CS 6235 Final Report: Machine Learning for Detection of Intrusions of CAN Communications

Phineas Giegengack

December 2, 2025

Contents

1 Introduction

1.1 CAN Communication

The Controller Area Network (CAN) protocol is a specification which describes how many electronic control units (ECUs) broadcast data over a shared bus. The CAN standard was developed by Bosch in the 1980s and later standardized as ISO 11898 [1] for use in automobiles to reduce the complexity of the electrical design of vehicles with an ever-growing array of electronic sensors and actuators. CAN was an answer to difficulties that arose from designing point-to-point connections between every ECU that needed to communicate in a car.

The CAN protocol is a 'bus' architecture. Figure 1.1 presents a high-level overview of the CAN bus in action. 3 ECUs are shown broadcasting messages over the bus. All ECUs may receive any message, but they will accept messages based on masks or filters configured by the system designer. A simplified view of the construction of the CAN message is presented in Figure 1.2. The CAN ID is an 11-bit header for the message which corresponds to that message's priority. CAN messages with lower CAN IDs will take precedence over those with higher CAN IDs. The CAN message can have a data payload of between 0 and 8 bytes. In this project, finer-grain understanding of the CAN packet (which is often vehicle and manufacturer dependent) was not considered, but suffice to say that in certain contexts the CAN data field could be broken into multiple sub-fields which have distinct meanings and that there are additional house-keeping bits and message formats which are not typically considered in intrusion-detection. We will see in Section 4 that this simplified view of CAN messages is a typical one in the intrusion detection space and is sufficient to develop a high-quality machine learning model for detecting anomalies on the CAN bus.

CAN does not by default have authentication, since it was designed for closed systems in which all ECUs are trusted, leaving it open to a variety of attacks, including Denial of Service (DoS), Fuzzing, and Spoofing. Without authentication, if an intruder gained access to the CAN bus, they could impersonate a legitimate ECU on the bus and send and receive arbitrarily.

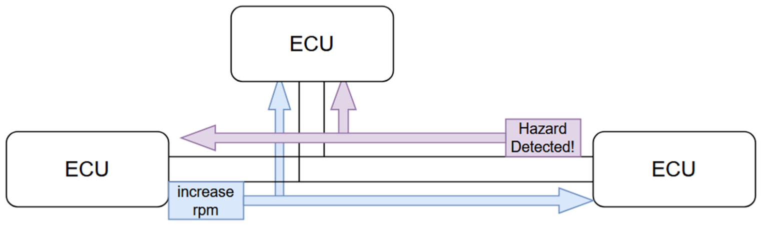


Figure 1.1: High-level overview of CAN bus operation.

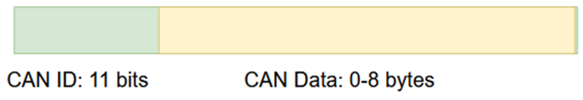


Figure 1.2: Simplified view of CAN message.

This interaction is shown in Figure 1.3.

In modern vehicles, and especially in autonomous vehicles, safety-critical systems are connected via CAN, so the ability to detect intrusions onto this network is of utmost importance.

The assumptions of the automakers who designed the CAN protocol many years ago– smaller number of trusted ECUs on the CAN bus, all contained within the vehicle with no outside contact except by service ports– has changed with the proliferation of wireless technology and the continuous explosion in the number of computing devices integrated into our vehicles. It has become a serious concern that hackers can gain unfettered access to the CAN bus wirelessly through the weakest link in the vehicle's ECU lineup.

In [2], the CAN bus attack vector is investigated to discover the range of actions a hacker could take if they simply gain undetected access to the CAN bus. The researchers were able to disable all CAN communication, manipulate and damage the engine, flash code to the vehicle's telemetry unit, and more. Clearly, a successful attack on the CAN bus can yield frightening results for the vehicle and driver.

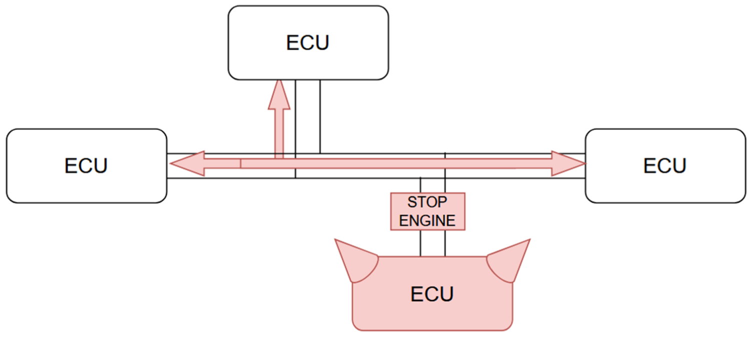


Figure 1.3: An unauthorized ECU gains access to the CAN bus and broadcasts its malicious messages.

1.2 Autoencoders for One-Class Classification

One-Class Classification (OCC) seeks to solve problems where sufficient training data exists only for one class (the known class), but the desire is to detect anomalous data falling outside this class (the unknown class). In the context of CAN bus intrusion detection, the known class is attack-free traffic, while the unknown class is traffic containing packets injected by a malicious ECU.

An autoencoder is a neural network consisting of an encoder that compresses input data into a latent representation and a decoder that reconstructs the original input from this representation, as shown in Figure 1.4.

For OCC, the autoencoder is trained to reconstruct only data from the known class. The hypothesis is that inputs from the unknown class will produce significantly higher reconstruction error (mean-squared error between original input and reconstruction) than inputs from the known class, which would enable the distinction of anomalies.

Liao et al. [3] identify a potential limitation of autoencoders in OCC, they tend to generalize well to unseen data, which is detrimental to the OCC use case, since it is desired that the model would have high reconstruction error for unseen/anomalous data. They propose a Batch-Wise Feature Weighting (BFW) which identifies and amplifies the aspects of the latent dimension which are most important in reconstructing the input vector. This project seeks to determine the effectiveness of this strategy in the context of CAN bus intrusion detection.

