Thesis for The Degree of Licentiate of Engineering

Deep learning based simulation for automotive software development

Dhasarathy Parthasarathy

Division of Computing Science
Department of Computer Science & Engineering
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden, 2022

Deep learning based simulation for automotive software development

Dhasarathy Parthasarathy

Copyright ©2022 Dhasarathy Parthasarathy
except where otherwise stated.
All rights reserved.

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 2022.

”

,,,,

Abstract

Context – The automotive industry is in the midst of a new reality where software is increasingly becoming the primary tool for delivering value to customers. While this has vastly improved their product offerings, vehicle manufacturers are increasingly facing the need to continuously develop, test, and deliver functionality, while maintaining high levels of quality. One important tool for achieving this is simulation-based testing where the external operating environment of a software system is simulated, enabling incremental development with rapid test feedback. However, the traditional practice of manually specifying simulation models for complex external environments involves immense engineering effort, while remaining vulnerable to inevitable assumptions and simplifications. Exploiting the increased availability of data that captures operational environments and scenarios from the field, this work takes a deep learning approach to train models that realistically simulate external environments, significantly increasing the credibility of simulation-driven software development.

Contributions – **First**, focusing on simulating the input dependencies of automotive software functions, this work uses techniques of deep generative modeling to develop a framework for realistic test stimulus generation. Such models are trained self-supervised using recorded time-series field data and simulate the input environment much more credibly than manually specified models. With the credibility of stimulus generation being an important concern, an important concept of *similarity as plausibility* is introduced to evaluate the quality of generation during model training. **Second**, this work develops new techniques for sampling generative models that enable the controlled generation of test stimulus. Allowing testers to limit the range of scenarios considered for testing, the Metric-based Linear Interpolation (MLERP) sampling algorithm automatically chooses test stimuli that are verifiably similar to a user-supplied reference, and therefore measurably credible. While controllability eases the design of tests, credibility increases trust in the testing process. **Third**, recognizing that sampling may be an inefficient process for stimulus generation, this work develops a technique that extracts properties from actual code under test in order to automatically search for appropriate test stimuli within the specified range of test scenarios. **Fourth**, further addressing the question of credible stimulus generation, this work introduces techniques that examine training data for biases in sample representation. **Overall**, by taking a data-driven deep learning approach, techniques and tools developed in this work vastly expands the credibility of incremental automotive software development under simulated conditions.

Future work – Work in the future plans to (1) expand the scope from open-loop stimulus generation to include output dependencies including elements of feedback in closed-loop software systems, and (2) use trained models, instead of hand-coded rules, to understand and extract properties from code under test.

Keywords – automotive software testing, generative adversarial networks, latent space arithmetic, explainable AI, sample selection bias

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 Karl Bäckström and, the closest of all, Anton Johansson. What I've learned from our interactions, Anton, cannot be measured, even by a sharp mathematician like you.

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. In addition to Henrik Lönn, my industrial thesis adviser and terminology engine, and Cecilia Ekelin, my go-to for any discussion, I thank the ever-dependable Abhineet Tomar and the latest gang of fellow-explorers Rasmus, Binay, Robin & co. I then thank my department manager Dan Walhström and, most importantly, my group manager, and my rock, Daniel Karlsson. The way you unleash me, Daniel, with all the support and encouragement I could ever need, is an education in management. I reserve special thanks for my erstwhile manager Ted Kruse for giving true meaning to this quest. After all, 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. Quite simply, I cannot express how much I owe you guys. 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.

List of Publications

Appended publications

This thesis is based on the following publications:

[A] **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, developed the algorithms, gathered datasets, designed, trained and evaluated the model, conducted most of the experiments and analysis, and wrote most of the paper.

[B] **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, developed the algorithms, gathered datasets, designed, trained and evaluated most of the model, conducted the experiments and analysis, and wrote most of the paper.

[C] **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, collaborated on developing the overlap index, gathered datasets, trained and evaluated models, conducted the experiments and analysis, and wrote most of the paper.

Contents

Chapter 1

Introduction

What drives the commercial vehicle? Over two centuries of its existence, advances in the vehicle have arguably been driven mainly by improvements in its structural composition and its fundamental dynamics. In just the last few decades, however, the automotive engineering landscape has changed dramatically. Not only is the internal combustion engine, long considered the soul of the vehicle, beginning to lose its primacy among propulsion technologies, the growing use of automatic control has vastly expanded the capabilities of a vehicle. The modern truck, bus, and construction equipment is safer, multi-functional, digitalized, connected, (relatively more) energy efficient, and is a critical part of complex industrial systems. At a time when even the technology for automating the driver/operator lies within grasp, it is safe to say that the modern commercial vehicle is being increasingly driven by electronics and software [1].

An increase in the use of software no doubt improves the efficiency of delivering solutions as well as the adaptability of vehicles during field operation. Nonetheless, vehicle manufacturers remain in a constant race to meet rapidly evolving market demands, especially when they field a large product portfolio, targeting a wide range of applications. Under these circumstances, automotive software engineering proves to be especially challenging. 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. 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 possible usage scenarios and consequently design, develop, and test software-intensive solutions, all under the shadow of constant market pressure, is immense.

One recent development, which is rapidly improving operational awareness is, of course, 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. Such unprecedented operational insights in turn have the potential to significantly improve the process of automotive software engineering, and particularly software testing. 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 the automotive electronic system and how it models vehicle state and behavior.

1.1 The vehicle electronics and software system

Most functions in modern vehicles are electronically controlled and the overall control system that governs this process is referred to as the automotive Electrical/Electronics (E/E) system [2]. The E/E system is normally realized as a distributed control system where a set of Electronic Control Units (ECUs) administer, monitor, and control different domains of vehicle functionality. Typical functionality domains include [3] (1) chassis, which controls aspects such as brake and steering, (2) powertrain, which controls the engine or electric propulsion motors, (3) body electronics, which controls functions like door locks, windows, ventilation, etc. and (4) driver assistance, which controls assistance functions like lane keeping and safety functions like emergency braking. These control-centric domains are normally complemented by the information-centric domains of (1) connectivity, which handles telematics functions and (2) infotainment that handles information display and entertainment functions. ECUs, controlling various aspects of different domains, are linked together using different networking technologies, the most significant of which are Controller Area Network (CAN) and Ethernet. The overall layout or topology of ECUs can take different forms depending upon the chosen E/E architecture [4]. For instance, the topology of a domain centralized E/E system could look like Figure 1.1 with one main ECU per domain.

While the E/E system may have traditionally been dominated by its electric and electronic components, the focus is rapidly shifting towards software. With a majority of new functionality being introduced through software [5], the automotive industry has been parallelly evolving a software-centric view of the E/E system. The software perspective considers ECUs mainly as execution platforms on which Software Components (SWCs) are deployed. A SWC implements one part of a larger control process and collaborates with other SWCs by exchanging *signals*. The example in Figure 1.2 illustrates a possible control process. A `BrakePedalSensor` SWC samples the brake pedal to signal `brake_pedal_position` to the `BrakeBlending` SWC. Using an algorithm to distribute the driver's brake request among different retarders, the `BrakeBlending` SWC calculates `right_front_pressure` and `left_front_pressure`, and signals them to a `FrontAxleBrakeActuator` SWC. Physically, SWCs involved in a functional flow may be allocated to different ECUs, in which case, signals are transported through the links which network them. Such a pattern of implementation reflects the basic principles of the AUTOSAR [6] industry standard.

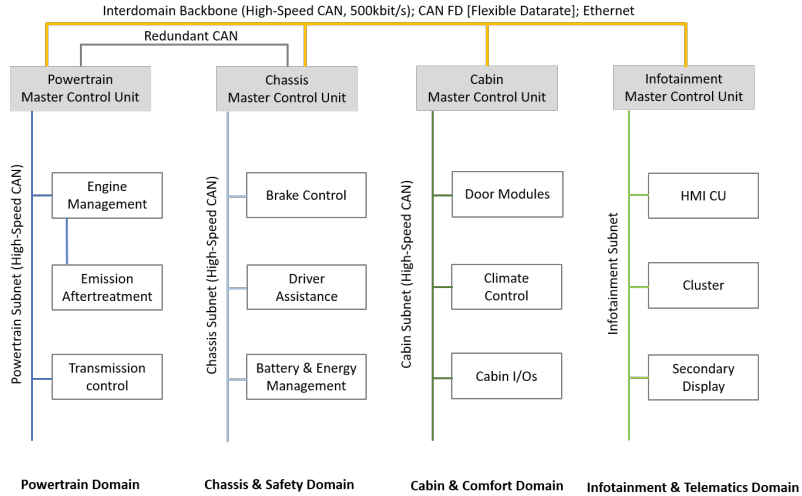

Figure 1.1: Example of an automotive E/E system from a hardware perspective

1.2 Signals and functions

As units that channel all transactions between SWCs, signals form the connective tissue of automotive software systems. Since a signal is an element of significance, both in this work and in general automotive software engineering, it helps to state the following definition.

