School of Computer Science
FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

UNIVERSITY OF LEEDS

Final Report

Sequence-Based Machine Learning and Synthetic Data Generation for Real-Time Vehicular Traction Loss Detection

Louis Joseph Bishop

Submitted in accordance with the requirements for the degree of BSc Computer Science (Digital and Technology Solutions)

2025/26

COMP3932 Synoptic Project

The candidate confirms that the following have been submitted.

Items	Format	Recipient(s) and Date
Final Report	PDF file	Uploaded to Minerva (27/04/26)
Link to online code repository	URL	Found in Appendix B (27/04/26)
Video of live implementation	URL	Found in Appendix B (27/04/26)

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of Student) LOUIS JOSEPH BISHOP

Summary

Whilst modern vehicles generate vast quantities of continuous telemetric data, identifying complex, non-linear traction loss events has traditionally been a challenging task relying on rigid physical thresholds or computationally expensive physics-based models. Additionally, collecting real-world data on traction loss events is difficult, dangerous and expensive. To overcome these challenges, this project utilises the BeamNG.tech soft-body simulation environment to generate synthetic telemetry across various traction loss scenarios. This data is used to train and evaluate a range of machine learning models including GRUs (Gated Reccurent Units), CNNs (Convolutional Neural Networks), LSTMs (Long Short-Term Models), Transformers and Random Forests in their capability to identify traction loss events in continuous, temporal data.

Evaluation of the final models after training, tuning and adjustments reveals that, even after omitting dominating driver control features, the models were able to understand the underlying kinematic patterns, with the GRU model achieving an accuracy of 95.44% and a macro $F1$-score of 0.904. Implementing a two-stage classification pipeline further bolstered performance and reduced the key metric of false negatives. This model was then implemented in a real-time inference system connected to the simulator, demonstrating the potential for practical deployment and timely traction loss identification.

Ultimately, this research demonstrates the viability of using high-fidelity synthetic data to train robust neural networks and proposes a shift in the focus of vehicle dynamics analysis away from reactive mathematical equations toward adaptable machine learning.

Acknowledgements

The author would like to thank the following individuals for their support and contributions to this research:

- Prof. Leandro Soares Indrusiak for his support and guidance throughout the planning and execution of this dissertation. Without his expertise, guidance in developing the concept of the project and his feedback on the report it would not be what it is today.
- Jack Jacobsen, Alex Northam, Luca McAtarsney, Eschal Najmi, George Hester, Rory Edmonds and the rest of my coursemates for their support during the writing process.
- The Leeds Gryphon Racing team for their support and providing practical context for the research, as well as engine mapping data that was used to inform the simulation parameters.
- My family and my partner Izzy Almond for their constant encouragement and belief in my abilities. Thank you for bearing with me while I was working away on this.
- The BeamNG.tech team for providing an academic license and technical support for the BeamNG simulator, without which this project would not have been possible.
- LucasBE for creating the Carbonworks F4 mod, which was used in the simulation environment to generate the synthetic telemetry data for this research.

Contents

Chapter 1

Introduction and Background Research

1.1 Introduction

Modern vehicles are equipped with a wide range of sensors that generate a significant and continuous stream of telemetric data for safety and performance monitoring [1, 2]. This high-resolution data collection is not limited to high-end sports cars or racing machines; even entry-level or vaguely underbaked student-built Formula Student [3] cars are extensively instrumented with electronic measurement systems that log high-frequency data to an Engine Control Module (ECM) or dedicated data logger [4].

Whilst this data is abundant, explicitly identifying complex non-linear dynamic events - such as a sudden loss of traction - requires sophisticated analytical techniques. Therefore, this dissertation proposes a machine learning-based approach to detect traction loss events directly from live telemetry [5].

To overcome the risks associated with collecting labelled, real-world skidding data, this project first employs automated synthetic data generation using the BeamNG.tech simulation environment [6] [7]. The decision to utilise synthetic telemetry was taken to simplify three key challenges: safety and cost of edge-case collection [8], data scarcity and ground-truth precision. The resulting continuous telemetry streams then undergo rigorous data preparation, utilising sliding window techniques to format the data for temporal processing and capture the sequential dependencies inherent in vehicle dynamics [9, 10].

Using these temporally processed datasets, a diverse range of machine learning architectures are trained and evaluated. The scope of this evaluation includes sequence-based Deep Learning models such as Long Short-Term Memory (LSTM) networks [11], Gated Recurrent Units (GRU) [12] and Transformer models [13], alongside Convolutional Neural Networks (CNN), Random Forests and specialised two-stage classifiers. All models are hyperparameter-tuned using Keras Tuner [14], systematically exploring learning rates, layer sizes, dropout rates and optimization strategies to maximize performance across validation metrics.

Following comprehensive validation on dedicated train and test sets, the final phase of this research bridges the gap between theoretical modelling and practical application. The highest-performing models are implemented to make live, real-time inferences on incoming simulator telemetry, demonstrating the viability of this approach for active driver assistance and data inference.

This report will detail the background research conducted whilst developing this project, the methodologies employed to generate, process and train on the telemetry data and the conclusions drawn from the results of the various models. The implications of this work for future research and real-world applications in vehicle dynamics will also be discussed.

1.2 Background Research

1.2.1 Vehicle Dynamics and Traction Loss

In the context of motoring - especially motorsports and high-performance driving - traction loss events encompass a range of dynamic occurrences where the relationship between the tyres and the road surface breaks down. This can manifest as understeer, oversteer or a complete loss of grip, leading to skidding or spinning [10]. Understanding the signature causes and characteristics of these events is crucial for developing effective detection algorithms.

Fundamentals of Vehicle Stability

It is initially important to introduce the concept of Tyre Slip Ratio. This defines the relationship between the rotational speed of the tyre and the ground speed of the vehicle. Demonstrably, a slip ratio of zero indicates perfect traction, whilst a positive slip ratio indicates that the tyre is spinning faster than the vehicle is moving (a common scenario in acceleration-induced traction loss) and a negative slip ratio indicates that the tyre is rotating slower than the vehicle's speed (often observed in braking-induced traction loss) [10]. While trivial to calculate within a simulation environment using absolute state data, deriving true slip ratio in real-world applications requires expensive, specialised ground-speed sensors [15]. Consequently, rather than utilising slip ratio as an input feature for the predictive models, this project leverages this exact mathematical definition strictly to establish definitive ground-truth labels. By calculating absolute slip offline, precise targets for traction loss events are generated, providing supervised training data for the models without introducing real-world data leakage.

In the context of motorsport, we seek to optimise vehicle performance by operating at the limits of traction. Whilst this is desirable for performance, it also increases the risk of traction loss events and makes their detection and accurate identification crucial for driver feedback and post-session analysis. Operating on the limit also means that it can be difficult to distinguish between normal driving and a traction loss event, as the telemetry signatures may not be as simple as a clear spike in a certain ratio - as such, this is where machine learning can prove particularly useful.

Types of Traction Loss

A tyre can only produce a certain amount of grip before losing traction. Many factors influence this limit (an approximation is given by the Pacejka Magic Formula in Figure 1.1 [16]), but it is generally accepted that the tyre must be able to withstand the combined forces of braking, acceleration and cornering without exceeding its grip threshold [10]. When this threshold is exceeded, the tyre enters a state of slip, which can lead to understeer (where the front tyres lose grip first), oversteer (where the rear tyres lose grip first) or a complete loss of traction (a lockup) where all tyres lose grip simultaneously [10]. These events can be triggered by various factors, including excessive speed, abrupt steering inputs, unwise braking or a combination of the three. Another common traction loss event occurs when a car launches; in the absence of mechanisms like Launch Control [17], it is possible for the driver to attempt to move off with a

wheel-speed that demands such grip that the tyres cannot keep up. The wheels spin for a duration before any traction is achieved. The ability to detect and classify these events in real-time is the foundation of this project, with the goal of providing quick, actionable feedback.

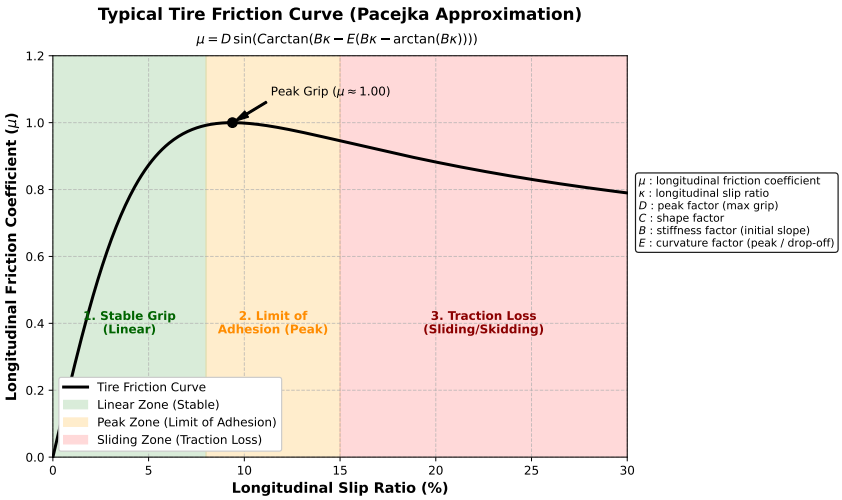

Figure 1.1: A theoretical tyre friction curve based on the Pacejka approximation, illustrating the non-linear transition from stable grip to active traction loss [16].

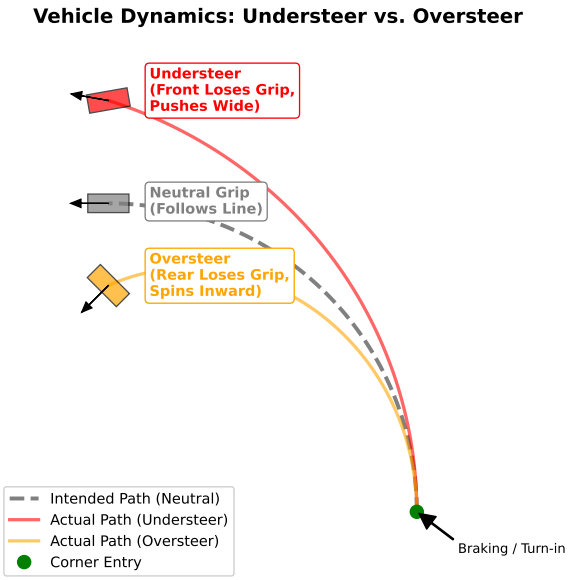

Figure 1.2: A visual representation of understeer and oversteer dynamics, showing the typical trajectories and slip angles associated with each type of traction loss [10].

Understeer and Oversteer

Understeer typically occurs when entering a corner with too much speed. The front tyres attempt to grip the road and rotate the car, but if the speed is too high, they exceed their grip limit and slide outwards, causing the car to take a wider line than intended or simply not turn at all. Oversteer, on the other hand, happens when the rear tres lose grip first. This can occur if the throttle is released mid-corner, or when accelerating out of a turn. The rear of the car

swings outwards, and if not corrected, it can lead to a spin [10]. A visualisation of these dynamics is given in Figure 1.2.

1.2.2 Vehicle Telemetry and Sensor Data

Modern vehicles in this era of data-driven performance are capable of generating a vast array of continuous telemetry data through their embedded sensors. This opens up a rich source of information that can be harnessed for our machine learning models to detect traction loss events. The key to leveraging this data effectively lies in understanding the types of sensors commonly found in vehicles and the nature of the data they produce.

CAN Bus and Onboard Sensors

The Controller Area Network (CAN) Bus is a standard protocol used in vehicles to allow embedded systems and sensors to communicate with each other [2]. Typically connected directly to the vehicle's ECM or a dedicated data logger, the CAN Bus provides a unified, continuous stream of data from various sensors. Commonly available data relevant to traction loss detection include wheel speeds, Inertial Measurement Unit (IMU) data, throttle position, brake position, steering angle and engine RPM [1]. The IMU is particularly important for our purposes as it provides critical information about the vehicle's acceleration and orientation. The wheel speed sensors can also provide direct insights into slip, whilst throttle and brake positions can help contextualise the driving inputs leading up to or during a potential traction loss event.

1.2.3 Feature Engineering for Vehicle Data

Effective machine learning on vehicle telemetry requires feature engineering to expose the underlying patterns of target events that may be latent or more obscure in the raw data [1]. Three categories of these features can be defined as such:

- **Temporal derivative features** capture the rate of change of key variables like Engine RPM, vehicle speed and wheel slip. Traction loss events often involve rapid changes in data; first order differentiation of these signals has been shown to improve detection of transient events that might be missed in the steady-state values [18].
- **Spatial aggregations** reduce sensor noise and reveal more vehicle-level dynamics. Averaging slip values by axle enables comparison between front and rear traction states, whilst computing variance across all four wheels provides a measure of slip asymmetry, which is a common signature of understeer or oversteer [19].
- **Physics-based features** encode domain knowledge from vehicle dynamics theory, including wheel speed differentials and statistical variances. Features such as the rate of individual wheel speed change, front-to-rear axle speed deltas, left-to-right averages, and overall wheel speed variance act as highly reactive, observable proxies for traction loss. These relative indicators provide interpretable, standard-sensor-derived signatures of instability.

1.2.4 Synthetic Data Generation and Simulation

An emergent challenge in developing advanced machine learning models for contexts like vehicle dynamics is the rarity, barrier to collection and safety risks associated with the data required to train these models effectively.

Synthetic data generation through simulation aims to address these challenges by providing a controlled, repeatable and often virtual environment for the conduction of data collection. Safety risks associated with real-world data collection can be mitigated, as the simulation allows for the creation of edge-case scenarios that would be dangerous, impractical or expensive to replicate in real life [8]. A virtual environment also allows for the scenarios to be repeated with either identical or controlled variance in the conditions, thus also enabling deliberate oversampling of rarer events to address potential class imbalance [9].