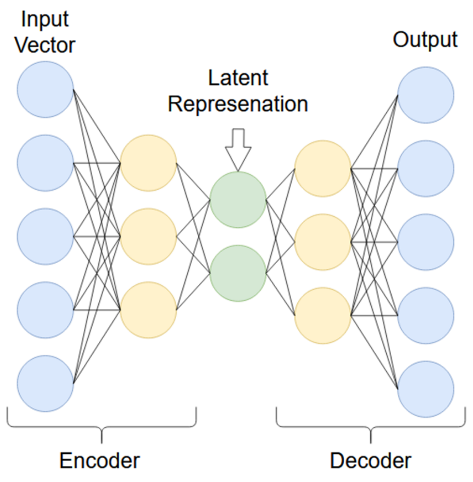


Figure 1.4: Autoencoder architecture with encoder, latent representation, and decoder

2 Motivation

unauthorized access to the CAN bus has been shown to have potentially disastrous effects on the vehicle and driver [2]. While autoencoders have been applied widely to CAN bus intrusion detection due to their ability to model normal traffic patterns, in Liao et al, the performance of an autoencoder for OCC was improved with implementation of a Batch-Wise Feature Weighting (BFW) [3]. However, BFW has not been evaluated for CAN bus intrusion detection. It is important to further the research in the area of CAN bus intrusion detection with the most recent advances in ML methodologies.

The goal of this project is to create a machine learning pipeline which uses an autoencoder to detect intrusions on the CAN bus. The effectiveness of using a Batch-Wise Feature Weighting (BFW) as described in [3] will be assessed for this use case, which has not been attempted in previous work. Other hyperparameters including learning rate, batch size, number of epochs and size of input vector will be compared.

The hypothesis is that the BFW will improve autoencoder performance for distinguishing attack-free and attack traffic on the CAN bus and that by tuning hyperparameters and experimenting with various feature extraction methods, a model can be produced which will detect most attacks in under a second of the attack beginning, allowing vehicle manufacturers to build in defensive actions that will quickly take effect when an intrusion on the CAN bus occurs.

An analysis of the feasibility of porting the model to embedded hardware will be done to determine the direction that future work on this project will take.

3 Related Work

A variety of methods to detect attacks/intrusions on the CAN bus using machine learning have been proposed.

In [4], a traditional neural net and multi-layer perceptron are compared for their performance in detecting CAN bus intrusions. They are trained and tested on a static dataset of CAN bus message traces. No effort is made to optimize the models for implementation on an embedded device, but the models achieve respectable performance.

Researchers in [5] made use of a GRU model to achieve high performance with a lightweight model intended to be deployed on embedded hardware. They also propose a system architecture for deploying, updating and training the model over-the-air (OTA) which would be desired for real-world applications to keep model performance up to date as CAN bus attacks evolve. [6] and [7] make use of support vector machines (SVMs) to do one-class classification which enables models to be trained only on 'good' CAN bus data and have the models infer when anomalous behavior is taking place without needing training data for those scenarios. This is appealing since attack methods are likely unknown at the time of training, and having one-class models would allow detection of novel attacks before they impact any vehicle.

[8] is one of the more recent contributions to the space and proposes another type of model, the TPE-LightGBM algorithm. They achieve good performance and the model is lightweight, although they do not deploy it on embedded hardware.

Three papers are especially of interest for this project. [3] Outlines the use of autoencoders as One Class Classifiers (OCC), and the addition of a Batch-Wise Feature Weighting as a means of combating the autoencoders inherent skill at generalizing which reduces its accuracy in OCC applications. [9] Attempts to use an autoencoder for CAN bus intrusion detection, but with low accuracy and without investigating the impact of adding a BFW. In [10], researchers describe the methods used to accumulate the can-train-and-test dataset, which will be used in this project to train and test the model.

This project goes beyond previous work in a few ways. First, the work in [3] to define a BFW for use in autoencoders has not been applied to CAN bus intrusion detection. Further, this project will assess the effect the size of the input window, that is how many CAN messages comprise a single input vector, has on the performance of the model for detection of CAN bus intrusions.

4 Dataset and Statistics

The data used in this project was select from can-train-and-test [10]. Researchers from the Technical University of Denmark collected CAN bus traffic from 4 vehicles under normal operation and under several attack scenarios. They created labeled datasets from the CAN traffic where messages generated by the malicious ECU are labeled as 'attack' and all others are labeled as 'normal' (or 0 for normal, 1 for attack) as can be seen in Figure 4.1. Each CAN message is accompanied also by a timestamp.

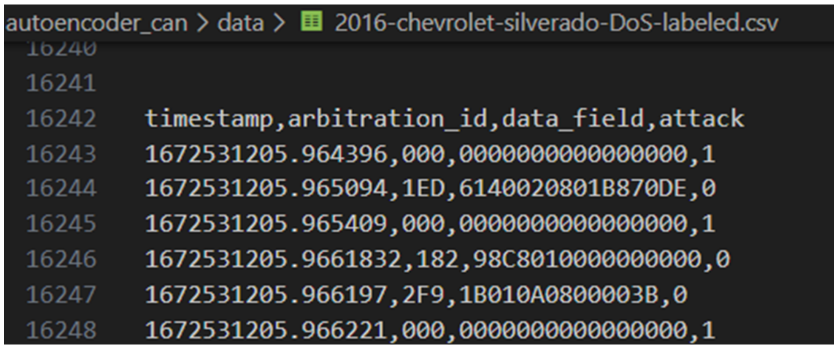
For this project, the model was trained only on attack-free data captured on a 2016 Chevrolet Silverado. The model's accuracy was assessed on captures taken during 8 attack scenarios including one 'combined' attack scenario that includes examples of several of the 7 other attacks. A comparison of the sparsity of the datasets (how many attack messages appear as a percentage of the total number of messages in the dataset) appears in Table 4.1.

