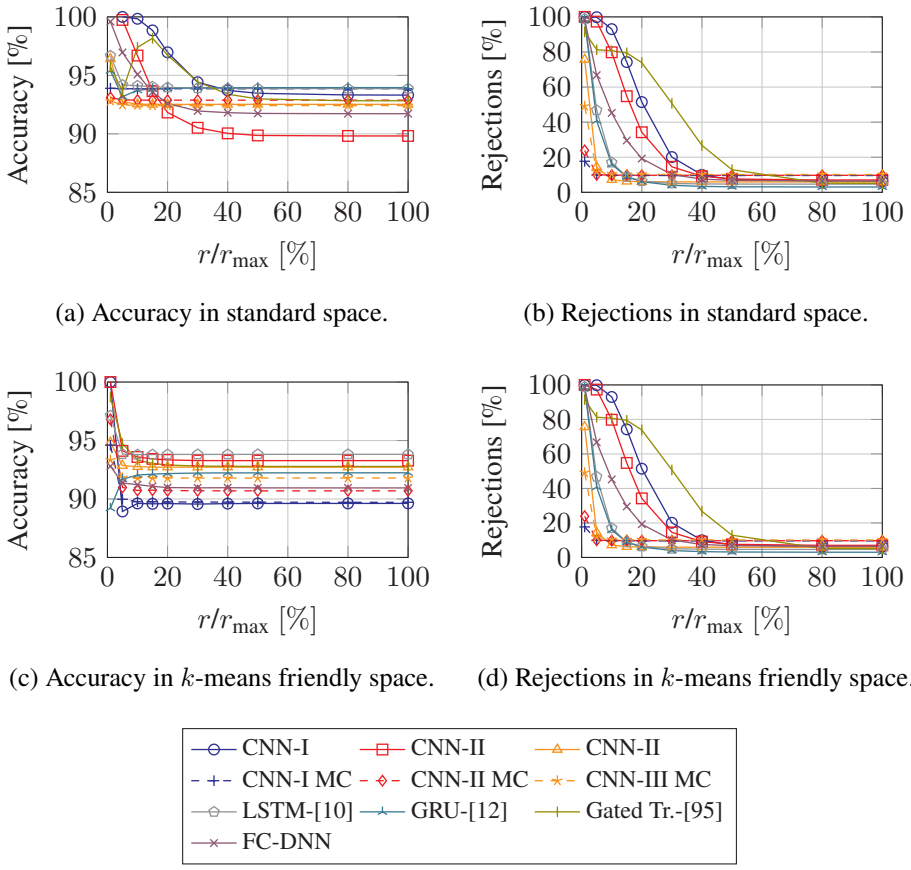
(c) Accuracy: Mahalanobis-based rejection. (d) Rejections: Mahalanobis-based rejection.

Figure A.5: Classification rejection in the standard feature embedding space using the p-th percentile distance as the distance r. The top row of plots (Fig. A.5a and Fig. A.5b) uses the Euclidean-based classification rejection rule; the bottom row of plots (Fig. A.5c and Fig. A.5d) uses the Mahalanobis-based classification rejection rule.

space. Moreover, we can now state that the feature embeddings are more meaningful after k-means friendly training.

A.2 Classification Rejection Rules

In this section, we briefly discuss the classification rejection rule results when varying the distance r (*cf.* Line 6 Algorithm 2). For more details on the simulation setup refer to Section 4.6.4.



(a) Accuracy in standard space.

(b) Rejections in standard space.

(c) Accuracy in k-means friendly space.

(d) Rejections in k-means friendly space.

Figure A.6: Classification rejection using Mahalanobis rejection rule and percentage of max distance as the distance r. The top row of plots (Fig. A.6a and Fig. A.6b) is in the standard feature embedding space; the bottom row of plots (Fig. A.6c and Fig. A.6d) is in the k-means friendly feature embedding space.

In Fig. A.5, we plot the results using both classification rejection rules in the standard feature embedding space. The top row depicts the Euclidean-based classification rejection rule and the bottom row depicts the Mahalanobis-based classification rejection rule. We choose the distance r to be the p-th percentile of the distances to the cluster centres of the training data. Similar to the results in the k-means friendly space, we see that the number of rejections falls when we increase the distance. However, we now observe in Fig. A.5b that the MC CNN architectures reject more samples than the other architectures. This corroborates the results we saw in Section 4.4.2.1: the clustering in the feature embedding space does not align well with the classes since

the average ARI is low. This is reflected in Fig. A.5a, where these architectures show a poor classification accuracy even after rejecting many samples. Using the Mahalanobis-based classification rejection rule, we see in Fig. A.5d that the number of rejections decreases with increasing distance. The average accuracies in Fig. A.5c all decrease monotonically where some architectures are able to classify with almost 100% accuracy when almost all samples are rejected. We observe (again) that the GRU-[12] architecture shows an increase in performance for larger distances (this was also observed in the k-means friendly spaces). This could lead us to conclude that the feature embeddings extracted by this architecture are not well suited for the proposed classification rejection rule.

In Fig. A.6, we plot the results in both feature embedding spaces using the Mahalanobis-based classification rejection rule. In these plots, we vary the distance as a percentage of r_{max}, where r_{max} is the maximum distance to a cluster centre in the training data. These plots allow us to evaluate how well clustered the different feature embedding spaces are. In the top row, we see the results in the standard feature embedding space; In the bottom row, we see the k-means friendly space. By comparing Fig. A.6b and Fig. A.6d, we directly see that the k-means friendly space is better clustered since the number of rejections falls faster. The classification performance in both feature embedding spaces shows a similar behaviour, and all architectures converge to a specific accuracy performance quickly. The architectures with a k-means friendly feature embedding space converge faster than in the standard embedding space, indicating better clustering of the features.

Dataset Distributions: Supplementary Results

B

B.1 Analysing Distributional Shifts in Highway Driving Datasets

In this section, we present additional results of the distributional shift analysis introduced in Chapter 5. In Fig. B.1, we plot the mean attributes of right lane changes from both datasets. We observe that the lateral velocity ($v_{\text{lat.}}$) and the longitudinal accelerations ($a_{\text{long.}}$) have similar distributions in both datasets—though, the standard deviation of the attributes is slightly different. On the other hand, the distribution of the other attributes is noticeably different. For example, the lateral acceleration ($a_{\text{lat.}}$) shows a different trend, where the data from the highD dataset first increase and then decrease before the driving manoeuvre. Moreover, the longitudinal velocities ($v_{\text{long.}}$) are noticeably higher on the highD dataset compared to the NGSIM dataset. As previously discussed, this difference can be explained by the different driving velocities on German vs. US highways. The distribution of the distances to the other vehicles is also different between the two datasets, e.g., the distance to the preceding vehicle to the right ($d_{\text{r, ahead}}$) has a different trend between the two datasets.

In Section 5.3.3, we introduce a statistical hypothesis test based on the MMD test statistic (*cf.* Section 5.3.3.2) to decide whether the sample sets come from the same distribution (the null hypothesis H_0) or whether they come from different distributions (the alternative hypothesis H_1). In Section 5.3.3.4, we reported the two-sample hypothesis test using the MMD test statistic for the case where we compare the distribution of one dataset to itself, e.g., the whole highD dataset is split into disjoint sets and the distribution of these subsets are compared with each other. In those results, we observed that there is no distributional shift within the dataset since the two-sample hypothesis test accepts H_0.

In this section, we compare the distributions of the different driving manoeuvres from each dataset to themselves, i.e., we test whether the distribution of the driving manoeuvres is the same within each dataset. In Fig. B.2, we plot the distribution of the MMD statistic under H_0 and H_1 for the three driving manoeuvres from both datasets.