Definition 1 *A signal* $(\mathbf{s},\ \mathbf{x}), \mathbf{x} \in \mathbb{R}$ *is the observed, instantaneous snapshot of one aspect of vehicle state* $\mathbf{s} \in S$*, with a set of possible states* S*. The term refers to both the unit of information and the means of its transaction between software components.*

A signal is therefore 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*

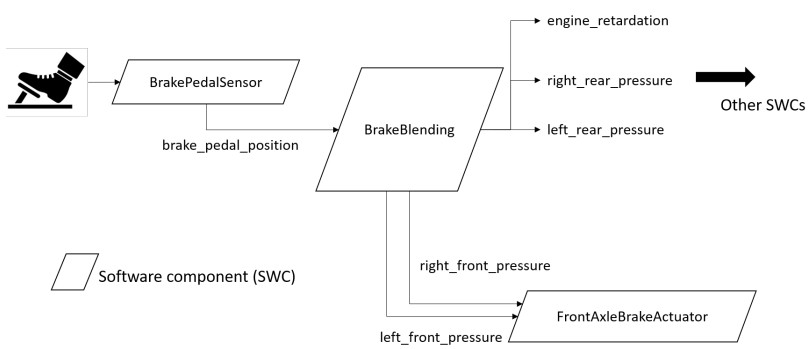

Figure 1.2: Example of an automotive function from a software perspective

could be a signal. It captures the instantaneous state of the brake pedal, presumably by observing the output port of the `BrakePedalSensor` SWC. Often, especially if the name of a signal is understood or unnecessary, a signal is simply represented by its value $\mathbf{x}$. Upon observing Figure 1.2, one can readily note that a SWC is simply a unit that transforms one set of signals into another, and useful vehicle functionality is realized by simply chaining a set of them together. Such is the predominance of this functional paradigm in automotive system design, that it helps to define the vehicle function.

Definition 2 *A vehicle function* $\bar{\mathbf{y}} = \mathbf{v}(\bar{\mathbf{x}})$ *maps a vector of input signals* $\bar{\mathbf{x}}$ *into a vector of output signals* $\bar{\mathbf{y}}$*, consequently manipulating vehicle state. It can also be a process that is realized as a composition of a set of individual vehicle functions.*

In fact, SWC is simply AUTOSAR terminology for a function that is programmed as code, usually in C. Thus, while signals embody vehicle state in the E/E system, functions actively manipulate state. A high-end commercial vehicle has hundreds of functions, which tend to fall under a handful of categories. Based upon the nature of operation, vehicle functions can be categorized into (1) *Control functions*, which regulate some entity or phenomenon, and (2) *Monitoring functions*, which observe an entity or phenomenon and report. Based upon their means of operation, control functions can be further categorized into (1) *Closed loop*, if there is active regulation with feedback, and (2) *Open loop*, if there is only passive regulation without feedback. At times, vehicle functions may be aided by support functions like diagnostics or logging, both of which take signals as input but produce diagnostic codes and trace logs respectively.

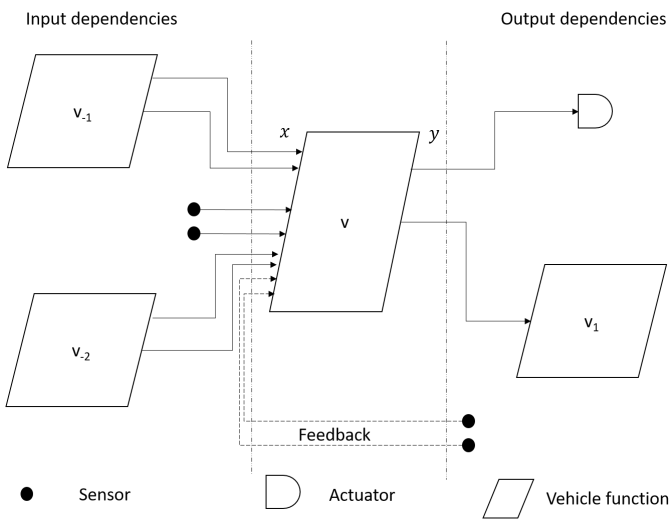

Figure 1.3: A vehicle function and its dependencies. Functions v_{-n} and v_n refer to upstream and downstream dependencies respectively

Let us now consider a typical vehicle function and its input and output dependencies (Figure 1.3). One part of its input consists of signals routed from

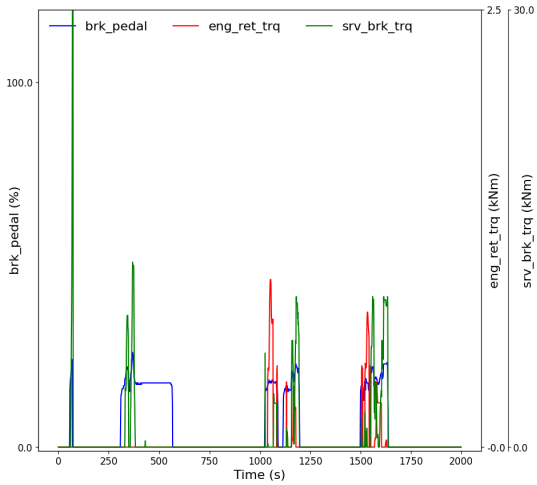

Figure 1.4: A trace of 3 signals capturing brake blending behavior

necessary upstream functions, while another part consists of sensors used to monitor some local phenomenon. Note that analog sensors are sampled (perhaps by a helper function) before they are included to the set of input signals. Upon assembling its input signals $\bar{\mathbf{x}}$, the function manipulates it to produce a set of output signals $\bar{\mathbf{y}}$, some of which may be information intended for downstream functions, while others may control an actuator. As noted earlier, closed loop control functions include an element of feedback, where additional sensors are used to measure the consequences of a control action. Such feedback can of course be included spatially into a set of input signals, but they expose the need for the temporal dimension in describing a function.

A vehicle function may map one signal vector into another, but this mapping is not instantaneous. Apart from finite delays introduced by transients in mechatronic elements, along with sampling and scheduled execution of software elements, a function may simply be designed to act only at some future time. For instance, it may adopt a control or monitoring strategy that involves assessing a relatively long-term history of its input stream. To better describe the interaction between signals and functions, it is helpful to define the notion of a signal trace.

Definition 3 *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* $\mathbf{N}, \mathbf{T} \in \mathbb{Z}^{+}$.

A signal trace that includes the input and output signals of a function represents one instance of its *observable behavior*, including elements of feedback, during a certain time window. For instance, the trace in Figure 1.4 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.

1.3 Simulation for continuous engineering

Prior to being deployed in the vehicle, the function v in Figure 1.3 goes through an elaborate engineering process. Text-book vehicle function engineering follows a sequential process involving the following broad phases (1) *Requirements*, where the intended behavior of the function is specified, (2) *Design*, where the key elements of a solution that achieves the intended behavior are envisioned, (3) *Implementation*, where the design is realized as software and hardware, (4) *Verification*, which checks whether implemented behavior meets intended behavior, and (5) *Integration*, where the function is deployed in its intended environment. Such a sequential process is seldom followed strictly since it is often impractical, for instance, either to implement only after the complete design is available, or to begin verifying only after the complete implementation is available. Especially, with vehicle manufacturers facing increasing pressure to rapidly deliver quality functionality, it has become even more important to loop through the phases quickly.