Table 4.1: Comparison of sparsity of datasets used in training and testing autoencoder.

dataset_type	**total_messages**	**normal_messages**	**attack_messages**	**attack_percentage**	**duration_seconds**	**messages_per_second**	**unique_can_ids**
attack-free	1254632	1254632	0	0	487.7283158	2572.399345	98
combined	920226	918266	1960	0.212991157	357.0358989	2577.404689	98
dos	252707	208924	43783	17.32559842	81.25668597	3109.984083	99
fuzzy	1059035	1008513	50522	4.770569433	392.2755001	2699.722516	2048
gear	1257602	1254632	2970	0.236163747	487.728138	2578.489741	98
interval	950657	943319	7338	0.771887232	366.811969	2591.673883	98
rpm	524212	522688	1524	0.290722074	203.2086239	2579.673982	98
speed	1255235	1254632	603	0.048038813	487.727381	2573.640622	98
standstill	524245	522688	1557	0.296998541	203.2089219	2579.832593	98

The 'attack-free' dataset was used in training and the other 8 datasets were used to evaluate the model.

There are a few things to notice about these datasets: first, they all have a similar number of messages per second, and the captures all include several hundred thousand to over a million messages each. Also, the testing datasets for the attack scenarios are quite sparse, with most including under one percent attack messages. It will be shown in the coming sections that this


autoencoder_can > data > 2016-chevrolet-silverado-DoS-labeled.csv

```
timestamp,arbitration_id,data_field,attack
1672531205.964396,000,0000000000000000,1
1672531205.965094,1ED,6140020801B870DE,0
1672531205.965409,000,0000000000000000,1
1672531205.9661832,182,98C8010000000000,0
1672531205.966197,2F9,1B010A0800003B,0
1672531205.966221,000,0000000000000000,1
```

Figure 4.1: Example dataset featuring messages which are part of a DoS attack. DoS messages have arbitration ID 0 (highest priority) and are labeled '1' for attack.

became a major limitation in evaluating the model and achieving desirable F1 scores.

All datasets have the same number of unique CAN IDs, save for the DoS attack which has the additional CAN ID 0, and the fuzzing attack which is characterized by the spamming of random CAN IDs, and thus has many more unique IDs.

5 Implementation

The code for this project was written in python, with extensive use made of the pytorch library for configuring, training, and running the autoencoder. The matplotlib library was used in creating plots and graphs, while numpy and scipy were used at various points for statistics and manipulation of the datasets.

5.1 Data Preprocessing

5.1.1 Dataset Loading

The datasets mentioned in Section 4 were downloaded as .csv files from the can-train-and-test bitbucket before their data was loaded into numpy arrays to prepare for feature extraction.

5.1.2 Feature Extraction

Two feature extraction strategies were compared in the early stages of the project, although both were rather naive in their approach.

The first involved taking groups or windows of N messages and setting a new timestamp for the first message in the group as 0. The timestamps for the subsequent messages in the group are set to the difference between their timestamp and the original timestamp for the first message in the group. Input vectors are constructed by appending the timestamps, CAN IDs and individual bytes of the CAN data field into a vector as floats.

The alternate feature extraction methodology simply involved ignoring the CAN data field. For this method, only the timestamp and CAN arbitration ID of each message was passed into the input vector. This method yielded better performance early on and was thus the chosen one for the analysis that is to come.

The superior performance of the model when ignoring the CAN data field suggests that the actual data bytes of CAN messages are less indicative of an attack versus normal traffic than the time between messages and their CAN

IDs. Furthermore, removing the CAN data field allows the input vector to be smaller, the model to be more efficient, and training the model to converge more rapidly.

5.1.3 Normalization

The features of the input vector were each normalized using z-score normalization through the scikit-learn python package. This causes all features to have a mean of zero and standard deviation of 1 so that each feature contributes equally to the error calculation. Otherwise, the timestamp in microseconds could dominate compared to the CAN IDs which are 11-bit values, or the CAN Data bytes, for example.

5.2 Model Architecture

The autoencoder model was implemented in PyTorch and consists of three main components: an encoder network that compresses input vectors into a lower-dimensional latent representation, an optional Batch-Wise Feature Weighting (BFW) module that amplifies important latent features, and a decoder network that reconstructs the original input. The architecture adapts dynamically based on the input vector dimension to maintain appropriate network capacity.

The input vector dimension is a function of the window size and feature extraction strategy described in Section 5.1.2, and thus needed to be dynamic to test the effects of different window sizes and feature extraction strategies.

5.2.1 Base Autoencoder Architecture

Given an input vector $\mathbf{x} \in R^d$, the encoder transforms it through a sequence of fully connected layers with ReLU activation functions, reducing dimensionality until a latent representation $\mathbf{z} \in R^{d_z}$ is reached, where $d_z = 16$ is the latent dimension. The encoder network is dynamic with input dimension d:

- For $d \geq 500$: Encoder layers are $[d, 128, 64, 32, 16]$
- For $200 \leq d < 500$: Encoder layers are $[d, 64, 32, 16]$

- For $d < 200$: Encoder layers are $[d, 32, 16]$

The encoder layers were programmed adaptively to ensure the network had the representational capacity for larger input vectors but didn't over-parameterize for smaller input vectors, which could lead to overfitting. Every hidden layer applies a linear transformation followed by a ReLU activation:

$$\mathbf{h}_i = \text{ReLU}(\mathbf{W}_i \mathbf{h}_{i-1} + \mathbf{b}_i)$$

where $\mathbf{h}_0 = \mathbf{x}$ is the input and $\mathbf{h}_n = \mathbf{z}$ is the latent representation. The decoder is the exact mirror of the encoder, that is:

- For $d \geq 500$: Decoder layers are $[16, 32, 64, 128, d]$
- For $200 \leq d < 500$: Decoder layers are $[16, 32, 64, d]$
- For $d < 200$: Decoder layers are $[16, 32, d]$

