Master Thesis Planning Report

Tokenization as a Neural Compression Strategy in Automotive Embedded Systems

Tim Boleslawsky, gusbolesti@student.gu.se

Emrik Dunvald, gusdunvem@student.gu.se

February 2026

Suggested Supervisor at CSE: Yinan Yu

Suggested Supervisor at Company: Dhasarathy Parthasarathy

1 Background

The In-Vehicle Embedded System: An in-vehicle embedded system is a specialized computer system integrated within a vehicle to perform dedicated functions, often in real-time, and is essential for controlling, monitoring, and enhancing various automotive operations. These systems typically consist of both hardware and software components, such as electronic control units (ECUs), sensors, actuators, and communication interfaces, which are responsible for tasks like engine management, safety features, infotainment, and advanced driver assistance systems [**??**].

In-Vehicle Networks and Event-Triggered Logging: Modern vehicles may contain dozens or even hundreds of these embedded systems, interconnected through in-vehicle networks (e.g., CAN, LIN, FlexRay, Ethernet), enabling efficient communication and coordination among different vehicle subsystems [**???**]. Event-triggered logging and diagnostic frameworks, which record data only when anomalies or threshold crossings occur, are often adopted to reduce data transmission and avoid bus saturation in complex systems, such as the in-vehicle embedded system. However, this selective approach can reduce holistic visibility of system health, as it may miss subtle degradation patterns or early warning signs that do not cross predefined thresholds. This complicates the detection of emerging faults and comprehensive condition monitoring [**???**]. Additionally, the need to carefully tune event thresholds and diagnostic criteria introduces maintenance challenges, as improper settings can lead to missed events or excessive false positives, further complicating system upkeep and reliability [**??**].

Traditional Compression: Compression, as originated in information theory by **?**, is the process of encoding information using fewer bits than the original representation. Compression techniques can be broadly categorized into lossless and lossy methods. Lossless compression is based on two principles: distribution modeling, sometimes called entropy modeling, and entropy coding. Entropy modeling involves creating a probabilistic representation of the data, while entropy coding assigns shorter codes to more frequent symbols based on their probabilities, thereby minimizing the average code length. Lossy compression allows for some loss of information in exchange for higher compression ratios. This is typically achieved through techniques such as transform coding and quantization [**?**]. For the purpose of this project, the focus will be on lossy compression as this is more suitable for common downstream ML tasks where some loss of fidelity is acceptable as long as the relevant information for the task is preserved.

Traditional compression methods, based on these information theory principles, often fall short in automotive applications, especially as a precursor for downstream ML tasks. For video/image compression, traditional methods like JPEG or MP3 are optimized for human perception (e.g., visual quality) rather than ML tasks or efficient downstream data use [**?**]. For time series data, algorithmic approaches like CHIMP or Gorilla depend on manually chosen parameters like window size and are sensitive to data characteristics such as entropy and signal variability. This limits their effectiveness in capturing the nuances required for accurate ML model performance in automotive contexts [**?**]. These algorithmic approaches were investigated by **?** in a previous Master's thesis project. This work builds upon this thesis by exploring an alternative approach to compressing vehicle telemetry data.

Rate-Utility Trade-off and Related Research: Constructing downstream ML models for automotive systems, or in fact Internet-of-Things (IoT) systems in general, is a constant trade-off between handling large quantities of data and maximizing model performance. Traditional compression techniques can reduce data volume, but often at the cost of losing critical information necessary for accurate ML tasks such as predictive maintenance, anomaly detection, and fleet analytics. The impact of this trade-off is well-documented in the literature. **?**, for example, study the impact of lossy compression techniques on time series forecasting tasks and observe a constant trade-off between compression ratio and forecasting accuracy.

Existing research approaches these challenges from three different angles: utility-aware adaptive telemetry, neural compression, and task-aware compression.