Automotive software development is increasingly trending towards incremental development, combined with continuous verification and integration [7]. Under such a continuous development model, one key requirement is to be able to isolate the function and its dependencies, so that it can be developed and tested in a fairly controllable manner to ensure repeatability and ease debugging. Such a sandbox or rig can be setup in multiple ways, mainly by manipulating the *integration levels* of its constituent components. A component can be considered to be highly integrated if it is in a form that is closest to its final rendering. Setting up a rig involves choosing integration levels along two dimensions (Figure 1.5) - (1) *vertical*, which pertains to functions and their execution environment, and (2) *horizontal*, which pertains to their input and output dependencies. This means that achieving the highest level of integration along both dimensions is only possible by using a real vehicle, with real software and hardware, and by operating it on a live mission. While this may be the ultimate 'rig' for software development, its time and cost-intensive nature means that incremental development and continuous integration is only possible at a very low cadence. Faster loops through the engineering process requires 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. To achieve this, we turn to *simulation*.

When it comes to integrating a function into a rig, there are three recognized levels [8], 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 in 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 [9], since in these cases the behavior and the execution environment are respectively simulated. A SIL rig, for instance, can provide an environment for running control software directly on a developer

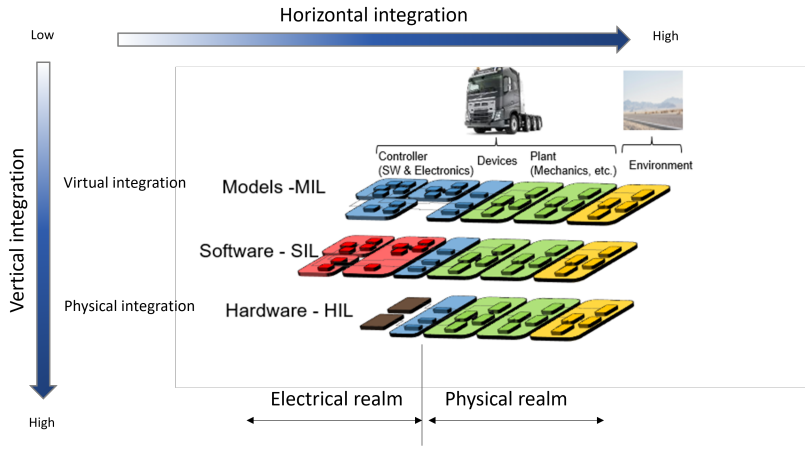

Figure 1.5: Function development rigs and their levels of integration

machine. It can do this by replacing the embedded microcontroller with a server-grade processor, and by porting necessary elements of the AUTOSAR Operating System (OS) to, say, Windows. By thus simulating embedded OS and hardware, it is possible to integrate the SWCs of a function with all dependent SWCs 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 *scope* of dependencies included in the rig. In elucidating horizontal scope, it is necessary to map the range of physical phenomena with which a vehicle function interacts. The software implementation of the vehicle function (the SWCs) resides and executes in an embedded environment. Such an electronic realm can be thought as 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 own 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 the phenomena that the function seeks to interact with. Part 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. The other part, like the road surface and maybe even the driver, lie outside. While scoping is often 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/operator where the scope also includes human interactions with the vehicle system.

For function development at high cadence, there is a clear necessity for simulating dependencies even in the horizontal dimension. Horizontally, the direst

needs for virtual integration comes from the need to simulate environments in the physical realm. For instance, the development of brake blending depends not just upon the state of the brake ECU, but also (among others) upon that of the pneumatic circuit. 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 [10] or Modelica [11] being common examples. When the horizontal scope is wide, manually specifying simulation models for a wide range of phenomena becomes a formidable challenge.

1.4 Practical limits for specifying behavior

The scope 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. When the model includes 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 does 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 increases 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 helps 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, of course, 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, all of which amounts to high engineering costs. 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 1.6), 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.

This work focuses solely on simulating the behavior elements in the horizontal dimension namely, devices, plant and the environment. It does not consider

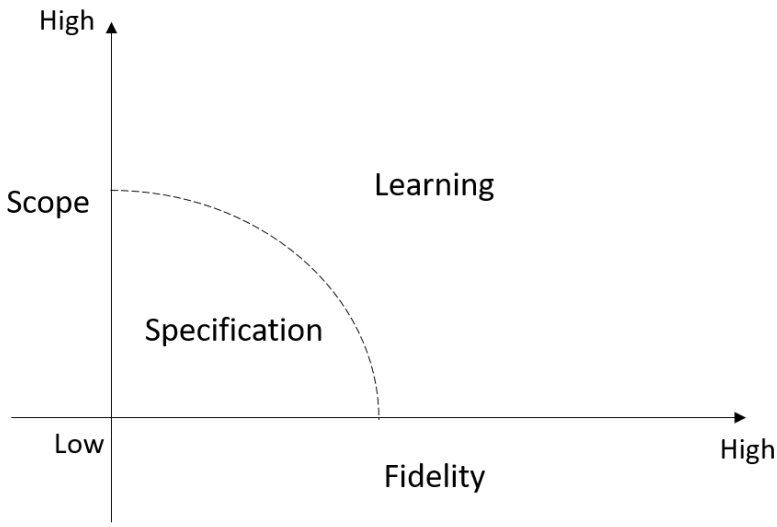

Figure 1.6: When high scope and fidelity are required, learning a simulation model may be easier than manually specifying it

simulating vertical dependencies like the execution environment. We make this explicit using the following definition.

Definition 4 *A simulation model (in this work) refers to a manually defined executable specification, describing the behavior of input and/or output dependencies of a vehicle function. Such a model virtually integrates with the function by aligning with corresponding signals in its I/O interface.*

1.5 Learning behavior from signal traces

Let us consider, once again, a vehicle function and its dependencies, but in a slightly different form. Figure 1.7 now represents a system of M vehicle functions, collectively processing a superset vector of input signals $\bar{\mathbf{x}}$, producing a superset vector of outputs $\bar{\mathbf{y}}$. Such a setup is typical for functions belonging to one functionality domain. The collective input and output dependencies of the system are represented by the functions g and h respectively with the overall I/O interface of the system represented by $\bar{\mathbf{w}}$ and $\bar{\mathbf{z}}$. Feedback, if any, is routed back from the output dependencies and included in $\bar{\mathbf{x}}$.

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. Moreover, a modern vehicle has a rich array of sensors which can be consciously

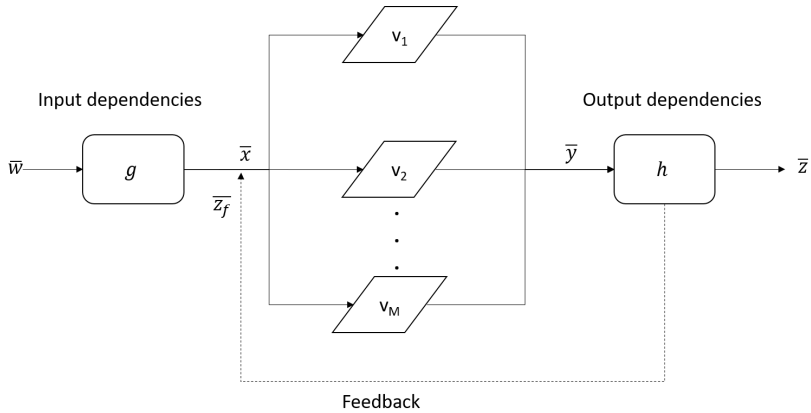

Figure 1.7: A system of vehicle functions

routed into the signal traffic. Expanding the trace to include such sensor data ensures that signals are recorded with sufficiently detailed context. Such a trace provides remarkably detailed information about the operational behavior of this system, and has become the de facto structure for vehicle operation data.

Ignoring feedback for the moment, let us then denote W and X as traces of signal vectors $\bar{\mathbf{w}}$ and $\bar{\mathbf{x}}$ recorded over T time steps. These respectively represent the inputs that g receives from sensors or upstream vehicle functions, and the output it produces, which is applied as stimuli for functions v_n. One mapping from W to X can then be seen as one sample from the conditional probability distribution $X \sim G(X|\, W)$, where G is the distribution of all possible mappings, i.e., the behavior. In cases where the scope and fidelity are manageable, we noted earlier that it may be possible to specify this behavior with reasonable accuracy. On the other hand, given a sufficiently large variety of detailed traces W and X, a machine learning approach can be used to train a model which approximates the true behavior G, even when scope and fidelity are high. The following chapter summarizes our work in using recorded signal traces to train deep generative models which approximate the behavior of complex vehicular systems and thereby provide realistic test stimulus for the vehicle functions in the system. Generative models, so trained, effectively simulate complex dependencies of vehicle functions without any need for manual specification of behavior. Techniques developed in this work significantly improves the credibility of simulating vehicle function dependencies, which, in turn, allows automotive software development at increased cadence, without compromising quality.