The complete forward pass without BFW can be expressed as:

$$\hat{\mathbf{x}} = \text{Decoder}(\text{Encoder}(\mathbf{x}))$$

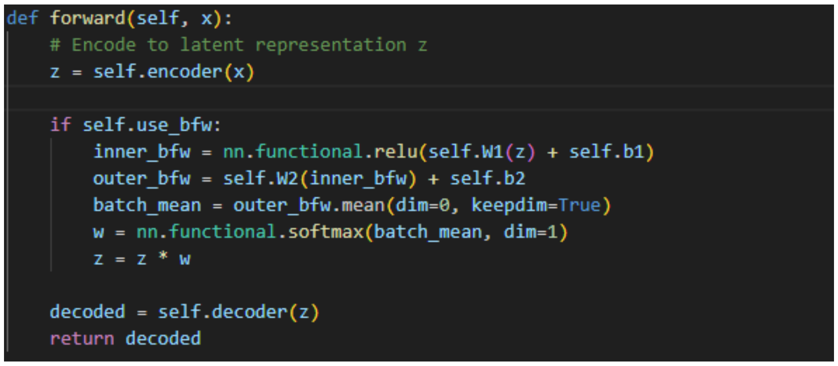
where $\hat{\mathbf{x}}$ is the reconstructed input. The model is trained to minimize the mean-squared error between the input and reconstruction:

$$\mathcal{L} = \frac{1}{d} \|\mathbf{x} - \hat{\mathbf{x}}\|^2$$

5.2.2 BFW Implementation

The Batch-Wise Feature Weighting module is inspired by Liao et al [3] and is intended to reduce the autoencoder's inherent ability to generalize to unfamiliar data, which would be detrimental to its usefulness in OCC. The BFW is expected to reduce its generalization capabilities by learning the aspects of the latent represenation most important for reconstruction of the input signal, which is the attack-free traffic.

The BFW is a two-layer neural network which operates on the latent representation $\mathbf{z}$. It computes a weight vector of the same dimension as the latent representation, and performs an element-wise multiplication of the latent representation during the forward pass:



```python
def forward(self, x):
    # Encode to latent representation z
    z = self.encoder(x)

    if self.use_bfw:
        inner_bfw = nn.functional.relu(self.W1(z) + self.b1)
        outer_bfw = self.W2(inner_bfw) + self.b2
        batch_mean = outer_bfw.mean(dim=0, keepdim=True)
        w = nn.functional.softmax(batch_mean, dim=1)
        z = z * w

    decoded = self.decoder(z)
    return decoded
```

Figure 5.1: Python code to implement BFW as described in [3]. Note that the BFW can be turned on and off dynamically.

$$\mathbf{h}_{BFW} = \text{ReLU}(\mathbf{W}_1\mathbf{z} + \mathbf{b}_1) \tag{5.1}$$

$$\mathbf{w}_{batch} = \mathbf{W}_2\mathbf{h}_{BFW} + \mathbf{b}_2 \tag{5.2}$$

$$\mathbf{w} = \text{softmax}(\text{mean}_{\text{batch}}(\mathbf{w}_{batch})) \tag{5.3}$$

$$\mathbf{z}' = \mathbf{z} \odot \mathbf{w} \tag{5.4}$$

The actual implementation in python can be seen in Figure 5.1
where $\mathbf{W}_1$, $\mathbf{W}_2 \in R^{16\times16}$, and $\odot$ is element-wise multiplication.

Training of the model should teach the BFW to increase weighting of features of the latent representation which are most relevant to normal, attack-free CAN traffic. Ideally, this will make the reconstruction error of anomalous CAN traffic much higher since it is presumed that the features of the latent dimension most important for reconstructing attack traffic differs from that of attack-free traffic.

The BFW and base autoencoder are trained at once with the same reconstruction errors. Both the autoencoder parameters θ_{AE} and BFW parameters θ_{BFW} are updated during backpropagation:

$$\theta_{AE}, \theta_{BFW} \leftarrow \theta_{AE}, \theta_{BFW} - \eta\nabla_\theta \frac{1}{d}||\mathbf{x} - \hat{\mathbf{x}}||^2$$

where η is the learning rate. The BFW can be turned on and off to compare its effects on model performance for OCC.

5.3 Optimizing Parameters

Selecting which parameters to perturb to get the best performance out of this model for OCC was one of the trickiest aspects of the project. Through the first months of the project, the area under the ROC curve (discussed in-depth in Section 5.5.1) was treated as a north star of sorts. Configurations that were seen to maximize AUC of ROC curve were held constant while others with less clear effects were iteratively tested at various values.

A python script, train_matrix.py was created to automate the process of training the model for a multitude of parameter combinations. The parameters that were experimented with are described in the next sections, but the model architecture described in Section 5.2 was mostly fixed.

5.3.1 Training Hyperparameters

Hyperparameters were iteratively tweaked to improve the convergence time and the reconstruction loss that the model settled on during training with the attack-free CAN traffic. These hyperparameters were learning rate, batch size, and number of epochs.

Higher batch sizes and lower epoch counts tended to be better since the model converged to its best reconstruction loss within tens or hundreds of epochs, and rarely was it observed to reduce reconstruction loss significantly after more than a few hundred epochs. Similarly, batch size did not seem to affect the reconstruction loss that the model ultimately settled on during training, but higher batch sizes allowed training to be completed faster.

The learning rate sweet spot seemed to be around $(15 - 25)e^{-4}$.

5.3.2 Other Variables

Other parameters that were experimented with were the window size N (and proportionally the input dimension) and whether or not BFW was enabled. These were some of the results that were most interesting. The model performance in OCC mostly increased with window size. The findings of the performance of the BFW are discussed in Section 6.

5.4 Inference and Anomaly Detection

After the model was trained to reconstruct attack-free CAN traffic, it was ready for use in CAN bus intrusion detection. Test datasets were passed through the same data processing pipeline as the training data as described in Section 5.1, and the label of the input vector was assigned '1' if any attack messages occurred within that window, and '0' otherwise for an attack-free window.

When the model has been trained, the expected use would be that a reconstruction error threshold is selected which best discriminates attack-free windows from those that contain at least one attack message. When a new input vector is presented to the model, it attempts to reconstruct it and if the reconstruction error falls below the threshold, the prediction will be that this is an attack-free input, otherwise, for input vectors with reconstruction error higher than the threshold, the prediction is that an attack is taking place.

