Autoencoder for Vibration Signal Compression on Raspberry Pi

Preston Malen
Math 5320 Fall 2025

December 5, 2025

Abstract

This report describes the design, training, and deployment of a 1D convolutional autoencoder for compressing vibration signals. The encoder is exported to ONNX and deployed on a Raspberry Pi 5, while the decoder and reconstruction analysis are performed on a desktop machine. We outline the data pipeline, model architecture, deployment strategy, and evaluation using a bearing vibration dataset.

Contents

1 Introduction

Modern condition monitoring systems often rely on high-frequency time series. Transmitting raw signals from embedded devices can be expensive in terms of bandwidth and energy. Autoencoders offer a way to learn a compact representation (latent space) of such signals.

In this project, we:

- Train a 1D convolutional autoencoder on bearing vibration data.
- Export the encoder to ONNX and deploy it on a Raspberry Pi 5.
- Log latent vectors on the Pi and analyze them on a desktop machine.
- Evaluate reconstruction quality and discuss potential for anomaly detection.

2 Background

2.1 Autoencoders

Autoencoders are not a particularly new architecture but they remain one of the most powerful frameworks for signal compression. In essence, our network learns a map $f_{\text{enc}} : \mathbb{R}^{512} \to \mathbb{R}^d$ which compresses the signals then a decoder $f_{\text{dec}} : \mathbb{R}^d \to \mathbb{R}^{512}$ which unzips the compression. We then measure the accuracy of the network as the reconstruction loss $\|f_{\text{enc}} - f_{\text{dec}}\|$. The intermediate space $\mathbb{R}^d$ is our latent space and we assume $d \ll 512$.

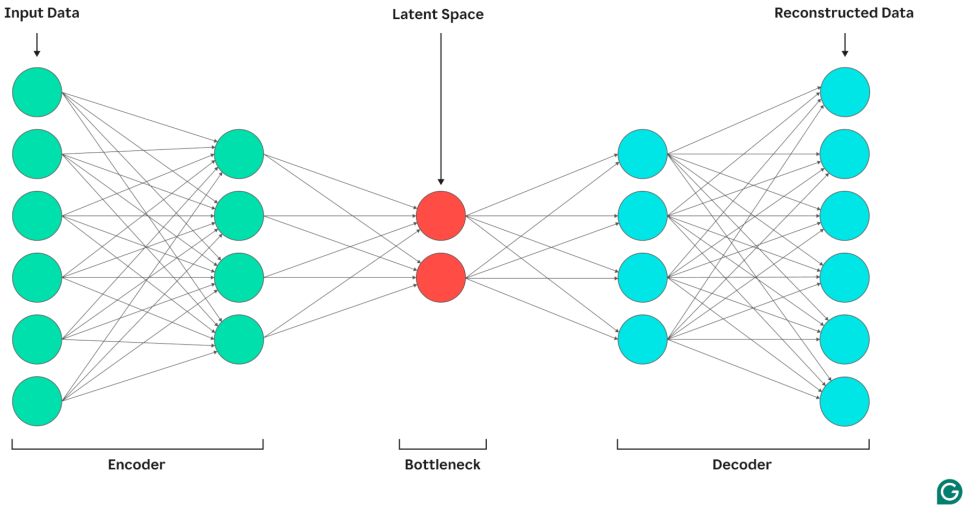


Figure 1: Autoencoder architecture Courtesy of Grammarly. f_{enc} maps from the green to red while f_{dec} maps from red to blue.

2.2 Embedded Systems and Raspberry Pi

The motivation for this project was Dr. White's research on space deployed systems. We wanted to create a network that would be powerful yet lightweight enough to deploy on a small embedded system on a satellite. So we use a Raspberry Pi 5 to simulate a small but start of the art embedded

system. The autoencoder was trained externally on a desktop machine then the encoder was moved to the Raspberry Pi. The Raspberry Pi would then be responsible for reading incoming signals, compressing them using the trained encoder, then logging/transmitting the compressesed signal information.

2.3 Bearing Vibration Dataset

We used the NASA bearing dataset accesible here. The datacard is below.