- First, utility-aware adaptive telemetry methods aim to employ policy learning methods to dynamically adjust telemetry parameters to reduce maintenance costs while preserving data utility for downstream tasks. Although this approach is still emerging, recent research has demonstrated promising results [**?**].

- Second, neural compression techniques learn data representations optimized for both compression efficiency and ML task performance. This research is heavily inspired by deep generative models like GANs, VAEs, and autoregressive models, but focuses on compressing the data, instead of generating realistic data samples [**?**]. Neural compression techniques extend the introduced lossy compression principles in two key ways. First, they offer an alternative to traditional distribution modeling by leveraging deep neural networks to learn complex data distributions directly from the data, capturing intricate patterns and dependencies that traditional statistical models may miss. Second, they substitute traditional approaches to transform coding and quantization with learned representations [**?**]. Studies as early as 2019 have shown that neural compression methods can outperform traditional compression techniques for image and video data, especially at low bitrates [**?**]. The same has been shown for time series data [**??**].
- Lastly, task-aware compression techniques focus on optimizing compression algorithms to retain information that is most relevant for specific tasks [**?**]. This idea has shown promise in handling time-series data more efficiently in IoT systems. **?** and **?** for example explore task-aware compression algorithms that adaptively prioritize data features based on their relevance to downstream tasks, demonstrating improved performance in resource-constrained environments.

2 Aim

The aim of this project will be to investigate the use of neural compression techniques, specifically tokenization-based approaches, for compressing automotive time series data. The goal is to develop a compression framework that effectively balances the trade-off between compression ratio and utility for downstream machine learning (ML) tasks, such as predictive maintenance and anomaly detection.

The motivation behind using tokenization as a neural compression strategy for automotive time series data is to produce discrete latent representations that simplify the entropy modeling task within the compression pipeline. By constraining the data to a finite set of tokens, the complexity of modeling the underlying data distribution is reduced, enabling the use of lightweight entropy models that are computationally efficient. This is particularly advantageous in automotive applications where computational resources are limited, and real-time processing is often required.

The goal will be to design, implement and evaluate a tokenization-based neural compression framework tailored for time-series data. Evaluation will be conducted by comparing relevant parameters between the proposed framework and traditional compression methods.

3 Problem Formulation

4 Limitations

5 Methodology

Within this section, we will discuss the methodology employed in our project to achieve the outlined aim. Specifically, we will discuss how the architecture of the baseline established compression model and the proposed tokenization-based compression framework are designed and implemented. We additionally discuss the experiment setup we will use to compare these two approaches.

5.1 Implementation Details

Baseline Model: First, we want to outline the architecture of the established compression method we will use as a baseline. The paper we will use as an inspiration for the autoencoder baseline model is "Temporal Convolutional Networks for Real-Time Anomaly Detection in Time Series" by Malhotra et al. (2019). It is important to note, that the paper introduces a sophisticated framework consisting of two models, the TCN-RNN model and the TCN-ARNN model, as well as a model selection mechanism. For simplicity sake, we

will only focus on one of these models, the TCN-RNN model. The TCN-RNN model described in the paper is a representative of state-of-the-art continuous-latent neural time-series compression methods. Compared to vanilla autoencoders, the models architecture is explicitly designed for long temporal dependencies and has demonstrated stronger rate-distortion performance. This allows us to evaluate our tokenization-based approach against a competitive neural compressor rather than a toy model. Another point to note about the TCN-RNN model, is its costly architectural design, namely deep temporal convolutional stacks with large receptive fields, recurrent (often sequential) decoding, and dense continuous latent representations. These components typically lead to increased FLOPs, higher activation memory, and longer inference latency compared to discrete, token-based compression methods.

On a high level, the TCN-RNN architecture consists of the following components:

- Input handling and preprocessing
- TCN encoder
- RNN bottleneck
- RNN decoder
- TCN decoder

For the input handling and preprocessing it is important that the time-series data is segmented into fixed-length windows. These windows are treated as inputs to the model. Overlapping windows can be used to increase the effective training data size and improve reconstruction quality at window boundaries. Additionally, each channel is normalized independently using statistics computed on the training set.

