Thesis for The Degree of Doctor of Philosophy

Journeys in vector space: Using deep neural network representations to aid automotive software engineering

Dhasarathy Parthasarathy

Division of Computing Science
Department of Computer Science & Engineering
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden, 2023

Journeys in vector space: Using deep neural network representations to aid automotive software engineering

Dhasarathy Parthasarathy

ISBN 978-91-7905-945-3
Doktorsavhandlingar vid Chalmers tekniska högskola, Ny serie nr 5411.
ISSN 0346-718X

Department of Computer Science & Engineering
Division of Computing Science
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden

This thesis has been prepared using LaTeX.

Printed by Chalmers Reproservice,
Gothenburg, Sweden 2023.

”

,,,,

Abstract

Context – The automotive industry is in the midst of a transformation where software is becoming the primary tool for delivering value to customers. While this has vastly improved their product offerings, vehicle manufacturers are facing an urgent need to continuously develop, test, and deliver functionality, while maintaining high levels of quality. Increasing digitalization in the past decade allows us to turn to an interesting avenue for addressing this need, which is *data*. With activities in engineering and operating vehicles being increasingly recorded as data, and with rapid advances in machine learning, this work takes a data-driven, deep learning approach to solve tasks in automotive software engineering.

Scope – This work focuses upon two automotive software engineering tasks, (1) assessing whether embedded software complies with specified design guidelines, and (2) generating realistic stimuli to test embedded software in virtual rigs.

Contributions – **First**, as the main tool for solving the design compliance task, we train `tasnet`, a language model of automotive software. Then, we introduce DECO, a rule-based algorithm which assesses the compliance of query programs with the Controller-Handler automotive software design pattern. Utilizing the property of semantic regularity in language models, DECO conducts this assessment by comparing the geometric alignment between query and benchmark programs in `tasnet`'s representation space. **Second**, focusing upon stimulus generation, we train `logan`, a deep generative model of in-vehicle behavior. We then introduce MLERP, a rule-based algorithm which takes user-specified test conditions and samples `logan` to generate realistic test stimuli which adhere to the conditions. Using the property of interpolation in representation space for semantic combination, MLERP generates novel stimuli within the boundaries of specification. **Third**, staying with the testing use case, we improve `logan` to train `silgan`, which simplifies the specification of test conditions. Then, noting that sampling a generative model is less efficient, we introduce GRADES, a rule-based algorithm that uses a specially constructed objective to search for stimuli. GRADES is built upon the fact that neural networks in `silgan` are differentiable, and, given an appropriate objective, a gradient descent-based search in model representation space efficiently yields suitable stimuli. **Fourth**, we note that our recipe for solving automotive software engineering tasks consistently pairs a self-supervised foundation model with a rule-based algorithm operating in the model's representation space. This paradigm for building predictive models, which we refer to as 'pre-train and calculate', not only extracts nuanced predictions without any supervision, but is also relatively transparent. **Fifth**, with our predictive approach relying heavily upon properties in abstract representation space, we develop techniques that explain and characterize selected high-dimensional vector spaces. **Overall**, by taking a data-driven deep learning approach, techniques we introduce reduce manual effort in undertaking two crucial engineering tasks. This has a direct effect on improving the cadence of automotive software engineering without compromising the quality of delivery.

Keywords – automotive software design and testing, large language models, generative adversarial networks, latent space arithmetic, explainable AI

Acknowledgment

I first thank Carl-Johan Seger, my thesis adviser at Chalmers, without whom I would not have been able to initiate my doctoral research. Your advice, perspectives, and support have been valuable for this journey. My next thanks would be to the wonderful Wallenberg AI, Autonomous Systems and Software Program (WASP) which, apart from funding my work, is exquisitely going about its ambitious mission of nurturing talent in critical technologies. I then thank my WASP collaborators, especially Anton Johansson, for insightful interactions.

At the Volvo Group, my professional universe for the last decade, there are too many people to thank, so any list of names will fall woefully short. Nevertheless, I make a feeble attempt. Enormous thanks to Daniel Karlsson and Dan Walhström, my immediate managers, for rock solid support and quick recognition of the potential of AI. Then comes the core-ML group - it is rare to come across such a committed bunch of comrades who come together to take on tough challenges. Our journeys are nothing short of remarkable. Thanks also to Abhineet Tomar - you're one person who has read every word that I've written. Finally, I reserve special thanks for my erstwhile manager Ted Kruse for giving true meaning to this quest. It's your visionary, yet practical, 'school' of research that I hail from.

The greatest amount of gratitude is reserved for my family, the center of my existence. Thanks Amma, Appa, and Sumaka for making me what I am. Thanks also to Usha amma for always wishing the best for me. Then of course comes Mamta, my juggernaut. The combined subject of my love and admiration, and my primary source of inspiration, this work is mainly for you.

Final word to the actual Parthasarathy. Appa, I hope this makes your banner fly higher.

List of Publications

This thesis is based on the following publications:

[A] **D. Parthasarathy**, C. Ekelin, A. Karri, J. Sun, P. Moraitis
"Measuring design compliance using neural language models: an automotive case study"
PROMISE 2022: Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering.

Judged best paper in PROMISE 22

Contributions – Conceived the overall idea, designed, largely developed, largely analyzed, and wrote the contributions. I collaborated with J. Sun and A. Karri on parts of the development and with C. Ekelin on parts of the analysis.

[B] **D. Parthasarathy**, K. Bäckström, J. Henriksson, S. Einarsdóttir
"Controlled time series generation for automotive software-in-the-loop testing using GANs"
IEEE International Conference On Artificial Intelligence Testing 2020.

Contributions – Conceived the overall idea, designed, largely developed, analyzed, and wrote the contributions. I collaborated with K. Bäckstrom in developing the MLERP algorithm.

[C] **D. Parthasarathy**, A. Johansson
"SilGAN: Generating driving maneuvers for scenario-based software-in-the-loop testing"
IEEE International Conference On Artificial Intelligence Testing 2021.

Contributions – Conceived the overall idea, designed, largely developed, analyzed, and wrote the contributions. I collaborated with A. Johansson in developing the expansion stage of the model.

[D] **D. Parthasarathy**, A. Johansson
"Does the dataset meet your expectations? Explaining sample representation in image data"
32nd Benelux Conference, BNAIC/Benelearn 2020.

Contributions – Conceived the overall idea, designed, largely developed, largely analyzed, and largely wrote the contributions. I collaborated with A.Johansson on developing the overlap index.

Contents

Abstract **v**

Acknowledgement **vii**

List of Publications **ix**

1 Introduction **1**

1.1 Automotive software: preparing for the next wave 1

1.2 Research objectives 3

1.3 Solution approach 6

1.4 Thesis structure 10

1.5 Relation to publications 11

2 Background **13**

2.1 Automotive application software 13

2.1.1 The importance of system thinking 15

2.1.2 The automotive E/E system 16

2.1.3 Vehicle application software as a system 18

2.2 Foundation models in deep learning 21

2.2.1 Domains, tasks and supervision in deep learning 22

2.2.2 Benefits and costs of supervised task specialization . . . 25

2.2.3 Flexibly applying foundation models for tasks 27

I Easing the process of software design compliance 31

3 Automotive software design 33

3.1 Patterns for designing vehicle application software 34

3.2 Limits of traditional design compliance assessment 37

3.3 Towards a deep learning approach for design compliance 38

4 Defining a system for design compliance assessment 41

4.1 The 'language' of design in code 42

4.2 Stating the problem of design compliance 43

4.3 The corpus and design pattern studied 44

5 Building a system for design compliance assessment 51

5.1 Constructing a system for assessing design compliance 51

5.2 Experiments in assessing design compliance 58

6 Discussions 67

6.1 On research questions 67

6.2 On techniques employed 69

6.3 Related work 71

6.4 Congruence with research objectives 71

6.5 Congruence with the solution approach 72

II Easing the process of virtual software testing 75

7 Virtual automotive software testing 77

7.1 Simulation for testing vehicle application software 79

7.2 Limits of traditional simulation methods 82

7.3 A deep learning approach to stimulus generation 84

8 Defining a system for test stimulus generation 87

8.1 A wealth of operational scenarios in signals 88

8.2 Stating the problem of stimulus generation 89

8.3 The system of signals and software studied 91

9 Building a system for sampling test stimuli 95

9.1 Constructing a system for sampling stimuli 95

9.2 Experiments in sampling stimuli 102

10 Building a system for searching test stimuli 109

10.1 Expanding the set of signals 109

10.2 Expanding the system for generating stimuli 113

10.3 Experiments in searching stimuli 117

11 Discussions 127

11.1 On research questions 127

11.2 On techniques employed 130

11.3 Related work 134

11.4 Congruence with research objectives 135

11.5 Congruence with the solution approach 135

III Principled operations in vector space 141

12 Vector operations - a joint re-examination 143

12.1 Learning representations of domains 144

12.2 Regularity for design compliance 146

12.3 Interpolation for stimulus sampling 148

12.4 Gradient descent for stimulus search 149

12.5 Representational similarity as substratum 151

12.6 Put together, pre-train and calculate 152
12.7 What about pre-train and prompt? 153

13 Explaining representation spaces **157**

13.1 Explaining domain adaptation in tasnet 158
13.2 Relative importance of tokens in a domain 159
13.3 Conducting a vocabulary challenge 161
13.4 Results 162
13.5 Discussion 164
13.6 Future extension 166

14 Explaining sample representation in data **169**

14.1 Interpretable assessment of sample representation 170
14.2 Explaining sample representation using annotations 171
14.3 Explaining sample representation using simulation 173
14.4 Discussion 179
14.5 Related work 181
14.6 Future extension 182

15 Conclusions **185**

15.1 Future work 186

Bibliography **187**

Chapter 1

Introduction

What drives the commercial vehicle? During most of its two century old history, advances in the vehicle have primarily been driven by the need to improve the efficiency of haulage and the energy consumed during the process. In just the last few decades, however, the profile of the vehicle has changed dramatically. Commercial vehicles are no longer seen as standalone units and are rather considered to be integral parts of complex industrial systems and logistics chains. New expectations are now being posed on performance, interoperability, and sustainability, consequently requiring the vehicle to be multi-functional and intelligent. This, in turn, has a direct impact on the automotive engineering landscape where, complementing the electromechanical disciplines, software has gained prominence as an important means of delivering value. Today, a wide range of vehicle functionality including fleet operations, driver assistance, and energy management have software at their core. At a time when even the technology for automating the driver lies within grasp, it is safe to say that the modern commercial vehicle is being increasingly driven by software [1].

1.1 Automotive software: preparing for the next wave

The range of software-intensive functionality provided by a modern commercial vehicle is visibly diverse. For instance, a quick glance at the driver guide of a Volvo FH truck [2] reveals functions ranging from configuring the door lock alarm, to draining the air-suspension system during ferry transport, and using the pre-installed Alexa voice assistant. All this is in addition, of course, to the core driving functionality provided by the truck that includes service and emergency braking, fuel-efficient driving, lane keeping assistance, blind spot monitoring, etc. In fact, software-intensive functions exposed to the driver in this particular truck can number close to two hundred. This is a far cry from its humble beginnings where software was applied for rudimentary engine and

brake control. As chronicled in [3], the present state of automotive software is the culmination of at least five major stages of evolution (the current being the 5th), each of which have remarkably increased its scope and influence. While automotive software may have a good run, its onward journey is not without issues. The trinity of major trends – automation, connectivity, and electrification – in addition to digitalization and the growing importance of transportation services, introduces new challenges in the process of engineering automotive software. Let us briefly examine some of them.

- *Evolving application landscape* – An increasingly urbanized and connected planet [4] is transforming the profile of commercial transport. When it comes to trucks, for instance, long-haul transport is no longer the dominant application and shares the spotlight with other applications like urban distribution, construction, and mining. In order to field a product portfolio that targets a wider range of applications, automakers face little choice but to reuse functionality across their vehicle platforms, including software [5].

- *Widening scenarios of operation* – With increased reuse, not only does the software system need to balance a wide set of applications, but it must also reliably operate under a variety of scenarios and conditions. For instance, the climate management system of a truck needs to be equally reliable no matter if it operates in a pit-mine in northern Europe or on a highway in north Africa. The same applies to a battery management system deployed either in a wheel loader in South America or on a bus in South Asia. The effort that vehicle manufacturers undertake to anticipate usage scenarios and consequently design, develop, and test software-intensive solutions, all under the shadow of market pressure, is immense [6].

- *Balancing traditional and emerging domains of functionality* – Control-oriented functionality like vehicle motion control or engine control, have been mainstays of the vehicle software platform for a few decades [3]. Due to their safety and/or time-critical nature, such functions are usually developed using control system or reliability engineering methods. More recently, information-centric domains like infotainment and connectivity have grown in importance. Lacking rigorous safety or timing needs, but introducing new challenges like security and privacy [7], the latter domains arguably lend themselves more easily to the pure software engineering approach. Successful integration of diverse functional domains requires the coexistence of diverse engineering approaches.

- *Balancing reliability and agility* – With vehicles becoming more software-driven, the market expects automakers to deliver functionality at an unprecedentedly high cadence. While it may be relatively easy to meet these expectations in certain functionality domains, others – especially those with safety implications – may find it difficult. Delivering functionality at a high pace, while simultaneously ensuring quality and reliability is among the most difficult challenges facing automotive software engineering today [8].

Simply put, with automotive functionality needing to cope with an expanding list of applications and scenarios, there is a critical need for mechanisms that

speed up the design, development, and testing of on-board software without comprising quality. While this certainly calls for a multipronged search for inspiration and solutions, the principal avenue that this work turns to is *data*. Information, in considerable detail and volume, about the environment and the scenarios under which a vehicle and its constituent systems actually operate, is becoming readily available. This, combined with parallel advances in machine learning techniques means that it is now possible to learn complex phenomena represented in the data. Insights, thus learned, have the potential to ease, and even automate, parts of the automotive software engineering process. Taking first steps in investigating this potential is the fundamental contribution of this work and, in elucidating its findings, we begin with an overview of its objectives.

1.2 Research objectives

Prior to introducing the objectives of this work, it is helpful to briefly set the larger context of engineering vehicle functionality. As we have seen, modern commercial vehicles subsume a few hundred functions that provide end-user value. The traditional approach to realizing one such function would be to use the venerable V-model of engineering. Variously described in [9–12], the V-model is a sequential process, which begins with gathering necessary requirements for the function. This is followed by analyzing and designing the function in a manner agnostic to its implementation. Then, implementation details are introduced by designing the system, sub-systems, and components that would realize the function. The design process is followed by the actual implementation, after which the process proceeds into the testing phases.

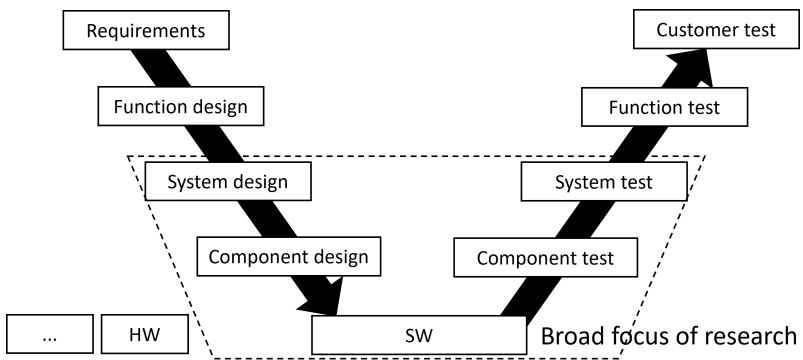


Figure 1.1: The V-model of realizing automotive functionality, with areas of engineering focused upon in this work

In this *function* → *systems* → *components* hierarchy of implementation, software as an entity typically makes its appearance at system and component levels. Here, it is often necessary to engineer software in concert with the associated hardware from the electronic, electric, mechanical and other physical domains. Within the elaborate process of engineering vehicle functionality, this work undertakes the objective of easing software design and testing. Specifically,

under the overall goal of developing tools and methods for data-driven software engineering, this work focuses on the following objectives.

1. **Easing the process of software design compliance**. Realizing vehicle functionality in accordance with the V-model involves design at the function, system, and component levels, including that of software. The design of software for, say, a new function is typically not re-invented in a vacuum. Rather, it is guided by pre-defined organizing principles which ensure that this particular design is not at odds with others, and the implementation remains extendable. Perhaps because of its need to coexist with other disciplines, the automotive industry has a particularly strong tradition in identifying guiding principles for designing software [13]. These principles address aspects like separating concerns, encouraging modularity, and promoting reuse, that ultimately enhance the long-term sustainability of the application, and the agility in its engineering. Practically however, at the source code level, inevitable engineering compromises cause the implementation to deviate from specified design principles [14]. If practically unavoidable, in order to ensure that design regression is at least manageable, it is essential to identify deviations and intervene as early as possible. Under current industrial practice, assessment of design compliance is largely manual, requiring the time and expertise of experienced software architects and engineering teams. While manual review ensures that the process is nuanced, the effort involved is often forbidding. The increasing availability of code corpora that represent design nuances in automotive software, we reason, offers an alternative way to automate assessment while simultaneously attending to its subtleties. This sets up the first objective undertaken by this work:

   Given a set of design principles, and application software (code) to which they are applicable, the first objective is to automate the process of assessing whether the code complies with specified design principles, at levels of nuance comparable to manual review.

   The primary tool we use to automate compliance assessment is a neural language model trained on source code. The performance of automatic compliance assessment is evaluated using an expert review process that assesses both the accuracy and nuance of predictions. Being a data-driven technique, it is important to note that the quality of automated assessment depends, among other factors, upon the extent of design nuances captured in code corpora and the ability of the language model in representing them.

2. **Easing the process of virtual software testing**. After being designed and implemented in accordance with the V-model, automotive embedded software goes through multiple phases of testing. As an entity that drives a real-time, safety-critical, multi-physics system, testing embedded on-board software is a formidable operation. While there are several perspectives, approaches, and methods in automotive software testing [15], one technique that has gained prominence during recent years is virtual testing. In this paradigm, testing is conducted with software in the loop with most (if not all) of its dependencies simulated [16]. Simulating physical dependencies like the driver or the engine allows testing the software under a wider range of

scenarios at a faster pace, elements that are critical for the agile development of reliable software. One issue with virtual testing, under current practice, is that the quality of simulation is not always at a level that makes virtual testing credible. Even if it is to only capture phenomena relevant to software under test, in order to be credible, simulation models need to correctly represent such phenomena at a considerable level of detail. With manual specification of dependency behavior being the current norm, enormous amounts of effort need to be expended to capture such details. The increasing availability of data that records the behavior of such dependencies, however, offers an alternative approach to offset this effort. This leads to a definition of the second objective undertaken by this work:

Given application software, and a physical in-vehicle system it depends upon, the second objective is to plausibly simulate this dependency, so that its behavior – as seen in recorded data – can be replicated as plausibly realistic stimuli to test the software.

The main tool used to achieve test stimulus generation is a deep generative model trained on recorded vehicle behavior. The quality of generation is evaluated by verifying that generated stimuli are novel, yet verifiably similar, improvisations of recorded stimuli. As yet another data-driven approach, we note that the quality of generated stimuli depends, among other factors, upon the diversity of behavior captured in the data and the ability of the generative model in representing them.

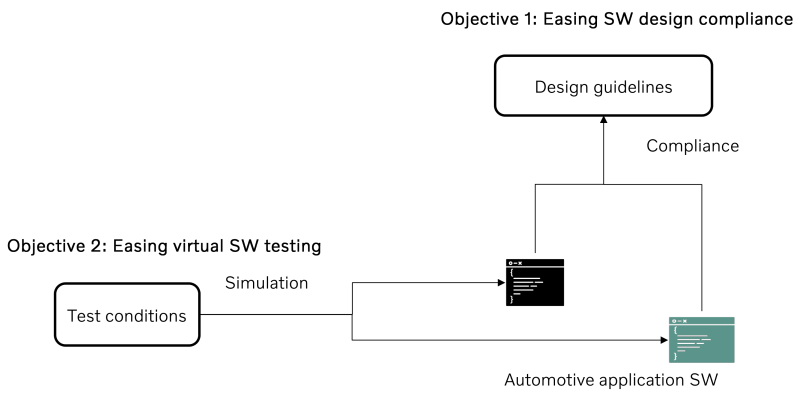


Figure 1.2: Research objectives focused upon in this work

As reiterated in Figure 1.2, this work places on-board application software at the center of attention and focuses upon easing its design and testing. By targeting the immediate upstream and downstream concerns in the process of developing automotive software, tools and methods developed in this work address core elements in the quest towards quickly delivering vehicle functionality without compromising quality. This, in turn, has the potential to mitigate several challenges foreseen in the development of the next wave of automotive software.

1.3 Solution approach

While previous sections may have hinted at it, we now explicitly state the approach used to address the stated research objectives. For ease of discussion, the statement is divided into three parts.

[A] This work focuses on developing data-driven, deep learning methods that ease the design and testing of automotive software.

[B] Using *foundation models* in the requisite domain as primary tools, we use *principled, rule-based, vector operations* in the model's representation space to automate stated software design and testing tasks. This achieves nuance and flexibility in solving tasks, while avoiding the costs of explicit supervision. Additionally, this approach results in predictive pipelines with relative transparency.

[C] Building upon such transparency, in addition to leveraging benchmark datasets, we use expert review for the design task and rule-based statistical analysis for the test stimulus generation task, to verify that the approach for automating either task is plausible.

Why we take this approach – To clarify why we take this approach, we begin with Part A and discuss its twin technical thrusts – big data and deep learning. They may be conjoined but, in order to get a better insight into their respective characteristics and merits, it is instructive to examine them separately.

- **Automotive big data is a reality**. Let us consider two basic facts, (1) commercial vehicles have been equipped with some form of connectivity technology for at least the last 10 years[1], and (2) the market statistics provider Statista[2] estimates that at least 10 million commercial vehicles have been sold annually during that period alone. There is a high likelihood, therefore, that there are more than 100 million connected vehicles plying the roads of the world today. With each of these vehicles reporting a whole host of operational data, the automotive industry is firmly in a big data existence. This development has sparked a number of business opportunities, many of which were unthinkable two decades ago [17]. While exploiting the business potential may have been the obvious first reaction to the development, it is inevitable that the use of data permeates into the engineering process. Engineering decisions, which may have relied more on assumptions and experience, can now be further supported by facts [18]. Data allows organizations to operationalize a build-measure-learn loop where observed insights can be directly applied to the subsequent cycles of software engineering [19]. Parallel to big data, the mainstreaming of open source [20–22] and the proliferation of software across industries [23], has led to an exciting new phenomenon - big *code* [24]. Source code of considerable volume and variety is beginning to become available,

[1] Volvo, for instance, has included a telematics gateway in its trucks since 2012
[2] https://www.statista.com/topics/3582/trucks-and-commercial-vehicles/

which opens up the possibility of using a data-driven approach for coding tasks. The combination of big data and code therefore sets up the perfect baseline to address the objective in focus here – easing software design and testing.

- **Deep learning can help use big data to aid engineering.** Automotive software engineering, like any other engineering enterprise, is a high dimensional activity. Several business and technical concerns, many of which contradict each other, need to be balanced while making decisions. While facts, derived from data, can inform engineers, if facts are not integrated in an accessible manner, they have the potential to overwhelm the engineering process. Worst case, the incomprehensibility of a sheer mass of data can paralyze decision-making. What could help is an intelligent process that abstracts the complexity by actively assisting, and not simply informing, the engineering process. One technology that is able to ingest vast quantities of data, learn complex phenomena that it represents, and use the knowledge to automate tasks is, of course, deep learning [25]. A clear technology megatrend, deep learning has impacted an extraordinarily wide range of domains [26], including software engineering. As surveyed in [27], recent years has seen deep learning being applied to tasks as complex as bug localization and code completion, leveraging both big data and code. There is, therefore, ample potential to apply this technology to assist automotive software design and testing.

Next, we turn to Part B of the approach that captures, in effect, the essence of the main contributions of this work. In order to explain factors that motivate this approach, it is helpful to examine the nature of the tasks specified in the research objectives. As predictive tasks, assessing design compliance or generating test stimuli do share characteristics with typical tasks like image classification or object detection. Yet, as examined below, engineering tasks like those defined in the objectives also differ in significant ways.

- **Engineering tasks are relatively fluid**. The popular impression that deep learning evokes is one of models that, say, classifies an image into one of a set of discrete categories, or those that draw bounding boxes, identifying objects in a scene. Learning to solve such tasks, which are typically difficult to address using hand-crafted algorithms, has been the hallmark of this technology. However, the stereotypical approach for creating such models has been to curate a dataset for each task, with necessary annotation, and train with explicit supervision. In the classification example, this could mean examining each image and assigning labels describing its content. If the task set grows to include locating objects in the image, then additional annotation in the form of bounding boxes is needed. Not only is such annotation widely recognized as incurring enormous costs, but is also an activity whose effort can grow exponentially with the number of tasks. Unlike these stereotypical tasks, where large-scale application may sometimes justify the costs of annotation, engineering tasks can be different. If we take a task like software design review, engineers need to examine code from any number of perspectives like safety, security, or testability. Treating each perspective as an individual task to automate may not be realistic because it is not always clear where one task ends and the other begins. Moreover, as an activity that is built upon human

judgement, engineering decisions are often debatable. During design review, it is quite common that reviewers disagree about firmly labeling a particular design as being (non-)compliant. While there may be broader agreement, finer details may elicit discussions that are hard to settle. Given such a nature, delineating engineering tasks, let alone curating annotated datasets for supervised training to solve each of them, could involve unsurmountable effort. This makes it difficult to take the typical approach of explicit supervision in order to train models that solve tasks like design compliance or simulation.

- **Engineering needs sufficient margin for decision-making**. Beyond the fluidity of tasks it involves, engineering is a creative process that involves experimentation, refinement, and trade-offs. As a result, engineering decision-making is seldom rigid or formulaic. In automotive software engineering, any number of complications, including business needs, legacy, regulation, etc., demand nuance in decision-making. For instance, there may be detailed guidelines on how software should be designed, but the implementation may need to violate it because, say, backward compatibility is broken otherwise. In such a case, it is necessary to make the decision that even if the specification is violated, it is justified under the given circumstance. The need for nuance has special implications for using trained models as engineering tools. Even if we look beyond the costs of annotation, a stereotypical deep learning model, like a cat/dog image classifier, specializes in a narrow task and gives a binary answer. In the design compliance use case, it is less helpful to have a model that makes a yes/no decision on compliance because there is little room for nuance. Given the continuum of possible violations, some of which may be acceptable, it is much more beneficial if model predictions span the inevitable shades of gray. This is yet another reason why explicit supervision with an annotated dataset is less suited to train models that solve tasks defined in our objective.

Finally, we come to Part C of the solution approach which notes our method for evaluating the quality of automating selected software engineering tasks. Though the need for systematic evaluation may seem self-evident, we highlight the essential concern for using trained models as engineering tools.

- **Engineering means humans in the loop**. With the objective of easing the design and testing of automotive software, the focus here is clearly on developing engineering tools. Unlike an application, which focuses on providing sharp value to lay end-users, engineering tools need to interact with human engineers. Therefore, tools need to respond to several needs, and perhaps a few eccentricities, of the human engineer. Engineers, as we reasoned earlier, often require tools that imbue necessary domain knowledge, flexibly tackle closely related tasks, and be answerable to scrutiny. Stated simply, a tool that attempts to assist engineers must reflect some characteristics of engineers themselves. Consequently, an important, but not always definable, expectation that engineers place upon tools is trust. Since tools are intricately involved in decision-making, it is only fair to expect reasonable confidence in the ability of tools to correctly perform tasks.

How we take this approach – Having described why we take the stated

solution approach, we turn to how we undertake it.

Pre-train and calculate – To address the special needs of developing useful engineering tools, we adopt the approach of pairing two powerful concepts (1) foundation models, and (2) principled vector operations (Figure 1.3). In this work, we use terminology introduced by the comprehensive report [28] which defines foundation models as those that learn general concepts in a domain, usually with the intention of being applied to several downstream tasks in that domain. Such knowledge is induced by training them on a large dataset that represents the domain in a self-supervised manner, a step that is commonly referred to as pre-training. Facing fluidity and uncertainty, the use of a domain generalist is analogous to working with an experienced human engineer who is familiar with a domain and is able to generalize to a set of related tasks. Our objective being solving tasks in software design and testing, we respectively train foundation models in domains of vehicle application software and internal vehicle behavior. Upon learning a domain, such generalist models can be used to seed specialist models that solve specific tasks within the domain. A popular method for achieving task specialization is fine-tuning, a commonly used transfer learning [29] technique, which typically involves a supervised training step with annotated data. But, as we reasoned previously, annotated datasets are not especially viable in our case, which makes it difficult to apply the typical 'pre-train and fine-tune' approach. Instead, leveraging the generalist knowledge of foundation models, as an alternative to explicit supervision, we extract predictions using *principled, rule-based vector operations*. Representations learned by foundation models may be abstract, but studies in past years have reported that certain properties are consistently observable in representation space. Using these properties, we build a principled rule-based pipeline of vector operations in the representation space of foundation models to solve the design and testing tasks set in the objective. We refer to this approach using the shorthand *'pre-train and calculate'*. Specifically, we develop the following.

- Addressing design compliance, we train `tasnet`, a neural language model of source code as a foundation model of automotive software. Using the principle of semantic regularity in the model's representation space, we develop DECO a rule-based algorithm that automates compliance assessment by comparing the geometric alignment between query and benchmark programs.
- Addressing test stimulus generation, we train `logan`, a deep generative model on recorded stimuli as a foundation model of selected vehicle behavior. We then develop a rule-based algorithm MLERP which uses interpolation and sampling in representation space to semantically combine recorded stimuli and generate novel stimuli.
- Noting that sampling is less efficient for targeted stimulus generation, we extend `logan` to train `silgan`. Then, using the differentiable nature of deep neural networks, we develop GRADES, a rule-based algorithm that searches for targeted stimuli using gradient-descent in representation space.

Review and rule-based verification – Having utilized a principled and relatively transparent pipeline to solve engineering tasks, we evaluate the quality

of automation in the following manner.

- For the design task, we evaluate the accuracy and nuance of compliance assessment using a benchmark dataset and expert review respectively. The expert review process also doubles up as a calibration mechanism, enhancing the interpretability of automatic design compliance assessment. Currently, our evaluation sets aside a few aspects like calibration of benchmarks across differing or evolving interpretations of the design pattern. Despite such limitations, both the prediction and evaluation techniques ensure that a capable automatic alternative to manual compliance assessment is viable.
- For the test stimulus generation task, we adopt the conservative approach of only generating novel stimuli that are improvised versions of recorded ground-truth stimuli. Further, we use statistical measures to ensure that generated stimuli have verifiable similarities to recorded stimuli that they are improvised from. One limitation of this approach is that the evaluation only produces a relative measure of quality. That is, generated stimuli is measured against semantically close recorded stimuli and not against an absolute physical model of the dependency that is being simulated. Despite using a relative measure of generative quality, our approach ensures that a much-needed alternative to the currently prevalent practice of hand-crafting stimuli is feasible.

Thus, not only do we automate engineering tasks with flexibility and nuance, but we do so with a predictive pipeline that is relatively transparent, explainable, and verifiable within reasonable limits. In the human-in-the-loop reality of automotive software engineering, the approach that we take increases the likelihood that engineers use these tools with relative ease and confidence, simplifying the development of automotive software. Further, in adopting this approach, this work can also be clearly located within the field of deep learning for software engineering [27]. Under this rubric, we introduce innovations that ease its application to use cases in the automotive context.

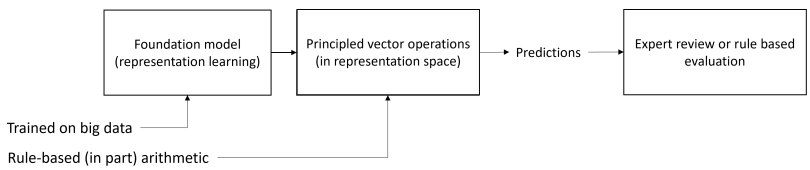


Figure 1.3: The 'pre-train and calculate' approach used for solving software engineering tasks set in the objective

1.4 Thesis structure

Following an overview of relevant background in automotive software engineering and deep learning in Chapter 2, this work devolves into three parts. Part I begins with an introduction of necessary background in automotive software design.

Then, it describes the neural programming language model and the vector operations in its representation space that we develop to solve the task of design compliance assessment. Part II, which deals with the testing objective, begins with an overview of relevant concepts in virtual software testing. Then it presents the deep generative model and associated vector operations that we develop to generate realistic test stimuli. Part III then recounts, compares, and contrasts vector operations that we use in representation space, in order to sketch our common 'pre-train and calculate' recipe for extracting nuanced and transparent predictions. It additionally discusses exploratory studies that we conduct on explaining high-dimensional spaces, before presenting a concluding analysis.

1.5 Relation to publications

This thesis is the amalgamation Papers A-D, listed in Page ix. The relationship between the contents of the thesis, and those in the articles, is mapped below.

- The introductory analysis in Chapter 1 establishes the overall relevance of Papers A-D for automotive software engineering. Further, the solution approach traced in this chapter previews the common recipe used in Papers A-D. Then, Chapter 2 provides necessary background in automotive software engineering and deep learning that sets a baseline for explaining studies in Papers A-D.
- Part I of this thesis, covering Chapters 3-6, is an extended version of Paper A, and describes background, analysis and implications with much more detail.
- Part II in the thesis, subsuming Chapters 7-11, combines and extends Papers B and C, similarly providing deeper explanations, analysis and insights.
- Part III begins with Chapter 12, which presents new analysis that distills the common recipe which Parts I and II employ to solve selected tasks in automotive software engineering. Following this, Chapter 13 presents a new experiment which, develops a method to test specific aspects of the foundation model described in Part I. Then, Chapter 14 presents the content of Paper D and connects it to the overall work conducted in all previous chapters.

Finally, the most significant new analysis presented in the thesis is the synthesis of techniques presented in Papers A-D into the 'pre-train and calculate' paradigm. This paradigm presents a model for nuanced, transparent, and unsupervised task specialization in automotive software engineering. Novel insights from this synthesis echo throughout the thesis, before culminating into a detailed examination of the paradigm and its characteristics in Chapter 12.

Chapter 2

Background

Seeking to apply deep learning to automotive (software) engineering, this work lies at the interface of two immense disciplines. When juxtaposed, there are few other pairs of disciplines that stand in such stark contrast. The former is firmly established and bears a tremendous weight of legacy and tradition. The latter is emerging, disruptive, and developing at a rapid pace. Since a comprehensive joint review of both fields is clearly challenging, this work takes a practical step-by-step approach in organizing the relevant background. As a first step, the current chapter provides relevant background on both disciplines. We begin with an overview of the automotive application software, easing the design and testing of which are our primary objectives. Then, we introduce foundation models, the primary tools used in our approach to ease these processes. Finally, chapters that follow – which actually describe how we apply deep learning to ease vehicle application software design and testing – further present deeper background and related practices in these fields.

2.1 Automotive application software

In the complex entity that is the commercial vehicle, the most tangible of all its facets is the functionality that it provides. Ranging from lane keeping that assists the driver, to air suspension that assists the ride, and assignment management that helps plan the transport mission, vehicle end-user functions are as diverse as they are intricate. As elements of significance, both in this work and in the larger context of automotive engineering, it helps to define a vehicle function as follows.

Definition 1 *An on-board vehicle function* $\bar{\mathbf{y}} = \mathbf{f}(\bar{\mathbf{x}})$ *can be seen as a process that maps a list of inputs to a list of outputs, providing tangible value for either end-users or for other vehicle functions, helping undertake an overall transportation mission. Reflecting the diversity of functionality provided by a vehicle,*

functions **f** *map a variety of input domains to a wide range of output domains.*

Like any function in mathematics or computer science, a vehicle function takes inputs, processes them, and produces outputs that generate value for other functions, or end-users like drivers, mechanics, and fleet operators. Analogous to a function in math or software, inputs to, and outputs from, vehicle functions come in myriad forms and involve human operators, fellow in-vehicle functions, and the external environment. Easing the development of these functions, specifically their design and testing, is the primary objective of this research. While detailing these research objectives, Section 1.2 also briefly introduced the process of developing automotive functionality using the V-model. Such is its recurrence in discussions on engineering automotive functionality [13, 30, 31] that it is considered near-canonical. Thus, in tracing the epic journey of designing and testing automotive software, the V-model is a point of origin as good as any. Even in the simplified interpretation seen in Figure 1.1, design and testing represent two thirds of the effort – a simple but clear indication of their importance. Taking cues from requirements, in the sequential world of V, the design activity is expected to minimize conceptual hurdles that hinder its implementation. Upon implementation, the testing activity ensures its quality, while also making sure that the requirements of the functions are indeed realized. That V is a sequential model is clear, but what is perhaps less apparent is that it entangles two different sequences. Let us briefly untangle and examine them since both bear significance to engineering vehicle software.

The many facets of V – One sequence evident in V is *specify* → *implement* → *verify*. Since this is a sequence of activities guiding the engineering process, it can be seen as a process-oriented sequence. In choosing to embody what is clearly a waterfall engineering process, the automotive V-model betrays at least two things. The first is its traditional roots, having blossomed in a period when the waterfall approach was very much in vogue. The second is the emphasis on reliability in developing an entity that possesses elements of safety-criticality. Even in this arguably post-waterfall era, automotive industry standards for safety [12] and cybersecurity [32] risk management continue to mandate waterfall V as the process to follow. The import that this bears on both the design and testing activities is quite significant. With standards or even regulatory attention being paid, the process and outputs of both activities become artifacts that are comparable in importance with the implementation itself. The next sequence identifiable in V is *function* → *system* → *component*, which surely betrays the multidisciplinary nature of engineering vehicle functionality. Take a function like suspension control, whose implementation spans varied sciences like pneumatics and electromechanics, apart from the computer science of hardware and software. A sequence of (de)composition, that implements a multi-physics function using a multitude of constituent systems and components, is necessary to manage the engineering complexity. It is therefore clear that software engineering is only one among multiple efforts, all of which need to harmonize. This also bears considerable significance upon software design and testing.

This brief examination makes it clear that the canonical V-model places weighty expectations on the design and testing. Yet, for all the import that it seems to

bear, it seems relatively silent on how to practically design, implement, and test a vehicle function, including its software. Notionally, the starting point that V offers for the entire process is requirements. But in reality, do requirements alone have enough substance to drive an engineering process that is clearly complex?

2.1.1 The importance of system thinking

Perhaps the greatest weakness in the V-model approach to engineering automotive functionality is its focus on individual functions. At the outset, such focus may seem warranted because functions provide concrete end-user value. After all, fleet operators pay for fuel-efficiency or driver assistance and not – we are told – for using, say, Linux in the truck. In a modern vehicle with hundreds of end-user functions, let us now consider the effects of excessive function-centrism.

Let us imagine a scenario of developing a vehicle P which realizes N different functions that are exposed to the user. Be it waterfall or agile, the function engineering process cannot usually skip specification, implementation and testing activities. So it is safe to assume that the process sequence of V is followed, in at least some loose sense. How the decomposition sequence is followed is, however, an open question. It is, for instance, possible to realize P, simply an amalgamation of N isolated systems $S_n, n = 1...N$, each dedicated to realizing one function. Simply put, there is nothing in the V decomposition sequence that prevents such absolute isolation between functions. However, it is easy to see that such isolation fragments the system drastically, creating silos of possibly redundant systems and infrastructure. While the scenario of absolute isolation considered here is extreme, system fragmentation introduced by function-centrism is a very real issue [33]. Also, since such systemic disorganization is ostensibly hidden from the customer, popular wisdom on customer priorities remains intact. But, it takes no major leap of imagination to realize that the lack of system organization will eventually impact the customer. The difficulty in evolving a disorganized system decreases the cadence of delivering features, and customers are bound to notice it. And, if using Linux helps organize the system and streamline the delivery and usability of functions, the customer may even demand it.

The multi-functional nature of the vehicle exposes a major inadequacy of V as a complete engineering process – it lacks a clear stance on reuse. When a new function is being engineered, it is impractical to (re-)invent its design, implementation, and tests in a vacuum. It is only practical to derive it from a pre-defined set of reusable principles, platforms, and infrastructure which ensure that the realization of one function is both viable and is not at odds with those of others. This is precisely why the automotive industry has a strong tradition of system thinking and architecture, which helps draft principles, rules, and concepts holistically [13]. Proactively architecting systems and developing infrastructure not only guards against the disorganization of function centrism but also provides concepts and tools that are readily reusable during function development.

An orthogonal system perspective is thus a useful, if not necessary, complement to the function perspective taken by V. Principles, in addition to platforms

and infrastructure, developed from the system perspective provide the necessary scaffolding to engineer several hundred functions with maximum reuse, efficiency, and quality. The importance of system thinking may be reasonably well recognized, but constant market demand for new functionality has given new urgency, thrusting it into the forefront. Take, for instance, the Scaled Agile Framework (SAFe) which was introduced for applying agile methods at enterprise scale and has heavily inspired the automotive industry [34]. SAFe pays close attention to system aspects and calls for the conscious development of a 'runway'[1] of system concepts and infrastructure that allows functionality to be implemented with minimal re-invention and delay. While the coexistence of waterfall V with agile methods like SAFe is far from settled and is a subject of current research [35], it is nevertheless a conspicuous example of the emphasis given to system thinking by modern approaches to automotive engineering.

2.1.2 The automotive E/E system

Among the unique characteristics of the vehicle – a theme that will echo throughout this work – is that it combines a wide range of sciences. To name just a few, there is the thermodynamics of combustion, the hydraulics of braking, the telecommunication of mobile networks, and the computer science of Android apps. So, even if one were to prioritize system thinking, the sheer vortex of disciplines makes it difficult to delineate, organize, and nurture systems. This is particularly true for software because it has traditionally been seen as a mere supporting entity that is deeply embedded within the multi-physics vehicle. Today, with its wide recognition as a key means of delivering end-user value, it is time to relegate the traditional view and pivot software to the foreground.

A network of control units – Perhaps the first step in the elevation of software as a system was the definition of the automotive Electrical/Electronic (E/E) system [36]. As the name suggests, the crucial innovation that it introduced is the recognition that electrics and electronics play an outsized role in managing all other physics. The elementary operating principle of the E/E system is simple – combining sensors and actuators with a microcontroller translates any physical domain into the digital domain, allowing software to control it. This directly leads to the definition of the first pivotal architectural element of the E/E system – the Electronic Control Unit (ECU). The ECU in a vehicle is usually a microcontroller-based embedded system. It interfaces with sensors and actuators, and runs control software, thus serving as a hub for automatic control. The ECU typifies a piece of system thinking that helps architect the E/E system and avoid some pitfalls of function centrism. As rallying points for automatic control functionality, ECUs naturally encourage reuse. This can be illustrated using an example. System thinking at the function level has long understood that vehicle functionality tend to fall under well-recognized domains [37]. Traditionally, functionality domains were primarily control-centric, examples of which include (1) chassis, which controls aspects such as brake and steering, (2) powertrain, which controls the engine or electric

[1] https://www.scaledagileframework.com/architectural-runway/

propulsion motors, (3) body electronics, which controls functions like door locks, windows, ventilation, etc., (4) driver assistance, which controls assistance functions like lane keeping and safety functions like emergency braking. More recently, they have been complemented by information centric domains like (1) connectivity, which handles telematics functions, and (2) infotainment that handles information display and entertainment functions. Using one main ECU and a handful of supporting ECUs for each of these domains virtually sets up the skeleton of a fully functional vehicle. The topology of the resulting distributed E/E system could look like Figure 2.1, with ECUs interlinked using different networking technologies like Controller Area Network (CAN) and Ethernet. Such interlinking ensures that ECUs can also collaborate to realize functionality. By simply combining two architectural principles – ECUs and domain centralization – a sound, yet generic, E/E system emerges. With this, engineering new functionality, even using V, will not happen in a vacuum. Best case, there is a domain, ECU, or even sensors and actuators ready for reuse.

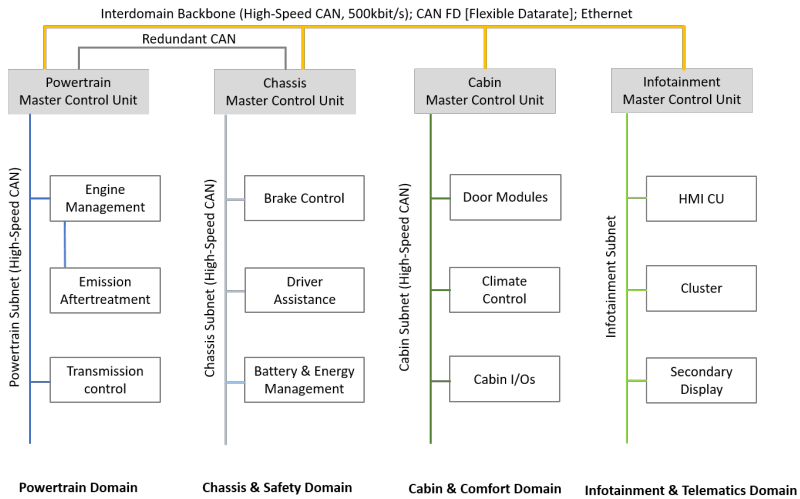


Figure 2.1: Example of a domain-centralized E/E system with the Electronic Control Unit (ECU) as the pivotal entity

It is no exaggeration to say that the ECU as an element has been a major driver for delivering software-based control functionality. However, if E/E system thinking places outsize focus on ECUs, it could mean that not enough attention is paid to nurturing the software that is deployed in it. This usually has the inadvertent side effect of tightly coupling control software with the hardware (microcontroller, sensors, and actuators) of a given ECU, limiting the potential of both. With ECU-centrism therefore leading to its own risks of limited reuse, delivering software intensive functionality with high quality at high cadence must clearly overcome this limitation. Further help is needed to elevate the potential of both hardware and, more importantly for this work, software beyond the restraints of ECU-centrism. This leads us to the next pivotal architectural concept of the E/E system – AUTOSAR.

2.1.3 Vehicle application software as a system

Layered control software – Automotive Software Architecture (AUTOSAR) [38], was nothing less than a groundbreaking development for automotive software when it was first introduced in the early 2000s. Today, as an established industry standard, it institutes a paradigm where software is defined as a system in its own right, and as an entity whose influence is palpable. To get a better insight into the innovations that AUTOSAR introduces, it is easiest to examine its layered architecture (Figure 2.2).

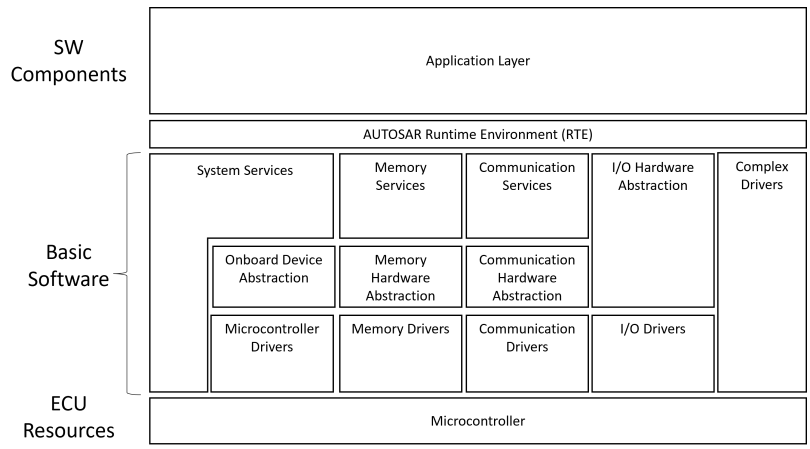


Figure 2.2: The layered AUTOSAR architecture

Bottom-up, classic AUTOSAR defines the following layers. The import of the 'classic' qualifier will be examined shortly.

- **Hardware** – which normally refers to the microcontroller and its peripherals in an ECU. With sufficient means, it can also include sensors and actuators. The recognition of a hardware layer is an act of decoupling as important for hardware as it is for software. After all, it is necessary to be able to evolve hardware components along with software, with minimal mutual interference.

- **Basic software** (BSW) – which can be paraphrased as platform software and has all the characteristics of an embedded real-time operating system. This means that one part of this layer is software that abstracts all hardware including the microcontroller, its peripherals, sensors, and actuators. The other part is core kernel services including the communication stacks for networking. That the classic AUTOSAR kernel chooses an event-triggered execution model [39] betrays its origins, and perhaps its preference, as a host for control-centric functionality. Using this combination of solutions, the BSW layer achieves healthy decoupling from ECU hardware.

- **Runtime environment** (RTE) – which allows automotive software to truly break the barriers of ECU-centrism. The BSW layer introduces crucial decoupling, but only within the confines of one ECU. The RTE takes it a step

further by essentially virtualizing the ECU. It provides an environment and abstractions that allow application software to execute and collaborate in a way that is transparent to the ECU they are actually deployed upon. This ensures that all software atop the RTE is decoupled from the ECU, helping delineate a true application software layer and pivoting software firmly to the foreground.

The AUTOSAR software component – Using the BSW and RTE as a two-step decoupling process, AUTOSAR is another piece of system thinking which architects an exclusive space for application software. That the vehicle is a multi-physics system with strong interdependencies is abundantly clear. But, the E/E system with dual architectural pillars of ECUs and AUTOSAR help abstract not just the electronic hardware but also the platform and application software. This is a tremendously powerful act of inversion that elevates software, which previously lived a deeply embedded existence, into a privileged position. In this relatively hermetic space for software, AUTOSAR introduces another crucial architectural abstraction called the Software Component (SWC) which can be defined as follows.

Definition 2 *In AUTOSAR, a software component* $\bar{\mathbf{y}} = \mathbf{V}(\bar{\mathbf{x}})$*, transforming inputs* $\bar{\mathbf{x}}$ *to outputs* $\bar{\mathbf{y}}$*, is a container for the application logic of the whole, or one part, of a vehicle function. As the basic unit of application software, it implements algorithmic aspects of the function using a set of C files. Reflecting, again, the diversity of in-vehicle functionality, inputs* $\bar{x}$ *and outputs* $\bar{y}$ *of the SWC span a wide variety of domains.*

It is with the definition of the AUTOSAR SWC that we truly enter a space where vehicle application software is properly abstracted. The impact of this development becomes clear upon revisiting the definition of the vehicle function (Definition 1). It may be clear that an automatic control function can encompass multiple physics, but the AUTOSAR SWC boils down all its application software into a set of C files. It is, however, important to note that it is not necessary to host the application software of an entire vehicle function in a single SWC. Considering that end-user functions like suspension control or lane keeping are fairly large and are themselves implemented using a number of discrete operations, AUTOSAR implicitly views an application as a composition of SWCs.

Definition 3 *An AUTOSAR software application* $\bar{\mathbf{y}} = \mathbf{W}(\bar{\mathbf{x}})$ *realizes all software functionality of a vehicle function by composing one or more software components* $\mathbf{W} = \circ\mathbf{V_i}, i \geq 1$*, spread across one or more ECUs.*

The compositional nature of an AUTOSAR application is depicted in Figure 2.3 using the example of the windshield wiper application. An automatic variant of this application consists of a rain sensor which detects a precipitation event, upon which the wiper motor is actuated. Using the virtualization provided by the RTE, the software aspects of this function can be collectively realized by multiple SWCs, in a manner that is largely decoupled from individual ECUs. As

shown in this example, a SWC for handling the rain sensor may be deployed in a separate ECU, perhaps one that is on the roof of the cab of the truck. The core algorithm for wiper control could be another SWC deployed, say, on a central computing ECU. Then, the SWC for handling the wiper motor may be deployed on yet another ECU that handles IO in the cab. Together, all these SWCs, come together to orchestrate the automatic windshield wiper control function. While composing SWCs to realize a function is standard practice, it is important to note that one SWC may not exclusively map to one vehicle function. For instance, in addition to automatic windshield wiping, `RainSensorHandler`, may also be reused for, say, automatically closing an open roof hatch.

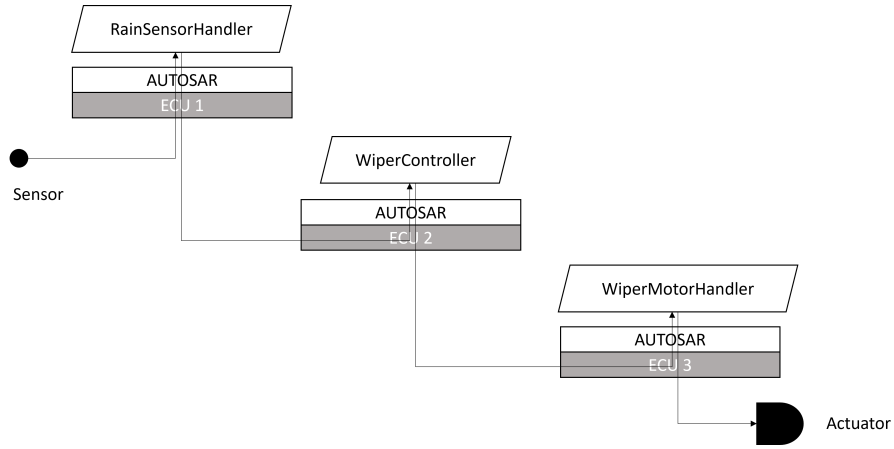


Figure 2.3: Composing multiple SWCs in the AUTOSAR application layer to realize one vehicle function

The E/E system with its combination of ECUs and AUTOSAR has easily been the heart of the software defined vehicle of the past decade. While it has reaped many successes, its limitations are also becoming apparent. For instance, it is easy to recognize that classic AUTOSAR, with its event-triggered, function-oriented model, is mainly designed for control-centric domains. It is, therefore, not directly suitable for a service-oriented, request/response model that is preferable for information-centric domains. Further, while it may virtualize hardware, several aspects of SWC deployment in AUTOSAR need to be frozen at design time, drastically reducing its flexibility. For this, and many other reasons, the AUTOSAR consortium has itself proposed an adaptive[2] version that addresses many of these issues. Today, the automotive industry is already moving towards a reality where classic AUTOSAR is all but one embedded middleware platform [40], sharing space with others like adaptive AUTOSAR, automotive Linux, or even edge middleware which directly interfaces on-board applications with services hosted on the cloud. This also means that AUTOSAR SWCs will be only a subset of all application software in future vehicles. Automotive software may have been on a long journey towards prominence and, since the vehicle is only expected to become more software defined, the journey will continue. But, for the purposes of this work,

[2]https://www.autosar.org/standards/adaptive-platform

it is sufficient to trace the journey up to this point. Deep learning based tools and methods developed in this work for easing the design and testing of vehicle application software can be readily examined from the standpoint of the AUTOSAR SWC. With forthcoming chapters describing these methods in detail, let us move the discussion forward to provide some background on the discipline of deep learning and, particularly, on foundation models which is the primary tool used in this work to ease automotive software engineering.

2.2 Foundation models in deep learning

The artificial intelligence renaissance in the 2010s is attributable to three main factors – our increasingly digitalized and networked existence which produces enormous amounts of data, the reducing cost and increasing power of computing platforms like the Graphical Processing Unit (GPU), and the development of a class of learning algorithms, termed deep learning. Deep learning is a machine learning technique that, at its foundation, composes an elaborate hierarchy of simple concepts to learn a complex concept [25]. Like any other machine learning technique, such learning takes place by understanding patterns that manifest in data, with the active guidance of a set of training objectives. Computationally, this hierarchy of concepts is usually realized using a Deep Neural Network (DNN), which is a stacked composition of simple non-linear functions, each of which transforms its inputs into a form that is more refined to help solve the task at hand. Each layer is composed of several parameters which are learned by a backpropagation mechanism based upon minimizing a differentiable loss function on a large dataset [25]. A famous early application of this seemingly simple formula is the AlexNet [41] image classification model, which dramatically outperformed competing approaches to win the 2012 edition of the ImageNet [42] challenge. In predicting the labels of images with unprecedented (at the time) accuracy, it accomplished a feat that no other hand-coded symbolic algorithm, or even any other machine learning approach, had managed. AlexNet proved to be an early demonstration of the many hallmarks of deep learning – a large training set, the use of efficient yet powerful computations like convolution coupled with rectifying non-linearities, parallel training on multiple GPUs, and – perhaps more importantly – an uncanny tendency to improve benchmarks with little regard to the nature of the problem. Through the decade that followed, the increasing availability of data along with improvements in DNN architectures, training methods, computational platforms, and several other factors, has resulted in deep learning techniques achieving successes across domains as varied as art and drug design. In yet another extension to a new domain, as our research objectives state, this work applies deep learning in automotive software engineering, in an attempt to ease the design and testing of vehicle application software.

2.2.1 Domains, tasks and supervision in deep learning

The spectacular successes of deep learning is most visible through the complex tasks that it has successfully managed to automate. The AlexNet case of classifying images is only one example. Another example in the vision domain would be the YOLO series of models [43] for recognizing and localizing objects in images. In the language domain, a litany of models[3] has been steadily improving benchmarks in translating text from one natural language to another. In the life sciences, the AlphaFold series of models [44] created a sensation by improving benchmarks in performing predictions on protein structure. One theme that pervades all these examples is the fact that they were able to take a big data, deep learning approach to perform tasks that were hitherto difficult to automate using hand-crafted algorithms. Perhaps the most sensational example, as yet, could be the ChatGPT[4] chatbot model that is capable of having free text conversations on a wide variety of topics. Having caught the imagination of society at large, the popular impact of ChatGPT may be hard to dispute. What is perhaps less known is that this chatbot has not been built from scratch and is, in fact, based upon one variant in the GPT-3 [45] series of foundation models. Extending far beyond this particular example, the phenomenon of foundation models have introduced a revolutionary paradigm where models for automating sophisticated tasks can be achieved more efficiently using pre-trained building blocks.

Fully supervised training for task specialists – One way to understand foundation models would be to contrast its characteristics with models that automate specific tasks, examples of which we saw in the previous paragraph. Let us now do such a contrastive examination using a simple example in the natural language domain – the task of sentiment classification [46]. As shown in Figure 2.4, this task can be described as detecting the sentiment expressed in one or more lines of text which, in its most basic form, could be positive or negative. A popular version of this task is classifying the sentiment of movie reviews, which can utilize the specially curated IMDB dataset [47]. This dataset consists of around fifty thousand data samples of the form $S = \{(x_1, y_1), ((x_2, y_2), ...\}$, where each English language review x_i is annotated with a binary label $y_i \in \{0, 1\}$ of whether the sentiment expressed is positive or negative. Using this dataset, it is possible to train a DNN F_T (2.1) that automates the task of sentiment classification. Training such a DNN involves two essential steps, the first of which would be to construct the network. Any DNN is a composition of several layers, each of which is a collection of differentiable[5] non-linear operations with learnable parameters. It is precisely this composition of layers that allows the network to recognize a complex concept like sentiment by learning a hierarchy of simpler concepts in language syntax and semantics. For ease of design, the DNN F_T is often composed as two main blocks (2.2). The first block is the encoder E_T which, by composing layers $e_l, l \in 1, ..., L$, concentrates the bulk of the intelligence for inferring the sentiment in the input x_i. The encoder output is then fed into a simpler head network H_T, which summarizes the inferred sentiment as an interpretable binary label $\hat{y}$. Having

[3] http://nlpprogress.com/english/machine_translation.html
[4] https://openai.com/blog/chatgpt/
[5] In this context, we mean differentiable almost everywhere

assembled the network, the next step is to train it by minimizing the loss or error $\mathcal{L}$ (2.3) between model's sentiment predictions on reviews in the training set and the corresponding ground truth label. These steps, used to train the sentiment classifier, constitute the stereotypical *fully supervised* training paradigm [48].

$$\hat{y_i} = F_T(x_i) \tag{2.1}$$

$$\hat{y_i} = H_T(E_T(x_i)), \quad E_T = \circ_{l=1}^{L} e_l \tag{2.2}$$

$$F_T = \underset{\hat{F}_T}{\operatorname{argmin}} \ \mathbb{E}_{(x_i,y_i)\in S} \ \ \mathcal{L}(\hat{F}_T(x_i), y_i) \tag{2.3}$$

INPUT	PREDICTION
The cinematography is breathtaking	Positive
The music is dull	Negative

Figure 2.4: An example of sentiment classification

The benefit in training the sentiment classification model F_T is that it automates what is a fairly complex task, but training with full supervision does incur costs. First, this model is a specialist in the sentiment classification task alone. If the task is slightly shifted from identifying sentiment to that of identifying emotions [49] – like joy, surprise, anger, etc. – the model F_T, in the absence of major adjustments, becomes unsuitable. As tasks, emotion and sentiment classification may be distinct, but it is also clear that there is a significant overlap between them. If we restrict our example to an English language setting, at the very least, both of them need to understand the syntax and semantics of the same language. In the attempt to recast F_T for emotion classification, one can train it from scratch using, say, the GoEmotions [50] dataset which curates English language sentences and assigns emotion labels from 27 possible options to each sentence. While this gives us an emotion classification specialist, it is easy to see that training afresh on emotion detection overwrites all knowledge gained by F_T, including aspects of language awareness which could have been reused for the new task. Later, if we need to shift the task further to, say, detecting sarcasm [46], training from scratch would, again, be largely redundant. The second major cost incurred in this process is the need for an annotated dataset to train this model. The effort involved in curating something like the IMDB dataset, having people read reviews and make a conclusion about the sentiment, is significant. On top of the inevitable ambiguity in the labeling process, the effort only multiplies if the labeling needs to be more detailed than a simple binary indicator.