Chapter 2

Summary of findings

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 which, at its foundation, composes an elaborate hierarchy of simple concepts to learn a complex concept [12]. 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 Deep Neural Networks (DNNs), 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. A famous early application of this seemingly simple formula was the AlexNet [13] image classification model, which dramatically outperformed competing approaches to win the 2012 edition of the ImageNet [14] 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 applications as varied as natural language processing and drug design. In yet another extension to a new domain, this work takes a deep learning approach to simulate vehicle behavior. Before describing techniques used to train such behavior, we begin with a few preliminaries.

2.1 Deep learning – a typical approach

Primary elements of a deep learning approach include - (1) a learning task, (2) a dataset that provides enough examples, and (3) a training objective that guides and evaluates the learning process. If we consider the archetypal classification problem, then the learning task would be to predict the label or the category y to which an input vector x belongs. For instance, the input x could be an image of a cat or a dog and $y \in \{0, 1\}$ would be a tag that identifies it as one or the other. Examples are provided as a dataset of K images $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{K}$. Next comes another critical element of the recipe – a DNN f, which is the object of training. A DNN is an L-layer stack of differentiable[1] non-linear operations, the most traditional form of which is found below.

$$\begin{aligned} f^l(x) &= \sigma(W_l f^{l-1}(x) + b_l), l > 1,\ f^1(x) = x \\ \hat{y} &= \frac{e^z}{\sum_{c=1}^{2} e^{z_c}},\ z = f(x),\ f = \circ_{l=1}^{L} f^l \end{aligned} \tag{2.1}$$

Here, the fundamental operation of f is an affine map followed by some non-linearity σ. The set of weights W_l and biases b_l collectively form the model's parameters $\Theta = \{(W_l, b_l)\}_{l=1}^{L}$. With a 2-class classification objective, the final layer f^L is designed to output a 2-dimensional vector z. The final layer output is then squashed into a probability distribution using the softmax function to give the vector $\hat{y}$, which is the model's prediction of the input x belonging to cat and dog categories. The DNN f is thus able to model the classification task end-to-end. At the start of the training process, model parameters Θ are randomly initialized, with the model giving random predictions. The objective of training is to find parameters Θ^* with which the model performs the classification task at an acceptable level of accuracy on a held-out validation set. Holding out a validation set from the larger dataset $\mathcal{D}$, and using it to test the model is a technique which ensures that model performance generalizes beyond simply the training set. A key element, the next one in the deep learning recipe, is the training objective or loss function, that guides the training process. This differentiable function $\mathcal{L}$ captures the discrepancy between the prediction $\hat{y}$ and the ground truth y and evaluates to 0 when they are identical. For classification, it is common to use the categorical cross entropy loss shown below.

$$\Theta^* = \min_{\Theta} \sum_{\{(x_i, y_i)\} \subset \mathcal{D}} \mathcal{L}(\hat{y}_i, y_i; \Theta),\ \mathcal{L} = -\sum_{c=1}^{2} y_i^c\ log(\hat{y_i}^c) \tag{2.2}$$

With differentiable f and $\mathcal{L}$, it becomes possible to use a gradient-descent search for Θ^* (where $\mathcal{L} \to 0$), the essence of which is shown below.

$$\theta = \theta - \alpha \frac{\partial \mathcal{L}}{\partial \theta} \mid \forall\ \theta \in \Theta \tag{2.3}$$

Using a learning rate α and the technique of back propagation for distributing the loss gradient to parameters across all layers of f, the stochastic gradient

[1] In the context of a DNN, we mean differentiable almost everywhere

descent process traverses the Θ-space, moving closer to the target by progressively minimizing $\mathcal{L}$ on batches of training data. Throughout the process, the classification performance is parallelly assessed on the validation set. When this reaches an acceptable level, training terminates, whereupon the network f has learned to accomplish the task of labeling. While this example illustrates a recipe for deep learning using a simple example, learning the behavior of vehicle systems differs in several aspects. Forthcoming sections identify these distinctions and detail how the recipe is customized for the new task.

2.2 Simulating vehicle behavior - scope and objectives

Section 1.5 introduced the overall idea of a system of vehicle functions and the idea of simulating its dependencies by learning their behavior using signal traces. In this work, however, we consider a slightly narrower scope. First, we prioritize the simulation of input dependencies alone and set aside those at the output. Further, we prioritize simulating the open-loop input signals of the functions and set aside elements of feedback. With this scope, the system of vehicle functions takes the form shown below.

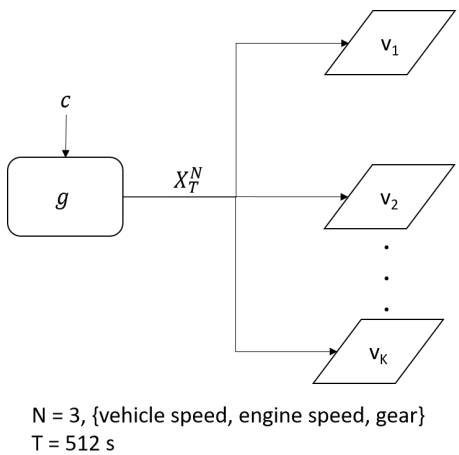

Figure 2.1: System of vehicle functions considered in this work

The new scope is still quite broad and includes all monitoring and open-loop control functions. The scope can also include close-loop control functions, if the rig includes alternative means for simulating output dependencies and feedback. Having scoped the problem, we define the first objective of this work.

Objective 1 *Generate open-loop stimuli as traces that plausibly approximate real input signal traffic, which a given set of functions is likely to encounter during actual field operation.*

Thus, the primary objective is not simply to train a model that learns the behavior of the input dependency. It is also put to use for generating plausible instances of input behavior to credibly test the system of functions. Considering the fact that a vast proportion of stimuli is manually specified in practical rig-based testing, this objective addresses an urgent gap in state-of-practice virtual integration.

Manual specification at the input takes two main forms. One form, as noted earlier, are behavioral specifications of the input dependency. An example would be a simulation model that specifies the fluid dynamics of exhaust gases, which is virtually integrated and executed to create test stimulus for after-treatment control. Another form, commonly seen in rig-testing, would be using hand-crafted traces test stimuli. For instance, in order to test a function that evaluates fuel-efficient driving, one can specify traces of acceleration, brake pedal, and other related signals to simulate the driver. Under high scope and fidelity needs, either form of specification is less likely to be credible. However, such specification does have an important advantage – manually designed or specified test stimuli makes it easier to specify a test oracle. The systematic coverage of a known set of scenarios makes it easy to understand test feedback and eases debugging. Thus, while we turn to trained models that credibly approximate real behavior with high fidelity and scope for test stimulus generation, we still require means to be able to manually control the generation process so that it takes place systematically. This leads to our next objective, stated below.

Objective 2 *The generation of open-loop stimuli must be controllable so that a given set of functions can be systematically subjected to realistic input scenarios.*

To meet this objective, we add condition c in the setup to help control the generation. When developing functions in a rig, the two most important needs are credibility and controllability. It is clear that the stated objectives assign primacy to exactly these needs.

A slight modification in the setup is that, as seen in Figure 2.1, the I/O interface of the input dependency g is combined into a single trace. Modeling g simply as a generator instead of a map usually affects neither behavior modeling nor test stimulus generation. Consider the trace in Figure 2.2 which depicts one behavioral instance of the driveline in terms of three signals – selected gear, engine speed, and vehicle speed. One way to model the driveline would be to consider the engine speed and the gear as input signals, and the vehicle speed as the output signal. Combining them into a single trace, however, keeps this causality and, more importantly, the overall driveline behavior they represent intact and learnable. Also, when such a trace is produced as a test stimulus, the presence of redundant signals clearly does no harm. On the other hand, if there is a specific need to be able to map from one set of signals to another one can, as we show later, treat input signals as a condition c.