Beyond vehicle dynamics, synthetic data generation is widely utilised in fields like robotics and healthcare to safely simulate edge cases and bypass privacy constraints. Simulation is therefore best treated as a controlled and accessible data generation environment that can be used to create large labelled datasets during model development, with the goal of real-world validation as a logical next step [6, 7].

1.2.5 Deep Learning for Time Series Classification

Time series classification regards the task of assigning a label to a sequence of data points by observing the patterns that exist within. This differs from traditional classification tasks that operate on static data, as the models must be sensitive to not simply immediate stimulus but also the temporal context in which it occurs [9].

Recurrent Neural Networks (RNNs)

One such architecture that has been widely adopted for time series classification is the Recurrent Neural Network [20]. They are designed to pass data sequentially through a network of interconnected layers, where the output of one layer at a given time step is fed back into the network as input for the next. This allows them to maintain a 'memory' of previous inputs and resulting state. Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRU) are variants of RNNs that are capable of capturing longer-term dependencies in the data [11, 12].

CNNs and Transformers

Whilst RNNs are often the default choice for time series data, Convolutional Neural Networks (CNNs) have also demonstrated capability to learn local temporal patterns by applying filters across neighbouring timesteps. Typically, CNNs are used to process image data by stacking layers of processing to extract and downsample patterns like edges and objects, but they are also often very efficient and competitive in identifying short-duration events [9]. Transformer models, on the other hand, evaluate the importance of different parts of the input sequence relative to one another. This process, called a "self attention mechanism", allows it to understand long-range context; often more effectively than recurrent architectures [13].

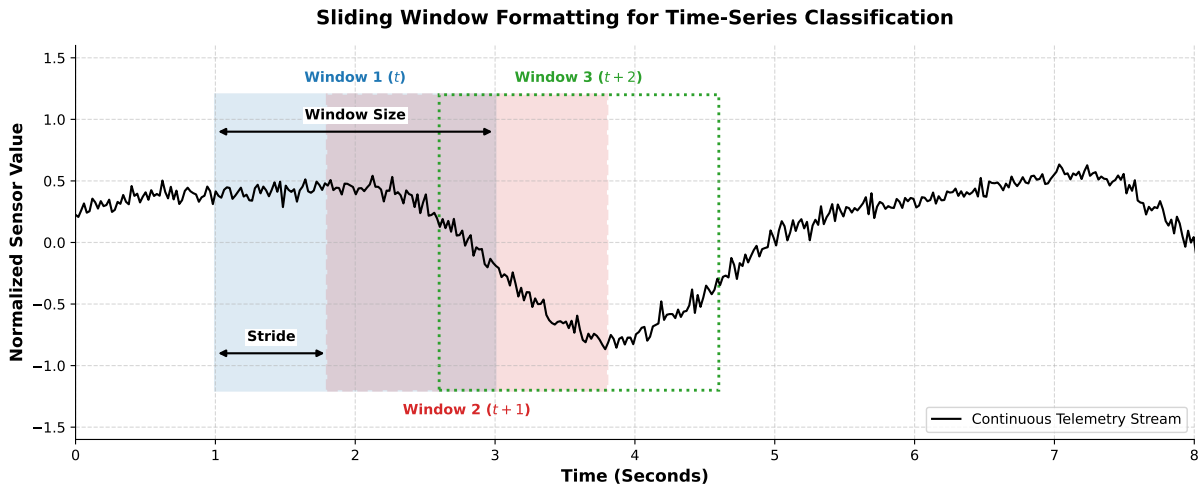

Figure 1.3: A visual representation of the sliding window technique for time series data, showing how a fixed-size window moves across the data sequence to create multiple training samples [9].

Sliding Window Approaches for Temporal Data

When dealing with continuous data or streams, a practical challenge arises in how to format the data for these models. Most supervised learning algorithms require an input of fixed size, but time-series data can be of arbitrary length. A common and effective solution to this issue is to implement a sliding window structure [9].

Constructing the sliding window involves defining a fixed-size window that 'moves' through the data sequence, taking a number of 'strides' each time (as seen in Figure 1.3). Each window captures a segment of the data, which can then be treated as an individual sample for training the model, since the model can observe the change in label within the window and how it relates to the surrounding points in time. Whilst this results in a larger number of larger training samples as data must be duplicated across windows, it allows the model to learn from the temporal context and dependencies that are crucial for accurate classification of events.

1.3 Literature Review

1.3.1 Vehicle Dynamics and Stability Control

The foundational mechanics of vehicle stability are well-established in works such as "Vehicle Dynamics and Control" by Rajamani [10]. It provides the core kinematic definitions of longitudinal and lateral slip that act as the theoretical baseline for traction loss. This project utilises Rajamani's theoretical frameworks of slip ratio exclusively to mathematically define and label the target events during the synthetic data generation phase. Furthermore, Pacejka's "Tire and Vehicle Dynamics" [16] demonstrates that the physical limit of tyre grip is highly non-linear and influenced by a myriad of factors, rendering explicit physical prediction models computationally expensive and fragile at the limit. This non-linearity directly motivates the use of machine learning; rather than mathematically calculating slip through explicit equations, the models are trained to capture these complex physical relationships by inferring the underlying traction state directly from raw telemetry and wheel speed differentials.

In consumer vehicles, the practical application of these dynamics has relied on threshold-based control systems. Van Zanten [21] details the implementation of Bosch's Electronic Stability Program which establishes the standard of using yaw rate and lateral acceleration to trigger corrective braking. Whilst this proves the efficacy of these features, it highlights the limitation and rigidly reactive nature of static thresholds in dynamic conditions. By applying sliding-window temporal classification, this project seeks to identify the subtle telemetry signatures of traction loss as an evolving sequence, rather than waiting for a static threshold to be crossed.

This reliance on static thresholds expands to motorsport and high-performance applications too. Honda [17] outlines the development of their Formula One traction control systems that modulate engine power delivery based on explicitly defined wheel-slip limits. Whilst this has proven effective for optimising pure longitudinal acceleration, the approach in the report requires extensive manual calibration for specific conditions and does not account for the complex scenarios where slip may occur due to transient dynamics in corners. The machine learning approach proposed in this project directly addresses the shortcomings of Van Zanten and Honda's work by learning from the data itself, rather than relying on pre-defined thresholds. By training sequence-based models on diverse synthetic telemetry, this project aims to create a more adaptable system that infers traction loss from multi-dimensional data patterns; removing the need for manually calibrated and one-dimensional thresholds.

1.3.2 Telemetry Extraction Techniques

The foundation of modern vehicle dynamic analysis lies in the continuous extraction of telemetry data. Taylor et al. [1] outline a standard methodology for mining vehicle telemetry and revealing the wealth of information available. Wheel speeds, throttle positions and IMU data are available through standard CAN bus interfaces and can be accessed through the ECM. Li et al. [2] further validate the reliability of these techniques. This project directly utilises ubiquitous CAN-standard signals identified in these works as the primary feature inputs for the machine learning models. Matsuzaki [22] advocates for the use of wireless specialised sensors directly within the tyre to achieve higher accuracy friction monitoring; this approach is heavily hardware dependent and impractical for widespread implementation. This project relies exclusively on standard, noisy CAN telemetry, actively shifting the burden of accuracy from specialised, expensive hardware to computational interpretation.

Since standard CAN bus data does not provide an absolute measurement of true ground speed or tyre slip, the industry currently relies on mathematical estimators to interpret the raw signals. Klomp et al. [18] propose a robust longitudinal velocity estimator specifically designed to detect excessive wheel slip in hybrid electric vehicles by analysing the discrepancy between driven and non-driven wheels. Zhang et al. [19] detail a novel four-wheel slip estimator tailored for low-friction, snowy conditions, utilising advanced filtering techniques to isolate slip events. While mathematically rigorous, both of these approaches attempt to continuously calculate slip by filtering raw sensor data through predefined physical vehicle models. This project shares the foundational goal of identifying slip from indirect sensors, but diverges significantly in execution. Mathematical estimators like those proposed by Klomp et al. and Zhang et al.

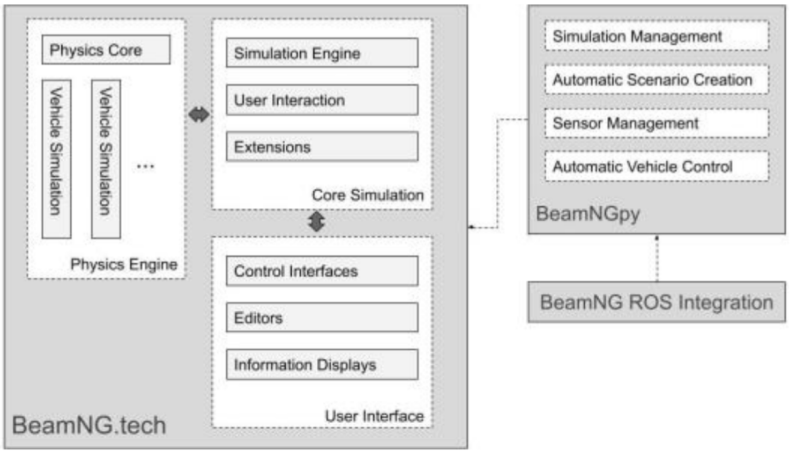

Figure 1.4: BeamNG.tech's architecture, from [7].

require precise, pre-defined vehicle parameters (such as mass and exact tyre radius) and and are vulnerable to noisy data during sudden, transient events. This project aimed to bypass these explicit physical equations by training models to recognise temporal data signatures of traction loss without requiring perfect physical parameterisation. Establishing a direct quantitative comparison with full physics-based slip-estimation pipelines was not within the scope of this dissertation and is addressed in discussion of future work.

Agrawal et al. [4] outline the challenges of real-time wheel slip detection for electric race cars, highlighting the necessity for high-frequency data logging to capture traction loss events before they destabilize the vehicle. Correspondingly, Croft-White's [23] research on rally car dynamics demonstrates the extreme difficulty of accurately measuring and analysing slip at high attitude angles, where traditional kinematic assumptions and standard sensor interpretations break down entirely. These studies underscore a critical operational gap: when a vehicle operates at the absolute limit of adhesion the rigid physical models used for standard slip estimation become unreliable. This dissertation directly addresses this gap by shifting the paradigm away from attempting to mathematically model these extreme non-linear edge cases, instead turning to machine learning to map raw telemetry patterns directly to specific traction loss events.

1.3.3 Simulation as a Validated Data Generation Tool

In automotive research, such as work by Rock et al. [15], the measurement gap experienced at the limit of adhesion is typically bridged using dedicated, high-cost optical ground-speed sensors to establish true, slip-independent planar velocity for validation. Translating this industry-standard validation method into a simulated environment is achieved by leveraging the simulator engine's absolute state data, acting as a perfect analogue to a physical optical sensor. This provides an uncompromised ground truth for vehicle speed. Instead of attempting to mathematically model these extreme non-linear edge cases, this research utilises this synthetic ground truth to safely supervise the training of neural networks, mapping the raw CAN telemetry patterns directly to specific traction loss events.

The validity and usefulness of this synthetic approach hinges entirely on the fidelity of the chosen simulation environment. Traditional racing simulators often rely on simplified, rigid-body kinematic models that use pre-calculated lookup tables to approximate grip, which is insufficient for our purposes. Maul et al. [7] outline the architecture of BeamNG.tech, including the architecture diagram in Figure 1.4, detailing its unique soft-body physics engine that simulates the deformation of the vehicle's structure and tyres in real-time. This technical capability, officially documented by BeamNG [6], ensures that the simulated vehicle behaves organically under load rather than following hard-coded artificial trajectories. By utilising BeamNG.tech, this project ensures that the resulting synthetic CAN bus telemetry possesses the required chaotic, non-linear characteristics of real-world physical limits and provides enough detail and insights to enable a machine learning approach.

This specific application of simulators for analysing traction loss and slip angles is supported by the work of Bazilinskyy [24], who explores the effects of auditory feedback on a driver's ability to maintain an ideal slip angle within a racing simulator. Whilst Bazilinskyy's primary focus is on human-machine interaction and sensory feedback, the foundational premise of his research relies on the simulator's ability to accurately output realistic, continuous slip angles relative to an ideal physical trajectory. This project builds upon that foundational premise but, instead of using the simulator's slip calculations to trigger auditory feedback, this project uses those exact ground-truth slip calculations to automatically label the synthetic telemetry streams. This effectively creates an annotated dataset for supervised machine learning, aligning with proven approaches in other domains where real-world data collection is similarly prohibitive [25, 26].

ETH Zurich and Intel [27] demonstrate a method for training a neural network policy in simulation, later transferring it to an advanced robotic system. Furthermore, OpenAI's seminal paper on Learning Dextrous In-Hand Communication [28] relies on a simulated environment with randomised physical properties, a key aspect of this project's data generation pipeline.

1.3.4 Machine Learning for Time Series Classification in Vehicles

The applications of machine learning to vehicle dynamics represents a significant departure from traditional control systems. Freudling [5] explicitly demonstrates the potential of this approach, successfully utilising machine learning to detect oversteering events in BMW automobiles directly from sensor data. This dissertation adopts Freudling's core idea that complex dynamic events can be computationally inferred rather than strictly calculated, but expands upon it by recognizing that vehicle telemetry is fundamentally sequential. Early or simpler machine learning implementations often evaluate data points in isolation. However, as Fawaz et al. and Wang et al. [9, 20] discuss, deep learning architectures specifically designed for Time Series Classification are vastly superior at extracting patterns from continuous, sequential data streams where the context of preceding events is crucial. As such, this project formats the incoming telemetry using sliding-window techniques outlined previously to ensure the models learn the temporal progression of a skidding event, rather than just reacting to a single, static snapshot of data.