Self-supervised pre-training for domain generalists – Alternatively, let us consider the popular BERT [51] language model, whose potential is best understood by examining its training corpus and one of its two training objectives. Let us denote this model as F_D and compose it in the same encoder and head pattern (2.4) as the task specialist that we saw before. Unlike the annotated dataset used in the previous case, the corpus used to train this model is $S = \{x_1, x_2, ...\}$, a mass of unlabeled text data from multiple sources including

Wikipedia. In the absence of labels, BERT trained by asking it to complete a series of cloze tests[6], where the task is to predict hidden words based upon the context. As shown in Figure 2.5, the premise of the task is exceedingly simple. Cloze involves taking a training sample $x_i \in S$ and creating a corrupted version m_i, where a random subset of J tokens are masked out. Upon presenting the masked input m_i, the task that the model needs to accomplish is to correctly predict the tokens that should appear in masked positions. The model is trained to do this task by minimizing the error $\mathcal{L}$ (2.5) between the tokens predicted by the model in masked positions and the ground truth in the same positions.

$$\hat{x_i} = F_D(m_i) = H_D(E_D(m_i)) \tag{2.4}$$

$$F_D = \underset{\hat{F}_D}{\operatorname{argmin}} \ \mathbb{E}_{x_i \in S} \ \mathcal{L}(\hat{F}_D(m_i)[j], x_i[j]) \ , \quad j \in J \tag{2.5}$$

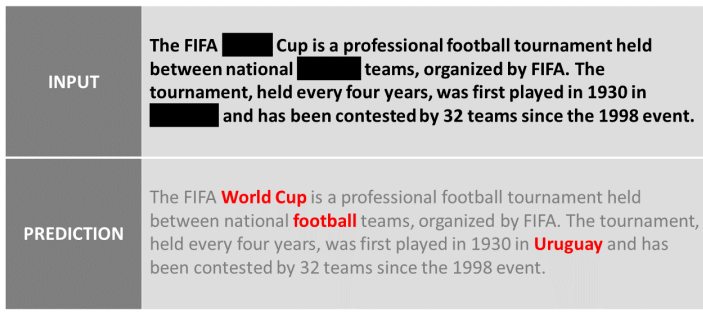


Figure 2.5: An example of the cloze task

It is clear that the cloze task, unlike sentiment classification, is simply too general to have many practical applications. Yet, when trained on millions of paragraphs of text data from Wikipedia and other sources, it is precisely the general nature of this task that renders BERT as a foundation language model. Put simply, as the model learns to correctly predict tokens in masked out positions in millions of cloze challenges during training, it indirectly gains a statistical understanding of the syntax, semantics, and even some underlying knowledge in the English language text with which it is trained. This is why, training with generalist objectives like cloze is typically referred to as *pre-training*. Thus, while cloze itself may have no practical utility, the knowledge of a vast domain that it induces in the model is readily reusable for virtually any task in the same domain. Sure enough, using the well-known approach of transfer learning [52] – a process which we will examine shortly – BERT has been reused to train task specific models that address the previously discussed examples like sentiment and emotion detection [53]. If we now contrast the BERT model F_D with the sentiment classification model F_T, some important characteristics of foundation models emerge. First, foundation models are not task-specific, which may seem meaningless except for the fact that their core purpose is to seed task-specific

[6] https://en.wikipedia.org/wiki/Cloze_test

models in the domain. Second, these models are typically trained on a computationally simple training objective. The cloze training steps of masking a fraction of the input and measuring errors in masked positions are not especially complicated. A vital complement to the computationally simple training objective is the typically large dataset the model is trained on, which is the third main characteristic. The simplicity of the objective and the scale of training with millions of samples go hand-in-hand to induce general knowledge in a domain. Finally, to stay true to its task agnostic nature, a vast amount of knowledge needs to be induced, and this is directly reflected in the number of learnable parameters in the model's layers. Depending upon configuration, the original BERT model has 100-300M learnable parameters [51]. Recent years have seen the introduction of billion-parameter models with GPT-3 [45] being a prominent example.

If BERT's reuse in a family of related tasks is one indication of the potential of foundation models, a far more dramatic showcase would be ChatGPT which is built, among other things, upon the domain understanding of English (and other languages) in GPT-3. While enhanced with other training, inheriting language skills and general knowledge from a large training corpus lies at the very foundation of the chatbot's ability to be conversant in a variety of topics. The language domain may have been the forerunner in mainstreaming the reuse of models with general domain understanding, but the deep learning community quickly grasped that the phenomenon need not be restricted to this domain alone. It did not take long for models with such capabilities to be introduced in domains like computer vision [54] and molecular chemistry [55]. Recognizing their special characteristics in being able to spawn countless task specialists, a new term – foundation models – was prominently adopted in [28] to refer to them. Explaining their reasoning for introducing this term, [28] identify at least two main factors that characterize foundation models.

- Foundation models are general-purpose models in a given domain, that are meant to be reused or adapted to several downstream tasks in its domain
- Correlating with the generality of their purpose, such models are trained self-supervised on datasets that represent their domain

In its short, yet eventful, history, deep learning has itself gone through several disruptions. Foundation models, and the reuse of knowledge that they enable, certainly rank among them.

2.2.2 Benefits and costs of supervised task specialization

Fine-tuning for efficient task specialization – While the importance and utility of foundation models is beginning to be better understood, current practice mostly leans away from directly using them for specific tasks[7]. Rather, the generally accepted way is to use a foundation model F_D as a seed to create

[7] A new paradigm, 'pre-train and prompt' is upending this typical approach. We discuss its implications, comparing it with our approach, in Chapter 12

a fine-tuned task specialist F_T. Let us take a moment to examine typical fine-tuning (Figure 2.6) by considering English language sentiment classifier as the task specialist F_T that we need to achieve. Considering that BERT is a readily available repository of knowledge in this domain, it is a natural candidate for the seed model F_D. With the foundation model being composed of encoder (E_D) and head (H_D) blocks, the first step in fine-tuning is to strip away the head block and discard it. As we noted earlier, the job of the head network is mainly to funnel the model's learning into a particular task. Since the pre-training task was cloze, the funneling provided by the BERT head network H_D is not particularly useful for the new sentiment classification task. What is useful is the BERT encoder block E_D that actually holds the domain knowledge, which is precisely why we retain it. The next step in fine-tuning is to take the BERT encoder E_D and composing it with a new head network H_T, that is meant to repurpose the pre-trained knowledge for the new task. Thus, with E_D derived from BERT, $F_T = H_T(E_D(x))$ can be trained as a sentiment classifier using the supervised training approach described previously in (2.3). Since this reconfigured training process receives a head start from the pre-trained BERT encoder, this process is referred to as fine-tuning. This recipe, therefore, constitutes a different training paradigm – *pre-train and fine-tune* – which is an efficient alternative to fully supervised training. In some cases, the pre-trained block is frozen – prevented from being modified – during training. In this case only the head, which has far fewer parameters, needs to be learned. One beneficial side effect of this is that fine-tuning can work with a much smaller training set than pre-training. This can be advantageous because task specific fine-tuning usually needs annotated data which, as we noted earlier incurs significant cost. Since fine-tuning needs a relatively smaller dataset, the annotation effort may be manageable.

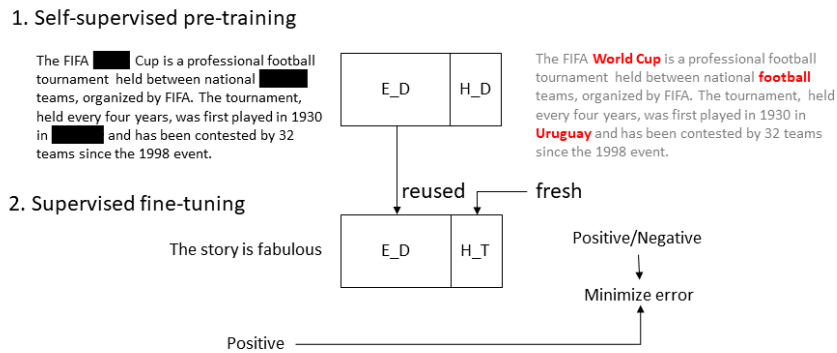


Figure 2.6: A typical fine-tuning process

Representation learning as a key enabler – Fine-tuning is much more efficient than fully supervised training from scratch from two main perspectives. First, only the head H_T of the task specialist F_T needs to be learned, and this contains relatively fewer parameters. Second, the generalist knowledge of the domain that reused encoder E_D learns to represent, is useful for more than one task in the domain. In order to understand the importance of representation, let us probe the fine-tuned model $F_T = H_T(E_D(x))$ further. When a movie review

x is fed forward through the network, the information it contains is sequentially refined layer by layer, until its sentiment ends up as a label at the output of the head. Apart from the input and output which, in this case, are human-readable, any intermediate prediction in the forward pass is an abstract vector of real numbers. Taking advantage of the two-block composition, let us tap into the network to capture the abstract output of the encoder block $e = E_D(x)$. In the context of language modeling especially, this is commonly referred to as an *embedding* of the input x. It is a vector that captures pertinent details about the input x at this stage in the network so that it can eventually complete the ultimate task it was trained upon. But, if we recall the discussion in the previous section, the training objective of BERT (of which E_D is the encoder) is the very general cloze task. The generality cloze forces BERT to embed the input into a semantically rich, yet relatively compact, representation. This representation, in being able to generalize across tasks like sentiment, emotion, or sarcasm detection, plays a crucial role in the success of the pre-train and fine-tune training paradigm. Further, considering that BERT is normally trained for the specific purpose of seeding task specialists downstream, its head H_D is usually meant to be discarded. This necessarily means that the objective of BERT is to learn E_D, a model that embeds or represents text inputs. Put otherwise, the express purpose of training encoder models like BERT can be seen as representation learning.

The intermediate representation, or embedding, e may hold a lot of information, but not all of it is necessary to assign a sentiment label. In fact, the need for the task-specific head H_T arises because the intermediate embedding e is far too rich and entangled to directly use for sentiment classification. While fine-tuning, by using a pre-trained model as a seed, does reduce the training effort, many of the previously identified costs in training a task specialist remain. The supervised training step can take place with a relatively smaller dataset, but there is, nevertheless, a need to annotate which incurs significant cost. Also, being trained on a dataset with binary annotations, the sentiment predicted by the head falls only into one of two predefined categories. Additional nuance in the prediction comes at the cost of additional annotation in the training data.

2.2.3 Flexibly applying foundation models for tasks

So far, we have seen two main paradigms for training models that solve tasks. The first is the fully supervised paradigm where the entire model is trained from scratch using annotated data. The second is pre-train and fine-tune, where a foundation model is pre-trained self-supervised, after which, a task specific head is trained supervised. The latter approach may be more efficient since parameters are inherited, but it continues to rely upon supervision. Since, as we have repeatedly noted, such supervision is challenging to achieve in software engineering tasks, we ask ourselves – can we simply not rely upon rich representations learned by foundation models to solve tasks? It turns out that there are interesting precedents that point to an affirmative answer.

Pre-train and calculate – If we are to rely only upon abstract representations learned by foundation models to solve tasks, we clearly need to turn to

properties that hold in this space. Fortunately, neural network embeddings like e belong to a vector space where several interesting properties apply. One well-known property is that of *representational similarity*, which essentially holds that embeddings that are close to each other in vector space tend to be semantically similar in the input space [56]. That is, given two inputs x_1 and x_2, the learning process tends to place their embedding vectors $e_1 = E_D(x_1)$ and $e_2 = E_D(x_2)$ close together in the embedding space if they happen to convey a similar meaning. This simple detail not only underpins representation learning, which is crucial to the very concept of the deep neural network, but also opens up several avenues for solving tasks. Semantic search is one example of a task which can be solved using the principle of representational similarity, without resorting to a fine-tuned task specialist head. In its most simplistic form, given a corpus $S = \{x_1, x_2, ...x_N\}$ of sentences and a query $q \in S$, semantic search can be defined as a task of finding the sentence $h \in S \setminus q$ such that h is closest in meaning to the query. If we use embeddings of the language model E_D as the index for the search, then one straightforward way to conduct the search would be $h = \text{argmin}_{x_i \in S \setminus q}\, d(E_D(q), E_D(x_i))$. Since embeddings of semantically related sentences should ideally be close in vector space, an appropriate distance measure d can be used to find the sentence whose embedding is closest to that of the query. Practical semantic search may not always use raw language model embeddings, but comparing embeddings in vector space is a common minimum [57]. The example of semantic search reveals how properties of neural network representations can be repurposed for predictive tasks using simple and transparent vector arithmetic.

Using this example as inspiration, going beyond full supervision and pre-train and fine-tune, we use *pre-train and calculate* as the paradigm for solving software engineering tasks. The theme of this approach, as charted in Section 1.3 (also see Figure 1.3), is to pair a domain generalist foundation model with a rule-based procedure. This procedure extracts predictions using simple and principled vector arithmetic. Representational similarity, which we saw in the semantic search example, is only one among many properties that can be used for extracting predictions. The technique that we use in Part I for design compliance assessment is based upon the principle of embedding *regularity*. This property refers to the observation that the embeddings of inputs related by the same concept tend to be arranged in a recognizable geometry [56]. Vector operations that we use for predicting design compliance therefore centers around measuring the alignment with the expected geometry. In Part II, where we deal with test stimulus generation, we turn to two well-recognized properties of DNNs and their embeddings. The first is the property that *interpolation* between embeddings leads to samples that proportionally combine semantics [58]. The second is that most practical DNNs are *differentiable*, which means that gradient descent can be used to identify inputs that satisfy specified conditions at the output of a DNN. After describing how we utilize all these properties to solve software engineering tasks in the first two parts, Part III jointly recounts them to distill our common recipe for building predictive engineering tools using pre-train and calculate. Developing such a principled toolkit for operating in the embedding spaces of foundation models is one way to subvert the cost involved in learning a task specialist in a supervised setting. Since the cost of

supervision under the current practice of automotive software engineering is substantial, such a toolkit not only offers a cheaper but, as we shall soon see, a much more nuanced alternative for task automation. An added advantage is that rule-based predictive steps built upon properties like representational similarity are much more transparent and explainable in comparison to a fine-tuned head. Such transparency increases the likelihood that engineers use these trained tools with confidence. Applying this principled approach to automate tasks in software design and testing, tools and methods we develop help address critical needs for the successful delivery of the next wave of automotive software.

Part I

Easing the process of software design compliance

Chapter 3

Automotive software design

Designing a vehicle function like air suspension or emergency braking may need to take into account all constituent physics but, as we saw in Chapter 2, the E/E system and, particularly, the layered AUTOSAR architecture allows us to focus upon its software in a relatively independent manner. This separation allows us to apply a software engineering approach to develop AUTOSAR applications, including the act of designing it. But where does the software design activity begin? Following the process sequence in V, compiling requirements for application software is certainly one starting point, but as traced extensively earlier, there is ample support available from orthogonal system thinking. Here, the E/E system architectural element of the AUTOSAR software application (Definition 3) especially holds promise. Let us now probe the potential of the software application as a starting point for the software design activity by visualizing its definition in the form of a relationship diagram (Figure 3.1).

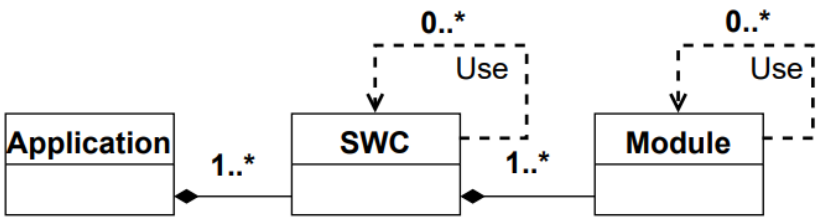


Figure 3.1: The relationship between AUTOSAR applications, software components and modules

As seen in the figure above, the AUTOSAR software application incorporates simple, yet durable, tools for software design. For instance, by allowing an application to be realized using multiple SWCs, the AUTOSAR SWC serves as a crucial pathway for decomposing application logic. Such a decomposition process helps prevent the application code from agglomerating into unmanageable monoliths. Also, the ability of AUTOSAR SWCs to collaborate in realizing vehicle functionality – which we saw in the windshield wiper example in Figure 2.3 –

allows the separation of concerns. The ability to compose allows us to write separate SWCs for sensing, processing, and actuation. This also lays the foundations for reuse where, should a similar sensor be used in a different application, the associated SWC can be reused. By thus offering several advantages, it may seem as if there is enough substance in the AUTOSAR software application as an entity to shoulder the design process. The catch, however, is that among the design semantics shown in Figure 3.1, only the decomposition process is codified. All other benefits are only implied. Put otherwise, AUTOSAR restricts itself mainly to defining the SWC and its composition into software applications. Designers are free to compose them in any manner, including using ways that do not really achieve any of the benefits enumerated earlier. More generally, while E/E system thinking helps establish a useful architectural baseline consisting of ECUs, reference topologies, and AUTOSAR with its various abstractions, additional support is needed to use these concepts and derive a viable software design.

3.1 Patterns for designing vehicle application software

Even if it is common to use the terms interchangeably in day-to-day engineering, it is important to recognize that the architecture and design of vehicle application software are distinct activities. Here, [13] is helpful in identifying their respective characteristics and their inter-relationship. It describes software architecture as very much a big-picture process, helping make choices that define the boundaries of the eventual design. Architectural choices are usually codified and communicated as principles, rules, and structures which can serve as starting points for the design activity. The layered AUTOSAR architecture, and the SWC abstraction, are cases in point. Software design, on the other hand, involves composing and specializing necessary architectural principles and structures to propose a design, while ensuring that the boundaries set by the overall architecture is respected. Thus, [13] makes it clear that architecture is mainly about setting the rules at a high level, while design is about following them at a low level. Considering this dichotomy, it is perhaps reasonable to conclude that the software application and the SWC are at too high a level of abstraction to solely shoulder the design process. Additional guidance is necessary to translate high level ideas to low level details. While this guidance can come from several avenues, a particularly helpful tool that helps translate principles of software architecture into tangible design solutions is the design pattern.

Design patterns, as the classic definition[1] goes, are solutions that can be reused for commonly occurring problems in software design. Simply put, within most software engineering communities, as design problems are encountered and solved, engineers are highly encouraged to document both. Needless to say, a wide range of pattern catalogs have accumulated to aid the design of vehicle application software. For example, [59] catalogs patterns for model-based application development, while [60] and [61] curate patterns for solving safety

[1] https://en.wikipedia.org/wiki/Software_design_pattern.html

and security-related design problems respectively. Also, vehicle application software is almost always embedded software, which means that a sweeping catalog of design patterns [62] for embedded software is always available for reuse. Such a rich palette of catalogs undoubtedly provides much needed system thinking to move towards detailed design, but examples listed above arguably tilt towards core application logic. Meaning, they pay more attention to designing the functional and non-functional properties of the logic contained in a SWC, but they do not necessarily consider the structure and organization of the SWCs themselves. Sure enough, patterns do exist that address this aspect and, since this is of primary interest to this work, let us discuss it further.

Need for design guidance in the AUTOSAR application layer – There is one issue in the AUTOSAR application layer that is crucial to note, and it can be understood by revisiting Figures 2.2 and 2.3. Together, these figures reveal the mixed approach to software architecture taken by AUTOSAR. The former reveals a layered approach from hardware up to the RTE, which virtualizes the hardware. But, as the latter reveals, beyond the RTE, there is a subtle shift from a layered architecture to a component based architecture [38]. That is, in the application layer, the AUTOSAR standard does not define much more than the SWC abstraction, taking no stance on how these components should be arranged. The minimalist approach may be understandable in what is, after all, an industry standard adopted on a large scale, but the component based architecture causes considerable anxiety in the community of automotive software developers and architects [40]. This is simply because, there is little guidance on how to structure and arrange what can be a vast network of SWCs.

Using Figure 3.2, we now illustrate possible issues caused by a lack of clear guidance on SWC organization. This figure shows a system with five different applications and, for ease of reasoning, we assume that they are realized using one SWC each. Let us begin with lane keeping assistance, which helps the driver stay within the lane by detecting lane markings using a camera. Further, let us assume that the design has packaged basic camera processing functionality with the lane detection logic. This is typical when the complete package has been purchased from one vendor. The absence of a clear separation between lane detection and camera processing logic can, however, disturb the overall system. If a pedestrian detection application – which also needs camera data – is now introduced, there is unnecessary coupling with lane detection simply because the latter has inadvertently become the source of all camera data. This can lead to further complications if there is a need to, say, introduce a vehicle variant that excludes lane keeping assistance but retains the legally mandated pedestrian detection application. With the wrongly coupled design, there is no choice but to extricate camera processing from lane keeping, usually at a heavy cost, and reuse it for pedestrian detection. Without taking extra care, the component-style architecture specified by AUTOSAR can have such unintended consequences. Alternatively, if there had been strong design guidance that called for a clean separation between low level device functionality and core application logic, such issues may be avoidable. Having followed such a guideline, the example in Figure 3.2 avoids unnecessary coupling between emergency braking and adaptive cruise control by designing a separate radar processing block. The

need for constant guidance in navigating an essentially flat structure is precisely what keeps software architects anxious. Worst case, even global tight couplings emerge. In our example, pedestrian and lane detection, emergency braking, and cruise control all need to know the ego vehicle speed. It is not uncommon to see practical designs where a speed estimation application is directly coupled to serve all others – otherwise recognizable as the infamous spaghetti design.

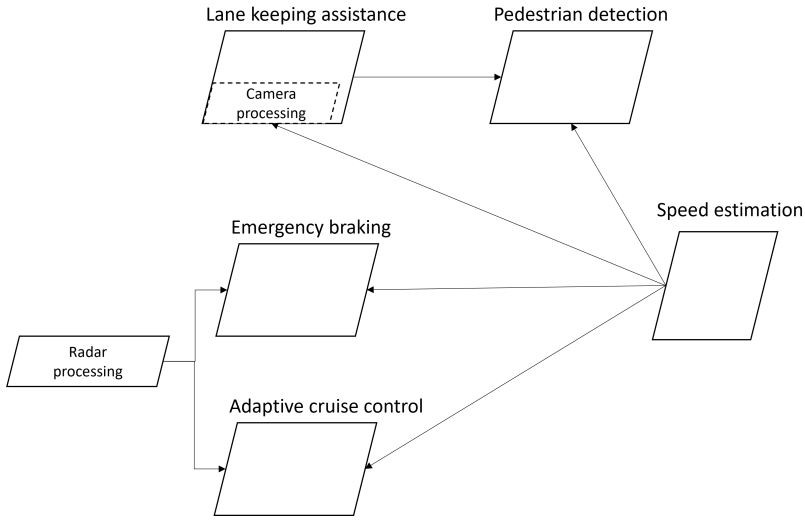


Figure 3.2: Possible (dis)organization in the absence of clear design patterns

With some potentially alarming design concerns on the horizon, AUTOSAR itself provides some guidelines for organization. The AUTOSAR design pattern catalog [63] is one example which addresses some of these issues. One pattern in this catalog encourages the creation of one dedicated SWC for each sensor and actuator, which abstracts all of its hardware details. Another pattern encourages the use of arbiter SWCs which mediate between multiple requesters or providers. Its primary intention may be to resolve conflicting requests, but it may indirectly introduce separation of concerns, helping reduce dependencies. The AUTOSAR design pattern catalog certainly helps address crucial issues but, in addressing three specific concerns, it is fairly limited. Additional guidance is also available in the form of AUTOSAR application interface standards[2]. These standards define external interfaces for routinely offered applications like windshield wiper, seat adjustment, etc. A focus on the external interface, however, only has an indirect interface on SWC structure. In parallel, efforts have also been undertaken outside the AUTOSAR consortium, with vehicle manufacturers defining additional patterns to solve other design issues. One such design pattern, specified by system architects at the Volvo Group, is *Controller-Handler* which encourages control logic to be distinct from device handling logic, introducing some layering in the otherwise flat vehicle application software. Since this work focuses extensively on this pattern, we discuss it in detail in Chapter 4.

Armed with a healthy collection of design pattern catalogs, designing the soft-

[2]https://www.autosar.org/standards/application-interface

ware of an end-user function seems less daunting. Starting with requirements, with the aid of ample system thinking in the form of architectures, design patterns and many more ideas, a viable design should be achievable. At which point of time it becomes pertinent to consider the question – how do we ensure that the subsequent implementation indeed complies with the specified design?

3.2 Limits of traditional design compliance assessment

The objective of software design, as derived from the waterfall V process, is to ensure that the implementation downstream is able to sufficiently address requirements defined upstream. As we also saw previously, concerns that vehicle application software needs to address can be quite diverse, covering aspects like safety and cybersecurity, in addition to the quality and structure of the code itself. In the effort to address them all, it is not uncommon that the design itself includes a hefty list of guidelines, and requirements that need to be fulfilled. Understanding the impact of all of this on the implementation process is therefore crucial. At one level, there is a need to ensure that there are no major flaws in the design itself. Methods like [64,65] help evaluate the viability of high level architectural principles, using which the design is derived. The design itself can be subjected to analysis using, for example, Failure Modes Effects and Analysis (FMEA) which assesses the risk of failures. At another level, there is the additional need to ensure that the implementation actually complies with the specified design. Sometimes, as noted earlier, demonstrating compliance may even be mandated by regulation. Even otherwise, ensuring that the implementation complies with the specified design guidelines eases continuous evolution of the software system, which is necessary for delivering functionality at high cadence and quality.

The challenges involved in the process of assessing design compliance can be examined through two main aspects. Loosely defined, the first aspect would be the ease of formalizing design requirements. The reasoning here is that if the design can be formalized as an algorithm, then it may be possible to construct an automatic static analysis procedure to check design compliance. The MISRA C[3] standard, which is a set of coding guidelines for safety critical systems serves as a good example here. For example, one MISRA C requirement is that a function cannot call itself. Practically, it is also possible to formalize an algorithm that spots recursive functions, which means that verifying compliance with this requirement can be automated. The second important aspect for design compliance is the level of ambiguity in judging compliance. If we take the same MISRA C requirement, it is clear that if a function indeed makes a recursive call, it unambiguously violates the requirement. Meaning, that an automatic compliance checking process can conclude with a true/false answer. Since MISRA C requirements are generally both formalizable and can be unambiguously judged, popular static analysis tools like Klocwork[4] come

[3] https://www.misra.org.uk/misra-c2012-amd3-published/
[4] https://help.klocwork.com/current/en-us/concepts/home.htm

pre-packaged with compliance checking tools for this standard.

Unsurprisingly, there are several cases in automotive software engineering where formalizing design requirements is difficult, and compliance judgment is ambiguous. This can be illustrated by returning to the example in Figure 3.2, where a better design is achieved if application and low level device logic are sufficiently decoupled. In the specific case of the lane keeping, this can translate to a requirement is that the application logic should not contain video processing code. Since (non-) compliance with this requirement is only visible in freely definable variable and method names in the code, any formalization is bound to be brittle. Moreover, it is also easy to see that there are several ambiguities that compromise the judgment of compliance. If lane keeping code, say, crops the image to focus on an area where lane markings are likely to be visible, is that considered to be basic video processing? Since there is sufficient margin for ambiguity in answering this question, the assessment of compliance cannot always result in a firm true/false answer.

To reiterate, if design requirements can be formalized and compliance assessment is unambiguous, it is possible to use a rule-based algorithmic approach for automating this assessment (Figure 3.3). If the ease of formalization reduces and if the ambiguity of assessment increases, both of which are all too common, there is a need for further intelligence and nuance during assessment. Under current practice, such nuance is mainly supplied by human code reviewers. Manual review can notice subtle indications in code for compliance, and an experienced reviewer can always weigh it against standard practice to make fine judgments on compliance. In fact, a study conducted in [66] concludes that during code review, the essential dialog between reviewers and coders centers around design. The problem, of course, is not only that manual review is time-consuming, its effectiveness depends, among other things, upon the experience [67–69] of the participants. The nuance supplied by reviewers – often the result of several years of programming and design – is not easy to cultivate and preserve. However, in the current era of big code, advances in deep learning and neural language processing open up the possibility that nuanced, but time-consuming, manual review can be automated. Thus, when the difficulty of formalizing design requirements and the ambiguity of assessing compliance increase, we reason (Figure 3.3) that it is beneficial to take a learning approach for automating design review.

3.3 Towards a deep learning approach for design compliance

Given an AUTOSAR software application $\bar{\mathbf{y}} = \mathbf{W}(\bar{\mathbf{x}})$, we seek to automatically assess whether the set $\mathbf{V} = \{\mathbf{V_1}, \mathbf{V_2}, ..., \mathbf{V_N}\}, \mathbf{W} = \circ_{i=1}^{N} V_i$ of SWCs which realize this application, complies with the expectations of a design pattern $\mathcal{D}$. If this pattern $\mathcal{D}$ is difficult to formalize and is ambiguous when judging compliance, both of which are not unusual, the design review task firmly lies beyond the conceptual boundary in Figure 3.3 within which rule-based checking is viable. The currently viable alternative of manual design review may have

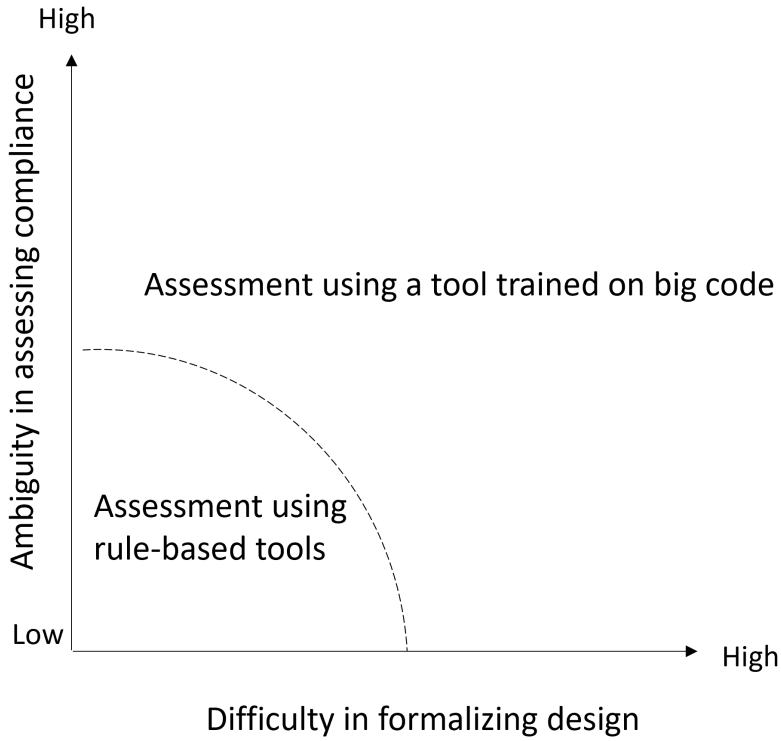


Figure 3.3: As the difficulty in formalizing design requirements and the ambiguity in compliance assessment increase, the availability of big code enables a deep learning approach for automating design review

limitations, but it does point to vital ingredients for a deep learning approach to automate design review. After all, the manual review process, which builds upon two closely related factors – domain expertise which, in turn, leads to nuanced judgments – is quite effective. The issue, of course, is that in the high cadence process that automotive software engineering aspires to, purely manual review cannot wield these crucial factors at scale. However, if discussions in Section 2.2 are any indication, both factors can be addressed using a tool derived from a foundation model that has domain expertise in vehicle application software.

Expert reviewers are likely to have spent a significant amount of time being involved in the process of engineering automotive software, helping them learn fine details of design. If not direct experience in the automotive domain, at the very least, expert reviewers are likely to be familiar with closely related disciplines like embedded software engineering or general C programming. Such experience in individual engineers may be a prized asset, but the emergence of big code – source code from thousands of software engineering projects – means that a bulk of this collective experience is available as data. If millions of pages of text can help train natural language domain generalists like BERT and GPT-3, it stands to reason that millions of files of source code can train

analogous domain generalists in programming language [70]. In fact, recent advances show that foundation language models trained on source code have served as able building blocks for automating tasks as complex as code completion and summarization [27]. Not only does this bode well for our quest to automate design reviews, it also serves as an important source of inspiration.

Foundation models trained on vehicle application software may very well provide the breadth of domain expertise but, in order to automate design compliance assessment, we need the complementary depth of task specialization. Here, as have repeatedly noted, it may not be viable to train a true/false classification head using labeled data. The example in Figure 3.2 shows that an assessment of whether a given implementation of AUTOSAR application software complies with a specific design pattern does not always end up with a true/false answer. Even if the label is not binary, as we reason in the next chapter, it is arguable that boiling down design compliance assessment to a small set of discrete labels can be restrictive. Stated simply, in a task as complex as design review there can be many shades of gray. This is why, instead of training a classifier using annotated data, we use a 'pre-train and calculate' approach to measure design compliance using principled vector arithmetic on language model embeddings. Not only does this avoid the considerable cost of annotation but, as demonstrated shortly, also helps conduct automatic design reviews with levels of nuance comparable to human reviewers. Minimizing the effort spent on manual design review, methods we develop help speed up the automotive software engineering process. This helps achieve the larger objective of increasing the cadence and the quality of deliveries, required by the next wave of automotive software.

Chapter 4

Defining a system for design compliance assessment

'If you think good design is expensive, try bad design', goes the adage. While this observation can headline any design effort, it is certainly a prime motivator in the design of software. Very generally, the process of software design attempts to envision a software solution that meets a given set of requirements [71]. This view, as traced in the previous chapter, continues to hold firm even in the world of automotive software engineering. The classic V engineering process undertakes the design activity using a combination of tools including principles, models, and patterns of architecture and design, etc., that instructs top-down, the eventual implementation of software. Meeting core business requirements may be its primary objective, but software design often aspires further. It tries to address several non-functional concerns and increase the likelihood that the solution operates and evolves sustainably [72]. The expanded set of concerns inevitably complicates the design process, which now becomes an act of trading-off concerns in multiple dimensions, under the shadow of constant uncertainty.

When reasoning a foray of the big data, deep learning approach into the design arena in the previous chapter, we also noted that neural language models pre-trained on large source code corpora have started becoming building blocks for automating a variety of complex programming tasks like code completion and program repair ([73, 74], for example). If such programming tasks, which often require nuanced judgment, can be automated, can a similar approach be applied to automate design tasks? We now take initial steps towards answering this question by investigating a use case in automotive software design compliance. In doing so, we introduce innovations that help automate a crucial and time-consuming design review task, easing the process of software design.

4.1 The 'language' of design in code

The application of neural language models for automating programming tasks is fundamentally based upon the naturalness hypothesis [70], which recognizes that software is a form of human communication. As presciently observed in the concept of 'literate' programming in [75], instead of simply viewing it as an entity that instructs a computer, a program can also be seen as a document that communicates with fellow engineers what we expect the computer to do. This subtle shift in perspective enables us to view programs as yet another communication medium, possessing similar statistical properties as natural language text. As traced exquisitely in [70], this means that neural Programming Language Models (PLMs) pre-trained on code corpora, can exploit such infused elements of human communication to learn a statistical model of programming, just like their natural language counterparts. Such knowledge lies at the foundation of a PLM's ability to automate complex programming tasks. In our attempt to extend PLMs for automating design-related tasks, we therefore start by considering whether design information is also naturally communicated in code.

Echoing the core tenets of literate programming, no matter the domain, any list of properties that characterize well-written code would note that it should be properly structured, readable, and clear. Stated otherwise, code as a software engineering artifact should aspire to be self-explanatory. Which brings up the question - what is the intent behind infusing explanatory elements in source code? Apart from promoting intellectual understanding, programmers generally choose to augment self-explanation in code so that fellow-programmers find it easy to extend. A basic explanatory technique like using well-worded program statements, in a clearly evident sequence, accompanied by lucid natural language comments clearly helps code extension in relatively local scopes. In parallel, carefully wording and characterizing entities like methods, modules, or classes, and the ways in which they relate, interact, and are packaged, promote more global extension. Infusing such explanation, which is largely complementary to program logic, clearly achieves many of the same objectives of a top-down design exercise. In fact, the co-evolution of design and solution – the 'code as design' approach – is itself a natural byproduct of using high-level programming languages [76].

The naturalness hypothesis, if stated differently, recognizes that source code is naturally bimodal. That is, code – when sufficiently well-written – has at least two recognizable channels namely (1) the algorithmic channel that is machine comprehensible and (2) the explanatory channel that is human comprehensible [70]. Then, if the naturally bimodal nature of code is jointly considered with the preceding reasoning about design being a natural part of code, it is reasonable to conclude that its explanatory channel is likely to include information about design. Put simply, irrespective of whether it emerges bottom-up as a result of programming or top-down as a result of an upstream design process, *elements of software design occur naturally in source code*. Given that (1) PLMs successfully understand statistical properties of natural programming, and (2) elements of design occur naturally in source code, we reason that *PLMs pre-trained on large code corpora are likely to understand elements of design*. It is easy to see that such a reasoning has tremendous

potential to serve our larger quest of easing the process of engineering vehicle application software. If design knowledge that naturally occurs in big code can be learned and wielded at scale, it can help address critical factors of experience and nuance that is needed to automate aspects of design review. The purpose of the forthcoming study is to both verify this reasoning and exploit its potential.

4.2 Stating the problem of design compliance

In order to help automate the complex and effort intensive process of assessing design compliance, we envision a system $\mathcal{S}$ that assesses whether a set of query programs/files Q, drawn from a corpus $\mathcal{Q}$, complies with a design pattern $\mathcal{D}$ specified for the corpus. The set of query programs Q, as we shall soon see, are implementations of software components that collectively realize one or more on-board vehicle functions. Building upon the reasoning that naturally occurring design knowledge in large code corpora can be learned and utilized, we further envision that the system $\mathcal{S}$ uses a PLM $\mathcal{F}$ trained on large code corpora as the primary tool. Finally, as denoted in (4.1), we require that a score m calculated by the system provides a measure of compliance.

$$m = \mathcal{S}(Q,\ \mathcal{D};\ \mathcal{F}),\ Q \subseteq \mathcal{Q} \tag{4.1}$$

By thus measuring the compliance of code with a set of design principles, codified as a design pattern (Figure 4.1), the system that we envision should be able to assist, if not automate, manual design review. To construct and evaluate such a system, we pose the following research questions.

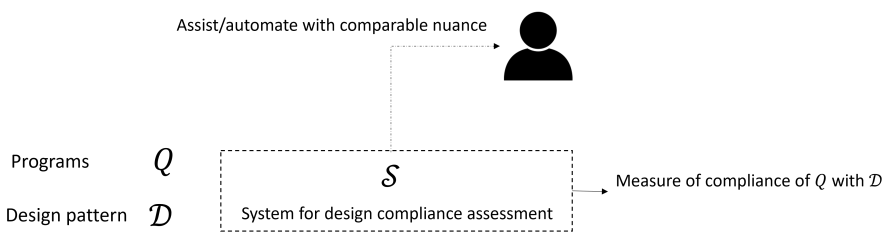


Figure 4.1: We envision a design compliance assessment system $\mathcal{S}$ that can assist or automate manual design review with nuance

RQ1 – Can the system $\mathcal{S}$ for assessing design compliance be constructed using a neural language model trained on code?

RQ2 – Does the assessment improve when the PLM is explicitly provided with information relevant to design pattern $\mathcal{D}$?

RQ3 – Can the measure m be communicated in a way that makes it easy for an architect to understand the compliance of Q with $\mathcal{D}$?

As described in forthcoming chapters in Part I, results from this study show that it is indeed possible to construct such a system for measuring design compliance. Such a neural language modeling approach to automatically assess design compliance has the potential to improve the chances of quickly identifying (and subsequently correcting) design violations, thus promoting faster, yet sustainable, evolution of the code base. Clearly, this can help achieve properties of increased cadence and quality required for the next wave of automotive software.

4.3 The corpus and design pattern studied

The Truck Application Software corpus – To begin understanding the system that we envision, let us first examine the corpus $\mathcal{Q}$ from which query programs are drawn. In this study, we use Truck Application Software (TAS), a corpus of ∼5k files of C-language code, that implements in-vehicle functionality for the Volvo Group's truck platforms. As an unvarnished corpus of vehicle application software, principles of software design adopted in TAS stem mainly from AUTOSAR. This, as noted in Section 2.1, means that the fundamental unit of application software that TAS contains, and the principal design abstraction that it builds upon, is the SWC. The collection of ∼200 SWCs in TAS implement tens of applications that deliver crucial end-user value to customers. This is why considerable measures need to be taken to ensure that the application software system, which these SWCs represent, is designed to evolve sustainably. Foremost among the design priorities is the regulation of dependencies between SWCs. When dependencies are sufficiently regulated, existing components can be modified, and new components can be introduced, with minimal disturbances to the overall system. Such regulation also ensures that scenarios like the spaghetti coupling illustrated in Figure 3.2 are avoided. The minimalist stance taken by AUTOSAR, however, means that there are not many safeguards in the essentially flat organization of components in its application layer that help minimize unnecessary dependencies. In order to fill this gap, software architects at the Volvo Group have defined *Controller-Handler* (CH), which is the design pattern $\mathcal{D}$ which we focus upon in this study.

The Controller-Handler design pattern – One simple way to illustrate this design pattern would be by considering an example application in TAS – roof hatch control – and its design. Trucks are sometimes equipped with a hatch on the roof (Figure 4.2), which the driver can control to adjust the flow of air and the amount of ambient light. The hatch is equipped with necessary motors that effect this control based upon driver input. The key design principle, used in TAS, to implement such a function is the separation of the core logic for hatch adjustment, the *Controller*, from the logic that handles the motors, the *Handler*. The main reason behind calling for such separation is to decouple hardware, specifically sensors and actuators, from the control logic.

An illustration of the otherwise proprietary roof hatch control application, one that follows the CH pattern, is shown in Listings 4.1 and 4.2. The controller (Listing 4.1) begins by reading the driver's request into a `Request_Type` object.

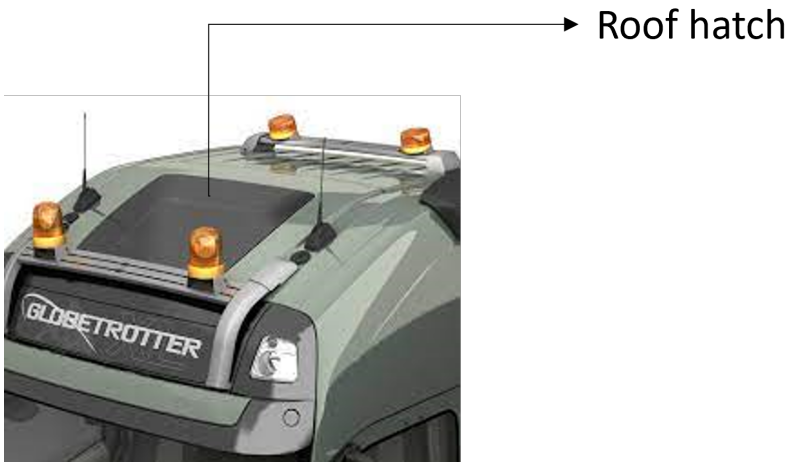


Figure 4.2: Adjustable roof hatch in a Volvo FH truck

In this example, we only show two possible requests, `RoofHatch_rqst_Open` and `RoofHatch_rqst_Close`. Upon reading the request, the controller maps them respectively to `Open` and `Close` commands of type `RoofHatch_Command`. The command is then transmitted to the handler through the AUTOSAR RTE using an `RTE_write` call. It is important to note that the controller only contains code for reading the driver input and issuing commands for operating the roof hatch. The code for issuing commands in `activateMotors` includes additional logic to (1) avoid indefinitely issuing the same command, and (2) prevent a rapid reversal of motor state which could cause undesirable inductive effects. Once the controller issues the command, the corresponding handler (Figure 4.2) reads it using an `RTE_read` call, and actuates the motor accordingly. Specifically, the motor is controlled using `Rte_call`, which operates digital output pins `IOHW_ON` or `IOHW_OFF` where the motor is physically wired. The essence of separating controller and handler software components is revealed by the nature of the handler code. The motor used for adjusting the roof hatch is physically located somewhere close to the hatch itself and is hardwired to pins on an ECU located in the vicinity. The handler, therefore, needs to be deployed on this particular ECU and use the designated pins to control the motor, imposing strict conditions on deployment. In contrast, the application logic in the controller – having been decoupled from the motor hardware – is not bound to a specific ECU and has relative freedom of deployment. A virtualized RTE that allows seamless communication between SWCs deployed on different ECUs is therefore an important element in realizing such a design.

```
1 // DATA TYPES
2 typedef struct {
3    RequestType_T                  Request_Type;
4 } RoofHatch_Ctrl_In_T;
5
6 typedef struct {
7     RoofHatch_T                   RoofHatch_Command;
8 } RoofHatch_Ctrl_Out_T;
9
```

```
10 // VARIABLES
11 static RoofHatch_T                m_previousCommand;
12 static RequestType_T              m_previousRequestType;
13 static Counter_T                  m_protectionCounter;
14
15 // PRIVATE FUNCTION PROTOTYPES
16 static void readAllData(RoofHatch_Ctrl_In_T *a_in);
17 static void writeAllData(const RoofHatch_Ctrl_Out_T *a_out);
18
19 // PRIVATE FUNCTIONS
20 static void readAllData(RoofHatch_Ctrl_In_T *a_in)
21 {
22    Std_ReturnType             readStatus;
23
24    readStatus  = Rte_Read_Request_Type_Request_Type(&(a_in->
       Request_Type));
25    if (readStatus != RTE_E_OK  )   {
26          a_in->Request_Type = m_previousRequestType;
27    }
28 }
29
30 static void writeAllData(const RoofHatch_Ctrl_Out_T *a_out)
31 {
32    (void)Rte_Write_RoofHatch_Command_RoofHatch_Command(a_out->
       RoofHatch_Command);
33 }
34
35 static void inactivateMotors(RoofHatch_Ctrl_Out_T *a_out)
36 {
37     m_protectionCounter = 0;
38     m_previousCommand = NO_COMMAND;
39     a_out->RoofHatch_Command = NO_COMMAND;
40 }
41
42 static void activateMotors(RoofHatch_T cmd, RoofHatch_Ctrl_Out_T *
       a_out)
43 {
44    if(m_previousCommand == cmd){
45       if(m_protectionCounter > THRESHOLD){
46          m_previousCommand = NO_COMMAND;
47          a_out->RoofHatch_Command = NO_COMMAND;
48       }else {
49          m_protectionCounter++;
50          a_out->RoofHatch_Command = m_previousCommand;
51       }
52    }else {
53       if(m_protectionCounter > THRESHOLD){
54          a_out->RoofHatch_Command = m_previousCommand;
55       }else {
56          m_protectionCounter=0;
57          m_previousCommand = cmd;
58          a_out->RoofHatch_Command = cmd;
59       }
60    }
61 }
62
63 // PUBLIC FUNCTIONS
64 FUNC(void, RTE_ROOFHATCHCTRL_APPL_CODE) RoofHatch_Ctrl_Init(void)
65 {
66    m_previousCommand = NO_COMMAND;
67    m_protectionCounter = 0;
68 }
```

```
69
70 FUNC(void, RTE_ROOFHATCHCTRL_APPL_CODE) RoofHatch_Ctrl_run(void)
71 {
72    RoofHatch_Ctrl_In_T  m_in;
73    RoofHatch_Ctrl_Out_T m_out;
74
75    readAllData(&m_in);
76
77    if(m_in.Request_Type == RoofHatch_rqst_Open){
78       activateMotors(OPEN,&m_out);
79    }else if(m_in.Request_Type == RoofHatch_rqst_Close){
80       activateMotors(CLOSE,&m_out);
81    }else {
82       inactivateMotors(&m_out)
83    }
84
85    writeAllData(&m_out);
86 }
```

Listing 4.1: An illustration of a possible roof hatch contoller implementation

```
1  // DATA TYPES
2  typedef struct {
3     RoofHatch_T                      RoofHatch_Command;
4  } RoofHatchHdlr;
5
6  // PUBLIC FUNCTIONS
7  FUNC(void, RTE_ROOFHATCHHDLR_APPL_CODE) RoofHatch_Hdlr_Init(void)
8  { }
9
10 FUNC(void, RTE_ROOFHATCHHDLR_APPL_CODE) RoofHatch_Hdlr_run(void)
11 {
12    RoofHatchHdlr            Handler_obj;
13
14    Handler_obj.RoofHatch_Command =
       Rte_Read_RoofHatch_Command_RoofHatch_Command(&Handler_obj.
       RoofHatch_Command);
15
16    if (Handler_obj.RoofHatch_Command == OPEN){
17         (void)Rte_Call_RoofHatchHdlr_Actuator_setOpen(IOHW_ON);
18         (void)Rte_Call_RoofHatchHdlr_Actuator_setClose(IOHW_OFF);
19    }else if(Handler_obj.RoofHatch_Command == CLOSE){
20         (void)Rte_Call_RoofHatchHdlr_Actuator_setOpen(IOHW_OFF);
21         (void)Rte_Call_RoofHatchHdlr_Actuator_setClose(IOHW_ON);
22    }else{
23        (void)Rte_Call_RoofHatchHdlr_Actuator_setOpen(IOHW_OFF);
24        (void)Rte_Call_RoofHatchHdlr_Actuator_setClose(IOHW_OFF);
25    }
26 }
```

Listing 4.2: An illustration of a possible roof hatch handler implementation

Such an act of decoupling has several implications on the implementation, deployment, and installation of applications like roof hatch control. First, the act of decoupling controller from handler logic allows a more efficient use of the generally limited computational resources available on-board. The motor control logic in the handler, which is typically some form of on/off toggling or pulse-width modulation, is not very compute intensive and can be placed on resource and cost-efficient I/O ECUs. The controller logic, though not very intensive in

this case, can be pooled with other applications in relatively powerful computational ECUs. Concentrating computational logic into a low number of general purpose embedded computers is increasingly being recognized not only as being cost-efficient system but an act that hastens the cadence of delivering new functionality [77]. Designing applications using the CH pattern clearly helps achieve this. Second, decoupling the controller from the handler is necessary to support a wide variety of product configurations required to fit several transport operations and market segments. Take, for instance, the cab of the truck where the roof hatch is installed. Based upon whether it is sold in Europe or North America, the design of the cab of Volvo Trucks differs significantly (Figure 4.3). Then, depending upon the specific cab geometry, motors may have different alignments and may need to articulate the hatch differently. If the roof hatch application is properly designed by decoupling controller logic from the handler logic, then the variability in cab geometry only affects the handler. The controller remains intact and can be simply reused across both variants. Third, if the roof hatch control application properly implements CH, it simplifies making changes in the system. If a new actuation technology arises and a new kind of motor is introduced, most of the impact should be limited only to the handler logic.

Figure 4.3: Variation in cab geometry in Volvo Truck products offered in North America (left) and Europe (right)

We may have used roof hatch control as an example, but it is clear that many of the design issues that we discussed are applicable to pretty much any control application that uses sensors and actuators. Considering the amount of automatic control functions in Volvo Trucks, it turns out that CH is the most prevalent design pattern used in the TAS corpus of vehicle application software. This is precisely why we focus upon this pattern when developing our automatic compliance assessment system. Formally (see Figure 4.4), the CH pattern advocates the implementation of an in-vehicle control application using a set of SWCs $P = \{C, H_1, H_2, ..., H_N\}$. Here, the *Controller* component C, implements the core control logic, while *Handler* components H_i implement hardware-specific logic. In practice, since the handler components are usually independent of each other, the CH design pattern can be defined as applying to each pair $P = (C, H_i)$ of controller and handler SWCs used to realize the overall application. Apart from roof hatch control, applications in TAS that adopt this design pattern include washer and wiper control, exterior lights control, and mirror heating control.

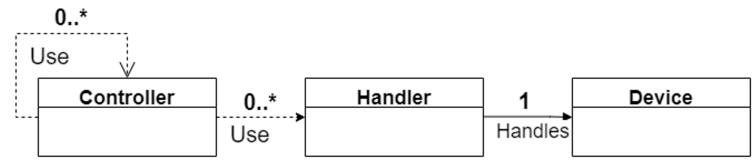


Figure 4.4: Controller-Handler software design pattern for automotive control systems

While the reasoning behind the CH pattern is intuitive, compliance with the pattern is not always simple. In the roof hatch example, if we focus upon the motor handling and application logic alone, it may be easy to conclude that the former goes into a handler SWC and the latter into a controller SWC. If we expand the focus to, say, the electrical safety logic in the `activateMotors` function, there is no easy answer on how this can straddle the CH divide. Placing all electrical safety logic in the handler may fit well with its hardware-centric profile, but this could lead to some safety logic being duplicated across several handler variants. This could be one reason why, like our example, such code could be placed in the controller. Another reason for doing so could be that electrical safety needs to factor other aspects in the system that lie far beyond the scope of motor control. More generally, considering the fact that the CH pattern is defined fairly loosely – as most design patterns are – there is sufficient room for ambiguity in interpretation. It is also important to note that tightening the definition may not always be useful. Worst case, patterns that are rigidly defined may simply not have enough margin for practical implementation and may simply remain unused. With a need to provide sufficient margin for flexibility, there is always bound to be some ambiguity in design patterns. This, as we noted in the previous chapter, is what makes design compliance complex. During manual review, experienced architects can not only hone in on issues that are known to be contentious, but they can also weigh it against relevant practice to make nuanced judgments about compliance. In our attempt to measure design compliance using a PLM, we consciously try to reflect some of these important factors.

Chapter 5

Building a system for design compliance assessment

Having fixed the query corpus $\mathcal{Q}$ as TAS, and the design pattern $\mathcal{D}$ as Controller-Handler for the study, we restate our objective. We aim to construct a system $\mathcal{S}$ that assesses whether an ordered pair $Q = (X, Y),\ X, Y \in \mathcal{Q}$ of SWCs comply with the Controller-Handler design pattern. Though either of the SWCs in this pair can be realized using multiple software modules (or programs), at this point it is simpler to consider the case where each SWC is realized as one program. We relax this condition at a later point. The following sections describe the process of constructing the compliance assessment system $\mathcal{S}$ that we envision in (4.1).

5.1 Constructing a system for assessing design compliance

Pre-training a PLM – In this work, we consider a program $X = (t_1, t_2, ..., t_N)$ to be a source code file containing a sequence of tokens t_i. We then define a PLM to be a language representation model of the form $\mathcal{F} : X_M \rightarrow X$ pre-trained as a masked language model, first introduced in BERT [51]. One practical issue that we face when building language models is that the program, which is fed-forward into the model, is a discrete sequence of words. DNNs like the Transformer encoder in BERT, on the other hand, only accept vectors (or more generally tensors) as inputs. Unlike image or time series data, which are usually easy to vectorize, the process of vectorizing programs is relatively complex. Since vectorization is an important step that allows DNNs to process data modalities that are not naturally vectors, it is useful to briefly examine the steps we use to vectorize code. As shown in Figure 5.1, we follow a two-step vectorization process

prescribed, among others, by [78]. First, the raw code is subjected to a tokenization process to split it into a sequence of word tokens. A naive version of tokenization would be to treat every character sequence separated by a whitespace as a token. This, however, turns out to be quite inefficient in our case because code tokens tend to agglomerate multiple words. In Figure 5.1, the highlighted token `pageset1_map` is one example. Taken separately, its subwords `page`, `set` , and `map` are likely to appear frequently in a code corpus. The combined sequence, on the other hand, can be rare. This is why contemporary language processing pipelines turn to subword tokenization, with Byte Pair Encoding (BPE) [79] being a popular way to achieve this. Using an iterative process that computes the frequency of subwords and their combinations, BPE splits rare words into frequently occurring subwords. In the example, `pageset1_map` is split into four subwords `page, set, 1_, map`. Tokenization using BPE thus results in a compact vocabulary, or a dictionary, of possible code tokens, where common subwords appear intact while larger words are broken into constituent subwords. Such a split, which results in a granular set of commonly occurring subwords, also helps the model to generalize. When testing with unseen code it is quite likely that the model comes across a rare agglomerated word which did not appear in the training corpus. In such cases, tokenization into subwords increases the chance that the subwords themselves are part of the training vocabulary. Upon tokenization, the second preprocessing step is to substitute each subword with its (integer) dictionary subword entry. Then an embedding layer, which is essentially a learnable look-up table, at the input of a DNN converts these integer sequences into semantically consistent real-valued vectors. Going beyond code tokens, the main intention behind this examination is to point out that, by adopting an analogous process, other domains can be similarly vectorized and fed-forward into DNNs. This is especially noteworthy for automotive software engineering where artifacts beyond code, like commit histories, UML models, network topologies, etc., are involved. With appropriate vectorization, a deep learning toolkit could very well be applied to solve problems in all these data modalities.

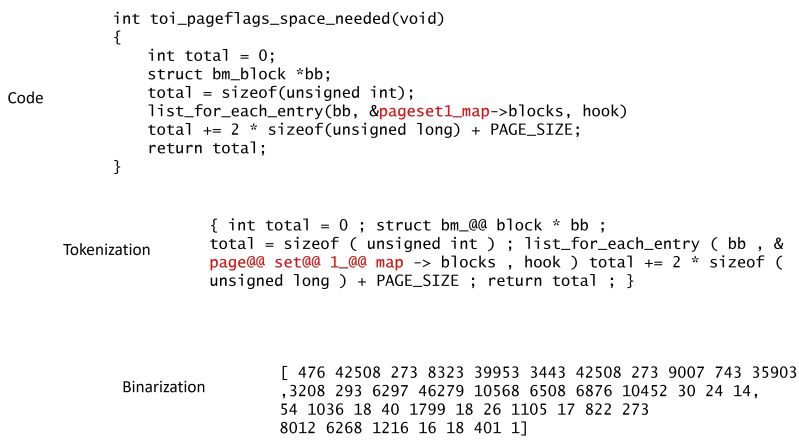


Figure 5.1: The two-step process of vectorizing source code

After tokenization and binarization the program $X = (t_1, t_2, ..., t_N)$ is now in

a format that can be readily processed by a DNN. Then, as we previously saw with a natural language example in Figure 2.5, borrowing a proven technique in training foundation models of language, the core task we use for pre-training the PLM is cloze or Masked Reconstruction (MR) shown in (5.1)[1]. In this task, the PLM is provided a masked program X_M which, using the BERT masking recipe, is produced by replacing a uniform randomly selected fixed fraction of tokens in X with a mask token $\mathbf{t}$. The model is then tasked to recover tokens in masked positions, as a result of which it learns contextual meanings of programs.

$$\begin{aligned} MR(X;\mathcal{F}) = \ & \mathcal{F}(X_M)[j] == X[j], j \in J \\ & J = \{i : t_i = \mathbf{t}, t_i \in X_M\} \end{aligned} \tag{5.1}$$

Since our aim is to assess design compliance in TAS, which is a C-language corpus, we pre-train a monolingual PLM on C code. As pre-training corpus $\mathcal{P}$, we use ∼75M files of C code derived from the GitHub public dataset[2]. The model is then pre-trained by minimizing the objective shown in 5.2.

$$\mathcal{F} := \underset{\hat{\mathcal{F}}}{\operatorname{argmin}} \quad \mathbb{E}_{X \in \mathcal{P}} \quad MR(X;\ \hat{\mathcal{F}}) \tag{5.2}$$

The larger system $\mathcal{S}$ that we envision in (4.1) aims, of course, to assess whether a pair of programs Q complies with the CH design pattern $\mathcal{D}$. Practically, however, feeding an entire program into the PLM is an issue because C programs tend to be long. The average length of a C program is ∼5k subwords in the GitHub corpus and ∼7k subwords in the TAS corpus. The Transformer architecture [80], which is the mainstay of several previously reported foundational PLMs like CodeBERT [81], is typically configured to handle input sequences of length 512-1024. This is because the vanilla self-attention mechanism is of quadratic compute and memory complexity, which makes it impractical for longer input sequences. To be able to assess long programs, we therefore base the PLM $\mathcal{F}$ on the more efficient Reformer [82] architecture. Combining locality-sensitive-hashing and reversible residual layers, the Reformer handles long sequences much more efficiently. By configuring the input sequence length to 8192, we are able to feed around 80% of programs in the TAS corpus into the Reformer-based $\mathcal{F}$ intact, with manageable memory and computational complexity. Programs longer than 8192 tokens are truncated to this length. The Reformer encoder $\mathcal{F}$ of ∼180M parameters with 6 self-attention layers (each with 8 heads) was trained from scratch on 16 Nvidia Tesla V100 GPUs until the MR accuracy on a validation set of 5k files reached 95.12%.

Assessing compliance by manual review – Recalling the CH design pattern described in Section 4.3, let us now consider how a human architect would assess whether a query pair (X, Y) of programs complies with this pattern. The architect would normally do this by reviewing the code (or the 'naturalness') of the programs and assess whether X and Y respectively embody the core principles of a controller and its associated handler. It is, however, important to note

[1] In practice, a differentiable cross-entropy loss is used

[2] https://console.cloud.google.com/marketplace/details/github/github-repos

that the CH pattern defines expectations jointly on the pair and not on the controller and handler programs individually. One simple illustration of the joint nature of expectations is that commands issued by the controller are received and processed by the handler, and not the other way around. The handler may, however, respond to certain commands in order to, say, report status. Therefore, in order to check properties jointly, an intermediate step that architects inevitably take is to juxtapose related parts from the pair (often mentally) and then conduct the assessment. We find it useful to refer to such a juxtaposition as XY – the 'jointness' of the two programs. It is on this representation XY that the architect assesses whether principles of the CH pattern are complied with.

On this abstract, joint, and mentally held, representation, there are several markers of (non-)compliance that architects could look for. If we use the illustrative roof hatch code in Listings 4.1 and 4.2 as an example, signs of violation that architects anticipate include checking if the controller code makes references to actuators, IO, PWM, etc. A corresponding violation could be the handler including, say, code that reads driver commands. Architects can also check if the interface for the handler is a pure abstraction of the hardware interface for the hatch motors and does not contain extra logic, for instance, to protect the motors from over usage. Logic of this nature is better placed in the controller. This simple list of markers, while certainly not exhaustive, is sufficient to illustrate the sheer range of concerns that need to be addressed. Not only is manually assessing the jointness XY for signs of deviation clearly difficult, but there are several factors that complicate the process further. First, any instance of the CH pattern is certain to contain code that falls outside the purview of the pattern itself. Roof hatch control could also include code for diagnostics or logging, aspects of which are less relevant to the CH design pattern. The presence of code which undertakes activities other than controlling or handling means that an architect will have to identify and assess tenets of the pattern in a noisy or diluted context. Second, as a relatively loose pattern, CH can be realized in several styles. An architect would therefore need to judge whether a given style of implementation is legitimate. Third, it is practically difficult to construct an ideal realization against which the query programs can be assessed. Usually, the architect relies on a subjective mental model of the pattern, which is not only difficult to explicitly state, but also affects the objectivity of the assessment. Addressing these concerns requires nuanced judgment, which is precisely what a human expert applies. In using a PLM as an alternative to a human expert, we now describe how we address some of these concerns.