Yet another important aspect of system scope would be the influence of observability in learning behavior. The trace X_T^N in Figure 2.2, combining $N = 3$ signals represents one instance of driveline behavior captured over $T = 512$

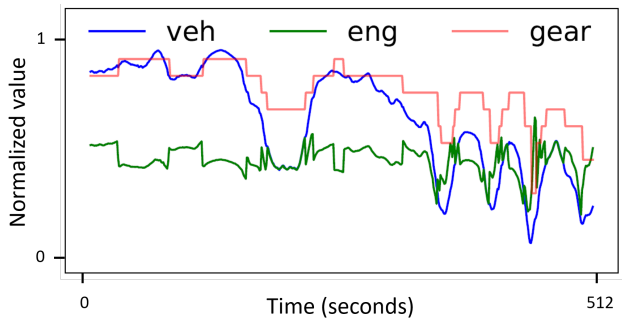

Figure 2.2: A trace of 3 signals capturing driveline behavior

time steps, with factors N and T representing important elements of spatial and temporal observability. While it is clear that there are many more aspects of state, like engine torque, retardation, or road inclination, that combine to represent driveline behavior, there are always practical limits in observing them. Perhaps the hardest limit comes from the fact that signals are observable only at the I/O interface of SWCs, and revealing a signal incurs a finite cost. Even if some state is exposed as a signal, including it into a recording campaign and sampling it at a certain rate incurs additional cost. Moreover, some exposed signals may even be too sensitive to record. For instance, raw GPS data is often considered personal data which can lead to the identification of a driver, and its recording may therefore be limited. Thus, the richness of information included in a set of traces is inevitably traded-off with the costs involved in capturing them. Inevitably, this places commensurate limits on learning the 'true' behavior of the underlying phenomenon. Traces used as training data in this work are limited to the configuration shown in Figure 2.2

2.3 Simulating vehicle behavior – approach

Section 2.1 identified several key elements of a deep learning recipe. Having defined the scope and objectives of simulating vehicle behavior, we now discuss how the basic learning recipe is customized to learning it. Objectives 1 and 2, concisely represented in (2.4), make it clear that the main task is to train a model which, given a condition, generates realistic test stimuli for a set of vehicle functions. Meaning, with a distribution of input traces $X \sim G$, the task is to train a model g that controllably generates a trace $\hat{X}$, which is a plausible sample from the distribution G that could have been, but was not (necessarily) recorded.

$$\hat{X} = g(c) \tag{2.4}$$

Experiments towards achieving these objectives are distributed among the appended papers in the following manner. Objective 1 is the primary concern of both **Paper A** and **Paper B**. In approaching Objective 2, **Paper A** mainly considers the case where the condition $c \coloneqq X \sim G$ is yet another trace. **Paper B**, on the other hand, considers the case where conditions $c \sim C$ are simplified

specifications of traces. **Paper C**, however, does not address the objectives from a training perspective, but does so from the perspective of explainability.

2.3.1 The learning task

Unlike the model $\hat{y} = f(x)$ in the basic recipe which models a discriminative task, g models a generative task. That is, while the main objective of f is to model the *image* $\rightarrow$ *label* conditional distribution $P(y \mid x)$, that of model g is to approximate the *condition* $\rightarrow$ *trace* distribution $P(X \mid c)$[2]. Generative modeling has a long history involving a variety of techniques [15]. More recently, the Generative Adversarial Networks (GAN) [16] and Variational Autoencoder (VAE) [17] frameworks, and their derivatives, have yielded impressive results in generative tasks on a wide range of modalities like images and text. With its focus on generating signal traces, this work falls under the general area of time series generation, which has also begun to see the application of these frameworks [18]. Thus, in achieving Objective 1, learning tasks in both **Paper A** and **Paper B** largely follow those of GAN and VAE frameworks, with some crucial extensions. In meeting Objective 2, however, each paper defines the learning task differently depending upon the nature of the condition c_i.

[A] In **Paper A**, the generative model g^A is tasked to generate a trace $\hat{X}_B$ that is different, but still resembles a user-applied trace X_A in some user specified manner. This is a particularly useful form of controlled generation, where a user enriches the test by asking the model to improvise within the specified 'neighborhood' of an interesting trace. In this case, since the target and condition come from the same distribution, the generative model is tasked to approximate only this distribution G. While, the generative model g^A is tasked to learn unconditionally, the model is still *sampled* conditionally by plugging in a trace X_A as the condition.

[B] In the previous case, the simplest way to apply a condition is to replay a recorded trace X_A. Otherwise, one would have to resort to hand-crafting traces which, in cases of high scope and fidelity, could be prohibitively difficult. To ease stimulus design without compromising credibility, **Paper B** considers cases where conditions follow a distribution that is distinct from G. Therefore, g^B is tasked to explicitly model the conditional distribution $P(X \mid c)$. In this case, it is also important to note that one condition c can map to different traces $\hat{X}$. Such one-to-many mapping, also referred to as multimodal generation, is also an important component of the learning task.

The nature of conditions C decides the level of complexity in test stimulus design. One way to simplify stimulus design would be to define the conditions in an *upstream* domain. For instance, C could be a distribution of driving trajectories, i.e., traces capturing lateral, longitudinal, and vertical displacement

[2]Though both conditional distributions resemble discriminative tasks, the latter is conventionally termed generative since it models the observable X given a target c

of the vehicle over T time steps. Then the generative model g can be tasked to map a trajectory into a set of downstream driveline behaviors of the form in Figure 2.2. This allows stimulus generation in what is perhaps a more useful end-to-end fashion. Another way to simplify stimulus design is to present an *abridged* form of a trace as the condition. To help achieve this, **Paper B**, defines the idea of a *template*, which is a piece-wise linear, univariate, and therefore a very loose approximation of a trace, that is easy to specify. The model g^B is then tasked to take this skeletal trace and generate its details, showing yet another way to simplify the design and generation of test stimulus.

2.3.2 Training data

In **Paper A**, the generative model is trained using a dataset $\mathcal{D}^A = \{X_i\}_{i=1}^K$ of K signal traces representing driveline behavior. **Paper B**, with the task of *template* $\rightarrow$ *trace* generation, assembles a set of traces in the form $\mathcal{D}^B = \{(X_i, c_{i,n} = t(X_i, n)_{n=1}^N)\}_{i=1}^K$ where X_i is the trace and $c_{i,n}$ is the template of the n^{th} signal in the trace. Each template is extracted using a simple procedure t that detects the edges of the corresponding signal trace. In both cases, the original source is a set of a few hundred thousand traces of driveline behavior recorded from 19 buses over a 3–5 year period. The construction of any training dataset reveals one important aspect of training, which is the level of supervision. The archetypal basic recipe introduced earlier, with the presence of labels y, clearly falls under the supervised training regime. Contrastingly, the absence of labels in $\mathcal{D}^A$ clearly places the unconditional generative model training in the unsupervised regime. The dataset $\mathcal{D}^B$ used for training the conditional model does have label information in the form of templates. But, with templates being rather trivial, yet not easily recoverable, transforms of the original data, it resembles the self-supervised learning regime which can involve the use of such pseudo-labels [19]. However, most reported self-supervised training approaches (for example [20]) include a classification task that is auxiliary to the main generation task. While we do not include a classification task, **Paper B** includes an auxiliary task involving the reconstruction of templates.

2.3.3 Network definition

The next element of the training process, the DNN g that actually accomplishes the mapping defined in (2.4). In setting up the generative process, appended papers use the following building blocks.

