**Hackathon Project: IQ Data Stream Compression for Testing & Measurement**

**Problem Statement**

Modern testing and measurement devices, such as oscilloscopes and signal generators, handle massive volumes of IQ (In-phase and Quadrature) data streams. These high-frequency signals require efficient storage and transmission, especially in remote processing applications.

The key challenges include:

* Handling large-scale IQ data transmission over limited bandwidth channels.
* Ensuring lossless or near-lossless compression to retain signal fidelity.
* Optimizing for real-time processing without overburdening the testing device.
* Enabling reconstruction of the original signal at the receiver end.

**Hackathon Challenge (Hosted by Rohde & Schwarz)**

The goal of this project was to develop an efficient **data compression algorithm** that minimizes the size of the IQ data stream while maintaining its usability for signal processing applications. The approach explored both **classical (PCA-based) and AI-driven (Autoencoder-based) techniques** to achieve optimal compression and reconstruction performance.

**Approach: Implementing Compression Solutions**

Two main approaches were tested:

**1. Principal Component Analysis (PCA) for Data Compression**

PCA is a classical **dimensionality reduction technique** that finds the most significant features in a dataset and removes redundant information.

* Applied PCA transformation to reduce data dimensionality while preserving variance.
* Selected the top principal components to retain essential signal information.
* Reconstructed the signal using inverse PCA transformation.
* Evaluated the compression ratio and reconstruction loss to optimize performance.

**2. Autoencoder-Based Data Compression**

An Autoencoder is a neural network architecture designed for unsupervised learning, capable of encoding and reconstructing data with minimal loss.

* Designed a deep learning model using TensorFlow/Keras.
* Implemented a bottleneck layer to compress the IQ data into a low-dimensional representation.
* Trained the network to minimize reconstruction loss using Mean Squared Error (MSE).
* Compared its efficiency to PCA in terms of compression ratio, accuracy, and processing time.