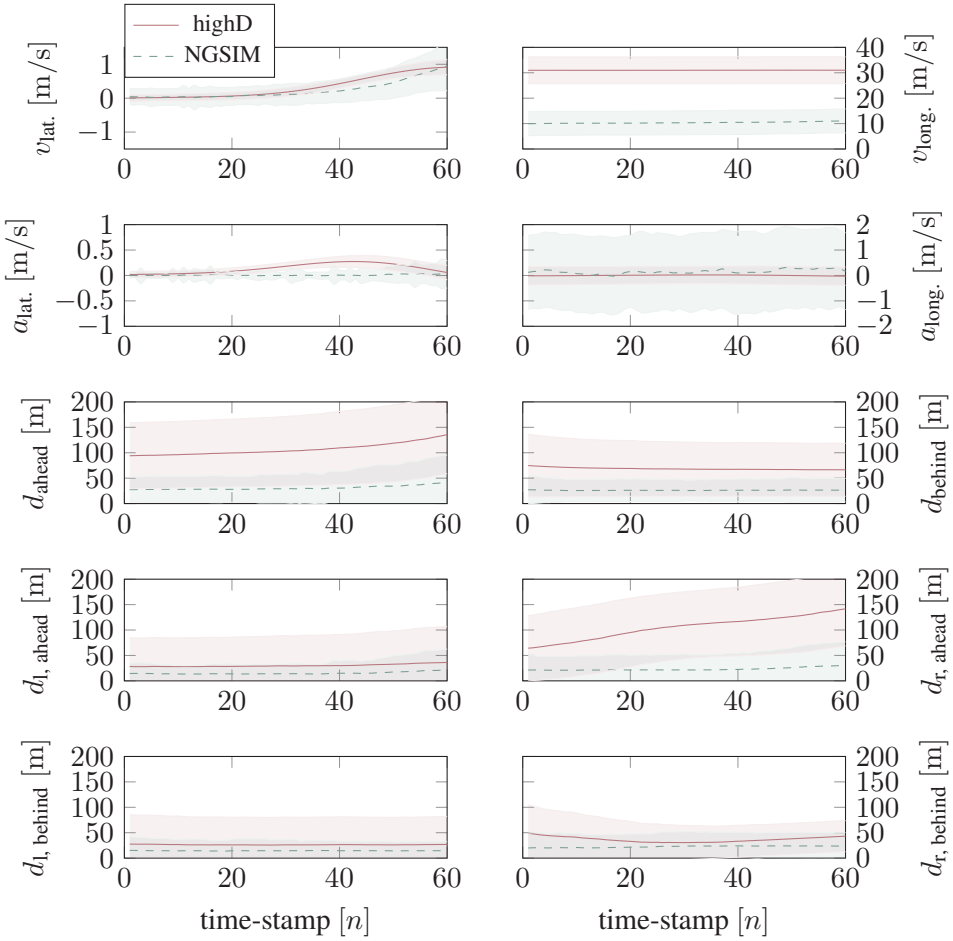


Figure B.1: Qualitative analysis of the datasets by visualizing the average attribute values at each time-stamp for right lane changes. The shaded areas represent one standard deviation away from the mean.

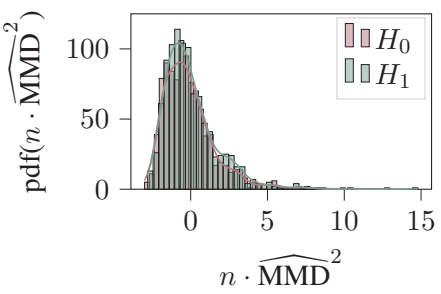
The p-values and True Positive Rates (TPRs) are noted in the plot captions. The left column of plots depict the highD dataset; the right column of plots depict the NGSIM dataset. We observe that the distribution under H_0 and H_1 are indistinguishable for these data, so the two-sample hypothesis test would accept H_0. Thus, we conclude that these data come from the same distribution. On top of that, the p-values are all greater than $\alpha = 0.05$, so the null hypothesis H_0 is accepted.

B.2 Fine-Tuning under Distributional Shifts

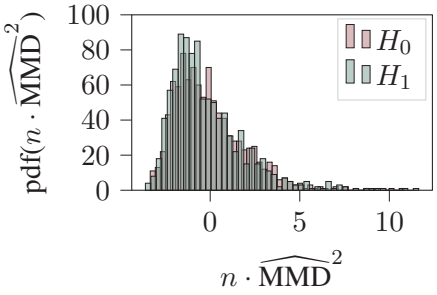
In this section, we demonstrate how the fine-tuning algorithm introduced in Section 5.4 (Algorithm 3) works on another Deep Neural Network (DNN) architecture. To this end, we fine-tune the Gated Recurrent Unit (GRU)-[12] architecture on both the highD dataset and the NGSIM dataset, i.e., we first train on one dataset and fine-tune on the other dataset. We observe in Fig. B.3 that by varying the trade-off parameter ζ, we are able to trade-off between forgetting the source domain and learning the target domain. In Fig. B.3a, we see that the GRU fine-tuned on $\mathcal{D}_{\text{NGSIM, train}}$ can be trained depending on the task at hand, e.g., if it is important not to forget the source domain, we can set $\zeta \to 1$, whereas if we want to learn the target domain, we can set $\zeta = 0$.

Controlling the forgetting when fine-tuning on the highD dataset (when $\mathcal{D}_T = \mathcal{D}_{\text{highD, train}}$) appears to be more challenging as we see in Fig. B.3b. By varying the trade-off parameter, we are able to influence the amount of forgetting, however, not as much as in the other setting. This could be due to the fact that the highD dataset is larger (*cf.* Section 5.3.1) and thus the DNN can learn more on the target dataset. Moreover, we observe that the 95% confidence intervals are much larger in this setting.

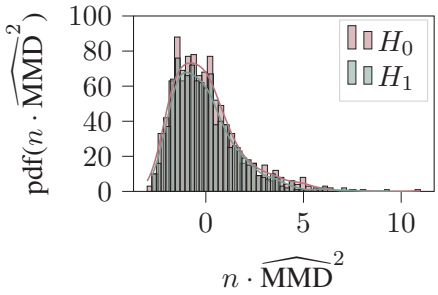
Overall, these results indicate that the fine-tuning algorithm works with another DNN architecture to trade off forgetting the source distribution and learning the target distribution when fine-tuning a network under distributional shifts.



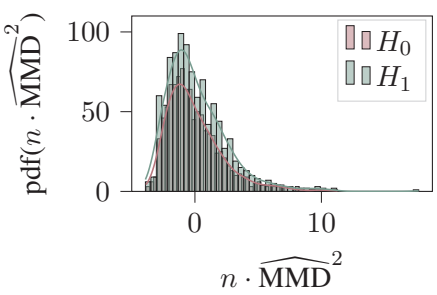
(a) $\mathcal{D}^{\text{keep}}_{\text{highD}}$ vs. $\mathcal{D}^{\text{keep}}_{\text{highD}}$; p-value : 0.772; TPR H_1 : 4.70%



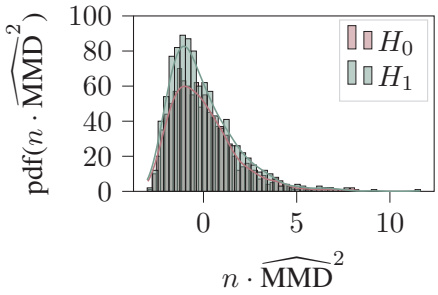
(b) $\mathcal{D}^{\text{keep}}_{\text{NGSIM}}$ vs. $\mathcal{D}^{\text{keep}}_{\text{NGSIM}}$; p-value : 0.397; TPR H_1 : 4.30%



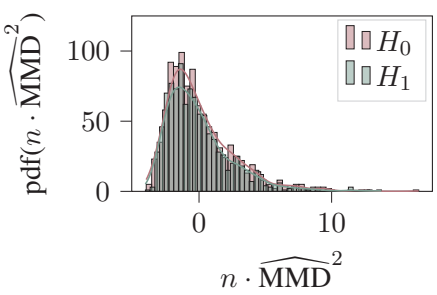
(c) $\mathcal{D}^{\text{left}}_{\text{highD}}$ vs. $\mathcal{D}^{\text{left}}_{\text{highD}}$; p-value : 0.122; TPR H_1 : 3.70%



(d) $\mathcal{D}^{\text{left}}_{\text{NGSIM}}$ vs. $\mathcal{D}^{\text{left}}_{\text{NGSIM}}$; p-value : 0.607; TPR H_1 : 5.50%

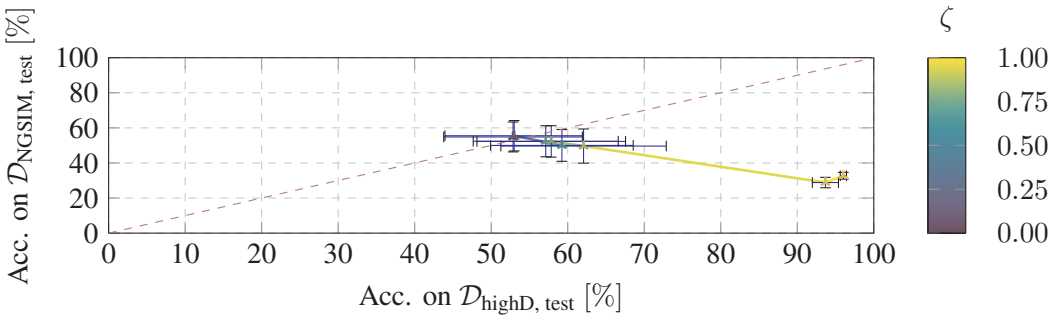


(e) $\mathcal{D}^{\text{right}}_{\text{highD}}$ vs. $\mathcal{D}^{\text{right}}_{\text{highD}}$; p-value : 0.432; TPR H_1 : 5.10%