Assessing compliance using program embeddings – The main tool we use for PLM-based compliance assessment is the *program embedding* e_X which, as we saw in the discussion on representation learning in Section 2.2, is a vector representation of the program X that reflects its semantic properties. However, as shown in [83], there are different ways to extract embeddings from contextual language models, each capturing different aspects of information. After some trial and error, we empirically decide to use the normalized output of the final (6th) layer of $\mathcal{F}$, shown below, as the program embedding.

$$e_X = \frac{\mathcal{F}_6(X)}{||\mathcal{F}_6(X)||} \tag{5.3}$$

The PLM $\mathcal{F}$ is pre-trained on the masked reconstruction task on millions of program examples. It is therefore reasonable to expect that the embedding e_X is a fairly robust representation of the program X and is insensitive to minor semantic variations. Thus, the process of assessing whether (X, Y) complies with the CH pattern is done, not in the code space, but in a vector space using embeddings (e_X, e_Y). While this pair of embeddings sufficiently represent the programs individually, an additional representation is needed to address the joint perspective XY. One simple model to capture the jointness of a pair of programs would be the offset, or the difference vector, between their embeddings.

$$r_{XY} = e_Y - e_X \tag{5.4}$$

At this point in the discussion, a pertinent question to pose would be – why choose the difference vector, and not some other representation, to capture jointness? The answer, as we have alluded to earlier, is that this choice of representation is inspired by the principle of embedding regularity. Regularity in language representation refers to the property that embeddings of pairs of inputs, related by the same concept, are usually arranged in a recognizable geometry. Most famously, [84] studied this property on embeddings of analogous pairs of words. Consider pairs of words $(Man, Woman)$ and $(King, Queen)$, which are related by the same concept – each word in a pair is the male/female form of the other. Given such a quartet of related words, [84] showed that the word2vec language model learns embeddings such that $e_{King} - e_{Man} + e_{Woman} \approx e_{Queen}$. That is, language models tend to reflect a proper understanding of the analogical relationship between pairs of words by arranging their embeddings in a parallelogram (Figure 5.2). Going beyond words, [85] observed that embeddings of pairs of sentences that are related by the same concept show a similar parallelogram geometry. Extrapolating this observation beyond words and sentences, we hypothesize that *embedding regularity extends to programs*. We know that (X, Y) are query programs, and we seek to assess their compliance with the CH design pattern. Let us also assume that we have access to some benchmark implementation of the CH pattern $(\mathcal{X}, \mathcal{Y})$, where $\mathcal{Y}$ is a perfectly compliant handler to the controller $\mathcal{X}$. Then, if the query program Y is a similarly compliant handler to its paired controller X, then it is reasonable to expect that embeddings of the quartet (X, Y) and $(\mathcal{X}, \mathcal{Y})$ form a parallelogram (Figure 5.2). Conversely, if the difference vector r_{XY} also serves as an effective offset for the benchmark CH pair, meaning if $e_{\mathcal{X}} + r_{XY} \approx e_{\mathcal{Y}}$, then (X, Y) aligns with the benchmark implementation of CH. This reasoning, based upon the principle of embedding regularity, provides sound premise for choosing the difference vector as the joint representation for programs (X, Y). It is important to note that, even standalone, the difference vector r_{XY} is a representation of jointness that is as reasonable as any other. The fact that it plays a pivotal role in the geometry of embedding regularity lends additional credence. This is the principled reasoning that drives the choice of the difference vector r_{XY} as a joint representation, and alignment with the parallelogram geometry as the measure compliance assessment.

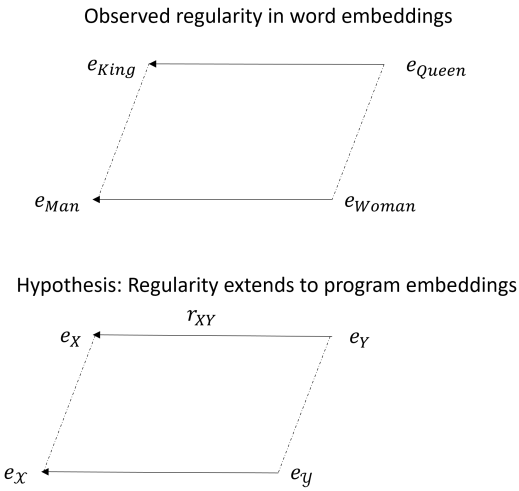


Figure 5.2: Extending embedding regularity from words to programs

Practical assessment of compliance using regularity – We may have arrived at a joint representation in a principled manner, but the problem, of course, is the construction of a benchmark. Should there exist a benchmark vector r, drawn perhaps from the ideal implementation $(\mathcal{X}, \mathcal{Y})$, capturing the required level of jointness as prescribed by the CH design pattern, then the assessment of design compliance reduces to checking the alignment between r_{XY} and r in the embedding space. Put otherwise, if r serves as an effective offset vector between the embeddings of the pair of programs (X, Y), i.e., if (5.5) is satisfied, then this pair comes close to realizing the principles specified by the CH pattern.

$$\hat{e}_Y := e_X + r \approx e_Y \tag{5.5}$$

Clearly, we do not have access to the ideal implementation of CH, and the corresponding ideal representation of jointness r. Therefore, as a practical alternative, we assess compliance with the *average* realization of the CH pattern, extracted from a set of known instances. That is, given a set $V = \{(C, H)\}_{i=1}^{N}$ of program pairs from the TAS corpus that are known to implement the CH pattern, we define a benchmark of average jointness (5.6), that averages offset vectors from pairs in V. If this benchmark serves as an effective offset for query programs (X, Y), satisfying (5.5), then this pair comes close to realizing the average implementation of the CH pattern seen in $|V|$ known instances. Apart from being an intuitive and practical benchmark that captures the average state of implementing the CH pattern, by pooling common traits from known instances, r serves as a stronger signal for the CH pattern compared to individual instances, where signatures of the pattern are likely to be diluted.

$$r := \frac{1}{|V|} \sum_{(C,H) \in V} e_H - e_C \tag{5.6}$$

As shown using an example in Figure 5.3, serving as an offset vector from e_X, if r is able to predict a handler embedding $\hat{e}_Y$ that is reasonably close to its actual counterpart e_Y, programs (X, Y) are likely to comply with the CH pattern. Such closeness between $\hat{e}_Y$ and e_Y is easily measurable using the cosine similarity between these two vectors. With this method, the assessment system for the CH design pattern $\mathcal{D}$, originally envisioned as (4.1), can be rewritten as follows.

$$m = \mathcal{S}((X, Y), \mathcal{D};\ \mathcal{F}, V) = \frac{e_Y \cdot (e_X + r)}{||e_Y||_2\ \ ||e_X + r||_2} \tag{5.7}$$

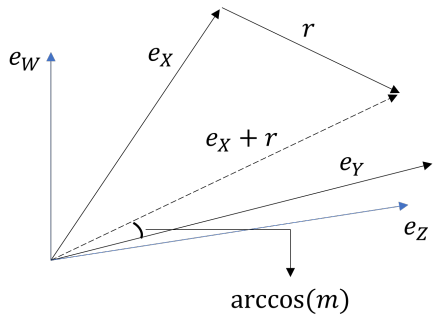


Figure 5.3: The alignment between the actual handler embedding e_Y and the predicted one $e_X + r$ reflects compliance. Vectors e_W, e_Z illustrate embeddings of programs $W, Z \in \mathcal{Q}$

Using cosine similarity as the metric measure – standard practice for comparing language model embeddings – results in $-1 \leq m \leq 1$. Then, $m \approx 1$ means that the predicted handler embedding $\hat{e}_Y$ closely aligns with that of the actual handler e_Y, indicating compliance. Thus, as a way to assess compliance with the CH design pattern, we substitute a complex code review process with a vastly simpler comparison of embeddings extracted from a neural language model.

Easing interpretation of compliance – With cosine similarity, while it is clear that $m = 1$ indicates perfect compliance, $m = 0$ indicates marked non-compliance respectively, and $m = -1$ indicates compliance, but with the pair (X, Y) ordered incorrectly, such perfect scores are rare. Scores in between, which are most likely in practice, are difficult to interpret. In order to provide an intuitive human-readable assessment, we convert similarity m into a *rank* k. The discrete rank k means that the predicted handler embedding is the k^{th} most similar to that of the actual handler when compared to the embeddings of all other programs in the TAS corpus. The best indicator of compliance is a rank of $k = 1$ when, among all programs in the TAS corpus $\mathcal{Q}$ (excluding the controller X) there is no better handler than Y for the controller X, as assessed by the benchmark r. Conversely, a rank of $|\mathcal{Q}| - 1$ means that the predicted embedding is least similar and any other program in the TAS corpus is a better handler than Y. This is the worst indicator of compliance. While the rank may be a more interpretable measure, its value is now dependent upon the spread of embeddings around e_Y. In the example shown in Figure

5.3, even if the prediction is reasonably good, it is of rank $k = 2$, since there is another program $Z \in \mathcal{Q}$, whose embedding is closer to that of the actual handler program Y. If there is considerable clustering in the close neighborhood of e_Y, then even a good prediction is unlikely to result in a rank close to 1. We therefore use a simple rule of thumb, where if the predicted embedding lies within 10% of embeddings most similar to e_Y, we define the assessment $l =$ `True` that the query (X, Y) complies with the CH pattern. If the predicted embedding lies among those of 90% of the least similar programs, we label the pair as non-compliant. The discrete rank k, in addition to a true/false binary assessment of compliance l, eases human comprehension of our PLM-based process of assessing design compliance. Putting all of this together, the complete process of compliance assessment is described in Algorithm 1. Since this is a procedure that measures design compliance, we refer to it as DECO.

Algorithm 1: Procedure for measuring design compliance, DECO

Parameters : Test input (X, Y), PLM $\mathcal{F}$, TAS corpus $\mathcal{Q}$, known instances of the CH pattern V

Function $M(e_A,\ e_B)$**:**
  $m = \frac{e_A . e_B}{||e_A||_2 \ \ ||e_B||_2}$
  return m

Function $\mathcal{S}(X, Y; \mathcal{F}, V)$**:**
  `/* Note:` $e_X = \mathcal{F}_6(X)/||\mathcal{F}_6(X)||$ `*/`
  $r = \frac{1}{|V|} \sum_{(C,H) \in V} e_H - e_C$
  $c = [M(e_Z,\ e_X + r) :\ Z \in\ \mathcal{Q} \setminus \{X\}]$
  $k =$ indexof (sort(c), $M(e_Y,\ e_X + r)$) `// rank`
  $l = k \leq 0.1 * |\mathcal{Q}|$ `// binary assessment of compliance`
  return k, l

5.2 Experiments in assessing design compliance

Query (X, Y) **and benchmark programs** V – The objective of the DECO algorithm is to check whether a pair of query SWCs (X, Y) complies with the average realization of the CH pattern seen in a separate set V of known instances. With the help of architects who are familiar with the TAS corpus, we first identified **21** known instances of the CH pattern and curated them into a set $\mathcal{V}$. Then, we designed a controlled experiment by selecting two types of queries.

• *The positive query* – where the query $Q^+ \in \mathcal{V}$ is known to be an implementation of the CH pattern that is likely to satisfy the condition specified in (5.5). Meaning, Q^+ is a pair of programs that DECO is expected to evaluate with a rank k close to 1 and a binary compliance assessment of $l =$`True`. The benchmark set in this case is $V = \mathcal{V} \setminus \{Q^+\}$, which is all known instances of the pattern excluding the instance chosen as the test input.

• *The negative query* – where the query $Q^- \in \mathcal{Q} \setminus \mathcal{V}$ is known to not implement the CH pattern and is therefore unlikely to satisfy (5.5). Therefore, DECO

should ideally evaluate this query with a rank k close to $|\mathcal{Q}| - 1$ and a binary compliance assessment of $l =$`False`. Here, the benchmark set $V = \mathcal{V}$ includes all known instances of the CH pattern in the TAS corpus. Since we expect negative queries to perform poorly during the assessment, they help establish a baseline for the evaluating the accuracy of the assessment process.

Consider a pair of SWCs $(C, H) \in \mathcal{V}$, that is known to implement the CH pattern. While it is most straightforward to implement each SWC in the pair as one program, this is not always practical. As discussed previously, and as shown in Figure 3.1, some SWCs include a lot of functionality in which case it is necessary to split its code into several programs or files. Practically, therefore, the SWCs are of the form $C = \{C_1, C_2, ..., C_M\}$ and $H = \{H_1, H_2, ..., H_N\}$, each of them being implemented using multiple programs. This complicates the assessment process since the system $\mathcal{S}$ is designed only to handle a pair of programs and not a pair of sets. A simple way to circumvent this limitation is to 'unroll' the set $\mathcal{V}$ into a Cartesian product set as follows.

$$\mathcal{V}^* = \{(c, h) : \ c \in C, h \in H \ : \ (C, H) \in \mathcal{V}\} \tag{5.8}$$

For every known instance of the CH pattern $(C, H) \in \mathcal{V}$, the product set $\mathcal{V}^*$ pairs each program in the controller SWC C with every program in the handler component H. This process results in a total of **63** pairs, which we used as likely queries in our experiments. By drawing queries $Q^+ \in \mathcal{V}^*$, the advantage is that we exhaustively present all combinations in a paired form that is suitable for assessment using (5.7). The disadvantage is that, even if at the component level every pair $(C, H) \in \mathcal{V}$ is a known instance of the CH pattern, not every pair $(c, h) \in \mathcal{V}^*$ at the program level is a 'true' controller-handler pair that implements elementary aspects of the CH pattern. Considering that several instances of the CH pattern are implemented using multiple programs and that the assessment system is currently designed to work only with a pair of programs, we accept the risk and loosen the definition of the CH pattern. Every pair in the product set $\mathcal{V}^*$ is considered as a true pair and is presented as a positive case for testing, while also being used to calculate r. Negative queries Q^- are simply drawn by picking two random programs from TAS as long as neither of them appear in $\mathcal{V}^*$.

The PLM $\mathcal{F}$ – As the heart of the automated compliance assessment system, the neural PLM $\mathcal{F}$ can be seen as the machine counterpart of a human architect who conducts the same assessment manually. With such an analogy, we now reason about the level of information with which $\mathcal{F}$ is trained and its relation to the quality of assessment. The model pre-trained using (5.2) on a C-language corpus extracted from GitHub – which we now denote as $\mathcal{F}_A$ – is a C-programming expert. Using this model is akin to asking a human expert in C-programming, but one who has no experience in automotive application design and development, to assess compliance with the CH pattern. While it is not impossible for such an expert to conduct this assessment, it is reasonable that an awareness of relevant domain and design concepts would ease the process. To a C-programming expert, we contend that such awareness can be introduced in three stages. The first stage would be to increase awareness about the automotive-domain, i.e. the pattern of token usage (its naturalness) in its

application code. Second comes design-related knowledge, mainly the concept of SWCs, which is fundamental to the definition of the CH pattern. Third, would be the concept of controllers and handlers, the subjects of assessment. Like [86], we achieve the first stage – improving domain-familiarity – by simply continuing to pre-train $\mathcal{F}_A$ on code from TAS. The second stage requires inducing the knowledge of a SWC – a set of programs that jointly realize functionality. We do this by first assembling a set $C = \{(A, P, N)\}_{i=1}^{M}$ of programs from TAS, such that A and P belong to the same SWC, while N belongs to a different SWC. Then, we use the triplet loss to cluster embeddings of programs that belong to a SWC, while keeping those of programs from different SWCs further apart. To simultaneously ensure that this SWC-based clustering does not majorly disrupt the embedding geometry, and to impart domain familiarity, we combine the MR task on the TAS corpus with SWC-clustering as shown below.

$$\begin{aligned} \mathcal{F}_B &= \underset{\mathcal{F}}{\operatorname{argmin}} \;\; E_{(A,P,N)\in C} \;\; TR(A, P, N;\; \mathcal{F}) + MR(A;\; \mathcal{F}) \\ TR(A, P, N;\; \mathcal{F}) &= \max[(||e_A - e_P||^2 - ||e_A - e_N||^2), 0] \end{aligned} \tag{5.9}$$

The resulting fine-tuned model $\mathcal{F}_B$ is thus more familiar with domain and design concepts related TAS in comparison to $\mathcal{F}_A$. For the third stage of inducing knowledge about controller and handler programs, we follow a similar approach of encouraging the PLM to respectively cluster these programs by type. To achieve this, we assemble (1) a set $D_C = \{(C_1, C_2, A)\}_{i=1}^{M}$ with C_1 and C_2 being controllers and A being a non-controller program from the TAS corpus, and (2) a set $D_H = \{(H_1, H_2, B)\}_{i=1}^{N}$, with H_1 and H_2 being handler programs and B being a non-handler program. We then fine-tune $\mathcal{F}_B$ using the triplet loss on the combined set $D = D_C \cup D_H$, resulting in a model $\mathcal{F}_C$ that is aware of the concept of controllers and handlers.

$$\mathcal{F}_C = \underset{\mathcal{F}}{\operatorname{argmin}}\, \mathbb{E}_{(A,P,N)\in D} \;\; TR(A, P, N;\; \mathcal{F}) + MR(A;\; \mathcal{F}) \tag{5.10}$$

By assessing design compliance using models $\mathcal{F}_A$, $\mathcal{F}_B$, and $\mathcal{F}_C$, respectively representing increasing awareness of concepts relevant to the assessment, we analyze the influence of such awareness. This assessment is conducted on an equal number of positive (Q^+) and negative (Q^-) queries. For each query, results are collected in terms of a discrete rank and a binary label (see Algorithm 1). Also, since we introduce steps that train PLM variants $\mathcal{F}$ on TAS code, we will henceforth refer to a PLM as `tasnet`. For ease of reference, we also include the variant $\mathcal{F}_A$ under the `tasnet` umbrella, though it is trained only on GitHub code.

Discriminative performance of DECO – The primary tool which we use for analyzing the results of the controlled experiment are the labels l collected for each query. This binary label indicates whether the query has been evaluated by the DECO algorithm to comply with or deviate from the CH pattern. The controlled experiment using positive and negative queries, which are known to comply and deviate from the pattern, allows the collection of results of each of these cases into lists L^+ and L^- respectively. Thus, true positive (TP) assessments are those labels in L^+ that evaluate to `True` and false negatives

(FN) are those that evaluate to `False`. False positive (FP) and true negative (TN) assessments are similarly identifiable from L^-, as shown below.

$$\begin{aligned} TP &: \{l \mid l == True, l \in L^+\} \quad FN : \{l \mid l == False, l \in L^+\} \\ FP &: \{l \mid l == True, l \in L^-\} \quad TN : \{l \mid l == False, l \in L^-\} \end{aligned} \tag{5.11}$$

Using this, we build the confusion matrix (Table 5.1) and performance metrics of the assessment process (Table 5.2). These metrics help us answer the research questions posed in our problem statement.

Table 5.1: Compliance assessment – confusion matrix[1,2]

Queries \ Prediction (l)	$\mathcal{F}_A$		$\mathcal{F}_B$		$\mathcal{F}_C$	
	True	**False**	**True**	**False**	**True**	**False**
True (L^+) - 63	22 (0.35)	41 (0.65)	37 (0.59)	26 (0.41)	50 (0.80)	13 (0.20)
False (L^-) - 63	8 (0.13)	55 (0.87)	7 (0.11)	56 (0.89)	4 (0.06)	59 (0.94)

[1] Confusion matrix on labels L^+ and L^- calculated according to Eq.5.11
[2] For definition of each label $l \in L^+ or L^-$ refer to Algorithm 1

Table 5.2: Compliance assessment – performance metrics

Metric	**with $\mathcal{F}_A$**	**with $\mathcal{F}_B$**	**with $\mathcal{F}_C$**
Accuracy	0.611	0.738	0.860
Recall	0.349	0.587	0.790
Precision	0.733	0.840	0.920
F1 score	0.473	0.691	0.850

Table 5.2 shows encouraging signs DECO is able to successfully discriminate known compliant instances of the CH pattern from known non-compliant ones. Even with the base `tasnet` variant $\mathcal{F}_A$, which is pre-trained purely on non-automotive code, the system is capable of identifying instances of the CH pattern with a precision of more than 0.70. As also seen in Table 5.1, with a high True Negative Rate (TNR) (0.87), the system is particularly adept at correctly identifying non-compliant instances of the pattern. The main concern, seen from the same table, is of course the very high False Negative Rate (FNR) of 0.65. That is, the system built using $\mathcal{F}_A$ is misclassifying a majority of known instances of the CH pattern as non-compliant. The high FNR, in turn, lowers the accuracy, precision and F1 score. Thus, while the performance of design compliance assessment using $\mathcal{F}_A$ is encouraging, it remains unsatisfactory. We reason that there are three main factors that could explain the high FNR. The first is the product set $\mathcal{V}^*$, which considers all possible pairs of programs from applications that are known instances of the CH pattern. The introduction of doubtful pairs could taint both the average jointness benchmark r and whether a positive test input is genuinely so. The second reason could be the lack of familiarity with TAS domain and design in $\mathcal{F}_A$, due to which program embeddings are arranged in such a way that the benchmark vector r does not serve as a good offset. The third reason could be some weakness in assessment using the average

jointness benchmark r. Results from testing with `tasnet` variants $\mathcal{F}_B$ and $\mathcal{F}_C$ show that it is less likely to be due to a weakness in the assessment approach.

Having been pre-trained only using the GitHub corpus, one weakness in $\mathcal{F}_A$ is that it is less aware of domain and design-related specializations in the TAS corpus. This is precisely why we train variants $\mathcal{F}_B$ and $\mathcal{F}_C$ by explicitly providing this information. Assessment using $\mathcal{F}_B$, which learns domain-specific naturalness and the concept of SWCs used in the TAS corpus, leads to a strong reduction of the FNR to 0.41. The consequent improvement in the F1 score to 0.7 is also noteworthy. This clearly indicates that inducing the knowledge of SWCs directly leads to an improvement in the quality of assessment. Using model $\mathcal{F}_C$ – which is trained to understand controller and handler programs – for the assessment leads to yet another strong reduction in the FNR to 0.2, due to which the precision and F1 score commendably increase to 0.92 and 0.85 respectively. The clustering objectives (5.9 and 5.10), are therefore likely to have resulted in an arrangement of embeddings that better satisfies (5.6). These observations clearly indicate that using a PLM with an increased level of awareness about the domain and its design results in a much more accurate assessment. Even with a marked improvement in the quality of assessment, the FNR remains a concern. To analyze this, there is a need to go beyond binary assessment to a finer method.

Calibrating DECO with expert review – Analyzing the binary labels of compliance (L^+ and L^-), using the confusion matrix and metrics derived from it, helps evaluate the performance of the assessment system. While this is necessary to build confidence in the system, from the perspective of an architect or developer, it is equally important to understand *why* the system assesses a query as complying or deviating from the CH pattern. Since this requires much more nuance than a binary label, we turn to the rank k to gain a finer interpretation of the assessment. Specifically, we analyze the distribution of K^+ and K^- of ranks respectively collected for positive and negative queries. For brevity, we confine our analysis to the best performing system that uses `tasnet` variant $\mathcal{F}_C$. First we begin by visualizing the spread of ranks shown in Figure 5.4. Inspecting the spread of ranks for the positive cases K^+, allows us to demarcate three intervals of ranks where results cluster. Next, we sample queries from each interval and have them assessed by architects who are familiar with TAS. The manual assessment of sampled queries follows an approach similar to the one described in Section 5.1. Using expert review we calibrate the results of the PLM-based compliance assessment system within each interval as described below.

• *Interval 1 (ranks 1-100)* – The interval where a majority of positive cases cluster, it consists mostly of queries that are assessed by architects to be good implementations of the CH design pattern. Some are even judged to be textbook cases with the right interface and responsibility split between controller and handler programs. The best ranking instances in this interval are also those which exhibit bidirectional exchange of information between the two programs. The exchange follows standard practice of using the AUTOSAR RTE (refer Section 2.1), seen in their use of `RTE_read` and `RTE_write` methods (refer to roof hatch example in Listings 4.1 and 4.2). Cases that perform relatively worse within this interval (rank close to 100) are observed to implement unidirectional

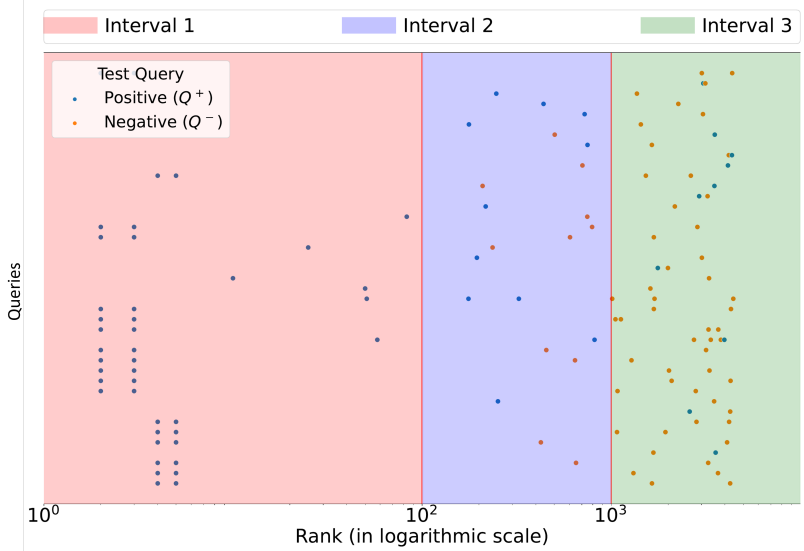


Figure 5.4: Calibrating the results of assessment into interpretable intervals using expert review

interaction, where the controller only reads from the handler, which is perfectly legitimate. Therefore, expert review generally considers test inputs that rank in this interval to be compliant with the CH pattern, with no need for refactoring. This is further strengthened by the fact that not a single negative test case is ranked by the system as being in this interval.

• *Interval 2 (ranks 100-1000)* – Expert review indicates that positive queries in this interval show subtle deviations from the standard implementation of the CH pattern. One deviation is that, while the responsibility split is correct, the controller and handler programs do not interact directly with each other. The actual interaction, in this case, usually happens through some other program in the controller SWC, which is excluded due to the constraint that the system operates only on pairs of programs. This is, therefore, not a genuine violation and results simply due to a limitation in the system. More significantly, the other observed deviation is where there is direct interaction, but it does not take place through the AUTOSAR RTE. This is a subtle deviation which could benefit from refactoring. The fact that the assessment system consistently places such cases in the second interval is an encouraging observation. However, the deviations observed by expert review in this interval also seem to be characteristics observable in pairs of programs that are not controllers or handlers. For instance, it is plausible that random sampling from the relatively small TAS corpus results in a pair of non-interacting programs, one of which contains some application-like code and the other containing some code related to hardware. This could explain why some negative cases end up being ranked in this interval. In general, when the system ranks a query in this interval, it could be a candidate for refactoring. However, it is best if the automated assessment is manually verified to ensure that it is a genuine case.

• *Interval 3 (ranks 1000-5000)* – Very few positive cases rank in this interval. In some cases, the query programs ranked in this interval implement diagnostic routines and not application logic. In others, the controller program is very small, containing only a few lines of code. Generally, therefore positive cases seem to rank in this interval because they are marked outliers compared to the average CH implementation. A cause for concern is the handful of cases which are genuine false negatives and are, in fact, assessed to be good implementations of the CH pattern. Moreover, a query ranked in this interval seems to deviate from the average implementation to such an extent that it is barely distinguishable from random queries drawn from TAS. A result in this interval therefore requires manual review by an expert.

Thus, the greatest advantage of the system is its ability to identify genuine compliance with the CH pattern. Such cases, as verified by experts, rank in the first interval. Also, its tendency to rank subtle variations – possible candidates for refactoring – in the second interval shows its ability make nuanced judgments. Finding such deviations is a strong indicator of its practical utility. The inconclusive nature of results in the third interval, and the presence of some negative cases in the second, indicate the boundaries of this process.

Overall observations – First, queries that fall within the first two intervals are remarkably similar in character, meaning that observations apply quite consistently to cases within a given interval. This reflects the consistency of automated assessment using the average jointness benchmark. Second, this consistency eases practical use because when a query ranks within an interval, we have a reasonably good idea why this happens. This means that any subsequent design intervention can be precisely targeted to rectify suspected deviations. Third, the calibration process makes it possible to decide the conditions under which an architect must intervene. Ranks in the first interval do not require human verification, while those in the second and (especially) third intervals need active intervention. These observations thus point to the ingredients of a protocol for interpreting the results and, thus, practically using the system.

However, we also observe a few caveats in the process which we now list. First, under the current process, the benchmark r needs to be recalculated whenever there is a new instance of the CH pattern. Since this is not a computationally heavy process, we do not rate this as a major concern. An alternative would be to fix 'golden' instances of the pattern so that the benchmark r is itself fixed. While choosing such instances, it is important to ensure that legitimate variations are included. It would also be necessary to periodically audit the golden instances to ensure that they are up-to-date with the latest understanding of the pattern. Second, the DECO ranking process depends upon all programs in the TAS corpus, meaning that the addition of new programs needs a recalibration of the results. In the worst case, the inclusion of a new set of highly specialized programs could severely disrupt the calibration. However, it is important to note that such risks are inherent to any benchmark that is derived from an evolving corpus. Third, there is a need to better understand the relationship between pattern compliance and rank. Consider the test input with a rank close to 100 (and thus in interval 1), but deviates from the textbook implementation

because here the controller only reads from, and does not write to, the handler. Such a deviation seems sufficient for ∼100 programs in the TAS corpus to come in between the predicted and actual handler embeddings. While the empirical calibration process allows us to circumvent this, it is essential to understand the nature of intervening program embeddings. This is an investigation that we prioritize for future work. Overall, results from this study demonstrate a promising method to construct an automated system for measuring design pattern compliance using neural language models trained on source code.

Chapter 6

Discussions

6.1 On research questions

Let us now discuss how the design compliance assessment system built using `tasnet` and DECO measure up to the research questions posed in Section 4.2

RQ1: assessing design compliance using neural PLMs – The discriminative performance of DECO, measured in Section 5.2, strongly indicates that a system for assessing design compliance can indeed be constructed using a neural language model trained on nothing but source code. After all, (variants of) `tasnet`, which is a BERT-like encoder trained progressively on GitHub and TAS code, lies at the heart of the assessment process. The ability of `tasnet` in being able to represent programs in a fairly large domain is the very starting point of the compliance assessment process. Using program embeddings, produced by `tasnet`, the design review activity shifts from the space of code tokens to that of semantic vector representations. Though the translated space of representations is abstract, the DECO algorithm ensures that compliance assessment conducted in this unintelligible space is simple and principled. The application of principled vector arithmetic begins with choosing the difference vector as the joint representation of the pair of query programs. This, as described in Section 5.1 is inspired by the principle of embedding regularity that a set of embeddings related by the same concept tend to have a predictable geometric arrangement. Then, as a benchmark for comparison, DECO uses the average difference vector of a set of program embedding pairs that are known to comply with the design pattern in question. With numerous difficulties in constructing a viable benchmark, the concept of average jointness employed in DECO is simple and durable. As a vector representation of the average or typical implementation of a design pattern, the concept of average jointness also has sound semantics. Practically, this means that DECO simply measures the alignment between the query implementation and an abstract 'mainstream' implementation. Such semantics is straightforward for developers and reviewers to comprehend. Put together, this examination amounts to an assessment that

`tasnet` and DECO constitute a method that is able to viably assess design compliance using neural PLMs and simple vector arithmetic.

RQ2: assessment using PLMs with increased knowledge – Results in Section 5.2 show an incontrovertible trend – as we step through the three `tasnet` variants, with sequentially increased exposure to the TAS domain, the discriminative performance of DECO improves. This amounts to a firm answer that the performance of design compliance assessment does improve with domain familiarity. This sequential approach to increasing domain awareness is, as noted in Section 5.2, based upon the intuition that human engineers themselves go through a similar learning process. Automotive (embedded) software engineers typically go through the cycle of gaining familiarity with general C programming, followed by learning the specificities of automotive or AUTOSAR programming. Increased exposure to AUTOSAR programming inevitably leads to comprehension of the finer elements of AUTOSAR software design. Basing the three-step training of `tasnet` variants on this intuition seems to have yielded desired results. This also shows that when combining `tasnet` with DECO, the relevance of the semantics captured by the program embedding matters greatly. Put otherwise, when stepping through different `tasnet` variants, the DECO procedure remains unchanged. In being able to better capture relevant semantics, the variant $\mathcal{F}_C$ is able to produce better performance. This, of course, does not mean that DECO is a silent participant in the process. Acts like averaging the jointness vectors of the benchmark set, using discrete ranks, and calibration by expert review, can be reasoned as operations that compensate for shortcomings in proper semantic representation. Clearly, it is the combined effect of representational quality and principled vector arithmetic that leads to an assessment of design compliance. In this combination, improving domain awareness in the `tasnet` model has a measurable effect on performance.

RQ3: easing interpretation of assessment – While the embeddings of query programs are clearly abstract, every step that DECO takes in processing them is transparent and explainable. Using the difference vector, as we have already noted, is derived from the observation of embedding regularities. Average jointness, the benchmark for comparison, is also based on an intuitive reasoning that averaging tends to strengthen the signal of the design pattern, simply because this is the main concept that relates program pairs in the benchmark set. Even the comparison between the query and benchmark jointness, essentially a real-valued similarity measure, is converted into a discrete rank for interpretability. Given a pair of query programs (X, Y) a rank of $k = 1$, indicates that no other program comes close to the predicted handler than the actual handler Y. Even a rank of, say, $k = 3$ is useful because it conveys that the predicted handler looks more like the program Z that is third closest to the expected handler. Engineers are free to analyze Z to see if this prediction is of any significance. For instance, Z may share some, but not all, characteristics with Y, and an insight into the shared characteristics could be valuable. This adds another layer of interpretability – the possibility of understanding the rank of a prediction by examining programs that correspond to this rank. Finally, the act of calibrating ranks with expert review further simplifies the interpretability of the overall assessment. Using the intervals drawn by calibration, engineers

are able to gain deeper insights into the prediction, including the possible reason for (non-)compliance. We can, therefore see that interpretability permeates through the entire predictive chain beyond `tasnet`. The use of simple, transparent, and principled predictive steps, as pointed out in Section 1.3, is what sets this approach apart from routine fine-tuning using annotated data.

6.2 On techniques employed

Observing versus utilizing embedding regularities – That a large part of the reasoning behind DECO is drawn from the property of embedding regularity is quite clear. In fact, the entire reasoning behind DECO can be reinterpreted from the perspective of embedding regularity alone. If `tasnet` learns to correctly encode the jointness that underlies the CH design pattern, embeddings of pairs of programs (C_1, H_1) and (C_2, H_2) that implement this pattern should approximate a parallelogram. In which case, $e_{C_2} + (e_{H_1} - e_{C_1}) \approx e_{H_2}$ must hold, which is a special case of the average jointness benchmark with one known pattern instance. If this parallelogram geometry consistently holds across several instances of the pattern, the average jointness vector r naturally serves as an effective offset between the program embeddings of any given instance (X, Y). Further, since regularity is essential for compliance using r as the offset, we reason that clustering objectives (5.9 and 5.10) strengthens it, improving the quality of the assessment process. Additionally, [87] formalized the idea of testing the regularity of one pair of related words using the average offset of other pairs of similarly related words – a technique that they refer to as 3CosAvg. Their use of the average offset closely reflects our construction of the average jointness r as the benchmark for assessment. While the connection with regularity bolsters the credibility of DECO, there is an interesting question to ponder – is the controlled experiment in Section 5.2 an act of confirming embedding regularity, or is it an act of using regularity to make predictions? Clearly, both these aspects are entangled. Meaning, only if regularity in program embeddings is observable, can this regularity be used for assessing compliance. It is fair to reason, therefore, that the controlled experiments that we conduct jointly studies both aspects. As an act of evaluating regularity, these experiments are similar to the seminal study in [84] which investigates regularity of word embeddings. However, unlike typical studies on regularity reported in literature, we repurpose the property for predictive purposes. It is however, important to note our study evaluates regularity only from the perspective of the CH relationship. In order to comprehensively evaluate regularity in program embeddings, other relationships also need to be studied, and this is an interesting avenue for future work.

The quality of program embeddings – The system we construct for assessing compliance with a design pattern is built upon program embeddings, which are vector representations of programs extracted from `tasnet` variants $\mathcal{F}$. The quality of the assessment process is therefore highly dependent upon the quality of the representation. For instance, when discussing RQ3, we noted that exposure to data (code) in the TAS domain plays a pivotal role in improving the quality of representation and, eventually, the performance of assessment.

Apart from data, another important factor that influences this quality, is surely the objective that is used to train the model itself. PLMs used in our study are primarily trained using the masked reconstruction, or cloze, task shown in (5.2). The simplicity of the MR task is undoubtedly its key advantage. However, a major shortcoming of the BERT masking recipe is that, by uniformly choosing 15% of the tokens to be masked, only tokens that are numerically abundant – but semantically less significant (like `;`) – are more likely to be masked. In order to successfully reconstruct a token like `;` it is often sufficient to simply learn concepts in a local scope, like the likelihood of the end of a statement. Thus, with the model rarely being tasked with reconstructing tokens that are semantically significant, it is relatively less equipped to learn global concepts like design. This could explain why the base model $\mathcal{F}_A$, which is pre-trained only using MR, performs worst. This weakness of MR is well-documented in literature and several interesting alternatives have been proposed that encourage the model to learn more global concepts. One option is to modify the masking recipe like [88], which masks selected phrases and [89], which masks larger spans of tokens. Another option is to use [90] and [91], which task a model to detect replaced, permuted, inserted, or deleted tokens. As tasks that are more complex than reconstructing simple tokens, they encourage the model to gain a deeper understanding of program contexts. Another interesting alternative class of training objectives are those that selectively obfuscate tokens. For instance, a de-obfuscation objective proposed by [92] obfuscates class, method, and variable names before tasking the model to recover them. Since the successful completion of this task requires a deeper and broader understanding of the program, they may lead to embeddings that are better suited for a design assessment. While we reason the fine-tuning objectives that improve domain and design-related awareness (5.9 and 5.10) are likely to remain important, setting a task that is more complex than MR may result in a much more powerful base model $\mathcal{F}_A$. We leave this investigation for future work.

Training beyond code – Our results clearly show that it is possible to construct a system for assessing design compliance using PLMs trained on source code. However, we do not necessarily advocate a code-only training approach for imparting design knowledge. In addition to source code, automotive software engineering, which follows the AUTOSAR standard, captures additional engineering information using the standard ARXML modeling language. From the perspective of design awareness, would it therefore be helpful to explicitly train `tasnet` with ARXML models? The answer depends, of course, upon whether such models provide additional design awareness. If most of the information in ARXML models is likely to be replicated in code, then using them for training is unlikely to enhance design understanding. On the other hand, if design models do contain some information not discernible in code, it may indeed be helpful to additionally train with such information. Assessing the usefulness of engineering information in ARXML for design compliance assessment is an investigation that we leave for future work.

6.3 Related work

To the best of our knowledge, our work is the first attempt to apply neural language models for measuring design compliance. In software engineering, our work closely relates to the task of design pattern detection. A recent survey of this area [93] reveals that around 20% of reported methods take a machine learning approach, mostly using classical algorithms. Examples include [94] which compares pattern instances by modeling them as graphs, and [95] and [96] which use artificial neural network and random forest models respectively to classify pattern instances. We reason that the key advantage of our use of neural language models is the level of nuance that it can apply to judging design. A BERT-like PLM, which has been shown to learn nuanced contextual information, could be vital for assessing design, where firm judgments are rare. Also, unlike the majority focus on pattern detection, we develop a technique for measuring compliance with a given pattern, including steps to identify the source of deviation. Moreover, our study focusing upon embedded control systems would also be a useful addition to an area that mostly focuses on object-oriented design.

As discussed in detail in Section 6.2, our approach to compliance assessment closely relates to the property of linguistic regularity observed in neural natural language models [84]. Most experiments, as surveyed in [97], study this property as a way to evaluate the quality of word embedding models. Few of them apply this property in a predictive setting by framing an analogy completion task where, given a triplet (A, B, C), they predict D such that (A, B) and (C, D) are analogical pairs. Studies [98] and [99] approach this task respectively using popular word2vec and GloVe embedding models, while [100] uses sense embeddings derived from word2vec. An example of the property being studied in a specialist domain is [101] which fine-tunes GloVe on a corpus related to radiology and uses its embeddings for the analogy completion task. Similar to our departure from word embedding models, [102] studies this property in pre-trained contextual neural language models. The work we survey can therefore be seen to relate to parts of our assessment system, but we build a pipeline that not only analyzes embedding regularity but also interprets it within the context of software and its design. In doing so, we also tie the property of embedding regularities to a concrete application.

6.4 Congruence with research objectives

In developing a system that automatically assesses design compliance, innovations introduced in this chapter take significant strides in addressing the first research objective (see Section 1.2) of easing the process of software design compliance. First, the design pattern that we choose to work with – Controller-Handler – is the linchpin of the TAS corpus in terms of design. This means that `tasnet` combined with DECO, which assess compliance with the CH pattern, addresss the most important software design practice used in engineering the flagship product of a major commercial vehicle manufacturer. Second, the

compliance assessment system is defined in such a manner that the pattern $\mathcal{D}$ itself is configurable. In fact, the choice of the design pattern for compliance assessment is only made when assembling the benchmark set V. Meaning, the system $\mathcal{S}$ becomes a mechanism to assess compliance with the CH design pattern simply because we populate V with CH pairs. If the assessment process needs to address some other design pattern, one may just need to revise the benchmark set with examples from the new pattern. This means that as long as we are dealing with AUTOSAR software components, which is a significant proportion of vehicle application software, our system should be extendable to similar design compliance tasks. Finally, we also reason that the mechanism is extendable beyond TAS into other automotive, or even non-automotive, domains. As long as a representative code corpus exists, the two-step adaptation process which we describe in (5.9) and (5.10), with some modifications perhaps, can be applied to the new domain. After which, an appropriate benchmark set of programs can be chosen to automate a design review task in the new domain.

6.5 Congruence with the solution approach

On the big data, deep learning approach – Analyzing how innovations introduced in design compliance assessment measure up to the solution approach that we charted in Section 1.3, let us begin with Part A. There, we state that we take a big data – in this case big code – approach to solve practical tasks in automotive software design. If we look at the TAS corpus in isolation, it is easy to see that the ∼5k files of source code that it contains certainly do not amount to big code. At first glance, this is of genuine concern because in order to teach the nuances of vehicle application software to a model, there needs to be a sufficient mass of examples. However, the lack of mass in one domain can be compensated by genuinely big code in a closely allied domain. Noting that vehicle application software is, after all, C code, we turn to the GitHub public dataset where there are millions of examples in C programming. Put together, as our results show, GitHub and TAS corpora constitute enough mass to induce sufficient knowledge in `tasnet` to solve design compliance assessment. It is important to emphasize this point because, while automotive big data is a reality, the phenomenon is not uniformly observable in all its disciplines. In areas that lack big data, it may be a viable option to borrow from neighboring domains where data is abundant. Further, the crux of a task like design review lies in interpreting expressions in code – its naturalness – and weighing its suitability against required practice. Considering the richness with which ideas are expressed in code, one can only automate design review if a machine model of code awareness which is comparably rich to that seen in human engineers is constructed. Here, in agreement with many other examples reported in literature, deep learning proves its worth by inducing such awareness in a fairly complex domain using big code.

On the use of foundation models – Once the big code corpus is assembled, we take the fairly well-trodden path of using the cloze or masked reconstruction task to pre-train `tasnet` variants. In that sense, the base variant $\mathcal{F}_A$ that we pre-train with the GitHub corpus bears the conventional image of the

foundation model that we refer to in Part B of the solution approach. Training steps (5.9) and (5.10) that progressively induce domain specific knowledge, however, represent a subtle shift in training strategy. Encouraging the model to cluster programs in its embedding space, by considering their design properties, is an innovation that turns out to be effective. As we reasoned in Section 6.2, such an arrangement of embeddings is likely to have assisted the design review task considerably. Even if we shift the training strategy with the use of clustering objectives, the key factor to note is that the training still remains self-supervised. This way, the domain-adapted `tasnet` variants $\mathcal{F}_B$ and $\mathcal{F}_C$ still satisfy the definition for foundation models specified in [28].

On pre-train and calculate – One of the core tenets of the approach that we adopt here is that foundation models can be used for solving tasks in automotive software engineering without the use of a supervised training step. If anything, discussions in this chapter show that the stance of avoiding explicit task supervision is no mere pretext and reflects the necessities in solving software engineering tasks. A task like assessing the compliance of vehicle application software with the CH design pattern is not well served by a binary answer. There are far too many ways in which implementation can deviate from ideal design, and not all of them are unjustified. In fact, as we discuss with the calibration exercise in Section 5.2, the ability of the assessment to reveal a variety of deviations is precisely what is needed. Rank intervals that correlate with possible deviations, is a far more nuanced analytical tool than binary labels. The act of calibration, as discussed earlier, does incur costs, but it is arguably less expensive than forcing architects into a labeling exercise that classifies compliance into a limited set of categories. Even more importantly, the DECO compliance assessment procedure that we propose in Algorithm 1 is based on simple, principled, rule-based vector arithmetic. Not only does this literal alignment with the 'pre-train and calculate' approach make the compliance assessment transparently interpretable, but it is also the result of a chain of reasoning (Section 5.1) that is sound and can be subjected to scrutiny. None of these properties are readily applicable to a task-specific head that is trained using explicit supervision. Thus, in avoiding supervised task specialization we simultaneously avoid steep costs of annotation and enrich the nuance of prediction, both of which work to the advantage of engineering necessities.

On evaluating the quality of design compliance assessment – Evaluating the quality of automatic design compliance assessment using the `tasnet` language model and the DECO algorithm is critical for successful application with trust and confidence. Before briefly re-examining the evaluation strategy that we undertake, we first establish some useful context. Under the current state of practice in the automotive industry, as traced in Chapter 3, the assessment of design compliance is largely manual. Design review of embedded application software is usually reserved for experts, and it may take several years for engineers to develop such expertise. Besides, under current practice, software development teams are distributed, making it difficult to coordinate the review process, demanding even more time from experts. There is, therefore a clear need for capable, standardized methods for conducting automatic design compliance assessment, and the `tasnet` – DECO combination caters to this

need. As described in Chapter 5, we take a two-level approach to evaluate the quality of automatic assessment, a recap of which is presented below.

1. As a first level, the TAS corpus and its labeled instances of CH pairs provides a ready benchmark for testing the accuracy of DECO.
2. Using true instances of compliance as positive cases and randomly paired instances as negative cases, we evaluate the binary discriminative performance of DECO using a standard confusion matrix.
3. Compliance with the CH pattern is, however, not a strictly binary affair, which is why the second level of evaluation examines nuance in assessment.
4. Using expert review, we examine the rank – a raw measurement of geometrical alignment – and identify clusters where rank values denote specific nuances in (non-)compliance recognized by experts.
5. Such an expert review process simultaneously verifies the capabilities of automatic assessment and calibrates it for better interpretability.

Evaluating the quality of assessment both at the levels of accuracy and nuance increases the confidence with which this method can be applied to automate design compliance. However, as previously noted, there remain limitations which need to be addressed. Though the DECO algorithm is designed to be extendable to other design patterns, this capability needs to be studied further. Particularly challenging could be cases where design patterns apply to more than two source modules, which may require an extension beyond the parallelogram geometry. Another aspect to address would be the evolving interpretations of design patterns. It is quite common for automotive software to be developed, or at least maintained, across several years. This increases the likelihood that definitions of design patterns evolve. The calibration of benchmarks across different versions of interpretations is another limitation which needs to be addressed. Despite such limitations, the neural language model coupled with the rule based mechanism that we develop introduces a useful and much-needed alternative to manual design review.

In summary, chapters in Part I demonstrates how neural language models trained on source code can be used to measure whether a set of programs comply with desired design properties. Compliance is measured by inspecting the geometrical properties – specifically the regularity – of query program embeddings. Our work also includes techniques that significantly improve the accuracy of the assessment by explicitly providing the model with domain and design-related information. We also present how the model predictions can be incorporated into a design review methodology in order to provide valuable feedback to automotive software architects. Overall, methods introduced in this chapter have the potential to significantly reduce the time taken for design review and therefore help achieve the overall objective of increasing the cadence of automotive software engineering without compromising quality.

Part II

Easing the process of virtual software testing

Chapter 7

Virtual automotive software testing

Having discussed the process, challenges, and the neural language model-based tool that we develop to ease the design of automotive application software, let us turn our attention to the second area of focus in this work – testing this software. In helping decide the content, structure, and composition of SWCs, the design process reasons about the application mainly from an internal perspective. Testing, on the other hand, has the responsibility of ensuring that the application indeed realizes all the required functionality, no matter the specific form of its design or implementation. Thus, testing can be seen as taking more of an external perspective, reasoning about the properties of an application by looking at it from the outside. Therefore, in order to trace the journey of the testing activity, a helpful starting point would be to begin with the external interface of an AUTOSAR application. After all, it is only through this interface that one can interact with, and also test, the application.

A good first glimpse of the external interface emerges upon examining Definition 3 of the AUTOSAR application. Since AUTOSAR views the application $\bar{\mathbf{y}} = \mathbf{W}(\bar{\mathbf{x}})$ as a function, the vector of inputs $\bar{\mathbf{x}}$ and outputs $\bar{\mathbf{y}}$ form its interface. The interface is denoted as vectors simply because practical AUTOSAR applications have several inputs and outputs. The AUTOSAR term for each element in the input or output of a SWC is *signal*. Since the application is inherently a composition of one or more SWCs, the interface of input and output signals applies recursively to every SWC in the composition. The importance of signals in the application software system can be highlighted using the example in Figure 7.1, which illustrates parts of a braking application.

In this example, a `BrakePedalHandler` SWC samples the brake pedal and sends its state as a signal `brake_pedal_position` to the `BrakeBlendingController` SWC. In order to distribute the driver's brake request among different brake actuators, the blending SWC computes the brake pressure to be applied at various retarders in the pneumatic circuit. Assuming that the brake system is

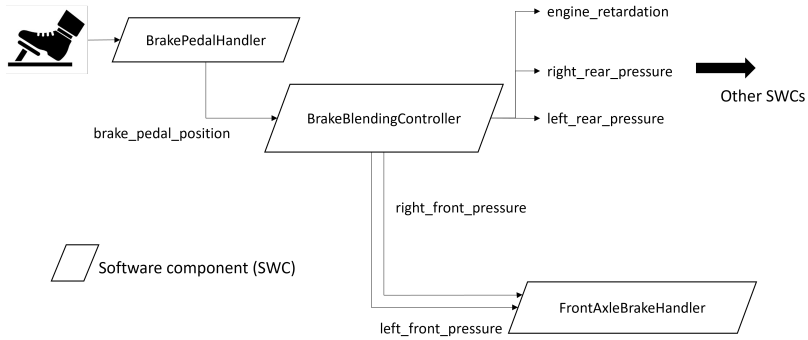


Figure 7.1: Example AUTOSAR application with signals being exchanged between software components

designed with one actuator SWC per axle, then the `right_front_pressure` and `left_front_pressure` signals are routed to a `FrontAxleBrakeHandler` SWC. Having been so commanded, using signals, to apply specific amounts of pressure, `FrontAxleBrakeHandler` operates the necessary actuators to fulfill the command. This example helps illustrate that signals constitute the connective tissue in helping SWCs collaborate to realize a larger function. As a clear element of significance in the E/E system, it is helpful to define the signal as follows.

Definition 4 *A signal* $(\mathbf{s},\ \mathbf{x}), \mathbf{x} \in \mathbb{R}$ *is the instantaneous snapshot of one aspect of vehicle state* $\mathbf{s} \in S$*, with a set of possible states* S*. Signal values are usually observable at the input or output of an AUTOSAR SWC.*

A signal is a key-value pair of a state label and value, observed at a given time. For example, `brake_pedal_position` : 20% *at* $07:42:17$ *UTC* is a signal. It captures the instantaneous state of the brake pedal, presumably by observing the output port of the `BrakePedalHandler` SWC. Often, especially if the name of a signal is understood, a signal is simply represented by its value $\mathbf{x}$. With signals at its input and output, the SWC can be essentially seen as a unit that transforms one set of signals into another. Thus, while signals embody vehicle state in the E/E system, SWCs embody behavior by actively manipulating state.

Signals as the gateway for simulation – The delineation of an external interface of signals underpins a fundamental precept of virtually testing a SWC. No matter their physical or statistical complexity, signals are the primary means through which a SWC experiences the behavior of its dependencies. From the perspective of a SWC, therefore, plausibly replicating signals in its interface is largely synonymous with simulating its dependencies. Now, upon identifying the external interface of a SWC, the act of testing can be boiled down to simply simulating or crafting input signals that probe a particular aspect of behavior, and verifying it by asserting expectations on the output signals. After all, exploiting the AUTOSAR application layer abstraction to treat the SWC in relative isolation does serve well in simplifying the design and implementation of vehicle application software. Can we not extend such isolationism to testing

too? Not always, because, if we widen the focus beyond a single SWC and its signal interface, as shown in Figure 7.2, a complex network of dependencies emerges. Given any SWC, it is likely that one part of its input consists of signals routed from a set of upstream SWCs, while another part consists of sensors used to monitor some local phenomenon. Correspondingly, some of its output signals are intended to inform or command downstream SWCs, while others may control an actuator. If the SWC orchestrates closed loop automatic control, there is the added element of feedback, where additional sensors are used to measure the consequences of a control action and is routed into the set of input signals. Thus, if a tester were to craft realistic input signals to test the SWC, the crafted signals need to plausibly imitate the behavior of upstream dependencies.

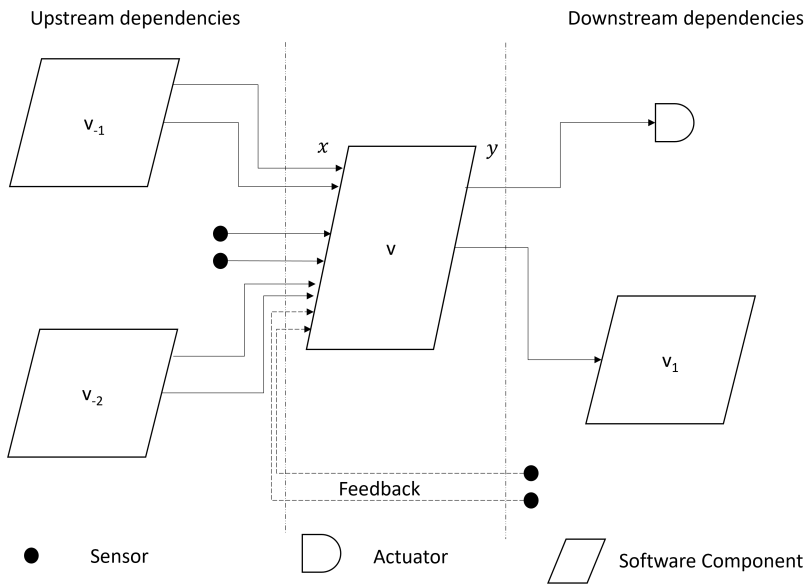


Figure 7.2: A SWC and its dependencies. SWCs v_{-n} and v_n refer to upstream and downstream dependencies respectively

7.1 Simulation for testing vehicle application software

The distinct realities of testing vehicle application software, in contrast to designing it, can be examined by returning to the V model and its entangled process and decomposition sequences. Beginning with the perspective of decomposition, in the well-abstracted space for application software defined by AUTOSAR, let us consider the merits of testing one SWC in isolation. If we turn to the example in Figure 7.1, it is straightforward to note that there are indeed advantages in testing the behavior of the `BrakeBlendingController` SWC as a standalone unit. Any algorithm for distributing a brake request to several retarders is bound to be complex, and unit testing at the SWC level surely aids its development.

The downside, however, is that testing `BrakeBlendingController` alone, without including closely dependent SWCs like the upstream `BrakePedalHandler` and the downstream `FrontAxleBrakeHandler` may fail to reveal errors in the larger braking application. In the larger vehicle system, moreover, we know that software alone is not sufficient to realize the complete braking application. In order to test the complete application, we need to proceed further upward the V to also integrate non-software elements like the pneumatic circuit, the brake actuator, and eventually all associated dependencies in the vehicle. Only by testing the braking application with all its dependencies, software or not, can one be truly confident about its quality. This introduces a dilemma because having specified and implemented a vehicle application, if we can only verify it by integrating all the way up, we end up in a true waterfall V process. This may deliver functionality with quality, but the time and cost involved is prohibitive.

Since the low cadence of a waterfall V is clearly unaffordable, the automotive industry has been steadily trending towards incremental software development, enabled by continuous integration and verification [16]. It is important to ensure that the testing activity is not confined purely to software, but it is equally important to include just the right mix of dependencies to ensure that testing is reasonably independent, quick, controllable, and repeatable. If we take the braking example, one way to set up a test rig would be to integrate all its SWCs and ECUs, restricting dependencies to mainly the electrical and electronic realms. This enables rapid updates to be made in, say, the brake blending algorithm with reasonable confidence. Whether to include the pneumatic circuit and the air pumps into the rig is, however, not straightforward to answer. Integrating them may improve the credibility and confidence of testing with this rig, but the electromechanical and hydraulic hardware reduce the speed of executing and debugging tests. In order to set up a viable test rig, it is therefore important to carefully select the *integration levels* of its constituent components.

Options for integrating test rigs – An AUTOSAR software application can be considered to be highly integrated if it is in a form that is closest to its final deployment. Put simply, an application is at the highest level of integration if its SWCs are deployed on their intended ECUs, which are in turn installed in a real vehicle with all dependencies, and is being operated on a live mission. While the vehicle may be the ultimate 'rig' for software testing, its time and cost-intensive nature means that development is only possible at a very low cadence. Faster loops through the engineering process require rig options where every driven mile is not a real mile and every millisecond of software execution is far less than a real millisecond. It also needs capabilities for systematically applying test scenarios, including rare or dangerous conditions. Excluding avoidable dependencies, as we have seen before, may be one way to achieve it, but this could lower the scope and confidence in the testing process. An exciting alternative that the industry is increasingly turning to is simulating necessary dependencies [16]. This allows preserving the scope of testing while making it faster and flexible by replacing real dependencies with virtual counterparts. Simulation is, therefore, a crucial element in lowering the integration level in a test rig, because any reduction in the level of integration can potentially be compensated by simulation.

Let us now examine how simulation plays an important role in choosing the integration level of a test rig. Generally, it is useful to reason about integration levels, as seen in Figure 7.3, along two dimensions - (i) vertical and (ii) horizontal.

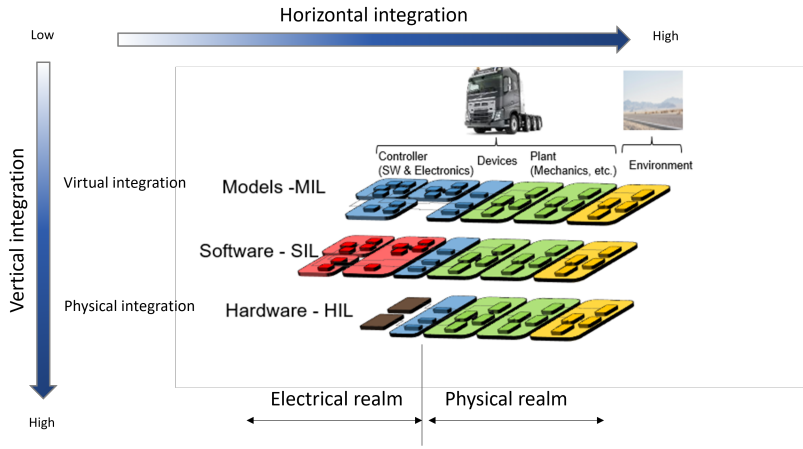


Figure 7.3: Test rigs for vehicle functions and their levels of integration

Vertical integration deals with the form of the application and its execution environment. When it comes to setting up a test rig for a vehicle function, there are three recognized levels of vertical integration [103], arranged lowest to highest (i) Model-in-the-loop (MIL), where a behavioral model of the function is used, (ii) Software-in-the-loop (SIL), where the programmed version (i.e. AUTOSAR SWCs) of the function is used, and (iii) Hardware-in-the-loop, which uses the compiled version of the program deployed on the target ECU. During incremental development, feature content or maturity can be arbitrary at all these levels, and it is the form alone that decides the integration level. Integrating at lower levels, i.e. MIL and SIL, is also referred to as *virtual* integration [104], since in these cases the behavior and the execution environment are respectively simulated. A SIL rig, for instance, can provide an environment for running SWCs directly on a developer machine. It can do this by replacing the embedded microcontroller with a server-grade processor, and by executing the SWCs on an AUTOSAR RTE that is ported to, say, Windows. By thus simulating embedded OS and hardware, dependent SWCs can be integrated purely at a software level. An immediate benefit of such virtual integration is that software development is relatively independent of hardware development, which usually proceeds at a slower cadence. A bigger advantage is that SWCs in the simulated environment can be executed at a faster rate than the real execution, enabling quicker verification of a wide range of scenarios.