To effectively model these temporal dependencies it is necessary to evaluate the most robust

sequential architectures available in the literature. This project evaluates Hochreiter and Schmidhuber's [11] LSTM networks and Cho et al.'s [12] GRUs as the foundational standards for tracking the evolving state of vehicle grip. However, whilst recurrent networks are powerful, they process data sequentially. This can occasionally limit their ability to weigh the importance of disparate data points across a long time window. To address this, we also integrate the Transformer architecture introduced by Vaswani et al., [13]. By contrasting standard RNNs against attention-based models, this project seeks to identify the optimal architecture for real-time telemetry interpretation.

It is common that data classification tasks can be naturally decomposed into multiple stages. A two-stage approach first performs coarse event detection and then refines that decision into a specific event category. This can be useful when the class structure is imbalanced or when distinguishing between normal operation and event-specific behaviour requires different levels of temporal sensitivity [29]. In this project, the same idea is applied by first detecting whether a temporal window contains a traction loss event and then classifying the particular event type.

Training these complex models on vehicle telemetry introduces practical data science challenges - primarily the issue of extreme class imbalance. In any realistic dataset, normal driving behaviour vastly outnumbers the rare instances of traction loss. If left unaddressed, models will simply learn to predict "normal driving" continuously to achieve high baseline accuracy. To force the models to learn the minority skidding events, this project draws upon established techniques such as the Synthetic Minority Over-sampling Technique proposed by Chawla et al. [30] to balance the training data, alongside the implementation of Focal Loss by Lin et al. [31], which dynamically scales the loss function to heavily penalize the model when it misclassifies edge cases. Additionally, to ensure the selected architectures are operating at peak efficiency, the Keras Tuner framework outlined by O'Malley et al. [14] is utilised to systematically optimise the network hyperparameters.

1.3.5 Research Gap and Summary of Aims

Whilst the physical foundation of vehicle stability [10, 16], the extraction of CAN bus telemetry [1, 2] and the idea of deep learning for time series data [9, 20] are well-documented in isolation, a distinct gap remains in their unified application. Current machine learning implementations in the automotive domain often focus on either self-driving applications [8] or predicting component failure [32], rather than an analysis of the vehicle's dynamic state. Contemporary dynamic control systems rely too heavily on reactive, hard-coded physical thresholds [21, 17]. By combining high-fidelity synthetic data generation with advanced, sliding-window temporal classification models, this research presents a framework methodology to identify and classify traction loss events in real-time, evaluating multiple models' performance. This approach actively shifts the paradigm of vehicle dynamics analysis away from rigid physical equations [18, 19] and toward adaptable, predictive computational pattern recognition. To achieve this, and to demonstrate that this kind of research can be made accessible, this project will also investigate the suitability of simulator-derived data for training machine learning models, and the practical challenges of tuning and implementing these models for real-time inference on live telemetry.

Chapter 2

Methodology

2.1 Decision to use Synthetic Data

The first aspect of the project to be developed was functionality to generate and label a training dataset synthetically using a simulator rather than attempting to collect or repurpose existing real-world data. As previously established, traction loss events are relatively rare and often unsafe to reproduce deliberately which creates an inherent challenge for collecting a dataset that is both large enough and accurately annotated enough to train and evaluate sequence models reliably. Simulation addresses this problem by allowing the same driving scenario to be repeated under controlled conditions while preserving access to the simulator's absolute state data. This makes it possible to derive ground-truth labels for traction loss events without introducing the uncertainty that comes with indirect estimation from noisy sensors. In practice, this means the dataset can be constructed around known event boundaries, consistent scenario conditions and repeatable vehicle behaviour, all of which are important for a temporal classification pipeline.

This decision does introduce a dependency on simulator fidelity; the synthetic telemetry must be realistic enough to support meaningful model training, but not so simplified that the resulting patterns fail to transfer to real-time inference. As such, the selection and configuration of the simulator and scenarios as well as feature selection based on the available telemetry were important considerations that will be discussed in this chapter.

2.2 Simulator Selection and Configuration

Previously detailed initial research led to the selection of BeamNG.tech [6] as the ideal environment for this project. After installation, registration and some preliminary experimentation with the Python API BeamNGpy [33] to confirm the telemetry extraction and scenario execution capabilities, the next step was to configure the simulator for dataset generation. This involved selecting a vehicle model, designing scenarios that would produce traction loss events and determining which telemetry features to extract for model training.

2.2.1 Vehicle Model Selection and Modification

The inspiration for this project can be traced originally to experience developing telemetry systems for Formula Student-style open wheel racing cars [3]. As such, vehicle selection was focused on finding a model that would be representative of a typical vehicle in that category. Nothing in the default BeamNG.tech vehicle library was an exact match for what was needed, but further searching among the vast array of user-created models let to the selection of the "Carbonworks F4" [34] by user LucasBE. Aiming to replicate a Formula 4 car (an entry level open-wheel racing category), this model would be ideal for the project as it allowed for

extensive configuration and featured an adequate physics model that would allow for realistic vehicle dynamics.

To align the model more closely with the Formula Student capabilities, a few manual modifications were made to the selected configuration. This involved modifying the underlying `.jbeam` files to adjust the engine power curve and characteristics, gear ratios and weight distribution to better reflect the performance characteristics of a typical Formula Student car. These modifications were informed by the Formula Student rules as well as engine mapping data from the Leeds Gryphon Racing [35] team's 2025 entry. Additionally, some aerodynamic components were removed to simplify the model, reduce downforce and encourage more frequent traction loss events.

2.3 Detection Target Selection and Scenario Design

The next step was to design scenarios that would reliably produce examples of traction loss events in a manner that was repeatable and consistent but also susceptible to variation in the telemetry patterns to avoid overfitting. The scenarios would all be executed on the baseline `smallgrid` environment provided by the simulator so as not to introduce additional complexity or overfit to specific track features.

To demonstrate the capability of the models, events were chosen that would be relatively common in a racing context and would have clear definition in the underlying physics of the vehicle. The first scenario designed was `standstill`, simulating a situation where a stationary vehicle loses traction on a launch and the wheels spin without the car moving. This was chosen as a simple, well-defined event that would be easy to label and lay the groundwork for more complex scenarios. Once this was established, the next scenario was `lockup` - an instance where a vehicle in motion experiences a sudden loss of traction on two or more wheels due to excessive braking force. As a common occurrence in racing, typically at the end of a long straight, this scenario would be more complex and require the models to recognise a more dynamic event with a less clear boundary. Finally, scenarios were implemented to capture examples of oversteer and understeer. To account for oversteer, `powerover` which involved excessive throttle input during a corner and `liftoff` which alternatively utilised excessive lift-off of the throttle during cornering were implemented. For `understeer`, a scenario was derived where the front wheels lost traction under steering due to excessive speed entering a corner. These scenarios would be the most complex, requiring the models to learn more subtle patterns in the telemetry data to identify the event boundaries.

These scenarios were all designed to fit a framework of automatic execution, variation and labelled data generation. This involved scripting the simulator to execute the scenarios with varying parameters such as steering angle, throttle input and braking force to create a diverse dataset. Their structure also allowed for simple modification and iteration to explore different event definitions and boundaries as needed during model development. The final thing to note is that the scenarios were designed to be executed in a loop, allowing for the generation of large amounts of data in a reasonable timeframe while maintaining consistency across runs.

2.4 Feature Selection and Engineering

Before the scenarios could be executed, it was necessary to determine the telemetry features that would be extracted from the simulator for both model training and real-time inference. The selection of features was informed by previous research regarding the capabilities of real-world sensors and the underlying physics of traction loss events. Notably, these features could be specifically included or excluded from training, so the initial selection extracted and recorded was intentionally broad to allow for flexibility in model development and experimentation. Additionally, some underlying environment data was extracted from the simulator for the ground-truth labelling of the events, but these would not be included in the training data as they would not be accessible in real-life applications and would be considered data leakage.

The initial selection of data returned from the simulator included engine RPM, wheel speeds, accelerometer values, angular velocity and driver inputs (brake, accelerator and steering angle). These features were chosen for their accessibility in actual racing vehicles and their utility in capturing the dynamics of the target events, as well as their capability to be engineered into more complex features.

Alongside these, the simulator's absolute state property of Ground Speed was extracted for the purpose of calculating "slip" values which would be used for labelling. A "slip" value was calculated for each wheel as the difference between the wheel speed and the ground speed, providing a direct measure of traction loss. This was particularly important for the more complex scenarios where the event boundaries were less clear as it provided a consistent and objective basis for labelling the data.

2.4.1 Engineered Features

Following the feature-engineering strategy introduced in Chapter 1, derived signals were computed from raw telemetry at each timestep t, separately within each run.

Let wheel speeds in m/s be $v_{FL}, v_{FR}, v_{RL}, v_{RR}$.

Derivative features First differences (sample-to-sample change) were computed as

$$\Delta x_t = x_t - x_{t-1},$$

with the first value in each run set to zero. This was applied to:

- engine speed: `rpm_rate` $= \Delta($`rpm`$)$
- steering input: `steering_rate` $= \Delta(\delta)$
- wheel speeds: `wheel_speed_rate_FL/FR/RL/RR` $= \Delta(v_{FL/FR/RL/RR})$

Spatial aggregation features Wheel-speed channels were aggregated to expose axle-level behaviour and lateral imbalance:

$$\texttt{wheel\_speed\_front\_avg} = \frac{v_{FL} + v_{FR}}{2}, \qquad \texttt{wheel\_speed\_rear\_avg} = \frac{v_{RL} + v_{RR}}{2}$$

$$\texttt{wheel\_speed\_left\_right\_diff} = (v_{FL} + v_{RL}) - (v_{FR} + v_{RR})$$

$$\texttt{wheel\_speed\_variance} = \mathrm{Var}(v_{FL}, v_{FR}, v_{RL}, v_{RR})$$

Vehicle-speed estimate and coupling feature An estimated vehicle speed was computed from front-wheel (typically not driven) speeds [18]:

$$\texttt{vehicle\_speed\_est\_ms} = \max\left(0, \frac{v_{FL} + v_{FR}}{2}\right).$$

This estimate was then smoothed using a 3-sample rolling median within each run.
A normalised axle-speed difference was defined as:
Let

$$v_f = \texttt{wheel\_speed\_front\_avg}, \quad v_r = \texttt{wheel\_speed\_rear\_avg}, \quad \hat{v} = \texttt{vehicle\_speed\_est\_ms}.$$

Then

$$\texttt{axle\_speed\_delta\_ratio} = \begin{cases} \dfrac{v_r - v_f}{\hat{v}}, & \hat{v} > 0.1, \\ 0, & \text{otherwise.} \end{cases}$$

Yaw response features Yaw response was represented using a normalised steering-speed denominator:

$$d_t = \delta_t \cdot \hat{v}_t,$$

where δ_t is steering input and $\hat{v}_t = \texttt{vehicle\_speed\_est\_ms,t}$.
A validity mask was applied so that yaw gain was only computed at non-trivial speeds and away from numerical singularities:

$$\mathrm{valid}_t \equiv (\hat{v}_t > 1.0) \ \wedge \ (|d_t| > 10^{-6}).$$

The yaw-gain feature was then defined as

$$\texttt{yaw\_gain}_t = \begin{cases} \dfrac{r_t}{d_t}, & \mathrm{valid}_t \\ 0.0, & \text{otherwise} \end{cases}$$

where r_t is yaw rate.
A "yaw status" feature was also defined as a centred, bounded version of yaw gain:

$$\texttt{yaw\_status}_t = \tanh(\texttt{yaw\_gain}_t).$$

2.4.2 Automatic Labelling

The final step in the dataset generation process was to derive ground-truth labels for the traction loss events based on the simulator's absolute state data. This was done using a hysteresis-based approach to define event boundaries based on wheel slip values, which provided a consistent and objective criterion for labelling. Labels were in the format of "GRIP" for normal driving and "EVENT" for traction loss, with the option to include a "PREP" label for a configurable number of timesteps before event onset to encourage early detection.

Each scenario had a configured slip threshold that defined the point at which the event was considered to have started, and an exit threshold multiplier that defined how much the slip value needed to reduce before the event was considered to have ended. This approach allowed for fast detection of event onset while avoiding jitter around the event boundaries that could arise from using instantaneous slip values alone.
For each timestep t, the maximum slip across all wheels was computed as

$$s_t = \max(s_{FL,t}, s_{FR,t}, s_{RL,t}, s_{RR,t}) \,.$$

An entry threshold T was defined from the configured slip threshold, and an exit threshold was defined as

$$T_{\text{exit}} = \beta T,$$

where β is the exit-threshold multiplier.
A short rolling history of slip values was maintained and used to compute

$$\bar{s}_t = \frac{1}{N_t} \sum_{i=1}^{N_t} s_i,$$

where N_t is the current history length. Event entry uses instantaneous slip s_t, whereas event exit uses the rolling mean $\bar{s}_t$ to reduce label jitter.
The state-transition logic was:

$$\text{if prep label is provided} \Rightarrow \begin{cases} \text{reset event phase and history} \\ \text{return prep label} \end{cases}$$

$$\text{if not in event phase and } s_t \geq T \Rightarrow \text{return event label}$$

$$\text{if not in event phase and } s_t < T \Rightarrow \text{return GRIP}$$

$$\text{if in event phase and } \bar{s}_t < T_{\text{exit}} \Rightarrow \text{return GRIP}$$

$$\text{if in event phase and } \bar{s}_t \geq T_{\text{exit}} \Rightarrow \text{return event label}$$

This asymmetric entry/exit criterion ensured fast detection of event onset while avoiding premature return to GRIP due to transient reductions in slip. The use of a rolling mean for exit also encouraged the models to learn patterns of sustained recovery.