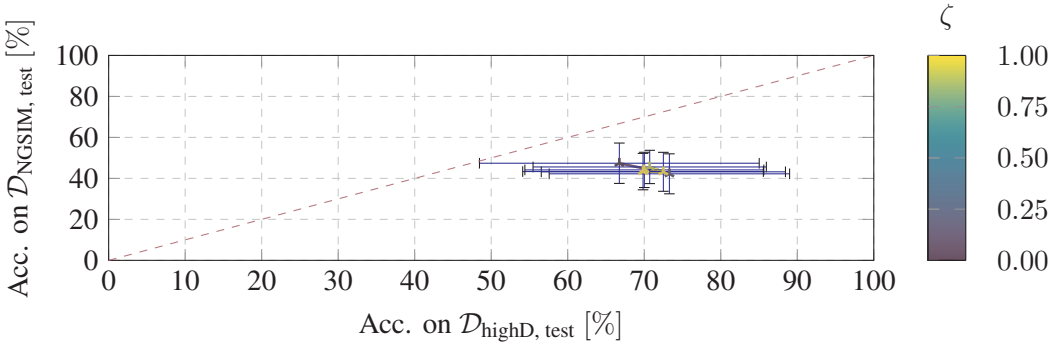


(f) $\mathcal{D}^{\text{right}}_{\text{NGSIM}}$ vs. $\mathcal{D}^{\text{right}}_{\text{NGSIM}}$; p-value : 0.290; TPR H_1 : 4.00%

Figure B.2: Estimates of the MMD test statistic distribution under H_0 and H_1 for the driving manoeuvres in the same dataset. In these cases, the hypothesis test would accept the null hypothesis H_0, indicating there is no distributional shift within the datasets.



(a) Classifiers trained on $\mathcal{D}_S = \mathcal{D}_{\text{highD, train}}$ and fine-tuned on $\mathcal{D}_T = \mathcal{D}_{\text{NGSIM, train}}$.



(b) Classifiers trained on $\mathcal{D}_S = \mathcal{D}_{\text{NGSIM, train}}$ and fine-tuned on $\mathcal{D}_T = \mathcal{D}_{\text{highD, train}}$.

Figure B.3: The average accuracy of the GRU architecture, trained on a source dataset $\mathcal{D}_S$ and fine-tuned on the target dataset $\mathcal{D}_T$. The models are then tested on both test datasets. The error bars represent the 95% confidence intervals.

C Network Architectures

In this appendix, we introduce the Deep Neural Network (DNN) architectures used in this dissertation. All architectures are implemented using PyTorch `version 1.11.0` [182] using the standard `torch.nn` library for the different layers (unless otherwise stated). Thus, all architectures—and results presented—should be reproducible.[1]

C.1 Time to Lane Change Classification Architectures

In this section, we list the DNN architectures used for the Time to Lane Change (TTLC) classification task. In total, we train: one Fully Connected (FC)-DNN architecture; six Convolutional Neural Network (CNN)-based architectures (including three Multi-Channel (MC) CNN designs); two Recurrent Neural Network (RNN)-based architectures, using a Long Term Short Term Memory (LSTM) cell and a Gated Recurrent Unit (GRU) cell; and an attention-based architecture.

We summarise the parameters of the various architectures in Table C.1. We have four linear layers in the FC-DNN design. The input to the FC-DNN is the number of input channels Γ times the number of sub-sections P times the number time-stamps per sub-section $N_{\text{sub.}}$. For the other architectures, we have Γ inputs (unless otherwise stated). The final linear layer of every architecture maps from $\mathbb{R}^{100} \rightarrow \mathbb{R}^{K}$ since we use a feature embedding space of dimension 100, and we have $K = |\mathbb{Y}|$ classes in the TTLC classification task.

The parameters for the CNN-based architectures are summarised in the middle rows of Table C.1. We have three CNN blocks for each architecture, where we use a different CNN layer for each architecture.[2] Furthermore, note that for the MC CNN architectures, we have the three CNN layers for each of the input channels, i.e., we

[1] Some of our implementations can be found at https://github.com/decandido/feature-validation and https://github.com/decandido/encouraging-validatable-features

[2] Note, the `Conv1dSeparable` and the `Conv1dLocal` layers are not directly defined in the standard `torch.nn` library, however, they can be built using `Conv1d` layers and `torch.nn.Parameter` objects, respectively.

have Γ times the same three CNN layers for each input channel since we consider Γ input channels; In most cases, we use $\Gamma = 10$ for the TTLC classification task. The bottom rows of Table C.1 show the RNN-based architectures. We see there is a single LSTM- or GRU-cell followed by a FC linear layer.

We adapt the implementation of the gated transformer method provided by the authors of [95].[3] We use the following parameters: a model dimension of 50; a query, a value, and a key dimension of 16; 16 attention heads; and 3 encoder blocks. The FC linear layer in the encoder blocks has a dimension of 100. The gated transformer processes the input data both channel- and time-stamp-wise, and we have 3 encoder blocks for the channels and the time-stamps.

An overview of the total number of parameters for each architecture type can be found in Table C.2. We note that with the parameter choices summarised in Table C.1, we have roughly the same number of learnable parameters for the FC-DNN and the CNN-based architectures. On the other hand, the RNN-based architectures require roughly a tenth of the number of parameters. This is due to the fact that RNN blocks are designed to perform well on time-series data by training the weights over time-stamps, which we observe in the results in Chapter 4 and Chapter 6. Moreover, the gated transformer model has more learnable parameters than the other architectures since it processes the input data both channel- and time-stamp-wise.

C.2 Deep Autoencoder Architecture

In Table C.3, we see the parameters for the Deep Autoencoder (DAE) architecture used in the proposed Lane Change Detector (LCD) algorithm in Chapter 6. We have a symmetric encoder/decoder design with the same number of input/output channels in each (just in the reverse order in the decoder). To ensure that the output of the DAE is the same dimension as the input, i.e., $\boldsymbol{X} \in \mathbb{R}^{\Gamma \times 25}$, we use the `ConvTranspose1d` layer in the decoder network. This layer computes a form of deconvolution to effectively upsample the signal between layers [183]. Furthermore, we have a latent space dimension of 10. The same architecture is used for the left, right, and keep DAEs (*cf.* Section 6.2). The encoder design is used as the CNN-based classifier in Chapter 6 with three output neurons for the three driving manoeuvres.

[3] https://github.com/ZZUFaceBookDL/GTN

Architecture	Layer	Input	Output	Kernel	Stride
FC-DNN	`Linear`	$\Gamma \cdot P \cdot N_{\text{sub.}}$ Ns	375 Ns	N/A	N/A
	`Linear`	375 Ns	380 Ns	N/A	N/A
	`Linear`	380 Ns	100 Ns	N/A	N/A
	`Linear`	100 Ns	K Ns	N/A	N/A
CNN-I	`Conv1d`	Γ Chs	211 Chs	1×8	1
	`Conv1d`	211 Chs	260 Chs	1×5	1
	`Conv1d`	260 Chs	100 Chs	1×3	1
	`Linear`	100 Ns	K Ns	N/A	N/A
CNN-II	`Conv1dSeparable`	Γ Chs	550 Chs	1×8	1
	`Conv1dSeparable`	550 Chs	552 Chs	1×5	1
	`Conv1dSeparable`	552 Chs	100 Chs	1×3	1
	`Linear`	100 Ns	K Ns	N/A	N/A
CNN-III	`Conv1dLocal`	Γ Chs	20 Chs	1×8	1
	`Conv1dLocal`	20 Chs	20 Chs	1×5	1
	`Conv1dLocal`	20 Chs	100 Chs	1×3	1
	`Linear`	100 Ns	K Ns	N/A	N/A
CNN-I MC	`Conv1d`	1 Chs	65 Chs	1×8	1
	`Conv1d`	65 Chs	102 Chs	1×5	1
	`Conv1d`	102 Chs	10 Chs	1×3	1
	`Linear`	100 Ns	K Ns	N/A	N/A
CNN-II MC	`Conv1dSeparable`	1 Chs	128 Chs	1×8	1
	`Conv1dSeparable`	128 Chs	256 Chs	1×5	1
	`Conv1dSeparable`	256 Chs	10 Chs	1×3	1
	`Linear`	100 Ns	K Ns	N/A	N/A
CNN-III MC	`Conv1dLocal`	1 Chs	10 Chs	1×8	1
	`Conv1dLocal`	10 Chs	11 Chs	1×5	1
	`Conv1dLocal`	11 Chs	10 Chs	1×3	1
	`Linear`	100 Ns	K Ns	N/A	N/A
LSTM-[10]	`LSTM`	Γ Ns	100 HNs	N/A	N/A
	`Linear`	100 Ns	K Ns	N/A	N/A
GRU-[12]	`GRU`	Γ Ns	100 HNs	N/A	N/A
	`Linear`	100 Ns	K Ns	N/A	N/A

Table C.1: The FC-DNN, CNN-based, and RNN-based architectures used throughout this dissertation. The input/output columns denote the number of neurons (Ns) for the FC linear layers, the size of the hidden neurons (HNs) for the RNN layers, and the number of channels (Chs) for the convolution layers. All layers have a bias.