5.5 Evaluation Methodology

The model is evaluated based on its ability to correctly predict whether an input contains an attack or not. The model needs to have a significantly greater reconstruction error for attack inputs than for normal inputs, which will allow selection of a reconstruction error threshold with high true positive rate and low false positive rate.

5.5.1 Receiver-Operator Characteristic Curve

The Receiver-Operator Characteristic (ROC) Curve plots false positive rate (FPR) against true positive rate (TPR) for every selection of error threshold. It is desired that the area under the (ROC) curve is as close to 1.0 as possible, as this would allow for a threshold selection which had TPR = 1, FPR = 0. The AUC of the ROC curve is resilient to class imbalance in the testing dataset, which makes it a preferable metric for this case in which the number of attacks in the test datasets is very low relative to the total number of messages, as discussed in Section 4.

In Figure 5.2 we can see how the distributions of reconstruction errors for inputs that have attacks versus those that are attack free has a direct impact on the ROC curve and the area under it. For the error distributions in Figure 5.2a, an obvious distinction between the reconstruction error for the normal traffic versus the attack traffic exists. One can imagine placing the error threshold near 0.8, which would result in a near complete bisection of the two classes. This results in a phenomenal ROC curve, seen in 5.2b and AUC of nearly 1.

Conversely, the distributions Figure 5.2c are not obviously separated (although the mean reconstruction error of the normal traffic appears to be very slightly lower), and indeed the ROC and AUC in Figure 5.2d are much worse.

5.5.2 Calculation of F1

The F1 score is the harmonic mean of a model's precision and recall, which makes it a good indicator of whether the model is achieving low FNR and FPR, and high TPR and TNR. It is calculated by:

$$F1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{5.5}$$

where precision and recall are given by

$$\text{Precision} = \frac{TP}{TP + FP} \tag{5.6}$$

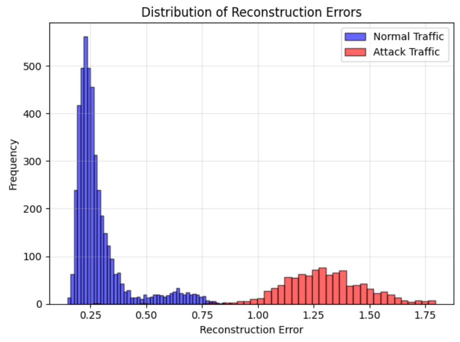
$$\text{Recall} = \frac{TP}{TP + FN} \tag{5.7}$$

F1 score is sensitive to class imbalance, since for a sparse positive class, as is the case for these datasets, false negatives will dramatically decrease the Recall and consequently the F1 score. Indeed it will be shown that even when the model achieves respectable AUC of the ROC curve, the F1 scores are quite poor.

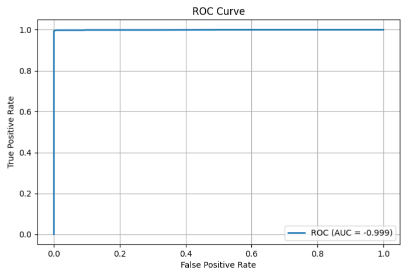
5.6 Challenges and Setbacks with Implementation

5.6.1 Long tail of Normal Traffic Distribution

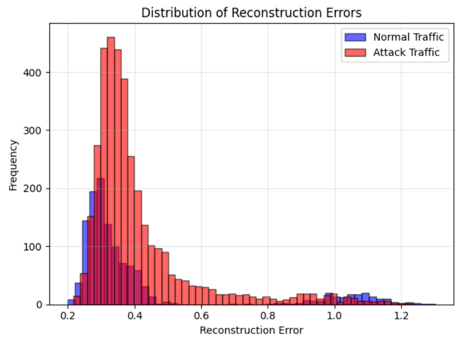
The first issue that is apparent in these methods is in the data itself. From the distribution of the reconstruction errors it is obvious that most of the



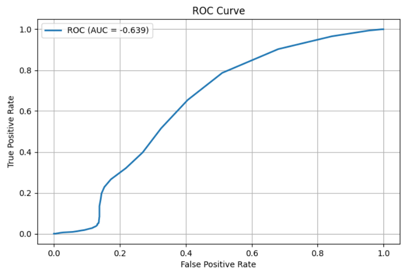
(a) Distribution of reconstruction errors for normal traffic (blue) and attack traffic (red).



(b) ROC curve and AUC very close to 1.0 shows that a threshold can be selected for which TPR $\approx$ 1, FPR $\approx$ 0



(c) Distribution of reconstruction errors for normal traffic (blue) and attack traffic (red). Distribution means do not appear to differ significantly.



(d) Poor AUC of about 0.6 means any selection of threshold will have either low TPR or high FPR or both.

Figure 5.2: Comparison of Error Distributions and Resulting ROC curves.

normal traffic samples fall into a tight distribution, but that there exists a long tail which complicates selecting a threshold to distinguish the known and unknown classes.

Looking at Figure 5.3 this problem is clear: Although the distributions of reconstruction error for normal and attack traffic are separated, the long tail of the normal traffic extends well beyond the distribution of attack traffic reconstruction errors. Any selection of reconstruction error threshold which maximizes true positive rate will cut off this tail and result in a large false