2.5 Data Pre-processing and Segmentation

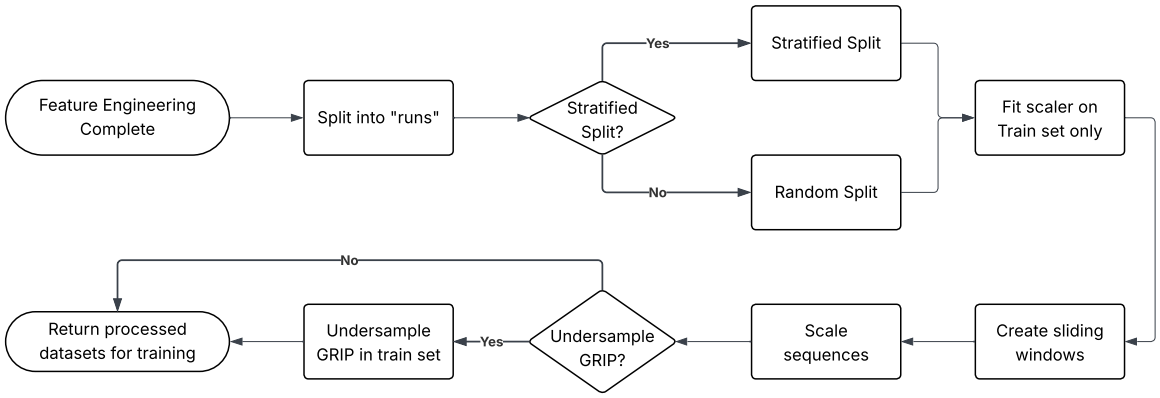

Figure 2.1: A flowchart dictating the pre-processing pipeline for data after feature engineering.

Once the raw data had been collected and additional features engineered, data was processed to prepare it for model training. As shown in Figure 2.1, this involved a variety of steps including splitting and sampling the data depending on the quantity and distribution.

- **Train/Test/Validation Split**: Once feature engineering was complete, the dataset was grouped at run level to prevent leakage between splits. This ensured that temporally adjacent samples from the same execution could not appear in both training and evaluation data. To preserve scenario diversity, stratified splitting by `test_type` was used when there were enough runs to support stable class-wise allocation. The minimum run count for stratification was set to

$$\max(10,\ 3 \times N_{\text{types}}),$$

  where N_{types} is the number of unique test types. If this condition was met, the data was split in two stages: first into training and temporary sets, then the temporary set into validation and test sets, preserving the requested proportions. If stratification failed (for example, due to insufficient members in one class after the first split), the pipeline automatically fell back to an unstratified random split.

  When the dataset was too small to satisfy the stratification threshold, the pipeline directly used random run-level splitting. In all cases, reproducibility was enforced with a fixed random state (42), and the selected run identifiers were merged back with the full sample table to produce train, validation and test dataframes.

- **Training-Only Scaling and Sliding-Window Segmentation**: To avoid data leakage, feature normalisation was fitted using the training partition only. Specifically, the scaler was fitted on the training feature matrix, learning per-feature mean and standard deviation:

$$x' = \frac{x - \mu_{\text{train}}}{\sigma_{\text{train}}}.$$

  No validation or test samples were used when estimating these statistics.

After splitting, each subset (train, validation, test) was independently converted into fixed-length temporal sequences using a sliding-window procedure applied per run. For each run, the first frame was repeated (`window_size` − 1) times as pre-padding so that predictions could be formed from the start of the run without discarding early timesteps. Windows were then advanced with stride `stride` (1 in the final implementation), and each window label was assigned from the final timestep in that window. This produced aligned sequence datasets for all three splits while preserving run boundaries and temporal order.

- **Class Imbalance Handling and Feature Scaling**: Feature scaling was applied to each split using the scaler parameters learned from the training partition. Thus, train, validation and test windows were transformed into a common standardised feature space while avoiding leakage of evaluation statistics into pre-processing.

  To address class imbalance, undersampling was applied only to the training set. Specifically, the majority GRIP class was randomly reduced to a configurable ratio relative to the total number of event samples:

$$N_{\text{GRIP,target}} = \texttt{grip\_ratio} \times N_{\text{event}}.$$

  In the final implementation, the grip ratio was set to 1. If the current number of GRIP samples was already below this target, no undersampling was performed. Validation and test sets were intentionally left unchanged so that evaluation reflected the natural class distribution rather than an artificially balanced one.

Once this is complete, the resulting datasets were ready for training. The processed data was saved in a format that allows for efficient later loading.

2.6 Model Training

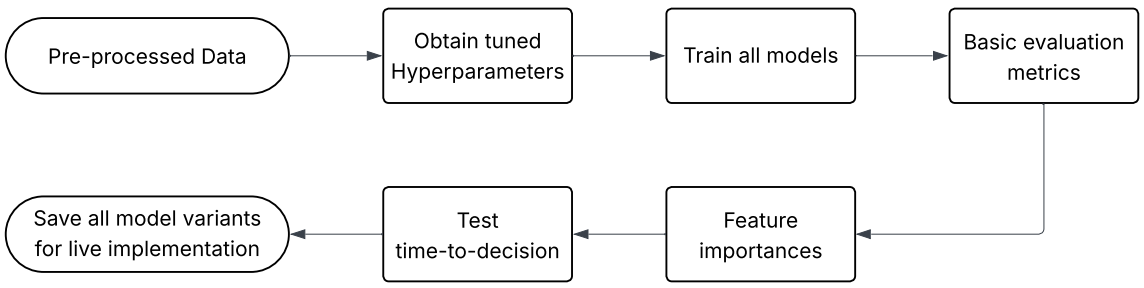

Figure 2.2: A flowchart dictating the model training and evaluation pipeline.

The model training pipeline as shown in Figure 2.2 was designed to be flexible and modular, allowing for experimentation with different architectures and direct, automatic comparison between them. After loading the pre-processed datasets, the candidate models chosen were trained under a consistent framework and evaluated using the same metrics to ensure fair comparison.

2.6.1 Models Selection and Two-Stage Approach

As mentioned in the previous chapter, the candidate models selected were a 1-Dimensional CNN, an LSTM network [11], a GRU [12], a Transformer model [13] and a Random Forest as a non-sequential baseline.

Additionally, the capability to train a two-stage classifier was implemented. This involved selecting the strongest performing model and using it to train a binary classifier for event detection, which would then feed into a second model that would classify the specific event type. This approach was motivated by the idea that separating the detection and classification tasks could allow for more focused learning in each stage and potentially improve overall performance [29].

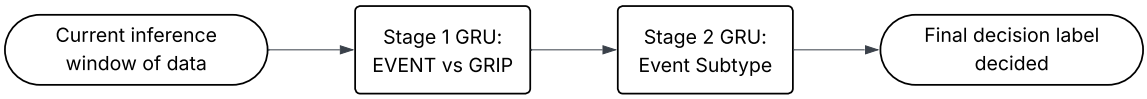

Figure 2.3: A flowchart dictating the two-stage classification approach.

2.6.2 Hyperparameter Tuning and Model Training

Some models, specifically the GRU, LSTM and Random Forest, lent themselves to hyperparameter tuning to optimise their performance. As part of the pipeline, a tuning framework was implemented that iteratively trains models with different hyperparameter configurations and evaluates their performance on the validation set to identify the best combination. For the deep learning models, this tuned parameters such as the number of layers, dropout rate, batch size and learning rate. For Random Forest, a grid search was implemented to explore different `n_estimators`, maximum depth, `min_samples_split`, maximum number of features and minimum samples per leaf. Additionally, tuning for the two-stage classifier was conducted to find the optimal decision threshold for passing between the first and second stage. The selection process was designed to be efficient and systematic, allowing for a thorough exploration of the possible hyperparameter space. At the end of tuning, the best performing hyperparameters were saved so that future training runs could be conducted with the optimal configuration without needing to repeat the tuning process.

Once optimal hyperparameters were identified, the final models were trained on the full training set and evaluated on the test set to compare their performance. A notable aspect of training was the strategy used to weight the loss function to account for class imbalance. Sparse categorical cross-entropy with class-weighted training was used for the deep learning models, where the weight for each class was inversely proportional to its frequency in the training data. The goal was to manipulate the models into paying more attention to the minority event class and to mitigate the bias towards predicting the majority GRIP class.

2.6.3 Immediate Model Evaluation Metrics

Throughout the pipeline, training information and model performance were recorded and visualised for analysis. This included tracking accuracy and loss curves during training, as well as computing evaluation metrics such as precision, recall and F1-score. Confusion matrices were also generated to provide insight into the types of errors being made by the models and guide further iteration and development.

One specific piece of evaluation that proved extremely useful was the calculation of feature importance for the models. This quantified the contribution of each feature to the model's predictions and provided critical insight into the underlying patterns being learned by the models. If this exposed the models being dominated by some specific features, it could indicate potential issues with the data or model architecture that would need to be addressed. Additionally, it provided a way to validate the models against domain knowledge and ensure that they were learning meaningful patterns rather than overfitting to noise in the data.

Finally, time-to-decision was also measured for the models to establish their suitability for real-time inference. This was done by feeding the model with raw training data at the rate it would receive it in a real-time application and measuring the time taken to produce a prediction. This provided a practical benchmark for the models' performance in a real-world context and helped to identify any potential bottlenecks or inefficiencies in the model architecture or implementation.

2.7 Real-Time Implementation

With models prepared and optimised, the final stage of development was to implement a real-time event detection and classification system that hooked into the simulator and processed telemetry data on the fly. As shown in Figure 2.4, the final implementation used the trained two-stage detector and the fitted pre-processor from the offline pipeline. Incoming BeamNG telemetry was sampled continuously, converted into the same engineered feature set used during training, and accumulated into a fixed-length sliding window. That window was then scaled using the training-set normalization parameters and passed to the model for inference. To reduce flicker from frame-to-frame noise, the live system also applied a small amount of temporal hysteresis, requiring repeated event evidence before starting or ending an event state.

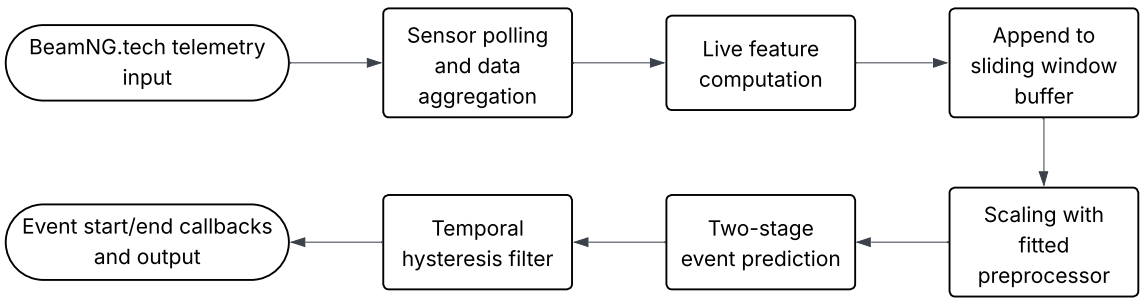

Figure 2.4: A flowchart illustrating the real-time model pipeline.

Chapter 3

Implementation and Validation

3.1 Dataset Generation

This section will cover the programmed implementation and configuration of the data generation pipeline.

3.1.1 Initial Simulator Configuration

BeamNG.py [33] provided the bulk of the functionality required to implement the data generation process, but some additional work was required to expose data necessary for this project's purposes. The most significant of these customisations was the development of a basic Lua mod that exposed individual wheel speeds to the API, which was necessary to implement the labelling strategy and for use as features. This Lua mod is injected into the simulator at runtime as a `.zip` file, and the wheel speed data is accessed through the API as an extension of the vehicle's electrics data.

3.1.2 Automated Scenario Execution and Labelled Data Generation

The project implements `beamng.sensors.Electrics`, `beamng.sensors.State` and `beamng.sensors.AdvancedIMU` classes for data acquisition as well as the `Scenario` and `Vehicle` classes for handling the simulation instance and the vehicle respectively. The coordinating file `generate_data.py` is responsible for setting up the simulator and executing the scenarios, but the scenario design and labelling strategy are implemented in separate files that are imported into `generate_data.py`. This allows for easy modification and extension of the scenarios and labelling strategy without needing to modify the core data generation process.

The data-generation scenarios are built on a shared abstract class, `BaseScenario`, which standardises how each scenario is defined and executed. It enforces a common interface through abstract members for scenario type and per-iteration behaviour, while providing reusable logic for iteration control, simulator reset, data collection and CSV writing. It also governs the aforementioned phase-based slip detector and the hysteresis overseeing it. This design keeps scenario implementations modular and concise, while ensuring consistent execution and labelling across all generated data. Each scenario implements the required members to define its unique behaviour and labelling strategy, as well as the scripting for vehicle behaviour. When `generate_data.py` is executed, it creates a single instance of BeamNG.tech and runs each scenario a predefined number of times. This, amongst other parameters, can be configured in the global `config.py` file. The resulting data is written into `generated_data.csv`.