Architecture	Parameter Count
FC-DNN	369,562
CNN-a)	370,458
CNN-b)	371,807
CNN-c)	374,847
CNN-a) MC	369,777
CNN-b) MC	377,217
CNN-c) MC	383,997
LSTM-[10]	45,507
GRU-[12]	34,307
Gated Tr.-[95]	684,117

Table C.2: An overview of the number of parameters for each of the DNN architectures.

Network	Layer	Input	Output	Kernel	Stride
Encoder	`Conv1d`	10 Chs	20 Chs	1×3	2
	`Conv1d`	20 Chs	40 Chs	1×3	2
	`Conv1d`	40 Chs	60 Chs	1×3	2
	`Linear`	60 Ns	10 Ns	N/A	N/A
Decoder	`Linear`	10 Ns	60 Ns	N/A	N/A
	`ConvTranspose1d`	60 Chs	40 Chs	1×3	2
	`ConvTranspose1d`	40 Chs	20 Chs	1×3	2
	`ConvTranspose1d`	20 Chs	10 Chs	1×3	2

Table C.3: The DAE architecture for the LCD algorithm. The input/output columns denote the number of channels (Chs) for the convolution layers and number of neurons (Ns) for the linear layers.

Bibliography

[1] National Center for Statistics and Analysis, "Traffic Safety Facts Annual Report Tables - National Statistics," Tech. Rep., 2020. [Online]. Available: https://cdan.nhtsa.gov/tsftables/NationalStatistics.pdf

[2] D. A. Pomerleau, "ALVINN: an autonomous land vehicle in a neural network," in *Advances Neural Inf. Process. Syst. (NeurIPS)*, 1989, pp. 305–313.

[3] S. A. E. International, "Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles," 2022. [Online]. Available: www.sae.org

[4] S. Liu, L. Li, J. Tang, S. Wu, and J.-L. Gaudiot, "Creating autonomous vehicle systems," *Synthesis Lectures on Comput. Sci.*, vol. 8, no. 2, pp. 1–216, 2020.

[5] S. Grigorescu, B. Trasnea, T. Cocias, and G. Macesanu, "A survey of deep learning techniques for autonomous driving," *J. Field Robot.*, vol. 37, no. 3, pp. 362–386, 2020. doi: 10.1002/rob.21918

[6] R. Salay, R. Queiroz, and K. Czarnecki, "An Analysis of ISO 26262: Using Machine Learning Safely in Automotive Software," SAE Tech. Report 2018-01-1075, Tech. Rep., 2018.

[7] R. Krajewski, J. Bock, L. Kloeker, and L. Eckstein, "The highD dataset: A drone dataset of naturalistic vehicle trajectories on German highways for validation of highly automated driving systems," in *Proc. IEEE 21st Intell. Transp. Syst. Conf. (ITSC)*. IEEE, 2018. doi: 10.1109/ITSC.2018.8569552

[8] J. Colyar and J. Halkias, "NGSIM - US Highway 101 Dataset," 2006. [Online]. Available: https://www.fhwa.dot.gov/publications/research/operations/07030/index.cfm

[9] ——, "NGSIM - Interstate 80 Freeway Dataset," 2006. [Online]. Available: https://www.fhwa.dot.gov/publications/research/operations/06137/index.cfm

[10] H. Q. Dang, J. Fürnkranz, A. Biedermann, and M. Hoepfl, "Time-to-lane-change prediction with deep learning," in *Proc. IEEE 20th Intell. Transp. Syst. Conf. (ITSC)*. IEEE, 2017. doi: 10.1109/ITSC.2017.8317674

[11] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997. doi: 10.1162/neco.1997.9.8.1735

[12] Z. Yan, K. Yang, Z. Wang, B. Yang, T. Kaizuka, and K. Nakano, "Time to lane change and completion prediction based on Gated Recurrent Unit Network," in *Proc. IEEE Intell. Vehicles Symp. (IV)*. IEEE, 2019, pp. 102–107. doi: 10.1109/IVS.2019.8813838

[13] K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio, "On the Properties of Neural Machine Translation: Encoder–Decoder Approaches," in *8th Workshop Syntax, Semantics and Struct. Statistical Transl.* Assoc. for Comput. Linguistics, 2014, pp. 103–111. doi: 10.3115/v1/W14-4012

[14] F. Wirthmüller, M. Klimke, J. Schlechtriemen, J. Hipp, and M. Reichert, "Predicting the Time Until a Vehicle Changes the Lane Using LSTM-based Recurrent Neural Networks," *IEEE Robot. and Automat. Letters*, vol. 6, no. 2, pp. 2357–2364, 2021. doi: 10.1109/LRA.2021.3058930

[15] O. De Candido, M. Koller, and W. Utschick, "Encouraging Validatable Features in Machine Learning-based Highly Automated Driving Functions," *IEEE Trans. on Intell. Vehicles*, vol. 8, no. 2, pp. 1837–1851, 2023. doi: 10.1109/TIV.2022.3171215

[16] O. De Candido, X. Li, and W. Utschick, "An Analysis of Distributional Shifts in Automated Driving Functions in Highway Scenarios," in *Proc. IEEE 95th Veh. Technol. Conf. (VTC2022-Spring)*. IEEE, 2022. doi: 10.1109/VTC2022-Spring54318.2022.9860453

[17] O. De Candido, M. Binder, and W. Utschick, "An Interpretable Lane Change Detector Algorithm based on Deep Autoencoder Anomaly Detection," in *Proc. IEEE Intell. Vehicles Symp. (IV)*. IEEE, 2021, pp. 516–523. doi: 10.1109/IV48863.2021.9575599

[18] O. De Candido, M. Koller, O. Gallitz, R. Melz, M. Botsch, and W. Utschick, "Towards Feature Validation in Time to Lane Change Classification using Deep Neural Networks," in *Proc. IEEE 23rd Intell. Transp. Syst. Conf. (ITSC)*. IEEE, 2020, pp. 1697–1704. doi: 10.1109/ITSC45102.2020.9294555

[19] X. Huang, D. Kroening, W. Ruan, J. Sharp, Y. Sun, E. Thamo, M. Wu, and X. Yi, "A Survey of Safety and Trustworthiness of Deep Neural Networks: Verification, Testing, Adversarial Attack and Defence, and Interpretability," *Comput. Science Review*, vol. 37, p. 100270, 2020.

[20] S. Mohseni, H. Wang, C. Xiao, Z. Yu, Z. Wang, and J. Yadawa, "Taxonomy of Machine Learning Safety: A Survey and Primer," *ACM Comput. Surv. (CSUR)*, vol. 1, no. 1, pp. 1–35, 2022. doi: 10.1145/3551385

[21] D. Hendrycks, N. Carlini, J. Schulman, and J. Steinhardt, "Unsolved Problems in ML Safety," *arXiv*, 2021. [Online]. Available: http://arxiv.org/abs/2109.13916

[22] "Explainable Artificial Intelligence (XAI)," DARPA, Tech. Rep., 2016. [Online]. Available: https://www.darpa.mil/attachments/DARPA-BAA-16-53.pdf

[23] W. Samek, G. Montavon, A. Vedaldi, L. K. Hansen, and K.-R. Müller, *Explainable AI: interpreting, explaining and visualizing deep learning*. Springer Nature, 2019.

[24] C. Molnar, *Interpretable machine learning*. Lulu. com, 2020.

[25] Z. C. Lipton, "The mythos of model interpretability," *Commun. of the ACM*, vol. 61, no. 10, pp. 35–43, 2018. doi: 10.1145/3233231

[26] A. Hare, "Algorithmic explanations: to become a mockingbird," Master's Thesis, University of Oxford, 2019.

[27] T. Räuker, A. Ho, S. Casper, and D. Hadfield-Menell, "Toward Transparent AI: A Survey on Interpreting the Inner Structures of Deep Neural Networks," *arXiv*, 2022. [Online]. Available: http://arxiv.org/abs/2207.13243

[28] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba, "Learning Deep Features for Discriminative Localization," in *Proc. IEEE Conf. Comput. Vision and Pattern Recognition (CVPR)*. IEEE, 2016, pp. 2921–2929. doi: 10.1109/CVPR.2016.319

[29] M. Bojarski, A. Choromanska, K. Choromanski, B. Firner, L. J. Ackel, U. Muller, P. Yeres, and K. Zieba, "VisualBackProp: Efficient visualization of CNNs for autonomous driving," in *Proc. Int. Conf. Robot. and Automat.* IEEE, 2018, pp. 4701–4708. doi: 10.1109/ICRA.2018.8461053

[30] L. A. Hendricks, Z. Akata, M. Rohrbach, J. Donahue, B. Schiele, and T. Darrell, "Generating visual explanations," in *Proc. Eur. Conf. on Comput. Vision*. Springer, 2016, pp. 3–19. doi: 10.1007/978-3-319-46493-0_1

[31] S. Krening, B. Harrison, K. M. Feigh, S. Member, C. L. Isbell, M. Riedl, and A. Thomaz, "Learning From Explanations Using Sentiment and Advice in RL," *IEEE Tran. Cogn. and Developmental Syst.*, vol. 9, no. 1, pp. 44–55, 2017. doi: 10.1109/TCDS.2016.2628365

[32] J. Kim and J. Canny, "Interpretable Learning for Self-Driving Cars by Visualizing Causal Attention," in *Proc. IEEE Int. Conf. Comput. Vision (ICCV)*. IEEE, 2017, pp. 2961–2969. doi: 10.1109/ICCV.2017.320

[33] W. Zeng, W. Luo, S. Suo, A. Sadat, B. Yang, S. Casas, and R. Urtasun, "End-to-end interpretable neural motion planner," in *Proc. IEEE Conf. Comput. Vision and Pattern Recognition (CVPR)*. IEEE, 2019, pp. 8652–8661. doi: 10.1109/CVPR.2019.00886

[34] P. Koopman, A. Kane, and J. Black, "Credible Autonomy Safety Argumentation," in *Proc. 27th Saf.-Crit. Syst. Symp. (SSS)*, 2019, pp. 1–27.