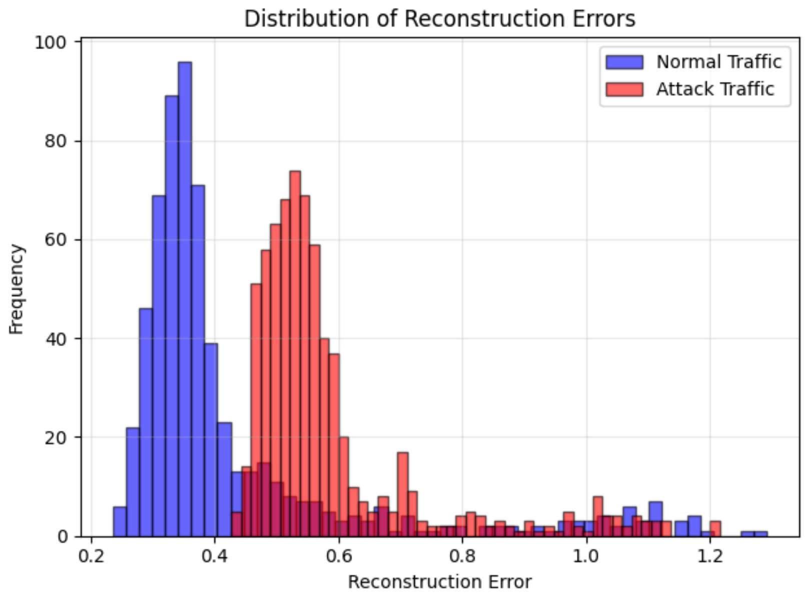


Figure 5.3: The long tail of the reconstruction error distribution for the normal CAN bus traffic complicates selection of an error threshold for distinguishing attacks.

positive rate.

The shape of the data indicates that CAN bus traffic is characterized by recurring, common messages (the main bell curve) but also infrequent or irregular messages which are still part of normal CAN bus operation but which pop up far less frequently (the long tail).

Perhaps more training data or more optimal hyperparameters can trian the model to better reconstruct inputs that currently belong to the long tail which will greatly improve model performance.

6 Evaluation

6.1 Best AUC Performance

Table 6.1 summarizes the parameters used to train the model for best performance in terms of AUC of the ROC curve for each of the attack types tested. Across all attack types, the autoencoder achieved high AUC values which indicates that the reconstruction error offers separability between normal and attack frames.

The model performs best with fuzzing attacks, which makes sense given the fuzzing attacks involve sending random CAN ID values on the bus, which the model would not have been trained on, while other attacks are more sophisticated in that they replay valid CAN IDs for some malicious intention.

the combined, gear, rpm, and standstill attack types all show AUC values near 0.9, which suggests the model is fairly good at predicting these as well.

The model performs relatively poorly on DoS, speed, and interval attacks, so further fine-tuning would be needed for the autoencoder's reconstruction error to be a reliable predictor for these.

The BFW was enabled for all of these maximal AUC models except for DoS and fuzzing attacks. This indicates that the inclusion of BFW may be a factor for improving the AUC to a greater extent as the model performance improves due to the configuration of the other parameters.

Overall, the promising AUC values show that the autoencoder is mostly successful in distinguishing attack frames from normal frames– that is, there is a significant difference in the reconstruction errors for attack and normal frames.

6.2 Best F1 Score

The AUC provides a threshold-independent assessment of the model's performance, but for the model to be deployed, a threshold would need to be determined, hence the F1 score is still an important metric.

The F1 score results (shown in Table 6.2) are troubling, since they show that even when the AUC for a given attack type is promising, the maximum

Table 6.1: Maximum AUC of ROC curve achieved for each attack type and parameters for best performance.

attack type	batch size	learning rate	epochs	msg per input	bfw	auc
combined	512	0.002	1000	256	1	0.932146771
dos	512	0.0001	1000	256	0	0.871945741
fuzzy	512	0.0005	1000	128	0	0.999661692
gear	512	0.002	1000	256	1	0.90093119
interval	512	0.0005	1000	32	1	0.708233916
rpm	512	0.002	1000	256	1	0.906106613
speed	512	0.0005	1000	256	1	0.857480673
standstill	512	0.002	1000	256	1	0.906834394

F1 score achieved may not be high.

Indeed, for all attack types other than dos, fuzzy, and interval, the F1 score is quite low and indicates the model would not be effective as a predictor of whether an attack was taking place.

The AUC may be high, indicating that there is a significant difference in the reconstruction errors for the two classes, but the F1 can still be low, which means no threshold selection has good precision and recall. This could be the case when the error distributions for the two classes have means that are separated substantially, but whose tails overlap significantly.

The observations here of whether the BFW is enabled on these highest-performing model configurations is identical to that in the previous section, which again suggests that the BFW is a promising method for improving model performance.

6.3 Effect of BFW on AUC and F1

Table 6.3 shows the average AUC value across all parameter configurations of the model with the BFW enabled, and for those without the BFW enabled. It is important to note that for every combination of parameter values for which the model was trained and evaluated, the model was trained and evaluated with and without the BFW enabled to allow for fair comparison of the average effect of using the BFW.

That said, although the inclusion BFW generally slightly increases the

Table 6.2: Maximum F1 score achieved for each attack type and parameters for best performance.

attack type	batch size	learning rate	epochs	msg per input	bfw	auc	max f1
combined	512	0.002	1000	256	1	0.932146771	0.188679245
dos	256	0.0018	1000	224	0	0.859448161	0.912710567
fuzzy	512	0.002	1000	224	0	0.999173709	0.998822144
gear	512	0.002	1000	256	1	0.90093119	0.215809285
interval	512	0.0001	1000	256	1	0.190671031	0.993494172
rpm	512	0.002	1000	256	1	0.906106613	0.275280899
speed	512	0.0005	1000	256	1	0.857480673	0.118012422
standstill	512	0.002	1000	256	1	0.906834394	0.277777778

AUC for most attack types, the high p-values resulting from a T-Test show that this difference in AUC is not statistically significant.

Similarly, Table 6.4 shows that the small improvement the BFW adds to the average F1 score is not statistically significant.

It cannot be said from these results that a BFW always improves autoencoder performance for OCC in CAN bus intrusion detection for all attack types, even though the improvement seen by enabling the BFW among the highest-performing parameter combinations of the model in tables 6.1 and 6.2 are promising.