The final dataset contained approximately 42000 rows of telemetry data across 10 to 50 iterations of each of the 5 scenarios, depending on the scenario's duration and the configured number of iterations. Before balancing, the dataset had a GRIP label distribution of

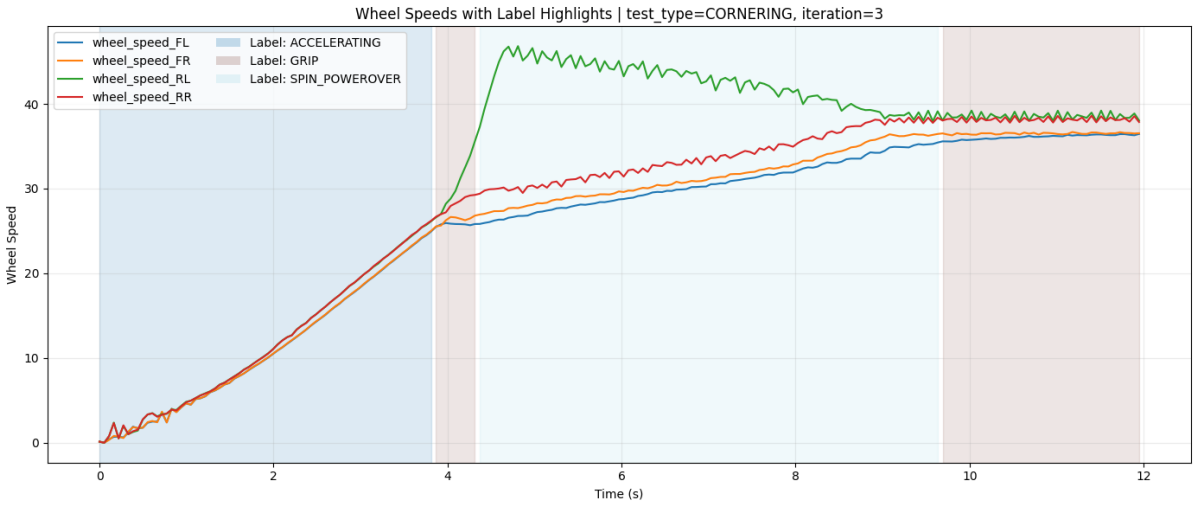

Figure 3.1: Raw telemetry data from a single run of the 'powerover' scenario. The y axis shows the wheel speeds, shading indicates label and the x axis shows time. The loss of traction on especially the rear-left wheel is visible.

approximately 80%. A visualisation of the raw data of a single run is shown in Figure 3.1.

3.2 Data Pre-processing

The pre-processing stage is implemented in the `EventDataProcessor` class in `preprocessing.py`. Its role is to transform the raw CSV output from the data-generation pipeline into fixed-length, normalised sequences suitable for sequence models. The class encapsulates loading, cleaning, feature construction, run-level splitting, window generation, scaling and optional class balancing so that the same pre-processing logic can be reused consistently during training and deployment.

3.2.1 Raw Data Cleaning and Label Encoding

The pipeline begins by loading the generated CSV file into a `pandas.DataFrame`. The `gear` column is coerced to numeric form with invalid values converted to zero, which ensures that all rows have a valid numeric representation. After this initial cleaning, the original string labels are mapped to integer class indices using the global `LABEL_TO_CLASS` mapping defined in `config.py`. Any label that is not explicitly mapped is assigned to the `GRIP` class. For convenience, a binary `is_event` column is also created, where values greater than zero indicate an event state. Finally, the dataframe is sorted by `test_type`, `iteration` and `time` so that all later sequence operations preserve the original temporal order.

3.2.2 Feature Construction in Code

After loading and cleaning the raw telemetry, the engineered features detailed in Chapter 2 are computed programmatically. These columns are added via the `engineer_features()` method, extending the original feature set. To ensure temporal accuracy, time-series operations such as the rate-of-change derivatives are calculated separately within each `test_type` and `iteration`

group, preventing invalid calculations across run boundaries. Once computations are completed, an automated pass replaces any resulting infinite or missing values with zero to ensure pipeline stability. The new feature names are then appended to the active feature list so they are automatically included in later scaling and windowing steps.

3.2.3 Run-Level Splitting and Windowing

Following feature extraction, the dataset is divided at the run level utilising the unique `test_type` and `iteration` pairs to produce the training, validation, and test partitions defined in the methodology. Once the split is complete, each subset is programmatically converted into fixed-length sliding windows, processed separately for each run. The window stride is passed as a configurable parameter, allowing the pipeline to quickly toggle between densely overlapping or more sparsely sampled sequences depending on the experimental setup.

3.2.4 Scaling and Class Balancing

During the scaling phase, the scaler object is fitted exclusively to the training dataframe prior to any sequence transformation. Following this, the pipeline executes the class balancing strategy by undersampling the `GRIP` class based on the configurable `grip_ratio` parameter. This operation is applied strictly to the training partition, outputting the final, balanced feature arrays ready for model ingestion.

3.2.5 Pre-processor Persistence

Once pre-processing is complete, the fitted processor and processed arrays are serialized to disk, making the exact configuration reusable for real-time deployment.

The pipeline processed 170 distinct simulator runs, generating sequences with a window size of 20 timesteps across 29 extracted and engineered features. Following the run-level split, the targeted undersampling strategy was applied exclusively to the training partition with `grip_ratio` set to 1.0. This reduced the majority `GRIP` sequences from 20,025 to 4,549, resulting in a balanced final training tensor of shape (9098, 20, 29). The balancing logic did not modify the evaluation partitions; the validation and test tensors, with shapes (8190, 20, 29) and (8177, 20, 29) respectively, retained their natural class distributions to preserve representative performance estimates.

3.3 Hyperparameter Tuning

Hyperparameter tuning was implemented as a separate stage in `hyperparameter_tuning.py` to keep model-search logic isolated from final training and evaluation runs. For recurrent deep-learning models (GRU and LSTM), Keras Tuner [14] Bayesian optimisation was used with validation accuracy as the objective, exploring the configuration space defined in the methodology. During each trial, early stopping was used to terminate poor configurations quickly. After search completion, the best hyperparameters were persisted to `models/tuned_<model>_hps.pkl` and the corresponding tuned model was saved to `models/tuned_<model>.keras`. A final fit was then performed using the selected configuration

with longer training and learning-rate reduction on plateau. This optimisation process yielded tangible improvements: the LSTM model's macro F_1-score increased from 0.863 to 0.894 (with accuracy rising to 94.89%), while the already well-optimised GRU saw a marginal macro F_1 increase from 0.904 to 0.908 (reaching 95.78% accuracy).

For the Random Forest baseline, the grid search was executed using stratified cross-validation on flattened sequence inputs. Weighted F_1 was used as the optimisation score to better reflect imbalanced multi-class performance. The best estimator and parameter set were saved for later reuse, successfully elevating the model's macro F_1-score from 0.466 in its original configuration to 0.483 post-tuning.

A dedicated threshold-tuning routine was also implemented for the two-stage classifier. Instead of retraining networks for each threshold, stage-one and stage-two probabilities were precomputed once on the validation set and then reused across a threshold sweep. Candidate thresholds were scored using a combined objective that penalised GRIP false positives more heavily than missed events:

$$\text{score} = \text{event\_detection} - 2 \times \text{grip\_fp\_rate}$$

The best threshold was saved and later consumed by the training pipeline.

3.4 Model Training

Final training and comparative evaluation were orchestrated in `train_model.py`. The script loaded the pre-processed train, validation and test tensors from `processed_data.npz` and logged all console output to `training_log.txt` via a tee-style logger for reproducibility. TensorFlow [36] was used for deep-learning model training, while scikit-learn [37] handled the Random Forest implementation. The two-stage detector was implemented as a custom Keras model that encapsulated both stages and applied the tuned threshold during inference.

Single-stage model training supported the five architectures previously outlined. For deep models, the implementation used sparse categorical output with optional focal loss [31]. The focal loss implementation used $\gamma = 2.0$ and incorporated class-dependent weighting. Training stability was controlled with early stopping and learning-rate reduction callbacks. Deep models were trained with fixed validation monitoring, while Random Forest used flattened sequence tensors and class-balanced fitting. All trained models were serialized to the `models/` directory, and training-history plots were exported to `models/graphs/` for deep-learning runs. In addition to single-stage training, the two-stage detector pipeline was programmed to execute sequentially; separate class-weighted training was applied in each stage and inference was scripted to combine stage-one confidence with stage-two class probabilities under the dynamically tuned threshold.

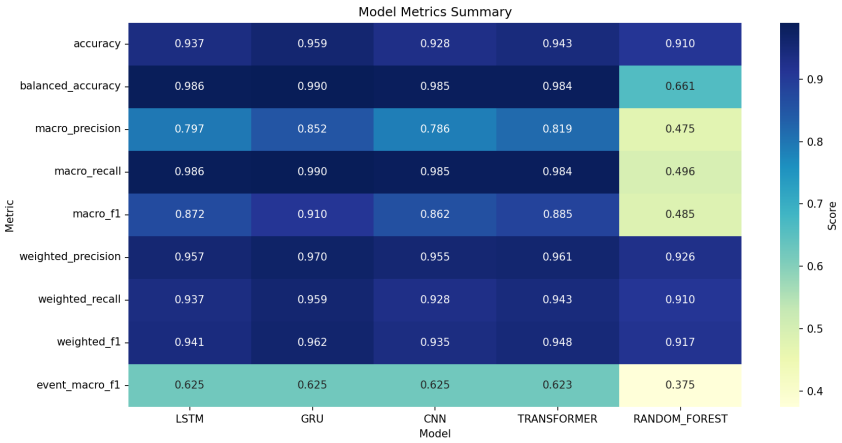

Figure 3.2: Heatmap of key evaluation metrics across candidate models.

3.4.1 Evaluation Metrics and Visualisations

Evaluation was implemented as a multi-level reporting pipeline rather than a single accuracy value. For each trained model, the test workflow generated:

- full multi-class classification reports across all configured classes,
- event-only metrics excluding GRIP (event macro F1, event detection rate and event type accuracy),
- GRIP false-positive rate as a safety-critical error indicator,
- balanced accuracy, macro and weighted precision/recall/F1 for cross-model comparison,
- weighted one-vs-rest ROC AUC, which measures how well the model separates each class from the rest using class-weighted probability scores where available.

Confusion matrices were generated per model and exported as figures. A metrics summary table was also exported as a CSV and rendered as a heatmap to provide a compact programmatic comparison across candidate models. An example is shown in Figure 3.2.

For post-hoc interpretability, feature-importance analysis was implemented algorithmically based on model architecture. Permutation importance was used for all models, gradient-based sensitivity was computed for deep models, and native feature importance was extracted and timestep-aggregated for Random Forest. Results were saved as both figures and CSV files, and a consensus ranking was computed across models.

Finally, confidence-threshold sweeps were run on the best-performing single-stage model to quantify the trade-off between GRIP false positives and event detection sensitivity, outputting calibration data for deployment-oriented operating-point selection.

3.5 Real-Time Monitoring Implementation

The live event detection system, implemented in `live_model.py`, utilises a multithreaded architecture for non-blocking, real-time performance. Integration with the simulator is managed

Figure 3.3: Screenshot of the live monitoring output during a test run. Detected events are printed with their predicted types and confidence scores.

by the `BeamNGEventMonitor` class, which dynamically injects a custom `wheeldata.lua` script as a `.zip` archive upon initialization. This exposes individual wheel speeds without requiring manual mod installation. A dedicated daemon thread then continuously polls the vehicle's `Electrics`, `State`, and `AdvancedIMU` sensors at a target rate of 20Hz.

Raw sensor values are packaged into a `SensorReading` dataclass and routed to the `LiveEventDetector`, which calculates the required rate-of-change, axle aggregate, and yaw response features on the fly to match the offline pipeline. These feature arrays are appended to a sliding window buffer (`collections.deque`) sized to the trained `window_size` (20 timesteps). Once full, the sequence is converted to a NumPy array and normalised using parameters from the persisted `EventDataProcessor.pkl` file.

Model inference runs in a concurrent loop to prevent computation from delaying sensor polling. The scaled sequence is passed to the deserialized `TwoStageEventDetector` for class and confidence predictions. To stabilise the live output, a state machine applies temporal hysteresis, evaluating consecutive predictions against configurable start and end confidence thresholds. This logic triggers `on_event_start` and `on_event_end` callbacks, allowing the monitor to accurately log event durations and compile final session statistics.

During simulation, the monitor outputs event boundaries, predicted types, and confidence scores to the console, followed by a comprehensive summary log at the end of the session. This provides immediate feedback on loss-of-traction events, enabling real-time monitoring of vehicle dynamics and facilitating future integration with control systems. A screenshot of the live monitoring output during a test run is shown in Figure 3.3.

Chapter 4

Results, Evaluation and Discussion

4.1 Initial Model Results and Adjustments

The initial evaluation of the models using the test set demonstrated strong baseline performance across a majority of the tested architectures. Specifically, the GRU model achieved the highest overall performance with an accuracy of 95.88% and a macro $F1$-score of 0.910. The LSTM, Transformer and CNN models also performed well with accuracies exceeding 90%. The Random Forest model, whilst still achieving a respectable accuracy, lagged behind in most other metrics compared to the deep learning models. These results are summarised in the heatmap of key evaluation metrics shown in Figure 4.1.

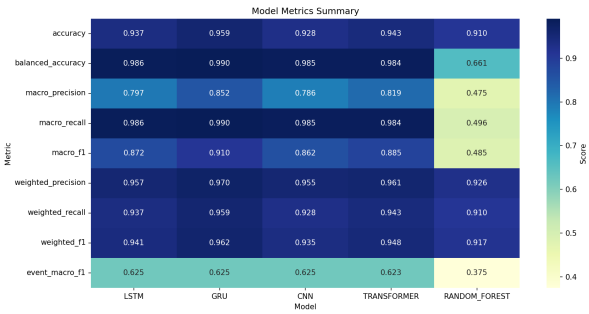

Figure 4.1: Heatmap of key evaluation metrics across candidate models.

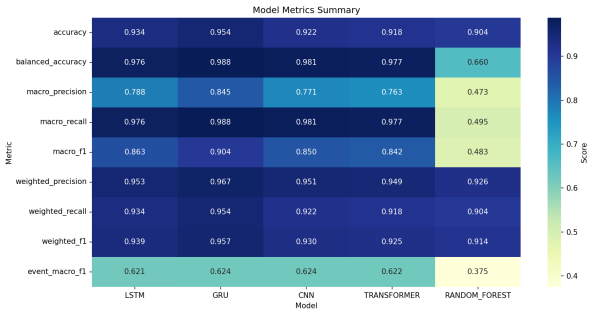

Figure 4.2: Heatmap of key evaluation metrics across candidate models (controls omitted).