Horizontal integration refers to the extent of input and output dependencies of the application included in the rig. In elucidating horizontal integration, it helps to briefly map the range of physical phenomena with which application software interacts. A software application resides and executes in an embedded environment composed of AUTOSAR and an ECU. Its electronic realm can be thought of extending up to silicon or circuit-board levels. Residing in this

electronic bubble, the function reaches out through the electrical realm of wire harnesses to connect, in one part, with other SWCs residing in their bubbles. In another part, they reach out to sensors and actuators, in which case the electrical realm extends up to the sensing or actuating element. Beyond this boundary lies the physical realm, comprising phenomena that the function interacts with. Parts of this physical realm, like the pistons of the engine or the air bellows of the suspension, lies within the purview of the vehicle system. Other parts, like the road surface and maybe even the driver, lie outside. While boundaries are fluid in practice, there are four broad levels of horizontal integration, (1) devices, where the scope extends only to the electrical realm, (2) plant, where the scope extends to include in-vehicle physics, and (3) environment where scope extends to include physical dependencies external to the vehicle system, and (4) driver where the scope also includes human interactions with the vehicle system.

For software development and testing at high cadence, there is a clear necessity for simulating dependencies even in the horizontal dimension. Horizontally, the direst need for virtual integration comes from the need to simulate environments in the physical realm. As we saw earlier, the development of brake blending depends (among others) upon that of the pneumatic circuit, and including it in a simulated form can help increase the credibility of the rig. Unlike simulating the embedded execution environment, the multi-physics character of dependencies like brake pneumatics or the road surface presents unique challenges for simulation. The traditional approach to simulating such multi-physics dependencies is to model their behavior using a suitably capable formalism or framework, with Simulink [105] or Modelica [106] being common examples. When the extent of included horizontal dependencies is wide, manually specifying simulation models for a wide range of phenomena becomes a formidable challenge.

7.2 Limits of traditional simulation methods

Here, we note that this work focuses exclusively on easing the testing process by simulating horizontal dependencies from the plant all the way to the environment. Let us therefore set aside vertical integration, which deals more with simulating the embedded environment, and place all attention on the horizontal dimension. The extent of horizontal integration, especially the level to which it extends beyond the electrical realm, is only one factor that decides the complexity of simulation. An equally important factor would be fidelity, which is the level of detail with which the model faithfully represents the simulated phenomena. Consider a rig for the brake blending function, where the mass of a truck is simulated. If the simulation models ego vehicle mass and that of the goods that it hauls, which can range from liquid nitrogen to timber, it is of wide scope. However, if it models all inertia as a point mass, it is of low fidelity and is only a naive representation of reality. Such a model may allow the brake blending function to be developed at high cadence, but may not be able to credibly evaluate properties like braking distance. On the other hand, if the model accurately captures the spatial mass distribution of important constituent parts, it is of higher fidelity and much more realistic. Such realistic simulation in-

creases the credibility of evaluating properties, meaning that the brake blending function can be matured to relatively high levels at high cadence using this setup. Highly credible virtual rigs in turn help vehicle manufacturers increase the amount of rig-based development and reduce the amount of field-testing, helping them meet evolving market demands without compromising quality.

The price to pay in setting up a credible virtual rig is, however, the engineering effort spent in developing credible simulation models. Manual high-fidelity modeling requires expertise in multiple domains of physics and requires significant time and effort to develop and verify. More importantly, no matter how high the fidelity, manual modeling will inevitably make assumptions. Factors like wear in the suspension, lubrication in the engine, and braking patterns of drivers on country roads, are challenging to realistically model as equations. Put otherwise (Figure 7.4), as one seeks to increase scope and fidelity to lend more credibility to simulation, there is a limit beyond which it becomes practically difficult to manually specify a simulation model. Beyond this limit, it may be cheaper and more effective to use a data-driven approach and train a simulation model.

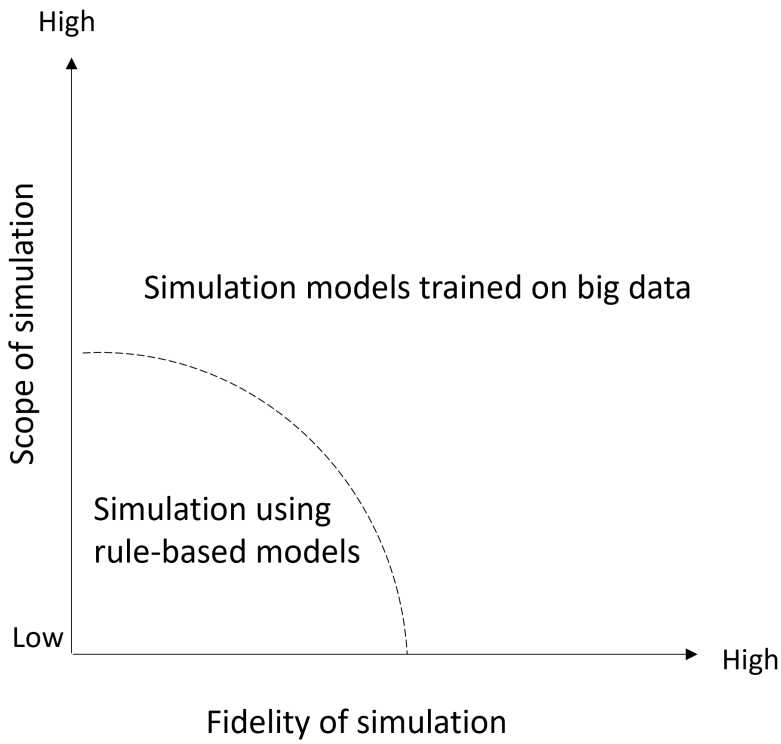


Figure 7.4: When high scope and fidelity are required, training a simulation model may be easier than manually specifying it

7.3 A deep learning approach to stimulus generation

An AUTOSAR application $\bar{\mathbf{y}} = \mathbf{W}(\bar{\mathbf{x}})$ may have many input and output dependencies, but most interactions between the application and its dependencies are observable in the input and output signals $\bar{\mathbf{x}}$ and $\bar{\mathbf{y}}$. Simply tapping into the signal traffic and recording these observations raises interesting possibilities for simulation. After all, given an application, if all traffic on its external interface is recorded under a variety of scenarios, we arrive at a detailed representation of its interaction with all dependencies. Also, as we noted early on, the automotive industry is firmly in a big data existence, meaning that recording such information is fairly routine. Surely, all this big data can be repurposed for simulation.

The time and effort involved is only one aspect of the challenges involved in handcrafting simulation models for test rigs. Another crucial aspect is that manual multi-physics simulation modeling – in a manner similar to manual design compliance assessment – builds upon tremendous engineering experience, which is a valuable commodity. Signal traffic, which records how different systems in the vehicle interact with each other, and with the environment, constitutes a repository of knowledge that can powerfully complement years of engineering experience. For instance, take $\bar{\mathbf{x}}$, the vector of input signals supplied to the application $\mathbf{W}$. This particular instance of the input can be seen as one sample from the distribution of all possible inputs $\bar{\mathbf{x}} \sim G(\bar{\mathbf{x}})$. The underlying process G, may it be based on physics, statistics, or any other science, is what engineers spend years of effort developing and, in order to help virtual testing, also modeling. Given a sufficiently large variety of signal vectors recorded from field operations, can we not use big data to learn the original behavioral process G? Having learned such a model, can we not use it to generate signal traffic that can emulate the dependency G? The field of deep generative modeling [107] focuses precisely on this kind of problem. And, since techniques from this field are important to the innovations introduced in this work, let us briefly examine it.

Deep generative models for simulation – A deep generative model is a DNN that is trained on big data so that it can generate new data with the same statistics as the dataset that it has been trained on [108]. When trained well, this DNN can credibly simulate the underlying process which created this big data. One way to understand the potential of generative models would be to take the example of the Generative Adversarial Network (GAN) framework [109], an early pioneer in this field. In its simplest form, the GAN framework pits two neural networks against each other in a zero-sum game (Figure 7.5). Given, a training dataset $\mathcal{X} = \{X_1, X_2, ...\}, X_i \sim G(X)$, the objective of the game is to generate fake samples $\hat{X}_i \sim G(x)$ which could have been plausibly drawn from the distribution G of real data. The game begins with one network – the *generator* ($\mathcal{G}$) – mapping a multidimensional random code $z_i \sim \mathcal{N}(0, I)$ into a fake sample $\hat{X}_i$. The other network, called the *discriminator* ($\mathcal{D}$) assesses the generator's output, evaluating the 'realness' of the generated samples by comparing its statistics with that of real samples in the training dataset. The classic adversarial training process tries to achieve an equilibrium between these competing objectives

by minimizing the minimax loss (7.1). This minimization seeks an equilibrium where an optimal $\mathcal{G}$ produces fake samples that are realistic enough to fool $\mathcal{D}$, and an optimal $\mathcal{D}$ that clearly distinguishes between real and generated samples.

$$\mathcal{L}_{gan} = \min_{\mathcal{G}} \max_{\mathcal{D}} \; \mathbb{E}_{X_i} log(\mathcal{D}(X_i)) + \mathbb{E}_{z_i} log(1 - \mathcal{D}(\mathcal{G}(z_i))), \quad X_i \in \mathcal{X}, \; z_i \sim \mathcal{N}(0, I) \tag{7.1}$$

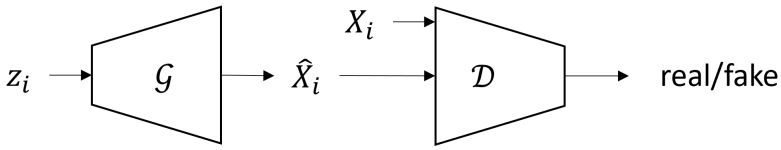


Figure 7.5: The Generative Adversarial Network (GAN)

GANs have been famously applied to the problem of generating images, particularly on generating images of human faces. That is, if a dataset like Faces [58] is used to train a GAN according to (7.1), then the generator network will eventually produce fake face images. Not only are these fake images unlikely to be found in the training dataset but also, when trained well, should be real enough to fool human observers. More importantly, the fact that the generator $\mathcal{G}$ produces realistic face images means that it has implicitly learned to estimate the true distribution of face structure as represented in the dataset $\mathcal{X}$. The basic GAN setup is ideal for unconditionally generating a mass of fake samples by randomly sampling latent codes z_i and mapping them into fake samples $\hat{X}_i$. Refined architectures influence the space of latent codes so that images with specific properties can be conditionally generated. A good example would be the StyleGAN [110] series of face image generation models, where properties like hair color, skin tone, facial expressions, etc., can be finely manipulated.

If we substitute faces with signals in the scenario discussed above, it is clear that there are tremendous opportunities for simulation. Techniques developed in this work, described in detail in following chapters, leverage precisely these abilities and demonstrate how generative models trained on recorded signals can credibly simulate the dependencies of vehicle application software. Specifically, we train GANs on signal data so that they produce fake, yet plausibly realistic, signal traffic. By imitating dependencies of SWCs using GANs, techniques we introduce significantly increase the credibility of virtual testing, bringing much needed speed and confidence to the process of developing vehicle application software. The increase in cadence, without compromising quality, directly addresses the essential requirements for delivering the next wave of automotive software.

Chapter 8

Defining a system for test stimulus generation

'If software is being tested in a running vehicle, it may already be too late'. A quip like this is equally apt either as a quote on a coffee mug or as the headline of a handbook, on engineers' desks. Of course, some form of vehicle testing is necessary when developing vehicle application software, but the essence of the statement is hard to dispute. Perhaps no software is extensively testable in its fully integrated form, but the costs of doing so is especially high in the automotive industry. In introducing this work in Chapter 1, we specifically noted the variety of applications, scenarios, and geographies in which commercial vehicles operate. Under such circumstances, relying extensively on vehicle testing is not practical because the material, financial, human, and maybe even environmental costs of test-driving hundreds of thousands of kilometers can be astronomical [111].

In the preceding chapter, we traced how the industry does recognize the futility of extensive reliance on drive testing and has been actively promoting simulation-based testing. Simulation with software-in-the-loop may allow millions of kilometers of test-driving in a largely virtual rig, but the credibility of such a testing process relies heavily on the quality of simulation. For all the costs that vehicle testing presents, it does retain considerable allure. Even if vehicle application software has been extensively tested under simulated conditions, automotive software engineers, not to mention product owners and managers, may not be fully confident in their delivery until they witness its operation during drive tests [15]. It is therefore necessary for any testing strategy to find a balance between vehicle-centric and simulation-centric approaches that combines each of their strengths [15,112,113]. The emergence of big data, as we reasoned in the previous chapter, opens up exciting possibilities for elevating the role of simulation. In this chapter, we begin exploring the use of deep generative models, trained on recorded signals, as a simulation tool for testing.

8.1 A wealth of operational scenarios in signals

In Section 7.3, we noted that the external signal interface of an AUTOSAR SWC $\bar{\mathbf{y}} = \mathbf{V}(\bar{\mathbf{x}})$ is useful for testing in at least two ways. First, the external interface is the most natural mechanism for probing and examining SWC behavior. Second, observing signal traffic on this interface, under real operating conditions, reciprocally provides useful insights into SWC behavior. In order to better understand the latter perspective, it is useful to define a new data structure, the signal *trace*.

Definition 5 *A signal trace* $\mathbf{X}_{\mathbf{T}}^{\mathbf{N}} \in \mathbb{R}^{\mathbf{N}\times\mathbf{T}}$ *represents the transition of* $\mathbf{N}$ *signals over a duration of* $\mathbf{T}$ *time steps, where* $\mathbf{N}, \mathbf{T} \in \mathbb{Z}^{+}$*. Implicit in this definition is that all* $\mathbf{N}$ *signals are sampled at some fixed rate which, in our case, is 1 Hz.*

A signal trace that includes the input and output signals of a SWC represents one instance of its *observable behavior*, including elements of feedback, during a certain time window. For instance, the trace in Figure 8.1 captures one observation of brake blending behavior. It shows how, upon pressing the brake pedal, a brake blending algorithm distributes the braking torque between the engine retarder and pneumatic service brakes. Thus, for credible simulation-based testing, especially for low-level control software like brake blending, test setups need to generate signal transitions like the example in the figure at a deep level of detail.

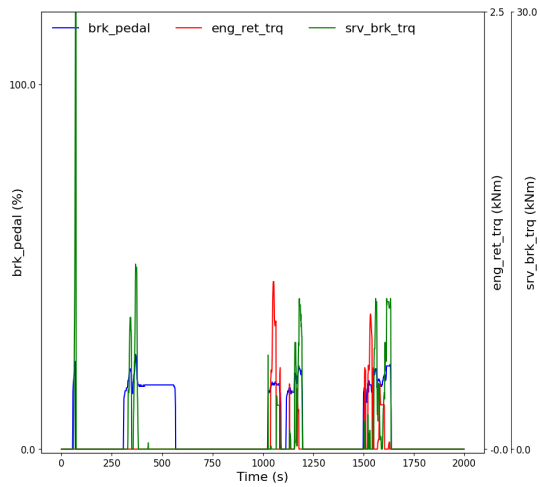


Figure 8.1: A trace of 3 signals capturing brake blending behavior

Today's SIL test rigs simulate these transitions by manually specifying explicit, domain specific, mathematical rules in the form of plant models. Worse, there are cases where signal traces are not really produced by models and are hand-crafted. Either as simulation models or signals themselves, hand-crafting may work well when testing individual SWCs, where plant models need to simulate transitions of only a handful of signals. As the scope expands to testing larger sub-systems of SWCs, under the influence of realistic driving, manual modeling

takes significant effort. This is because, during a driving maneuver that lasts several minutes, the continuous, multidimensional vehicle state-space undergoes complex transitions, many of which are difficult to model.

Owing primarily to the time-critical nature of several vehicle functions, most of them are executed periodically at a fixed rate of, say, $10Hz$. This, in turn, means that the flow of signals between various parts of the system is also periodic. Under such conditions, it is possible to tap into the signal traffic to capture N signals in this system as a trace over a duration of T steps. Such a trace provides remarkably detailed information about the operational behavior of this system and has become the de facto structure for recording vehicle operation data. If many such signal traces are recorded from vehicles operating under a variety of scenarios, a new big signal data regime emerges. Earlier, we saw that deep generative models like GANs, when trained on a sufficiently large dataset, implicitly learn to model the underlying process that produced the training data and can be used for finely controlled generation of realistic fakes. This observation, as we soon show, is extendable to the domain of vehicle signals. GANs trained on large datasets of signal traces learn to model the behavior of vehicle systems. Trained GANs can then be sampled for the controlled generation of fake, yet plausibly realistic, instances of vehicle behavior in the form of signal traces, which can serve as valuable stimuli for testing vehicle application software.

8.2 Stating the problem of stimulus generation

In order to begin the process of envisioning a system $\mathcal{S}$ for stimulus generation, let us revisit a SWC V and its dependencies, which we saw earlier in Figure 7.2, and rearrange it into a slightly different form (Figure 8.2). As a scoping measure, we set aside output dependencies of the SWC, and focus solely on its input. In addition, we also set aside elements of feedback, routed from the output to the input. While the new scope does exclude a notable proportion of SWCs, the level of exclusion is not drastic. Simply because not all SWCs, even those involved in control functions, implement closed-loop feedback. If we refer back to the braking system example in Figure 7.1, we see that the separation of concerns is hierarchical. From the perspective of `BrakeBlendingController`, which issues the command, it is the job of the brake handler to accomplish required actuation. This means that only the `FrontAxleBrakeHandler` SWC, which adjusts pneumatic pressure, is likely to implement closed-loop control. Moreover, as we have seen in Part I, a hierarchical or layered approach to control is essential from a software design perspective. Thus, even with refinement, `BrakeBlendingController` falls within the scope, and testing it with generated traces of brake pedaling at its input, is of enormous value. However, elements of feedback are an important aspect of vehicle behavior, and must eventually be incorporated into the stimulus generation process. Addressing this gap is therefore a crucial avenue for future work. As a final act of refinement, we subsume all input dependencies into the super-dependency g, which now channels all input signals into a trace $\mathbf{X}_{\mathbf{T}}^{\mathbf{N}}$ that is applied to the SWC V. Since the number N and duration T of traces are usually well-understood

in the context, we simplify the notation of a trace to X. An individual trace can be considered as one instance $X \sim G(X)$ drawn from the distribution of possible traces that is output by the super dependency g. The primary aim of the stimulus generation system $\mathcal{S}$, as shown in Figure 8.2, is to plausibly substitute the input dependency g and generate traces $\hat{X} \sim G(X)$ that imitate the characteristics of the original dependency, as seen in recorded data. Following the reasoning that deep generative models estimate the underlying data distribution to generate realistic fake data, we envision the primary tool in the stimulus generation system to be a GAN $\mathcal{G}$. A prerequisite for training the GAN is to assemble a dataset $\mathcal{X} = \{X_1, X_2, ...\},\ X_i \sim G(X)$ of signal traces recorded from the external interface of the real dependency g.

Having discussed the outputs and process of stimulus generation, we then consider what is perhaps its least apparent aspect – its input. Once it is trained on a dataset of recorded traces $\mathcal{X}$, the easiest way to use the GAN would be to sample it randomly to generate a mass of simulated traces. The problem, of course, is that such unconditional sampling may not at all help the overall test objective. After all, in the real world, testers tap into extensive domain knowledge to condition the test stimuli in accordance with the objective. A comparable conditioning mechanism is therefore an essential element in the stimulus generation system that we envision. For the moment, let us simply state that stimulus generation will be based on a condition $c \sim C$, where C is the distribution of possible conditions. This study suggests a few alternatives to condition the stimulus generation process, and we will examine them in forthcoming chapters. With the requirement of conditional sampling, there is a need to slightly revise the GAN. In order to encode the condition c into the stimulus generation process, we include an encoder network and denote the GAN as a pair of networks $\mathcal{E}$ and $\mathcal{G}$.

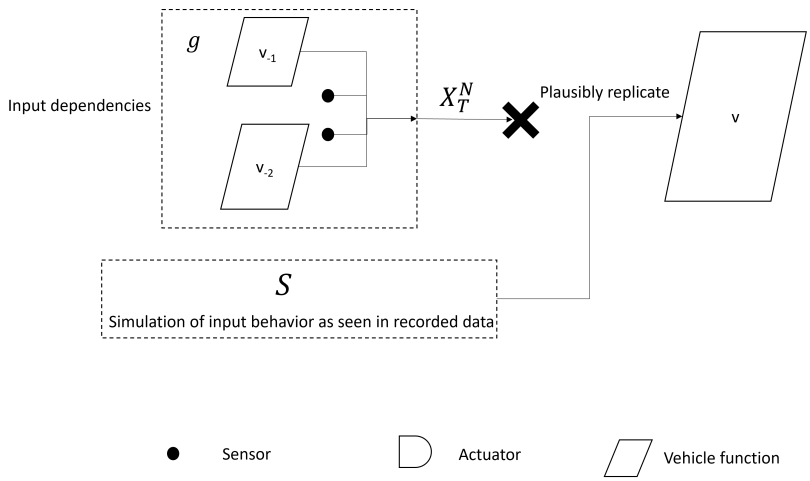


Figure 8.2: We envision a stimulus generation $\mathcal{S}$ that can plausibly replicate the input dependencies of a SWC V

Putting all of this together, the data-driven, deep learning system that we envision for generating plausibly realistic test stimuli can be concisely denoted as follows.

$$\hat{X} := \mathcal{S}(c,\ V;\ \mathcal{E},\ \mathcal{G}), \quad c \sim C,\ \hat{X} \sim G(X),\ X \in \mathcal{X} \tag{8.1}$$

Helping us guide the construction and evaluation of this system, we pose the following research questions.

RQ1 – Can the system $\mathcal{S}$ for plausibly realistic stimulus generation be constructed using a deep generative model trained on signal traces?

RQ2 – How can testers gain confidence that the stimuli generated by the system are plausibly realistic?

RQ3 – How can testers control the generation process so that generated stimuli help achieve their test objective?

RQ4 – Can the stimulus generation process be further refined given explicit knowledge about the code under test?

As the following chapters reveal, results from this study show that such a system for realistic test stimulus generation can indeed be constructed. By helping avoid laborious effort in manually specifying physical dependencies with high scope and fidelity, generative models trained on big signal data present a viable data-driven alternative. Trained stimulus generation models have the potential to significantly improve the credibility of simulation-based testing, allowing us to reduce the reliance on vehicle testing. Faster testing loops in credible virtual test rigs clearly help increase the cadence of software engineering, without compromising the quality, easing the delivery of the next wave of automotive software.

8.3 The system of signals and software studied

The traces X considered – The primary role of the system for stimulus generation, that we envision, is to realistically substitute the vehicle subsystem g which is an input dependency for SWCs under test. In this study, the vehicle system that we choose to broadly focus upon is the powertrain of the vehicle. Also known as the driveline or the drivetrain, the powertrain bears overall responsibility of propelling the vehicle forward. Since it is a core vehicle subsystem that strongly influences a major proportion of vehicle application software, several test rigs are incomplete without including a model of the powertrain as a dependency. The powertrain, which is itself composed of a variety of subsystems spanning multiple sciences, is clearly complex, but if we refer back to the brake system example that we sketched in Figure 8.1, its behavior can be represented by a carefully selected trace of signals. Consider the trace $X := X_T^N$ in Figure 8.3 which depicts transitions of $N = 2$ signals – *engine speed*, and *vehicle speed* for a duration of $T = 512$s. Of course, many more properties are required to describe the behavior of the powertrain with high fidelity, but the interplay between these two signals captures the mechanics of the powertrain with a fairly

wide scope. Capturing the state of the most important component of the powertrain – the engine – and the resultant speed with which it propels the vehicle, these two signals provide a reasonable snapshot of the dynamic state of the vehicle. As a starting point, we limit the system of signals to engine and vehicle speed alone, meaning that the stimulus generation system $\mathcal{S}$ generates traces that capture the interplay between these signals in a plausibly realistic manner.

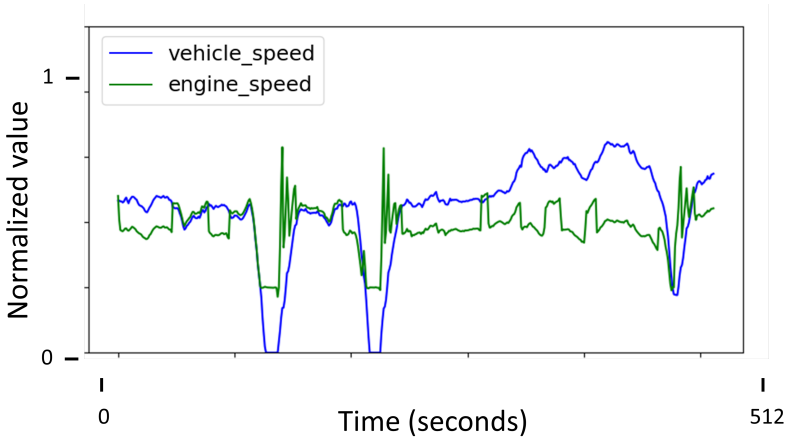


Figure 8.3: A trace of 2 signals capturing powertrain behavior

With the number of signals chosen for this study seeming small, we first clarify that most characteristics of the test stimulus generation system that we develop can be readily examined using a minimal set of signals. More importantly, while it is natural to think that simulation at high fidelity requires many more signals, it must be noted that there is always cost involved in choosing to include signals. In a trace X_T^N, such cost can be analyzed in terms of two important aspects, the number of signals N and the duration T, because recording each second of each signal incurs cost. Since signals are recorded by tapping into the external interface of SWCs, the first element of cost would be to expose signals externally. The true purpose of signals is to provide a channel for SWCs to collectively realize functionality. Recording it as data is an incidental act of repurposing. If an interesting signal is not available in the SWC interface, code needs to be updated to expose the signal, which is a significant cost. Even if a signal is exposed, another element of cost would be to include it in a recording campaign. Given the finite amount of resources like network bandwidth and storage, not every exposed signal can be recorded. Thus, while a rich system of signals improves the fidelity of modeling, the richness of information needs to be traded off with the costs involved in recording them. One important reason that this study focuses upon vehicle and engine speed is that they are key pieces of information, globally used in large parts of vehicle application software. This means that they are freely exchanged between SWCs and can be recorded at reasonable cost.

In addition to the cost of recording, including more signals introduces other concerns, especially in the process of generating test stimuli. In fact, to illustrate one such concern, and how we mitigate it, we introduce an additional signal

– selected gear – later on. Preliminarily limiting the system of signals to vehicle and engine speed, as a prerequisite for constructing the GAN that underpins this system, we assemble the dataset $\mathcal{X}$ as ∼100k 512s-long traces consisting of these two signals, recorded from 19 buses, of similar configuration, over a 3-5 year period. Samples from the training dataset are shown in Figure 8.4.

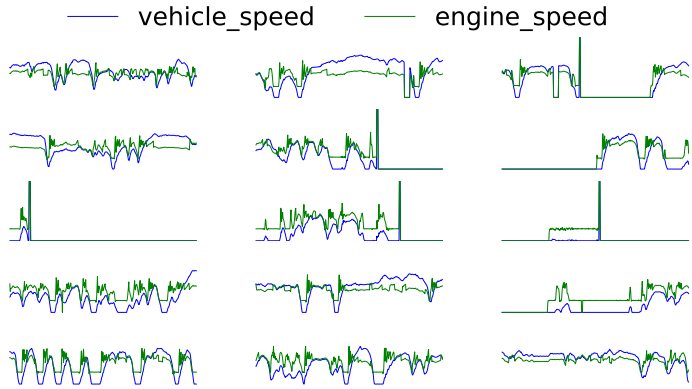


Figure 8.4: Samples from the dataset of signal traces

The nature of the SWC V considered – Having listed the two signals which will be generated and applied as test stimuli, let us shift the focus downstream to the subject of application – the SWC V. As we noted earlier, the system of signals that we choose in this study – vehicle speed and engine speed – are useful for many applications. Apart from propulsion, the engine is the source of all electrical and mechanical power used internally in the vehicle. For instance, SWCs that control pumps and motors in applications like cab climate control and power steering, which uses power originating from the engine, need to be informed about the engine speed. This also applies to SWCs managing external machines (like a concrete mixer) that take-off mechanical power[1] from the engine. Another example would be the brake system, where slip control SWCs use the engine speed to assess whether the vehicle is trying to accelerate. The other signal, vehicle speed, is such a critical piece of information that this signal is routed to numerous SWCs on the vehicle. If we take the TAS corpus of truck software that we studied in Chapter 4, the vehicle speed is routed to a staggering number of constituent SWCs. It is equally important to note that many of these SWCs listen to these signals in an open-loop manner, directly influencing neither. Put simply, generating traces consisting of just two signals – engine and vehicle speed, the envisioned stimulus generation system $\mathcal{S}$ helps test any open-loop SWC that takes one or both of these two signals as inputs, and there several SWCs that qualify.

[1] https://en.wikipedia.org/wiki/Power_take-off

Chapter 9

Building a system for sampling test stimuli

Upon defining the stimulus generation system $\mathcal{S}$, we proceed to build it in two different configurations, the first of which is described in this chapter. We refer to the first configuration as `logan`, – a portmanteau of 'logs', the popular term automotive engineers use for recorded signal traces, and GAN, the technique that we use for generating test stimuli. The `logan` system is designed for the context of black box stimulus generation, meaning that it generates traces without using any information about the actual software V under test. Forthcoming sections describe how the system is constructed, trained, and subsequently sampled to enable controlled generation of realistic test stimuli.

9.1 Constructing a system for sampling stimuli

Defining conditions c for stimulus generation – How do automotive software engineers specify test cases? Of course, this question does not have one answer. Unit tests for SWCs, attempting to achieve code coverage of individual functions, closely resemble unit tests written for any other embedded C code. Many of its dependencies would be mocked, inputs would be targeted to cover a particular branch of code, and expectations on outputs would be asserted. When the level of integration goes up and entire SWCs, or systems of SWCs, are tested, an interesting pattern begins to emerge. Since the external interface of SWCs consists of signals, the act of testing pretty much boils down to specifying signal traces to stimulate the input and asserting expectations on the SWC's output signal traces. This observation should come as no surprise because the stimulus generation system that we envision (Figure 8.2) itself attempts to generate a signal trace. Interestingly, the prevalence of using signal traces as test specifications extends even up to the vehicle level. Perhaps the most striking example of this mechanism would be standardized drive cycles which are literally

specified as a trace of vehicle, sometimes including engine, speed(s). Take the Worldwide Harmonized Light Vehicles Test Procedure (WLTP) [114], specified for benchmarking emissions and fuel consumption in cars. Attempting to simulate typical car driving, one version of the test specifies a drive cycle in the form of a 30-minute vehicle speed trace shown in Figure 9.1. Standards aside, specifying drive cycles for testing is common in day-to-day automotive engineering. For instance, engineers at Volvo use typical trips between selected cities in Sweden as a drive cycle to benchmark fuel consumption of long-haul trucks. When such drive cycles are tested under simulation, necessary conditions are simply specified as signal traces. With trace-based specification of test conditions being highly prevalent, we make the natural choice of considering the condition c, used as input for the stimulus generation system $\mathcal{S}$, as being just another signal trace.

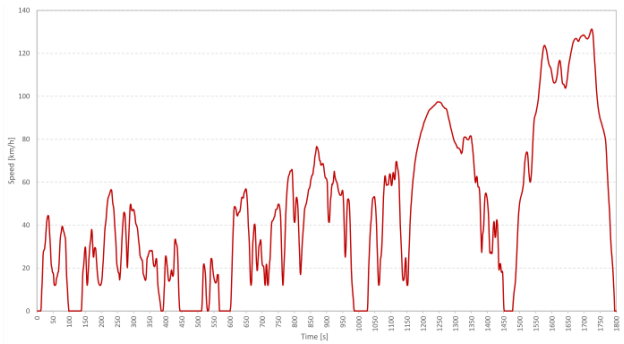


Figure 9.1: An example of a standardized drive cycle expressed as a vehicle speed trace

There is yet another factor that justifies the choice of framing test conditions as signal traces. If we examine Figure 8.2, the super-dependency g that we aim to simulate is an agglomeration of SWCs. This means that the input interface of g would be signals and its traffic can itself be represented as signal traces. Conditions can therefore be imagined to be outputs $c \sim H(c)$ of a process h that is upstream to g. While this strengthens the notion of specifying conditions as signal traces, it is not always helpful to literally replicate the original inputs of g as the condition. Especially, if we note in our case g broadly refers to the powertrain, requiring conditions to replicate powertrain inputs actually increases the burden of test specification. Instead, we use a simpler approach where, in order to generate a trace $\hat{X}_j \sim G(X)$, the condition is simply a real trace drawn from the same distribution $c := X_i \sim G(X)$.

At first glance, it may seem counterintuitive that both the condition c and the output $\hat{X}$ of the `logan` system are traces containing the same signals. However, basing one test condition upon another is a fairly standard practice in automotive software engineering. Before examining this using an example, we introduce some useful terminology. Vehicles often execute recognizable driving patterns, and we refer to a family of similar patterns as a driving *scenario*. One example of a driving scenario would be takeoff, where the vehicle starts rolling and subsequently begins to cruise. Depending upon a variety of factors like vehicle mass, road inclination, surface and traffic conditions, there are

several ways of executing takeoff. We refer to one specific execution of a driving scenario as a driving *maneuver*. Each trace X, by now a familiar construct to us, therefore depicts one maneuver. Figure 9.2 captures ∼10-minute long maneuvers of takeoff and stop scenarios using the vehicle speed and engine speed signals. Takeoff, for instance, is interesting because during its execution, apart from several core control actions, additional ones like automatic engine start, cabin heat circulation, and parking brake release are activated. Testing the behavior of such functions under many possible instances of takeoff can therefore be of great value. This is where the approach of specifying conditions c as traces brings benefits. One can set the condition $c := X_i$ as one instance of takeoff, perhaps the same takeoff instance shown in Figure 9.2, and task the stimulus generator to improvise new traces that possess the essential characteristics of takeoff. Putting together the prevalence of the signal trace as a mechanism for test specification, the fact that improvising one trace from another is valuable for testing and that a time series trace is a fairly routine data structure that DNNs can process, our choice of specifying conditions as trace seems quite sound.

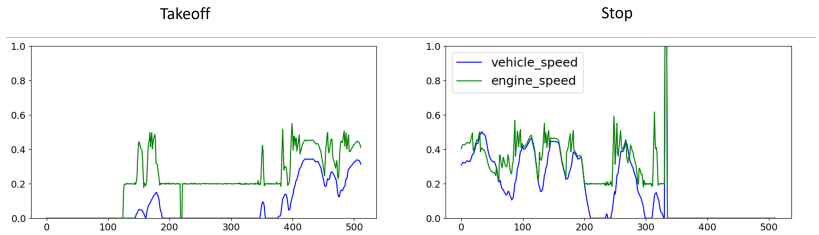


Figure 9.2: Examples of traces depicting maneuvers of a vehicle taking off and stopping

A strategy for conditionally sampling stimuli – In choosing to apply the condition c as yet another trace, the generative model that we envision in `logan` can be framed as a *trace* → *trace* mapping. If we refer back to the discussion in Section 7.3, we see that the vanilla GAN is much more suitable for an unconditional *latent code* → *trace* mapping, converting an abstract random code z_i into a trace $\hat{X}_i$. In order to explicitly model the condition, the vanilla GAN architecture (Figure 7.5) needs slight modification. As shown in Figure 9.3, with the inclusion of an additional encoder network $\mathcal{E}$, stimulus generation becomes a two-step process where (1) the encoder $\mathcal{E}$ takes a condition c_i as input and maps it into a latent code z_i, and (2) the generator network $\mathcal{G}$ maps the code z_i into a trace $\hat{X}_i$. Now, if the condition $c_i := X_i$ is defined as a signal trace, we seem to be proposing what is essentially an auto-encoding structure, mapping a trace X_i into a reconstructed version of itself $\hat{X}_i$. This is clearly of little use for test stimulus generation because, once testers have an interesting takeoff or stop trace X_i as a seed condition, why would they want to reconstruct the same trace? What is more useful, as we pointed out shortly before, would be to generate a trace $\hat{X}_j$, which is a variation of the condition X_i, but still preserving some of its essential characteristics. To achieve this efficiently, we undertake a two-step process depicted in Figure 9.3. First, we indeed train `logan` as an auto-encoder, mapping a condition $c_i := X_i$ to a reconstructed version of itself $\hat{X}_i$. Since we do not always know a priori the nature of the variation $\hat{X}_j$ that

testers want to generate, training a direct mapping from X_i to $\hat{X}_j$ is far from straightforward. On the other hand, from a training perspective, auto-encoding is a cheap self-supervised process to achieve mapping within the same domain of traces. Second, after training an auto-encoding GAN, we frame the conditional generation of related traces as a sampling problem. Specifically, we use the property of representational similarity which we first saw in Section 2.2. In the auto-encoder configuration $\mathcal{G}(\mathcal{E}(X_i))$, the latent code $z_i = \mathcal{E}(X_i)$ is the intermediate output of a composition of neural networks. This way, it is no more than a vector in a multidimensional embedding space, where representational similarity holds. This means that closely located latent codes z_i and z_j map to traces X_i and X_j that are likely to be semantically similar. Applying this insight, upon using the encoder to map the condition into a latent code $z_i = \mathcal{E}(X_i)$, we use some rule-based vector arithmetic r to map a new latent code $z_j = r(z_i)$. The newly mapped code z_j is then fed into the generator network to produce a trace $\hat{X}_j = \mathcal{G}(z_j)$. This way, due to the rule-based mapping r in vector space, we are able to efficiently arrive at a trace $\hat{X}_j$ which is a well-defined variation of the condition X_i, making it more useful as test stimulus.

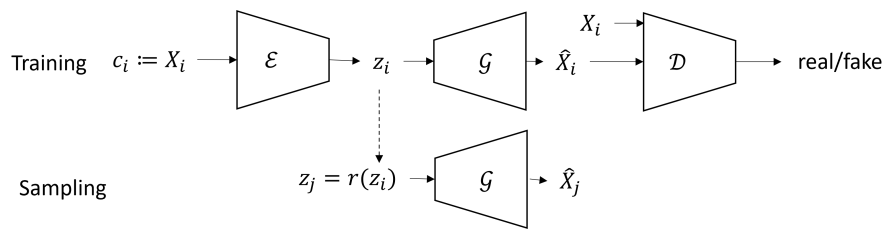


Figure 9.3: The `logan` test stimulus generation framework

Training `logan` – Even if the heart of the stimulus generation system is the $\mathcal{E} \circ \mathcal{G}$ auto-encoding process, it is not viable to train it simply as a reconstructing pair of networks. In a classic auto-encoder, the training process increases the likelihood that the mapping $X_i \to z_i \to \hat{X}_i$ is achievable. Here, we have the additional requirement that a latent code z_j, that is closely related to the encoded latent code $z_i = \mathcal{E}(X_i)$, can also be plausibly mapped into a realistic sample. To achieve smooth generation in a larger distribution of latent codes, the network $\mathcal{G}$ needs to be a true generator that can generally map latent codes sampled from a given distribution into traces. Put otherwise, `logan` must simultaneously achieve both auto-encoding and generation.

Actually, the importance in organizing a useful latent space is evident even in the vanilla GAN training objective that we saw in (9.2). Each training step samples codes $z_i \sim \mathcal{N}(0, I)$ and maps these codes into generated samples. In doing this repeatedly, the generator $\mathcal{G}(z_i)$ should ideally generalize to map any code sampled from the distribution $\mathcal{N}$ into realistic samples. One way to combine auto-encoding with generation would be to ensure that the latent codes $z_i = \mathcal{E}(X_i)$ also follow the same distribution $\mathcal{N}$. This way, the generator $\mathcal{G}$ can map latent codes no matter if they have been sampled from $\mathcal{N}$, or encoded by $\mathcal{E}$. In fact, the Variational Auto-encoder (VAE) [115], a generative modeling framework that was proposed contemporaneously with the GAN, introduces a

technique to achieve this. Unlike the GAN which sets up an adversarial game between the $(\mathcal{G}, \mathcal{D})$ pair of networks in Figure 9.3, the VAE relies purely on the $(\mathcal{E}, \mathcal{G})$ pair. This way, it bears closer resemblance to the classic auto-encoder, but with important modifications. Instead of mapping a sample into a latent code, the encoder is trained to output the parameters of a distribution $\mu_i, \sigma_i = \mathcal{E}(X_i)$. Then, the latent code is sampled from a distribution with the encoded parameters $z_i \sim \mathcal{N}(\mu_i, \sigma_i)$. One crucial advantage of this modification is that an additional training objective can be posed so that the distribution $\mathcal{N}(\mu_i, \sigma_i)$ can be matched to, say, the standard normal distribution $\mathcal{N}(0, I)$. This, in effect, achieves the organization of the latent space that we seek. The overall training objective of the VAE, as shown in (9.1), combines two sub-objectives.

$$\begin{aligned} z_i \sim & \mathcal{N}(\mu_i, \sigma_i), \quad \mu_i, \sigma_i = \mathcal{E}(X_i) \\ \mathcal{L}_{rec} = & ||X_i - \mathcal{G}(z_i)||_2 \\ \mathcal{L}_{kl} = & KL[\mathcal{N}(\mu_i, \sigma_i * I),\ \mathcal{N}(0, I)] \\ \mathcal{L}_{vae} = & \min_{\mathcal{E},\mathcal{G}}\ \mathbb{E}_{X_i}\ \ (\mathcal{L}_{rec} + \mathcal{L}_{kl}), \end{aligned} \tag{9.1}$$

In the absence of a discriminator, the reconstruction loss L_{rec} echoes the classic auto-encoder and simply minimizes the difference between input and generated samples. The distribution matching L_{kl} minimizes the Kullback-Leibler (KL) divergence[1] between the distribution of encoded parameters and the standard normal distribution. Put together, it does seem like the VAE offers both the essential elements that we seek – auto-encoding and smooth generation in an organized latent space. One drawback, however, is that the element-wise reconstruction error L_{rec}, that the VAE uses, has been observed to produce samples of poorer quality. The GAN, which uses a dedicated discriminator network $\mathcal{D}$ to assess the quality of generation has been seen to produce better samples. Making precisely the same chain of reasoning, the VAE/GAN framework [116] combines both, and we mainly base `logan` on this combined framework.

Considering the new auto-encoding structure, which is a departure from the vanilla GAN, the original adversarial objective (7.1) needs to be slightly modified. Instead of sampling latent codes from the standard normal distribution, it is drawn from the distribution whose parameters are output by the encoder.

$$\mathcal{L}_{gan} = \min_{\mathcal{G}}\ \max_{\mathcal{D}}\ log(\mathcal{D}(X_i)) + \mathbb{E}_{z_i} log(1 - \mathcal{D}(\mathcal{G}(z_i))),\ z_i \quad \sim \mathcal{N}(\mu_i, \sigma_i) \tag{9.2}$$

Next, noting the shortcomings of element-wise reconstruction, VAE/GAN proposes an alternative where reconstruction is conducted on features extracted from the discriminator. That is, instead of minimizing the differences between the real and generated samples, VAE/GAN suggests that the differences between their features, drawn from the l^{th} layer of the discriminator, be minimized.

[1] https://en.wikipedia.org/wiki/Kullback-Leibler_divergence

$$\mathcal{L}_{rec} = ||\mathcal{D}_l(X_i) - \mathcal{D}_l(\hat{X}_i)||^2, \quad \hat{X}_i = \mathcal{G}(z_i) \;\; z_i \quad \sim \mathcal{N}(\mu_i, \sigma_i) \tag{9.3}$$

Finally, preserving the KL loss for distribution matching, the VAE/GAN framework proposes that the encoder, generator, and discriminator networks be trained end-to-end as follows.

$$\mathcal{L}_{logan} = \min_{\mathcal{E},\mathcal{G}} \;\; \max_{\mathcal{D}} \;\; \mathbb{E}_{X_i} \; (\mathcal{L}_{gan} + \mathcal{L}_{rec} + \mathcal{L}_{kl}) \tag{9.4}$$

Upon trial-and-error in architecture selection, the encoder, decoder/generator, and discriminator are designed as 4–layer 1D convolutional neural networks with a kernel size of 8. Then, using the ∼100k traces of vehicle and engine speed signals that we collect in the dataset $\mathcal{X}$, we train the VAE/GAN end-to-end using the objective (9.4). Among suggestions in [58] to improve stability of GAN training, using the Adam optimizer with a learning rate of $2 \cdot 10^{-4}$ and a momentum of 0.5 greatly helped in achieving training convergence, while the recommendation of using *LeakyReLU* and *tanh* activations did not. Based on visual inspection, *ReLU* and *sigmoid* activations are found to produce samples of better quality, perhaps because of a significant amount of baseline-zero values in recorded signals, when the buses were turned off.

Evaluating `logan` – Naturally, one important aspect to consider during the construction of `logan` is the evaluation of generated samples. Since its eventual purpose is to serve as a virtual alternative to real SWC dependencies, signal traces that it generates should be plausibly realistic. In fact, evaluating the quality of generation becomes important even during training because the training process itself can be stopped only if the model is observed to be producing plausibly realistic traces. After training, when the GAN is sampled for test stimuli, techniques for evaluation continue to be important for measuring the quality of generated traces that are to be applied as test stimuli. Evaluating the quality of samples generated by GANs is an active area of research, with several new measures being routinely proposed. However, as pointed out in [117], most of the proposed techniques for evaluating GAN samples focus on the image domain, rendering them largely unsuitable for our purposes. In addition, with the growing use of neural language models for text generation, there are also a raft of techniques (examples in [118]) being reported for evaluating the quality of generated text, but these too are not directly reusable in our context. With a relative lack of readily available techniques that evaluate the quality of generated time series, and specifically signal traces, we resort to the basics.

Similarity as plausibility – the principle for evaluation – The mechanism that we adopt for evaluation is a two-sample test, where the quality of generation is evaluated by taking samples of real and generated traces and comparing their statistics. The fact that our primary strategy is one of conditional sampling, however, allows us to make useful adaptations. As sketched in Figure 9.3, the process that we adopt for conditional sampling involves encoding[2] a

[2]The actual encoding process is $z_i \sim \mathcal{N}(\mu_i, \sigma_i)$, $\mu_i, \sigma_i = \mathcal{E}(X_i)$, but we use $z_i = \mathcal{E}(X_i)$ for notational simplicity

trace $z_i = \mathcal{E}(X_i)$, choosing another code in its latent neighborhood $z_j = r(z_i)$, and decoding it into a new trace $\hat{X}_j = \mathcal{G}(z_i)$. The evaluation that we seek may be whether $\hat{X}_j$ is plausibly realistic but, since this process is seeded by the condition X_i, let us instead consider the question – given the input X_i, is the model's reconstruction $\hat{X}_i = \mathcal{G}(\mathcal{E}(X_i))$ of sufficiently good quality? Using a measure $M(X_i, \hat{X}_i)$ that compares the similarity between the condition and its reconstruction, this question is reasonably simple to answer. If the answer is yes, meaning the reconstruction is indeed similar to the original, it means that the latent code z_i, derived by encoding X_i maps to a plausibly realistic sample. Such an observation also makes it likely that the latent code z_j, as long as it does not stray too far away from z_i, also maps to a plausibly realistic trace $\hat{X}_j$. We refer to this line of reasoning *'Similarity as Plausibility'* (SAP), and this principle underpins our approach to both evaluating the quality of generation and the approach we develop for selective sampling. For now, we focus on using SAP to monitor the quality of generation during the training process.

During training, using the SAP reasoning, we monitor the quality of generation by conducting the two-sample test between a ∼10k-strong test set, randomly held out from the dataset $\mathcal{X}$, and their auto-encoded reconstructions produced by the model. Since the task at hand is to measure the similarity between a withheld test trace X_i and its reconstruction $\hat{X}_i$, a simple distance measure like L_1 may suffice as the metric M. Plain distance measures, however, compare values element-wise, and do not specifically focus upon larger spatial and temporal trends in the signal trace. Therefore, instead of distance measures we turn to measures of similarity that look beyond element-wise comparison. Traces X are multi-variate time series, which means that we can turn to any one of the numerous metrics and measures proposed [119] for comparing two time series. One interesting example is Dynamic Time Warping (DTW) [120], which measures the distances between two time series as a function of their (mis-)alignment in time. Further, considering that traces are of fixed duration (of 512 s), they can be treated in an image-like fashion, and a metric like Structural Similarity Index (SSIM) [121] can be used to compare traces. Originally designed for measuring similarity between two images, SSIM uses a wide range of statistics like the mean, variance, and covariance between the two inputs to compute a measure of perceptual similarity. When using DTW and SSIM to evaluate the quality of reconstruction, the latter is preferable to a certain extent because unlike DTW, which is a distance measure, SSIM between two traces evaluates to a real number $m \in [0, 1]$. The semantics of SSIM is therefore quite clear where $m = 0$ indicates no similarity while $m = 1$ indicates high structural similarity. Nevertheless, we use both metrics to monitor training because, as we soon see, the combined picture of reconstruction quality that they present is quite useful.

Having chosen metric measures, we evaluate the quality of test set reconstruction continuously during the training process. Figure 9.4 shows the progression of three metrics during training. From the figure we see that after a while, reconstruction SSIM begins to settle slightly above 0.9, indicating good structural similarity in reconstruction. Not merely taking this number at face value, we also visually inspect randomly selected test set samples to see that the reconstruction is indeed good. An extra element of confidence is that when

SSIM settles around 0.9, we also see that the DTW begins to settle at 2. In addition to these objective metrics of comparison, we also find it useful to monitor the quality of reconstruction using the subjective measure of $\mathcal{L}_{rec}$ defined in (9.3). This measure, which compares the difference between the abstract features of real and reconstructed test samples extracted from the l^{th} layer of the discriminator, is a statistical measure of similarity that the training process is tasked to minimize. When all three similarity measures stabilize, which happens around 15000 training steps (or batches), we have triple confirmation that the model is able to plausibly reconstruct test samples, and we stop training. This, according to the SAP principle enhances the likelihood that conditional sampling using this trained GAN should also lead to plausibly realistic traces.

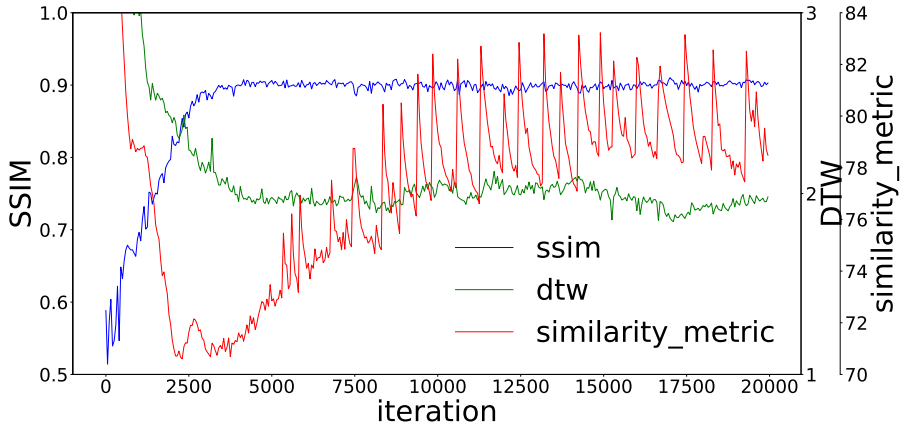


Figure 9.4: Monitoring the quality of generation during training by measuring the similarity between a test set and its reconstruction

We have thus seen how the SAP principle helps judge the quality of generation and the overall convergence of the process of training `logan`. In following sections, we show how this principle continues to crucially guide the sampling process, helping build confidence in the plausibility of generated stimuli.

9.2 Experiments in sampling stimuli

Selective sampling to address the problem of plenty – A blunt statement of the power of a GAN is that it can generate realistic samples, but an infinite number of them. Even a cursory glance at the original GAN (Figure 7.5), which maps a continuous space of latent codes to samples, makes this clear. The practical issue in sampling, therefore, is one of choice. The provision of a condition trace X_i to seed the generation process is a good start, but improvising further to produce related traces $\hat{X}_j$ presents challenges. Considering representational similarity in the latent space, a simple formula for improvisation may be, as shown below, to sample in the ϵ-neighborhood of an encoded condition trace.

$$
\begin{aligned}
z_i^\epsilon &= \{z \in \mathcal{Z} : \|z_i - z\|_2 \leq \epsilon\},\ z_i = \mathcal{E}(X_i) \\
\hat{X}_j &= \mathcal{G}(z), \quad z \sim U[z_i^\epsilon]
\end{aligned}
\tag{9.5}
$$

Sampling in the latent neighborhood of a condition trace is a simple mechanism for ensuring that generated traces are closely related to the condition, bringing much needed selectivity. While sampling in z_i^ϵ does not really eliminate the issue of infinite options, a more important issue is the choice of ϵ. Choosing a small value might mean that generated traces are far too similar to the condition and are not valuable as improvised stimuli. Conversely, choosing a large value may render the condition irrelevant because generated traces are far too dissimilar. Moreover, sampling in a large ϵ space is, to some extent, exactly the opposite of what the SAP principle calls for. In regions too far away from the condition trace, it may not be viable to base the plausibility of generation on the fitness of reconstructing the condition trace. We reason, therefore, that using a single condition trace may not be adequate to sufficiently bound an infinitely large latent space. Instead, we contend that it is more useful to elevate the geometry from a single point in the latent space to, say, a line.

Metric-based linear interpolation in the latent space – Let us now consider the case of using not one, but two traces X_1 and X_2 as conditions, and the two variants of stop scenarios, shown in Figure 9.5, serve as appropriate candidates. The first trace depicts a maneuver where the vehicle smoothly comes to a stop, while in the second trace, the vehicle is much more jerky in stopping. One interesting approach for testing would be to apply traces inbetween these two maneuvers as test stimuli. Unlike the case of using a single condition, by setting two traces as firm boundaries for the stimulus generation process, the intent of the tester is much more clear. Then, as notably shown by [58] in the context of images, we use the property of interpolation in the latent space to achieve semantic combination. That is, by interpolating a latent code, say, $z_{12} = 0.5 * (z_1 + z_2)$, the midpoint on the straight line between $z_1 = \mathcal{E}(X_1)$ and $z_2 = \mathcal{E}(X_2)$, we can generate a sample $\hat{X}_{12} = \mathcal{G}(z_{12})$ (see Figure 9.5) that mixes the characteristics of X_1 and X_2. Clearly, this is of valuable assistance for stimulus generation. The tester may have set two recorded traces as boundaries for stimuli, but `logan` is able to improvise a novel intermediate maneuver that is not recorded. This is an efficient and valuable process for augmenting test maneuvers in a controlled fashion, and vector arithmetic in the GAN latent space achieves this trivially. If we can interpolate new maneuvers, an obvious extension would be to ask – can we sample many more traces in between the two conditions to effectively 'sweep' this bounded test space? A straight line between codes z_1 and z_2 is a much more bounded space, but it is also a continuous one, where infinite points can be sampled. For further selectivity in sampling, we introduce a technique for Metric-based Linear Interpolation (MLERP), whose core innovation is to achieve selective sampling by jointly considering two spaces – the latent space z and the trace space X. Prior to presenting the algorithm itself, we first introduce its underlying reasoning.

Interpolation in the latent space z may achieve interesting semantic combination, but it takes place in an abstract space that is unintelligible to testers. Testers are

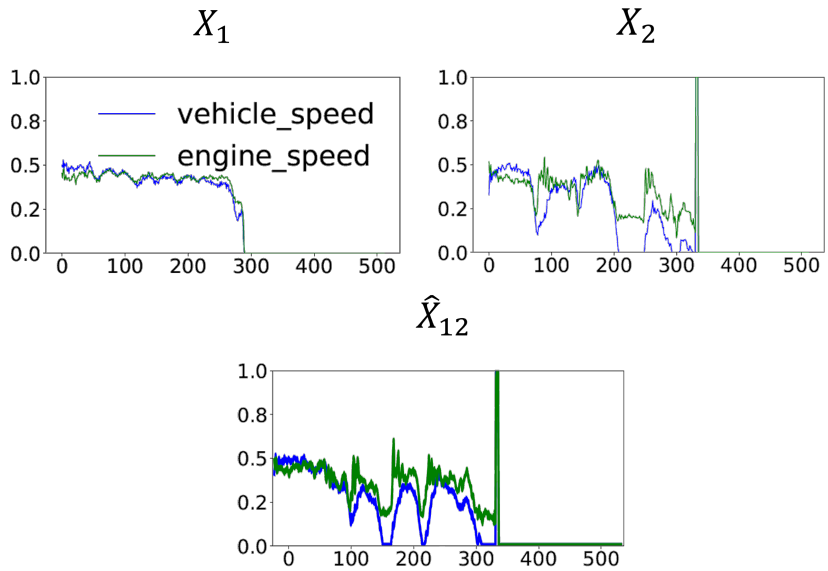


Figure 9.5: Recorded maneuvers of smooth (X_1) and jerky (X_2) stopping, and a generated trace combining both characteristics

more familiar with the space of traces X, and their ultimate objective remains choosing traces that can serve as valuable test stimuli. In order to allow testers operate in the much more familiar trace space, while still using the power of latent representations in the background, a joint reckoning of both spaces is necessary. To help bridge these two spaces, the essential operation in the MLERP algorithm remains linear interpolation in the latent space, combining latent codes of conditions X_1 and X_2 according to the proportion p, as shown in (9.6). With this, as shown below, conditionally sampling `logan` effectively reduces to choosing values in the space $p \in [0, 1]$, which we will refer to as the p-space.

$$\begin{aligned} z &= p \cdot \mathcal{E}(X_2) + (1-p) \cdot \mathcal{E}(X_1), \quad 0 \leq p \leq 1 \\ \hat{X} &= \mathcal{G}(z) \end{aligned} \tag{9.6}$$

As this sampling, with infinite possibilities, takes place in p-space, the question that testers face is, which among these are interesting, and plausible, to select as test stimuli? Since it is clearly difficult to visually inspect generated traces for suitability and plausibility, we propose the use of a metric M to quantitatively compare the properties of generated traces with those of a condition trace. While the introduction of a metric to guide the sampling process marks the beginning of the construction of MLERP, using metrics for comparing real and generated traces should be recognizable to us. It is, in fact, the core tenet of the SAP principle that we defined in the previous section for evaluating the quality of generated stimuli. While SAP calls for the use of a metric like SSIM

to evaluate the similarity between a test set and its reconstruction, MLERP expands its use to conditional sampling. Even in the straight line between points z_1 and z_2, a much more bounded latent space, MLERP imposes an additional measure of selectivity and plausibility by ensuring that generated traces follow a verifiable metric relation with one of the condition traces. Continuing to use SSIM as the metric M for comparison, and one of the condition traces X_2 as the reference for comparison, the metric evaluation shown below evaluates the structural similarity between the reference trace a generated trace $\hat{X}$.

$$m = M(X_2, \hat{X}), \quad \hat{X} = \mathcal{G}(z) \tag{9.7}$$

With the latent code z having been sampled according to (9.6), we have made a clear link between the GAN latent space, which is not interpretable, and a *metric* space which is very much interpretable. A value of $m = 0.5$ in the metric space means that the generated trace is halfway similar in structure when compared with the reference trace X_2. On the other hand, $p = 0.5$ simply indicates that the resulting latent code z is halfway in between reference latent codes. It does mean that z combines semantics equally in the latent space, but it is not always clear which characteristics end up being combined. Using the shorthand m-space to refer to this interpretable metric space, we quickly realize that this space is also bounded. A choice of $p = 0$ means that the generated trace is $\hat{X}_1$, a reconstructed version of the first reference trace X_1. Similarly, a choice of $p = 1$, generates $\hat{X}_2$, a reconstruction of X_2. Then, the boundaries of the m-space is given, according to (9.7), by $m_1 = M(X_2, \hat{X}_1)$ and $m_2 = M(X_2, \hat{X}_2)$. Within this bounded metric space, as shown in Figure 9.6, the strategy we choose in designing MLERP is to direct sampling in the p-space in such a way that it leads to any smooth change of metrics in the m-space that the tester desires.

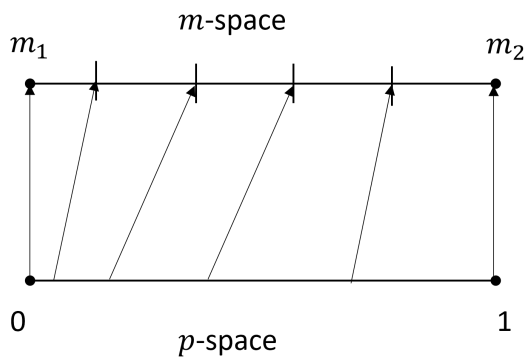


Figure 9.6: The essence of MLERP is that (1) testers specify a sequence of metrics that generated traces are expected to follow, (2) the algorithm finds corresponding codes in latent space that leads to the required metric variation

Reflecting this intent, the MLERP algorithm (Algorithm 2), requires testers to specify an increasing sequence of metric measures m_s, which generated traces are required to follow. A simple example, as shown in Figure 9.6, would be to

Algorithm 2: MLERP

Input : Trajectory m_s desired in the metric space, oversampling ratio s
Output : Traces $\hat{X}$ with metrics that comply with the specified trajectory

1 $z_1 \leftarrow \mathcal{E}(X_1)$, $z_2 \leftarrow \mathcal{E}(X_2)$ # boundaries in latent space
2 $\hat{X}_1 \leftarrow \mathcal{G}(z_1)$, $\hat{X}_2 \leftarrow \mathcal{G}(z_2)$ # boundaries in generated trace space
3 $m_1 \leftarrow M(X_2, \hat{X}_1)$, $m_2 \leftarrow M(X_2, \hat{X}_2)$ # boundaries in metric space
4 $N \leftarrow len(m_s)$
5 $p_s \leftarrow \text{linspace}(0, 1, N)$ # naive sampling
6 $q_s \leftarrow \text{linspace}(0, 1, s \cdot N)$ # oversampling
7 **for** $q \in enumerate(q_s)$ **do**
8 $\quad z_q = q \cdot z_2 + (1 - q) \cdot z_1$
9 $\quad m_q \leftarrow M(X_2, \mathcal{G}(z_q))$
10 $\quad ind \leftarrow \text{argmin}_{i \in [0,\ldots,N]}(|m_s[i] - m_q|)$
11 $\quad p \leftarrow p_s[ind]$, $z_p = p \cdot z_2 + (1 - p) \cdot z_1$
12 $\quad m_p \leftarrow M(X_2, \mathcal{G}(z_p))$
13 $\quad$ **if** $m_q < m_p$ **then**
14 $\quad\quad p_s[ind] \leftarrow q$
15 $\quad$ **end**
16 **end**

set $m_s = \texttt{linspace}(m_1, m_2, N)$, dividing the m-space into N equal partitions. With this, the tester is stating the intent that N traces need to be generated, and they are all required to linearly increase in SSIM from m_1 to m_2. Noting that the condition traces are two instances of the stop scenario, the linear m-space partitioning expresses the tester's requirement that generated traces change smoothly in SSIM from the first to the second stop maneuver. Of course, one can use linear interpolation in the latent space, according to (9.6), to generate N intermediate traces, but there is no guarantee that generated traces follow the metric trajectory m_s specified by the tester.