[35] S. Shalev-Shwartz, S. Shammah, and A. Shashua, "On a Formal Model of Safe and Scalable Self-driving Cars," *arXiv*, 2017. [Online]. Available: http://arxiv.org/abs/1708.06374

[36] ——, "Vision Zero: on a Provable Method for Eliminating Roadway Accidents without Compromising Traffic Throughput," *arXiv*, 2018. [Online]. Available: https://arxiv.org/abs/1901.05022

[37] P. Koopman, B. Osyk, and J. Weast, "Autonomous vehicles meet the physical world: RSS, variability, uncertainty, and proving safety," in *Proc. Comput. Saf., Rel. and Secur.* Springer International Publishing, 2019, pp. 245–253. doi: 10.1007/978-3-030-26601-1_17

[38] I. Hasuo, C. Eberhart, J. Haydon, J. Dubut, R. Bohrer, T. Kobayashi, S. Pruekprasert, X. Y. Zhang, E. A. Pallas, A. Yamada, K. Suenaga, F. Ishikawa, K. Kamijo, Y. Shinya, and T. Suetomi, "Goal-Aware RSS for Complex Scenarios Via Program Logic," *IEEE Trans. Intell. Vehicles*, pp. 1–33, 2022. doi: 10.1109/TIV.2022.3169762

[39] D. Nistér, H.-L. Lee, J. Ng, and Y. Wang, "The Safety Force Field," 2019. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/self-driving-cars/safety-force-field/the-safety-force-field.pdf

[40] "ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary," *ISO/IEC/IEEE 24765:2017(E)*, pp. 1–541, 2017. doi: 10.1109/IEEESTD.2017.8016712

[41] B. Boehm, "Verifying and Validating Software Requirements and Design Specifications," *IEEE Softw.*, vol. 1, no. 1, pp. 75–88, 1984. doi: 10.1109/9781118156674.ch5

[42] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An efficient smt solver for verifying deep neural networks," in *Int. Conf. Comput. Aided Verification (CAV)*, vol. 10426 LNCS. Springer, Cham, 2017, pp. 97–117. doi: 10.1007/978-3-319-63387-9_5

[43] G. Katz, D. A. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljić, D. L. Dill, M. J. Kochenderfer, and C. Barrett, "The Marabou Framework for Verification and Analysis of Deep Neural Networks," in *Int. Conf. Comput. Aided Verification (CAV)*, vol. 11561 LNCS, 2019, pp. 443–452. doi: 10.1007/978-3-030-25540-4_26

[44] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev, "AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation," in *Proc. IEEE Symp. on Secur. and Privacy*. IEEE, 2018, pp. 3–18. doi: 10.1109/SP.2018.00058

[45] P. Cousot and R. Cousot, "Abstract interpretation frameworks," *J. Log. and Comput.*, vol. 2, no. 4, pp. 511–547, 1992. doi: 10.1093/logcom/2.4.511

[46] A. Lomuscio and L. Maganti, "An approach to reachability analysis for feed-forward ReLU neural networks," *arXiv*, 2017. [Online]. Available: https://arxiv.org/abs/1706.07351

[47] W. Xiang, P. Musau, A. A. Wild, D. M. Lopez, N. Hamilton, X. Yang, J. Rosenfeld, and T. T. Johnson, "Verification for machine learning, autonomy, and neural networks survey," *arXiv*, 2018. [Online]. Available: https://arxiv.org/abs/1810.01989

[48] G. Schwalbe and M. Schels, "A Survey on Methods for the Safety Assurance of Machine Learning Based Systems," in *Proc. Eur. Congr. Embedded Real Time Soft. and Syst. (ERTS)*, 2020.

[49] H. Winner, K. Lemmer, T. Form, and J. Mazzega, "PEGASUS—First Steps for the Safe Introduction of Automated Driving," in *Proc. Road Vehicle Automat.* Springer International Publishing, 2019, pp. 185–195. doi: 10.1007/978-3-319-94896-6_16

[50] R. Krajewski, T. Moers, D. Nerger, and L. Eckstein, "Data-Driven Maneuver Modeling using Generative Adversarial Networks and Variational Autoencoders for Safety Validation of Highly Automated Vehicles," in *Proc. IEEE 21st Intell. Transp. Syst. Conf. (ITSC)*. IEEE, 2018, pp. 2383–2390. doi: 10.1109/ITSC.2018.8569971

[51] R. Krajewski, T. Moers, A. Meister, and L. Eckstein, "BézierVAE: Improved Trajectory Modeling using Variational Autoencoders for the Safety Validation of Highly Automated Vehicles," in *Proc. IEEE 22nd Intell. Transp. Syst. Conf. (ITSC)*. IEEE, 2019. doi: 10.1109/ITSC.2019.8917297

[52] A. Corso, R. Lee, and M. J. Kochenderfer, "Scalable Autonomous Vehicle Safety Validation through Dynamic Programming and Scene Decomposition," in *Proc. IEEE 23rd Intell. Transp. Syst. Conf. (ITSC)*. IEEE, 2020. doi: 10.1109/ITSC45102.2020.9294636

[53] T. Hailesilassie, "Rule Extraction Algorithm for Deep Neural Networks: A Review," *Int. J. Comp. Science and Information Security*, vol. 14, no. 7, pp. 376–381, 2016.

[54] J. Rabold, M. Siebers, and U. Schmid, "Explaining Black-Box Classifiers with ILP – Empowering LIME with Aleph to Approximate Non-linear Decisions with Relational Rules," in *Proc. Int. Conf. Inductive Log. Program.*, vol. 11105 LNAI. Springer, Cham, 2018, pp. 105–117. doi: 10.1007/978-3-319-99960-9_7

[55] J. Kim, A. Rohrbach, T. Darrell, J. Canny, and Z. Akata, "Textual explanations for self-driving vehicles," in *Proc. European Conf. on Comput. Vision (ECCV)*. Springer, Cham, 2018. doi: 10.1007/978-3-030-01216-8_35

[56] O. Gallitz, O. De Candido, M. Botsch, and W. Utschick, "Interpretable Feature Generation using Deep Neural Networks and its Application to Lane Change

Detection," in *Proc. IEEE 22nd Intell. Transp. Syst. Conf. (ITSC)*. IEEE, 2019, pp. 3405–3411. doi: 10.1109/ITSC.2019.8917524

[57] O. Gallitz, O. De Candido, M. Botsch, R. Melz, and W. Utschick, "Interpretable Machine Learning Structure for an Early Prediction of Lane Changes," in *Int. Conf. Artif. Neural Netw. and Mach. Learn. (ICANN)*, vol. 12396 LNCS. Springer, Cham, 2020. doi: 10.1007/978-3-030-61609-0_27

[58] ISO, "ISO 26262-1:2018 Road vehicles — Functional Safety Standard," 2018.

[59] ——, "ISO/PAS 21448:2022 Road vehicles — Safety of the intended functionality," 2022.

[60] O. M. Kirovskii and V. A. Gorelov, "Driver assistance systems: Analysis, tests and the safety case. ISO 26262 and ISO PAS 21448," *IOP Conf. Ser.: Mater. Sci. Eng.*, vol. 534, no. 1, p. 012019, 2019. doi: 10.1088/1757-899X/534/1/012019

[61] P. Koopman and M. Wagner, "Toward a Framework for Highly Automated Vehicle Safety Validation," *SAE Tech. Papers*, 2018. doi: 10.4271/2018-01-1071

[62] T. Kelly and R. Weaver, "The Goal Structuring Notation – A Safety Argument Notation," in *Proc. Dependable Syst. and Netw.* Citeseer, 2004.

[63] T. P. Kelly, "Arguing safety-a systematic approach to safety case management," DPhil Thesis, University of York, UK, 1998.