The TCN encoder consists of a stack of Temporal Convolutional Network (TCN) blocks with increasing dilation factors. Each TCN block contains: A 1D convolution operating along the temporal dimension, normalization (BatchNorm or LayerNorm), a nonlinear activation function (ReLU or LeakyReLU), optional dropout for regularization, and a residual connection to stabilize training. The outcome is a continuous latent representation whose size is controlled by the downsampling factor and channel width.

The encoder output is processed by a recurrent neural network, specifically a gated recurrent unit (GRU), the RNN bottleneck. The RNN captures longer-range temporal dependencies that are not easily modeled by convolutions alone. At this stage, we produce a sequence of hidden states that serves as the core compressed signal. The hidden state dimensionality and number of layers define the strength of the bottleneck. The compression is achieved implicitly by restricting temporal resolution and latent dimensionality rather than by discretization. Afterwards, the RNN decoder reconstructs a latent sequence from the bottleneck representation. The decoder mirrors the bottleneck RNN in depth and hidden size.

Finally, a mirrored TCN stack upsamples the latent sequence back to the original temporal resolution. Transposed convolutions or interpolation-based upsampling are used. A final linear projection maps features back to the original channel dimension. The overall training objective is the reconstruction loss (e.g., MSE or MAE) between input and reconstructed signal.

Tokenization-Based Compression Framework: Next, we outline the architecture of our proposed tokenization-based compression framework. The architecture is inspired by recent advances in neural compression for speech and audio data, namely the Wavtokenizer paper. The key idea is to replace continuous latent representations with discrete tokens drawn from a learned codebook.

We again want to first outline the high-level components of the architecture:

- Input handling and preprocessing
- Lightweight encoder
- Vector quantization / tokenization
- Decoder head (for reconstruction) or downstream head (for task-specific loss)

The first component is (and should be) identical to the TCN-RNN baseline model. This ensures that any performance differences can be attributed to the compression method rather than preprocessing variations.

The encoder layer functionally has the same goal s the TCN encoder used in the baseline model, but has a different architecture. The encoder is designed to be lightweight, avoiding heavy temporal convolutions or recurrent layers. Instead, a shallow neural network (e.g., small CNN or MLP applied per timestep) maps the input signal to intermediate embeddings. The encoder implements the learned transform stage of a lossy compression pipeline. Its purpose is not to predict labels or reconstruct the input per se, but to reshape the data into a representation that can be efficiently discretized. Specifically, the encoder learns a mapping that removes redundancy and correlations present in the raw time series, concentrates important information into a low-dimensional, stable representation, and aligns the geometry of the representation space with the constraints of quantization. The objective is usually joined with the quantization and decoder stages.

In contrast to autoencoder-based methods, where the encoder is free to produce high-entropy continuous latents, the encoder here is explicitly shaped to produce quantization-friendly representations that support discrete tokenization and entropy coding.

The next component is the vector quantization / tokenization stage. Here, intermediate embeddings are discretized using a learned codebook. The codebook very simply looks like this: codebook = nn.Parameter(torch.randn(K, D)), with K = number of tokens and D = embedding dimension. Each embedding vector is replaced by the index of its nearest codebook entry. This step introduces the only explicit source of information loss. The output is a sequence of discrete token IDs drawn from a finite vocabulary.

Lastly, we implement a head to enable loss calculation and backpropagation. Depending on the use case, this head can be a decoder for reconstruction tasks or a downstream head for task-specific losses (e.g., classification or regression). The head architecture is flexible and can be adapted to the specific application. The calculated loss is then used to jointly optimize the encoder, codebook, and head parameters.

5.2 Experiment Setup

To compare and evaluate the two methods outlined above, we will conduct a series of experiments. In this subsection we outline how we ensure a fair comparrison, what metrics we will use, and what data we will use for the experiments.