Now, we highlight a few observations. First, as shown in (9.6) and (9.7), the operations that map the latent p-space to the interpretable m-space are linear interpolation, the generator network $\mathcal{G}$ and the comparison metric M. Second, all these operations, including the DNN $\mathcal{G}$ and the SSIM metric M, are continuous functions. Third, knowing that the boundaries 0 and 1 in the p-space map to boundaries m_1 and m_2 in m-space, the intermediate value theorem[3] holds that any desired intermediate metric value m_s between the limits of m-space is reachable from the p-space. Put together, these three observations direct the design of the MLERP algorithm. Now, linear interpolation by naively partitioning the p-space into N points (Line 5) may not achieve the desired metric trajectory m_s, but it nevertheless sets a baseline. Knowing that there are points in the p-space that are guaranteed to map to required metrics in m-space, we increase the partitioning in the p-space by a factor of s (Line 6) to create a denser sequence of points q_s for linear interpolation in the latent space. Then, by iterating through the denser sequence q_s, MLERP selects traces generated from the oversampled latent space that follow the metric trajectory better than the naively sampled sequence p_s. Note that, due to the

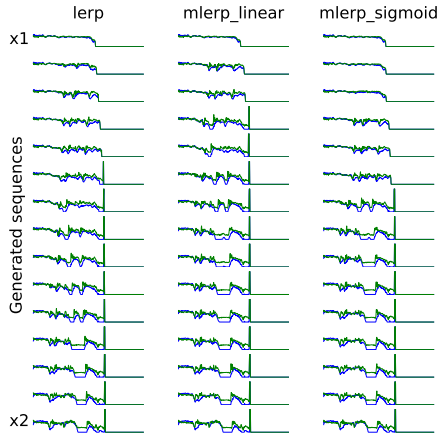
[3] https://en.wikipedia.org/wiki/Intermediate_value_theorem

intermediate value theorem, given a high enough oversampling factor, we are bound to discover points in the p-space that map to generated traces which, in turn, match any metric trajectory m_s specified by the tester.

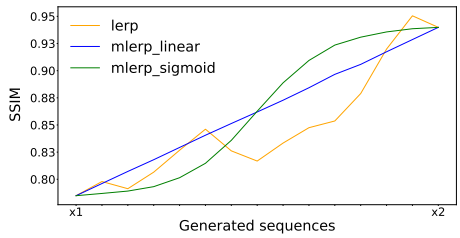
For our specific case of generating intermediate traces between the two stop maneuvers X_1 and X_2 shown in Figure 9.5, the result of using MLERP is shown in Figure 9.7(a). In this figure, we show three columns, the first of which marked `lerp` shows traces generated using naive linear partitioning in p-space. As also shown in Figure 9.7(b) such naive linear interpolation neither leads to a smooth nor a predictable change in the metric space. If we now turn our attention to the `mlerp_linear` column of generated traces, at first glance, we may not be able to spot major differences from naive interpolation. However, if we look at its metric trajectory, we can see that the generated traces closely approximate the linear increase in SSIM that the tester requires. Using metric-guided sampling, we therefore show that generated traces are *verifiably plausible* and *finely selectable*, providing a much-needed tool that allows testers to improvise realistic unrecorded stimuli with fine-grained control. The sequence of traces in the `mlerp_linear` column indicates that MLERP can be used to generate a properly reasoned and controlled series of traces, which can in turn be applied as stimuli for testing vehicle application software.

If we examine, once again, generated traces in Figure 9.7(b), there is a third `mlerp_sigmoid` column. This, as shown in the corresponding metric variation in Figure 9.7(b), shows that MLERP can adapt to trajectories more complex than linear partitioning in metric space. Here, samples are specified to follow the trajectory of a sigmoid function, producing more samples around $m = 0.5$ than the linear case. This is yet another demonstration of the fine-grained control that MLERP offers testers. It also goes without saying that the metric M for evaluating properties in the trace space can also be changed, as long as it remains a continuous function. Finally, it is important to note that MLERP is computationally heavier when compared to naive interpolation. One indication of the increased computational load is the oversampling ratio, where MLERP iterates through s times as many latent codes as the naive case. The choice of s, which is essentially the level by which the sampling space is enlarged, also needs to be tuned according to the desired trajectory. In the examples we show, the linear trajectory in metric space was achieved using oversampling by a factor of 15, while the sigmoid trajectory required a factor of 30. In any case, the computational cost of oversampling can be minimized by generating traces once and reusing it for many test runs, a common practice in regression testing.

Thus, the `logan` system with the MLERP technique for selective stimulus generation is the culmination of three major developments. First is the GAN itself, that is trained on signal traces, which provides a platform for generating fake, yet plausibly realistic, signal traces that can be applied as test stimuli. Second, the concept of choosing one or more reference traces, in combination with conditional sampling in the latent space, allows testers to efficiently impose granular levels of control in the stimulus generation process. Third, MLERP provides another level of control, allowing testers to use a metric and automatically select traces that are both plausibly realistic and useful for a particular



(a) Generated traces



(b) SSIM of generated traces

Figure 9.7: Generating intermediate samples between two stop maneuvers X_1 and X_2 using naive linear interpolation (*lerp*), MLERP with linear increase in SSIM s=15 (*mlerp_linear*), and MLERP with sigmoid increase in SSIM and s=30 (*mlerp_sigmoid*)

test. Together, these developments present a viable platform for controlled generation of realistic stimulus that improves the credibility of virtual testing.

Chapter 10

Building a system for searching test stimuli

Having described the first configuration of the envisioned stimulus generation system, we now proceed to describe the second. We refer to the new configuration as `silgan`, or 'software-in-the-loop' GAN, which introduces two main innovations. First, it simplifies the specification of condition traces c, easing stimulus generation even with an expanded number of signals in traces. Next, unlike `logan` which is a black box stimulus generator, `silgan` uses information about software under test V to actively guide the stimulus generation process. We now proceed to describe how `silgan` is constructed, trained, and applied.

10.1 Expanding the set of signals

Implications in expanding the system of signals – So far, we have seen experiments where traces of two signals – engine and vehicle speed – have been generated as potential test stimuli. Attempting to simulate elements of powertrain behavior, as acknowledged in Section 8.3, these two signals offer a relatively limited picture. For credible testing in a virtual rig, there may be a need to expand the scope of simulated behavior beyond just these two signals. One straightforward way to expand scope is to expand the trace itself and include more signals that capture additional aspects of powertrain behavior. In fact, we had previously stated our intention to conduct such an exercise in Chapter 8, expanding the trace to add one more signal, selected gear. We now undertake this expansion. Returning to the dataset $\mathcal{X}$ of recorded traces that was used to train `logan`, we include the selected gear signal in each trace. Traces remain $T = 512$s long but now contain $N = 3$ signals, and one example of the newly expanded trace is shown in Figure 10.1. The added selected gear signal, depicting the state of the transmission, does provide extra insights into powertrain behavior. Unrelated to the expansion in signals, we also take the opportunity to

increase the size of the training set to ∼200k samples. While we had pointed out earlier that expanding the list of signals always incurs cost, we had mostly discussed costs associated with recording them. We now additionally consider the impact of an expanded system of signals in the process of training and sampling generative models. In some ways, including more signals has minimal impact on training the GAN. An expanded set of signals does mean that `logan` needs to be retrained, but if we revisit the network architecture and training objectives discussed in Section 9.1, we can see that neither is particularly vulnerable to the number of signals in traces X. Then, since the DNNs in the GAN need to deal with more signals, it is advisable to increase the number of learnable parameters (like the number of layers). This, in turn, increases compute and memory costs of training, but only marginally. While a relatively meager expansion in the system of signals may not significantly affect GAN training, we reason that the effect on GAN sampling, especially conditional sampling, can be acute.

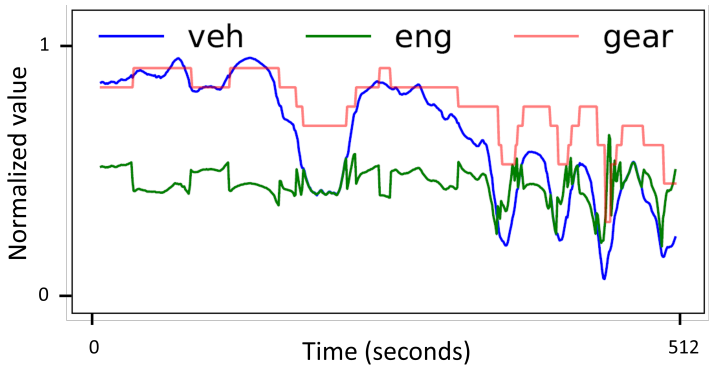


Figure 10.1: A trace of 3 signals capturing powertrain behavior

In the MLERP algorithm, the primary mechanism used for conditional sampling are condition traces X_1 and X_2, which we use to bound the latent sampling space to a straight line. Clearly, increasing the number of signals in each trace bears no additional burden on such a conditioning technique. The larger problem, however, centers around the selection of traces that can be used as conditions. All experiments in test stimulus generation that we have seen so far assume that testers have curated interesting traces – like start and stop maneuvers – that can be applied as conditions. Unfortunately, such curation is not unlike annotation and can take significant effort. If conditional sampling of test stimuli were to depend upon curated traces alone, then the effort involved in curation becomes a major impediment in the overall process of stimulus generation. There is one alternative to curating traces, a technique that we have briefly come across in earlier discussions, and that is handcrafting signal traces. When it comes to testing, there is some contention between using handcrafted and curated traces as stimuli. It is useful to spend a moment examining it because at the heart of this contention lies the issue of an expanding set of signals.

Even if it is a technique that scales poorly, testers routinely handcraft signal traces as test specifications. There is, in fact, a simple reason for this preference – handcrafting has the ability to capture the tester's intent clearly, especially

when real-life recorded counterparts of intended test conditions are rare. Take the WLTP drive cycle, an example of which we saw in Figure 9.1, which is essentially a handcrafted trace of the vehicle speed signal. A standardized test cycle like this is unlikely to occur in real-life driving and, unless it comes from vehicles in a test track, it is near-impossible find such a maneuver in recorded traces. In order to construct such benchmark cycles, especially if it is expressed as a single signal (vehicle speed in the WLTP example), handcrafting may actually be most suitable. Now, if the number of signals expands by two or more, the explosion of possible multidimensional temporal transitions makes handcrafting far more difficult. Taking our newly expanded list of signals as an example, it may be quite impractical to jointly handcraft transitions of vehicle speed, engine speed, and selected gear, keeping all their correlations intact, beyond a fairly limited duration. Even if one tried to handcraft such a multi-variate trace, it may end up being a fairly poor imitation of the actual trace. Thus, for traces with two or more signals lasting hundreds of seconds, even if it bears extra cost, testers are better off avoiding handcrafting in favor of using curated traces. Even if it depends upon curation, the `logan` system in fact offers a way to at least keep the curation effort manageable. A GAN that does *trace→trace* mapping can augment a limited set of curated traces. If such a complex mapping is achievable, can we not use the capabilities of generative models to directly map handcrafted imitations into realistic traces? And, in doing so, can we not bridge the two contending schools of test specification?

Specifying conditions as template traces – The difficulty in handcrafting a realistic trace arises from two problems - (i) the need to specify detailed short and long term transitions of each constituent signal, and (ii) capturing possible inter-signal dependencies. What would help, therefore, is a specification mechanism that ignores details and focuses only on the basic profile of the trace. Here, we propose piece-wise linear approximation as a simple way to avoid specifying transient details of a signal. The resulting *template*, as shown in Figure 10.2 serves as a rough, intuitive sketch of the long-term signal profile. Linear approximation may have simplified handcrafting one signal, but how do we scale to a multi-variate case? Sketching one template for each constituent signal is one option but, even with templates, the challenge of keeping their correlations plausible still remains. Having witnessed the capabilities of conditional generation in a GAN, we propose a much simpler approach of specifying a driving scenario by sketching a template for *any one* constituent signal. Then, we train a GAN to directly map a univariate template into a realistic multi-variate trace. This completely eliminates the need to address inter-signal dependencies in specifying conditions, something that scales well to an expanding set of signals. As seen in Figure 10.2, this also means that a tester can choose to define a takeoff scenario by specifying only a vehicle speed template. The same tester can choose to define a cruise scenario by only specifying a template for the engine speed. This flexibility allows test case design to focus on a single chosen control signal, further reducing the specification effort. Then, as we soon show, we simply train a generative model to restore the details of the sketched signal, and also generate realistically inter-related accompanying signals. A method, which takes a handcrafted univariate template as input and generates a realistic multivariate trace as output surely bridges the two contending techniques of test specification.

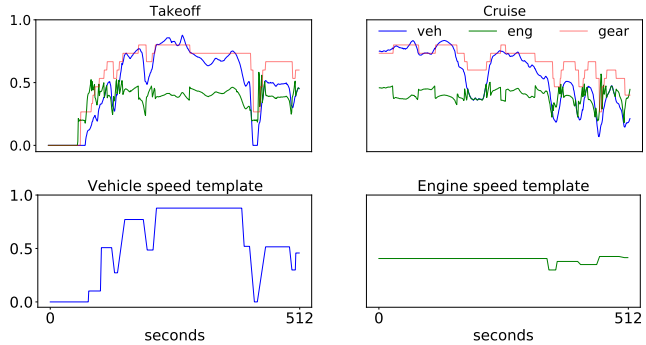


Figure 10.2: Recorded driving maneuvers (a) and corresponding templates (b). Signal values normalized

More broadly, we reason that handcrafted templates may very well constitute a generally reusable mechanism for defining scenarios of vehicle operation. As surveyed in [122], recent years have seen many proposals that help specify scenarios for testing automotive software systems. However, a major limitation that they share is a focus on specifying scenarios only in terms of high-level driving characteristics. For example, popular ontologies like [123] describe driving scenarios mostly in terms of tactical aspects like avoiding obstacles or changing lanes. Such proposals may have been used to analyze and test driver assistance or autonomous driving controllers (for example, [124]), but their focus on tactical scenarios comes at the cost of operational granularity. Vehicle control software at the operational level, i.e. those for functions like fuel injection or automatic gear shifting, react to transitions of a multitude of internal vehicle systems that are often of little concern at the tactical level. Testing these controllers under different driving scenarios would therefore require that testers are able to specify operational scenarios at a granular level of detail. Templates can serve as a powerful tool to specify such operational scenarios. The template of engine speed, something that often falls outside the scope of existing scenario specification frameworks, can capture the operational perspective that is greatly needed for testing the majority of vehicle application software. Also, by specifying long-range driving scenarios at a level of detail suitable for testing entire sub-systems of SWCs, templates are also different from alternative proposals like [125], which focus on specifying test scenarios for individual software routines. While templates clearly have the potential to ease the specification of vehicle operation scenarios, it is equally important to note that the idea of a template is deeply entangled with the idea of a generative model. Restoring the details that the template eschews is quite complex and is difficult to achieve using rule-based algorithms.

10.2 Expanding the system for generating stimuli

Using templates, instead of traces, for conditional sampling significantly changes the process of stimulus generation. It is, therefore, time to upgrade the previously discussed `logan`, to the new configuration `silgan`. The 'sil' in the new system primarily refers to the software-in-the-loop paradigm where simulation models, including the trained variant we develop, are predominantly used. The 'sil' prefix is poised to take much more significance since, as shown shortly, we actually connect vehicle application software in a feedback loop with a GAN.

Translating templates to traces – Let us define the domain of univariate template conditions as $\mathcal{C}$ and, reusing the notation of the training dataset, that of N-dimensional traces as $\mathcal{X}$. Let $P_n(c_n)$ denote the distribution of templates for the n^{th} signal of the N-dimensional system, and $P(X)$ denote the distribution of traces. Denoting the duration of templates and traces by T, a template $c_n \sim P_n(c_n)$, is a time series in $\mathbb{R}^{1\times T}$, while a trace $X \sim P(X)$ is a time series in $\mathbb{R}^{N\times T}$. The objective is to learn the *template→trace* mapping that can translate a template c_n into a realistic trace $\hat{X} \sim P_n(X|c_n)$, that is faithful to the specified profile in the n^{th} signal and generates plausible related variations for all $(N-1)$ other signals. Translating a template into a trace may be complex but, given a recorded trace X, piecewise linear templates $c_{(N)} = (c_1, ...c_N)$ for all N signals can be automatically extracted using relatively simple procedures. Upon smoothing a signal using windowed mean and 1-D Sobel[1] filtering, we simply extract the template as flat regions around major points inflection, with straight-line edges connecting these regions (refer Figure 10.2). Using extracted templates, we restructure the training data into a set of pairs $(c_{(N)}, X)$ of templates $c_n \sim P_n(c_n)$ and recorded traces $X \sim P(X)$ they were extracted from. Technically, pairing templates with traces makes this labeled data, where templates count as annotations. It is, however, easy to see that the template extraction process is trivial and poses costs so minimal that the labeling process can still be reasoned to fall under the self-supervised regime. In using this paired training set, it is important to note our assumption that templates specified by a tester can be plausibly sampled from the distribution of extracted templates $P_n(c_n)$. Finally, since a template c_n is always associated with n, the signal which it approximates, for ease of notation, we cease associating it, or any of its mappings, explicitly with n.

Designing and training `silgan` – With the task of *trace→trace* mapping, originally defined for `logan`, changing into a more complex *template→trace* mapping, the `silgan` network layout needs redesign. As shown in Figure 10.3, the updated network design is more elaborate, involving a forward and reverse process, amounting to five networks and almost as many intermediate steps.

Perhaps the easiest way to deconstruct this network is to first examine the forward process which closely reflects the network design of `logan`. Given a template c as input, a translated maneuver $\hat{X}$ is expected to realistically render

[1] https://en.wikipedia.org/wiki/Sobel_operator

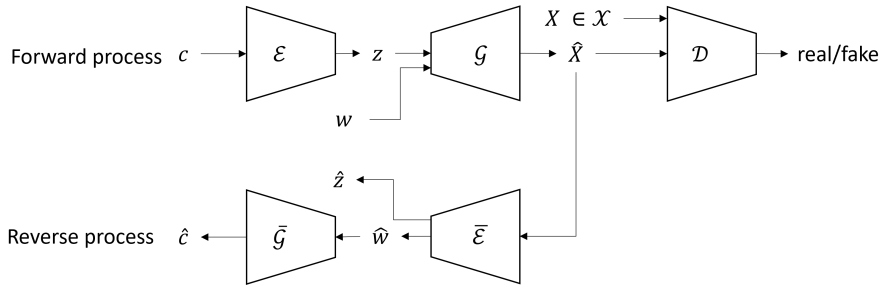


Figure 10.3: The `silgan` network for template-based stimulus generation

the template c as its n^{th} signal. Since c and $\hat{X}$ clearly share some characteristics, let us define $\mathcal{Z}$ as a latent intermediate domain that encodes this shared information. Let $\mathcal{E} : \mathcal{C} \rightarrow \mathcal{Z}$ be a learnable encoder that maps a template into this domain. However, the translated maneuver $\hat{X}$ is also required to render realistic transitions of all other signals. Let us therefore define another latent domain $\mathcal{W}$ as the source of all this additional information. Allowing the learning process to optimally structure its information, codes from this domain are sampled as $w \sim \mathcal{N}(0, I)$. The translation process can then be completed by defining a learnable generator $\mathcal{G} : \mathcal{Z} \times \mathcal{W} \rightarrow \mathcal{X}$, that produces a translated trace by combining information from both latent domains. This completes the generation (or translation) part of the forward process, which can be formally denoted as follows.

$$\hat{X} = \mathcal{G}(z, w), \quad z = \mathcal{E}(c), \; w \sim \mathcal{N}(0, I) \tag{10.1}$$

Technically, the encoder also needs to know n, the signal for which c serves as a template, but we ignore this for notational simplicity. Then, in order to ensure that the translated maneuver $\hat{X}$ is realistic, just like the `logan` setup, we use a discriminator $\mathcal{D}$ to assesses the realism of translated traces. Though we could have adapted the adversarial objective defined for `logan` for the new setup, we make an important adjustment. The `logan` objective (9.4), reflecting the original GAN adversarial objective, trains the discriminator as a classifier. That is, the discriminator takes a set of traces at its input and labels them as either real or fake. As pointed out in [126], training the discriminator as a classifier has a few drawbacks, the most important of which is that of vanishing gradients. Due to the very nature of the binary-cross entropy loss used to train the classifier, as soon as the discriminator successfully classifies an image as real or fake, it begins to provide weaker feedback (gradients) to the training process. This eventually ends up affecting the quality of generation. In order to alleviate this problem, we use an alternate least-squares discriminator [127] objective which trains the discriminator as a regressor. Using a mean squared error to calculate the loss between the discriminator's assessment and the ground truth 0(fake) and 1(real) labels, the discriminator is able to provide useful feedback (stronger gradients) even if a generated sample has been assessed to be on the correct side of the decision boundary. This approach has been observed to make GAN training far more stable, resulting in generated samples of better

quality. The modified adversarial objective the forward process is shown below.

$$\begin{aligned}\mathcal{L}_{Gen} &= \mathbb{E}_{c,w}\ (D(\hat{X}) - 1)^2 \\ \mathcal{L}_{Dis} &= \mathbb{E}_{X}\ (D(X) - 1)^2 + \mathbb{E}_{c,w}\ D(\hat{X})^2\end{aligned} \tag{10.2}$$

The description of the forward process that we have seen so far may seem to largely conform with the `logan` approach, differing only in fine details. However, as shown in Figure 10.3, the biggest difference is the multi-modality in translation. That is, upon encoding a template into z, the generator network $\mathcal{G}$ can combine it with different codes w to produce different traces, all of which satisfy the profile requirements set by z. Such a technique of defining disentangled, partially shared, information domains for multi-modal domain translation, first proposed in [128], has profound implications for stimulus generation. Not only does `silgan` allow minimally defined templates to be translated into realistic traces, but it also provides an additional degree of variety in improvising signals whose profile is not bound by the template.

Noting the rich promise of multimodal translation, we now highlight a potential pitfall which can limit this capability. Normally, the adversarial loss alone is sufficient to ensure both translation into a realistic maneuver and adherence to the profile in the template. However, we find that this adherence is sensitive to random seeding and therefore not always achievable. A stronger imposition of adherence while translating template c, can come in the form of a pairing loss, shown below. This loss encourages the translated trace $\hat{X}$ to closely resemble the corresponding real trace X that the template is extracted from. Now, it is important to note that strict reconstruction, forcing the template to map to one particular translation, clearly restricts the diversity inherent in multimodal transition. Use of this loss term, should therefore follow careful consideration.

$$\mathcal{L}_{Pair} = \mathbb{E}_{c,w,X}\ ||\ \hat{X} - X\ ||_1 \tag{10.3}$$

Having traced the forward process, let us now examine the reverse process in `silgan`, starting with why it is useful. When using a GAN to translate from one domain into another, prior work has shown that the quality of translation can be improved by encouraging cycle consistency [129]. That is, having translated template c into a maneuver $\hat{X}$, it is beneficial to reverse the translation and recover the template. We therefore define inversion networks $\bar{\mathcal{E}} : \mathcal{X} \rightarrow \mathcal{Z} \times \mathcal{W}$ and $\bar{\mathcal{G}} : \mathcal{Z} \rightarrow \mathcal{C}$ that reverse the translation. Unlike template extraction procedures used in preparing the training set which achieve the same end, this reverse translation is differentiable. This allows the following cycle consistency objective, the L_1 loss between input and recovered templates, to back propagate gradients from the reverse to the forward process during training.

$$\begin{aligned}\hat{c} &= \bar{\mathcal{G}}(\hat{z}),\ \hat{z}, \hat{w} = \bar{\mathcal{E}}(\hat{X}) \\ \mathcal{L}_{Cyc} &= \mathbb{E}_{c,w,X}\ ||\ \hat{c} - c\ ||_1\end{aligned} \tag{10.4}$$

Further, upon close examination of the network layout in Figure 10.3, one can notice several auto-encoding maps. The composition $\mathcal{E} \circ \bar{\mathcal{G}}$ achieves a *template*→*template* mapping, while $\bar{\mathcal{E}} \circ \mathcal{G}$ achieves *trace*→*trace* mapping. More interestingly, perhaps, the $\mathcal{G} \circ \bar{\mathcal{E}}$ composition auto-encodes latent codes z and w. As shown in [128], exploiting this latent auto-encoder, sampling these codes from priors $\mathcal{N}(0, I)$ and reconstructing them is a far more simple method to organize the distribution of the latent space when compared to the VAE approach seen in (9.1). Thus, we additionally use the code reconstruction objective shown below.

$$\mathcal{L}_{Crec} = \mathbb{E}_{z,w} \; ||\bar{\mathcal{E}}(\mathcal{G}(z, w)) - (z, w) \; ||_1, \quad z, w \sim \mathcal{N}(0, I) \tag{10.5}$$

Bringing the objectives of translation, cycle reconstruction, and code reconstruction together, `silgan` is trained end-to-end using the following composite objective. Just like `logan`, each network is composed using 4 1-D convolutional layers.

$$\mathcal{L}_{silgan} = \min_{\mathcal{E},\mathcal{G},\bar{\mathcal{E}},\bar{\mathcal{G}}} \; (\mathcal{L}_{Gen} + \mathcal{L}_{Pair} + \mathcal{L}_{Cyc} + \mathcal{L}_{Crec}) + \min_{\mathcal{D}} \; \mathcal{L}_{Dis} \tag{10.6}$$

Upon training, we are able to translate univariate templates into multimodal, multi-variate traces, examples of which are shown in Figure 10.4. This, we reason, is quite a powerful demonstration of `silgan`'s expanded properties of stimulus generation. As seen in the figure, all we need to generate a rich variety of realistic stimuli is a rudimentary sketch of one of its signals.

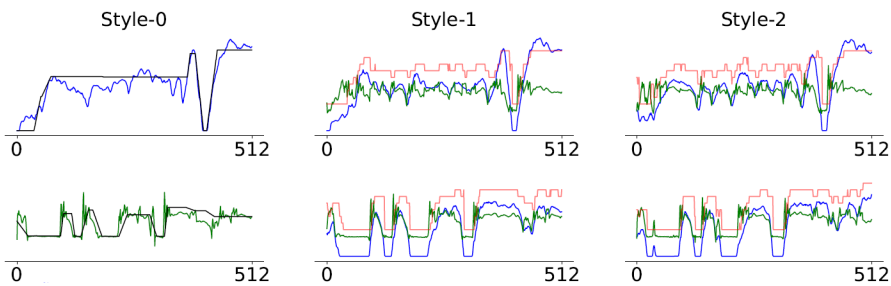


Figure 10.4: Examples of template to trace translation using `silgan`

Despite its many affinities with the `logan` setup, the new `silgan` layout differs in significant ways, the most important of which is the nature of the $\mathcal{E} \circ \mathcal{G}$ composition of networks. In `logan`, these networks constitute an auto-encoder while in `silgan`, they undertake multimodal translation. By definition, multimodal translation means that one template can translate into multiple traces and this means that there is no firm ground truth against which translated traces can be assessed. This raises an important implication – in the absence of a ground truth to assess the translation, the SAP principle for GAN evaluation is not directly applicable. When it was originally defined for `logan`, SAP was based on the presence of an auto-encoder where the ground truth for reconstruction is readily available. This allowed us to evaluate the quality of reconstruction

and extrapolate the judgment to near neighborhoods in latent space. In order to better fit the realities of `silgan`, SAP needs to be reinterpreted. If we take a step back and observe the complete layout, we see that there is a *template→template* cycle-reconstruction process $\mathcal{E} \circ \mathcal{G} \circ \bar{\mathcal{E}} \circ \bar{\mathcal{G}}$. Though this is not an auto-encoder in the traditional sense, it takes a template c as input, traces through the forward and backward processes and recovers the template as $\hat{c}$. As an end-to-end indication of the quality of `silgan`, we therefore monitor the quality of template cycle-reconstruction during training. In this reinterpretation of SAP for `silgan`, the two-sample test is conducted by holding-out a test set of ∼20k traces and corresponding extracted templates. Then during training, the quality of cycle reconstruction is periodically measured as the average SSIM between a test set template c and 4 cycle translations $\hat{c}$, each using a different random code w. After training for ∼15 epochs, cycle reconstruction SSIM, averaged over the entire test set, settles around 0.95. While such high similarity of cycle reconstruction indicates healthy training, the quality of translation is additionally verified by visually inspecting around hundred randomly selected test template translations. Visual inspection of time series maneuvers confirms adherence to the template and plausibility of characteristic features like correlations between vehicle and engine speed under the influence of gear shifts.

Measuring the quality of template cycle-reconstruction may be a convenient reinterpretation of SAP, but it is, nonetheless, a weaker form of evaluation. Cycle-reconstruction is, after all, only an auxiliary objective introduced to improve training convergence. Upon training, the network is still meant to be mainly used for translation, which the reinterpreted SAP does not directly address. We noted earlier that the primary obstacle in applying SAP is the multimodal nature of translation and the diversity of traces it generates. One way to work around this issue is to increase the importance – essentially scale it by a large factor – of the $\mathcal{L}_{Pair}$ loss defined in (10.3). The express purpose of this loss is to encourage that the template c translates much more faithfully into the trace X from which it was extracted. This way, there is a ground truth X against which the translated trace $\hat{X}$ can be compared and SAP becomes eminently applicable. What we lose, in taking this approach, is the rich diversity of multimodal translation. What we gain, is that SAP becomes applicable in a largely intact fashion. That is, if templates are verified to translate into ground truth traces, then it is likely that codes in the near latent neighborhood of encoded templates are also likely to translate into plausibly realistic traces.

10.3 Experiments in searching stimuli

The potential and eventual limits of sampling – Translating templates into traces may be a straightforward way to produce test stimuli but a much more potent act, following the tradition of `logan`, would be to conditionally sample in the latent space. With the expanded capabilities of `silgan`, instead of resorting to curated traces, the boundaries for conditional sampling can be set much more easily by sketching templates. Right from our first discussions on conditionally sampling stimuli (see Section 9.2), the number of conditions, and the

geometry of the latent subspace that they map out, loom large. When only one condition c is specified, it maps to a single latent point z, providing little clarity on the latent subspace within which conditional sampling should take place. Put otherwise, a single condition is not very effective in communicating the boundaries of sampling. On the other hand, specifying two conditions is a far more clear way to demarcate the boundaries of sampling. Not only is the tester clearly communicating that only traces in-between the given conditions are of interest, but the latent space is also bounded to a straight line. Within this bounded space, the MLERP technique allows much more selectivity and control in sampling stimuli. Now, what if the tester feels that two traces, or a straight line in latent space, is far too narrow and would like to impose, say, three conditions?

Let us, in fact, consider the vehicle speed templates in Figure 10.5 as the triplet of conditions that the tester would like to impose. The first (c_1) is a helpful *null* template where the vehicle is completely still, and the second (c_2) is a takeoff template where the vehicle stays still for half the time before beginning to roll. The third (c_3) is a variation of takeoff where the vehicle stops once before taking off. Allowing all intermediate combinations of these three conditions, the tester is demarcating a larger space for sampling stimuli.

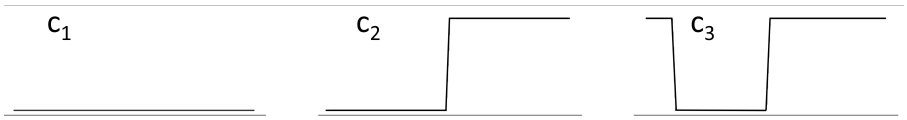


Figure 10.5: Three vehicle speed templates used as conditions for sampling stimuli

This triplet of conditions $c_{(3)} = (c_1, c_2, c_3)$ maps to a triplet of codes $z_{(3)} = (z_1, z_2, z_3)$ in the latent domain Z, meaning that the latent subspace bounded by these conditions can be imagined as a triangle. When using two conditions with `logan`, we saw that linear interpolation, or sampling on the straight line connecting the latent codes of the two conditions, results in a trace that combines characteristics of both conditions. Surely, this property should be extendable beyond just two codes. That is, given the triplet $z_{(3)}$ of latent codes, a convex linear combination of these codes $z = \sum_{i=1}^{3} p_i z_i,\ \sum_{i=1}^{3} p_i = 1, p_i \geq 0$, should achieve semantic combination of the three input conditions. Sampling such codes z and companion codes w from the other latent domain $\mathcal{W}$, and decoding them using the generator $\hat{X} = \mathcal{G}(z, w)$, as seen in Figure 10.6, shows that the property does indeed hold. Generated traces can be seen to consistently combine characteristics of the null, takeoff, and stop-takeoff conditions. Stated otherwise, this means that traces produced by sampling a code in the latent hyper-triangle traced by the three conditions is guaranteed to generate a trace that only combines characteristics of the three specified conditions.

Having progressively elevated the latent subspace for conditional sampling from a single point to a line, and then a triangle, it is time to generalize this progression. The act of specifying a set of K conditions $c_{(K)} = (c_1, c_2, ..., c_K)$ for conditional sampling results in a latent hyperplane with K vertices $z_{(K)} =$

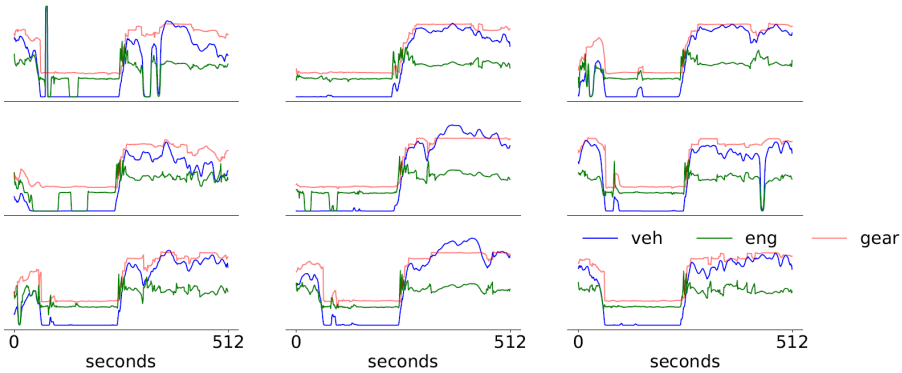


Figure 10.6: Traces generated by simplex sampling in the latent subspace bounded by the three vehicle speed templates

$(z_1, z_2, ..., z_K)$. Sampling codes on this hyperplane, each of which is essentially a convex combination of its vertices, and decoding them produces plausibly realistic traces that proportionally combine semantics of the K specified conditions. Thus, as a tool for systematically sampling on this hyperplane, we have generalized the p-space that we originally defined when discussing MLERP. Sampling points on a latent hyperplane of K vertices is achieved, as shown below, by drawing a simplex $p_{(K)} = (p_1, p_2, ..., p_K)$, $\sum_{i=1}^{K} p_i = 1$, from a Dirichlet distribution of order K and linearly combining it with codes $z_{(k)}$.

$$\begin{aligned} &p_{(K)} \sim Dir(K, \mathbb{1}^K), \;\; z_{(K)} = \big(\mathcal{E}(c_1), \mathcal{E}(c_2), ..., \mathcal{E}(c_K)\big)^T \\ &\hat{X} = \mathcal{G}(p_{(K)} \cdot z_{(K)}, w), \;\; w \sim \mathcal{N}(0, I) \end{aligned} \tag{10.7}$$

Specifying multiple conditions is, without doubt, a powerful technique for demarcating a latent subspace for conditional sampling. Testers can specify as many conditions as necessary to precisely target a subspace that is most interesting for the test objective in question. Further, using the simplex sampling technique in (10.7) it is also possible to systematically explore this subspace, generating a rich, yet targeted, variety of traces that can be applied as test stimuli. But, no matter how smartly conditions have been chosen or how perfectly these spaces are demarcated, if we continue to rely upon sampling as the primary technique for exploring these spaces, we will always need to contend with the issue of infinite possibilities. Even if we use techniques like MLERP to select better samples from this space, sampling will remain a slow an inefficient technique for covering an infinite space. If we use information about the actual test objective, can we not search this space to hone in on interesting stimuli much more directly?

Promoting code to the foreground – When we originally defined the system $\mathcal{S}$ for stimulus generation in (8.1), we made it a point to include the SWC V that is the object of testing. Until now, however, the actual code that is under test has been mostly in the background, and we have been solely discussing stimulus generation from a black box perspective. We now begin to revise this setup, albeit gently, by promoting code under test to the foreground. As we

have previously seen in some detail, vehicle application software is often written in C. We make a temporary exemption to present a toy example of vehicle application software as Python code. The example that we show in Figure 10.7 is a toy version of a monitoring function. This particular function, as its name indicates, monitors if the driver is performing an aggressive takeoff maneuver. It does so by observing vehicle and engine speed signals to check whether the engine largely idles before the vehicle quickly takes off and cruises at increasing speeds. Now, a quick glance at the function is sufficient to know that it is an example that has been contrived to work with traces generated by the GANs that we train. Nevertheless, if we go beyond the form and look at its intent, we will actually find that monitoring functions of this kind are quite common in vehicle application software. Many SWCs include logic that monitors signals, for instance, to identify and log driving maneuvers that are of interest or concern.

```
def detect_aggressive_takeoff(v, e):
 m1 = mean(v[256:350])
 m2 = mean(v[350:512])
 m3 = mean(e[0:256])
 if(0.49 < m1 < 0.51) &
   (0.59 < m2 < 0.61) &
   (m3 < 0.08):
  ret1 = 1
 else:
  ret1 = 0
 return ret1
```

Figure 10.7: A toy example of software under test. This function illustrates a monitoring function for detecting aggressive takeoff events

Using this function to emulate code under test, let us formulate the test objective as code coverage. In this particular case, this would mean identifying appropriate stimuli of vehicle and engine speed traces that satisfy the single branching condition specified in this function. Turning to `silgan` as the source of generated stimuli, let us also reuse the triplet of takeoff scenarios $c_{(3)}$ (Figure 10.5) that we discussed shortly before as the boundaries for generation. Even in the clearly bounded latent hyper-triangle, finding stimuli that satisfy the branching condition by random sampling turns out to be impractical. In experiments trying to achieve code coverage, we were unable to find even one matching trace after sampling around 100k latent codes in this bounded space. Random sampling to generate traces may be suitable for exploratory testing, subjecting application software to a targeted, yet wide, range of realistic stimuli. For a much more focused objective like code coverage, sampling is unlikely to be efficient.

Searching for stimuli with software in the loop – Inefficiency in sampling stems from two main factors, (1) a latent subspace, no matter how small, presents infinite options for sampling, and (2) in a black-box approach, where stimuli are sampled with no awareness of code under test, there is simply no feedback that informs whether sampled traces indeed serve as interesting stimuli. If such feedback were available, then aimless sampling can be substituted by a targeted procedure for *searching* interesting stimuli. We now describe one viable feedback mechanism for the code coverage objective, built using fairly simple

arithmetic. Code coverage entails satisfying branching conditions, each of which is typically defined using a composition of boolean operations. Since these operations evaluate to only `true` or `false`, there is no real-valued indication of how distant a test input is from making an `if` condition evaluate to, say, `true`. One advantage that we have in the monitoring function in Figure 10.7 is that the boolean operations in the branch condition evaluate real values. This, of course, does not encompass the entirety of possible branching conditions, but boolean evaluations on real valued operands are quite common in decision-making logic found in vehicle application software. In such cases, inspired by a method proposed in [130], we smooth discrete boolean evaluations into real-valued measures. Setting aside boolean operators $=$ and $\neq$ which are not directly applicable to checking real values, atomic operations boolean operations $<$ and $>$ on real operands can be readily converted into difference functions shown below.

$$\begin{aligned} \texttt{lt}(a, b) &= a - b \\ \texttt{gt}(a, b) &= b - a \end{aligned} \tag{10.8}$$

When difference operations on real valued operands, defined above, evaluate to a positive value, it is equivalent to their discrete counterparts evaluating to `false`. More usefully, a large positive value indicates that the operands are quite far away from meeting the boolean condition. Such an indication is clearly unavailable in the discrete case. Thus, if a branching condition evaluates $\texttt{lt}(x, 0.49)$ with x being the input stimulus, then $x - 0.49 >> 0$ indicates that x is nowhere near meeting this condition, which is exactly the kind of feedback that can guide the stimulus generation process. Conversely, a negative value guarantees that the condition has been met and is the equivalent of the discrete boolean operation evaluating to `true`. Using this reasoning, a real-valued version of the atomic boolean $\neg$ operator is simply achieved using arithmetic negation.

$$\texttt{not}(c) = -c \tag{10.9}$$

Note that defining `not` as arithmetic negation restricts it to a subset of possible solutions. That is, $\neg(a < b) = a \geq b$, but `not`, as defined above, only finds solutions $a > b$, which is still valid because $=$ does not apply for real operands. Finally, we round off this line of reasoning by addressing operations $\wedge$ and $\vee$, which operate both on real-valued operands and on atomic operations involving them. Here, functions `max` and `min` serve as real-valued counterparts because they respectively check whether all, or any, of the constituent operations to a negative value which, as we noted earlier, is our real-valued equivalent of `true`.

$$\begin{aligned} \texttt{and}(c, d) &= max(c, d) \\ \texttt{or}(c, d) &= min(c, d) \end{aligned} \tag{10.10}$$

Having defined such real-valued counterparts for discrete boolean operations involving real-valued operands, one simple check of consistency is if the newly defined operations satisfy DeMorgan's laws. As shown below, they indeed do.

$$\texttt{not}(\texttt{and}(a,b)) = -max(a,b) = min(-a,-b) = \texttt{or}(\texttt{not}(a),\texttt{not}(b)) \quad (10.11)$$

In transforming boolean operations on real values into another real value, these functions give continuous feedback on how distant a test input is from covering a branching condition on real values. We therefore refer to them as *coverage indicators*. Then, by traversing the abstract syntax tree of the code under test (Figure 10.7), we replace each boolean operation in a branching condition by a coverage indicator. In the case of compound branching conditions involving several boolean operations, we simply `and` individual coverage indicators to form a composite indicator for the branch. All these code-level modifications can be done automatically to transform the function under test into a search function $S : \mathcal{X} \rightarrow \mathbb{R}$ that, given an input trace X, provides a real-valued indication of how distant the trace is from covering the branching condition in the function under test (Figure 10.8). This also means that if a generated trace $\hat{X}$ is applied as input to the search function, and it evaluates to $S(\hat{X}) < 0$, it is guaranteed that the trace $\hat{X}$ satisfies the branching condition defined in the test function. Thus, to find stimuli that achieve the test objective of code coverage, we do not use the code under test directly. Instead, we use its transformed counterpart – the search function – that provides useful feedback to the stimulus generation process.

```
def S(v, e):
  m1 = mean(v[256:350])
  m2 = mean(v[350:512])
  m3 = mean(e[0:256])
  c0 = and(lt(0.49, m1),
     lt(m1, 0.51))
  c1 = and(lt(0.59, m2),
     lt(m2, 0.61))
  c2 = lt(m3, 0.08)
  c = and(and(c0, c1), c2)
  return c
```

Figure 10.8: An automatic transformation of the monitoring function in Figure 10.7 that returns one coverage indicator per branching condition

With the search function S, the stimulus generation process has a concrete target – identifying the trace(s) that makes S evaluate to a negative value. We refer to a trace that satisfies the branching condition as a *hit* $\hat{H}$. Having set the triplet of templates $c_{(3)}$ (Figure 10.5) as the test condition, let us now trace the revised stimulus generation approach, depicted in Figure 10.9. First, a code z is sampled from the latent hyper-triangle demarcated by the three conditions. Second, the generator network $\mathcal{G}$ maps this code (and a sampled w) into a trace $\hat{X}$. Finally, this trace is applied as input to the search function to see if the coverage indicator S evaluates to a negative value. Now, we highlight a crucial property – upon sampling the latent codes, both operations that follow, namely the generation and the coverage indication, are differentiable[2]. This means that the search for a hit $\hat{H} := \hat{X} \mid S(\hat{X}) < 0$, can be done using *gradient descent in the latent space*.

[2] $max(c,d)$ and $min(c,d)$ are not differentiable at $c = d$, but this can be disregarded for real-valued c and d

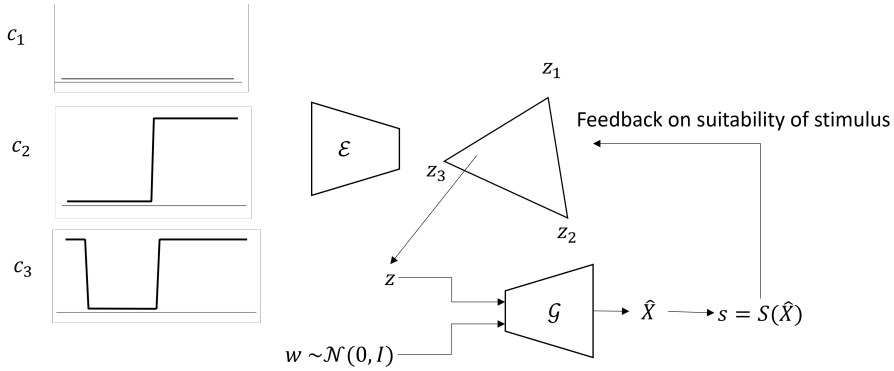


Figure 10.9: Traces generated by gradient descent in the latent subspace bounded by the three vehicle speed templates

While the revised sampling operation, including the search function, may be differentiable, the end-to-end mapping between the latent code and the coverage indicator is likely to feature many local minima. In fact, a brief exploration of this landscape using simplex sampling, seen in Figure 10.10 which visualizes the value of S by location in the latent triangle, confirms this. The figure shows the presence of large flat regions, where S hardly changes. In these areas, where the gradient of S with respect to the position of the latent code is virtually zero, search by gradient descent cannot take place. To address this issue, we propose a method that combines sampling and gradient-descent to search for test stimuli that match the test objective specified by S.

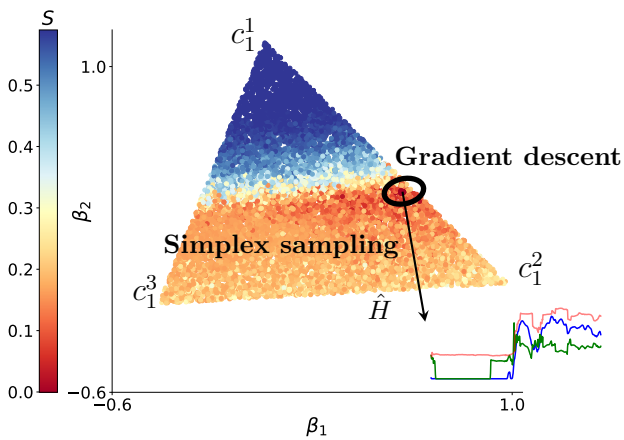


Figure 10.10: With test conditions $c_{(3)}$, automatic code coverage using Algorithm 3 begins with simplex sampling in the triangle with vertices $z_{(3)}$ in the latent space $\mathcal{Z}$. Once a promising area is found, gradient-descent takes over to find $\mathcal{H}$ that satisfies the condition. For visualization in 2-d, we plot the triangle by taking steps β_1, β_2 along two randomly chosen basis vectors that span its plane. A similar search is jointly conducted in C_2, but it is not visualized here.

Algorithm 3: GRADES: Gradient descent-based search for test stimuli

Input : Template conditions $c_{(K)}$, search function S that encodes the test objective
Output : Generated trace $\hat{H}$ satisfying the test objective
Parameters : n_{sim} - max sampling steps, s_{sim} - exit threshold for sampling, n_{gd} - max gradient descent steps, η - gradient step size

$z_{(K)} = \big(\mathcal{E}(c_1), ..., \mathcal{E}(c_K)\big)^T$
\# the sampling stage
for $n \in [n_{sim}]$ **do**
  $p_{(K)} \sim Dir(K, \mathbb{1}^K)$, $w \sim \mathcal{N}(0, I)$
  $\hat{X} = \mathcal{G}(p_{(K)} \cdot z_{(K)}, w)$
  if $S(\hat{X}) < 0$ **then**
    $\hat{H} = \hat{X}$
    return $\hat{H}$
  else if $S(\hat{X}) < s_{sim}$ **then**
    `break`
  end
end
\# the search stage
$q_{(K)} = \texttt{logit}(p_{(K)})$
for $n \in [n_{gd}]$ **do**
  $p_{(K)} = \texttt{sigmoid}(q_{(K)})$
  $p_{(K)} = p_{(K)} / \sum_{i=1}^{K}(p_i)$
  $\hat{X} = \mathcal{G}(p_{(K)} \cdot z_{(K)}, w)$
  if $S(\hat{X}) < 0$ **then**
    $\hat{H} = \hat{X}$
    return $\hat{H}$
  $q_{(K)} = q_{(K)} - \eta \nabla_{q_{(K)}}(S)(\mathcal{H})$
  $w = w - \eta \nabla_w(S)(\mathcal{H})$
end

GRADES (Algorithm 3), the method we develop for stimulus search, takes two inputs – the tuple of conditions $c_{(K)}$ and the search function S, which is the transformed formed of the code under test V that calculates the coverage indicator. The output of the process would be a hit $\hat{H}$, which is a trace that is guaranteed to cover the branch condition in code under test. Practically, this means that the real objective of the algorithm is to find the simplex $p_{(K)}$ – the location in the latent hyperplane – where the latent code decodes to $\hat{H}$. Of course, the procedure should also search for the companion code w in the second latent domain $\mathcal{W}$, which is necessary to complete the decoding step. The search begins with simplex sampling for a maximum of n_sim iterations in order to survey the coverage landscape in the latent hyperplane. If the sampling process chances upon $\hat{H}$, the objective is achieved, ending the search. Otherwise, the algorithm identifies a promising area, i.e. values of $p_{(K)}$ and w that generates a maneuver resulting in $S \approx 0$, around which finding $\hat{H}$ is likely. This is where the threshold s_{sim} helps where, if $S(\hat{X}) < s_{sim}$, the sampling stage exits.

The search is then taken over by gradient descent, which attempts to iteratively minimize S by jointly varying $p_{(K)}$ and w along directions of the fastest decrease in S. Now, while doing gradient descent in latent space, special care needs to be taken to ensure that the search does not stray beyond the bounds of the latent hyperplane. During the sampling stage, drawing from the Dirichlet distribution guarantees that the simplex $p_{(K)}$ sums up to 1, which means that $p_{(K)} \cdot z_{(K)}$ is guaranteed to fall within the bounded hyperplane. If gradient descent is able to modify $p_{(K)}$ unchecked, then there is every chance that the boundaries of the hyperplane are violated. This is why we reparameterize $p_{(K)}$ using the `logit` and `sigmoid` pair of operations. The former preserves the most promising latent code identified by the sampling stage, and the latter ensures that the subsequent gradient descent stays within boundaries, even when the code is iteratively updated. This search persists for a maximum of n_gd steps during which, if S becomes negative, $\hat{H}$ has been found and the search is complete. Otherwise, the search times out without finding $\hat{H}$ under the given test scenario.

Figure 10.10 shows the case of simplex sampling being unable to find $\hat{H}$ even after setting $n_{sim} \approx 100k$. Combining the respective strengths of sampling and gradient-descent, both of which use feedback from coverage indicators, it is often sufficient to sample for 10–50 iterations, with the subsequent gradient search taking no more than a few 10s of iterations to find $\hat{H}$. Further, the example monitoring function in Figure 10.7 may contain a single branching condition, but the method can be extended to multiple independent *if-else* conditions, even when they are nested. A search function, in this case, returns a vector containing one minimizable coverage indicator per independent branching condition. Due to their independence, this vector can be collectively minimized by executing one search per coverage indicator in parallel.

Even if it is in a transformed state, connecting code under test in a feedback loop with a GAN is where `silgan` truly shows its potential. With the availability of the search function that provides differentiable feedback, the use of gradient descent makes finding the right stimuli orders of magnitude quicker. The search paradigm also opens up possibilities where, for instance, testers are much more free in demarcating the limits of testing. No matter the size if the test space, a search technique that combines sampling and gradient descent is clearly more efficient than sampling alone. In demonstrating a search-based approach for stimulus generation, we however discuss only a case of code coverage with boolean operations made differentiable using coverage indicators. While the principle of targeted, combined sampling and gradient based search of the latent space is vital, additional measures may be necessary to scale the technique for automatically testing code with non-differentiable intermediate operations.

Overall, building upon `logan` which we saw in the previous chapter, `silgan` introduces three main enhancements. First, in defining templates as a mechanism for specifying conditions, the process of stimulus generation scales with an expansion in the number of signals. Second, the definition of a framework to define and operate with multiple conditions gives testers the freedom to define the test space with arbitrary complexity. Third, by developing a mechanism for providing a differentiable assessment of generated test stimuli, we are able

to connect software in the loop with a GAN. Put together, the `silgan` toolkit significantly improves the potential of simulation-based testing in virtual rigs.

Chapter 11

Discussions

11.1 On research questions

We now discuss how the test stimulus generation systems described in previous chapters measure up to the research questions originally posed in Section 8.2.

RQ1: on stimulus generation using GANs – Experiments in sampling and searching stimuli, described in previous chapters, provide compelling evidence that deep generative models trained on signal traces can indeed be used to generate test stimuli. Beginning with `logan`, a good first indication of the ability to conditionally generate useful stimuli comes from the use of naive linear interpolation in the latent space between two curated stop traces to generate a novel intermediate trace. In trivially generating a realistic trace with a desired profile, latent space interpolation presents itself as a tool for stimulus generation that can be powerfully employed for testing. Then comes the MLERP technique which introduces useful refinements in latent space interpolation, helping exercise controllability and selectivity in generating stimuli. Allowing testers to quantitatively examine traces using an interpretable metric measure, MLERP guarantees that generation produces stimuli that follow a sequence of metric measures that testers specify. The `silgan` system further expands the toolkit for stimulus generation by first introducing the notion of templates that vastly ease the specification of test conditions. Largely avoiding the use of curated traces, templates allow testers to handcraft an arbitrary number of test conditions in the form of simple sketches of signal profiles. Mapping all the conditions to the latent space, simplex sampling on the hyperplane bounded by conditions guarantees that generated stimuli stay within the boundaries of test specification. Finally, the mechanism we introduce for extracting differentiable feedback from code under test means that test stimuli can be efficiently searched, and not only sampled, within bounded latent spaces using the GRADES technique. At the core of this steady progression of techniques lies a GAN trained on a few hundred thousand recorded signal traces. The training itself uses variations of the adversarial objective (9.2) that pits a pair of generator and discriminator networks against

each other in a zero-sum game. Thanks to our strategy of conditional sampling using vector arithmetic in latent space, adversarial training alone seems sufficient for generating test stimuli. The roster of models, tools, and methods that we demonstrate surely amounts to a definitive answer that GANs trained on signal traces do serve as a useful framework for test stimulus generation.

RQ2: on plausible realism of generated stimuli – Perhaps the most pressing issue encountered whenever generative models are used is whether generated samples are plausibly realistic. The utmost importance that `logan` and `silgan` systems pay to this issue is first revealed, perhaps innocuously, by the fact that they mainly promote the idea of conditional sampling. In this sampling regime, the act of generating any test stimulus $\hat{X}$ actually begins with the specification of a condition c. Of course, a practical reason why conditional generation is essential is that testers need to be able to clearly specify expectations on the eventual nature of generated stimuli. After all, a random mass of traces, generated without clear expectations, may not really serve as valuable test stimuli. Since conditions are anyway necessary to usher the generation process to satisfy the overall test objective, we repurpose them for checking the quality of generation itself. This is the very essence of the SAP principle that we use for measuring the quality of generation. If the condition verifiably maps into a plausible trace, the SAP reasoning goes, then traces in the near latent neighborhood of the condition should also map to plausible traces. This approach, which is arguably conservative, takes no position on the quality of an unconditionally generated trace. Rather, it ties the quality of generation firmly with the condition itself. In the case of `logan`, the application of SAP is quite straightforward since the condition c is itself a trace X. This provides a ready-made mechanism to measure the quality of generation – the similarity $M(X, \hat{X})$ between the condition X and its reconstruction $\hat{X}$. If the reconstruction is verifiably similar to the condition, then it is plausible that codes in the near latent neighborhood of the condition also map to good quality traces. This naturally leads to the next challenge – successfully demarcating the 'near' neighborhood. The approach we take for demarcation is to use multiple conditions to bound latent subspaces within which sampling can take place. If each condition reconstructs well, then the hyperplane that they trace in the latent space may be considered as being safe for sampling. Should this reasoning not suffice, there is always MLERP which provides an additional layer of scrutiny. Given a suitable metric M, MLERP selects only those codes in the latent subspace which map to traces with a specified metric similarity. Yet again, a mechanism for selective sampling doubles up as a mechanism for verifying plausible realism.

When upgrading to `silgan`, as we pointed out previously, the SAP principle does not necessarily extend smoothly. Multimodal translation may be a key strength of `silgan`, but the lack of a clear ground truth that it entails means that SAP is not directly applicable. An option would be to curb multimodality and force the model to faithfully translate templates to the actual traces they were extracted from. This restores the relevance of a ground truth for translation, and SAP becomes applicable, albeit with a loss of diversity in translation. It is also important to note that we have discussed SAP only in the sampling context. Since `silgan` additionally uses the GRADES procedure for gradient

descent-based search in the latent space, we also need to consider the plausible realism of traces generated by searching. One strong baseline is that the search is being conducted within a bounded latent space. And, since this bounded space is still the near neighborhood of conditions, SAP remains applicable. The larger issue is that, unlike MLERP, the search process is less controlled and can chance upon a latent code which decodes to a trace that satisfies the test objective, but may not be plausibly realistic. In letter, the SAP principle can be folded in as an additional objective during search, meaning that the search can be tasked to find a trace that simultaneously satisfies the test objective while also meeting some metric criteria. MLERP, after all, achieves this combination in a sampling paradigm. There may, however, be practical issues in jointly meeting both these conditions, increasing the likelihood that the search process takes a longer amount of time to find a hit. Considering these concerns, identifying an efficient and firm extension of SAP to `silgan` sampling and searching regimes is an important avenue for future work. Overall, however, the strategy of conditional sampling and searching, combined with SAP and MLERP, ensures that stimulus generation is selective and that generated stimuli can be quantitatively assessed for plausible realism by comparing them with references.

RQ3: controlling stimulus generation to meet test objectives – The previous discussion on evaluating the quality of generated traces throws up an interesting observation. Controlled (or selective) generation and evaluating the quality of generation are, in fact, two sides of the same coin. The way we use conditional sampling captures this duality best – generation is limited to a well-demarcated space so that techniques of evaluation are easier to apply. In some sense, the techniques that we introduce, starting from bounding the latent space using conditions, to selective sampling, and searching, progressively make it easier for testers to narrow down choices. Facing infinite possibilities in stimulus generation, a strategy of narrowing down choices actually ends up improving selectivity. Since we have already detailed mechanisms for selectivity when discussing the previous research question, let us turn to the other important aspect of the current research question, which is directing the selection of traces to meet a particular test objective. When it comes to conditional sampling, it is abundantly clear that the onus to direct stimulus generation is on the tester. In the black box `logan` system, testers select conditions which bound the test space, and, when using MLERP, also specify metric criteria which generated traces should meet. This general recipe is quite extendable to specifying a wide range of testing intentions. Testers may demarcate a space to broadly explore whether the behavior of software under test is generally consistent within its boundaries. Alternatively, if testers want to probe a specific aspect of software behavior, the same set of tools can be reused to set a tighter test space. The use of standard drive cycles (like WLTP) as conditions is an extreme example of the latter category where testers are specifically interested in, say, fuel consumption during one standard drive cycle. Either way, in the black box sampling approach, it is the tester's experience and expertise that guides the process of generating stimuli. An interesting sidebar would be to deliberate whether such dependency on the tester is advisable. In fact, when describing the research approach taken in this work (see Section 1.3), we specifically addressed this aspect. Software engineering activities involve humans in the loop, and needs sufficient margin

for decision-making. Taken together, it becomes quite clear that depending upon the tester's agency is not only well-advised, but also essential. No matter the suite of tools used, it is a human engineer who ultimately signs off on a testing activity. If this engineer has not been sufficiently involved, it only complicates the assessment of the overall testing process. Not only do the `logan` and `silgan` systems require a tester's active involvement, but they also take active measures to ensure that the burden of involvement is minimized. May it be conditions, templates, or MLERP, each instrument that we develop for selective generation reduces the tester's burden in some way and, in turn, encourages their involvement. Nevertheless, if there is a need to reduce the tester's involvement in directing the sampling process, combining `silgan`, with the gradient descent-based GRADES search technique, shows that there is scope for further automation. The search process serves to showcase that given enough information about the software and the test objective, the stimulus generation process can be highly targeted. Putting all this together, we can reasonably conclude that the spectrum of tools we introduce for stimulus generation can be quite effective in selecting the right stimuli for the given objective.