Encoder-decoder architecture – In the basic training approach, the DNN f maps an input x to a label vector y. The label is one alternative, albeit concise, form of representing the input, with the network f realizing this *encoding* process. Additionally, in mapping a high dimensional input into a lower dimensional label, f reduces the dimensionality of representation. In concert with the training objective, this forced minimization in dimensionality induces the model to discard unnecessary information and forward only information through its

layers that is relevant to the task. This principle – called the information bottleneck – is of profound importance to many deep learning approaches [21]. While the bottleneck appears naturally in a classification task, it may not occur in a generative task. For instance, the generative process g^B maps a lower dimensional template to a higher-dimensional trace. In such cases, it is common practice to *enforce* a bottleneck using an Encoder-decoder architecture. That is, the mapping process (2.4) is achieved using a two-stage process shown in (2.5), where f^E and f^G are encoder and decoder (or generator) neural networks.

$$z = f^E(c),\ \hat{X} = f^G(z) \tag{2.5}$$

The special case, where $c \coloneqq X$ leads, of course, to the autoencoder which is the primary structure used in **Paper A**. One critical benefit of the Encoder-decoder structure is the availability of a d-dimensional latent representation (or code) $z_i \in \mathcal{Z} \coloneqq \mathbb{R}^d$, which can be independently manipulated to influence generation. In learning to encode information under a bottleneck, f^E tends to place semantically similar inputs close together in $\mathcal{Z}$ [22]. **Paper A**, for instance, exploits this fact to achieve controlled conditional sampling by first mapping X_A to a code z_A, and sampling in some ϵ-neighborhood as shown below. This ensures that generated traces $\hat{X}_A^\epsilon$ are of controllable similarity to the condition X_A.

$$\begin{aligned} z_A^\epsilon &= \{z \in \mathcal{Z} : \|z_A - z\|_2 \leq \epsilon\},\ z_A = f^E(X_A) \\ \hat{X}_A^\epsilon &= \{f^G(z) : z \sim U[z_A^\epsilon]\} \end{aligned} \tag{2.6}$$

An issue with the structure defined in (2.5) is that one condition maps to one code and hence one trace. This is clearly a limitation for g^B where one template can map into many possible traces. Therefore, in **Paper B**, using an idea originally introduced in [23], such a limitation is overcome by adding one (or more) randomly sampled codes to the decoder, as shown in (2.7), which follows a given distribution and controls an additional mode of generative freedom. All networks in **Paper A** and **Paper B** use the Encoder-decoder architecture, but also extend them into other structures to meet different training objectives.

$$z^0 = f^E(c),\ \hat{X} = f^G(z^0, z^1),\ z^1 \sim \mathcal{N}(0, I),\ k > 1 \tag{2.7}$$

Convolutional neural networks – The basic DNN f, which uses an affine map followed by some non-linearity as its fundamental operation, is also called a fully-connected network. This is because in each layer, the weight matrix W_l is defined in such a way that every output feature depends upon every input feature. One problem with this configuration is that with high feature dimensionality, or with a complex learning task, the number of parameters in f can become prohibitively high. More importantly, many data modalities like images, text, and even signal traces exhibit spatial and temporal dependencies, which a fully-connected network clearly ignores. An alternative technique that exploits these dependencies, while bringing many other benefits, are learnable convolution filters. The essence of a 1D convolution operation using a learnable filter $W_l^{s \times N}$ of size s in the l^{th} layer of a DNN that processes an N-variate input sequence x of length T is captured below.

$$f^l(x) = \sigma(\sum_{n=1}^{N} W_l^{s,n} \cdot x_{t-\bar{s}:t+\bar{s}}^n + b_l),\ \bar{s} = \lfloor \frac{s}{2} \rfloor,\ t = 1...T \tag{2.8}$$

Exploiting spatio-temporal dependencies by using one filter to stride across the entire input sequence, the convolutional network significantly reduces the number of parameters in f. The convolution operation is extendable to both higher input feature dimensions and a higher number of filters. Unlike a fully-connected layer which compresses information globally, a convolutional layer with multiple filters naturally extracts a variety of local features. When used with a pooling operation, which is an information bottleneck, the resulting network is naturally encouraged to learn a hierarchical composition of concepts to represent the input. Introduced initially for processing images, convolutional layers have quickly become general DNN building blocks across data modalities, including time series (several examples in [24] and [25]). Both **Paper A** and **Paper B** primarily use convolutional layers in encoder and decoder networks to process traces and conditions. While specialist architectures for time-series like recurrent networks [26] and causal convolutional networks [27] exist, we find that vanilla 1D convolution is well-suited for generating instances of vehicle behavior. This is primarily because we task the model g to generate the entire trace at once, allowing us to treat the trace in an image-like fashion. Thus, instead of imposing a memory or causality model, within the remit of 1D convolution, g is free to model any relationship that suits the task.

2.3.4 Training objectives

The generative process defined in (2.5) is realized using a (mainly convolutional) Encoder-decoder structure, as shown below.

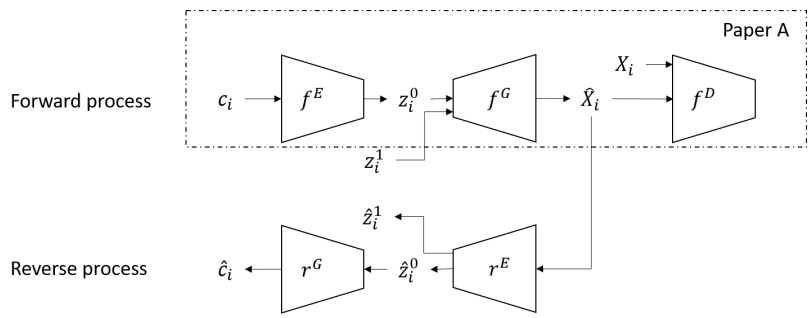

Figure 2.3: Model architecture for the generative process

Paper A uses only the forward generative process, while **Paper B** also adds a process to reverse it. The composite forward process is trained using the powerful adversarial learning framework, first introduced in [16]. The additional network f^D, which is tasked with evaluating the 'realness' of generated traces X_i, is called the discriminator. The classic adversarial training objective is a minimax game that simultaneously minimizes two competing objectives (2.9). This objective seeks an equilibrium where an optimal f^G produces traces that are realistic enough to fool f^D, and an optimal f^D that clearly distinguishes between real and generated traces. **Papers A** and **B** uses customized versions

of this objective to guide the learning process.

$$\mathcal{L}_{gan} = \min_{f^G} \max_{f^D} \quad \mathbb{E}_{X_i} log(f^D(X_i)) + \mathbb{E}_{z_i} log(1 - f^D(f^G(z_i))) \tag{2.9}$$

As previously discussed, latent space sampling plays an important role in controlled generation. Additional training objectives are therefore included to ensure that latent spaces that are jointly encoded and sampled follow a predictable distribution. Treating conditions as traces, in **Paper A**, the composition $f^E \circ f^G$ ostensibly resembles an autoencoder. But, since there is a need to selectively sample the latent space, it is trained as a variational autoencoder using the Evidence Lower Bound (ELBO) loss (2.10) introduced in [17]. Note that in **Paper A**, ELBO is used in addition to $\mathcal{L}_{gan}$ in the VAEGAN [28] configuration to improve the quality of generation.

$$\begin{aligned} \mathcal{L}_{rec} &= \min_{f^G} \ \mathbb{E}_{X,z^0 \sim \mathcal{N}(\mu,\sigma I)} \| f^G(z^0) - X \|_2 \\ \mathcal{L}_{KL} &= \min_{f^E} \ \ \mathbb{E}_c \ D_{KL}(\mathcal{N}(\mu,\sigma I), \mathcal{N}(0, I)) \\ \mathcal{L}_{vae} &= \mathcal{L}_{rec} + \mathcal{L}_{KL}, \ \mu, \sigma = f^E(c) \end{aligned} \tag{2.10}$$

The semi-supervised nature of the learning task with the presence of pseudo-labels allows **Paper B** to use an auxiliary cycle-translation [29] objective to help the mapping process. The entire generation process is reversed to recover the template $\hat{c}$ and the composite Encoder-decoder structure is optimized end-to-end using a reconstruction loss $\mathcal{L}_{ID}(c, f^E \circ f^G \circ r^E \circ r^G)$ (see (2.11)). For multimodal generation, in addition to code z^0 that is obtained by encoding a template c, a sampled code z^1 is necessary. The presence of the reverse channel allows the choice of a simpler reconstruction loss $\mathcal{L}_{ID}(z^1, f^G \circ r^E), z^1 \sim \mathcal{N}(0, I)$ to ensure that the encoded space $\hat{z}^1$ follows a predictable distribution, avoiding the complex ELBO objective. Further details about how these training objectives are customized and combined can be found in the appended papers.

$$\mathcal{L}_{ID}(x, f) = \min_{f} \quad \mathbb{E}_x \| f(x) - x \|_2 \tag{2.11}$$

2.3.5 Training process

Using the appropriate combination of training objectives, the forward process, and the complete process are respectively trained in **Papers A** and **B** in a joint, end-to-end fashion. The gradient descent process that seeks optimal parameters of the composite generative model follows the general principle described in (2.3), albeit with much more sophistication. Both **Papers A** and **B** use the Adam optimizer, details of which can be found in [30]. Training generative models, especially GANs, is widely acknowledged to be difficult. As pointed out in [31], GAN training needs to address three major concerns

Vanishing gradients – In training with the classic adversarial objective (2.9), one recognized problem is that when f^D is strong, it fails to provide useful feedback to f^G. This problem is usually mitigated by choosing a modified version of the adversarial objective. For instance, **Paper B** addresses this problem by

using the Least Squares GAN (LSGAN) [32] objective, which does two things - (1) f^D outputs a score instead of a probability, (2) the training of both f^G and f^D are defined as regression tasks instead of classification tasks. This provides a stronger feedback even when f^D is strong, stabilizing the training process.