Table 6.3: Effect of BFW on mean AUC of ROC curve.

attack type	Average AUC w/ BFW	Average AUC w/o BFW	Difference in AUC mean	AUC Mean T Test p-value	Percent Improvement in AUC mean with BFW
combined	0.826095762	0.816539836	0.009555926	0.761548365	1.170295157
dos	0.80550158	0.814371122	-0.008869542	0.714614287	-1.089127731
fuzzy	0.977799241	0.977122082	0.000677159	0.97296619	0.069301324
gear	0.683509097	0.657284032	0.026225065	0.477526173	3.989913629
interval	0.572302369	0.564028981	0.008273388	0.84092411	1.466837379
rpm	0.79943465	0.785840837	0.013593813	0.647115285	1.72984298
speed	0.763801429	0.759404113	0.004397316	0.804667648	0.579048183
standstill	0.797168415	0.786118394	0.011050021	0.702841116	1.40564338

Table 6.4: Effect of BFW on mean F1 score.

attack type	Average F1 with BFW	Average F1 without BFW	Difference in F1 mean	F1 Mean T Test p-value	Percent Improvement in AUC mean with BFW
combined	0.117979391	0.111959004	0.006020388	0.590715696	5.37731434
dos	0.864217452	0.872967772	-0.00875032	0.591450952	-1.002364591
fuzzy	0.945836447	0.944790223	0.001046225	0.977353242	0.110736176
gear	0.086892499	0.078125988	0.008766511	0.378618931	11.22099173
interval	0.793008022	0.786842447	0.006165575	0.911550916	0.783584394
rpm	0.161738164	0.152070173	0.009667991	0.484399436	6.357585246
speed	0.075177555	0.073300329	0.001877225	0.689016663	2.561005362
standstill	0.162079824	0.155605887	0.006473938	0.644390019	4.160470943

7 Conclusion

More time is needed to draw concrete conclusions about the effectiveness of autoencoders for one-class classification of CAN bus attacks and whether novel techniques like the Batch-Wise Feature Weighting are a means of improving performance.

What can be said from this project so far is that the autoencoder can be trained to have lower reconstruction error for attack-free CAN bus traffic than for traffic under certain attack scenarios, which leads to good AUC of ROC curves for these datasets. That said, more work is needed to cull the long tail discussed in Section 5.6.1 and generally improve the F1 score which would enable this model to be deployed with sufficient precision and recall to be useful in a vehicle.

The effects observed of the BFW are promising, but at this time are not statistically significant. More work is needed here too to define exactly where and how much the BFW can improve performance of OCC for CAN bus intrusion detection.

8 Future Work

There are a number of problems with the current implementation which will be tackled in the coming months, as well as future goals that go beyond the scope of this project.

- **Test more model parameters:** The hyperparameters chosen to experiment with in this project yielded good AUC, but poor F1. The most pressing issue with the project is this poor F1 performance, so the logical next step is to test the effect variation of currently staitc parameters has on the F1 score. For example, varying the latent dimension or size of the hidden layers of the BFW, or using a more creatiev feature extraction method could yield better results.

- **Port to embedded hardware:** One of the original goals of this project was to port the model to the Arduino-Uno Q, which is a new (2025) device with an embedded microcontroller typical of other Arduino systems for real-time tasks, as well as a Qualcomm MPU which is adept at AI workloads. This combination at the relatively lwo price point of 45$ makes the platform appealing for this project. The original idea was to have the microcontroller handle CAN bus sniffing and feature extraction and make the input vectors available to the MPU over a shared buffer, at which point the MPU could forward propagate the input vector through the model and detect anomalous behavior. The device has sufficient RAM to handle the model out of the box, although other modifications may need to be made to the model to get it to run in real time on the relatively limited hardware. This action item is critical since it will take this project from a theoretical one to a potentially useful one.

9 Deliverables

- **main github:** The github folder linked has all the code and deliverables for the final of this project: https://github.com/finncg1234/RTES-2025/tree/main/autoencoder_can
- **README.md:** has instructions for how to run each of the above files and links to all deliverables
- **autoencoder.py:** file where the pytorch neural network is defined.
- **can_main.py:** file which specifies how the model is trained and evaluated
- **dataset.py:** file which handles all data preprocessing and feature extraction
- **config.py:** has some base data classes used by the above files.
- **RTES-2025-stats.xlsx:** excel spreadhseet used to get final results and tables for this report
- **out/:** folder containing all the weights for the model trained under numerous parameter combinations.
- **Final Presentation:** youtube link to final presentation: https://youtu.be/gu0wGI2u8mg

10 Skill Learning

10.1 Trend Recognition and Prediction

The trend this project centers on is the proliferation of computing devices in automobiles, especially with the increased presence of autonomous vehicles. This isn't exactly a novel trend, but it is one I can project will continue: more and more features of automobiles which are currently mechanical or are siloed away from other systems will be integrated wirelessly and with computing nodes.

10.2 Testing assumptions in a Controlled Reality

This was a major area of learning for this project. The machine learning pipeline I created on my laptop was, in effect, a controlled reality. Some of my assumptions were that reconstruction error would be greater for unseen/anomalous CAN traffic, that the BFW would reduce an autoencoder's ability to generalize to unseen data, and that high AUC would correspond to high F1 score. I proved or disproved each of these assumptions in the controlled environment of my laptop– equipped with an NVidia GPU.

10.3 Differentiating Fact from Fiction

While analyzing my model's performance, I needed to sift through various metrics to determine if the model was *truly* performing well. From just looking at AUC (as I did when first evaluating the model) it appeared that it was performing quite well. But after taking into account the F1 score– a threshold-dependent metric– it was clear that the AUC alone portrayed a fictional account of the model's promise. THe F1 scores revealed that there is still much work to be done.

10.4 Cyber-Physical Gap

I didn't quite get to my goal of deploying the model on embedded hardware, but if I had, I would have gotten first-hand experience of identifying the cyber-physical gap. Surely there will be surprising hang-ups with deploying the model on embedded hardware and with interpreting CAN data in he real world, outside of the dataset I've been using so far. I did learn a lot about the cyber-physical gap through the many papers I read as background for this project an the difficiulties they had transitioning from the lab to the real world.

10.5 Reading and Writing Skills

This paper is likely the most challenging writing assignment I have done to date. There is a lot of complicated technical information to convey, and I must consider audience and message. The writing task is made more difficult by the fact that my findings are not as clear as I hoped they would be. During the course of the semester, I refined my writing on the background and related work for this project by incorporating feedback from peer reviewers and TAs and by reading more and more about the technical field.