RQ4: refining stimulus generation using code under test – Experiments and discussions on stimulus generation may have been dominated by tools for black box sampling, but Section 10.3 shows that code itself can be repurposed to search for the right stimuli. For an objective like code coverage, we have seen how simple functions can be automatically converted into coverage indicators in order to provide real-valued feedback to guide stimulus generation. The linchpin in the process is the fact that the generator DNN is differentiable. The differentiable nature of a DNN is, of course, fundamental to deep learning itself. Descending in parameter space using the gradient of the loss function is the very essence of neural network training, using which the backpropagation process updates learnable parameters of the network. Unlike training, the GRADES gradient descent-based search that we conduct is not in the parameter space, but in the *input* space of the generator network. However, any optimizer designed for gradient based optimization in parameter space is reusable for searching in the input, which in this case, is the latent space. While the high dimensional parameter space, to a certain extent, is well-designed for gradient based optimization, the search space for stimuli is not. This is why we additionally include a sampling step to survey the search landscape to find areas where there are useful gradients before initiating the search. Using coverage indicators and combining the respective strengths of sampling and searching, we thus show that code under test can be used to search for stimuli that target a specific test objective.

11.2 On techniques employed

Choice of convolutional networks – In both `logan` and `silgan` systems, each constituent network has been designed as a Convolutional Neural Network (CNN). Considering that the networks essentially process time series signal traces, the choice of CNNs may seem odd. After all, CNNs are popularly considered to be much more appropriate for tasks in computer vision. For processing

sequential input, recurrent models like the Long Short-Term Memory (LSTM) or Gated Recurrent Units (GRUs) are meant to be more appropriate. Reality, however, belies such popular belief and, as surveyed in [131], CNNs have been applied for time-series problems in equal measure. While the memory of the recurrent networks is often considered to be better suited for modeling long term dependencies, given enough capacity, CNNs have often proven adept even when working with sequences. Moreover, CNNs, unlike recurrent networks, are parallelizable, which has actually made them quite efficient for sequence tasks. The clinching reasoning for choosing CNNs, however, lies in our construction of the stimulus generation problem. Traces that we generate as stimuli are of fixed duration, meaning that they are much more like images. This means that there is no real necessity to treat them as sequences of indefinite length, and `logan` or `silgan` networks can be built up as CNNs. However, whether 1-D CNN is the most optimal architecture for designing GANs for signal traces is, to a certain extent, an open question. It is also a question that is difficult to definitively answer because, simply put, there are far too many choices involved in modeling the network architecture. Even within the CNN family, there are specializations like dilated [132], causal [133], or temporal (which combines the two) convolution which have been proposed for sequence processing. Searching for the optimal network by sweeping all these architectural and subsequent hyperparameter choices takes an inordinate amount of time and resources. Facing such challenges, there are but two main options. The first one is to turn to techniques of neural architecture search [134], where the search of a DNN architecture can be modeled as a systematic search or even as an optimization problem. This could automate modeling steps that are currently manual, but the material costs involved in such search is not trivial. Practical compromises in conducing architecture search usually entail restricting the search space, which could lead to suboptimal solutions. The other option, trends of which are beginning to emerge in deep learning practice, is to rely upon a smaller set of widely used network architectures. Taking this cue, an interesting future extension would be to base stimulus generators on the Transformer architecture [80]. Originally proposed as an efficient parallel processing paradigm for univariate sequence data (text), Transformers have come to dominate DNN modeling across data modalities including time series [135]. In similarly basing signal trace generators on this well reused architecture, costs involved in making architectural choices can possibly be minimized further. Training `logan` and `silgan` using a set of Transformer networks would be an interesting avenue for future work.

Structuring the latent space – Previous chapters, describing experiments in searching and sampling stimuli, make the pivotal role of latent space operations abundantly clear. We may have used conditions to demarcate the subspace within which sampling or searching is conducted, but little effort has been spent in consciously structuring this space. To illustrate this point, let us temporarily set aside the stimulus generation application and examine StyleGAN [110], the face generating GAN model which we briefly saw earlier. One of the most important properties of this network is the seamless manipulation of specific facial characteristics. StyleGAN, stated simply, has such a structured latent space that the facial semantics it encodes are highly disentangled. The resultant effect is that changing the latent code along specific dimensions leads to changes

in specific facial characteristics, while others stay largely intact. Coming back to `logan`, while its auto-encoder may create a latent space of reduced dimension, it is not clear whether latent codes are sufficiently disentangled to finely manipulate specific characteristics of the trace. It would be beneficial if, say, changing the third dimension of the latent code results in traces with rapidly increasing vehicle speed. The `silgan` system, on the other hand, does actively disentangle the latent space. As seen in Figure 10.3, the template or the profile of one signal, maps into the latent domain $\mathcal{Z}$. The source of all other information in generating the trace is the latent domain $\mathcal{W}$. While this disentanglement does ease multimodal translation, it falls well short of the kind of disentangled manipulation seen in StyleGAN. One advantage that StyleGAN has over our approach is that it relies on a dataset that has been annotated with great detail. Annotated variables largely map into the latent space and serve as control knobs for fine manipulation. Attempts to similarly annotate traces may not go down well with testers initially but, if benefits become apparent, dedicated efforts for annotation may very well commence. An interesting track for future investigation would be to develop methods for latent space engineering which introduces disentanglement without excessively relying upon labels. If the `silgan` approach is any indication, there seem to be ways to achieve this.

Going beyond raw code for stimulus search – A well-engineered latent space, as we saw in the `silgan` system, is conducive for gradient descent-based search for test stimuli, as long as the right kind of feedback is available. In the code coverage example that we showcase in Section 10.3, we were able to provide such feedback by automatically converting branching conditions in code under test into coverage indicators. The differentiable real-valued feedback provided by these indicators is helpful in guiding the process for searching stimuli. However, when code under test is more elaborate than the example that we work with, or when the test objective goes beyond code coverage, alternative mechanisms are needed to provide appropriate feedback. More generally, the task of stimulus search can be seen as identifying the correspondence between the code and stimulus spaces. In the case of `logan` and `silgan`, using techniques of deep generative modeling, we have DNNs that learn a representation space for stimuli. For the test coverage objective, specifically, by converting branching conditions into coverage indicators we were able to assemble a parallel representation space for code. Then, using gradient descent-based search, correspondence between stimulus and code spaces is identified, achieving the task of stimulus search for code coverage. Alternatively, as we saw in Part I, techniques of neural language modeling offer a far more general approach to learn a representation space for code. The `tasnet` model, which converts raw code into program embeddings, is a perfect example. Can we not, therefore, learn or construct a mechanism that associate representation of stimuli (learned by GANs) and those of code (learned by language models)? Such a multimodal approach, which jointly considers code and stimuli, is surely closer to how engineers approach software testing in practice. More importantly, relying completely upon learned representations of stimulus and code spaces eliminates the need to construct specialized representations like coverage indicators, allowing stimulus search to scale beyond specific test objectives like code coverage. Combining language models of code and generative models of stimuli to aid software

testing is yet another tantalizing prospect that we leave for future work.

Generative models beyond GANs – The core technique that underpins stimulus generation using `logan` and `silgan` systems is the adversarial learning objective (9.2) originally introduced in [109]. When it was first introduced, GANs revolutionized generative AI and was quickly recognized as a breakthrough technique [136]. Deep learning techniques, as we have repeatedly seen, are rarely in stasis, and it should come as no surprise that vastly improved techniques for generative modeling have risen. As surveyed in [137], the original GAN has itself has seen dozens of variations in data modalities, training objectives, and network composition. In fact, `silgan` itself adopts elements of multimodal translation from an image-to-image translation GAN [128] in addition to the least-squares discriminator introduced in [127]. Apart from variations, there are many other generative modeling alternatives to GAN, an example of which would be the VAE [138], elements of which we borrowed in constructing `logan`. Apart from the GAN and VAE frameworks, as further surveyed in [139], normalizing flows, energy-based, and autoregressive modeling constitute further alternatives. A dramatic contemporary example for generative modeling would be the DALL-E [140] family of text-based image generation models trained, among other objectives, using the energy-based technique of denoising diffusion [141]. Displaying remarkable ability in translating natural language text prompts into high quality images, DALL-E, along with ChatGPT, has fast become the byword for the capabilities of modern AI in popular imagination. The point of this discussion may be to elucidate that there are numerous alternatives to GANs, but the larger question, of course, is their implication on generating signal traces as test stimuli. The `logan` and `silgan` systems use a combination of GAN and VAE techniques but, considering the phenomenal successes of the more recent diffusion models, an interesting experiment would be to see if newer techniques can be applied to stimulus generation. In undertaking such extension, one important aspect to consider would be the strategy of conditional sampling in the latent space, which our approach of stimulus generation crucially depends upon. This strategy imposes an encoder-decoder architectural style which, in turn, exposes a latent space that can be manipulated. Stated otherwise, our strategy fundamentally bases itself upon so-called latent space models, and using alternative generative techniques need to factor this. An encouraging observation is that most approaches, including diffusion and autoregression, do offer mechanisms to model and manipulate a latent space [142]. Another interesting possibility, thanks to the emerging interest in dialog systems or chatbots, would be the use of text prompts instead of traces or templates as conditions. Apart from opening up alternative proposals of expression, text prompting moves conditioning logic from a latent space to the input space. This could help avoid the dependency on latent space models for stimulus generation. On the other hand, whether the signal trace can be effectively substituted by a natural (or formal) language description is an open question. We leave all these interesting considerations for future work.

11.3 Related work

Analogous to the `logan` approach, GANs, following its introduction, have been applied for synthetic data generation in a variety of domains. Reflecting our scope, we focus on surveying reports of continuous-valued sequence generation and highlighting the following aspects – (i) the model architecture and (ii) measures to check plausibility of generated samples. Using a recurrent GAN, generation of real-valued medical time-series was demonstrated by [143], where two plausibility measures were shown. One is measuring Maximum Mean Discrepancy (MMD) [144] between populations of real and synthetic samples, and the other to train a separate model using synthetic/real data and evaluate it using real/synthetic data. Using a recurrent architecture, [145] demonstrated music generation, with generated samples evaluated using observable features such as polyphony and repetitions. From an application perspective, [146] comes close, where a recurrent GAN was applied to generate time series of automotive perception sensors, for simulation-based verification. Plausibility is shown using the Jensen-Shannon (JS) divergence, as a measure of symmetric relative entropy between populations of real and generated samples. A combined Long-Short Term Memory (LSTM) and Mixture Density Network (MDN) GAN for generating sensor data has been shown by [147], where GAN loss and discriminator prediction are used as evaluation measures. Convolutional GANs have been applied for sequence generation in [148] and [149]. Both of them apply MMD and Wasserstein-1 [150], while the latter additionally applies classical machine learning methods like k-means clustering to measure plausibility. Beyond deep generative models, purely statistical methods of generating and evaluating time series (examples [151] and [152]) have been reported. However, such methods are, in the existing literature, inherently dependent on features selected by a domain expert. In comparison, the GAN approach provides the capability to learn the necessary features automatically, thus being able to optimally adapt to the dataset on which it is applied. Then, analogous to the *template→trace* domain adaptation/translation approach taken by `silgan`, previous work has employed domain adaptation to extract physiologically invariant features in a clinical setting [153] and the creation of a dataset invariant to specific characteristics of individual blast furnaces [154]. While parallels to the `silgan` translation approach are rare in the domain of time series, certain aspects have analogies in the image domain. Examples include sketch to image [128, 129, 155], and image outpainting [156].

Previous work on deep learning for test case generation includes text generation for testing mobile apps [157], and protocol frame generation for testing process control equipment [158]. Parallel work on dynamic software testing, includes identifying worst case branching for stress testing [159] and GUI testing for apps [160]. Unlike these methods which target specific aspects of testing in different domains, by introducing techniques for specification, generation, and test automation, we demonstrate an end-to-end framework for virtually testing vehicle application software. Finally, while we show a simple example of using GANs for stimulus search, in extending our work to more general cases, previous work on ways to smooth boolean conditions [130, 161] may be helpful.

11.4 Congruence with research objectives

The `logan` and `silgan` systems, with associated methods for sampling and searching for test stimuli take significant strides towards addressing the second research objective (see Section 1.2) of easing the process of virtually testing vehicle application software. First, in choosing a system of signals that capture the essential behavior of the powertrain of the vehicle, we train GAN systems that simulate this behavior. Considering the importance of the powertrain in the larger scheme of automatic control, the fact that these systems simulate the behavior of such a critical dependency makes it useful for testing a significant proportion of vehicle application software. Second, the stimulus generation systems that we develop are largely agnostic to the set of chosen signals. This means that training GANs that simulate other dependencies can be straightforwardly accomplished by choosing a new set of signals. Third, the methods for conditional sampling and searching that we introduce allow fine-grained control of the stimulus generation process. This means that human engineers retain firm control of the testing process, while the GAN systems reduce their burden in specifying or even handcrafting test stimuli in detail. Fourth, the fairly strict adherence to the SAP principle means that testers are able to verify the plausible realism of generated stimuli using metrics of their choice. Finally, we reason that tools and methods for stimulus generation that we develop can extend into testing software in non-automotive or even non-embedded domains. As long as software can be tested using multivariate time series, the GAN systems we develop remain applicable. With sufficient modification, extension into data modalities other than time series also remains possible.

11.5 Congruence with the solution approach

On the big data, deep learning approach – Beginning with Part A of the research approach charted in Section 1.3, let us analyze how the stimulus generation systems that we develop measure up against the big data, deep learning approach that it declares. With the core objective being the generation of fake, yet realistic, signal traces that can act as test stimuli, the choice of a deep learning technique like GANs is well justified. After all, deep generative modeling has a good, if not unparalleled, track record in producing realistic fakes. As we ourselves have noted in previous discussions, deep generative techniques have improved to such an extent that the generation of synthetic data across modalities is very much a reality. When it comes to the big data perspective, however, it is fair to ask whether a training dataset of a few hundred thousand samples is truly large. The relative paucity of data is only emphasized when compared to the hundreds of millions of samples used for training `tasnet`. First, it is important to state that the smaller training dataset used for training `logan` and `silgan` does not necessarily reflect a general lack of vehicle signal data. With millions of connected vehicles plying the streets of the world, there is ample potential to assemble truly large signal datasets. Nevertheless, it is also important to recognize that datasets of natural language, programming language, pictures, video,

etc., remain significantly larger and will likely maintain an edge in abundance in the near future. There is, quite simply, much more of these kinds of data being exchanged on global communication networks than vehicle signal data. These observations, combined with the fact that the tech industry far outstrips the automotive industry in promoting the creation and curation of data, make the relative smallness of signal data more conspicuous. This chain of reasoning only reinforces the urgency with which the automotive industry needs to treat data as a valuable asset. If the use of a training dataset with a few hundred thousand samples is able to produce encouraging results, the use of hundreds of millions of samples – as seen in parallel application domains – will only improve them.

On the use of foundation models – If we refer back to `tasnet`, it is clear that this programming language model aligns perfectly with the generally accepted idea of a foundation model. A transformer encoder with millions of parameters, trained on a large corpus using the cloze objective checks all the boxes of the archetype foundation model. Do `logan` and `silgan`, with their hundreds of thousands of parameters and training samples, qualify as foundation models? We reason that they actually do. If we refer back to the seminal discussion on foundation models [28], the only firm definition it ventures is that these models are trained self-supervised. The adversarial objectives (9.4 and 10.6) used to train `logan` and `silgan`, are stellar examples of self-supervised training. Even if we go beyond the letter of the definition, the larger purpose of foundation models is to be generalists, capable of being adapted to multiple tasks in a domain. The fact that we pretty much use the same model to sample and search for test stimuli amply indicates its versatility, burnishing its credentials as a foundation model. In fact, one of the earliest applications of GANs [58] showed that, after adversarial training, the discriminator can be repurposed as a classifier. Last, but certainly not least, [28] itself identifies GANs as a viable family of foundation models, especially in the vision domain. We, of course, repurpose the technique to train foundation models of signal traces and use it downstream as a stimulus generator.

On pre-train and calculate – Not only do `logan` and `silgan` qualify as foundation models for signal traces, but the MLERP and GRADES algorithms show that they can be readily repurposed for stimulus generation using rule-based vector operations. The pairing of `logan` and MLERP show that conditional sampling, with selectivity and measurable plausibility, is achievable using trivial interpolation in the latent space. The fact that MLERP is based upon the well-observed property of latent interpolation for semantic combination makes its construction principled. Parallelly, the combination of `silgan` with GRADES show that stimuli for specific test objectives can be found by a simple procedure for searching in bounded latent subspaces. Using the property that the generator in `silgan` is differentiable, and a specially designed differentiable technique for coverage indication, GRADES shows that gradient descent-based search can automatically find appropriate stimuli. Put simply, both these pairings serve as excellent examples of 'pre-train and calculate', the paradigm for prediction that we set forth in Section 1.3 for solving tasks in automotive software engineering. Moreover, two crucial aspects of predictions in this paradigm – nuance and relative transparency – are also clearly visible. In either GAN framework, testers have the flexibility to finely adjust the test space, allowing them to

use their experience and design nuanced test conditions based upon the code under test. Here, `silgan`, with the introduction of templates significantly eases specification of testing conditions. Other tools which testers can use to tweak the process include the choice of the metric measure in MLERP and the coverage indicator in GRADES. Finally, the rule-based MLERP and GRADES algorithms, based upon principled operations, are patently transparent. An interesting question to consider at this point is whether stimulus generation can in fact be solved using the stereotypical paradigm of explicit supervision. Whether test stimulus generation lends itself to a regime of supervised task specialization hinges upon the nature of the test objective. Given some code, if the objective is to explore its behavior under a wide range of scenarios, testers are better served by a general stimulus generator that is untethered to code. This way, testing is more directed by the tester's own understanding of expected behavior and the GAN systems that we train are perfectly suited for this purpose. There are, however, cases where the test process is much more code dependent. The case of code coverage, which we saw in the context of stimulus search, is a good example. With code coverage, since there is a much more direct mapping between the code and a set of stimuli that satisfies branching conditions, one can contemplate a supervised regime. One could possibly curate pairs of code and corresponding stimuli that is guaranteed to cover branching conditions in this code. Using this paired dataset, a model can be trained to take code as input and predict a set of stimuli that satisfy coverage. However, it does not take much effort to note that the cost of such curation is enormous. Attempting to pair code with stimuli, not only do we impose the need for curation, but we are also asking engineers to match curated traces with code. Short of code-stimuli pairing with enormous effort, it is nevertheless interesting to explore the possibility of directly mapping code and stimulus spaces. As we noted shortly before, the `tasnet` model does learn to represent the space of code while the GAN models do this for the space of stimuli. Rather than embarking upon a supervised mapping of code to stimuli, as we previously noted, connecting latent spaces of code and stimulus representations in a preferably unsupervised manner, is an interesting avenue for future work.

On evaluating the quality of test stimulus generation – Evaluating the quality of generative models is always challenging and the evaluation of `logan` and `silgan` models, and the associated MLERP and GRADES algorithms, is no exception. Before a recap of how we address evaluation, however, it is important to clarify some context. Under current state of practice in the automotive industry, as traced in Chapter 7, techniques for generating test stimuli are severely limited. At one level, there are cases where multi-physics simulation models are themselves used for generating stimuli. While this is positive, physics-based simulation models bring their own set of limitations. We have already noted that developing such models requires tremendous amount of time and specialist expertise. In many cases, once developed, they are also difficult to extend. A typical example would be incorporating weather conditions in driveline or vehicle dynamics models. The physics-centric approach in developing these models often clashes with the statistical approach in modeling weather, slowing down their integration. At another – far more serious level – is the problem that a vast proportion of test stimuli is hand-crafted. While stimuli may be specified

using programming or scripting languages, the fact remains that testers are often hand-crafting signal transitions for testing software in virtual rigs. Being a slow and laborious process, hand-crafting increases the likelihood of not covering scenarios, while the inevitable atrophy means that tests do not keep up with changes in functionality. There is, therefore, a clear need for more proficient methods of stimulus generation, and the capabilities of `logan` and `silgan` cater to this clear demand. Having said that, it is equally important to ensure that stimuli generated by these GANs are of good quality. As traced through Chapters 9 and 10, our approach for measuring the quality of generation is similarity as plausibility. We briefly re-examine the SAP approach here.

1. Under SAP, generation using GANs is always conditional, where conditions are themselves recorded (or ground-truth) traces.
2. New traces are generated by sampling (or searching) in latent subspaces demarcated by the conditions. This makes generated traces follow the characteristics of conditions.
3. When sampling, we only choose generated traces that are verified to have proportionally similar characteristics to ground truth conditions.
4. As long as generated traces are verified to follow these similarity constraints, we reason that the generated traces are plausibly realistic.

Put simply, the conservative SAP strategy undertakes trace *improvisation* rather than outright novel trace generation. Generated traces always share characteristics of recorded traces, and they are verified to do so. Such a strategy of only generating traces that do not stray too far away from the characteristics of recorded traces is clearly limited. But, this is the trade-off that we make to ensure that the generated traces are plausibly realistic. The SAP approach therefore takes a relative approach for evaluating quality or realism, by always comparing generated traces to semantically close recorded (or ground-truth) traces. Ideally, however, there needs to be an absolute measure for evaluating the quality of generation. Limits to the relative approach are apparent - it does not provide a direct mechanism to check whether a generated trace generalizes well-beyond the distribution of recorded traces it is trained upon. Any dataset of recorded datasets will have gaps in coverage. Due to the considerable costs of recording campaigns, it will be impossible to record every vehicle variant in all possible operating conditions. It is only when a model is shown to reasonably generalize beyond gaps in recording do we have true simulation capabilities. One option for absolute evaluation would be to test the properties of generated traces using a physics-based simulator. Such an approach would need to find ways to reconcile limitations like the physics simulator not modeling all phenomena represented in the trace. Another interesting avenue would be to make the GAN aware of the underlying physics. Recent years have seen substantial progress in making neural networks physics aware [162]. Using these techniques to make `logan` and `silgan` aware of the physics of in-vehicle dependencies could be useful for improving absolute quality. We leave the investigation of such absolute verification mechanisms for future work. Despite

the limitations of our relative approach to evaluation, the GAN models and the sampling approaches we develop cautiously elevates the state of practice and introduces a necessary alternative to current methods of generating stimuli.

In conclusion, chapters in Part II demonstrate how deep generative models trained on signal traces can be used to simulate software dependencies in a virtual test rig. By constructing GANs with accessible latent spaces, we exploit properties of DNN embeddings – notably latent space interpolation for semantic combination – to develop methods for controllable stimulus generation. Further, exploiting the differentiable nature of a DNN itself, we demonstrate a method to place software-in-the-loop with the GAN to ease the search of test stimuli. Thus, methods introduced in these chapters have the potential to significantly ease the process, and improve the credibility, of virtual software testing. This allows the software testing process to reduce its dependency on vehicle testing, increasing the cadence of software engineering without compromising quality.

Part III

Principled operations in vector space

Chapter 12

Vector operations - a joint re-examination

In the epic journey that is the development of vehicle application software, this work has focused upon the activities of design and testing. The former paves the way for implementation by clarifying the broader principles which code should follow when realizing functionality. Upon, or even during, implementation, the latter verifies whether the code manages to fulfill intended functionality. Practically, at least in the automotive context, disciplines of design and testing do contrast strongly. Since automotive software design is mostly about defining the nature of code, it is practiced in the industry predominantly as a software discipline. Meaning that design practitioners are usually software engineers who use tools and methods – like design patterns – that are recognizable in the general practice of developing software. For example, a design pattern like Controller-Handler (see Chapter 4) allows a common approach to designing the structure and content of software, largely independent of whether the system it operates is, say, air suspension or battery monitoring. Since we are discussing vehicle control software, however, it is always important to remind ourselves that the larger application includes mechatronic hardware elements in addition to software. This is why testing, as we noted earlier, cannot always be isolated to code and therefore bears a much more multidisciplinary character. Depending upon the specific system that the software controls or monitors, concerns from several disciplines begin to be included into the testing process. For instance, testing battery monitoring software is as much about physics and chemistry as it is about software engineering. The sizeable contrast between the automotive software testing and design disciplines have sparked fields of research, industry niches, and professional careers specializing in either one of them. Yet, in our own tryst with these contrasting disciplines, previous chapters show that we have been able to address both using a unified recipe of principles, tools, and methods, which we subsume under the term 'pre-train and calculate'. The focus of this chapter is to recount and re-examine this recipe, and elucidate the elements which unify them. In doing so, we demonstrate

how a minimal set of principles utilize powerful representations learned by foundation models to solve practical software engineering tasks in a largely rule-based manner, without needing to resort to expensive supervision.

12.1 Learning representations of domains

The first step in our common recipe for addressing software design and testing tasks is identifying the domains of information that we choose to work with. In Part I, which deals with the design activity, we focus on source code, and in Part II, dealing with testing, we work with signal traces. As modes of information, code and signal traces are quite disparate. The former is a univariate sequence of discrete tokens, which combine elements of formal and natural language. The latter is a real-valued, multi-variate time series of fixed duration, depicting some aspect of vehicle operation. However, as we noted in Section 2.2, dealing with disparate domains is among the greatest strengths of deep learning. At its most fundamental level, any DNN is a computational graph that transforms one vector (or tensor) to another. So, as long as data from some domain is representable as a vector, it can be processed by a DNN. Real-valued time series signal traces are inherently vectors, but a sequence of code tokens is clearly not. This is why, as explained in Section 5.1, raw code is subjected to a tokenization process which converts discrete code tokens into vectors. From the perspective of a DNN, upon vectorization, the mechanism for processing inputs from domains as disparate as code and signal traces are practically indistinguishable. Let us therefore denote either vectorized domain – source code or signal traces – as $\mathcal{X}$, where each element $X \in \mathcal{X} := \mathbb{R}^n$ is an n-dimensional vector.

Then, the second step in our common recipe is to train a foundation model $\mathcal{E} : \mathcal{X} \rightarrow \mathcal{Z}$, which maps the input domain $\mathcal{X}$ into one or more abstract domains $\mathcal{Z}$. In our case, foundation models are, of course `tasnet` for the domain of source code, and `logan` or `silgan` for the domain of traces. For ease of expression we will refer to `logan` and `silgan` jointly as `xgan`. In the abstract domain of representations, each element $z \in \mathcal{Z} := \mathbb{R}^m$ is an m-dimensional vector, which captures necessary information from the input vector X that is useful for a downstream task. Now, it is interesting to ask ourselves – why is this transformation necessary? As pointed out in [163], practical predictive tasks may actually need to work on specific aspects, and not always on all the information, encoded in the raw input vector X. Instead of hand-crafting algorithms that extract necessary information from the input based upon the task at hand – an effort-intensive approach – self-supervised training helps learn representations with minimal human effort. Further, since foundation models, and the representations that they learn, are expected to cater to innumerable tasks downstream, this self-supervised training step is usually referred to as pre-training. With the network $\mathcal{E}$ encoding the input domain into an abstract domain of representations, we refer to it as an encoder. In learning to represent our two domains of interest – source code and signal traces – we use two different pre-training objectives. For `tasnet`, which learns representations of code, we use the cloze or Masked Language Modeling (MLM) objective, which we detail in Chapter 4. In contrast, for `xgan`,

which learns representations of signal traces, we use the adversarial objective detailed in Chapter 9. In adversarial learning, it is important to note that the encoder network alone is not sufficient. Since the essence of adversarial training is to pit a generator (which generates fake samples) against a discriminator (which assesses generated samples), we chain the `xgan` encoder with two additional networks. The generator $\mathcal{G} : \mathcal{Z} \rightarrow \mathcal{X}$, which is a decoder network that maps representations back into the input domain, and a discriminator $\mathcal{D} : \mathcal{X} \rightarrow \{0, 1\}$, which assesses whether generated samples belong to the input domain.

Straightaway, as visualized in Figure 12.1, similarities in representational approaches for compliance assessment and stimulus generation become obvious. Both `tasnet` and `xgan` use encoder networks to map their respective input domains into an abstract representation space. In the former case, representations are referred to as (program) embeddings, while they are referred to as latent codes in the latter case. The choice of names is simply a matter of tradition, reflecting preferred nomenclature of the language modeling and GAN communities. The terms themselves can be used interchangeably. More importantly, in either case, the character of the encoded representation vector z is the same. Based upon the guidance provided by the pre-training objective, z distills semantically important information from the input. The pivotal role that latent vectors plays in design compliance assessment and in stimulus generation is clearly visible in discussions in preceding chapters, and also in forthcoming sections which recount them. Having noted the similarity between `tasnet` and `silgan`, let us also briefly highlight their differences. The primary application of `tasnet` is to take a pair of programs as input and predict a compliance score. Such a task, like stereotypical image classification, is discriminative in nature, which is why an encoder alone suffices. The `xgan` networks, with their objective of simulating multi-physics vehicle dependencies, need to generate realistic traces as stimuli. Since this use case requires explicit generative capabilities, an encoder alone does not suffice. In order to endow generative capabilities using the adversarial objective, `xgan` is extended to include a generator $\mathcal{G}$ and a discriminator $\mathcal{D}$.

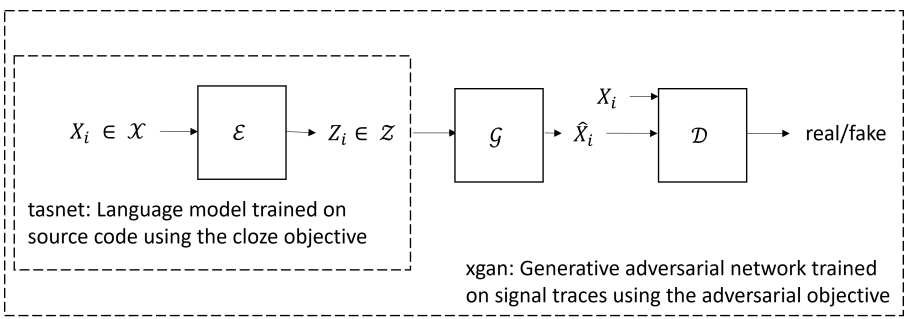


Figure 12.1: A joint visualization of the layout of `tasnet` and `xgan` networks

An interesting corollary to consider is whether these two approaches used for representation learning can be interchanged. That is, does it make sense to learn a program embedding model using the adversarial objective and a signal trace encoder using the cloze task? Indeed, as seen in [164], GANs have been used for source code generation and, as reported in [165], cloze has been used to

learn time series representations. While it seems like interchanging pre-training objectives is possible, we reason that it is inefficient to do so for the tasks at hand. Since design compliance assessment is a clearly discriminative task, training it as a conditional GAN with the `xgan` layout is wasteful. We only need the encoder for compliance assessment, which means that the generator and discriminator need to be discarded after training. Contrarily, using cloze for training time series representations produces an encoder alone. Since we need generative capabilities, we may as well add extra networks and train them end-to-end using the adversarial objective. Of course, alternative objectives and model architectures are possible for learning representations of either code or traces, and we have already discussed them in detail in preceding chapters. So far, we have seen the first two steps in our common recipe – identifying the input domain and training a foundation model to construct a representation space for the chosen input domain. Then, depending upon the specific task, the next step in the recipe is to choose appropriate vector operations in representation space to calculate necessary predictions in a rule-based manner.

12.2 Regularity for design compliance

In Part I, where we assess design compliance of vehicle application software, $\mathcal{X}$ is the domain of source code and $\mathcal{E}$ is the programming language encoder model `tasnet`. Given a pair of programs $(X_1, X_2),\ X_1, X_2 \in \mathcal{X}$, the objective is to assess whether they jointly comply with the principles of the Controller-Handler (CH) design pattern. With the task being conducted on a pair of inputs, one immediate need is to define a representation that jointly captures necessary properties of both inputs. As discussed in Chapter 5, the representation we choose to capture such jointness is the difference vector $r = z_2 - z_1,\ z_i = \mathcal{E}(X_i)$ between their embeddings. Using the difference vector between input embeddings spotlights a key component of our recipe, which is the use of simple vector operations in representation space to solve predictive tasks. However, Figure 12.2, which visualizes the jointness vector, clearly shows that r is not the only way to capture jointness. The vector q, the midpoint between input embeddings, is at least one an alternative representation of jointness. In fact, as we will revisit shortly, the midpoint q is one joint representation that we use for conditional sampling in the stimulus generation use case. Why, then, do we choose the difference vector r as a representation of program jointness? The answer to this question highlights another key component of our recipe – the operations we use to extract predictions are both simple, and based upon well-founded properties in representation space.

During discussions in Chapter 5, we noted that the key reasoning behind choosing the difference vector as a representation of jointness is the principle of embedding regularity. Following the introduction of some of the earliest neural language models, one important study [84] observed that pairs of analogous concepts are likely to form a parallelogram in the representation space of language models. Briefly, given pairs of pairs (X_1, X_2) and (X_3, X_4), where constituents of each pair are related by the same concept, their embeddings approximate a parallelogram. In the case of [84], the subject of analysis was

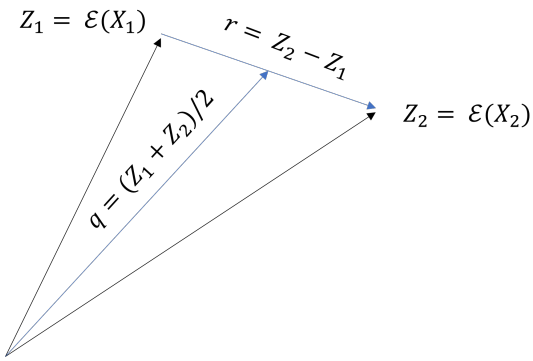


Figure 12.2: Two simple techniques that we use for joint representations. The difference vector r, which we use for compliance assessment with `tasnet` and the interpolation q, which we use for conditional generation using `xgan`

a quartet of words (`King`, `Queen`) and (`Man`, `Woman`), with each pair of words having the same relation – words in a pair are the male/female form of each the other. In our case of design compliance assessment, we analyzed a quartet of programs like (`MirrorHeating`, `PassMirrorHeater`) and (`CabTiltLock`, `CabTiltLockMotor`) related by the same concept – the rules of the CH design pattern. This is why the bedrock principle of the DECO algorithm to measure design compliance (Algorithm 1) is to assess whether the parallelogram geometry is observable in the program embedding space. Given a quartet of embeddings, a straightforward way to verify the parallelogram geometry is to check whether $z_3 + (z_2 - z_1) \approx z_4$. Note that a key element in conducting this check is the difference vector $r = (z_2 - z_1)$, which is what we use as a representation for the jointness of the pair of programs (X_1, X_2). Using the difference vector as a representation of jointness, based upon the property of regularity in language embeddings, DECO simply uses the level of alignment with the parallelogram as a measure of compliance. Adding to the fact that we construct a predictive model using trivial vector arithmetic, the fact that we repurpose a well-observed property like regularity makes our approach principled and explainable. In contrast, a task specific model trained with explicit supervision not only incurs significant cost, but is always likely to be more of a black-box.

Beyond [84], the property of regularity has been consistently observed in word embedding models like [56, 166, 167], and also in contextual language models that embed more than one token [85, 168]. So common, in fact, is this empirical observation, that assessing regularity has emerged as a method to evaluate language model training [97]. Our work simply extends this empirical observation beyond word and sentence embeddings and applies it to language models of vehicle application software. Apart from empirical studies, numerous efforts have also been undertaken to understand explain regularity from a theoretical perspective. As noted in one relatively recent analysis [169], explanations of embedding regularity have been attributed to several causal factors including the intrinsic algebraic structure of embeddings, the notion of paraphrasing where related terms can be interchangeably used, and the co-occurrence of related

terms in training corpora. Like many other properties of trained DNNs, a firm theoretical basis of embedding regularity remains to be found. The fact that empirical observation of regularity is constantly supplemented by theoretical examination bodes well for the application of this property to predictive tasks.

12.3 Interpolation for stimulus sampling

If we now come to Part II, where we simulate dependencies of vehicle application software in a virtual rig by generating test stimuli, the domain $\mathcal{X}$ represents signal traces. Unlike the simpler network layout of `tasnet`, the encoder $\mathcal{E}$ is only one of several networks in `xgan`. But, just like the method for measuring design compliance, the latent space to which the GAN encoder $\mathcal{E}$ encodes is pivotal to our approach of conditional generation. A basic form of conditional sampling used in `xgan` involves setting two conditions X_1 and X_2, which are signal traces themselves, and tasking `xgan` to generate a novel trace that combines the semantic properties of condition traces. Echoes of vector operations used in the previous design compliance use case are hard to miss, with the immediate need being the construction of a joint representation of the two condition traces. Where we used the difference vector r as the joint representation for design compliance, the midpoint vector q (Figure 12.2) serves as a better representation of jointness for conditional sampling. While the choice of r is inspired by the property of embedding regularity, the choice of q for conditional sampling is based upon the property of latent interpolation for semantic combination. As noted in Chapter 9, among the first to observe this property was an experiment in [58], which conducted a 'walk' in the latent space between two codes to produce semantic intermediates. In the experiment they conducted, the input was two images and intermediate codes, produced by linear interpolation between latent codes of the inputs, were observed to proportionally combine the semantic properties of the inputs. That is, when these intermediate codes are decoded, they produce novel images that smoothly combine characteristics of the two input images. Since producing a novel intermediate between two trace conditions is highly beneficial for virtual software testing, we use the linear interpolation q between latent codes of conditions as the primary mechanism for joint representation.

While the property of latent interpolation for semantic combination serves as our main tool for conditional stimulus sampling, methods we present in Part II extend this property in several ways. The MLERP algorithm is a good example of extension, where interpolation is jointly conducted in two spaces – the abstract latent space of `logan`, and a user-specified metric space. Not only does this ease the tester's effort in selecting interesting samples, but the use of a metric to compare generated traces with a reference trace directly measures the quality of generation. The `silgan` framework further extends interpolation in two ways. First, it allows interpolation to take place in a representation space of handcrafted templates, before translating them into realistic traces. The `logan` model, in contrast, only employs interpolation in a (variational) auto-encoder setting. Second, `silgan` generalizes interpolation from a straight line to a hyperplane of K vertices in the latent space. Just like interpolation between two latent codes, any

latent code z on this hyperplane is a convex combination of the latent vertices, which means that it proportionally combines semantic properties of K (or an arbitrary number) of conditions. Clearly, these extensions of the latent interpolation property constitute a powerful toolkit for conditional stimulus sampling.

Mirroring the observational record of regularity in language models, the property of interpolating latent codes in achieving semantic combination has been consistently observed in generative models. While first observations like [58] were reported on generative models in image domains like human faces, this property has also been observed in generative models of text [170], video [171], protein structures [172], etc. Our work applies and extends this principle in the domain of time series signal traces. Unlike embedding regularity, which is rarely used in predictive contexts, it is much more common to see latent space interpolation used for predictive tasks. GAN inversion [173] is one example of an entire field of research that focuses upon using latent space operations to achieve conditional generation. In addition to empirical observations made on GANs trained in different domains, latent space interpolation has also been subjected to theoretical analyses. As pointed out in two examples [174, 175], theoretical inquiries focus mostly on understanding the geometrical nature of the latent space itself and the semantic import of taking different trajectories in this space. While linear interpolation may be the most commonly used method, spherical and normalized interpolation [175] are viable alternatives. The study in [175] also observes that there is no perfect option, with the effectiveness of the interpolation technique depending upon the domain. Investigating whether alternative interpolation techniques improve the quality of conditional generation is an interesting avenue of future work. Thus, in addition to embedding regularity, latent space interpolation constitutes the second major property of representation spaces that we use for sharp predictive tasks in our work. Apart from being based on trivial vector arithmetic, the well-attested ability of GAN latent space interpolation to achieve semantic combination means that our approach for generating test stimuli is explainable, and based upon repeatedly observed properties of generative models.

12.4 Gradient descent for stimulus search

Besides sampling, one other technique which we introduced in Part II for conditional stimulus generation is searching in the latent space. The act of defining multiple conditions may bound the space for stimulus sampling, but any `xgan` latent subspace offers infinite possibilities for stimuli. This makes any sampling technique, even highly selective ones like MLERP, relatively inefficient. As an alternative to sampling, we therefore defined the GRADES algorithm which, given a test objective, conducts gradient descent-based search in a demarcated latent subspace to automatically and efficiently find necessary stimuli. In previous sections, we recounted that the DECO and MLERP algorithms, used respectively for design compliance assessment and test stimulus sampling, are based upon well-observed properties of representation space. GRADES, on the other hand, is derived from an elementary property of DNNs themselves. Most practical DNNs, especially the generator network $\mathcal{G}$ in `xgan` which maps the latent space

to the domain of signal traces, are differentiable. The differentiable nature of a DNN is, in fact, a property that is fundamental to training using backpropagation. Descending in the network's parameter space, using the gradient of the loss function, remains the primary technique for finding network parameters that help achieve the training objective. While the differentiability of practical DNNs is primarily applied for gradient descent in parameter space, we repurpose this property to conduct gradient descent in latent space to search for stimuli.

As seen in Figure 12.1, the latent space $\mathcal{Z}$ is at the input of the generator network $\mathcal{G}$. The objective, in the case of GRADES, is to find latent codes which produce a trace (or a hit) that satisfies a branching condition in the code under test. This way, searching in the latent space using gradient descent is essentially an act of optimizing the input of the generator network, in order to achieve some objective at its output. Such a gradient descent-based approach for input optimization is, in fact, a technique that has been previously used for numerous applications. Arguably, the best known application of gradient-based input optimization in deep learning is the search for adversarial examples. A seminal experiment in [176] on image classification networks showed that, given an input X, for which the network assigns the correct label, gradient descent can be used to find a minimally perturbed version of the input X', that causes the network to drastically misclassify. One remarkable aspect of this experiment is that the search can be constrained so that perturbation in X' is barely perceptible and, to the average human eye, X' looks exactly like X. Yet, this imperceptible perturbation seems sufficient to cause the image classifier to emphatically mislabel the perturbed input. Since they are deliberately optimized to induce faulty behavior, such inputs have been called adversarial inputs, and gradient descent based optimization in the input space has proven to be highly effective in identifying them. The study in [176] has since sparked an entire research discipline of identifying adversarial techniques to both attack and defend trained DNNs, in an attempt to understand and improve the robustness of trained networks [177]. The probing capabilities of input optimization has also been used as a tool to explain the predictions of the network [178]. Put simply, the gradient of the input with respect to the output is a rudimentary measure of the sensitivity of network predictions. The gradient of the input with respect to the output has therefore been used as a building block to construct techniques of explanation which attribute network prediction to salient parts of the input. Taking inspiration from adversarial inputs and gradient-based attribution, in our approach to search for the right stimuli, we simply reappropriate gradient descent-based input optimization and apply it to the latent space of `xgan`.

It is important to note, however, that our extension of gradient-based input optimization to `xgan` for stimulus search required substantial innovation, the first of which was the definition of coverage indicators. Any act of gradient-based input optimization needs a differentiable objective to guide the search. In the basic case of adversarial examples, the objective is to achieve erroneous prediction – a relatively straightforward condition at the classifier's output. In the case of stimulus search, however, we do not set an objective directly on `silgan` output, meaning the traces it generates. Instead, the objective for latent optimization is based upon the code to which generated traces are applied as stimuli. Code,

like the toy function we saw in Figure 10.7 is rarely differentiable. Therefore, targeting code coverage, we defined real-valued equivalents of discrete boolean operations and transformed raw code into a search function, which outputs a differentiable coverage indicator. The availability of such a search function is a prerequisite to using GRADES. Even with a concrete target to achieve – searching for a hit that makes the search function evaluate to a negative value – GRADES undertakes additional adjustments. The search landscape, or the mapping between latent codes and coverage indication, is far from optimal for gradient descent. As shown in Figure 10.10, there are large regions in latent subspace where the coverage indicator remains practically unchanged, providing either weak or zero gradients with respect to the latent code. This is why GRADES first samples the landscape to identify regions where strong gradients are available before deferring to gradient descent. Thus, while gradient based input optimization may be a well-known technique, the notion of coverage indicators and the technique of sampling before search introduced in GRADES are needed to bridge the gap and produce a viable technique for stimulus search. Overall, in basing GRADES upon gradient descent in the latent space, we use yet another property that is well-understood and repeatedly used in deep learning, bolstering the soundness and explainability of our approach for searching stimuli.

12.5 Representational similarity as substratum

In Parts I and II, techniques that we develop to ease the design and testing of vehicle application software repeatedly refer to one more fundamental property of embedding spaces – representational similarity. Stated simply, representational similarity means that if X_1 and X_2 are semantically close in the input space, then their representations $z_1 = \mathcal{E}(X_1)$ and $z_2 = \mathcal{E}(X_2)$ are likely to be geometrically close in the space of vector embeddings. So far, in recounting DECO, MLERP and GRADES, we have mainly portrayed these algorithms as being respectively based upon properties of embedding regularity, latent interpolation, and DNN differentiability. What we highlight now, is that the property of representational similarity is an equally crucial component in all three approaches.

In DECO, we may have begun by using the average difference vector as the benchmark for jointness. Immediately afterwards, the mechanism that we use for checking the alignment between the query and benchmark jointness is measuring the similarity between predicted and actual handler embeddings. That is, to see if the query programs are compliant with the CH pattern, we check if the predicted handler embedding, calculated using the benchmark, is semantically similar to the actual handler embedding. Further, the succeeding operation of converting this similarity measure into a discrete interpretable rank is also based upon similarity. Instead of simply measuring the semantic similarity of the predicted handler embedding with the ground truth handler embedding alone, it is measured against all programs in the test corpus. The index of the ground truth handler in the sorted list of similarity measures serves as the rank. The rank, in turn, plays a crucial role in the calibration exercise which demarcates the three intervals for judging compliance. In the

sequence of steps used to assess design compliance, representational similarity plays a central role in all but one step, which is a resounding measure of its importance to DECO. Moving on to MLERP, though we do not directly measure semantic similarity of latent codes during conditional sampling, the notion of representational similarity looms large. To some extent, semantic combination achieved by latent space interpolation is derivable from representational similarity. Given this property, the spatial separation between two points in latent space should reflect the semantic similarity of their counterparts in input space. Then, a point midway on the straight line connecting two latent codes is likely to represent a sample that is half as similar as either input. Generalizing beyond two codes, points on a latent hyperplane, just like points on a line, are convex combinations of their vertices, and are bound by semantic similarity to proportionally combine their characteristics. Results in Section 10.3, where we randomly sample a latent hyper-triangle, support this reasoning. Not only is this reasoning relevant for conditional sampling, but it also extends to searching. In GRADES, we take special care to restrict the search to the latent hyperplane demarcated by conditions. A point outside the hyperplane is not a convex combination of its vertices, and is no longer bound by semantic similarity to combine the characteristics of specified conditions. With representational similarity strongly underpinning latent interpolation, the extent to which we rely upon interpolation for conditional sampling and search sufficiently highlights the indirect, yet important, role played by this property in our stimulus generation enterprise. Simply put, all three algorithms that we develop to solve tasks in software design and testing depend, at some level, upon this important property. It is also useful to note that, just like the properties of regularity and interpolation, representational similarity has also been subjected to empirical and theoretical scrutiny. As described in one recent analysis [179], such studies focus upon aspects like understanding the mechanics of learning representations, and identifying appropriate methods for measuring representational similarity.

12.6 Put together, pre-train and calculate

As a final act of re-examination, let us review the three techniques that we develop for solving tasks in automotive software design and testing.

1. To assess compliance of vehicle application software with the CH design pattern, we use (a) `tasnet`, a foundation language model for automotive software, and (b) DECO, a rule-based mechanism which assesses embedding regularity in `tasnet` representation space to measure compliance.

2. To generate realistic stimuli for testing vehicle application software in virtual rigs, we use (a) `logan`, a foundation GAN model of signal traces, and (b) MLERP, a rule-based mechanism which primarily uses latent space interpolation for conditionally sampling stimuli.

3. To search for realistic stimuli that satisfies a test objective like code coverage, we use (b) `silgan`, a foundation GAN model that translates templates to

signal traces, and (b) GRADES, a rule-based algorithm that mainly uses gradient descent in latent space to search for appropriate stimuli.

As a distillation of the common recipe that we use for solving software engineering tasks, this three-point review speaks for itself. The predictive approach that we employ has two essential steps (a) a foundation model pre-trained on the domain of operation, and (b) a rule-based algorithm that calculates a prediction using simple vector operations. Together, they constitute 'pre-train and calculate', the paradigm that we originally set forth in Section 1.3 as the main approach we use to solve automotive software engineering tasks. Studies in Parts I and II, along with their re-examination in this chapter, clearly demonstrates that this paradigm successfully automates complex tasks in automotive software design and testing, the overall objective which we set in Section 1.2. Thus, pairing pre-trained foundation models with rule-based 'heads' for calculating predictions institute a viable model for unsupervised task specialization in software engineering. Moreover, this approach is vastly more nuanced than stereotypical fully supervised training, or fine-tuning, which use supervision to predict a relatively rigid discrete set of labels. Predictions with 'pre-train and calculate', in contrast, are much more nuanced, easing the application of these tools in real-world tasks that require margins and trade-offs in decision-making. Further, the use of properties, which are empirically observed and theoretically studied, in calculating predictions makes this approach principled, relatively transparent, and explainable. This increases the likelihood that engineers use these tools with confidence.

12.7 What about pre-train and prompt?

The exercise of re-examination, conducted in this chapter, helps culminate the reasoning that 'pre-train and calculate' constitutes a paradigm for unsupervised, nuanced, and explainable task specialization in automotive software engineering. In the extended presentation of this reasoning, over several previous chapters, we have repeatedly compared this paradigm with two dominant alternatives, one being fully supervised training, the other being 'pre-train and fine-tune'. As pointed out in Section 2.2, both alternative approaches require annotation or supervision, the former at large scale and the latter at relatively smaller scale. Tasks in automotive software engineering, as we have repeatedly noted, do not easily lend themselves to the rigid recipe for task specialization that either supervised paradigm offers. Engineering tasks are fluid, involve several trade-offs, and involve humans in the loop. Many tasks therefore, particularly the software design and testing tasks that we have examined, cannot be easily depicted as annotated datasets, precluding supervised task specialization. The alternative 'pre-train and calculate' approach, which we demonstrate, avoids explicit supervision and, using principled rule-based predictive approaches, introduces much needed nuance and transparency. While 'pre-train and calculate' may have clear advantages over supervised paradigms for solving certain types of tasks, there is one other paradigm, something that been massively disruptive in recent years, which is interesting to additionally consider, and that is 'pre-train and prompt'.

When discussing foundation models in Section 2.2, we stated that these models are typically not used directly to solve tasks. We had also qualified this statement, noting that this stance has been challenged, quite successfully, by the 'pre-train and prompt' paradigm. Let us therefore briefly examine the essence of this new paradigm, and we can do this by reverting to sentiment and emotion classification tasks, which we used as running examples in Section 2.2. These tasks involve processing a natural language sentence like `'I miss my dog'`, and classify the attitude or feeling it conveys. On a positive/negative scale, a stereotypical sentiment classifier would classify this sentence as conveying a negative sentiment. Similarly, an emotion classifier, using a much more elaborate discrete scale, would classify it as conveying, say, 'sadness'. Let us then turn to a foundation model of natural language like BERT which, being trained upon a large corpus of natural language text, should be able to process the sentence in question. In fact, it is precisely this capability that makes it useful for seeding a task specialist using 'pre-train and fine tune' paradigm. While BERT can certainly be fine-tuned, can we use it without any extra training for sentiment or emotion prediction? BERT, if we recall, has been pre-trained using the cloze objective. The model is presented a sentence with some of its tokens masked, and is tasked to predict tokens that should appear in masked positions. After pre-training, therefore, let us consider presenting the input `'I miss my dog. I feel <mask>'`. Leveraging its training, BERT is likely to complete the sentence by predicting `'sad'`. This way, without using any additional training, BERT can be subverted into predicting the emotion of a sentence. The key element of subversion is, of course, the 'prompt' that we add to the original sentence. The cleverly worded appendage `'I feel <mask>'` sufficiently prompts BERT to predict its emotion. The benefit of this approach is that we get a task specialist with no extra effort – the pre-trained foundation language model suffices. The cost, of course, is that we need to identify an appropriate prompt for the task at hand.

While cleverly prompting BERT opens up an interesting paradigm for prediction, generative language models like GPT-3 have proven to be far more amenable to prompting. Generative language models are typically pre-trained on the Causal Language Modeling (CLM) objective. Under this objective, the model is presented with a sequence of tokens and is tasked to predict tokens that should follow. This, as can be clearly seen, is very much the pattern in which the BERT prompt `'I feel <mask>'` is constructed. Except that in the CLM case, there is no need for `<mask>` and the model directly predicts tokens that are likely to follow `'I miss my dog. I feel'`. CLM pre-training, therefore, produces models which can be prompted much more freely. One example, from [45], illustrates this capability, where the prompt `'Translate English to French: cheese =>'` makes GPT-3 correctly predict the most likely subsequent token `fromage`. It may have begun with simple recipes of this nature but today, the act of constructing prompts has become a discipline in its own right – prompt engineering. As we speak, this discipline is pushing boundaries, constructing prompts with cleverness and sophistication [180], extracting predictions of increasing value.

Primarily, this discussion serves to show that we now have access to a powerful alternate 'pre-train and prompt' paradigm, which extracts predictions from foundation models with no extra training. What, therefore, are its implications on

the 'pre-train and calculate' paradigm, which we have used for unsupervised task specialization in automotive software engineering? Let us begin by first understanding their similarities. Clearly, both paradigms base themselves upon pre-trained foundation models. In both cases, the use of domain generalists provides the necessary breadth to solve more than one task. Another aspect of similarity, perhaps less evident, is that both of them are based upon empirically observed properties of foundation models. In 'pre-train and calculate', for instance, a well-observed property like embedding regularity has been used for assessing design compliance. Interestingly, there are parallel well-observed properties in prompting. For example, experiments in prompting revealed that generative language models consistently underperform tasks involving complex reasoning. In [181], for instance, the following multistep math word problem was posed as a prompt:

```
Q: A juggler can juggle 16 balls.  Half of the balls are golf
balls, and half of the golf balls are blue.  How many blue golf
balls are there?  A:
```

Many models, including GPT-3, responded to this prompt with wrong answers. Astoundingly, [181] observed that simply appending the string `'Let's think step by step'` to the prompt, causes the model to not only predict the correct answer, but also provide the following step-by-step reasoning.

```
There are 16 balls in total.  Half of the balls are golf balls.
That means that there are 8 golf balls.  Half of the golf balls
are blue.  That means that there are 4 blue golf balls
```

Though it may seem bizarrely arbitrary, this prompt is nonetheless an empirically observed 'property' of generative large language models that can be used for extracting predictions. In fact, [181] performed this experiment on more than a dozen models, using about half a dozen benchmark tests, to confirm the consistency and versatility of this prompt. In that sense, this prompting observation is not materially different from the consistent observation of embedding regularity in language models, which we repurpose for predicting design compliance. Further, the example above also makes it clear that predictions can be quite descriptive and self-explanatory, and prompting can be nuanced. What, then, are the differences between 'pre-train and prompt' and 'pre-train and calculate'?

The first major difference is that prompting takes place in the interpretable input space, while calculation takes place in abstract representation space. Therefore, no matter how clever or descriptive it is, the effectiveness a prompt is bound to the syntactic and semantic capabilities of the formalism used for prompting. Take the example of semantic combination, which we used in MLERP for generating novel stimuli that are semantically intermediate between two conditions. Achieving this semantic combination in the input space of signal traces is clearly difficult but in latent space, it is trivially achievable using linear interpolation. In this case, the `xgan` representation space has traded-off interpretability in favor of simplicity in semantically arranging and manipulating information. As a result, a semantic combination that is difficult to express (or prompt) in input space, is trivially accomplished in representation space. The crucial insight that can be drawn from this is that representation space projects additional capabilities that

are not easily available in input or prompting space. The second major difference is that, currently at least, the prompting paradigm is heavily reliant upon language. Even if a model predicts modalities other than text, the prompting interface largely remains textual. As an example, the DALL-E image generation model which we noted earlier, may take images at its inputs but still relies upon text for a bulk of the prompting. How such a text-heavy prompting process extends to a modality like signal traces, remains an open question. Exposing a dedicated text input modality in `xgan`, simply for the purpose of prompting, can incur significant cost. For instance, unlike images which co-occurs with text quite often, cases where signals and text occur together are extremely rare. Thus, while it is likely that prompting becomes increasingly multimodal, it remains to be seen how prompting scales to incorporate niche modes of information, which are specific to a discipline like automotive software engineering. Innovation of new multimodal prompting paradigms for solving software engineering tasks in the automotive context is a clearly interesting area for future investigation.

Also, the fact that prompting focuses on the input and calculation focuses upon representations means that the techniques can complement each other. A combination of the relative strengths of both paradigms has the potential to reveal powerful techniques for extracting predictions. Practically, prompting has been typically used on decoder-only generative models which do not expose a representation space. There are, however, language models like BART [91], which do expose a latent space. Such generative latent space models provide another interesting future avenue for combining prompting and calculating paradigms.

Chapter 13

Explaining representation spaces

The previous chapter recounted the 'pre-train and calculate' paradigm, which we use for building tools that ease design compliance assessment and virtual testing of vehicle application software. Key elements of the paradigm are foundation models that learn abstract representations of input domains, and rule-based algorithms that use simple vector operations in representation space. These vector operations are themselves based upon well-observed properties of representation spaces like semantic similarity, regularity, and interpolation. Relying upon inherent properties of representation spaces may have allowed us to take a principled approach to calculating predictions, but it is important to acknowledge that representation spaces are both high dimensional and abstract. This usually means that they do not lend themselves easily to scrutiny, making it difficult to identify and evaluate properties upon which predictive techniques can be based. As a result, there is a clear need to develop necessary techniques that probe foundation models in order to better understand their properties. The previous chapter itself points out many such techniques that attempt to explain how properties like similarity or regularity manifest. Such explanation was crucial to our construction of DECO, which repurposes these properties for assessing design compliance. While such probing studies are valuable, it is important to note that probing techniques for explaining foundation models need to address the fact that these models have domain focus, and not task focus. In the context of stereotypical task specialists like a cat/dog classifier, most probing techniques ([182] reviews a collection) focus on the task to explain properties more 'locally'. That is, given an input and a prediction made by the classifier, one common technique for explanation is to identify parts of the input that has most influence in causing the model to predict this label. Such an explanation can, for instance, show that the classifier chose a cat label mainly based on the ears, eyes and whiskers of the given cat image in the input. While such explanations are useful, they only apply to one particular input and its prediction, which makes the explanation local and task-specific. With foundation models having a relatively

global domain-level focus, explanations of properties, therefore, need to go well beyond one input and one task, to somehow address a domain as a whole.

Recent years have produced studies that focus specifically on explaining domain comprehension, examples of which are [183], which studies properties of image representation, and [184] which does the same in the domain of clinical health data. In parallel, with the rapid advance of patently generalist language models, a number of benchmarks have been introduced that probe, test, and explain various aspects of language and knowledge comprehension [118]. Taking inspiration from such probing techniques, we now describe two studies that we ourselves conduct in explaining specific high dimensional abstract spaces. The first, described in this chapter, focuses on the `tasnet` model and seeks to explain its adaptation to the domain of vehicle application software. The second, described in the next chapter, focuses on training data itself, and seeks to explain the nature of its sample representation. At the outset, we wish to clarify that the two studies that we describe are preliminary and are not fully explored. We mainly present them as initial findings that could potentially be expanded in future work.

13.1 Explaining domain adaptation in tasnet

In Part I, we presented the `tasnet` model and its application to the task of design compliance assessment. While describing the use case, we noted that under current practice, assessing whether a pair of programs comply with the principles of the CH design pattern is a manual activity that is reserved for experienced programmers and software architects. The need for using expert knowledge arises simply because judging compliance with CH requires specialist understanding of different aspects of engineering vehicle application software. This rationale is, in fact, faithfully reflected in the manner in which we train `tasnet` as an approximation of this expertise. As explained in Chapter 5, the `tasnet` model undergoes a three-step training process. First, the `tasnet` variant $\mathcal{F}_A$ is trained on C code from the GitHub public dataset to impart a fundamental knowledge of C-programming. Then, $\mathcal{F}_A$ is adapted into the variant $\mathcal{F}_B$ by continuing to pre-train it on the TAS corpus, according to the training objective (5.9), so that specific knowledge of AUTOSAR application programming is imparted. Finally, $\mathcal{F}_B$ is adapted into $\mathcal{F}_C$, using the objective (5.10), to impart knowledge of controller and handler SWCs. As also noted in the results of the controlled experiment in Chapter 5, proceeding to train with the domain specific TAS corpus clearly results in better performance on the design compliance assessment task. When analyzing the improvement in performance that results with domain specific training in Chapter 6, one contributing factor that we reasoned is that the training results in an arrangement of embeddings with better regularity. Another factor, which we reasoned, was that training with a domain-specific corpus leads to a better understanding of the domain itself, which results in program embeddings that better capture necessary semantics. In this chapter, we seek to probe the latter reasoning. That is, we seek to explain whether the `tasnet` variant $\mathcal{F}_B$ is much more knowledgeable about AUTOSAR application programming in the TAS domain, compared

to $\mathcal{F}_A$. If confirmed, such an observation increases the confidence with which `tasnet` can be applied for solving any task, including design compliance, in the TAS domain. One alteration we wish to point out is that the `tasnet` model, which we discuss in this chapter, has a slightly different neural network architecture compared to the original `tasnet` model presented in Part I. Unlike the original model, which uses Reformer layers in its encoder, the `tasnet` model discussed in this chapter uses the vanilla Transformer layers. Using Transformer layers, `tasnet` variants studied in this chapter accept a maximum input sequence length of 1024 tokens. Apart from this, all major details, including the training corpus and hyperparameters, are largely identical.