Following this evaluation, a feature importance analysis shown in Figure 4.3 was conducted to understand the primary drivers behind each model's predictions and to identify any evidence of overfitting. This revealed that certain features, particularly those related to driver inputs, were consistently dominating the models' decisions. This raised immediate concerns about the models focusing too much on the programmatic inputs to the scenarios - whilst they had been varied to some extent, each scenario was still temporally scripted similarly in its execution each time.

To address this, the dominating features were removed from the pre-processed dataset and the model training pipeline re-run. This did not result in a significant drop in performance, as shown in the updated heatmap in Figure 4.2, with the GRU model maintaining an accuracy of 95.44% and a macro $F1$-score of 0.904. Further feature importance analysis (Figure 4.4) showed a much more balanced distribution of importance across the remaining features. This suggested that the models were now learning more generalisable patterns in the data rather than overfitting to specific input features. A full breakdown of average feature importance across all models is included in Appendix C.

Because the no-control variant effectively maintains high accuracy while offering a more robust understanding of the vehicle's true state, it provides a significantly more valuable framework

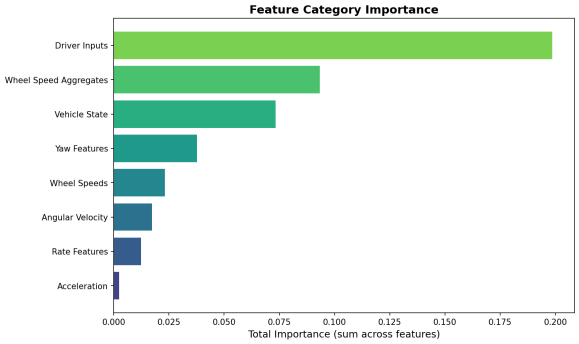

Figure 4.3: Feature importance across different categories.

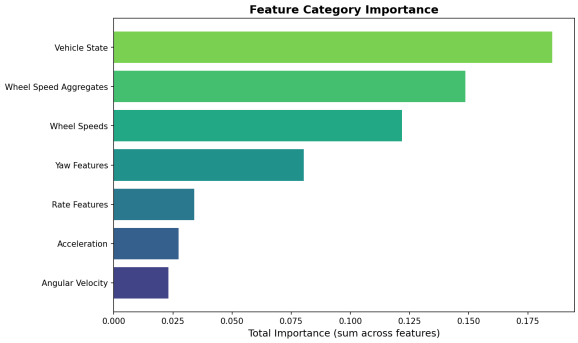

Figure 4.4: Feature importance across different categories (controls omitted).

for evaluation. Consequently, the remainder of the analysis and discussion will focus on the results from this variant of the models.

4.2 Model Comparison and Performance Analysis

Reviewing the metrics of the adjusted models, the recurrent architectures (GRU and LSTM) demonstrated the highest performance. The GRU was the strongest, achieving an accuracy of 95.44% and a macro F_1-score of 0.904. The LSTM followed with an accuracy of 93.42% and a macro F_1-score of 0.863. The strong performance of recurrent architectures is heavily aligned with the theoretical requirements of the task. Identifying traction loss requires an understanding of how kinematic variables evolve over time and the recurrent nature of GRUs and LSTMs allows them to inherently captures this temporal continuity, making them sensitive to the rapidly changing dynamics of a slip event.

The CNN and Transformer architectures also yielded strong results, achieving macro F_1-scores of 0.850 and 0.842 respectively. The CNN's success suggests that traction loss events possess distinct "spatial" patterns across the telemetry feature window, which 1D convolutions can successfully extract as feature maps. Meanwhile, the Transformer's self-attention mechanism proved capable of identifying complex interactions between different timestamps in the telemetry window. That being said, neither surpassed the GRU, suggesting that the strict chronological, inductive bias of recurrent networks is particularly well-suited for this specific scale of physical telemetry data.

Conversely, the baseline Random Forest struggled significantly with minority class identification, achieving a macro F_1-score of just 0.483. Because Random Forests evaluate data points as independent observations, they inherently lack the memory mechanisms required to process time-series data seamlessly. Without heavy manual feature engineering to represent temporal shifts (e.g., explicitly coding rolling averages or delta values), the Random Forest is largely blind to the trajectory of the vehicle's state, leading to a high rate of false negatives.

Confusion matrices and training curves for applicable models are provided in Appendix C.

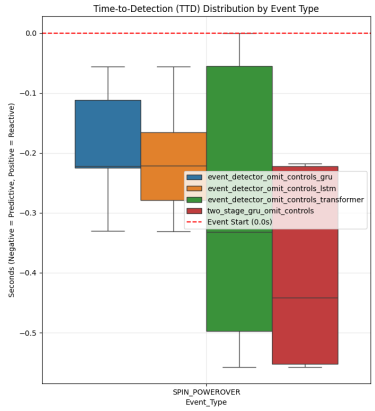

Figure 4.5: Comparison of Time-to-Detect (TTD) across models.

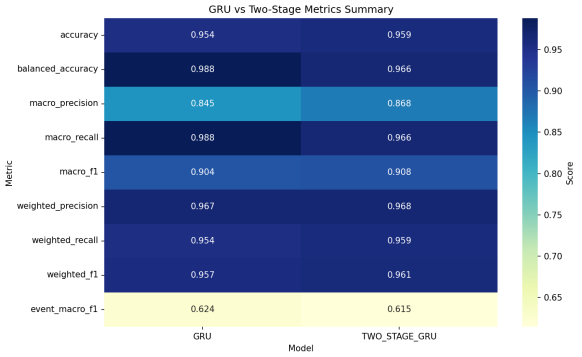

Figure 4.6: Comparison of key evaluation metrics between the single-stage GRU model and the two-stage GRU model.

4.2.1 Event Detection and Time-to-Detect (TTD)

A Time-to-Detect (TTD) analysis was performed to ensure the models were not only accurate but timely in their predictions. TTD was calculated as the time difference between the model's first correct detection of an event and the actual occurrence of the event as defined by the dataset's labels. The results highlight the divide between deep learning sequence models and traditional machine learning algorithms. The CNN, GRU, LSTM, and Transformer models successfully detected almost all events (averaging roughly 1070 detections out of a possible 1083). Crucially, these deep learning models exhibited negative Mean TTD values (e.g., -0.315 seconds for the CNN and -0.264 seconds for the GRU). A negative TTD indicates a predictive capability: the models are recognising the onset telemetry signatures of traction loss fractions of a second before the labelled event threshold is technically breached. Against unseen, noisy data, this margin is invaluable for making decisions in real-time. In stark contrast, the Random Forest model functionally failed as an event detector, missing 961 events and successfully identifying only 122. This discrepancy confirms that whilst immediate metrics may be reasonable, it is entirely unsuited for the continuous, predictive sequence monitoring required in live vehicle telemetry.

4.2.2 Two-Stage Model Performance

The two-stage model, based on the GRU architecture, demonstrated a capability to address the main weakness of the single-stage models at the cost of a slightly lower overall accuracy and higher time-to-detect 4.5. The original models' confusion matrices revealed the majority of misclassifications were false negatives, where the model failed to identify a traction loss event and instead classified as "GRIP". After tuning to locate the optimal threshold for passing events from the first stage to the second, the two-stage model successfully reduced false negatives by over 20%. As shown in the metric comparison in Figure 4.6, the model achieved incrementally higher scores across most metrics. However, it also exposes a slight decrease in Recall, indicating the model is now slightly more conservative in its classifications, which is a trade-off for the improved precision and reduced false negatives. This robustness is particularly

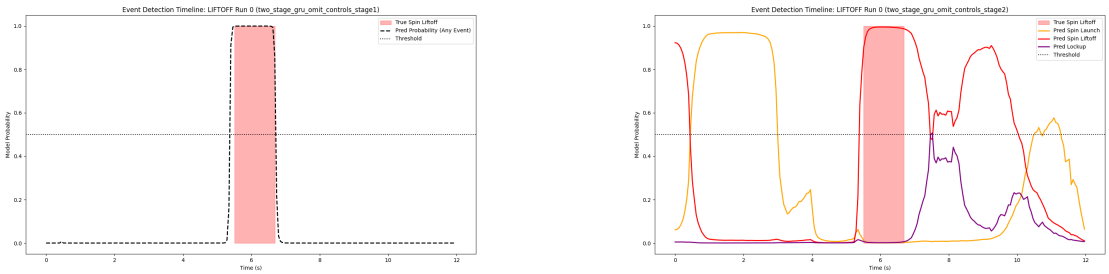

Figure 4.7: Active predictions for the two-stage GRU model.

valuable in a real-time implementation, where the cost of a false negative (failing to detect a traction loss event) is significantly higher than a false positive (incorrectly flagging an event that isn't actually a traction loss).

4.3 Real-Time Implementation Performance

The real-time implementation demonstrated the practical viability of the developed models in a live simulation environment. Utilising the two-stage GRU model, the system was able to process incoming brand new telemetry data in real-time and identify events as soon as they recognisably occurred in the simulation. A video of the simulator alongside the live identifications is provided in the supplementary materials and linked to in Appendix B. There were some minor instances of events being misidentified because of overlapping physical signatures but overall the ability to recognise an event was successfully demonstrated. The system's performance in real-time was consistent with the offline evaluation, with the model successfully identifying traction loss events with a similar time-to-detect as observed in the test set. This confirmed that the models were not only effective in a controlled, offline setting but also robust enough to handle the variability and noise inherent in real-time telemetry data. The decision to use the control feature-omitted models was also supported by the live implementation wherein the models were significantly less responsive to events dissimilar to the training data compared to the final ones.

Figures 4.7 show the live predictions of the two-stage GRU model during a sample run of the simulator. The first stage successfully identifies the time during which an event is taking place, whilst the second correctly classifies the event as "LIFTOFF" oversteer. Also visible is the second stage making constant predictions even whilst the first stage is inactive. This is a consequence of the model's architecture and training which allows it to make predictions based on the input data regardless of the first stage's output. The second stage's predictions are only considered valid when the first stage indicates that an event is occurring.

4.4 Conclusions

The results of this research demonstrated a robust proof-of-concept for real-time traction loss detection using deep learning model architectures. Through static evaluation and live testing

in-simulator, neural networks proved the most capable of capturing the temporal dynamics of vehicle telemetry with the optimised GRU model, achieving the best balance of accuracy, precision, recall and time-to-detect. The success of the deep learning models compared to the Random Forest baseline demonstrated a necessity for a level of context awareness and temporal processing that traditional machine learning algorithms struggle to provide without extensive feature engineering.

A critical milestone was the identification and mitigation of input-bias. By removing the driver control features from the dataset, the research demonstrated that traction loss can be accurately identified based purely on the physical and kinematic state of the vehicle. This not only improved the generalisability of the models but also provided a more robust framework for real-world application where driver inputs may be highly variable and less predictable.
This finding validates a core hypothesis of this project: that the use of a simulation environment to gather training data can enable the development of models that are not only accurate but also robust. By demonstrating that a model trained entirely on simulated physics can identify slip events without relying on programmatic driver inputs, this research provides a strong foundation for future work in this area, including the potential application of these models to real-world telemetry data and their deployment in embedded systems.

The development and deployment of the two-stage model validated the offline metrics and confirmed that the model can actively detect and classify specific events amidst the noise of noisy real-time data. This framework ultimately provides a capable and adaptable foundation for advanced vehicle monitoring using machine learning models and the use of simulation systems to train them.

4.5 Future Work

Given additional time and resources, the training dataset could be extended significantly to include a wider variety of scenarios and environmental conditions. This would further improve the generalisability of the models and their robustness to edge cases. This extra compute could also be used to explore more complex model architectures such as deeper recurrent networks, hybrid models that combine CNNs and RNNs or more advanced attention-based models.

Whilst the capability of simulators to generate large and useful datasets is a significant aspect of this research, there is also value in exploring the application of these models to real-world telemetry data. This would involve a laborious process of collecting and annotating real-world data, but it would provide a critical test of the models' performance and generalisability in real-world conditions.

Finally, the models developed in this research could be optimised for deployment in embedded systems or edge devices. TensorFlow [36] and PyTorch [38] both offer tools such as Pico-TFLMicro [39] for model quantization and optimisation that could be explored to reduce the computational requirements of the models whilst maintaining their performance. This would be a crucial step towards real-world implementation in vehicles or other systems where computational resources are limited.

References

[1] P. Taylor, N. Griffiths, A. Bhalerao, S. Anand, T. Popham, X. Zhou, and A. Gelencser. Data mining for vehicle telemetry. *Applied Artificial Intelligence*, 30(3):233–256, 2016.

[2] H. Li, E. Saldivar-Carranza, J. K. Mathew, et al. Extraction of vehicle can bus data for roadway condition monitoring. Technical report, Purdue University / Joint Transportation Research Program, 2020.

[3] ImechE. Formula Student. https://www.imeche.org/events/formula-student, 2026. [Accessed 24-04-2026].

[4] A. Agrawal, L. Cotter, et al. Wheel slip detection for an electric racecar. Technical report, Dartmouth Digital Commons, 2025.

[5] T. Freudling. Detecting oversteering in bmw automobiles with machine learning. *MatLab Technical Articles*, 2018.

[6] BeamNG GmbH. BeamNG.tech. https://www.beamng.tech/, 2025. [Accessed 24-04-2026].

[7] P. Maul, M. Mueller, F. Enkler, E. Pigova, T. Fischer, and L. Stamatogiannakis. BeamNG.tech Technical Paper. Technical report, BeamNG GmbH, Bremen, Germany, 2021.

[8] J. G. Adigun. Ground truth data for object detection in autonomous vehicle using a driving simulator - beamng.tech. Master's thesis, Tallinn University of Technology, 2021.

[9] H. Ismail Fawaz, G. Forestier, J. Weber, L. Idoumghar, and P.-A. Muller. Deep learning for time series classification: a review. *Data Mining and Knowledge Discovery*, 33(4):917–963, 2019.