10.6 Analyzing Related Work and Recognizing Potential Opportunities

I read over a dozen papers thoroughly to get a sense of the state of the art of CAN bus intrusion detection with ML. I made use of techniques I read about in these papers in my project. This is likely the area of most improvement this semester since I haven't done much forma research, so I got experience organizing research and using tools like Zotero and Latex to organize my readings.

11 References

[1] *ISO 11898-1*, version 3, 2024. Accessed: Nov. 30, 2025. [Online]. Available: https://cdn.standards.iteh.ai/samples/86384/ca6e28c79b6b4388a5104dfda2332 ISO-11898-1-2024.pdf.

[2] J. N. Brewer and G. Dimitoglou, “Evaluation of Attack Vectors and Risks in Automobiles and Road Infrastructure,” in *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*, Las Vegas, NV, USA: IEEE, Dec. 2019, pp. 84–89, ISBN: 978-1-7281-5584-5. DOI: 10.1109/CSCI49370.2019.00021. Accessed: Nov. 7, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/9071073/.

[3] Y. Liao and B. Yang, “To Generalize or Not to Generalize: Towards Autoencoders in One-Class Classification,” in *2022 International Joint Conference on Neural Networks (IJCNN)*, Padua, Italy: IEEE, Jul. 18, 2022, pp. 1–8, ISBN: 978-1-7281-8671-9. DOI: 10.1109/IJCNN55064.2022.9892812. Accessed: Nov. 7, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/9892812/.

[4] F. Amato, L. Coppolino, F. Mercaldo, F. Moscato, R. Nardone, and A. Santone, “CAN-Bus Attack Detection With Deep Learning,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 22, no. 8, pp. 5081–5090, Aug. 2021, ISSN: 1524-9050, 1558-0016. DOI: 10.1109/TITS.2020.3046974. Accessed: Nov. 7, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/9325944/.

[5] H. Ma, J. Cao, B. Mi, D. Huang, Y. Liu, and S. Li, “A GRU-Based Lightweight System for CAN Intrusion Detection in Real Time,” *Security and Communication Networks*, vol. 2022, C. Chen, Ed., pp. 1–11, Jun. 27, 2022, ISSN: 1939-0122, 1939-0114. DOI: 10.1155/2022/5827056. Accessed: Nov. 7, 2025. [Online]. Available: https://www.hindawi.com/journals/scn/2022/5827056/.

[6] J. Guidry, F. Sohrab, R. Gottumukkala, S. Katragadda, and M. Gabbouj, “One-Class Classification for Intrusion Detection on Vehicular Networks,” in *2023 IEEE Symposium Series on Computational Intelli-*

gence (SSCI), Mexico City, Mexico: IEEE, Dec. 5, 2023, pp. 1176–1182, ISBN: 978-1-6654-3065-4. DOI: 10.1109/SSCI52147.2023.10371899. Accessed: Nov. 7, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/10371899/.

[7] C. Chupong, N. Junhuathon, K. Kitwattana, T. Muankhaw, N. Ha-Upala, and M. Nawong, "Intrusion Detection in CAN Bus using the Entropy of Data and One-class Classification," in *2024 International Conference on Power, Energy and Innovations (ICPEI)*, Nakhon Ratchasima, Thailand: IEEE, Oct. 16, 2024, pp. 157–160, ISBN: 979-8-3503-5677-9. DOI: 10.1109/ICPEI61831.2024.10748816. Accessed: Nov. 7, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/10748816/.

[8] L. Liang et al., "Intrusion Detection Model for In-vehicle CAN Bus Based on TPE-LightGBM Algorithm," in *2025 IEEE 34th Wireless and Optical Communications Conference (WOCC)*, Taipa, Macao: IEEE, May 20, 2025, pp. 419–423, ISBN: 979-8-3315-3928-3. DOI: 10.1109/WOCC63563.2025.11082193. Accessed: Nov. 7, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/11082193/.

[9] B. M. Tóth and A. Bánáti, "Autoencoder Based CAN BUS IDS System Architecture and Performance Evaluation," in *2025 IEEE 19th International Symposium on Applied Computational Intelligence and Informatics (SACI)*, Timisoara, Romania: IEEE, May 19, 2025, pp. 000 099–000 104, ISBN: 979-8-3315-1547-8. DOI: 10.1109/SACI66288.2025.11030168. Accessed: Nov. 7, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/11030168/.

[10] B. Lampe and W. Meng, "Can-train-and-test: A New CAN Intrusion Detection Dataset," in *2023 IEEE 98th Vehicular Technology Conference (VTC2023-Fall)*, Hong Kong, Hong Kong: IEEE, Oct. 10, 2023, pp. 1–7, ISBN: 979-8-3503-2928-5. DOI: 10.1109/VTC2023-Fall60731.2023.10333756. Accessed: Nov. 7, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/10333756/.

[11] M. Kemmler, E. Rodner, E.-S. Wacker, and J. Denzler, "One-class classification with Gaussian processes," *Pattern Recognition*, vol. 46, no. 12, pp. 3507–3518, Dec. 2013, ISSN: 00313203. DOI: 10.1016/j.patcog.2013.06.005. Accessed: Nov. 7, 2025. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0031320313002574.

[12] J. Lee, S. Park, S. Shin, H. Im, J. Lee, and S. Lee, "ASIC Design for Real-Time CAN-Bus Intrusion Detection and Prevention System Using Random Forest," *IEEE Access*, vol. 13, pp. 129 856–129 869, 2025, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2025.3585956. Accessed: Nov. 7, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/11071663/.

[13] F. Sohrab, J. Raitoharju, M. Gabbouj, and A. Iosifidis, "Subspace Support Vector Data Description," in *2018 24th International Conference on Pattern Recognition (ICPR)*, Beijing: IEEE, Aug. 2018, pp. 722–727, ISBN: 978-1-5386-3788-3. DOI: 10.1109/ICPR.2018.8545819. Accessed: Nov. 7, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/8545819/.