[64] S. Burton, L. Gauerhof, and C. Heinzemann, "Making the Case for Safety of Machine Learning in Highly Automated Driving," in *Proc. Int. Conf. on Comput. Saf., Rel., and Secur. (SAFECOMP)*. Springer, Cham, 2017. doi: 10.1007/978-3-319-66284-8

[65] L. Gauerhof, P. Munk, and S. Burton, "Structuring Validation Targets of a Machine Learning Function Applied to Automated Driving," in *Proc. Int. Conf. on Comput. Saf., Rel., and Secur. (SAFECOMP)*. Springer, Cham, 2018, pp. 45–58. doi: 10.1007/978-3-319-99130-6_4

[66] R. David, L. Gauerhof, R. Hawkins, C. Picardi, and C. Paterson, "Assuring the Safety of Machine Learning for Pedestrian Detection at Crossings," in *Proc. Int. Conf. on Comput. Saf., Rel., and Secur. (SAFECOMP)*. Springer, Cham, 2020, pp. 197–212. doi: 10.1007/978-3-030-54549-9_13

[67] E. Wozniak, C. Cârlan, E. Acar-Celik, and H. J. Putzer, "A Safety Case Pattern for Systems with Machine Learning Components," in *Proc. Int. Conf. on Comput. Saf., Rel., and Secur. (SAFECOMP)*. Springer, Cham, 2020, pp. 370–382. doi: 10.1007/978-3-030-55583-2_28

[68] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org

[69] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning - Data Mining, Inference, and Prediction*. Springer, 2009.

[70] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006.

[71] V. Badrinarayanan, A. Kendall, and R. Cipolla, "Segnet: A deep convolutional encoder-decoder architecture for image segmentation," *IEEE Trans. Pattern Anal. and Mach. Intell.*, vol. 39, no. 12, pp. 2481–2495, 2017. doi: 10.1109/TPAMI.2016.2644615

[72] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask R-CNN," in *Proc. IEEE Int. Conf. Comput. Vision (ICCV)*. IEEE, 2017, pp. 2961–2969. doi: 10.1109/ICCV.2017.322

[73] G. Plastiras, C. Kyrkou, and T. Theocharides, "You Only Look Once: Unified, Real-Time Object Detection," in *Proc. IEEE Conf. Comput. Vision and Pattern Recognition (CVPR)*. IEEE, 2016, pp. 779–788. doi: 10.1109/CVPR.2016.91

[74] R. Girshick, "Fast R-CNN," in *Proc. IEEE Int. Conf. Comput. Vision (ICCV)*. IEEE, 2015, pp. 1440–1448. doi: 10.1109/ICCV.2015.169

[75] I. Cherabier, C. Hane, M. R. Oswald, and M. Pollefeys, "PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation," in *Proc. IEEE Conf. Comput. Vision and Pattern Recognition (CVPR)*. IEEE, 2017, pp. 652–660. doi: 10.1109/CVPR.2017.16

[76] E. Keogh, S. Chu, D. Hart, and M. Pazzani, "An online algorithm for segmenting time series," in *Int. Conf. on Data Mining (ICDM)*. IEEE, 2001, pp. 289–296. doi: 10.1109/icdm.2001.989531

[77] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, 1998. doi: 10.1109/5.726791

[78] L. Sifre, "Rigid-motion scattering for image classification," Ph.D. dissertation, 2014.

[79] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *Proc. IEEE Conf. Comput. Vision and Pattern Recognition (CVPR)*. IEEE, 2017, pp. 1800–1807. doi: 10.1109/CVPR.2017.195

[80] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vision and Pattern Recognition (CVPR)*. IEEE, 2015. doi: 10.1109/CVPR.2015.7298594

[81] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the Inception Architecture for Computer Vision," in *Proc. IEEE Conf. Comput. Vision and Pattern Recognition (CVPR)*. IEEE, 2016, pp. 2818–2826. doi: 10.1109/CVPR.2016.308

[82] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," *arXiv*, 2017. [Online]. Available: http://arxiv.org/abs/1704.04861

[83] Y. Zheng, Q. Liu, E. Chen, Y. Ge, and J. L. Zhao, "Exploiting multi-channels deep convolutional neural networks for multivariate time series classification," *Frontiers of Comput. Sci.*, vol. 10, no. 1, pp. 96–112, 2016. doi: 10.1007/s11704-015-4478-2

[84] M. Lin, Q. Chen, and S. Yan, "Network in network," in *Proc. Int. Conf. on Learn. Representations (ICLR)*. PMLR, 2014.

[85] P. J. Werbos, "Backpropagation Through Time: What It Does and How to Do It," *Proc. of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990. doi: 10.1109/5.58337

[86] S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber, "Gradient Flow in Recurrent Nets: The Difficulty of Learning Long Term Dependencies," in *A Field Guide to Dynamical Recurrent Networks*. Wiley-IEEE Press, 2001, pp. 237–243.

[87] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Trans. Neural Networks*, vol. 5, no. 2, pp. 157–166, 1994. doi: 10.1109/72.279181

[88] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," in *Proc. Empiricial Methods in Natural Lang. Process. (EMNLP)*. Assoc. for Computat. Linguistics, 2014, pp. 1724–1734. doi: 10.3115/v1/d14-1179

[89] A. Graves, *Supervised Sequence Labelling with Recurrent Neural Networks*. Springer Berlin Heidelberg, 2008.

[90] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances Neural Inf. Process. Syst. (NeurIPS)*, 2017, pp. 5999–6009.

[91] J. Devlin, M. W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. North Amer. Chapter Assoc. Comput. Linguistics: Human Lang. Technologies, Vol. 1*. Assoc. for Comput. Linguistics, 2019, pp. 4171–4186.

[92] T. B. Brown, J. Kaplan, and A. Others, "Language Models are Few-Shot Learners," in *Advances Neural Inf. Process. Syst. (NeurIPS)*, 2020, pp. 1877–1901.

[93] A. Ramesh, M. Pavlov, G. Goh, S. Gray, C. Voss, A. Radford, M. Chen, and I. Sutskever, "Zero-Shot Text-to-Image Generation," in *Proc. Int. Conf. Machine Learning (ICML)*. PMLR, 2021, pp. 8821–8831.

[94] N. Parmar, A. Vaswani, J. Uszkoreit, L. Kaiser, N. Shazeer, A. Ku, and D. Tran, "Image Transformer," in *Proc. Int. Conf. Machine Learning (ICML)*. PMLR, 2018, pp. 4055–4064.

[95] M. Liu, S. Ren, S. Ma, J. Jiao, Y. Chen, Z. Wang, and W. Song, "Gated Transformer Networks for Multivariate Time Series Classification," *arXiv*, 2021. [Online]. Available: http://arxiv.org/abs/2103.14438

[96] X. Zhai, A. Kolesnikov, N. Houlsby, and L. Beyer, "Scaling Vision Transformers," in *Proc. IEEE Conf. Comput. Vision and Pattern Recognition (CVPR)*, 2022, pp. 1204–1213. doi: 10.1109/cvpr52688.2022.01179

[97] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Proc. IEEE Conf. Comput. Vision and Pattern Recognition (CVPR)*. IEEE, 2016, pp. 770–778. doi: 10.1109/CVPR.2016.90

[98] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer Normalization," *arXiv*, 2016. [Online]. Available: http://arxiv.org/abs/1607.06450

[99] J. Masci, U. Meier, D. Cireşan, and J. Schmidhuber, "Stacked convolutional auto-encoders for hierarchical feature extraction," in *Int. Conf. on Artif. Neural Networks (ICANN)*. Springer, 2011, pp. 52–59. doi: 10.1007/978-3-642-21735-7_7

[100] Z. Wang, W. Yan, and T. Oates, "Time series classification from scratch with deep neural networks: A strong baseline," in *Int. Joint Conf. on Neural Netw.* IEEE, 2017, pp. 1578–1585. doi: 10.1109/IJCNN.2017.7966039

[101] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986. doi: 10.1038/323533a0

[102] D. P. Kingma and J. L. Ba, "Adam: A method for stochastic optimization," in *Proc. Int. Conf. Learn. Representations (ICLR)*. PMLR, 2015.

[103] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proc. Int. Conf. Machine Learning (ICML)*. PMLR, 2015, pp. 448–456.

[104] A. Alahi, K. Goel, V. Ramanathan, A. Robicquet, L. Fei-Fei, and S. Savarese, "Social LSTM: Human trajectory prediction in crowded spaces," in *Proc. IEEE Conf. Comput. Vision and Pattern Recognition (CVPR)*. IEEE, 2016, pp. 961–971. doi: 10.1109/CVPR.2016.110

[105] A. Gupta, J. Johnson, L. Fei-Fei, S. Savarese, and A. Alahi, "Social GAN: Socially Acceptable Trajectories with Generative Adversarial Networks," in *Proc. IEEE Conf. Comput. Vision and Pattern Recognition (CVPR)*. IEEE, 2018, pp. 2255–2264. doi: 10.1109/CVPR.2018.00240

[106] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?" in *Advances Neural Inf. Process. Syst. (NeurIPS)*, 2014, pp. 3320–3328.