**Results & Key Takeaways**

**Findings:**

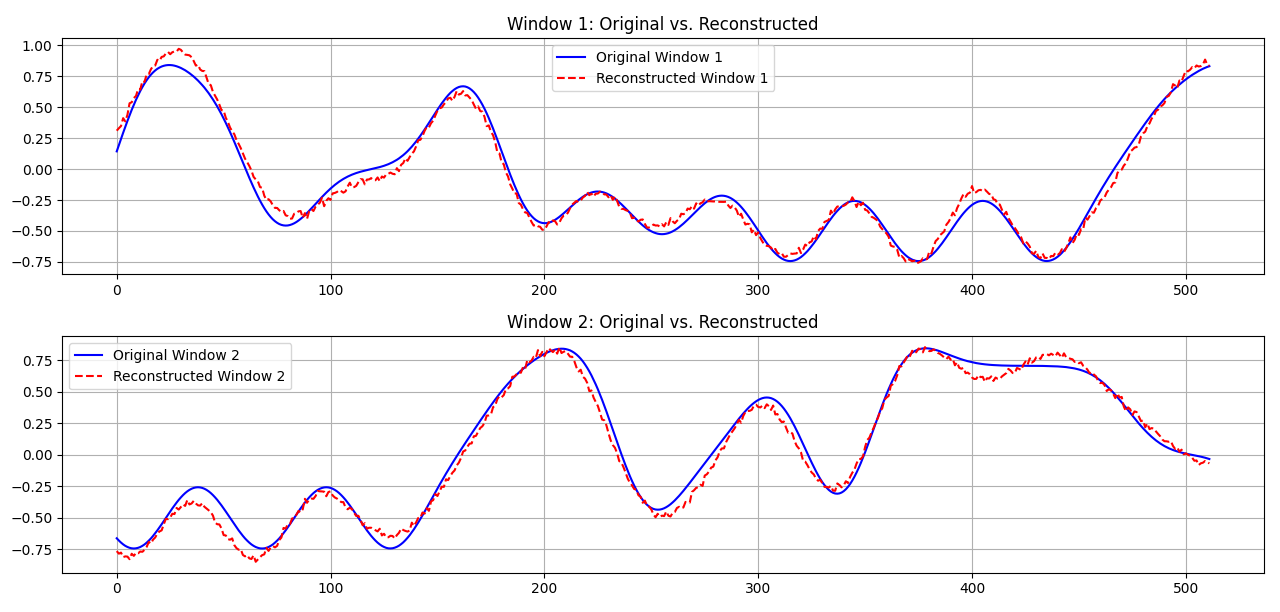
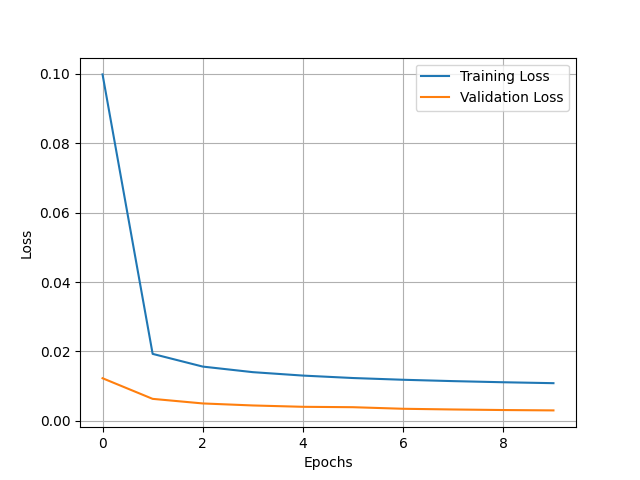
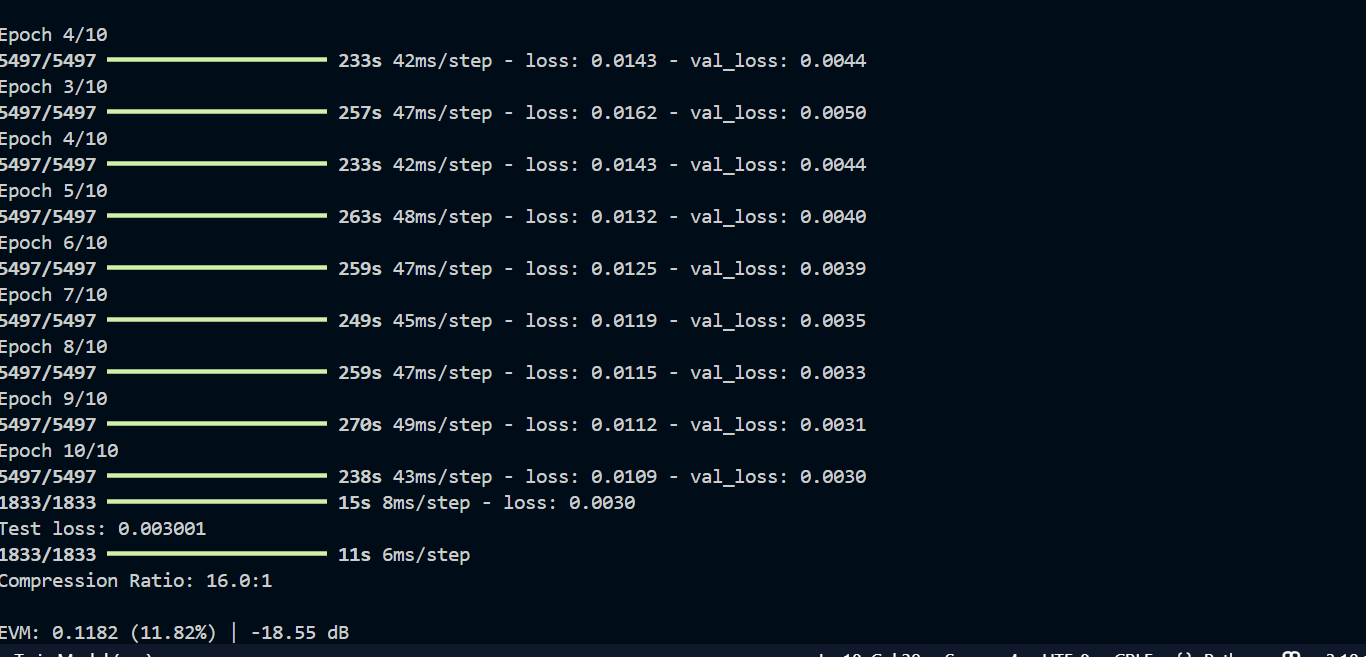
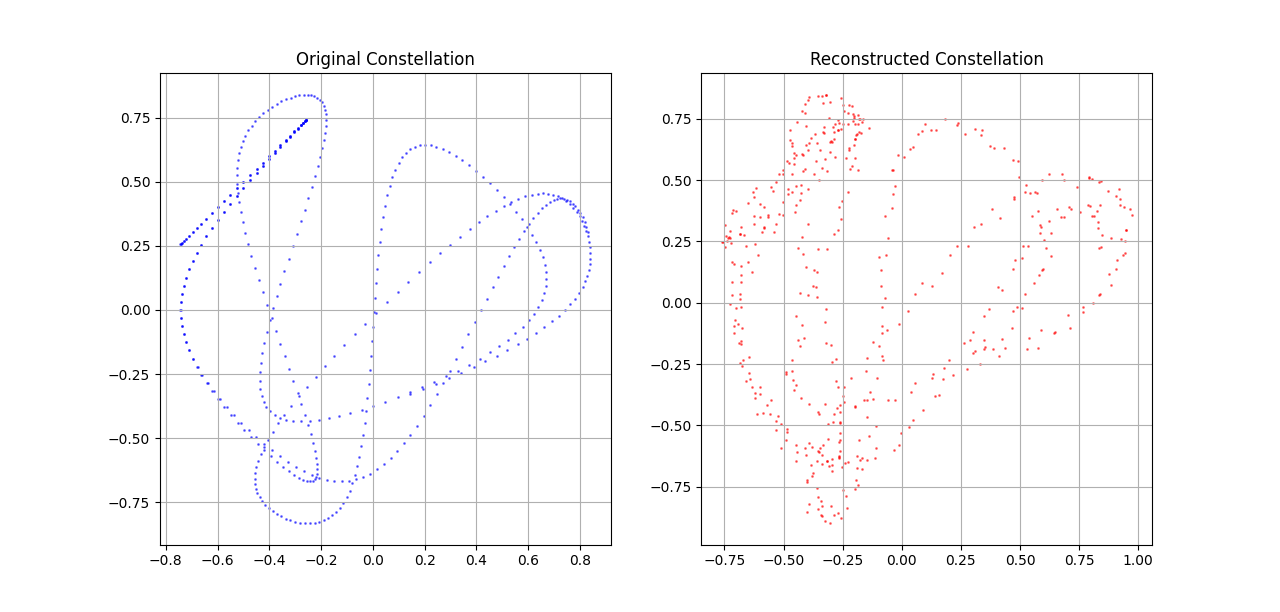
* PCA performed well for linear signal features, providing a straightforward compression technique with minimal computational cost.
* Autoencoder models captured more complex, nonlinear patterns, making them superior for handling more intricate IQ signal structures.
* A hybrid approach combining PCA for preprocessing and Autoencoder for final compression could be an optimal solution.