Comparrison and Metrics: There are three dimensions along which we want to compare the two compression methods. First, we want to evaluate the *compression performance* of both methods. This includes the metrics compression rate and reconstruction error. These metrics align with the evaluation standards in the neural compression literature. We will measure these metrics by training the two models on a reconstruction task and evaluating the reconstruction quality and effective compression rate. Second, we want to evalute the *downstream task performance* when using compressed representations from both methods. This includes metrics such as accuracy for classification tasks or mean squared error for regression tasks. This evaluation reflects the practical utility of the compressed data for real-world applications. Third, we want to compare the *model efficiency* of both methods. This includes metrics such as parameter count, computational cost (FLOPs), inference latency, and memory footprint. These metrics are important for assessing the feasibility of deploying the compression methods in resource-constrained environments.

An important note to make regarding comparability of these two metrics, especially regarging the downstream task performance, is that the two methods produce different types of compressed representations. The TCN-RNN baseline produces a contionuous latent representation and the tokenization-based method produces discrete tokens. To ensure a fair comparison, we will need to design a lightweight unintrusive adapter for each representation type that maps them into a common format suitable for the downstream model. The adapters should have minimal capacity to avoid introducing bias. This way, we can isolate the effect of the compression method itself on downstream performance. The envisioned design for the adapters is as follows:

- The adapter for the continuous latent representation will take the sequence of real-valued vectors output by the TCN-RNN model, average them over time to produce a single vector, and then pass this vector through a linear layer to match the input size of the downstream model.

- The adapter for the token-based representation will take the sequence of token IDs output by the tokenization model, map each token ID to a small vector using an embedding lookup, average these embeddings over time to produce a single vector, and then pass this vector through a linear layer to match the input size of the downstream model.

Data:

6 Risk Analysis

7 Timeline

We want to roughly divide the project into four stages, each with specific goals and deadlines. First, we will set up a working environment, and put time into understanding the problem and plan for the project. This would be **Stage 1: Setup and Preparation**. Within this stage we have the following sub-goals:

- Planning the project with appropriate subtasks and a timeline. Then writing the project plan.
- Creating a working environment. This include setting up necessary hardware (at Volvo), git repositories, and coding environments.
- Defining and Training a downstream ML model on uncompressed data that will serve as a benchmark for evaluating the performance of different compression methods. Right now, we aim for one regression and one classification task based on automotive telemetry data.
- Implementing a baseline model based on an established neural compression frameworks such as `https://www.sciencedirect.com/science/article/pii/S1568494623008153`. The planned architecture is discussed in section 5.

We envision, that this stage will take roughly five weeks and set a preliminary deadline for the 20th of February.

The second stage, **Stage 2: Development**, will focus on developing the core components of the proposed compression approach leveraging tokenization. This includes:

- Develop a compression framework that leverages tokenization. This includes implementing an encoder, a quantization method using a codebook, and a decoder as discussed in section 5.
- Conducting some first initial experiments to validate the implementation.
- Writing the halftime report.
- Start writing on the final thesis report. At this stage we aim to have the theory part of the report done.

The second stage should take around five weeks as well, with a deadline for the 27th of March.

After that, we will start **Stage 3: Experiments**, which will focus on evaluating the performance of the proposed compression method against the baseline model. This stage will include:

- Compressing the data using both the baseline compression method and the proposed tokenization approach.
- Training the downstream model on the compressed data using both compression methods.
- Evaluating the experiments by capturing the defined metrics for both compression methods.

This stage is expected to take three weeks, with a deadline for the 17th of April.

Finally, we will enter **Stage 4: Finalizing**, which will involve:

- Conducting an ablation study of the proposed tokenization compression framework.
- Writing the methodology, results, discussion, and conclusion part for the final report.

This stage is expected to take four weeks. We plan to have a first draft of the final report ready with a deadline for the 15th of May.