[107] N. Papernot and P. McDaniel, "Deep k-Nearest Neighbors: Towards Confident, Interpretable and Robust Deep Learning," *arXiv*, 2018. [Online]. Available: http://arxiv.org/abs/1803.04765

[108] A. Bendale and T. E. Boult, "Towards open set deep networks," in *Proc. IEEE Conf. Comput. Vision and Pattern Recognition (CVPR)*. IEEE, 2016, pp. 1563–1572. doi: 10.1109/CVPR.2016.173

[109] N. Cammarata, S. Carter, G. Goh, C. Olah, M. Petrov, L. Schubert, C. Voss, B. Egan, and S. K. Lim, "Thread: Circuits," *Distill*, 2020. doi: 10.23915/distill.00024

[110] D. Erhan, Y. Bengio, A. Courville, and P. Vincent, "Visualizing higher-layer features of a deep network," University of Montreal, Montreal, Tech. Rep. 1341, 2009.

[111] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proc. 13th Int. Conf. Artif. Intell. and Stat.* JMLR, 2010, pp. 249–256.

[112] L. McInnes, J. Healy, and J. Melville, "UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction," *arXiv*, 2018. [Online]. Available: http://arxiv.org/abs/1802.03426

[113] C. J. Van Rijsbergen, *Information Retrieval*, 2nd ed. Butterworths, 1979.

[114] E. Fix and J. L. Hodges, *Discriminatory Analysis, Nonparametric Discrimination: Consistency Properties*, 4th ed. USAF school of Aviation Medicine, 1951.

[115] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, *Classification and regression trees*. Belmont, CA: Wadsworth & Brooks, 1984.

[116] F. Pedregosa *et al.*, "Scikit-learn: Machine Learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, 2011.

[117] W. M. Rand, "Objective criteria for the evaluation of clustering methods," *J. Amer. Statistical Assoc.*, vol. 66, pp. 846–850, 1971. doi: 10.1080/01621459.1971.10482356

[118] L. Hubert and P. Arabie, "Comparing partitions," *J. Classification*, vol. 2, no. 1, pp. 193–218, 1985. doi: 10.1007/BF01908075

[119] B. Yang, X. Fu, N. D. Sidiropoulos, and M. Hong, "Towards K-means-friendly spaces: Simultaneous deep learning and clustering," in *Proc. Int. Conf. Machine Learning (ICML)*. PMLR, 2017, pp. 5888–5901.

[120] M. M. Fard, T. Thonet, and E. Gaussier, "Deep k-means: Jointly clustering with k-means and learning representations," *Pattern Recognit. Letters*, vol. 138, pp. 185–192, 2020.

[121] A. Griewank and A. Walther, *Evaluating derivatives: principles and techniques of algorithmic differentiation*, 2nd ed. SIAM, 2008.

[122] D. Bersimas, P. Jaillet, A. Delarue, and S. Martin, "The price of interpretability," 2019. [Online]. Available: http://arxiv.org/abs/1907.03419

[123] K. Lee, K. Lee, H. Lee, and J. Shin, "A simple unified framework for detecting out-of-distribution samples and adversarial attacks," in *Advances Neural Inf. Process. Syst. (NeurIPS)*, 2018, pp. 7167–7177.

[124] K. P. Murphy, *Machine learning: a probabilistic perspective*. MIT press, 2012.

[125] X. Li, "Validation of Machine Learning Modules by investigating Distributional Shifts," Research Internship, Technical University of Munich, 2021.

[126] L. Chen, "Methods for Out-of-Distribution Detection and Domain Adaptation in the Presence of Distributional Shifts," Research Internship, Technical University of Munich, 2022.

[127] ——, "Robustifying Machine Learning-based Automated Driving Functions against Distributional Shifts," Master's Thesis, Technical University of Munich, 2023.

[128] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," *arXiv*, 2020. [Online]. Available: http://arxiv.org/abs/2001.08361

[129] J. Yang, K. Zhou, Y. Li, and Z. Liu, "Generalized Out-of-Distribution Detection: A Survey," *arXiv*, 2021. [Online]. Available: http://arxiv.org/abs/2110.11334

[130] S. J. Pan and Q. Yang, "A Survey on Transfer Learning," *IEEE Trans. on Knowl. and Data Eng.*, vol. 22, no. 10, pp. 1345–1359, 2009. doi: 10.1109/TKDE.2009.191

[131] F. Zhuang, Z. Qi, K. Duan, D. Xi, Y. Zhu, H. Zhu, H. Xiong, and Q. He, "A Comprehensive Survey on Transfer Learning," *Proc. of the IEEE*, vol. 109, no. 1, pp. 43–76, 2021. doi: 10.1109/JPROC.2020.3004555

[132] M. Delange, R. Aljundi, M. Masana, S. Parisot, X. Jia, A. Leonardis, G. Slabaugh, and T. Tuytelaars, "A Continual Learning Survey: Defying Forgetting in Classification Tasks," *IEEE Trans. on Pattern Anal. and Machine Intell.*, vol. 44, no. 7, pp. 3366–3385, 2022. doi: 10.1109/TPAMI.2021.3057446

[133] J. Quinonero-Candela, M. Sugiyama, N. D. Lawrence, and A. Schwaighofer, *Dataset Shift in Machine Learning*. MIT Press, 2009.

[134] H. Shimodaira, "Improving predictive inference under covariate shift by weighting the log-likelihood function," *J. Statistical Planning and Inference*, vol. 90, no. 2, pp. 227–244, 2000. doi: 10.1016/s0378-3758(00)00115-4

[135] G. I. Webb, R. Hyde, H. Cao, H. L. Nguyen, and F. Petitjean, "Characterizing concept drift," *Data Mining and Knowl. Discovery*, vol. 30, no. 4, pp. 964–994, 2016. doi: 10.1007/s10618-015-0448-4

[136] V. M. Souza, D. M. dos Reis, A. G. Maletzke, and G. E. Batista, "Challenges in benchmarking stream learning algorithms with real-world data," *Data Mining and Knowl. Discovery*, vol. 34, no. 6, pp. 1805–1858, 2020. doi: 10.1007/s10618-020-00698-5

[137] J. G. Moreno-Torres, T. Raeder, R. Alaiz-Rodríguez, N. V. Chawla, and F. Herrera, "A unifying view on dataset shift in classification," *Pattern Recognit.*, vol. 45, no. 1, pp. 521–530, 2012. doi: 10.1016/j.patcog.2011.06.019

[138] C. C. Takahashi and A. P. Braga, "A Review of Off-Line Mode Dataset Shifts," *IEEE Comput. Intell. Mag.*, vol. 15, no. 3, pp. 16–27, 2020. doi: 10.1109/MCI.2020.2998231

[139] D. Amodei, C. Olah, J. Steinhardt, P. Christiano, J. Schulman, and D. Mané, "Concrete Problems in AI Safety," *arXiv*, 2016. [Online]. Available: http://arxiv.org/abs/1606.06565

[140] S. Shafaei, S. Kugele, M. H. Osman, and A. Knoll, "Uncertainty in machine learning: A safety perspective on autonomous driving," in *Workshop on Artif. Intell. Saf. Eng. (WAISE)*. Springer, 2018, pp. 458–464. doi: 10.1007/978-3-319-99229-7_39

[141] A. Torralba and A. A. Efros, "Unbiased Look at Dataset Bias," in *Proc. IEEE Conf. Comput. Vision and Pattern Recognition (CVPR)*, 2011, pp. 1521–1528. doi: 10.1109/CVPR.2011.5995347

[142] B. Recht, R. Roelofs, L. Schmidt, and V. Shankar, "Do ImageNet classifiers generalize to ImageNet?" in *Proc. Int. Conf. Machine Learning (ICML)*. PMLR, 2019, pp. 9413–9424.

[143] P. W. Koh *et al.*, "WILDS: A Benchmark of in-the-Wild Distribution Shifts," in *Proc. Int. Conf. Machine Learning (ICML)*. PMLR, 2021, pp. 5637–5664.

[144] A. Malinin *et al.*, "Shifts: A Dataset of Real Distributional Shift Across Multiple Large-Scale Tasks," in *Advances Neural Inf. Process. Syst. (NeurIPS)*, 2021.

[145] A. Rudenko, L. Palmieri1, M. Herman, K. M. Kitani, D. M. Gavrila, and K. O. Arras, "Human Motion Trajectory Prediction: A Survey," *Int. J. of Robot. Res.*, vol. 39, no. 8, pp. 895–935, 2020.

[146] G. J. Szekely and M. L. Rizzo, "Testing for equal distributions in High Dimension," *InterStat*, vol. 5, no. 16.10, pp. 1249–1272, 2004.