Set	Property	Value
1	Recording Duration	Oct 22, 2003 – Nov 25, 2003
	No. of Files	2,156
	Channels	8
	Channel Mapping	B1: Ch 1–2; B2: Ch 3–4; B3: Ch 5–6; B4: Ch 7–8
	Sampling Interval	10 min (first 43 at 5 min)
2	Recording Duration	Feb 12, 2004 – Feb 19, 2004
	No. of Files	984
	Channels	4
	Channel Mapping	B1: Ch 1; B2: Ch 2; B3: Ch 3; B4: Ch 4
	Sampling Interval	10 min
3	Recording Duration	Mar 4, 2004 – Apr 4, 2004
	No. of Files	4,448
	Channels	4
	Channel Mapping	B1: Ch 1; B2: Ch 2; B3: Ch 3; B4: Ch 4
	Sampling Interval	10 min

Table 1: Relevant summary of bearing test-to-failure datasets.

3 Methods

3.1 Data Preprocessing

- **Windowing of raw signals.** The continuous accelerometer signals from each channel are first segmented into fixed-length windows of 512 samples. For each file and each channel, we slide a window of length 512 over the raw time series and extract contiguous, non-overlapping segments. Segments that do not contain the full 512 samples at the end of a recording are discarded to ensure that every example fed to the network has the same length. For multi-channel experiments, windows from the same time interval are stacked along the channel dimension to form an input tensor of shape (channels $\times$ samples), which is then passed to the encoder.

- **Normalization strategy.** To reduce the influence of absolute amplitude and slowly varying trends, we apply per-window z-normalization. For each window $x \in \mathbb{R}^{C\times 512}$ (where C is the

number of channels), we compute the mean μ and standard deviation σ over all samples in that window (and all channels), and transform

$$\tilde{x} = \frac{x - \mu}{\sigma + \varepsilon}$$

where ε is a small constant added for numerical stability. This ensures that every window has approximately zero mean and unit variance before being used as input.

3.2 Model Architecture

Our autoencoder is a 1D convolutional architecture designed to process windows of length 512 samples. The encoder progressively reduces temporal resolution while increasing channel depth, ultimately flattening the feature map into a latent vector of dimension d_{latent}. The decoder mirrors this process through transposed convolutions.

The full architecture is summarized below.

Encoder. The encoder $f_{\text{enc}} : \mathbb{R}^{1\times 512} \to \mathbb{R}^{d_{\text{latent}}}$ consists of:

- Conv1D(1, 16, kernel=5, stride=2) + ReLU
- Conv1D(16, 32, kernel=5, stride=2) + ReLU
- Conv1D(32, 64, kernel=5, stride=2) + ReLU
- Flatten
- Linear($64 \times 64 \to d_{\text{latent}}$)

After the final convolution the temporal dimension has been reduced from 512 to 64. This feature map is flattened and projected to the latent space via a fully connected layer. In all experiments we used latent sizes between $d_{\text{latent}} = 16$ and 64, with $d_{\text{latent}} = 32$ used for deployment due to its balance of reconstruction accuracy and computational cost.

Decoder. The decoder $f_{\text{dec}} : \mathbb{R}^{d_{\text{latent}}} \to \mathbb{R}^{1\times 512}$ approximately inverts the encoder. It begins by expanding the latent vector back into a structured feature map:

- Linear($d_{\text{latent}} \to 64 \times 64$), reshape to (64 channels, 64 samples)
- ConvTranspose1D(64, 32, kernel=5, stride=2) + ReLU
- ConvTranspose1D(32, 16, kernel=5, stride=2) + ReLU
- ConvTranspose1D(16, 1, kernel=5, stride=2)

The final transposed convolution returns the temporal dimension to exactly 512 samples.

Stage	Operation	Output Shape
Input	–	(1, 512)
Encoder 1	Conv1D(1,16,5,2) + ReLU	(16, 256)
Encoder 2	Conv1D(16,32,5,2) + ReLU	(32, 128)
Encoder 3	Conv1D(32,64,5,2) + ReLU	(64, 64)
Flatten	–	(4096)
Latent FC	Linear(4096,d_{latent})	$\mathbb{R}^{d_{\text{latent}}}$
Decoder FC	Linear(d_{latent},4096)	(4096)
Reshape	–	(64, 64)
Decoder 1	ConvT(64,32,5,2) + ReLU	(32, 128)
Decoder 2	ConvT(32,16,5,2) + ReLU	(16, 256)
Output Layer	ConvT(16,1,5,2)	(1, 512)

Table 2: Summary of 1D convolutional autoencoder architecture.