From previous discussions in Chapter 5, one can recall that both $\mathcal{F}_A$ and $\mathcal{F}_B$ `tasnet` variants involve pre-training using the cloze or MLM objective. Therefore, one simple way to test comprehension of the TAS domain is to subject both models to a cloze test on a held out set of programs from TAS. The results of such a test is shown in Table 13.1 clearly shows that $\mathcal{F}_B$, which is fine-tuned on the TAS corpus, predicts masked tokens with 97% accuracy on held out TAS code. In contrast, $\mathcal{F}_A$, which has no knowledge of TAS, exhibits a relatively lower – but still surprisingly high – 93% accuracy in predicting masked TAS tokens. Is this not an indication that continuing to train using cloze on the TAS corpus imparts sufficient knowledge of the TAS domain?

Table 13.1: Mean top-1 cloze test[1] accuracy

Model	On the GitHub corpus[2]	On the TAS corpus[3]
$\mathcal{F}_A$	0.948	0.927
$\mathcal{F}_B$	0.914	0.972

[1] By uniform randomly masking 15% of tokens
[2] On held-out 5k files
[3] On held-out 10% of TAS files

13.2 Relative importance of tokens in a domain

Let us examine the sufficiency of the cloze test in testing domain comprehension using a code snippet adapted from TAS (Figure 13.1). This snippet reads the state of charge of a battery in the vehicle. Then, let us consider three hypothetical cloze tests conducted on this snippet, where tokens {, `==`, and `signal` are respectively masked individually. The first test, with { alone masked, is primarily a test of C syntax. The second cloze test, with `==` masked, goes beyond syntax and tests the model's understanding of the flow of logic in the snippet. However, if a token like `signal` is masked, then the test is actually about a concept that is crucial in the TAS domain. The notion of `signal`, as we saw earlier, is of fundamental importance to AUTOSAR, helping transact information between SWCs. Figure 13.1, which shows how a signal is typically read and validated, is an example of an oft-reused pattern for handling signals. Given such an exemplary snippet, as observed in comparable settings in [185] and [186], if `tasnet` is able to correctly predict a masked `signal` token,

it demonstrates the model's ability in understanding and summarizing the surrounding context of reading and validating signals. Stated otherwise, if the model correctly predicts masked instances of `signal` in the third cloze test, it is a strong indication that it has understood an important programming context in TAS. Thus, the question if the cloze test results in Table 13.1 reflects sufficient understanding the TAS domain boils down to whether enough tokens of domain significance are masked during the cloze test. This reasoning is also intuitive because a human C-programmer can demonstrate an understanding of TAS domain by showing how some of its key concepts like `signal` are used in practice.

```
// Get BatterySOC signal
input->batterySOC.status = SIG_NOK;
rteRet = Rte_Read_BatterySOC_rqst(&input->batterySOC.data);
if (rteRet == RTE_E_OK)
{
    rangeRet = validateSignal(input->batterySOC.data);
    // Check range
    if (rangeRet == SIG_IN_RANGE)
    {
        // Everything OK, assign signal
        input->batterySOC.status = SIG_OK;
    }
}
```

Figure 13.1: An example snippet adapted from TAS that reads battery state of charge. As indicators of knowledge in the TAS domain, highlighted tokens in this code snippet are much more important.

In any code corpus, tokens are not uniformly abundant, with programming language keywords occurring much more often. In TAS, for instance, the { token occurs more than 200k times. On the other hand, a domain keyword like `signal` occurs only around 12k times as a unigram and around 40k times as a subword. This means that the uniformly random BERT masking recipe (used for tests in Table 13.1), which is seemingly egalitarian, disproportionately masks more numerous syntactic tokens at the expense of rarer domain keywords. Such a skewed process is detrimental to the evaluation of domain comprehension in many ways. First, the mean cloze test accuracy is numerically dominated by syntactic tests. Mistakes in predicting domain keywords, which indicate a weakness in domain comprehension, do not significantly influence the score. Second, the predominant masking of syntactic tokens presents a relatively pristine code context, making it a weaker test for domain comprehension. For instance, applying BERT masking to the code snippet in Figure 13.1 is likely to mask only a tiny fraction, perhaps a single instance, of domain keywords like `signal`. If only one instance of `signal` is masked, and many more of its instances are visible, there is enough 'leakage' of information from the context for the model to correctly guess the masked token. With syntactic tokens being dominantly masked, domain-sensitive hints in the context may not be significantly suppressed. Third, masking a fixed proportion (15%) of tokens for each test is clearly narrow. If we take the same `signal` token in the example above, we may sometimes need to mask all (or 100%) of its instances for the test to be sufficiently challenging.

Nevertheless, the cloze test remains an effective and intuitive means for evaluating any language model that operates with code tokens. Therefore, in order to test the domain comprehension of `tasnet`, we simply realign the cloze test by masking tokens non-uniformly based upon their importance to the domain.

13.3 Conducting a vocabulary challenge

Based upon the reasoning presented above, we conduct a modified cloze test to check domain comprehension using the following steps.

Curating a vocabulary of domain keywords – As illustrated using Figure 13.1, even in a specialist code corpus like TAS, individual tokens are of varying significance to the domain. Based upon their significance to the TAS domain, we divide tokens (subwords) into three categories, (1) *special tokens* which are important markers for TAS domain awareness, (2) *basic tokens* which are language keywords and characters, and (3) *other tokens* which pools all that remain. Curating basic tokens in the C language is straightforward, but curating special tokens needs some effort. Using expert knowledge available at Volvo, we achieve this by examining the vocabulary of the TAS corpus and manually selecting tokens of domain significance. The list of special tokens that we select for testing TAS domain comprehension are `signal`, `read`, `write`, `run`, `CAN`, `Rte`, `call`, `controller`, and `handler`. Manual curation, even with expertise in TAS, undoubtedly tends to be subjective and non-exhaustive. However, we find that curating this compact, yet healthy, list of domain keywords yields interesting results. It is important to note that subwords, so curated, are treated as seed keywords. Other subwords, in which a seed keyword is a substring, is also included. For instance `runnable`, in which the seed keyword `run` is present, is also included as a domain keyword. Put together, the total number of special tokens used in our modified cloze test of domain awareness accumulates to 125.

Masking non-uniformly – In order to test domain comprehension, we then design a recipe that masks each of the three token categories differently. First, we minimize syntactic tests setting the masking rate for basic tokens as $M_B \sim U(0, 0.04)$. We then shift domain keywords to the 'foreground' by masking special tokens at a rate $M_S \sim U(0, 1)$. This ensures a full spectrum of foreground masking, where the model is challenged with none to all the instances of domain keywords suppressed. This then leaves other tokens, which we mask with the rate we set $M_O \sim U(0, 0.35)$. With the basic and other token categories together constituting the 'background', M_O primarily decides its level of suppression. Put together, this recipe of non-uniformly masking tokens based upon their domain significance results in an effective average masking rate that is quite close to the BERT rate of 15%, with which `tasnet` is trained. A comparable non-uniform masking recipe, but based upon token frequency and not importance, was proposed by [187] for language model pre-training.

Conducting the cloze test – In each test iteration, based upon sampled masking rates M_S and M_O, a fraction of foreground and background tokens

in a randomly drawn test code snippet c is replaced by a mask token m. This creates the masked code snippet c_M, which is then presented as input to the `tasnet` variant under test. Then, the top-1 prediction accuracy per token category is calculated according to (13.1), by comparing the predicted token with the masked ground truth. Accuracy scores are averaged over ∼1M test iterations for each variant of `tasnet` tested. Among measured scores, the accuracy of predicting special tokens of domain significance (u_S) thus becomes a direct measure of domain awareness.

$$\begin{aligned} u &= \mathbb{E}_{c \in C}\ \mathcal{F}(c_M)[k] == c[k], \\ k &= \{i : t_i = m, t_i \in c_M\} \end{aligned} \tag{13.1}$$

Since our redesigned cloze test, in effect, measures a model's capability in understanding the context of a curated domain vocabulary, we refer to it as the *vocabulary* challenge. For each model, we run two instances of the challenge, setting the length of the input code snippet L_c to 512 and 1024 subwords respectively.

13.4 Results

The results of the vocabulary challenge are summarized in Table 13.2, which shows model performance in terms of three quantities. Two are the accuracy of predicting basic (u_B) and other (u_O) tokens, where we show the lowest accuracy among the two experiments with different L_c options. Next is the measure (u_S), the accuracy of predicting special tokens which is our primary indicator of domain awareness. Accuracy scores are aggregated over ∼1M iterations for each model-challenge set combination.

Table 13.2: Mean accuracy u[1] on the vocabulary challenge[2]

Model	$\mathbf{u_B}$	$\mathbf{u_O}$	$\mathbf{u_S}$	
			$\mathbf{L_c = 512}$	$\mathbf{L_c = 1024}$
$\mathcal{F}_A$	0.97	0.88	0.67 ± 0.2	0.74 ± 0.2
$\mathcal{F}_B$	0.99	0.95	0.82 ± 0.1	0.87 ± 0.1

[1] Divided by category B-Basic, O-Others, S-Special
[2] On held-out 10% of TAS files

Based upon these results, we make the following observations

1. The vocabulary challenge reveals that the base `tasnet` variant $\mathcal{F}_A$, trained only on GitHub code, is markedly weak in predicting TAS domain keywords. As seen in the table, $\mathcal{F}_A$ shows significantly worse awareness of the domain (∼67%) compared to an excellent proficiency in syntax, and perhaps logic (∼97%). When we tested domain adaptation earlier using a cloze test with simple BERT-masking (Table 13.1), we saw an accuracy of ∼93% in predicting masked TAS tokens. Clearly, the modified cloze test in the vocabulary

challenge, with non-uniform masking, reveals weaknesses in domain understanding which simple BERT masking, with its optimistic results, does not.

2. It is equally clear that $\mathcal{F}_A$ does not drastically mispredict domain keywords. For instance, by correctly predicting ∼67% of domain keywords in TAS, the model shows noteworthy capability in handling key TAS contexts.

3. Doubling the length L_c of the code snippet shows a marked improvement in predicting domain keywords. This is intuitive, since the model has a better chance of understanding and summarizing the domain context if it has access to more information. However, it is important to note that even with a token length of 1024, it is inevitable that domain contexts are unfavorably truncated, adversely limiting the prediction of keywords. This is one key observation that justified the choice of a different architecture in the `tasnet` model in Part I, one that allowed us to increase the context length to 8192.

4. Continued pre-training on the TAS corpus is the single most important factor in improving the prediction of domain keywords. In the `tasnet` variant $\mathcal{F}_B$, the improvement in keyword prediction accuracy from ∼67% to ∼82% is substantial. With double the context length, the accuracy improves to ∼87%.

While continuing to pre-train on the TAS corpus and doubling the context length seem to reveal better domain awareness, one concern is that the variance in the accuracy of predicting domain keywords is significant ($0.1 < 1\sigma < 0.2$). Particularly, if the accuracy of prediction is negatively correlated with the foreground masking rate, it is an indication of weaker domain awareness. Stating this differently using the example in Figure 13.1, such negative correlation is akin to saying that `tasnet` is more likely to predict masked instances of `signal` only if a significant proportion of fellow-special tokens – more likely other instances of `signal` itself – are left unmasked. Ideally, with good domain awareness, even if a significant proportion of keywords are masked, the model should be able to use background information to understand the context. To confirm if foreground masking does indeed have a strong influence in the prediction of domain keywords, we analyze the respective influence of various factors on the prediction accuracy of special tokens. Two of these are the masking rate of special tokens M_S, or the foreground masking rate, and the background masking rate M_{BO}, the combined masking rate of basic and other tokens in the snippet. The third is u_{BO}, the background prediction accuracy. As a simple indicator of influence, over ∼1M trials of the vocabulary challenge on each model, we measure the Pearson correlation coefficients of three test quantities (M_S, M_{BO}, u_{BO}) with the accuracy of predicting of special tokens u_S. Though they capture only linear influence, correlation coefficients shown in Table 13.3 reveals interesting observations.

1. For the base variant $\mathcal{F}_A$ with $L_c = 512$, a correlation of ∼-0.8 with M_S indicates that domain keywords tend to be correctly predicted only when the foreground masking rate is low. That is, when the model does predict correctly, it is mostly due to cues 'leaked' from the foreground instead of context gleaned from the background. A close to zero correlation with both background masking rate and accuracy, M_{BO} and u_{BO} reinforces that prediction

Table 13.3: Correlation[1] of the accuracy of predicting design keywords (u_S) with key test factors[2]

Model	$\mathbf{L_c}$	$\mathbf{M_s}$	$\mathbf{u_{BO}}$	$\mathbf{M_{BO}}$
$\mathcal{F}_A$	512	-0.79	0.09	-0.07
	1024	-0.77	0.09	-0.07
$\mathcal{F}_B$	512	-0.72	0.17	-0.14
	1024	-0.70	0.16	-0.13

[1] Pearson correlation coefficient
[2] L_c: length of code snippet, M_S: foreground masking rate, M_{BO}: background masking rate, u_{BO}: background prediction accuracy

of special tokens is minimally affected either by the quantity of background masking or by the model's accuracy in predicting them. The disproportionate dependence on foreground information indicates a relatively weak and shallow understanding of the domain. Doubling the length of the context to $L_c = 1024$ slightly reduces the influence of the foreground masking. While this minimally favorable development may account for the relatively small improvement in prediction accuracy, foreground dependence remains significant.

2. Upon fine-tuning with TAS, for $\mathcal{F}_B$, correlation with the foreground masking rate drops to ∼-0.7. There is a corresponding increase in background dependence, both on its masking rate and prediction capability. With increased familiarity in the target domain, when the model is challenged with the vocabulary challenge, the model is beginning to depend more on the background to solve it. This reveals better domain awareness. Doubling the context, further induces a slight reduction on the influence of foreground masking. Thus, while this attribution exercise confirms an unfavorable dependence on foreground information, fine-tuning on the corpus and increasing the context size does alleviate it.

In summary, encouraging signs of domain awareness from the vocabulary challenge are that the base variant $\mathcal{F}_A$ is not drastically weak in predicting domain keywords. Increased familiarity in the TAS domain, and increasing the input context seen by the model reveal increased domain awareness. Overall, these observations indicate that the training step that adapts the base variant $\mathcal{F}_A$ to $\mathcal{F}_B$ results in a measurable improvement in domain awareness. Despite these positive observations, it is also clear that there is room for improving the level of domain awareness beyond the ∼87% accuracy of predicting domain keywords.

13.5 Discussion

Subjecting `tasnet` to a vocabulary challenge is one example of how global properties of a foundation model can be understood by subjecting it to a probing test. In this case, the property under investigation is `tasnet`'s comprehension of the TAS domain. The method of investigation is to subject model variants

to a modified version of the cloze test. The modified challenge prioritizes the masking of specially curated domain keywords, and checks if the model under test sufficiently understands the domain context to correctly predict keywords. The test, as analyzed in the previous section, reveals that continuing to pre-train on the TAS corpus does induce significant improvement in domain awareness. We now discuss the implications of this analysis on some important factors.

On design compliance assessment – In many ways, the primary subject of analysis in the vocabulary challenge is the training step (5.9), which adapts variant $\mathcal{F}_A$ to $\mathcal{F}_B$. In this chapter, we have seen that this step induces an improved awareness of the TAS domain, measured by a better comprehension of domain keywords like `signal` and `Rte`. In parallel, as seen in Table 5.2, this training step also results in better performance in the task of design compliance assessment. Taking these parallel observations together, it is plausible to reason that increased domain awareness is a causal factor for improved performance in compliance assessment. At first glance, such a reasoning may seem superfluous but, in order to understand its significance, it is helpful to recount the nature of DECO. In the compliance assessment process conducted in DECO, the predominant focus is upon the arrangement of embeddings. The fact that the algorithm is based upon embedding regularity only serves to amplify its geometrical aspect. This may, however, communicate the impression that it is the arrangement alone, and not the semantic quality of the embedding, that leads to better performance. The vocabulary challenge, however, clearly reveals that adaptation from $\mathcal{F}_A$ to $\mathcal{F}_B$ leads to both a better program embedding, and a better arrangement of embeddings. Thus, rather than either individual factor, it is their combination that is likely to have led to better compliance assessment. The reasoning that increased domain awareness leads to better compliance assessment also correlates with intuition. After all, a human engineer, who possesses a better knowledge of the TAS domain, is likely to better equipped to assess compliance with CH, which is a pattern that regulates key concepts in the TAS domain.

On the pre-training objective – In attempting to reveal domain awareness in `tasnet`, the construction of the vocabulary challenge begins by pointing out a weakness in BERT masking. The simple recipe of uniform randomly masking 15% of tokens specified for BERT, and adopted for `tasnet`, results in a masking regime that is heavily skewed by the frequency of tokens in the pre-training corpus. Since syntactic tokens occur far more frequently than semantically important tokens, BERT masking tends to favor the former. If, as reasoned previously, this leads to weaker assessment of domain awareness during testing, does it not induce an equivalent weakness during training? When training, if semantically important tokens are not masked often, the model is clearly not being sufficiently challenged to predict tokens of import. This, in fact is the stance taken by [187], which proposes a masking recipe that is sensitive to the frequency of token occurrence. Should we not, therefore, deprecate BERT masking in favor of stronger alternatives? The issue, here, is that BERT masking may indeed provide a weaker signal during training, but its overwhelming advantage is that it is simple to apply at scale. Language model pre-training, as we have witnessed, can involve corpora consisting of billions of tokens. Pre-training at such a large scale is well-served by a masking recipe

that is simple and scalable. A masking recipe that is, say, cognizant of token frequency, needs one full pass through the corpus at worst case. While there are means to achieve this more efficiently, considering the scale of pre-training, the cost of incorporating token frequency in the masking recipe, may be substantial. The simplicity of BERT, on the other hand, allow us to tolerate and compensate for some of its weaknesses. In fact, this observation is repeatedly reflected in practice where, even recently announced encoder foundation language models like [188], continue to use the BERT masking recipe for pre-training.

On pre-train and calculate – The vocabulary challenge also has positive implications for the 'pre-train and calculate' paradigm which we employ for unsupervised task specialization. One fundamental component of the paradigm is the availability of a foundation model that faithfully and efficiently represents semantics of inputs from a domain. Stated otherwise, only if the embedding is well-constructed can it be properly applied for predictions using rule-based calculations. A probing test like the vocabulary challenge, which explains the quality of adaptation to the domain of interest, gives an indication to the quality of the semantics captured by the embedding. In the case of `tasnet`, for instance, the fact that the vocabulary challenge reveals a healthy level of adaptation to the TAS domain means that the model represents data in this domain with quality. Embeddings of good quality, as revealed by the performance of the design compliance assessment task, leads to better predictions using rule-based calculations. Simultaneously, the revelation that there is room for improvement in domain adaptation is equally valuable because concerted efforts can be undertaken to address gaps.

Related work – In evaluating `tasnet` using the vocabulary challenge, we reuse several techniques from the area of Natural Language Inference (NLI), a comprehensive survey of which is presented in [189]. Such techniques seek to evaluate (or train) specific aspects of natural language understanding. NLI evaluation, just like our approach, typically involves the curation of a challenge set, a prediction task (including cloze), and an evaluation measure. NLI approaches comparable to ours include [190] which uses the cloze test on curated English language sentences to check the subject-verb agreement. Another example is the story cloze test [191], which is a benchmark for evaluating narrative structure learning. Closer to our setting, [192] tests a bimodal model of natural and programming language by similarly curating and masking a selected set of keywords, though without taking token frequency into consideration.

13.6 Future extension

While the vocabulary challenge reveals useful information about the level of domain adaptation in `tasnet`, it is clear that this challenge merely scratches the surface. Considering that `tasnet` is a foundation language model for automotive software, and the range of tasks to which it can be possibly applied, many more benchmarks are required to better characterize its properties. An inspiring benchmark that we can look up to would be HELM [118], which is arguably the current gold-standard for evaluating large language models. By the sheer num-

ber of properties that it evaluates, it sets a high bar to emulate. If not HELM which, after all, tests general natural language understanding, we can instead turn to comparable domain-specific benchmarks. Good examples are the ones used in [193] for evaluating a language model for finance, [194] which measures comprehension of biomedical language, and [195] which benchmarks a variety of coding tasks. In startk contrast to any of these examples, there is clear lack of sufficient benchmarks for evaluating models of automotive natural and programming language. More generally, the fact that language modeling in the automotive domain is very much in its infancy means that substantial work is needed to identify and define benchmarks in this area. Many challenges are likely to crop up when attempting to define such benchmarks. Information modalities used in automotive software engineering go beyond just code and accompanying natural language documentation. Code, in automotive software engineering, exists in an ecosystem that includes formal descriptions of software architecture, test cases at different levels of integrations, ECU configurations, vehicle configurations, etc. A holistic benchmark of language models of vehicle software thus needs to holistically consider the larger multidisciplinary world in which software is developed. Developing such a holistic benchmark and, thereby, a language model that spans the breadth of knowledge in the automotive domain is a crucial avenue for future work. In undertaking this long, yet ultimately useful and necessary journey, a test like the vocabulary challenge is a starting point as healthy as any.

Chapter 14

Explaining sample representation in data

The previous chapter describes a technique which can be used to explain an abstract high-dimensional space of representations, but this chapter considers explaining a different high-dimensional space – data. A large dataset may not always be abstract but, in the deep learning context at least, is almost always high-dimensional. The code corpora used to train `tasnet` consists of millions of files, with thousands of tokens each. Comparably, the dataset of signal traces used to train `xgan` contains hundreds of thousands of minutes-long multi-variate time series. Surely, these are very good examples of large high-dimensional datasets. In training any DNN, a fundamental reality is that the choice of the 'right kind' of data plays an important role in ensuring that the model, with which this data is trained, is able to generalize. For instance, having trained `tasnet` on GitHub and TAS corpora, can we expect it to generalize to software engineering tasks in industrial process control? We could perhaps venture an answer to this question if we knew whether code from the process control domain is included in the training corpus. Setting aside the difficulty in drawing firm boundaries between domains, when it comes to a specialist corpus like TAS, we can conclude with high confidence that it does not include process control code. Can we come to a similarly confident conclusion for the GitHub corpus that contains hundreds of millions of files of source code? Clearly difficult. As training corpora in different domains rapidly expand, understanding the diversity of data that they represent becomes a challenge. Without a firm grip on the diversity represented in data, it is not easy to reason about the generalization of models trained with this data.

In order to help understand sample representation in datasets, in this chapter, we present a study that examines the diversity of representation in a reasonably large dataset. Departing from our world of software engineering and its information domains of source code and signal traces, we conduct the study on a dataset of simple geometric images of hand-drawn circles and squares. The rationale behind the departure is that the relative simplicity of information contained in this

image dataset provides a measure of control in conducting the study. Forthcoming sections present our experiments and findings, after which we discuss implications and extensions to information domains in automotive software engineering.

14.1 Interpretable assessment of sample representation

Let us begin with a brief illustration of concerns in sample representation using the proximate application area of driver assistance or self-driving functionality. Relying upon images from cameras is standard practice for applications in this area. Consider, therefore, a traffic dataset $\mathcal{S}$ of images $X_i \sim P(X|Y)$ and annotations $Y_i \sim P(Y)$. A major practical concern in such datasets is whether it adequately represents corner cases like intersections with stop signs, roundabouts with five exits, etc. With the true/target distribution of traffic scenes $P(X, Y)$ clearly containing instances of such cases, *any under-representation* in $\mathcal{S}$ can be broadly framed as shortcomings in data collection and processing, otherwise known as *sample selection bias* [196]. Given that the dataset is eventually used to train a model that is deployed in a safety-critical system, engineers may actively seek to properly comprehend and account for such bias. But how does one express such bias in human interpretable terms? One clue comes from annotations $Y_i \sim P(Y)$. In typical traffic datasets, Y encodes object class labels and bounding box positions. If necessary and feasible, Y can be expanded to contain information such as location, lighting conditions, weather conditions, etc. When Y is adequately detailed, the distribution of annotations $P_{\mathcal{S}}(Y)$ clearly becomes a reasonable, low-dimensional, and therefore a human interpretable measure of sample representation in $\mathcal{S}$. Now, such a notion can also be reversed to *specify* a distribution of annotations $P_{\mathcal{T}}(Y)$, expressing the sample representation that is *expected* in the dataset. While the target distribution of annotations $P(Y)$ may be unknowable, $P_{\mathcal{T}}(Y)$ is an explicit declaration of the sub-space that the dataset is expected to cover at the minimum. If $\mathcal{S}$ is equivalently labeled, then selection bias (and thereby sample under-representation) is simply given by the mismatch between expectations $P_{\mathcal{T}}$ and reality $P_{\mathcal{S}}$. In practice, however, due to the effort and expense involved in labeling, $\mathcal{S}$ may either lack labels or may be completely unlabeled, meaning that $P_{\mathcal{S}}(Y)$ is often unavailable. Combining simulation, outlier detection, and input attribution, we show that it is possible to explain sample representation in a comprehensible low-dimensional form, even when annotations are not explicitly available in $\mathcal{S}$.

Delving into the less-explored area of *explaining* sample representation in a dataset, this chapter demonstrates a method that (1) explains sample representation in interpretable terms for annotated data, and (2) uses parametric simulation and outlier detection to do the same for non-annotated data. Such explanations helps engineers better understand data as a crucial ingredient of the training process. Downstream, this helps them re-asses data collection methods and to verify, reason, or argue about – at times a requirement for standards compliance [197] – the overall dependability of models trained with this data.

14.2 Explaining sample representation using annotations

Visualizing sample representation – We now introduce a simple running example of examining sample representation in a dataset $\mathcal{S}$ containing images of two hand-drawn shapes[1] – circles and squares (Figure 14.1). With the shape as the sole available label, one can define $\mathcal{S} = \{(X_i, Y_i^1)\},\ i = 1...N$, where X_i is a grayscale image of size $(128, 128)$ and $Y_i^1 \in K = \{0, 1\}$ is the shape label, corresponding to circle and square respectively. Understanding sample representation in this dataset may be necessary when it is a candidate for training a model that, for example, either recognizes or generates shapes. To ensure dependable model performance, system designers may want to confirm that images of adequate variety are represented in $\mathcal{S}$. In a dataset of grayscale geometric shapes, it is intuitive to analyze sample representation in terms of concerns such as the size and position of the shapes on the image canvas, and the average brightness of pixels in the shape. All these concerns can be captured by defining a 6-d annotation vector $Y = (Y^1, ..., Y^6)$, including shape-type, which is known. With $\mathcal{U}$ denoting the discrete uniform distribution, designers can begin with defining an expected spread of shape-size using a latent label $Y^S \sim \mathcal{U}\{30, 120\}$, denoting the side-length in pixels of a square box bounding the shape. This can be followed by defining expectations on the spread of (i) the top-left corner of the bounding box, $Y^2, Y^3 \sim \mathcal{U}\{0, 128 - Y^S\}$, (ii) the bottom-right corner of the bounding box $Y^4, Y^5 \sim \mathcal{U}\{Y^S, 128\}$, and (iii) the average pixel brightness $Y^6 \sim \mathcal{U}\{100, 255\}$. Put simply, $P_{\mathcal{T}}(Y)$ expects shapes of a specified range of sizes and brightness to be uniformly represented in the dataset $\mathcal{S}$. All positions are also expected to be uniformly represented, as long as the shape can be fully fit in the image canvas.

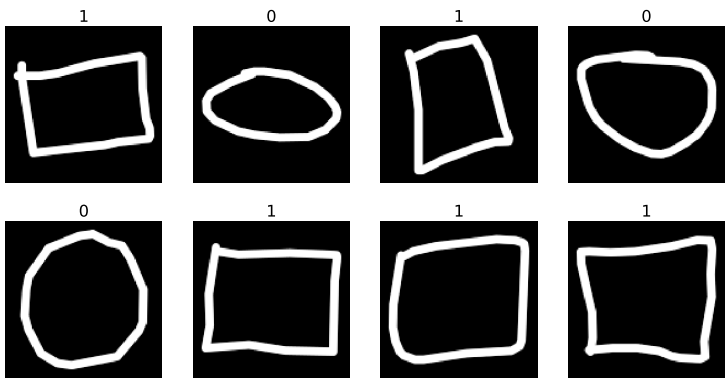


Figure 14.1: Samples from the dataset $\mathcal{S}$. Only the class label Y^1 is available

To illustrate the idea of explaining sample representation using annotations, an automatic labeling scheme $Y_i = L(X_i)$ is used to produce complete 6-d annotations for X_i. For circles and squares, it is easy to define a scheme that looks at the extent of the shape and draws bounding boxes. The average brightness is

[1] Collected from Quick, Draw! with Google – https://quickdraw.withgoogle.com/data

given by the mean of non-zero pixels in the canvas. The availability of labels Y_i helps assemble the actual distribution of samples in the dataset $P_{\mathcal{S}}(Y)$, allowing direct comparison with expectations $P_{\mathcal{T}}(Y)$. Jointly visualizing label distributions for each shape (Figure 14.2) shows that, along all design concerns Y^j, the spread of $P_{\mathcal{T}}$ (marked black) is much wider than the very narrow $P_{\mathcal{S}}$ (marked red). This shows that, while $P_{\mathcal{T}}$ expects shapes of a broad range of sizes, positions and brightness to be represented, $P_{\mathcal{S}}$ is clearly biased and massively over-represents large and bright shapes located in the center on the canvas. As long as the annotation vector Y is of manageable length, joint visualization becomes an interpretable qualitative explanation of sample representation in the dataset.

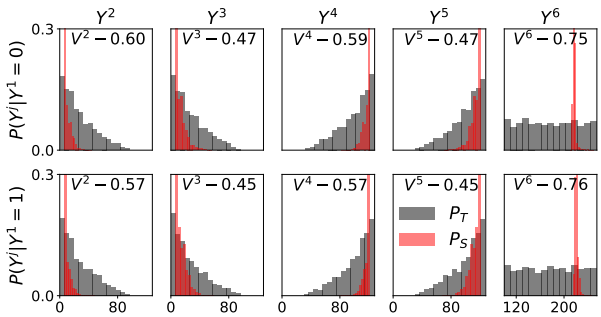


Figure 14.2: Explaining sample representation

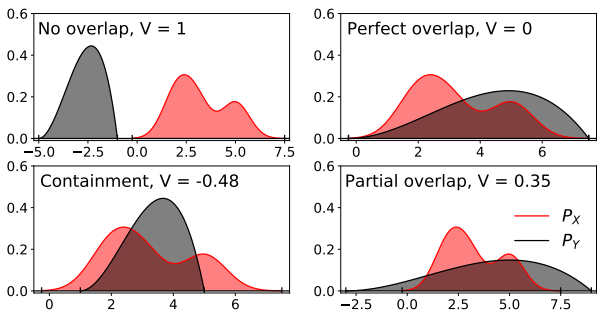


Figure 14.3: Illustration of $V(P_X, P_Y)$

Quantifying sample representation – By framing sample selection bias, and thereby sample under-representation, as the mismatch between expected and true label probability distributions, it becomes possible to quantify it using measures of statistical similarity. Choosing the right measure, however, requires a proper understanding of the nature of each distribution. Having calculated it using true labels of each sample, it is clear that $P_{\mathcal{S}}(Y)$ represents the actual sample distribution in $\mathcal{S}$. The distribution of expectations $P_{\mathcal{T}}$ is of a slightly different nature and, to better understand it, let us consider the expectation $P_{\mathcal{T}}(Y^6) = \mathcal{U}\{100, 255\}$, placed on the representation of average brightness of shapes in the dataset. While the expectation on brightness being spread between specified lower and upper limits is strict, imposing the spread to be uniform is arbitrary. This is a deliberate measure of simplification to ease the

considerable burden in modeling expectations $P_{\mathcal{T}}$, and let it simply convey the critical range of interest in the target distribution. Put simply, expected sample representation is primarily encoded by the *support* (14.1) of $P_{\mathcal{T}}$. By specifying strict support, but arbitrary distribution of mass, sample representation can be quantified as the level of *overlap* between the actual sample distribution $P_{\mathcal{S}}$ and the expected sample representation $P_{\mathcal{T}}$. To achieve this, we propose an overlap index $V(P_X, P_Y)$ (14.2), which is a measure of whether the supports of two distributions are similar. Denoting set difference as the operator Δ and 1-d Lebesgue measure[2] (length) of a set as λ, V is essentially the Steinhaus distance [198] with an added term I to make $-1 < V < 0$ indicate containment of P_Y within P_X. When not contained, for some positive likelihood in both distributions, as illustrated in Figure 14.3, $V = 0$ when they exactly overlap, $V = 1$ when they do not overlap, and $0 < V < 1$ when the overlap is partial. Indices $V^j(P_{\mathcal{T}})$ (14.3) quantitatively measure the level of overlap between true and expected distributions for each label. Complementing the visual explanation, overlap indices $0.4 < V^j(P_{\mathcal{T}}) < 1$ seen in Figure 14.2, indicate that there is only slight partial overlap between expectations and reality, confirming notable sample selection bias and, therefore, significant sample under-representation.

$$R_X = \{x \in \mathbb{R} : P_X(x) > 0\} \tag{14.1}$$

$$V(P_X, P_Y) = I\ \frac{\lambda(R_X\ \Delta\ R_Y)}{\lambda(R_X\ \cup\ R_Y)},\quad I = \begin{cases} -1 & R_Y \subset R_X \\ +1 & otherwise \end{cases} \tag{14.2}$$

$$V^j(P) = V(P_{\mathcal{S}}(Y^j\ |Y^1), P(Y^j\ |Y^1)),\quad j = 2...6 \tag{14.3}$$

It is therefore clear that, given the expected representation and actual distribution of labels in the dataset, it is possible to comprehensibly explain sample under-representation both visually and quantitatively. However, the overlap index, which eschews mass and uses only support, is an incomplete measure of sample selection bias, the pros and cons of which we discuss later.

14.3 Explaining sample representation using simulation

The dataset $\mathcal{S}$ contains information X_i in the image domain, while lacking information Y_i in the annotation domain. Expectations, on the contrary, are expressed using annotations $\hat{Y}_i \sim P_{\mathcal{T}}(Y)$, but lacks images. It is this gap in information that prevents estimation of sample under-representation by direct comparison. There are two possible ways to bridge this gap, one of which is the labeling scheme $Y_i = L(X_i)$ introduced earlier. Another way could be to use the labels as parameters to generate images $\hat{X}_i = G(\hat{Y}_i)$, which is essentially *parametric simulation*. In this case of circles and squares, it is possible to use a graphics package[3] to draw shapes using size, position, and brightness labels as parameters. We, in fact, choose this simple dataset because both labeling and simulation of samples are easy, helping illustrate both ways of bridging the

[2] https://en.wikipedia.org/wiki/Lebesgue_measure

[3] We use OpenCV – https://opencv.org/

gap and cross-checking the plausibility of estimating sample representation. In many practical cases, however, the right method to bridge the gap is difficult to judge since the relative expense is domain and problem specific. Addressing those instances where unlabeled data is available and labeling is expensive, we now show that it is possible to bridge the gap using simulation. This is done using a two-step process, described below, of (i) detecting outlier annotations and (ii) estimating marginal sample representation.

Step 1 - Detecting outlier annotations – Let us consider a simple question – to a dataset that mainly contains large, centered shapes, do simulated small off-centered shapes appear as outliers? In order to explore the essence of this question, we pose the following outlier hypothesis.

A test annotation $\hat{Y}_i$, that is unlikely to be observed in $\mathcal{S}$, maps to a simulated test sample $\hat{X}_i = G(\hat{Y}_i)$, that appears as an outlier to $\mathcal{S}$.

If the outlier hypothesis does hold, the problem of detecting sample selection bias turns into one of detecting outlier images. In order to test this hypothesis, we construct an outlier detector $E_{\mathcal{S}}$ (Figure 14.4) that samples test annotations from $P_{\mathcal{T}}$ and maps them into images using a simulator, creating a test set $\mathcal{T} = \{(\hat{X}_i, \hat{Y}_i)\},\ i = 1...M$ (examples in Figure 14.5). We may now have simulated samples, but how do we check if they appear as outliers to the real dataset $\mathcal{S}$? Following [199], we simply use a classifier $F(X) = P_{\mathcal{S}}(Y^1|X;\theta)$, trained on the dataset $\mathcal{S}$ as the core means of detecting outliers. At the outset, such a classifier F takes an input and predicts if it is a circle or a scare. Now, if a simulated circle or square image $\hat{X}_i$ is applied as input, and if it actually happens to be an outlier, the essence of the findings in [199] is that the classifier F should predict the label of the outlier input with less certainty. In this case of a binary classifier F, the output is a 2 dimensional vector $F(\hat{X}_i) = [S^0, S^1]$, such that $S^0 + S^1 = 1$. Here, S^0 is the classification score of the input being a circle, while S^1 is its assessment of the input being a square. A simple sign of predictive uncertainty is a prediction $F(\hat{X}_i) = [0.5, 0.5]$, when the classifier provides no clear indication of the category to which the input belongs. A lack of certainty makes it likely that the input $\hat{X}_i$ itself is an outlier, which is an observation that is clearly useful for testing the outlier hypothesis. Following this reasoning, the complete detector of outlier annotations $E_{\mathcal{S}}$ is formally described below in (14.4).

$$S_i = E_{\mathcal{S}}(\hat{Y}_i, F, T) = \max_{k \in K} \frac{exp(F_k(G(\hat{Y}_i))/T)}{\sum_{k \in K} exp(F_k(G(\hat{Y}_i))/T)}, \hat{Y}_i \sim P_{\mathcal{T}}(Y), K = \{0, 1\} \tag{14.4}$$

$$\hat{Y}^- = \{\hat{Y}_i : P_{\mathcal{S}}(\hat{Y}_i) = 0\}, \qquad \hat{Y}^+ = \{\hat{Y}_i : P_{\mathcal{S}}(\hat{Y}_i) > 0\} \tag{14.5}$$

Here F_k is the logit score for the k^{th} label and T is the temperature parameter which, as shown later, eases the detection process. With F using a softmax output layer to produce the 2 dimensional prediction, we use maximum softmax score as the measure of certainty. If this maximum softmax score is close to 1, it means that the classifier is certain about its prediction, and the input is

unlikely to be an outlier. Contrarily, if the maximum score is 0.5, then the model is less certain and the input is likely to be an outlier. Put simply, with sets of outlier and familiar annotations (14.5), the outlier hypothesis asserts that a good detector $E_{\mathcal{S}}$ assigns low scores S_i for outlier annotations $\hat{Y}^-$ and high scores for familiar ones $\hat{Y}^+$.

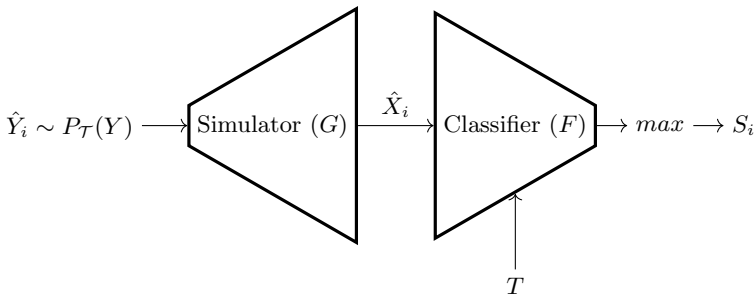


Figure 14.4: Detecting outlier annotations

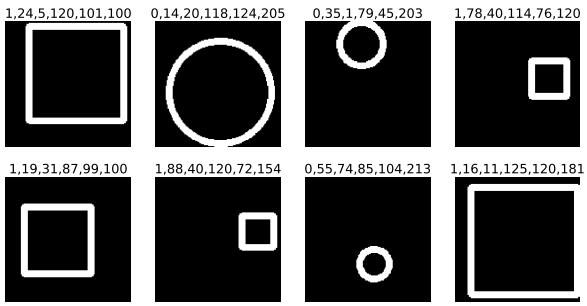


Figure 14.5: Samples $\hat{X}_i$ from the test set $\mathcal{T}$

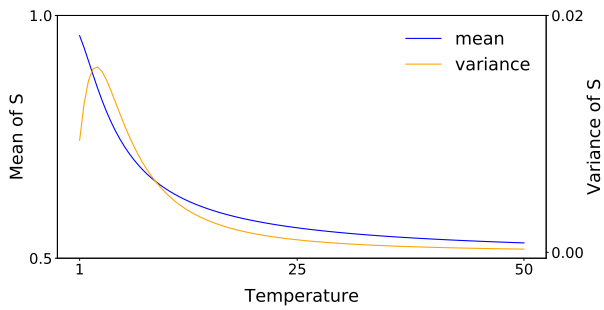
To test the outlier hypothesis, we train four variants of the classifier F, all of which follow the VGG architecture [200]. The variants mainly differ in the number of layers, with VGG05 (5 layers) and VGG13 (13 layers) being the shallowest and deepest respectively. Each F is trained[4] for 5 epochs on $\mathcal{S}$ with 50k samples using the Adam optimizer [138] to achieve validation accuracy (on a separate set of 10k samples) greater than 97%.

While measures of certainty are central to our process of outlier detection, [201] shows that deep neural nets tend to predict with high confidence. That is, if a sample $\hat{X}_1$ is not an outlier and evaluates to a certainty score of, say, $S_1 = 0.993$, an outlier $\hat{X}_2$ could evaluate to a score $S_2 = 0.989$. In this case, the model is indicating that $\hat{X}_2$ is indeed an outlier, but by a very narrow margin. Considering the emphatic confidence with which neural networks normally predict, raw maximum softmax scores are poor measures of predictive certainty. One a simple way to mitigate this is temperature scaling, i.e. setting $T > 1$, in (14.4). As seen in Figure 14.6(a), scores S_i are tightly clustered at $T = 1$ with relatively low variance, which makes it difficult to identify differences in predictive certainty

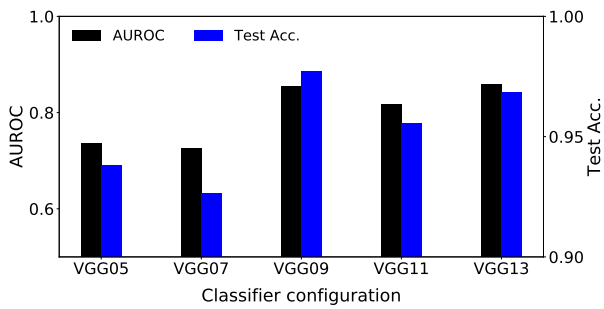
[4]Each classifier trains within 10 – 15 minutes on an NVidia GTX 1080 Ti GPU

between familiar and outlier annotations. There is, however, a range of temperatures at which scores are better spread and can exaggerate these differences. While a temperature that maximizes the variance of the score distribution seems appropriate, as seen in Figure 14.6(a), scaling also reduces its mean. Therefore a safeguard may be necessary to prevent the mean certainty score from reducing to a level that questions the confidence of predictions. These twin requirements can be achieved by the search objective (14.6), which ensures a good spread in scores S_i, while keeping its mean close to the chosen safeguard S^T.

$$
\begin{aligned}
T^* &= \underset{T}{argmin}\ L^T - L^V, \quad L^T = \left(\mu_S - S^T\right)^2 \\
L^V &= \frac{\sum_{i=1}^{M}\left(E_{\mathcal{S}}(\hat{Y}_i, F, T) - \mu_S\right)^2}{M}, \quad \mu_S = \frac{\sum_{i=1}^{M} E_{\mathcal{S}}(\hat{Y}_i, F, T)}{M}
\end{aligned}
\tag{14.6}
$$



(a) Effect of temperature scaling on the distribution of uncertainty scores F=VGG13



(b) AUROC for detecting outlier annotations per classifier at $T = T^*$ and $S^T = 0.7$

Figure 14.6: Testing the novelty hypothesis

Upon temperature scaling with T^*, which exaggerates the spread of certainty scores, the effectiveness of the detector $E_{\mathcal{S}}$ in separating outlier annotations $\hat{Y}^-$ from familiar ones $\hat{Y}^+$ can be measured using the Area Under Receiver Operating Characteristic (AUROC). This is shown for each F, averaged over 5 separate training runs, in Figure 14.6(b). Based on an informal grading scheme for classifiers using AUROC score suggested in [199][5], detectors using

[5]Quality of classification based on AUROC score - 0.9—1: Excellent, 0.8—0.9: Good,

VGG05 and VGG07 receive a 'fair' grade in identifying outlier annotations, while the deeper networks get 'good' grades. The best outlier detectors, with AUROC ≈ 0.85, are those with F as VGG09 and VGG13. These results clearly endorse the viability of the outlier hypothesis that simulated images that are under-represented in $\mathcal{S}$, in terms of specified design concerns, appear as outliers to the right classifier trained on $\mathcal{S}$. While $P_{\mathcal{S}}$, derived from labeling, is used as a benchmark to test the outlier hypothesis, it is important to observe that (i) classifiers that are good at outlier detection are, as seen in Figure 14.6(b), those that have the highest accuracy in predicting shape labels on the test set $\mathcal{T}$, and (ii) the temperature T^*, at which the classifiers become good outlier detectors, depends only upon the statistical properties of scores S_i. Together, these observations mean that a good detector of under-represented annotations can be assembled using only simulation, without any need for labeling.

Step 2 - Estimating marginal sample representation – As presented in Section 14.2, we seek to comprehensibly explain sample representation in the dataset $\mathcal{S}$ of geometric shapes on the basis of intuitive design concerns like size, position, and brightness. However, the detector $E_{\mathcal{S}}$ that we have constructed can only assess whether a single combined 6-d test annotation is an outlier. It is much more useful if we were to break down the sample representation and explain it in terms of all required design concerns. To independently assess sample diversity in terms of each label, we turn to techniques of input attribution. Given the detector $E_{\mathcal{S}}$, attribution techniques estimate the contribution of each input label $\hat{Y}_i^j$ to its outlier score S_i. Among proposed methods for input attribution [202], one promising framework is Shapley Additive Explanations (SHAP) [182]. Using principles of cooperative game theory, SHAP estimates *marginal influence* ϕ_i^j (14.7), which indicates how label $\hat{Y}_i^j$ independently influences the uncertainty score S_i.

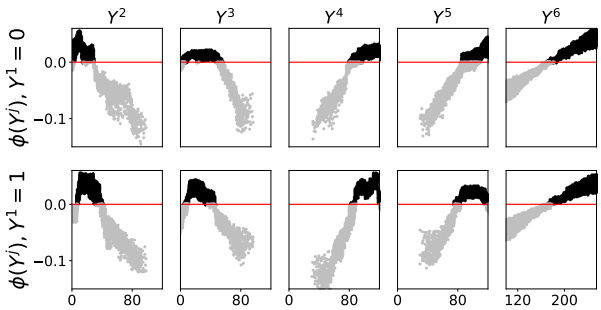
$$S_i = E_{\mathcal{S}}(\hat{Y}_i, F, T) = \phi^0 + \sum_{j=2}^{6} \phi_i^j \tag{14.7}$$

In satisfying an additive property, SHAP values are also semantically intuitive. A negative SHAP value ϕ_i^j indicates that the label $\hat{Y}_i^j$ has a negative influence on the score S_i. Correspondingly, a positive or zero value of ϕ_i^j indicates positive or neutral influence on the score. For instance, let $\hat{Y}_1^6 = 255$ be the pixel brightness of the simulated circle $\hat{X}_1$. If this label maps to a positive SHAP score, it means that an increase in the certainty of predicting this simulated shape, in relation to all other predictions, can be attributed this particular value of brightness. According to the outlier hypothesis whose validity we just confirmed, this means that it is likely that bright circles are highly likely to be represented in the dataset $\mathcal{S}$. This is precisely the kind of explanation of sample representation that is both clear and useful. Therefore, more generally, SHAP value $\phi_i^j > 0$, which indicates that the individual label value $\hat{Y}_i^j$ tends to improve S_i, becomes an indicator of that label being represented in $\mathcal{S}$. Through a campaign directed by the test set $\mathcal{T}$, which systematically covers the specified range of scenarios $P_{\mathcal{T}}$, non-negative SHAP values identify sample representation in the dataset $\mathcal{S}$ in terms of

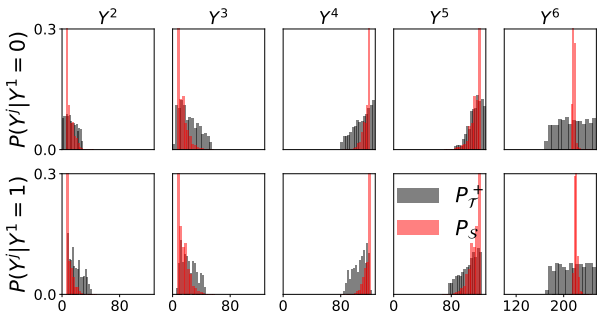
0.7—0.8: Fair, 0.6—0.7: Poor, 0.5—0.6: Fail

each individual label. This can be seen in Figure 14.7(a), where label values with a high incidence of non-negative SHAP values (marked black) are likely to be represented in $\mathcal{S}$. Thus, by simply measuring the incidence of non-negative SHAP scores, we are able to explain that the dataset $\mathcal{S}$ mainly represents relatively large, centered, and bright circles and squares. Put otherwise, we have arrived at a comprehensible explanation that small, off-center, and dim shapes are not well represented in $\mathcal{S}$. Further, in order to express it as a distribution, we convert a SHAP based measure into a likelihood. The likelihood of test label $Y^j = l$, $Y^j \sim P_{\mathcal{T}}$ being represented in the set $\mathcal{S}$ is simply the proportion of test labels $\hat{Y}_i^j$, in a sufficiently small interval δ around l, whose SHAP values are non-negative.

$$P_{\mathcal{T}}^{+}(Y^j = l \mid Y^1 = k) = \frac{|\{\hat{Y}_i^j : \phi_i^j \geq 0,\ \hat{Y}_i^j \in Y^l\}|}{|\{\hat{Y}_i^j : \phi_i^j \geq 0\}|},\ j = 2...6,\ \hat{Y}_i \in \hat{Y}$$
$$Y^l = \{l - \delta, l + \delta\}, \qquad \hat{Y} = \{\hat{Y}_i : \hat{Y}_i^1 = k\},\ k \in K \tag{14.8}$$



(a) Sample-representation from SHAP scores



(b) Marginal sample representation

Figure 14.7: Explaining sample representation using simulation (F=VGG13, $T = T^*$, $S^T = 0.7$)

Assessing the explanation – By expressing expected diversity $P_{\mathcal{T}}$ in terms of specified design concerns, we thus use a two-step process to identify sample representation. Using a simulated test set, we calculate representation using non-negative influence on predictive certainty. From the original broadly spread expectations $P_{\mathcal{T}}$ (Figure 14.2), the process correctly eliminates a significant amount of outliers in each label dimension, producing $P_{\mathcal{T}}^{+}$ (Figure 14.7(b)).

$P^+_{\mathcal{T}}$ shows label values likely to be observed in the dataset $\mathcal{S}$ and has a roughly similar spread as the actual distribution $P_{\mathcal{S}}$. Also, using a test set with M=10k samples, the process estimates sample representation in a much larger dataset with N=50k samples. Introduced originally in Section 14.2 to quantify bias between expected and actual distributions of annotations, the overlap index V is also suitable for measuring similarity between $P^+_{\mathcal{T}}$ and $P_{\mathcal{S}}$. This helps quantify the effectiveness of estimating sample representation using simulation. The visual observation that $P^+_{\mathcal{T}}$ is a better estimate of true sample distribution, compared to the broad range of expectations $P_{\mathcal{T}}$, is confirmed by better a mean overlap score $V^j(P^+_{\mathcal{T}})$ (see Table 14.8), over all labels and shapes, compared to mean $V^j(P_{\mathcal{T}})$. While this holds true for both classifier instances shown in the table, the detector using F=VGG13 at $T = T^*$, which has the best AUROC score in detecting outliers, produces the closest estimate with a mean overlap score of 0.27. VGG05, with poorer AUROC, has a weaker average overlap score of 0.39. The close correlation between AUROC and V further confirms the plausibility of estimating marginal sample representation using SHAP scores. This shows that, while facing an expensive labeling process, with the right means of parametric simulation, one can conduct a campaign from a low-dimensional space of specified design concerns to estimate sample representation in a given dataset and comprehensibly explain sample selection bias.

14.4 Discussion

Under-representation and outlier detection – A good outlier detector $E_{\mathcal{S}}$ of under-represented samples must blur the distinction between simulated and real images while emphasizing the distinction between over and under-represented images. Figure 14.6(b) shows both conditions are jointly achievable, with classifiers that have a high test set accuracy, and therefore generalize well, also having better AUROC scores in detecting representation. However, as seen in Figure 14.9, using regularization measures like batch normalization layers after each convolutional block, while improving test accuracy, reduces AUROC scores for all classifier instances. This is probably because it tends to blur [203] both forms of distinction. The figure also shows that dropout increases the test accuracy without any major effect on AUROC scores, giving no special domain separation advantage in detecting under-representation. Among the classifier configurations investigated here, vanilla VGG, with the strongest correlation between AUROC and test set accuracy, is observed to best addresses both forms of domain distinction.

The importance of effective simulation – It is crucial to note that high test accuracy reflects the combined effect of plausible simulation and good generalization. It is equally essential, therefore, that the simulator produces samples that are plausibly real. Ensuring effective simulation, while supporting a variety of parameters, is undoubtedly a challenge for realistic datasets with richer content. As noted earlier, while this is domain and problem dependent, for images at least, rapid advancements in the quality and range of graphics tools ([204], [205]), potentially makes effective simulation plausible. However, with no-

T	P	Y^1	$V^j(P)$					Mean $V^j(P)$
			j=2	3	4	5	6	
-	$P_{\mathcal{T}}$	0	0.60	0.47	0.59	0.47	0.75	0.57
		1	0.57	0.45	0.57	0.45	0.76	
T^* $S^T = 0.7$	$P^+_{\mathcal{T}}$ F =VGG13	0	0.49	0.14	0.17	0.35	0.55	0.27
		1	-0.19	0.26	0.16	0.17	0.56	
	$P^+_{\mathcal{T}}$ F =VGG05	0	0.47	0.33	0.29	0.44	0.60	0.39
		1	0.31	0.34	0.27	0.25	0.56	
1	$P^+_{\mathcal{T}}$ F =VGG13	0	0.49	0.30	0.40	0.15	0.69	0.36
		1	0.30	0.28	0.21	0.13	0.69	
	$P^+_{\mathcal{T}}$ F =VGG05	0	0.22	0.14	0.54	0.47	0.65	0.43
		1	0.57	0.29	0.56	0.15	0.70	

Figure 14.8: Quantitative bias estimation

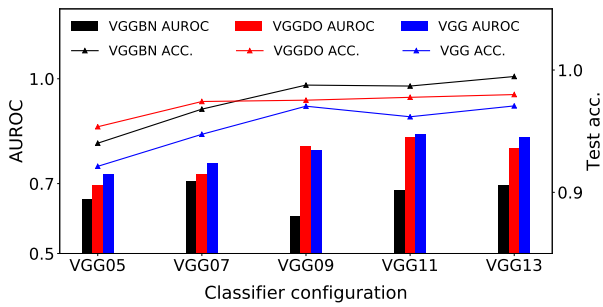


Figure 14.9: Effect of regularization on AUROC

table progress in techniques that automate parts of the labeling process [206], it is also important to assess whether labeling is cheaper for the dataset in concern.

Improving estimation of representation – Figure 14.7(b) shows that while the estimated sample representation $P^+_{\mathcal{T}}$ comes close, it does not overlap perfectly with the true label distribution $P_{\mathcal{S}}$. As quantified in Table 14.8, even the best detector (F=VGG13 at $T = T^*$) has a mean overlap index of 0.27 indicating relatively close, but only partial, overlap on average. At the individual label level, index values show varying accuracy in support-matching. The representation of pixel brightness $0.5 < V^6(P_{\mathcal{T}}) < 0.8$ is consistently underestimated, while those of bounding box coordinates are better estimated. It is however clear from Table 14.8 that temperature scaling ($T = T^*$ vs 1) and deeper classifiers (F =VGG13 vs VGG05) improve estimation, indicating that more sophisticated techniques of predictive outlier detection, like methods in [207], can improve estimation.

Balancing detail in specifying expectations – The level of detail specified in the expectations $P_{\mathcal{T}}$ plays a key role in deciding the cost and benefit of explaining sample representation. An overly detailed breakdown of design factors involves significant engineering effort, degrades interpretability, and overlooks the remarkable benefits of generalization offered by deep learning. But well-balanced expectations can provide valuable insight into training data. Take an application like self-driving vehicles, where engineers actively seek a certain level of understanding of operational scenarios [208] to ensure safe

operation. Such understanding can be exploited to systematically explain, analyze, and manage the data used to train models deployed in the system, thereby improving overall confidence in its dependability. While balancing details in the specification may not always be easy, one advantage of this method is that it is semi-supervised. Annotations included in the analysis impacts only the simulated test set $\mathcal{T}$ and has no effect on the actual dataset $\mathcal{S}$.

14.5 Related work

Sample selection bias – Sample selection bias has been addressed in existing literature from the perspective of domain adaptation [209]. Previous methods to mitigate sample selection bias have mainly attempted to modify the training procedures or the model itself to yield classifiers that work well on the test distribution. Methods such as importance re-weighting [210], minimax optimization [211], kernel density estimation [212] and model averaging [213] all fall in this category. While these methods can yield classifiers that are able to generalize, the accuracy can suffer when the two distributions differ greatly in the overlap of their support or in the distribution of their mass. Our immediate goal, on the other hand, does not seek to obtain a classifier that generalizes, but instead we seek to obtain a high level *understanding* of the deficiencies of our training data and where the bias stems from. This goal does not necessarily require a full specification of $P(Y)$, instead we work with the weak proxy of $P_{\mathcal{T}}(Y)$ which attempts to match $P(Y)$ only through the support. However, by eschewing mass-modeling, we gain a few advantages, one of which is the reduced effort in defining expectations. More importantly, since several existing methods for correcting sample selection bias work only if the support of $P_{\mathcal{T}}$ is included in that of $P_{\mathcal{S}}$ and our method of explanation tests precisely for this condition. Overlap indices $V^j(P_{\mathcal{T}}) \leq 0$ guarantees that the support of the biased distribution includes that of the expectations and correction measures like importance re-weighting are applicable. If $0 < V^j(P_{\mathcal{T}}) \leq 1$, expanding the diversity of data collection is unavoidable. Thus seeking to understand and explain the data set can allow for an improved understanding of the validity for methods that directly impacts the generalization performance.

Understanding sample representation – Besides clustering approaches [214] and feature projection methods such as t-SNE [215], previous research into providing a high level understanding of the training set has, for example, applied tree-based methods to detect regions of low point density in the input space [216]. High-dimensional explanations in the input space, however, adversely affects interpretation, and ways to extend these methods to yield explanations using an interpretable low-dimensional space of annotations are not immediately clear.

Bias estimation using simulation – Closer to our purpose are the methods [217] and [218] which detect inherent biases in a trained model using parametric simulation and Bayesian optimization. While their goal is to find input samples where the model is locally weak, our goal is to ensure that a given dataset meets global expectations defined by a test set. This can verify

that a system is dependable for all considered scenarios, like [219], which is a standardized set of tests. However, in reformulating bias detection as outlier detection, our method – unlike the aforementioned methods – trades-off the ability to detect unknown unknowns [220] in favor of a faster, global evaluation of bias. Combining our global and their local approaches may, therefore, help ensure better overall dependability.

Shapley-based outlier detection – Previous work using Shapley values for outlier detection, such as [221] and [222], focus mainly on providing interpretable explanations for why a data point is considered to be an outlier. It may also be possible to extend their data-space explanations to the annotation-space, like we do, using parametric simulation. However, pixel-wise reconstruction error has well-known drawbacks in capturing structural aspects of data [116]. It is therefore not immediately clear whether their use of auto-encoder reconstruction error is as good at detecting structural under-representation as our technique of using predictive certainty, which is calculated from the feature space of a classifier.

14.6 Future extension

We may have demonstrated a technique to explain sample representation in a dataset of hand-drawn shapes, but the larger question is whether such a technique viably extends to the domains and datasets that we have used to solve tasks in automotive software engineering. Since we use two domains of information, code and signal traces, to train two kinds of foundation models, `tasnet` and `xgan`, we analyze extending this technique to either domain.

Extension to signal traces and xgan – The training corpus used to train `xgan` contains a few hundred thousand signal traces drawn from a set of vehicles. In this dataset, however, we do not have a clear idea whether trips of sufficient variety are represented. For instance, it may be of interest to know if there are enough trips with vehicles making rapid starts and stops, both on flat roads and on winding roads that go uphill. Such scenarios may be of interest because of, say, the sensitivity of the engine start functionality under repeated use. Without explicit labels, it can be challenging to identify whether these kinds of trips are available in the dataset. The sample representation technique that we present in this chapter can conceivably be extended to measure such representation. One crucial element of our explanation technique is the availability of a discriminator trained on the data that is being scrutinized. Fortunately, in `xgan`, we in fact have a readily available option – the discriminator of the GAN itself. In the adversarial learning objective used to train `xgan`, the discriminator is specifically tasked to identify the realism of generated samples. It is therefore possible to show simulated samples of desired profile to the discriminator to obtain a measure of realism. This brings us to the next requirement of our sample representation technique, which is the availability of a parametric simulator. Here, while we may not have a parametric mechanism for generating traces, we do have `silgan` which can generate traces based upon handcrafted templates. If we recall the discussion in Chapter 10, a template is a piece-wise

linear approximation of one signal in a trace. A parametric mechanism for drawing, say, vehicle speed templates with a configurable number of starts and stops, is quite feasible. In fact, this is not all that different from the parametric mechanism that we use for drawing geometric shapes. Thus, by building a parametric method for drawing templates and translating them into traces using `silgan`, the discriminator score of generated traces can be used to explain sample representation in training data. Best case, such a pipeline reveals under-representation of important driving scenarios, and corrective action can be taken to compensate for it. Apart from a parametric template generator, all other parts of the explanation pipeline are readily available from the GAN itself. Therefore, an extension of this explanation technique to the domain of signal traces can be readily experimented in the near-future.