Quality and diversity of generation – One important aspect to address during training is the evaluation of generated samples. Defining measures for evaluating samples is an active area of research, with several new techniques being routinely proposed. However, as pointed out in [18], many of them are specific to the image domain. The evaluation of generated time series therefore follows a domain independent two-sample test approach, which compares the statistics of real and generated samples. Using a similar approach **Papers A** and **B** introduce the evaluation principle of *Similarity as plausibility* (SAP). This principle actively exploits the presence of Encoder-decoder pairs (refer Figure 2.3) in the generation process. In **Paper A**, $f^E \circ f^G$ is a (variational) autoencoder. We therefore evaluate the quality of generation by measuring the reconstruction similarity on a held-out validation set. Different measures, some defined and some learned, are used for scoring the fitness of reconstruction. Even **Paper B**, where $f^E \circ f^G$ is not an autoencoder, we extend the SAP principle to measure translation fitness. This is possible because the training set $\mathcal{D}^B$ is aligned, with c_i being a template that is extracted from the trace X_i. This means that the similarity between the translated trace $\hat{X}_i$ and X_i can be used as a measure to evaluate generation. An issue, of course, is that with g^B allowing multimodal translation, these traces need not be similar. Prioritizing measurable evaluation, we impose an additional objective to encourage the model to produce translations that are close to the ground truth. In using this approach, we pay the price of reduced diversity in sample generation. The resulting translation is still multimodal, but less diverse.

2.3.6 Sampling

Having trained the generative model g, the final – and perhaps most important – step is to sample the model. Instances of realistic driveline behavior produced by sampling can then be applied as test stimulus for necessary vehicle functions. As noted earlier, while g may be trained unconditionally or conditionally, the sampling process is always conditioned, ensuring that stimulus generation is controlled and systematic. The measures introduced by **Papers A** and **B** to achieve this fall under two broad categories.

Latent space sampling – This technique is mainly used in **Paper A** which trains g unconditionally to realize the *trace* $\rightarrow$ *trace* mapping. The information bottleneck in the generation process exposes a latent space $\mathcal{Z}$, where similar traces are highly likely to cluster together. In this space, as shown in (2.6), it is possible to sample the neighborhood of a reference trace. **Paper A** introduces a *Metric-driven Linear interpolation* (MLERP) technique to make this sampling selective. With the help of a user-defined metric, MLERP helps select latent codes with guaranteed similarity with the reference trace, as measured by the chosen metric. One can note that this is simply an extension

of the SAP principle to sampling. This principle can be applied to sampling in other geometries in the latent space. Two reference traces X_A and X_B give a straight line between points z_A and z_B. As shown first in [22], traversing this line smoothly interpolates between the reference traces. MLERP refines this technique by selectively sampling only those codes on this straight line with guaranteed similarity with either or both reference traces. Using a combination of reference traces, either hand-crafted or recorded, and appropriately defined similarity metrics, the generative model becomes a powerful and credible, yet finely controllable, stimulus generator.

Latent space search – While selective sampling is an effective technique for controlled exploration of the stimulus space, it has a few limitations. The first is, of course, the requirement of reference traces, which **Paper B** simplifies by using templates. The second is that the continuous latent space, or any of its subspaces, provides infinite possible stimuli. If there is a need to produce targeted stimuli, sampling is a clearly inefficient way to achieve it. **Paper B** addresses selectivity further by introducing a scenario-based stimulus searching technique. Upon bounding a polygonal subset of the latent space using an arbitrary number of templates, the technique searches for appropriate stimuli using an objective extracted from the source code of the vehicle function under test. By ensuring that the search objective is differentiable, the technique uses a gradient-descent process to find a stimulus that directly satisfies a test condition. Avoiding extensive sampling of latent subspaces, this targeted technique provides yet another powerful mechanism for controlled stimulus generation.

2.3.7 Explanation

The deep learning process for training the generative model g, which began with the definition of the training task, thus goes through an elaborate sequence of steps. In most cases, the learning process ends with an evaluation of whether the model generalizes beyond the training set. For the basic DNN f, this is usually done by evaluating prediction performance on a held-out validation set. The evaluation of generative models, as described earlier, is much less straightforward. This work, for instance, achieves this using the SAP principle to reconstruct a held-out validation set. However, the learning process – a data-driven search through a parameter space of thousands, if not millions or even billions of parameters – is essentially a black box. Therefore, even if a trained model is observed to generalize well, the large number of parameters and the black-box nature of the learning process tends to impact the confidence in practically deploying and using the model. Recent years have seen an increasing use of techniques that *explain* what the model has learned [33] which, by providing an insight into how the model undertakes its task, is one way of unveiling the learning process. For predictive models like the basic DNN f, an explanation process is typically a function of the form $\mathbf{\Phi} : x, \hat{y} \rightarrow \phi$, where the explanation ϕ details why the model f chose to assign the label $\hat{y}$ to the input x. Assessing such explanations for a sufficient number of predictions is one way of increasing the confidence in using the model.

In the case of the stimulus generator g, explaining what the model has learned is much less straightforward. One main reason for this is that, to generate samples effectively, the core objective of the model is learning to the underlying data distribution. While the learning process, no doubt, plays an important role in sufficiently estimating this distribution, the role of the training dataset is arguably more critical. If the training data is a biased representation of the true data distribution then, even with a perfect training process, it is impossible for the generative model to effectively estimate the true distribution. With Objective 1 clearly stating the need for credibly approximating the input signal traffic for vehicle functions under test, it is especially important that the dataset used to train the model is not unreasonably biased. Going beyond the traditional format of explaining what the model has learned, **Paper C** addresses principles for evaluating and explaining the training data. The primary objective of **Paper C** is to evaluate whether a potential training dataset $\mathcal{D}$ captures a sufficient variety of samples. The evaluation process begins with a specification of *required* levels of diversity, which is embodied by a simulated dataset $\hat{\mathcal{D}}$. Techniques of explainability are then used to reveal any deficiency between the expected sample representation $\hat{\mathcal{D}}$ and the actual sample representation $\mathcal{D}$. Such a process of explaining sample representation allows the assessment of training data at an early stage and correcting its biases, before training the model. Such an informed process indirectly, but critically, addresses the need for credible generation defined in Objective 1.

2.4 Contributions – summary and synthesis

The previous section, which described our approach to simulating vehicle behavior, briefly introduced the research contributions of appended papers. Reiterating the focus of each paper in Figure 2.4, this section further details their contributions and derives additional observations.