3.3 Training Procedure

The full autoencoder was trained in PyTorch on a desktop GPU using the following procedure:

Loss Function. We used the Smooth L1 loss,

$$\mathcal{L}(x, \hat{x}) = \text{SmoothL1}(x - \hat{x})$$

which tends to be more stable than MSE for vibration signals that occasionally contain large transients. Smooth L1 also reduces the influence of outliers while still encouraging precise reconstruction.

Optimizer and Hyperparameters. Training was performed using the Adam optimizer with:

$$\text{learning rate} = 10^{-3}, \qquad \text{weight decay} = 10^{-5}$$

We used a batch size of 256 and trained for 40–60 epochs depending on the experiment. A ReduceLROnPlateau scheduler was applied with patience 5, decreasing the learning rate by a factor of 0.5 when validation loss saturated. Gradient clipping at a maximum norm of 1.0 prevented instability when windows contained unusually high-amplitude spikes.

Train/Test Split. Each bearing dataset was segmented into windows as described previously. We randomly assigned 80% of windows to the training set and 20% to the test set, stratified by file index so that consecutive time intervals did not leak across sets. No windows from the same time step appear in both sets.

3.4 Deployment Pipeline

As explained previously, the autoencoder was deployed using a hybrid workflow between the desktop machine and the Raspberry Pi.

a) Desktop: Training and ONNX Export. After training the full autoencoder in PyTorch, we exported only the encoder using:

```
torch.onnx.export(encoder, example_input, "encoder.onnx", opset_version=17)
```

The exported model contains only the forward graph for f_{enc}, enabling efficient inference on the Pi without storing the decoder.

b) Raspberry Pi: Embedded Inference. The Raspberry Pi loads the ONNX file using `onnxruntime-gpu` (or CPU fallback, depending on installation). Each incoming signal window $x \in \mathbb{R}^{1\times 512}$ is normalized, passed through the encoder, and mapped to a latent vector $z = f_{\text{enc}}(x)$. The vector is then appended to a lightweight log file:

```
timestamp, z1, z2, ..., z_dlatent
```

The Pi performs no reconstruction and no decoding, reducing its workload to 0.1–0.3 ms per window.

c) Desktop: Latent Retrieval and Reconstruction. Latent logs are transferred back to the desktop via `scp`. The decoder f_{dec} (stored only locally) reconstructs:

$$\hat{x} = f_{\text{dec}}(z)$$

We then compute:

$$\|x - \hat{x}\|_1 \quad \text{or} \quad \text{SmoothL1}(x, \hat{x})$$

and visualize reconstructed signals. This workflow simulates a real-world monitoring loop where only compressed information is transmitted from the embedded device.

3.5 PyTorch Autoencoder

The PyTorch implementation follows the architecture described above. Modules are implemented using `nn.Conv1d`, `nn.ConvTranspose1d`, and `nn.Linear`, and the forward method returns both the reconstruction and latent vector when operating in full-autoencoder mode.

3.6 ONNX Export and Pi Inference

The ONNX export required ensuring all tensor shapes are static and that no PyTorch-only operations remain in the graph. On the Pi, inference was run through `onnxruntime.InferenceSession`, which efficiently executes the encoder on the ARM CPU.