[10] R. Rajamani. *Vehicle Dynamics and Control*. Springer Science & Business Media, 2nd edition, 2011.

[11] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[12] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, 2014.

[13] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

[14] T. O'Malley, E. Bursztein, J. Long, F. Chollet, H. Jin, L. Invernizzi, et al. Keras Tuner. https://github.com/keras-team/keras-tuner, 2019. [Accessed 24-04-2026].

[15] K. Rock, S. Beiker, S. Laws, and J. Gerdes. Validating gps based measurements for vehicle control. In *Proceedings of the ASME International Mechanical Engineering Congress and Exposition*, 01 2005.

[16] H. B. Pacejka. *Tire and Vehicle Dynamics*. Butterworth-Heinemann, 3rd edition, 2012.

[17] Honda Automobile R&D Center. Development of traction control systems for formula one. Technical report, Honda, TODO.

[18] M. Klomp, Y. Gao, and F. Bruzelius. Longitudinal velocity and road slope estimation in hybrid electric vehicles employing early detection of excessive wheel slip. *Vehicle System Dynamics*, 52(sup1):172–188, 2014.

[19] D. Zhang, Q. Song, G. Wang, and C. Liu. A novel longitudinal speed estimator for four-wheel slip in snowy conditions. *Applied Sciences*, 11(6):2809, 2021.

[20] Z. Wang, W. Yan, and T. Oates. *Time series classification from scratch with deep neural networks: A strong baseline*. IEEE, 2017.

[21] A. T. van Zanten. Bosch esp systems: 5 years of experience. *SAE Transactions*, 109:428–436, 2000.

[22] R. Matsuzaki and A. Todoroki. Wireless monitoring of automobile tires for intelligent tires. *Sensors*, 8(12):8123–8138, 2008.

[23] M. Croft-White. Measurement and analysis of rally car dynamics at high attitude angles. Master's thesis, Cranfield University, TODO.

[24] P. Bazilinskyy. The effects of auditory feedback in a racing simulator on a car's slip angle relative to the ideal slip angle. Master's thesis, 2017. Research Paper/Thesis.

[25] R. J. Chen, M. Y. Lu, T. Y. Chen, D. F. Williamson, and F. Mahmood. Synthetic data in machine learning for medicine and healthcare. *Nature Biomedical Engineering*, 5(6):493–497, 2021.

[26] M. Frid-Adar, I. Diamant, E. Klang, M. Amitai, J. Goldberger, and H. Greenspan. Gan-based synthetic medical image augmentation for increased cnn performance in liver lesion classification. *Neurocomputing*, 321:321–331, 2018.

[27] J. Hwangbo, J. Lee, A. Dosovitskiy, D. Bellicoso, V. Tsounis, V. Koltun, and M. Hutter. Learning agile and dynamic motor skills for legged robots. *Science Robotics*, 4(26):eaau5872, 2019.

[28] M. Andrychowicz, B. Baker, M. Chociej, R. Józefowicz, B. McGrew, J. Pachocki, A. Petron, M. Plappert, G. Powell, A. Ray, et al. Learning dexterous in-hand manipulation. *The International Journal of Robotics Research*, 39(1):3–20, 2020.

[29] C. N. Silla and A. A. Freitas. A hierarchical classification approach for time series data. *Data Mining and Knowledge Discovery*, 22(1-2):31–72, 2011.

[30] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16:321–357, 2002.

[31] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár. Focal loss for dense object detection. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2980–2988, 2017.

[32] A. Theissler, J. Pérez-Velázquez, M. Kettelgerdes, and G. Elger. Predictive maintenance enabled by machine learning: Use cases and challenges in the automotive industry. *Reliability Engineering & System Safety*, 215:107864, 2021.

[33] BeamNG. BeamNGpy. https://github.com/BeamNG/BeamNGpy, 2025. [Accessed 24-04-2026].

[34] LucasBE. Carbonworks F4. https://www.beamng.com/resources/carbonworks-f4.27561/, 2026. [Accessed 24-04-2026].

[35] Leeds Gryphon Racing. Leeds Gryphon Racing. https://www.leedsgryphonracing.com/, 2026. [Accessed 24-04-2026].

[36] Google Brain Team. TensorFlow. https://www.tensorflow.org/, 2025. [Accessed 24-04-2026].

[37] Scikit-learn Developers. scikit-learn. https://scikit-learn.org/, 2025. [Accessed 24-04-2026].

[38] PyTorch Team. PyTorch. https://pytorch.org/, 2025. [Accessed 24-04-2026].

[39] raspberrypi. Pico TensorFlow Lite Micro. https://github.com/raspberrypi/pico-tflmicro. [Accessed 24-04-2026].

Appendix A

Self-appraisal

A.1 Critical self-evaluation

For this project, I researched the capability of machine learning models to classify events based on time-series data. I also investigated the aptitude of simulator-derived data to train such models and the importance of different features for classification.

I am particularly proud of the modular nature of the codebase developed for this project. Each phase of the pipeline is encapsulated in its own module, with some decomposing further into submodules. This design allowed for easy experimentation with different models and configurations, as well as straightforward debugging and maintenance. The class architecture for simulator scenarios was also a practical highlight as it made testing and authoring new scenarios much more efficient. The validation and evaluation of the models was also a strong point with comprehensive metrics and visualisations generated automatically throughout the training process.

During development, I encountered several challenges that shaped the final perspective of the paper and the project as a whole. Initially in testing, I had used the simulator-derived Ground Speed value to calculate Slip for each wheel as a feature for the models which, I realised later, was an example of data leakage as this compromised the real-world applicability of the models. This made itself evident in the models performing far too well than expected in static testing but abysmally in the live implementation. However, having to work around this realisation led me to researching how solving this issue is done in actual automotive experimentation and I was able to shape my justification for using the simulator around suggesting an alternate and more accessible way of conducting such experiments in the real world.

BeamNG.tech was the perfect tool for this project as it provided a unique soft-body simulation and python API that allowed for the execution of complex scenarios and the collection of detailed data. However, I spent an extended period attempting to extract individual wheel speeds from the simulator; there were at least three other methods attempted that just returned a single average value which was not informative for the models at all. Eventually, I developed the Lua mod mentioned in the report which exposed my desired values to the API, but I spent a significant amount of time on this problem which could have been used for other aspects of the project.

Overall, I am satisfied with the outcome of the project and the report. I believe it demonstrates a solid understanding of machine learning concepts and their application to time-series classification, as well as an appreciation for the challenges and limitations of using simulator-derived data. The project also provided valuable experience in software development, experimentation, and critical analysis.

A.2 Personal reflection and lessons learned

Whilst working on this project I became intimately familiar with many practical and theoretical concepts I had not before encountered.

I learned to use the BeamNG.tech simulator and its accompanying Python API which, despite playing the game (BeamNG.drive) in the past, was a completely new experience for me. I also had to grasp some basic Lua and the internal scripting of the software to expose the wheel speed values needed for the project. Finally, to modify LucasBE's Carbonworks F4 mod to befit the project better, I had to learn how vehicles in BeamNG are structured in terms of their internal `.jbeam` files and how to modify them.

Despite having experience training models using `scikit-learn` in the past, I had not worked with TensorFlow or Keras before this project. I became familiar with these libraries and the process of building, training and evaluating the deep learning models used in the project. Beyond this, I also broadened my knowledge of the capabilties of different model architectures, especially in the context of temporal data and time-series classification.

Research for this project taught me a lot about the challenges of using simulator-derived data for training machine learning models, especially in terms of data leakage and the importance of feature selection. I also learned about the practicalities of conducting experiments in the automotive domain, which shaped my goal of demonstrating how simulators can be used to overcome some of these challenges.

A.3 Legal, social, ethical and professional issues

A.3.1 Legal issues

Use of the BeamNG.tech simulator under an education license is compliant with the terms set by BeamNG GmbH, which allows for the use of their software for educational and research purposes. The project does not involve any commercial use or distribution of the software, ensuring that it adheres to the licensing agreement. Additionally, all external software and assets used in the project are properly cited and credited, respecting intellectual property rights. This includes the use of the Carbonworks F4 mod created by LucasBE, which is acknowledged in the report.

Even though the data used in this project is generated from a simulator, it is important to ensure that any future real world telemetry data collected for similar research adheres to data protection laws and regulations, such as the General Data Protection Regulation (GDPR) in the European Union. This would involve obtaining informed consent from participants, ensuring data anonymization, and implementing appropriate security measures to protect sensitive information.

Misclassification of events by the models in a real-world automotive setting could create legal exposure if a model is treated as a safety-critical system. However, the models developed in this project are intended for research and educational purposes only and are not deployed in

any real-world applications. It is crucial to clearly communicate the limitations of the models and the scope of their applicability to avoid any potential legal issues arising from misuse or misinterpretation of the results.

A.3.2 Social issues

Simulator-based research in the automotive domain lowers the cost barriers to entry compared to track testing, making experimentation more accessible for students and smaller teams looking to conduct similar studies. However, increased automation in the domain may shift roles from manual monitoring towards more supervisory or diagnostic tasks, which could have implications for employment and the skill sets required in the industry.

The project also contributes to the broader field of machine learning and its applications in potentially safety-critical systems, which has significant social implications. The development of robust models for event classification in automotive settings could enhance safety and reduce accidents, but it also raises questions about accountability and the ethical use of AI in decision-making processes.

A.3.3 Ethical issues

The slip-feature issue discussed earlier is an example of how apparently strong results can be misleading if feature design leaks target information. Ethically, the problem was handled and acknowledged in the report with the conclusions reframed a more realistic capability. It is also important to not claim real-world readiness without robust validation; it is stated multiple times in the report that the models are not ready for real-world deployment and that the results are only indicative of the potential of simulator-derived data for training such models in concept.

When weighting the models' decisions, false negatives are valued more than false positives as the former could lead to a failure to detect a critical event, whereas the latter would just lead to an incorrect classification. This is a common ethical consideration in safety-critical systems where the cost of missing a critical event is much higher than the cost of a false alarm.

A.3.4 Professional issues

Professionally, the project evidences good software development and ML practices. The project made great use of version control and the implementation being modulae and well-maintained betray a level of traceability that should be expected in professional software work.

Identifying and correcting the data leakage and feature bias issues shows a reflective practice and responsible model development. It is also critical to have presented this system as a research prototyle and not a production ready "safety controller" or similar. The report is careful to not overstate the results and to acknowledge the limitations of the models and the data used, which is an important aspect of professional integrity in research.

A.4 Security Appraisal

The project does not involve any sensitive data or real-world deployment so there are no significant security concerns directly associated with the work. However, if the models were to be deployed in a real-world automotive setting, it would be crucial to ensure that they are robust against adversarial attacks and that appropriate security measures are in place to protect against unauthorized access or manipulation of the system. This would involve implementing secure coding practices, conducting thorough testing and validation, and ensuring that any data used for training or inference is properly protected - a strength that simulator derived data has over real-world data in that it can be generated and used without the same privacy and security concerns.

Appendix B

External Material

B.1 External Software & Assets

BeamNG.tech Education License provided by BeamNG GmbH. All rights reserved. The BeamNG.tech Education License allows for the use of the BeamNG.tech software for educational purposes, including research and development in the field of machine learning and autonomous systems. The license prohibits commercial use and distribution of the software, ensuring that it is used solely for academic and research purposes. For more information on the terms and conditions of the BeamNG.tech Education License, please refer to the official documentation provided by BeamNG GmbH [6].

Carbonworks F4 BeamNG mod originally created by LucasBE [34].

B.2 Live Demonstration Video

The video of the live demonstration of the two-stage GRU making live observations in the simulator is available at https://www.youtube.com/watch?v=5sr14xF8zwo.

B.3 Python Dependencies

The code for the project is available in the public GitHub repository at https://github.com/LouWasHere/Dissertation

The full, pinned environment is listed in `requirements.txt`. A summary of external libraries used is given in Table B.1.

Table B.1: Summary of Python dependencies used in this project.

Category	**Packages (version)**
Simulator API	beamngpy (1.34.1)
Data and ML	numpy (2.3.5), pandas (2.3.3), scipy (1.16.3), scikit-learn (1.7.2), joblib (1.5.2)
Deep learning	tensorflow (2.20.0), keras (3.13.1), keras-tuner (1.4.8)
Visualisation	matplotlib (3.10.7), seaborn (0.13.2), pillow (12.0.0)
Notebook support	ipykernel (7.1.0), ipython (9.7.0), jupyter-client (8.6.3), jupyter-core (5.9.1)

Appendix C

Additional Results and Graphs

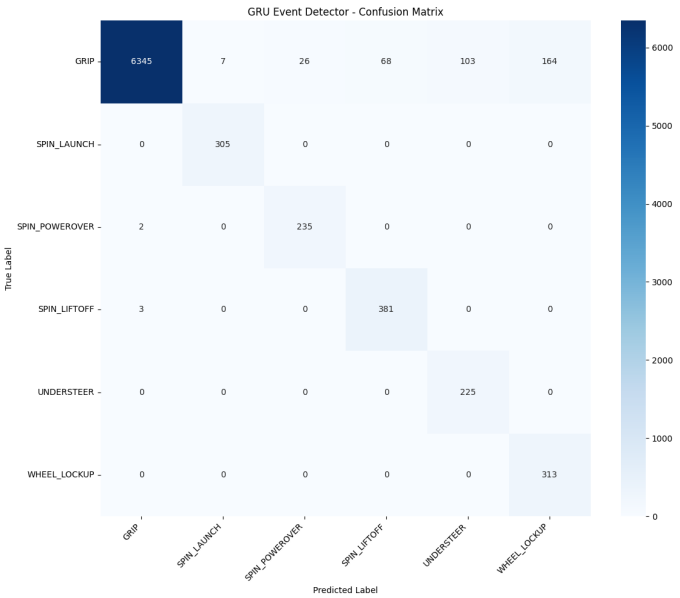

Figure C.1: Confusion matrix for the GRU model (controls omitted).