[147] W. Utschick, V. Rizzello, M. Joham, Z. Ma, and L. Piazzi, "Learning the CSI Recovery in FDD Systems," *IEEE Trans. on Wireless Commun.*, vol. 21, no. 8, pp. 6495–6507, 2022. doi: 10.1109/TWC.2022.3149946

[148] A. Gretton, K. M. Borgwardt, M. J. Rasch, B. Schölkopf, and A. Smola, "A Kernel Two-Sample Test," *J. Mach. Learn. Res.*, vol. 13, no. 1, pp. 723–773, 2012.

[149] D. Lopez-Paz and M. Oquab, "Revisiting classifier two-sample tests," in *Proc. Int. Conf. on Learn. Representations (ICLR)*. PMLR, 2017.

[150] H. Ismail Fawaz, G. Forestier, J. Weber, L. Idoumghar, and P. A. Muller, "Deep learning for time series classification: a review," *Data Mining and Knowl. Discovery*, vol. 33, no. 4, pp. 917–963, 2019. doi: 10.1007/s10618-019-00619-1

[151] E. Tzeng, J. Hoffman, N. Zhang, K. Saenko, and T. Darrell, "Deep Domain Confusion: Maximizing for Domain Invariance," *arXiv*, 2014. [Online]. Available: http://arxiv.org/abs/1412.3474

[152] M. Long, Y. Cao, J. Wang, and M. I. Jordan, "Learning transferable features with deep adaptation networks," in *Proc. Int. Conf. Machine Learning (ICML)*. PMLR, 2015, pp. 97–105.

[153] M. Binder, "Reconstructing and Classifying Highway Pilot Driving Maneuvers using Autoencoders," Bachelor's Thesis, Technical University of Munich, 2020.

[154] I. Y. Tyukin, A. N. Gorban, S. Green, and D. Prokhorov, "Fast construction of correcting ensembles for legacy Artificial Intelligence systems: Algorithms and a case study," *Inf. Sci.*, vol. 485, pp. 230–247, 2019. doi: 10.1016/j.ins.2018.11.057

[155] A. N. Gorban, V. A. Makarov, and I. Y. Tyukin, "High-dimensional brain in a high-dimensional world: Blessing of dimensionality," *Entropy*, vol. 22, no. 1, pp. 82–98, 2020. doi: 10.3390/e22010082

[156] A. N. Gorban, I. Y. Tyukin, and I. Romanenko, "The Blessing of Dimensionality: Separation Theorems in the Thermodynamic Limit," *IFAC-PapersOnLine*, vol. 49, no. 24, pp. 64–69, 2016. doi: 10.1016/j.ifacol.2016.10.755

[157] A. N. Gorban and I. Y. Tyukin, "Stochastic separation theorems," *Neural Networks*, vol. 94, pp. 255–259, 2017. doi: 10.1016/j.neunet.2017.07.014

[158] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and Harnessing Adversarial Examples," in *Proc. Int. Conf. on Learn. Representations (ICLR)*, 2015.

[159] F. Karim, S. Majumdar, and H. Darabi, "Adversarial attacks on time series," *IEEE Trans. on Pattern Anal. and Machine Intell.*, vol. 43, no. 10, pp. 3309–3320, 2020. doi: 10.1109/tpami.2020.2986319

[160] H. I. Fawaz, G. Forestier, J. Weber, L. Idoumghar, and P. A. Muller, "Adversarial attacks on deep neural networks for time series classification," in *Proc. Int. Joint Conf. on Neural Networks (IJCNN)*. IEEE, 2019. doi: 10.1109/IJCNN.2019.8851936

[161] S. Harford, F. Karim, and H. Darabi, "Adversarial attacks on multivariate time series," *arXiv*, 2020. [Online]. Available: https://arxiv.org/abs/2004.00410

[162] K. Eykholt, I. Evtimov, E. Fernandes, B. Li, A. Rahmati, C. Xiao, A. Prakash, T. Kohno, and D. Song, "Robust Physical-World Attacks on Deep Learning Visual Classification," in *Proc. IEEE Conf. Comput. Vision and Pattern Recognition (CVPR)*. IEEE, 2018, pp. 1625–1634. doi: 10.1109/CVPR.2018.00175

[163] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards Deep Learning Models Resistant to Adversarial Attacks," in *Proc. Int. Conf. on Learn. Representations (ICLR)*. PMLR, 2018.

[164] J. H. Metzen, T. Genewein, V. Fischer, and B. Bischoff, "On detecting adversarial perturbations," in *Proc. Int. Conf. on Learn. Representations (ICLR)*. PMLR, 2017.

[165] T. Uelwer, F. Michels, and O. De Candido, "Learning to Detect Adversarial Examples Based on Class Scores," in *Proc. German Conf. on Artif. Intell. (Künstliche Intelligenz)*. Springer, 2021, pp. 233–240. doi: 10.1007/978-3-030-87626-5_17

[166] R. Chalapathy and S. Chawla, "Deep learning for anomaly detection: A survey," *arXiv*, 2019. [Online]. Available: http://arxiv.org/abs/1901.03407

[167] H. Wang, M. J. Bah, and M. Hammad, "Progress in Outlier Detection Techniques: A Survey," *IEEE Access*, vol. 7, pp. 107 964–108 000, 2019. doi: 10.1109/ACCESS.2019.2932769

[168] H. Bourlard and Y. Kamp, "Auto-association by multilayer perceptrons and singular value decomposition," *Biol. Cybernetics*, vol. 59, no. 4-5, pp. 291–294, 1988. doi: 10.1007/BF00332918

[169] T. Wen and R. Keyes, "Time Series Anomaly Detection Using Convolutional Neural Networks and Transfer Learning," *arXiv*, 2019. [Online]. Available: http://arxiv.org/abs/1905.13628

[170] P. R. Roy and G. A. Bilodeau, "Road user abnormal trajectory detection using a deep autoencoder," in *Proc. Int. Symp. on Visual Comput. (ISVC)*. Springer, 2018, pp. 748–757. doi: 10.1007/978-3-030-03801-4_65

[171] X. Feng, Z. Cen, J. Hu, and Y. Zhang, "Vehicle Trajectory Prediction Using Intention-based Conditional Variational Autoencoder," in *Proc. IEEE 22nd Intell. Transp. Syst. Conf. (ITSC)*. IEEE, 2019, pp. 3514–3519. doi: 10.1109/ITSC.2019.8917482

[172] P. H. Le-Khac, G. Healy, and A. F. Smeaton, "Contrastive Representation Learning: A Framework and Review," *IEEE Access*, vol. 8, pp. 193 907–193 934, 2020. doi: 10.1109/ACCESS.2020.3031549

[173] F. Chiaroni, M. C. Rahal, N. Hueber, and F. Dufaux, "Self-Supervised Learning for Autonomous Vehicles Perception: A Conciliation between Analytical and Learning Methods," *IEEE Signal Process. Mag.*, vol. 38, no. 1, pp. 31–41, 2021. doi: 10.1109/MSP.2020.2977269

[174] S. Kornblith, M. Norouzi, H. Lee, and G. Hinton, "Similarity of Neural Network Representations Revisited." PMLR, 2019.

[175] S. Barannikov, I. Trofimov, N. Balabin, and E. Burnaev, "Representation Topology Divergence: A Method for Comparing Neural Network Representations," 2021. [Online]. Available: http://arxiv.org/abs/2201.00058

[176] C. Guo, G. Pleiss, Y. Sun, and K. Q. Weinberger, "On calibration of modern neural networks," in *Proc. Int. Conf. Machine Learning (ICML)*. PMLR, 2017, pp. 2130–2143.

[177] I. Chevyrev and H. Oberhauser, "Signature moments to characterize laws of stochastic processes," *J. Mach. Learn. Res.*, vol. 23, pp. 1–42, 2022.

[178] P. Koopman, "The Heavy Tail Safety Ceiling," in *Automated and Connected Vehicle Syst. Testing Symp.* SAE, 2018.

[179] K. H. Kim, S. Shim, Y. Lim, J. Jeon, J. Choi, B. Kim, and A. S. Yoon, "RAPP: Novelty Detection with Reconstruction Along Projection Pathway," in *Proc. Int. Conf. on Learn. Representations (ICLR)*. PMLR, 2020.

[180] P. W. Koh and P. Liang, "Understanding Black-box Predictions via Influence Functions," in *Proc. Int. Conf. Machine Learning (ICML)*. PMLR, 2017, pp. 1885–1894.

[181] Z. Kong and K. Chaudhuri, "Understanding Instance-based Interpretability of Variational Autoencoders," in *Advances Neural Inf. Process. Syst. (NeurIPS)*, 2021, pp. 2400–2412.

[182] A. Paszke *et al.*, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances Neural Inf. Process. Syst. (NeurIPS)*. Curran Associates, Inc., 2019, pp. 8024–8035.

[183] M. D. Zeiler, D. Krishnan, G. W. Taylor, and R. Fergus, "Deconvolutional networks," in *Proc. IEEE Conf. Comput. Vision and Pattern Recognition (CVPR)*. IEEE, 2010, pp. 2528–2535. doi: 10.1109/CVPR.2010.5539957