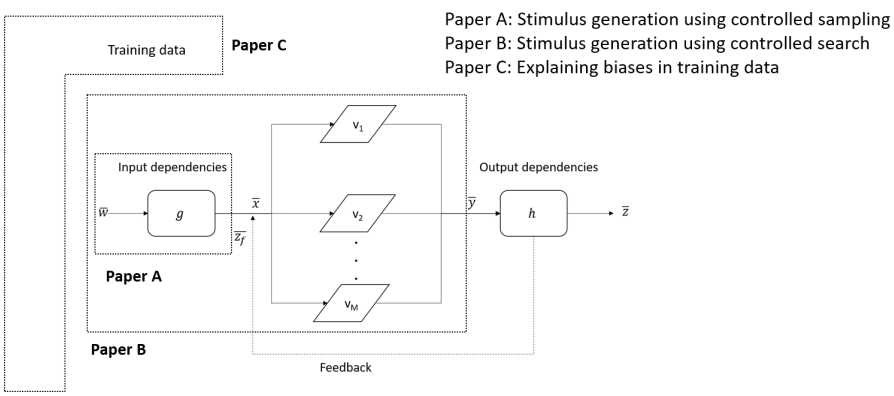

Figure 2.4: Focus of each appended paper

Paper A – The **first** contribution of this paper is the extension of deep learning, and specifically deep generative models, to the domain of simulation

and virtual integration for automotive software development. It does so by presenting a solution for learning vehicle behavior unsupervised from recorded operational data. Unlike most comparable literature involving GANs, the paper has a sharper focus on sampling than on training. This stems directly from of the need to achieve Objective 1 and provide credible test stimulus in virtually integrated rigs. To address credibility, the paper makes its **second** contribution by introducing the principle of *similarity as plausibility*, which uses reconstruction and translation fitness as a measure of the quality of sample generation. The principle is first applied to evaluate the GAN during training. As part of this evaluation exercise, the paper makes its **third** contribution in checking reconstruction fitness using both objective measures like the Structural Similarity Index (SSIM) and subjective measures like the VAEGAN similarity metric learned by f^D. A technique to calibrate learned metrics with objective metrics is also described so that the use of learned similarity metrics is better referenced. The paper then addresses Objective 2 by extending the SAP principle to GAN sampling. This is achieved using its **fourth** contribution – the MLERP algorithm for selectively sampling the latent space using any similarity metric of choice, objective or learned. In summary, this paper introduces a deep generative method for test stimulus generation that is both credible and controllable, addressing two of the most critical needs of virtual integration.

Paper B – While the previous paper introduces a credible and controllable technique for stimulus generation, it has a few limitations. Among them is the need to specify test conditions or scenarios using reference traces. The **first** contribution of **Paper B** is to simplify test scenario specification for vehicle functions by introducing *templates*. Its **second** contribution is to recast the generative process into one of translation, mapping templates to traces for subsequent use as test stimuli. The resulting model, trained with the assistance of SAP evaluation, is also much more versatile. Not only does it translate multimodally but also expands the timeline of the translated trace by generating forecast and backcast components. This is achieved by training the model with an additional random-cropping adversarial loss, the **third** contribution of the paper. The paper therefore vastly expands the toolkit for meeting Objective 1. Another limitation of the previous paper is its reliance on sampling, which can be very inefficient in the continuous latent space. This paper addresses it using its **fourth** contribution – a technique that extracts differentiable search conditions from code under test and uses it for a gradient-based search for test stimuli that satisfy the search condition. This process is controlled by the specification of templates that limit the search space. The result is a powerful method for targeted stimulus generation that directly meets Objective 2. Combining all these elements, this paper makes a **fifth** – and perhaps most significant – contribution. This is the connection of software in-the-loop with a GAN, for automatic test stimulus search (hence its name SilGAN), demonstrating the possibility of end-to-end scenario specification and test automation. While **Paper B** demonstrates a technique to automate a test objective like code coverage, the overall framework can be extended to other test objectives as long as testable properties are extractable in a form that aids latent space search. This reveals a path towards using stimulus generators for automating property-based software testing.

Paper C – Previous papers meet Objectives 1 and 2 in a direct manner. This paper, however, meets them in an indirect, but equally important, manner. In addition to the soundness of the training processes, the ability of the generative model to credibly represent behavior depends upon a critically important element – the training data. More specifically, the capability of a model to emulate behavior depends upon the diversity of samples represented in the training data. If there is a significant bias in the representation of samples, the model is likely to assimilate only a limited subset of the true behavior. This paper, therefore, addresses the less-explored area of *explaining* sample representation in a given dataset. The **first**, and overall, contribution of this paper is to explain sample representation by utilizing annotations, when available, and using parametric simulation otherwise. A primary ingredient of this explanation process is *parametric specification*, which partitions the data space in an interpretable form. The paper acknowledges that such specification may not be feasible for many practical high-dimensional datasets. This is one reason why it conducts experiments on images of simple geometric shapes, and not signal traces. The **second** contribution of the paper is the proposal of an *overlap index* as a measure of sample representation bias. Given distributions of expected and actual sample representation in terms of input space parameters, this index quantifies the discrepancy between them. When annotations are available, the quantification process is direct. For cases where they are not available, the paper uses an indirect method – its **third** contribution – to measure bias. The first step of this indirect process is to train a DNN on the non-annotated dataset whose sample representation is to be explained. The second step is to test this trained DNN with simulated data that follow the expected sample representation. The third step, the **fourth** contribution of the paper, is to estimate actual sample representation by assessing the trained DNN's familiarity with simulated samples. This estimated sample representation, when compared with the expected sample representation using the overlap index, exposes representation bias. Even if this paper does not work with signal traces, in providing a technique to explain training data, it addresses the plausibility perspective of Objective 1. Considering the practical limits of recording traces noted in Section 2.1, examining the data prior to training a generative model may be a helpful step. The important question is whether the technique tested on geometric shapes can be used to examine a dataset of signal traces. As noted earlier, an important ingredient of this process is able to simplify and parameterize the space of signal traces. One important step is the introduction (in **Paper B**) of templates as a technique for simplifying the space of traces. One interesting possibility can then be to use parameterized templates and a re-designed translation process to assess sample representation in a dataset of signal traces. Such parameterized templates can also simplify the specification of test conditions, additionally addressing Objective 2.

2.5 Conclusions

The automotive industry is already software driven, with the importance of software as the primary means of delivering customer value only set to grow.

Under such conditions, it is essential that vehicle manufacturers strengthen their ability to rapidly iterate through the software development process. While the use of simulation-driven development and testing has begun to help achieve this, the complex nature of a vehicle's operating environment mean that the state of practice of manually specifying simulation models only offers limited credibility. Exploiting the increased availability of operational data, this work shows that it is possible to use techniques of deep learning to train models that credibly imitate real operating conditions. Such a data-driven approach to simulation allows us to develop simulation models with high scope and fidelity, virtually eliminating the need for time-consuming manual specification. Techniques developed in this work adapt the deep generative modeling framework to train models that credibly generate open-loop test stimuli for software vehicle functions. In learning to approximate the real behavior of software input dependencies, test stimuli generated by these models are vastly more credible than manual specifications. Additionally, this work introduces several mechanisms to enable controlled-generation of stimuli, easing test design and increasing the confidence in the testing process. Considering the importance of credible generation, this work introduces several techniques at the data selection, model training, and sampling stages to improve the quality of stimulus generation. Overall, the tools and techniques developed in this work have immense potential to improve the credibility, and therefore confidence, in simulation-driven automotive software development, helping achieve the pressing need in the automotive industry for rapid delivery of quality software.

2.6 Future work

Simulating output dependencies – This work focuses only on simulating input dependencies. The immediate focus of future work would therefore be to expand the scope and additionally simulate output dependencies as trained DNNs. There are two major aspects in addressing output dependencies. The first is that the DNN simulator should react to outputs from upstream functions. For instance, a DNN that simulates the driveline must take inputs from acceleration, brake pedal sensor, or cruise control SWCs and realistically map it into the resultant distance and speed of the vehicle. The second aspect is that such outputs are often fed back, meaning that there is an additional need to maintain temporal consistency. In order to address these aspects, This track would combine several techniques including deep-learning based time series prediction or forecasting [25] and data-driven control system identification [34].

Machine learning based extraction of testable properties – While this work, specifically **Paper B**, automatically extracts test properties from code, it does so using simple hand-coded rules. A parallel track of future work would therefore focus on taking a machine learning approach to test property extraction. The first step, training a model that understands AUTOSAR SWCs, will be approached using neural language modeling techniques for source code [35]. Once such a model is trained, one potential downstream task would be the automatic extraction of test properties.

Generative model evaluation – This work addresses credible stimulus generation using several ways. The SAP principle and explaining sample representation in training data are prime examples. However, as noted earlier, evaluating the quality of a generative model is difficult. Therefore, one possible area of future work would be to expand processes of evaluation and explanation to further increase the level of confidence in stimulus generation. This work would expand ideas like [36] which propose evaluation techniques and measures for evaluating deep generative models.