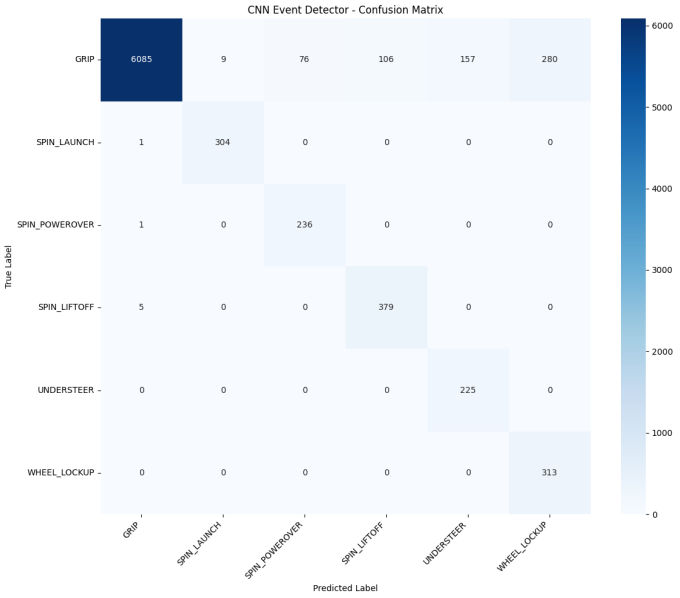

Figure C.2: Confusion matrix for the CNN model (controls omitted).

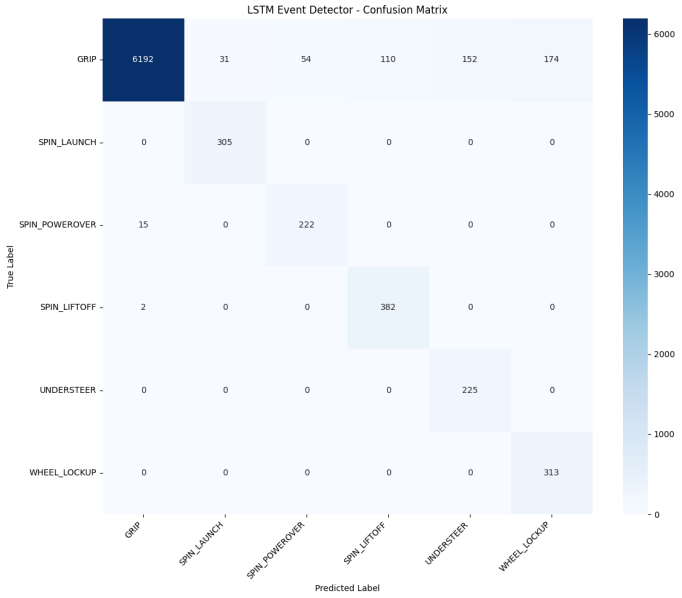

Figure C.3: Confusion matrix for the LSTM model (controls omitted).

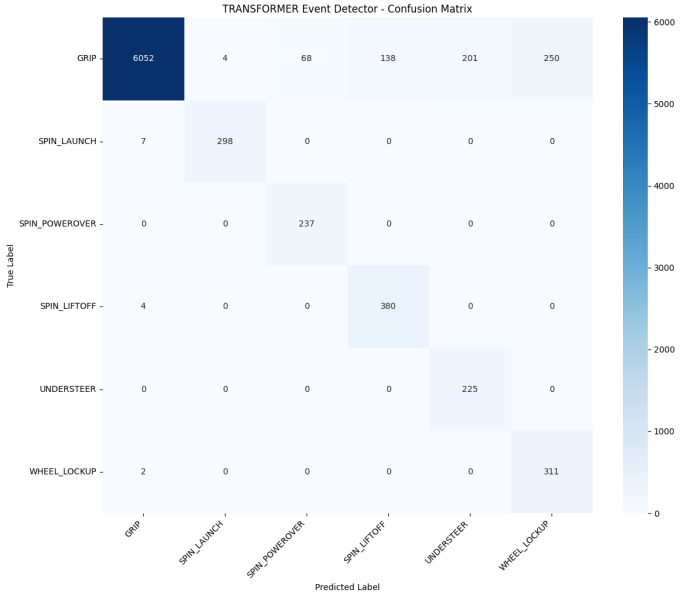

Figure C.4: Confusion matrix for the Transformer model (controls omitted).

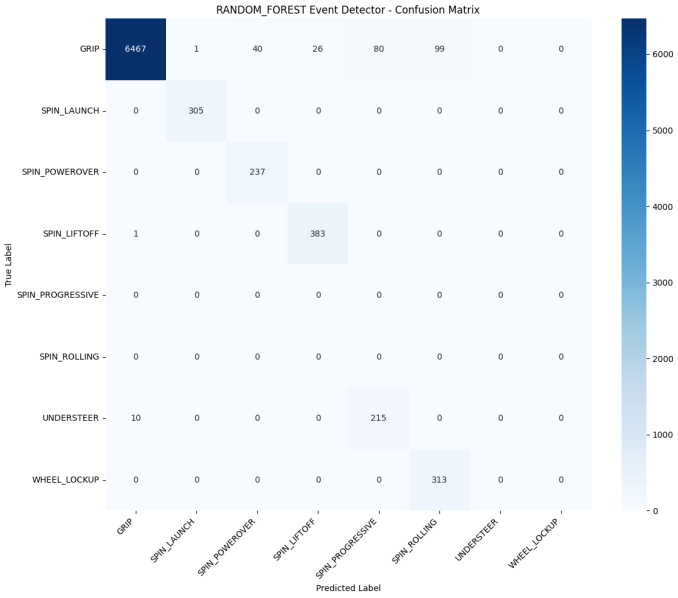

Figure C.5: Confusion matrix for the Random Forest model (controls omitted).

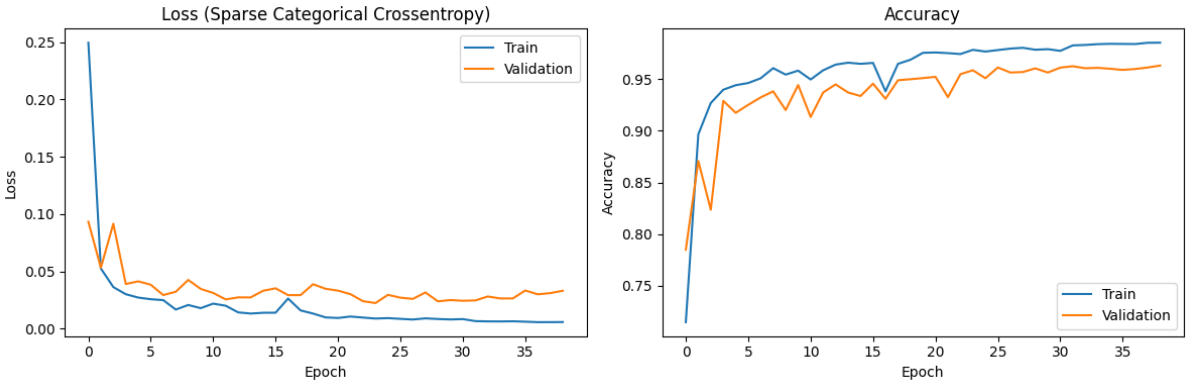

Figure C.6: Training history for the GRU model (controls omitted).

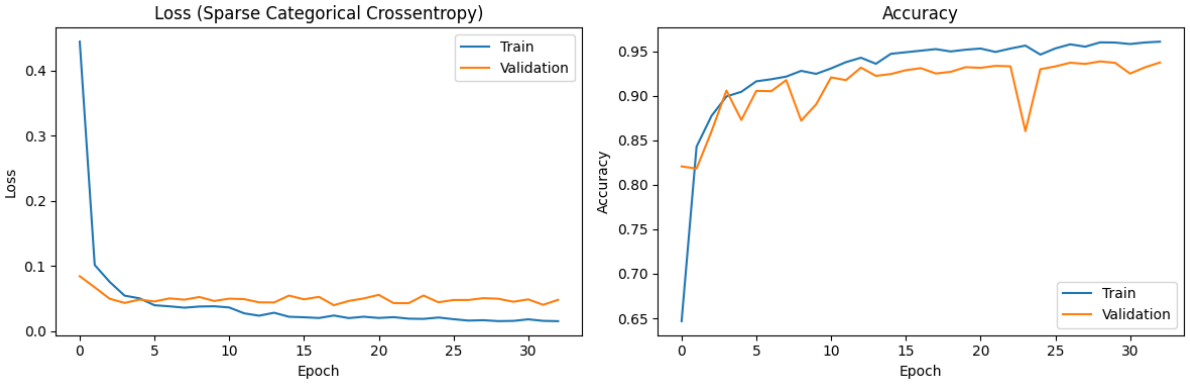

Figure C.7: Training history for the CNN model (controls omitted).

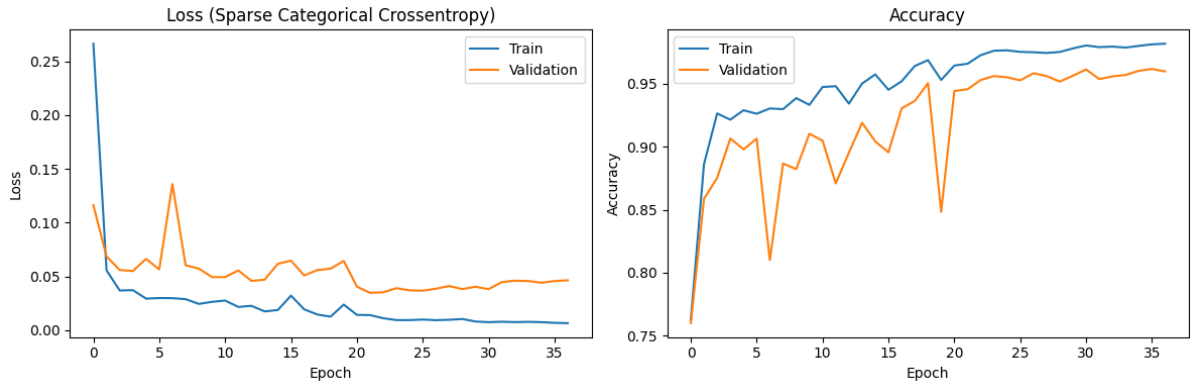

Figure C.8: Training history for the LSTM model (controls omitted).

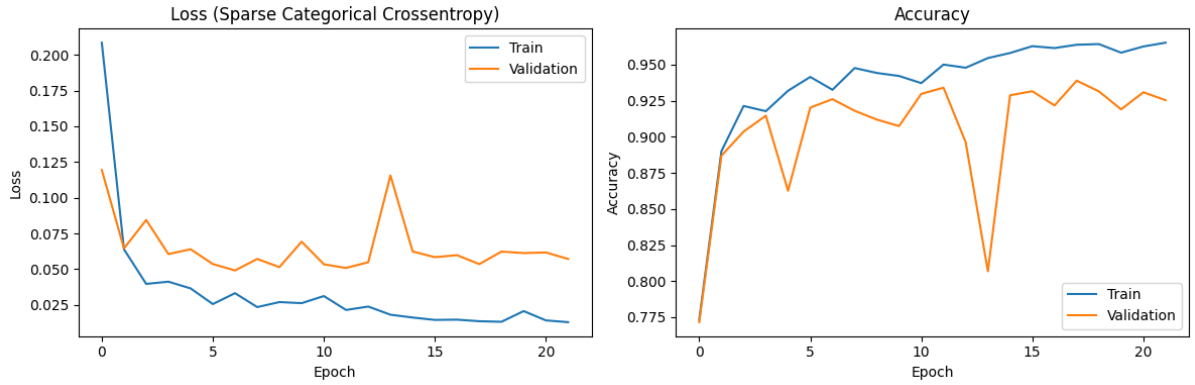

Figure C.9: Training history for the Transformer model (controls omitted).

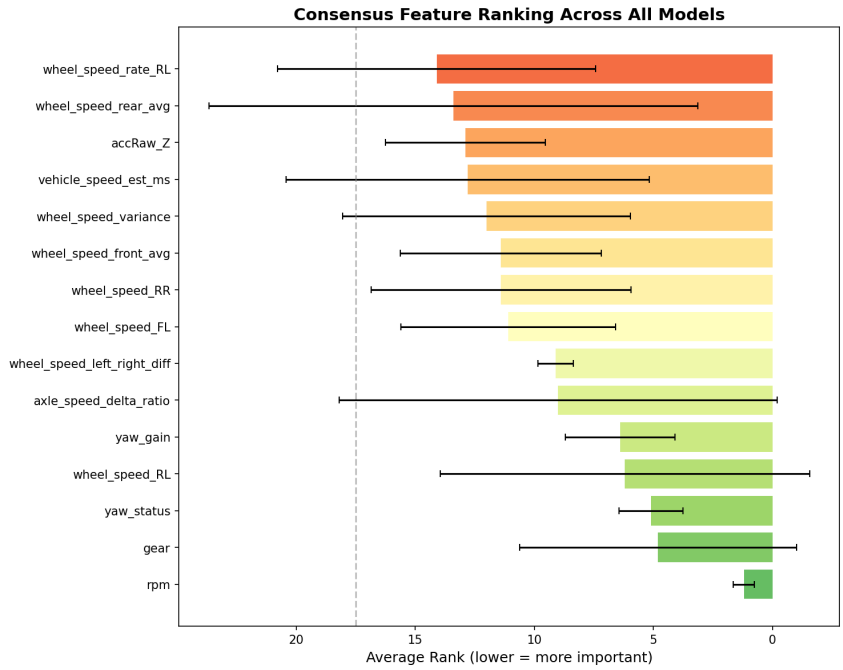

Figure C.10: Feature importance for all models (controls omitted).