Extension to software and tasnet – Analogous to the previous case, the `tasnet` model has been trained on millions of files of source code, and we similarly lack information about its sample representation. While we know that the GitHub pre-training corpus contains hundreds of millions of files of C code, it would be valuable to understand how much of it is, say, real-time embedded, or safety critical code. Such insights are interesting because these are characteristics that apply to an important section of vehicle application software. However, unlike the previous case of signal traces, the sample representation technique that we present in this chapter does not extend easily to the domain of source code. This is mainly because, there is no readily available stereotypical discriminator, which can be probed using code snippets to measure sample representation. In the previous case, we had access to the GAN discriminator, but `tasnet` is trained as a BERT-like masked encoder that does not output a binary label. Further, it is quite difficult to construct a parametric generator of code, further complicating the extension of this technique. However, instead of directly extending the sample representation technique presented in this chapter, it may be more interesting to draw inspiration from the vocabulary challenge described in the previous chapter. Dispensing with a parametric generator, let us assume that we have access to a relatively small corpus of snippets that represent a category of interest like safety-critical code. Then, following the recipe of the vocabulary challenge, we could curate a set of keywords that capture concepts that are essential to safety. Then, by subjecting `tasnet` to a cloze challenge with the selected keywords masked, and by analyzing the distribution of token predictions, it may be possible to distill a measure of sample representation. By thus replacing a traditional classifier with a masked language model, and using selective masking, essential elements of the method presented in this chapter may be extendable to explain sample representation in a code corpus.

Considering such possibilities, extending this technique of explanation to information domains in software engineering can be an interesting avenue for future work. The result of such extension would be a better understanding of sample representation in datasets, in turn leading to a better reasoning about generalization of models, including foundation models, trained on these datasets.

Chapter 15

Conclusions

The automotive industry is already in a software driven reality, and the importance of software as the primary means of delivering customer value is only set to grow. Under such conditions, it is essential that vehicle manufacturers strengthen their ability to rapidly iterate through the software engineering process, without compromising quality. This work focuses upon two crucial engineering tasks of design compliance and test stimulus generation. Under current practice, both tasks require experienced engineers to spend considerable amount of time manually reviewing code and specifying test scenarios. Facing significant workload, not only is manual effort error-prone, but it also ends up reducing the cadence of delivery. In order to help mitigate these issues, this work introduces several techniques. Using `tasnet`, a language model of automotive software, we construct DECO, a rule-based algorithm that conducts compliance assessment in the model's representation space. The DECO algorithm is itself based upon the well-observed property of embedding regularity in language models, which holds that inputs related by the same concept follow a predictable geometry in representation space. Then, targeting test stimulus generation, we turn to GANs as a technique for learning underlying phenomena in signal traces that capture in-vehicle behavior. Using `logan`, a GAN trained on time-series signal traces, we demonstrate conditional generation of realistic stimulus. Here, the rule-based MLERP algorithm uses the property of latent interpolation for semantic combination to achieve controllable and verifiable stimulus generation. Finally, to make stimulus generation efficient, we introduce a technique for searching stimuli based upon the test objective. We demonstrate this using `silgan` and GRADES, a rule-based algorithm that exploits the end-to-end differentiability of DNNs in the GAN. We then go on to show that this combination of foundation models and principled rule-based algorithms constitutes a 'pre-train and calculate' paradigm for solving tasks. Requiring no explicit supervision, that this approach helps setup predictive models that are far more nuanced and transparent. Further, since 'pre-train and calculate' is heavily dependent upon properties in the abstract representation spaces, we present preliminary studies that characterize and explain high-dimensional vector spaces. Overall, tools and techniques developed in this work have immense potential to improve the

cadence and quality of automotive software development, helping achieve the pressing need in the automotive industry of rapid delivery of reliable software.

15.1 Future work

Previous chapters have pointed out several interesting avenues for future work, and we catalog and recount salient instances here for convenience. In Part I, where we explored the design compliance use case, possible future investigations include investigating compliance with design patterns other than Controller-Handler, using alternatives to the simple and semantically weaker cloze pre-training objective, and using artifacts beyond code to induce design knowledge in `tasnet`. The DECO algorithm can also be probed further to understand why relatively smaller violations leads to large variations in predicted rank. Then, in Part II, where we explored stimulus generation, apart from moving to the much more mainstream Transformer architecture, the use of alternative deep generative modeling techniques like diffusion is one avenue for future work. Further, taking inspiration from state-of-the-art techniques for realistic generation of human faces, we identified latent space engineering in `xgan` as a topic that requires future attention. When it comes to stimulus search, we use a rule-based mechanism to convert code under test into coverage indicators. Future work can investigate if coverage indicators can be substituted by embeddings of code under test, making it much more scalable. In Part III, where we recounted 'pre-train and calculate', we noted that combining its essential principles with the complementary 'pre-train and prompt' approach could lead to a powerful paradigm that leverages the relative strengths of both. Further, we only present preliminary studies in explaining high-dimensional spaces, both of which can be extended. The vocabulary challenge, which we develop for testing domain familiarity in `tasnet` needs to be part of a larger set of benchmark tasks. Developing a suite of tasks for evaluating language models of automotive software is a crucial task for the future. Then, techniques for explaining sample representation have only been studied on a simple dataset of images. Extending this toolkit to a dataset of signal traces would be valuable. More generally, while this work investigates design and testing tasks, automotive software engineering is replete with many more. Extending our techniques to other engineering tasks is only likely to further ease the development of the next generation of automotive software with high cadence and quality. Finally, it would also be interesting to extend the 'pre-train and calculate' paradigm to see if the rule-based calculation of nuanced predictions is applicable to domains beyond automotive software engineering.

Bibliography

[1] L. L. Bello, R. Mariani, S. Mubeen, and S. Saponara, "Recent advances and trends in on-board embedded and networked automotive systems," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 2, pp. 1038–1051, 2019.

[2] V. Trucks, "Volvo fh driver guide," 2023. [Online]. Available: https://driverguide.volvotrucks.com/lang/en/chassi/A0000FH/home

[3] C. Ebert and J. M. Favaro, "Automotive software," *IEEE Softw.*, vol. 34, no. 3, pp. 33–39, 2017. [Online]. Available: https://doi.org/10.1109/MS.2017.82

[4] U. Nations, "682050," 2023. [Online]. Available: https://www.un.org/development/desa/en/news/population/2018-revision-of-world-urbanization-prospects.html

[5] P. Wallin, S. Johnsson, and J. Axelsson, "Issues related to development of E/E product line architectures in heavy vehicles," in *42st Hawaii International International Conference on Systems Science (HICSS-42 2009), Proceedings (CD-ROM and online), 5-8 January 2009, Waikoloa, Big Island, HI, USA*. IEEE Computer Society, 2009, pp. 1–10. [Online]. Available: https://doi.org/10.1109/HICSS.2009.276

[6] S. Khastgir, S. Birrell, G. Dhadyalla, and P. Jennings, "The science of testing: An automotive perspective," in *WCX World Congress Experience*. SAE International, apr 2018. [Online]. Available: https://doi.org/10.4271/2018-01-1070

[7] C. Salzmann and T. Stauner, *Automotive Software Engineering*. Boston, MA: Springer US, 2004, pp. 333–347. [Online]. Available: https://doi.org/10.1007/1-4020-7991-5_21

[8] B. Gallina and M. Nyberg, "Reconciling the ISO 26262-compliant and the agile documentation management in the Swedish context," in *CARS 2015 - Critical Automotive applications: Robustness & Safety*, M. Roy, Ed., Paris, France, Sep. 2015. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01192981

[9] "Systems and software engineering – system life cycle processes," International Organization for Standardization, Standard, 2015.

[10] "Information technology — process assessment — process assessment model for software life cycle processes," International Organization for Standardization, Standard, 2021.

[11] I. C. on Systems Engineering, Ed., *INCOSE Systems Engineering Handbook*, 2015, vol. 4.0.

[12] "Road vehicles - functional safety - part 9: Automotive safety integrity level (asil)-oriented and safety-oriented analyses," International Organization for Standardization, Standard, 2011.

[13] M. Staron, *Automotive Software Architectures - An Introduction, Second Edition*. Springer, 2021. [Online]. Available: https://doi.org/10.1007/978-3-030-65939-4

[14] A. Strasser, B. Cool, C. Gernert, C. Knieke, M. Körner, D. Niebuhr, H. Peters, A. Rausch, O. Brox, S. Jauns-Seyfried, H. Jelden, S. Klie, and M. Krämer, "Mastering erosion of software architecture in automotive software product lines," in *SOFSEM 2014: Theory and Practice of Computer Science - 40th International Conference on Current Trends in Theory and Practice of Computer Science, Nový Smokovec, Slovakia, January 26-29, 2014, Proceedings*, ser. Lecture Notes in Computer Science, vol. 8327. Springer, 2014, pp. 491–502. [Online]. Available: https://doi.org/10.1007/978-3-319-04298-5_43

[15] H. Altinger, F. Wotawa, and M. Schurius, "Testing methods used in the automotive industry: Results from a survey," in *Proceedings of the 2014 Workshop on Joining AcadeMiA and Industry Contributions to Test Automation and Model-Based Testing*, ser. JAMAICA 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 1–6. [Online]. Available: https://doi.org/10.1145/2631890.2631891

[16] H. Kaijser, H. Lönn, P. Thorngren, J. Ekberg, M. Henningsson, and M. Larsson, "Towards Simulation-Based Verification for Continuous Integration and Delivery," in *ERTS 2018*, ser. 9th European Congress on Embedded Real Time Software and Systems (ERTS 2018), Toulouse, France, Jan. 2018. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02156371

[17] G. Wang, A. Gunasekaran, E. W. Ngai, and T. Papadopoulos, "Big data analytics in logistics and supply chain management: Certain investigations for research and applications," *International Journal of Production Economics*, vol. 176, pp. 98–110, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925527316300056

[18] J. Bosch and H. H. Olsson, "Digital for real: A multicase study on the digital transformation of companies in the embedded systems domain," *J. Softw. Evol. Process.*, vol. 33, no. 5, 2021. [Online]. Available: https://doi.org/10.1002/smr.2333

[19] H. H. Olsson and J. Bosch, "From opinions to data-driven software r&d: A multi-case study on how to close the 'open loop' problem," in

40th EUROMICRO Conference on Software Engineering and Advanced Applications, EUROMICRO-SEAA 2014, Verona, Italy, August 27-29, 2014. IEEE Computer Society, 2014, pp. 9–16. [Online]. Available: https://doi.org/10.1109/SEAA.2014.75

[20] M. O'Neil, X. Cai, L. Muselli, F. Pailler, and S. Zacchiroli, *The coproduction of open source software by volunteers and big tech firms*. News Media Research Centre, University of Canberra, 2021.

[21] R. Inc. (2022) The state of enterprise open source: A red hat report. [Online]. Available: https://www.redhat.com/en/resources/state-of-enterprise-open-source-report-2022

[22] G. Inc. (2022) Octoverse 2022: 10 years of tracking open source. [Online]. Available: https://github.blog/2022-11-17-octoverse-2022-10-years-of-tracking-open-source/

[23] R. Alt, J. M. Leimeister, T. Priemuth, S. Sachse, N. Urbach, and N. Wunderlich, "Software-defined business," *Bus. Inf. Syst. Eng.*, vol. 62, no. 6, pp. 609–621, 2020. [Online]. Available: https://doi.org/10.1007/s12599-020-00669-6

[24] M. Vechev and E. Yahav, "Programming with "big code"," *Foundations and Trends® in Programming Languages*, vol. 3, no. 4, pp. 231–284, 2016. [Online]. Available: http://dx.doi.org/10.1561/2500000028

[25] Y. LeCun, Y. Bengio, and G. E. Hinton, "Deep learning," *Nat.*, vol. 521, no. 7553, pp. 436–444, 2015. [Online]. Available: https://doi.org/10.1038/nature14539

[26] S. Dong, P. Wang, and K. Abbas, "A survey on deep learning and its applications," *Computer Science Review*, vol. 40, p. 100379, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1574013721000198

[27] Y. Yang, X. Xia, D. Lo, and J. C. Grundy, "A survey on deep learning for software engineering," *CoRR*, vol. abs/2011.14597, 2020. [Online]. Available: https://arxiv.org/abs/2011.14597

[28] R. B. et al., "On the opportunities and risks of foundation models," *CoRR*, vol. abs/2108.07258, 2021. [Online]. Available: https://arxiv.org/abs/2108.07258

[29] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 10, pp. 1345–1359, 2010.

[30] J. Schaeuffele and T. Zurawka, *Automotive Software Engineering, Second Edition*. SAE International, sep 2016.

[31] A. Haghighatkhah, A. Banijamali, O. Pakanen, M. Oivo, and P. Kuvaja, "Automotive software engineering: A systematic mapping study," *J. Syst. Softw.*, vol. 128, pp. 25–55, 2017. [Online]. Available: https://doi.org/10.1016/j.jss.2017.03.005

[32] "Road vehicles - cybersecurity engineering," International Organization for Standardization, Standard, 2011.

[33] D. Reinhardt and M. Kucera, "Domain controlled architecture - A new approach for large scale software integrated automotive systems," in *PECCS 2013 - Proceedings of the 3rd International Conference on Pervasive Embedded Computing and Communication Systems, Barcelona, Spain, 19-21 February, 2013*, C. Benavente-Peces and J. Filipe, Eds. SciTePress, 2013, pp. 221–226.

[34] R. Mutschler, O. Trost, and J. Crepin, "Agile methodologies in the development of automotive embedded software," *ATZelectronics worldwide*, vol. 15, no. 7, pp. 44–49, Jul 2020. [Online]. Available: https://doi.org/10.1007/s38314-020-0225-z

[35] J.-P. Steghöfer, E. Knauss, J. Horkoff, and R. Wohlrab, "Challenges of scaled agile for safety-critical systems," in *Product-Focused Software Process Improvement*, X. Franch, T. Männistö, and S. Martínez-Fernández, Eds. Cham: Springer International Publishing, 2019, pp. 350–366.

[36] S. Jiang, "Vehicle e/e architecture and its adaptation to new technical trends," in *WCX SAE World Congress Experience*. SAE International, apr 2019. [Online]. Available: https://doi.org/10.4271/2019-01-0862

[37] A. Bucaioni and P. Pelliccione, "Technical architectures for automotive systems," in *2020 IEEE International Conference on Software Architecture, ICSA 2020, Salvador, Brazil, March 16-20, 2020*. IEEE, 2020, pp. 46–57. [Online]. Available: https://doi.org/10.1109/ICSA47634.2020.00013

[38] S. Bunzel, "AUTOSAR - the standardized software architecture," *Inform. Spektrum*, vol. 34, no. 1, pp. 79–83, 2011. [Online]. Available: https://doi.org/10.1007/s00287-010-0506-7

[39] "Specification of operating system," AUTOSAR, Standard, 2018. [Online]. Available: https://www.autosar.org/fileadmin/standards/classic/4-4-0/AUTOSAR_SWS_OS.pdf

[40] A. Magnusson, L. Laine, and J. Lindberg, "Rethink EE architecture in automotive to facilitate automation, connectivity, and electro mobility," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, F. Paulisch and J. Bosch, Eds. ACM, 2018, pp. 65–74. [Online]. Available: https://doi.org/10.1145/3183519.3183526

[41] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, P. L. Bartlett, F. C. N. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds.,

2012, pp. 1106–1114. [Online]. Available: https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html

[42] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[43] T. Diwan, G. Anirudh, and J. V. Tembhurne, "Object detection using yolo: challenges, architectural successors, datasets and applications," *Multimedia Tools and Applications*, Aug 2022. [Online]. Available: https://doi.org/10.1007/s11042-022-13644-y

[44] J. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Žídek, A. Potapenko, A. Bridgland, C. Meyer, S. A. A. Kohl, A. J. Ballard, A. Cowie, B. Romera-Paredes, S. Nikolov, R. Jain, J. Adler, T. Back, S. Petersen, D. Reiman, E. Clancy, M. Zielinski, M. Steinegger, M. Pacholska, T. Berghammer, S. Bodenstein, D. Silver, O. Vinyals, A. W. Senior, K. Kavukcuoglu, P. Kohli, and D. Hassabis, "Highly accurate protein structure prediction with alphafold," *Nature*, vol. 596, no. 7873, pp. 583–589, Aug 2021. [Online]. Available: https://doi.org/10.1038/s41586-021-03819-2

[45] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., 2020. [Online]. Available: https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

[46] L. Zhang, S. Wang, and B. Liu, "Deep learning for sentiment analysis : A survey," *CoRR*, vol. abs/1801.07883, 2018. [Online]. Available: http://arxiv.org/abs/1801.07883

[47] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, "Learning word vectors for sentiment analysis," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*. Portland, Oregon, USA: Association for Computational Linguistics, June 2011, pp. 142–150. [Online]. Available: http://www.aclweb.org/anthology/P11-1015

[48] S. B. Kotsiantis, "Supervised machine learning: A review of classification techniques." NLD: IOS Press, 2007, p. 3–24.

[49] S. Zad, M. Heidari, J. H. J. Jones, and O. Uzuner, "Emotion detection of textual data: An interdisciplinary survey," in *2021 IEEE World AI IoT Congress (AIIoT)*, 2021, pp. 0255–0261.

[50] D. Demszky, D. Movshovitz-Attias, J. Ko, A. S. Cowen, G. Nemade, and S. Ravi, "Goemotions: A dataset of fine-grained emotions," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, D. Jurafsky, J. Chai, N. Schluter, and J. R. Tetreault, Eds. Association for Computational Linguistics, 2020, pp. 4040–4054. [Online]. Available: https://doi.org/10.18653/v1/2020.acl-main.372

[51] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds. Association for Computational Linguistics, 2019, pp. 4171–4186. [Online]. Available: https://doi.org/10.18653/v1/n19-1423

[52] Z. Alyafeai, M. S. AlShaibani, and I. Ahmad, "A survey on transfer learning in natural language processing," *CoRR*, vol. abs/2007.04239, 2020. [Online]. Available: https://arxiv.org/abs/2007.04239

[53] A. Chiorrini, C. Diamantini, A. Mircoli, and D. Potena, "Emotion and sentiment analysis of tweets using BERT," in *Proceedings of the Workshops of the EDBT/ICDT 2021 Joint Conference, Nicosia, Cyprus, March 23, 2021*, ser. CEUR Workshop Proceedings, C. Costa and E. Pitoura, Eds., vol. 2841. CEUR-WS.org, 2021. [Online]. Available: http://ceur-ws.org/Vol-2841/DARLI-AP_17.pdf

[54] L. Yuan, D. Chen, Y. Chen, N. Codella, X. Dai, J. Gao, H. Hu, X. Huang, B. Li, C. Li, C. Liu, M. Liu, Z. Liu, Y. Lu, Y. Shi, L. Wang, J. Wang, B. Xiao, Z. Xiao, J. Yang, M. Zeng, L. Zhou, and P. Zhang, "Florence: A new foundation model for computer vision," *CoRR*, vol. abs/2111.11432, 2021. [Online]. Available: https://arxiv.org/abs/2111.11432

[55] O. Méndez-Lucio, C. A. Nicolaou, and B. Earnshaw, "Mole: a molecular foundation model for drug discovery," *CoRR*, vol. abs/2211.02657, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2211.02657

[56] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2013. [Online]. Available: http://arxiv.org/abs/1301.3781

[57] D. Chandrasekaran and V. Mago, "Evolution of semantic similarity—a survey," *ACM Comput. Surv.*, vol. 54, no. 2, feb 2021. [Online]. Available: https://doi.org/10.1145/3440755

[58] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track*

Proceedings, Y. Bengio and Y. LeCun, Eds., 2016. [Online]. Available: http://arxiv.org/abs/1511.06434

[59] F. de la Parra, "Discovery of patterns in simulink systems," Ph.D. dissertation, Queen's University at Kingston, Ontario, Canada, 2017. [Online]. Available: https://hdl.handle.net/1974/20102

[60] A. Armoush, "Design patterns for safety-critical embedded systems," Ph.D. dissertation, RWTH Aachen University, 2010. [Online]. Available: http://darwin.bth.rwth-aachen.de/opus3/volltexte/2010/3273/

[61] B. H. Cheng, B. Doherty, N. Polanco, and M. Pasco, "Security patterns for automotive systems," in *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, 2019, pp. 54–63.

[62] B. P. Douglass, *Design Patterns for Embedded Systems in C*. Boston: Newnes, 2011. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9781856177078000145

[63] "Application design patterns catalogue," AUTOSAR, Standard, 2022. [Online]. Available: https://www.autosar.org/fileadmin/standards/classic/22-11/AUTOSAR_TR_AIDesignPatternsCatalogue.pdf

[64] R. Kazman, M. Klein, and P. Clements, "Atam: Method for architecture evaluation," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU/SEI-2000-TR-004, 2000. [Online]. Available: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=5177

[65] P. Bengtsson, N. H. Lassing, J. Bosch, and H. van Vliet, "Architecture-level modifiability analysis (ALMA)," *J. Syst. Softw.*, vol. 69, no. 1-2, pp. 129–147, 2004. [Online]. Available: https://doi.org/10.1016/S0164-1212(03)00080-3

[66] A. Sutherland and G. Venolia, "Can peer code reviews be exploited for later information needs?" in *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Companion Volume*. IEEE, 2009, pp. 259–262. [Online]. Available: https://doi.org/10.1109/ICSE-COMPANION.2009.5070996

[67] P. C. Rigby, B. Cleary, F. Painchaud, M. D. Storey, and D. M. Germán, "Contemporary peer review in action: Lessons from open source development," *IEEE Softw.*, vol. 29, no. 6, pp. 56–61, 2012. [Online]. Available: https://doi.org/10.1109/MS.2012.24

[68] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, and M. W. Godfrey, "Investigating code review quality: Do people and participation matter?" in *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015*, R. Koschke, J. Krinke, and M. P. Robillard, Eds. IEEE Computer Society, 2015, pp. 111–120. [Online]. Available: https://doi.org/10.1109/ICSM.2015.7332457

[69] A. Bosu, M. Greiler, and C. Bird, "Characteristics of useful code reviews: An empirical study at microsoft," in *12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015, Florence, Italy, May 16-17, 2015*, M. D. Penta, M. Pinzger, and R. Robbes, Eds. IEEE Computer Society, 2015, pp. 146–156. [Online]. Available: https://doi.org/10.1109/MSR.2015.21

[70] M. Allamanis, E. T. Barr, P. T. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Comput. Surv.*, vol. 51, no. 4, pp. 81:1–81:37, 2018. [Online]. Available: https://doi.org/10.1145/3212695

[71] "Software Engineering — Guide to the software engineering body of knowledge (SWEBOK)," International Organization for Standardization, Geneva, CH, Standard, Sep. 2015.

[72] L. Chen, M. A. Babar, and B. Nuseibeh, "Characterizing architecturally significant requirements," *IEEE Software*, vol. 30, no. 2, pp. 38–45, 2013. [Online]. Available: https://doi.org/10.1109/MS.2012.174

[73] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. T. Devanbu, "On the naturalness of software," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. IEEE Computer Society, 2012, pp. 837–847. [Online]. Available: https://doi.org/10.1109/ICSE.2012.6227135

[74] Y. Pu, K. Narasimhan, A. Solar-Lezama, and R. Barzilay, "sk_p: a neural program corrector for moocs," in *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity, SPLASH 2016, Amsterdam, Netherlands, October 30 - November 4, 2016*. ACM, 2016, pp. 39–40. [Online]. Available: https://doi.org/10.1145/2984043.2989222

[75] D. E. Knuth, *Literate programming*, ser. CSLI lecture notes series. Center for the Study of Language and Information, 1992, vol. 27.

[76] J. W. Reeves, "What is software design," *C++ Journal*, vol. 2, no. 2, pp. 14–12, 1992.

[77] V. M. Navale, K. Williams, A. Lagospiris, M. Schaffert, and M.-A. Schweiker, "(r)evolution of e/e architectures," *SAE International Journal of Passenger Cars - Electronic and Electrical Systems*, vol. 8, no. 2, pp. 282–288, apr 2015. [Online]. Available: https://doi.org/10.4271/2015-01-0196

[78] B. Rozière, M. Lachaux, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. [Online]. Available: https://proceedings.neurips.cc/paper/2020/hash/ed23fbf18c2cd35f8c7f8de44f85c08d-Abstract.html

[79] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," in *Proceedings of the 54th Annual*

Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers. The Association for Computer Linguistics, 2016. [Online]. Available: https://doi.org/10.18653/v1/p16-1162

[80] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, 2017, pp. 5998–6008. [Online]. Available: https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[81] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *EMNLP 2020*, ser. Findings of ACL, vol. EMNLP 2020. Association for Computational Linguistics, 2020, pp. 1536–1547. [Online]. Available: https://doi.org/10.18653/v1/2020.findings-emnlp.139

[82] N. Kitaev, L. Kaiser, and A. Levskaya, "Reformer: The efficient transformer," in *ICLR 2020*. OpenReview.net, 2020. [Online]. Available: https://openreview.net/forum?id=rkgNKkHtvB

[83] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations," in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2018, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 1 (Long Papers)*. Association for Computational Linguistics, 2018, pp. 2227–2237. [Online]. Available: https://doi.org/10.18653/v1/n18-1202

[84] T. Mikolov, W. Yih, and G. Zweig, "Linguistic regularities in continuous space word representations," in *Human Language Technologies: Conference of the North American Chapter of the Association of Computational Linguistics, Proceedings, June 9-14, 2013, Westin Peachtree Plaza Hotel, Atlanta, Georgia, USA*. The Association for Computational Linguistics, 2013, pp. 746–751. [Online]. Available: https://aclanthology.org/N13-1090/

[85] X. Zhu and G. de Melo, "Sentence analogies: Linguistic regularities in sentence embeddings," in *COLING 2020*. International Committee on Computational Linguistics, 2020, pp. 3389–3400. [Online]. Available: https://doi.org/10.18653/v1/2020.coling-main.300

[86] S. Gururangan, A. Marasovic, S. Swayamdipta, K. Lo, I. Beltagy, D. Downey, and N. A. Smith, "Don't stop pretraining: Adapt language models to domains and tasks," in *ACL 2020*, pp. 8342–8360. [Online]. Available: https://doi.org/10.18653/v1/2020.acl-main.740

[87] A. Drozd, A. Gladkova, and S. Matsuoka, "Word embeddings, analogies, and machine learning: Beyond king - man + woman = queen," in *COLING 2016*. ACL, 2016, pp. 3519–3530. [Online]. Available: https://aclanthology.org/C16-1332/

[88] Y. Sun, S. Wang, Y. Li, S. Feng, X. Chen, H. Zhang, X. Tian, D. Zhu, H. Tian, and H. Wu, "ERNIE: enhanced representation through knowledge integration," *CoRR*, vol. abs/1904.09223, 2019. [Online]. Available: http://arxiv.org/abs/1904.09223

[89] M. Joshi, D. Chen, Y. Liu, D. S. Weld, L. Zettlemoyer, and O. Levy, "Spanbert: Improving pre-training by representing and predicting spans," *Trans. Assoc. Comput. Linguistics*, vol. 8, pp. 64–77, 2020. [Online]. Available: https://transacl.org/ojs/index.php/tacl/article/view/1853

[90] K. Clark, M. Luong, Q. V. Le, and C. D. Manning, "ELECTRA: pre-training text encoders as discriminators rather than generators," in *ICLR 2020*. OpenReview.net, 2020. [Online]. Available: https://openreview.net/forum?id=r1xMH1BtvB

[91] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "BART: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*. Association for Computational Linguistics, 2020, pp. 7871–7880. [Online]. Available: https://doi.org/10.18653/v1/2020.acl-main.703

[92] B. Rozière, M. Lachaux, M. Szafraniec, and G. Lample, "DOBF: A deobfuscation pre-training objective for programming languages," *CoRR*, vol. abs/2102.07492, 2021. [Online]. Available: https://arxiv.org/abs/2102.07492

[93] H. Yarahmadi and S. M. H. Hasheminejad, "Design pattern detection approaches: a systematic review of the literature," *Artif. Intell. Rev.*, vol. 53, no. 8, pp. 5789–5846, 2020. [Online]. Available: https://doi.org/10.1007/s10462-020-09834-5

[94] H. Thaller, L. Linsbauer, and A. Egyed, "Feature maps: A comprehensible software representation for design pattern detection," in *2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2019, pp. 207–217.

[95] R. Oberhauser, "A machine learning approach towards automatic software design pattern recognition across multiple programming languages," in *Proceedings of the Fifteenth International Conference on Software Engineering Advances*. IARIA, 2020, pp. 27–32.

[96] N. Nazar, A. Aleti, and Y. Zheng, "Feature-based software design pattern detection," *Journal of Systems and Software*, vol. 185, p. 111179, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121221002624

[97] A. Bakarov, "A survey of word embeddings evaluation methods," *arXiv preprint arXiv:1801.09536*, 2018.

[98] E. Grave, P. Bojanowski, P. Gupta, A. Joulin, and T. Mikolov, "Learning word vectors for 157 languages," *arXiv preprint arXiv:1802.06893*, 2018.

[99] S. Lim, H. Prade, and G. Richard, "Classifying and completing word analogies by machine learning," *International Journal of Approximate Reasoning*, vol. 132, pp. 1–25, 2021.

[100] J. R. da Silva and H. d. M. Caseli, "Generating sense embeddings for syntactic and semantic analogy for portuguese," *arXiv preprint arXiv:2001.07574*, 2020.

[101] T. L. Chen, M. Emerling, G. R. Chaudhari, Y. R. Chillakuru, Y. Seo, T. H. Vu, and J. H. Sohn, "Domain specific word embeddings for natural language processing in radiology," *Journal of biomedical informatics*, vol. 113, p. 103665, 2021.

[102] A. Ushio, L. Espinosa-Anke, S. Schockaert, and J. Camacho-Collados, "Bert is to nlp what alexnet is to cv: can pre-trained language models identify analogies?" *arXiv preprint arXiv:2105.04949*, 2021.

[103] B. Broekman and E. Notenboom, *Testing Embedded Software*. Addison-Wesley, 2002. [Online]. Available: http://www.amazon.de/Testing-Embedded-Software-Broekman-Bart/dp/0321159861/

[104] P. Giusto, A. Ferrari, L. Lavagno, J.-Y. Brunel, E. Fourgeau, and A. Sangiovanni-Vincentelli, "Automotive virtual integration platforms: why's, what's, and how's," in *Proceedings. IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 2002, pp. 370–378.

[105] S. Documentation, "Simulation and model-based design," 2020. [Online]. Available: https://www.mathworks.com/products/simulink.html

[106] Modelica Association, "Modelica® - a unified object-oriented language for physical systems modeling. Tutorial," Dec. 2000. [Online]. Available: http://www.modelica.org/documents/ModelicaTutorial14.pdf

[107] Z. Hu, Z. Yang, R. Salakhutdinov, and E. P. Xing, "On unifying deep generative models," in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. [Online]. Available: https://openreview.net/forum?id=rylSzl-R-

[108] L. Ruthotto and E. Haber, "An introduction to deep generative modeling," *CoRR*, vol. abs/2103.05180, 2021. [Online]. Available: https://arxiv.org/abs/2103.05180

[109] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. C. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in Neural Information Processing Systems 27: Annual*

Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada, 2014, pp. 2672–2680. [Online]. Available: http://papers.nips.cc/paper/5423-generative-adversarial-nets

[110] A. Bermano, R. Gal, Y. Alaluf, R. Mokady, Y. Nitzan, O. Tov, O. Patashnik, and D. Cohen-Or, "State-of-the-art in the architecture, methods and applications of stylegan," *Computer Graphics Forum*, vol. 41, no. 2, pp. 591–611. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.14503

[111] D. Zhao and H. Peng, "From the lab to the street: Solving the challenge of accelerating automated vehicle testing," 2017. [Online]. Available: https://arxiv.org/abs/1707.04792

[112] V. Garousi, M. Felderer, Çağrı Murat Karapıçak, and U. Yılmaz, "Testing embedded software: A survey of the literature," *Information and Software Technology*, vol. 104, pp. 14–45, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584918301265

[113] D. Verburg, A. van der Knaap, and J. Ploeg, "Vehil: developing and testing intelligent vehicles," in *Intelligent Vehicle Symposium, 2002. IEEE*, vol. 2, 2002, pp. 537–544 vol.2.

[114] "Un regulation no. 154 - worldwide harmonized light vehicles test procedure (wltp)," United Nations, Standard, 2021.

[115] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," in *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014. [Online]. Available: http://arxiv.org/abs/1312.6114

[116] A. B. L. Larsen, S. K. Sønderby, H. Larochelle, and O. Winther, "Autoencoding beyond pixels using a learned similarity metric," in *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, ser. JMLR Workshop and Conference Proceedings, M. Balcan and K. Q. Weinberger, Eds., vol. 48. JMLR.org, 2016, pp. 1558–1566. [Online]. Available: http://proceedings.mlr.press/v48/larsen16.html

[117] E. Brophy, Z. Wang, Q. She, and T. Ward, "Generative adversarial networks in time series: A survey and taxonomy," *CoRR*, vol. abs/2107.11098, 2021. [Online]. Available: https://arxiv.org/abs/2107.11098

[118] P. Liang, R. Bommasani, T. Lee, D. Tsipras, D. Soylu, M. Yasunaga, Y. Zhang, D. Narayanan, Y. Wu, A. Kumar, B. Newman, B. Yuan, B. Yan, C. Zhang, C. Cosgrove, C. D. Manning, C. Ré, D. Acosta-Navas, D. A. Hudson, E. Zelikman, E. Durmus, F. Ladhak, F. Rong, H. Ren, H. Yao, J. Wang, K. Santhanam, L. Orr, L. Zheng, M. Yuksekgonul, M. Suzgun, N. Kim, N. Guha, N. Chatterji, O. Khattab, P. Henderson, Q. Huang, R. Chi, S. M. Xie, S. Santurkar, S. Ganguli, T. Hashimoto, T. Icard, T. Zhang, V. Chaudhary, W. Wang, X. Li, Y. Mai, Y. Zhang, and Y. Koreeda, "Holistic evaluation of language models," 2022.

[119] C. Cassisi, P. Montalto, M. Aliotta, A. Cannata, and A. Pulvirenti, "Similarity measures and dimensionality reduction techniques for time series data mining," in *Advances in Data Mining Knowledge Discovery and Applications*, A. Karahoca, Ed. Rijeka: IntechOpen, 2012, ch. 3. [Online]. Available: https://doi.org/10.5772/49941

[120] H. Sakoe and S. Chiba, "Dynamic programming algorithm optimization for spoken word recognition," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 26, no. 1, pp. 43–49, February 1978.

[121] Zhou Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: from error visibility to structural similarity," *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, April 2004.

[122] D. Nalic, T. Mihalj, M. Bäumler, M. Lehmann, and S. Bernsteiner, "Sceneario based testing of autoamted driving systems: A literature survey," 11 2020, p. 1, fISITA Web Congress 2020. [Online]. Available: https://go.fisita.com/fisita2020

[123] S. Ulbrich, T. Menzel, A. Reschka, F. Schuldt, and M. Maurer, "Defining and substantiating the terms scene, situation, and scenario for automated driving," in *IEEE 18th International Conference on Intelligent Transportation Systems, Gran Canaria, Spain, September 15-18*. IEEE, 2015. [Online]. Available: https://doi.org/10.1109/ITSC.2015.164

[124] D. J. Fremont, E. Kim, Y. V. Pant, S. A. Seshia, A. Acharya, X. Bruso, P. Wells, S. Lemke, Q. Lu, and S. Mehta, "Formal scenario-based testing of autonomous vehicles: From simulation to the real world," in *23rd IEEE International Conference on Intelligent Transportation Systems, ITSC 2020, Rhodes, Greece, September 20-23*. IEEE, 2020, pp. 1–8. [Online]. Available: https://doi.org/10.1109/ITSC45102.2020.9294368

[125] J. Greenyer, M. Haase, J. Marhenke, and R. Bellmer, "Evaluating a formal scenario-based method for the requirements analysis in automotive software engineering," ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 1002–1005. [Online]. Available: https://doi.org/10.1145/2786805.2804432

[126] Z. Wang, Q. She, and T. E. Ward, "Generative adversarial networks in computer vision: A survey and taxonomy," *ACM Comput. Surv.*, vol. 54, no. 2, pp. 37:1–37:38, 2021. [Online]. Available: https://doi.org/10.1145/3439723

[127] X. Mao, Q. Li, H. Xie, R. Y. K. Lau, Z. Wang, and S. P. Smolley, "Least squares generative adversarial networks," in *IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017*. IEEE Computer Society, 2017, pp. 2813–2821. [Online]. Available: https://doi.org/10.1109/ICCV.2017.304

[128] X. Huang, M. Liu, S. J. Belongie, and J. Kautz, "Multimodal unsupervised image-to-image translation," in *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September*

8-14, 2018, Proceedings, Part III, ser. Lecture Notes in Computer Science, V. Ferrari, M. Hebert, C. Sminchisescu, and Y. Weiss, Eds., vol. 11207. Springer, 2018, pp. 179–196. [Online]. Available: https://doi.org/10.1007/978-3-030-01219-9_11

[129] J. Zhu, T. Park, P. Isola, and A. A. Efros, "Unpaired image-to-image translation using cycle-consistent adversarial networks," in *IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017*. IEEE Computer Society, 2017, pp. 2242–2251. [Online]. Available: https://doi.org/10.1109/ICCV.2017.244

[130] J. L. Eddeland, K. Claessen, N. Smallbone, Z. Ramezani, S. Miremadi, and K. Åkesson, "Enhancing temporal logic falsification with specification transformation and valued booleans," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 39, no. 12, pp. 5247–5260, 2020. [Online]. Available: https://doi.org/10.1109/TCAD.2020.2966480

[131] J. C. B. Gamboa, "Deep learning for time-series analysis," *CoRR*, vol. abs/1701.01887, 2017. [Online]. Available: http://arxiv.org/abs/1701.01887

[132] A. van den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. W. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," in *The 9th ISCA Speech Synthesis Workshop, Sunnyvale, CA, USA, 13-15 September 2016*. ISCA, 2016, p. 125. [Online]. Available: http://www.isca-speech.org/archive/SSW_2016/abstracts/ssw9_DS-4_van_den_Oord.html

[133] S. Hou, W. Xu, J. Chai, C. Wang, W. Zhuang, Y. Chen, H. Bao, and Y. Wang, "A causal convolutional neural network for motion modeling and synthesis," 2021.

[134] P. Ren, Y. Xiao, X. Chang, P.-Y. Huang, Z. Li, X. Chen, and X. Wang, "A comprehensive survey of neural architecture search: Challenges and solutions," 2021.

[135] Q. Wen, T. Zhou, C. Zhang, W. Chen, Z. Ma, J. Yan, and L. Sun, "Transformers in time series: A survey," 2023.

[136] A. I. Miller, *10 Ian Goodfellow's Generative Adversarial Networks: AI Learns to Imagine*, 2019, pp. 87–98.

[137] A. Jabbar, X. Li, and B. Omar, "A survey on generative adversarial networks: Variants, applications, and training," *ACM Comput. Surv.*, vol. 54, no. 8, pp. 157:1–157:49, 2022. [Online]. Available: https://doi.org/10.1145/3463475

[138] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1412.6980

[139] S. Bond-Taylor, A. Leach, Y. Long, and C. G. Willcocks, "Deep generative modelling: A comparative review of vaes, gans, normalizing flows, energy-based and autoregressive models," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 44, no. 11, pp. 7327–7347, 2022. [Online]. Available: https://doi.org/10.1109/TPAMI.2021.3116668

[140] A. Ramesh, M. Pavlov, G. Goh, S. Gray, C. Voss, A. Radford, M. Chen, and I. Sutskever, "Zero-shot text-to-image generation," *CoRR*, vol. abs/2102.12092, 2021. [Online]. Available: https://arxiv.org/abs/2102.12092

[141] A. Ramesh, P. Dhariwal, A. Nichol, C. Chu, and M. Chen, "Hierarchical text-conditional image generation with CLIP latents," *CoRR*, vol. abs/2204.06125, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2204.06125

[142] H. Cao, C. Tan, Z. Gao, G. Chen, P. Heng, and S. Z. Li, "A survey on generative diffusion model," *CoRR*, vol. abs/2209.02646, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2209.02646

[143] C. Esteban, S. L. Hyland, and G. Rätsch, "Real-valued (medical) time series generation with recurrent conditional gans," *CoRR*, vol. abs/1706.02633, 2017. [Online]. Available: http://arxiv.org/abs/1706.02633

[144] B. K. Sriperumbudur, A. Gretton, K. Fukumizu, B. Schölkopf, and G. R. Lanckriet, "Hilbert space embeddings and metrics on probability measures," *J. Mach. Learn. Res.*, vol. 11, pp. 1517–1561, Aug. 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1756006.1859901

[145] O. Mogren, "C-RNN-GAN: continuous recurrent neural networks with adversarial training," *CoRR*, vol. abs/1611.09904, 2016. [Online]. Available: http://arxiv.org/abs/1611.09904

[146] N. M. Edvin Listo Zec, Henrik Arnelid, "Recurrent conditional gans for time series sensor modelling," in *Time Series Workshop at International Conference on Machine Learning*, Long Beach, California, Jan. 2019.

[147] M. Alzantot, S. Chakraborty, and M. B. Srivastava, "Sensegen: A deep learning architecture for synthetic sensor data generation," *CoRR*, vol. abs/1701.08886, 2017. [Online]. Available: http://arxiv.org/abs/1701.08886

[148] E. Brophy, Z. Wang, and T. E. Ward, "Quick and easy time series generation with established image-based gans," *CoRR*, vol. abs/1902.05624, 2019. [Online]. Available: http://arxiv.org/abs/1902.05624

[149] C. Zhang, S. R. Kuppannagari, R. Kannan, and V. K. Prasanna, "Generative adversarial network for synthetic time series data generation in smart grids," in *2018 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids, SmartGridComm 2018, Aalborg, Denmark, October 29-31, 2018*, 2018, pp. 1–6. [Online]. Available: https://doi.org/10.1109/SmartGridComm.2018.8587464

[150] S. Kolouri, S. R. Park, M. Thorpe, D. Slepcev, and G. K. Rohde, "Optimal mass transport: Signal processing and machine-learning applications," *IEEE Signal Processing Magazine*, vol. 34, no. 4, pp. 43–59, July 2017.

[151] Y. Kang, R. J. Hyndman, and F. Li, "GRATIS: generating time series with diverse and controllable characteristics," *CoRR*, vol. abs/1903.02787, 2019. [Online]. Available: http://arxiv.org/abs/1903.02787

[152] L. Kegel, M. Hahmann, and W. Lehner, "Feature-based comparison and generation of time series," in *Proceedings of the 30th International Conference on Scientific and Statistical Database Management, SSDBM 2018, Bozen-Bolzano, Italy, July 09-11, 2018*, 2018, pp. 20:1–20:12. [Online]. Available: https://doi.org/10.1145/3221269.3221293

[153] P. Gupta, P. Malhotra, J. Narwariya, L. Vig, and G. Shroff, "Transfer learning for clinical time series analysis using deep neural networks," *CoRR*, vol. abs/1904.00655, 2019. [Online]. Available: http://arxiv.org/abs/1904.00655

[154] C. Schockaert and H. Hoyez, "Mts-cyclegan: An adversarial-based deep mapping learning network for multivariate time series domain adaptation applied to the ironmaking industry," *CoRR*, vol. abs/2007.07518, 2020. [Online]. Available: https://arxiv.org/abs/2007.07518

[155] P. Isola, J. Zhu, T. Zhou, and A. A. Efros, "Image-to-image translation with conditional adversarial networks," in *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*. IEEE Computer Society, 2017, pp. 5967–5976. [Online]. Available: https://doi.org/10.1109/CVPR.2017.632

[156] M. Sabini and G. Rusak, "Painting outside the box: Image outpainting with gans," *CoRR*, vol. abs/1808.08483, 2018. [Online]. Available: http://arxiv.org/abs/1808.08483

[157] P. Liu, X. Zhang, M. Pistoia, Y. Zheng, M. Marques, and L. Zeng, "Automatic text input generation for mobile testing," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. IEEE / ACM, 2017. [Online]. Available: https://doi.org/10.1109/ICSE.2017.65

[158] H. Zhao, Z. Li, H. Wei, J. Shi, and Y. Huang, "Seqfuzzer: An industrial protocol fuzzing framework from a deep learning perspective," in *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*. IEEE, 2019, pp. 59–67. [Online]. Available: https://doi.org/10.1109/ICST.2019.00016

[159] J. Koo, C. Saumya, M. Kulkarni, and S. Bagchi, "Pyse: Automatic worst-case test generation by reinforcement learning," in *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*. IEEE, 2019, pp. 136–147. [Online]. Available: https://doi.org/10.1109/ICST.2019.00023

[160] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, "Reinforcement learning based curiosity-driven testing of android applications," in *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*. ACM, 2020. [Online]. Available: https://doi.org/10.1145/3395363.3397354

[161] J. P. Inala, S. Gao, S. Kong, and A. Solar-Lezama, "REAS: combining numerical optimization with SAT solving," *CoRR*, vol. abs/1802.04408, 2018. [Online]. Available: http://arxiv.org/abs/1802.04408

[162] Z. Hao, S. Liu, Y. Zhang, C. Ying, Y. Feng, H. Su, and J. Zhu, "Physics-informed machine learning: A survey on problems, methods and applications," 2023.

[163] Y. Bengio, A. C. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 35, no. 8, pp. 1798–1828, 2013. [Online]. Available: https://doi.org/10.1109/TPAMI.2013.50

[164] Y. Zhu, Y. Zhang, H. Yang, and F. Wang, "Gancoder: An automatic natural language-to-programming language translation approach based on GAN," in *Natural Language Processing and Chinese Computing - 8th CCF International Conference, NLPCC 2019, Dunhuang, China, October 9-14, 2019, Proceedings, Part II*, ser. Lecture Notes in Computer Science, J. Tang, M. Kan, D. Zhao, S. Li, and H. Zan, Eds., vol. 11839. Springer, 2019, pp. 529–539. [Online]. Available: https://doi.org/10.1007/978-3-030-32236-6_48

[165] G. Zerveas, S. Jayaraman, D. Patel, A. Bhamidipaty, and C. Eickhoff, "A transformer-based framework for multivariate time series representation learning," in *KDD '21: The 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, Singapore, August 14-18, 2021*, F. Zhu, B. C. Ooi, and C. Miao, Eds. ACM, 2021, pp. 2114–2124. [Online]. Available: https://doi.org/10.1145/3447548.3467401

[166] J. Pennington, R. Socher, and C. Manning, "GloVe: Global vectors for word representation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1532–1543. [Online]. Available: https://aclanthology.org/D14-1162

[167] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Transactions of the Association for Computational Linguistics*, vol. 5, pp. 135–146, 2017. [Online]. Available: https://aclanthology.org/Q17-1010

[168] A. Ushio, L. E. Anke, S. Schockaert, and J. Camacho-Collados, "BERT is to NLP what alexnet is to CV: can pre-trained language models identify analogies?" in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1-6, 2021*. Association

for Computational Linguistics, 2021, pp. 3609–3624. [Online]. Available: https://doi.org/10.18653/v1/2021.acl-long.280

[169] H. Chiang, J. Camacho-Collados, and Z. A. Pardos, "Understanding the source of semantic regularities in word embeddings," in *Proceedings of the 24th Conference on Computational Natural Language Learning, CoNLL 2020, Online, November 19-20, 2020*, R. Fernández and T. Linzen, Eds. Association for Computational Linguistics, 2020, pp. 119–131. [Online]. Available: https://doi.org/10.18653/v1/2020.conll-1.9

[170] T. Shen, J. Mueller, R. Barzilay, and T. S. Jaakkola, "Educating text autoencoders: Latent representation guidance via denoising," in *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, ser. Proceedings of Machine Learning Research, vol. 119. PMLR, 2020, pp. 8719–8729. [Online]. Available: http://proceedings.mlr.press/v119/shen20c.html

[171] K. J. Shih, A. Dundar, A. Garg, R. Pottorf, A. Tao, and B. Catanzaro, "Video interpolation and prediction with unsupervised landmarks," *CoRR*, vol. abs/1909.02749, 2019. [Online]. Available: http://arxiv.org/abs/1909.02749

[172] V. K. Ramaswamy, S. C. Musson, C. G. Willcocks, and M. T. Degiacomi, "Deep learning protein conformational space with convolutions and latent interpolations," *Phys. Rev. X*, vol. 11, p. 011052, Mar 2021. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevX.11.011052

[173] W. Xia, Y. Zhang, Y. Yang, J. Xue, B. Zhou, and M. Yang, "GAN inversion: A survey," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 45, no. 3, pp. 3121–3138, 2023. [Online]. Available: https://doi.org/10.1109/TPAMI.2022.3181070

[174] M. Y. Michelis and Q. Becker, "On linear interpolation in the latent space of deep generative models," *CoRR*, vol. abs/2105.03663, 2021. [Online]. Available: https://arxiv.org/abs/2105.03663

[175] L. Mi, T. He, C. F. Park, H. Wang, Y. Wang, and N. Shavit, "Revisiting latent-space interpolation via a quantitative evaluation framework," *CoRR*, vol. abs/2110.06421, 2021. [Online]. Available: https://arxiv.org/abs/2110.06421

[176] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1412.6572

[177] S. Y. Khamaiseh, D. Bagagem, A. Al-Alaj, M. Mancino, and H. W. Alomari, "Adversarial deep learning: A survey on adversarial attacks and defense mechanisms on image classification," *IEEE Access*, vol. 10, pp. 102 266–102 291, 2022.

[178] M. Ancona, E. Ceolini, C. Öztireli, and M. Gross, "Towards better understanding of gradient-based attribution methods for deep neural networks," in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. [Online]. Available: https://openreview.net/forum?id=Sy21R9JAW

[179] S. Kornblith, M. Norouzi, H. Lee, and G. E. Hinton, "Similarity of neural network representations revisited," in *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. PMLR, 2019, pp. 3519–3529. [Online]. Available: http://proceedings.mlr.press/v97/kornblith19a.html

[180] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing," *ACM Comput. Surv.*, vol. 55, no. 9, pp. 195:1–195:35, 2023. [Online]. Available: https://doi.org/10.1145/3560815

[181] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," in *NeurIPS*, 2022. [Online]. Available: http://papers.nips.cc/paper_files/paper/2022/hash/8bb0d291acd4acf06ef112099c16f326-Abstract-Conference.html

[182] S. M. Lundberg and S. Lee, "A unified approach to interpreting model predictions," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, 2017, pp. 4765–4774. [Online]. Available: http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions

[183] K. K. Wickstrøm, D. J. Trosten, S. Løkse, K. Ø. Mikalsen, M. C. Kampffmeyer, and R. Jenssen, "RELAX: representation learning explainability," *CoRR*, vol. abs/2112.10161, 2021. [Online]. Available: https://arxiv.org/abs/2112.10161

[184] L. Ma, J. Gao, Y. Wang, C. Zhang, J. Wang, W. Ruan, W. Tang, X. Gao, and X. Ma, "Adacare: Explainable clinical health status representation learning via scale-adaptive feature extraction and recalibration," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 01, pp. 825–832, Apr. 2020. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/5427

[185] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 40:1–40:29, 2019. [Online]. Available: https://doi.org/10.1145/3290353

[186] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo,*

Italy, August 30 - September 4, 2015. ACM, 2015, pp. 38–49. [Online]. Available: https://doi.org/10.1145/2786805.2786849

[187] A. Conneau and G. Lample, "Cross-lingual language model pretraining," in *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, 2019, pp. 7057–7067. [Online]. Available: https://proceedings.neurips.cc/paper/2019/hash/c04c19c2c2474dbf5f7ac4372c5b9af1-Abstract.html

[188] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, M. Davaadorj, J. Lamy-Poirier, J. Monteiro, O. Shliazhko, N. Gontier, N. Meade, A. Zebaze, M.-H. Yee, L. K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. Murthy, J. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, N. Fahmy, U. Bhattacharyya, W. Yu, S. Singh, S. Luccioni, P. Villegas, M. Kunakov, F. Zhdanov, M. Romero, T. Lee, N. Timor, J. Ding, C. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, J. Robinson, C. J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C. M. Ferrandis, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, "Starcoder: may the source be with you!" 2023.

[189] S. Storks, Q. Gao, and J. Y. Chai, "Recent advances in natural language inference: A survey of benchmarks, resources, and approaches," 2020.

[190] J. Wei, D. Garrette, T. Linzen, and E. Pavlick, "Frequency effects on syntactic rule learning in transformers," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*. Association for Computational Linguistics, 2021, pp. 932–948. [Online]. Available: https://aclanthology.org/2021.emnlp-main.72

[191] N. Mostafazadeh, M. Roth, A. Louis, N. Chambers, and J. Allen, "Lsdsem 2017 shared task: The story cloze test," in *Proceedings of the 2nd Workshop on Linking Models of Lexical, Sentential and Discourse-level Semantics, LSDSem@EACL 2017, Valencia, Spain, April 3, 2017*. Association for Computational Linguistics, 2017, pp. 46–51. [Online]. Available: https://doi.org/10.18653/v1/w17-0906

[192] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," 2020.

[193] S. Wu, O. Irsoy, S. Lu, V. Dabravolski, M. Dredze, S. Gehrmann, P. Kambadur, D. Rosenberg, and G. Mann, "Bloomberggpt: A large language model for finance," 2023.

[194] U. Naseem, A. G. Dunn, M. Khushi, and J. Kim, "Benchmarking for biomedical natural language processing tasks with a domain specific albert," 2021.

[195] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *CoRR*, vol. abs/2102.04664, 2021. [Online]. Available: https://arxiv.org/abs/2102.04664

[196] B. Zadrozny, "Learning and evaluating classifiers under sample selection bias," in *Machine Learning, Proceedings of the Twenty-first International Conference (ICML 2004), Banff, Alberta, Canada, July 4-8, 2004*, ser. ACM International Conference Proceeding Series, C. E. Brodley, Ed., vol. 69. ACM, 2004.

[197] J. Birch, R. Rivett, I. Habli, B. Bradshaw, J. Botham, D. Higham, P. Jesty, H. Monkhouse, and R. Palin, "Safety cases and their role in ISO 26262 functional safety assessment," in *Computer Safety, Reliability, and Security - 32nd International Conference, SAFECOMP 2013, Toulouse, France, September 24-27, 2013. Proceedings*, ser. Lecture Notes in Computer Science, F. Bitsch, J. Guiochet, and M. Kaâniche, Eds., vol. 8153. Springer, 2013, pp. 154–165.

[198] A. Gardner, J. Kanno, C. A. Duncan, and R. R. Selmic, "Measuring distance between unordered sets of different sizes," in *2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2014, Columbus, OH, USA, June 23-28, 2014*. IEEE Computer Society, 2014, pp. 137–143.

[199] D. Hendrycks and K. Gimpel, "A baseline for detecting misclassified and out-of-distribution examples in neural networks," *CoRR*, vol. abs/1610.02136, 2016. [Online]. Available: http://arxiv.org/abs/1610.02136

[200] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1409.1556

[201] C. Guo, G. Pleiss, Y. Sun, and K. Q. Weinberger, "On calibration of modern neural networks," in *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, 2017, pp. 1321–1330. [Online]. Available: http://proceedings.mlr.press/v70/guo17a.html

[202] C. Olah, A. Satyanarayan, I. Johnson, S. Carter, L. Schubert, K. Ye, and A. Mordvintsev, "The building blocks of interpretability," *Distill*, 2018. [Online]. Available: https://distill.pub/2018/building-blocks/

[203] Y. Li, N. Wang, J. Shi, J. Liu, and X. Hou, "Revisiting batch normalization for practical domain adaptation," in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Workshop Track Proceedings*. OpenReview.net, 2017. [Online]. Available: https://openreview.net/forum?id=Hk6dkJQFx

[204] Q. Chao, H. Bi, W. Li, T. Mao, Z. Wang, M. C. Lin, and Z. Deng, "A survey on visual traffic simulation: Models, evaluations, and applications in autonomous driving," *Comput. Graph. Forum*, vol. 39, no. 1, pp. 287–308, 2020.

[205] N. Ersotelos and F. Dong, "Building highly realistic facial modeling and animation: a survey," *The Visual Computer*, vol. 24, no. 1, pp. 13–30, 2008.

[206] Q. Cheng, Q. Zhang, P. Fu, C. Tu, and S. Li, "A survey and analysis on automatic image annotation," *Pattern Recognit.*, vol. 79, pp. 242–259, 2018.

[207] A. Shafaei, M. Schmidt, and J. J. Little, "A less biased evaluation of out-of-distribution sample detectors," in *30th British Machine Vision Conference 2019, BMVC 2019, Cardiff, UK, September 9-12, 2019*. BMVA Press, 2019, p. 3. [Online]. Available: https://bmvc2019.org/wp-content/uploads/papers/0333-paper.pdf

[208] M. Gyllenhammar, R. Johansson, F. Warg, D. Chen, H.-M. Heyn, M. Sanfridson, J. Söderberg, A. Thorsén, and S. Ursing, "Towards an Operational Design Domain That Supports the Safety Argumentation of an Automated Driving System," in *10th European Congress on Embedded Real Time Software and Systems (ERTS 2020)*, TOULOUSE, France, Jan. 2020. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02456077

[209] W. M. Kouw, "An introduction to domain adaptation and transfer learning," *CoRR*, vol. abs/1812.11806, 2018. [Online]. Available: http://arxiv.org/abs/1812.11806

[210] V. Tran, "Selection bias correction in supervised learning with importance weight. (l'apprentissage des modèles graphiques probabilistes et la correction de biais sélection)," Ph.D. dissertation, University of Lyon, France, 2017. [Online]. Available: https://tel.archives-ouvertes.fr/tel-01661470

[211] A. Liu and B. D. Ziebart, "Robust classification under sample selection bias," in *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds., 2014, pp. 37–45. [Online]. Available: http://papers.nips.cc/paper/5458-robust-classification-under-sample-selection-bias

[212] M. Dudík, R. E. Schapire, and S. J. Phillips, "Correcting sample selection bias in maximum entropy density estimation," in *Advances in Neural Information Processing Systems 18 [Neural Information Processing Systems, NIPS 2005, December 5-8, 2005, Vancouver, British Columbia, Canada]*, 2005, pp. 323–330. [Online]. Available: http://papers.nips.cc/paper/2929-correcting-sample-selection-bias-in-maximum-entropy-density-estimation

[213] W. Fan and I. Davidson, "On sample selection bias and its efficient correction via model averaging and unlabeled examples," in *Proceedings of the Seventh SIAM International Conference on Data Mining, April 26-28, 2007, Minneapolis, Minnesota, USA*. SIAM, 2007, pp. 320–331. [Online]. Available: https://doi.org/10.1137/1.9781611972771.29

[214] J. Chen, Y. Chang, B. Hobbs, P. J. Castaldi, M. H. Cho, E. K. Silverman, and J. G. Dy, "Interpretable clustering via discriminative rectangle mixture model," in *IEEE 16th International Conference on Data Mining, ICDM 2016, December 12-15, 2016, Barcelona, Spain*, F. Bonchi, J. Domingo-Ferrer, R. Baeza-Yates, Z. Zhou, and X. Wu, Eds. IEEE Computer Society, 2016, pp. 823–828.

[215] L. van der Maaten and G. Hinton, "Visualizing data using t-SNE," *Journal of Machine Learning Research*, vol. 9, pp. 2579–2605, 2008. [Online]. Available: http://www.jmlr.org/papers/v9/vandermaaten08a.html

[216] X. Gu and A. Easwaran, "Towards safe machine learning for CPS: infer uncertainty from training data," in *Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems, ICCPS 2019, Montreal, QC, Canada, April 16-18, 2019*. ACM, 2019, pp. 249–258.

[217] D. J. McDuff, S. Ma, Y. Song, and A. Kapoor, "Characterizing bias in classifiers using generative models," H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett, Eds., 2019. [Online]. Available: http://papers.nips.cc/paper/8780-characterizing-bias-in-classifiers-using-generative-models

[218] D. J. McDuff, R. Cheng, and A. Kapoor, "Identifying bias in AI using simulation," *CoRR*, vol. abs/1810.00471, 2018. [Online]. Available: http://arxiv.org/abs/1810.00471

[219] E. Thorn, S. C. Kimmel, and M. Chaka, Sep 2018, ch. A Framework for Automated Driving System Testable Cases and Scenarios, tech Report, DOT HS 812 623. [Online]. Available: https://rosap.ntl.bts.gov/view/dot/38824

[220] H. Lakkaraju, E. Kamar, R. Caruana, and E. Horvitz, in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*. AAAI Press, 2017, pp. 2124–2132. [Online]. Available: http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14434

[221] I. Giurgiu and A. Schumann, "Additive explanations for anomalies detected from multivariate temporal data," in *Proceedings of the 28th ACM International Conference on Information and Knowledge Management, CIKM 2019, Beijing, China, November 3-7, 2019*, W. Zhu, D. Tao, X. Cheng, P. Cui, E. A. Rundensteiner, D. Carmel, Q. He, and J. X. Yu, Eds. ACM, 2019, pp. 2245–2248.

[222] L. Antwarg, B. Shapira, and L. Rokach, "Explaining anomalies detected by autoencoders using SHAP," *CoRR*, vol. abs/1903.02407, 2019. [Online]. Available: http://arxiv.org/abs/1903.02407