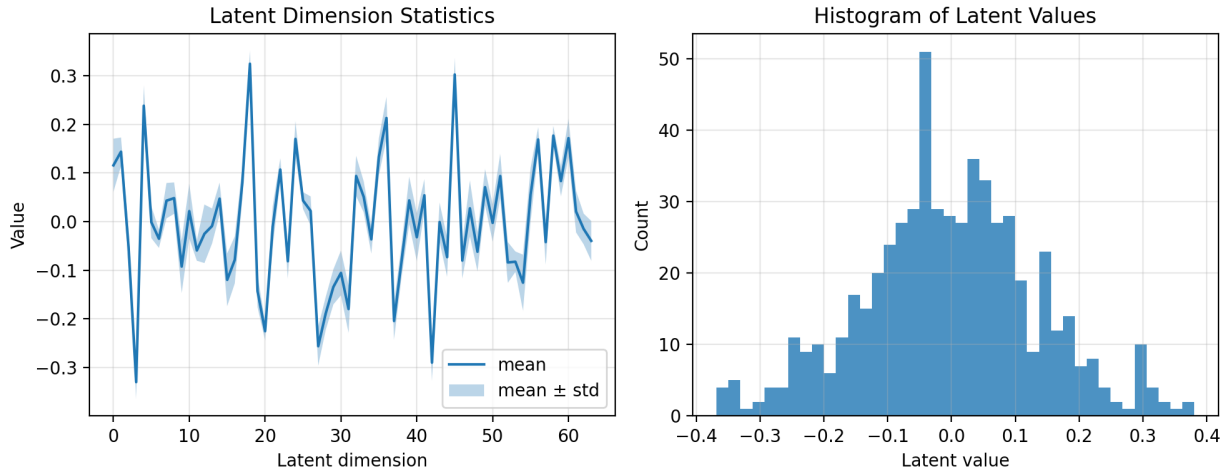
3.7 Logging Latent Vectors

Latent vectors were written to disk in CSV format with timestamps. This enables downstream filtering, window alignment across channels, and visual analysis (PCA, clustering, anomaly detection, etc.).

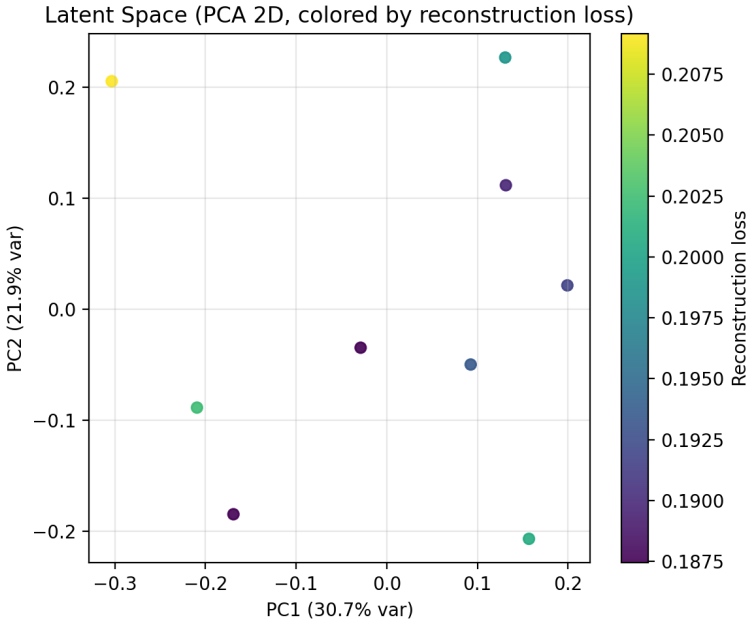
4 Results

4.1 Latent Space Visualization

To analyze the behavior of the deployed autoencoder, we collected pairs (x, z) from the Raspberry Pi, where $x \in \mathbb{R}^{512}$ is the input window and $z \in \mathbb{R}^{64}$ is the corresponding latent vector. Each z is decoded to obtain a reconstruction $\hat{x}$, and the Smooth L1 loss between x and $\hat{x}$ provides a per-sample reconstruction error.



We apply PCA to the latent vectors to obtain a two-dimensional embedding and plot each sample in this reduced space, coloring points by their reconstruction loss. This visualization highlights how the autoencoder organizes the data: dense, low-error regions correspond to well-represented signals, while outliers with higher loss indicate inputs that the model reconstructs less accurately. Together with basic statistics of each latent dimension, these plots provide a concise view of the structure and variability of the learned representation.



The reconstruction loss is summarized over 9 runs where the average was taken from 1000 iterations each.

Sample #	**Reconstruction Loss**
1	0.198857
2	0.189693
3	0.202334
4	0.209174
5	0.187486
6	0.187442
7	0.200802
8	0.193537
9	0.191842
Mean	0.195685
Std. Dev.	0.007094
Min	0.187442
Max	0.209174

Table 3: Reconstruction Loss Summary

5 Conclusion

The autoencoder worked extremely well. The reconstruction loss values were small and consistent, the latent values are normal, and the latent dimension values had a small standard deviation. Overall, these results are extremely positive and promising. The CNN was small enough to be loaded onto the Raspberry Pi with plenty of memory to spare while still being plenty powerful. The reconstruction loss proved that the accuracy of the decoder was more than sufficient. In the future, we hope to gather live sensor data and make the signal compression and transmissions in real time. But as a proof of concept, this pipeline showed lightweight networks can still achieve extraordinary performance while operating on minimal hardware.