**Future Directions**

* Refining autoencoder models with advanced architectures like Variational Autoencoders (VAEs) or Transformer-based compression.
* Integrating real-world datasets for further validation.
* Deploying the solution in edge computing environments to enable real-time signal processing on low-power devices.
* Exploring adaptive compressiontechniques that dynamically adjust based on the channel conditions.

**Figures**

**Constellation Diagram, EVM metric, Train Loss and Validation loss and input signal and reconstructed signal**

****

**Fundamentals: PCA & Autoencoders**

**Principal Component Analysis (PCA) Overview**

* Reduces dimensionality while retaining most data variance.
* Uses Eigen decomposition of the covariance matrix.
* Best suited for linear, structured data.
* Computationally efficient but lacks adaptability for nonlinear data.

**Autoencoders Overview**

* Neural network architecture with encoder (compression) and decoder (reconstruction) components.
* Learns complex, nonlinear transformations in data.
* More powerful than PCA but requires training data and higher computational cost.

**Conclusion**

This project demonstrated the power of AI-driven and classical compression techniques for IQ data stream optimization. The findings from this Hackathon pave the way for scalable solutions in the testing & measurement industry, optimizing data transmission and storage without compromising signal integrity.

By leveraging machine learning, deep learning, and classical signal processing, future solutions can achieve higher compression ratios, lower latency, and improved adaptability to real-world conditions.

Table for evaluation:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Hyper Parameters | Iter 1/ 6 Epochs | Iter 2/10 epochs | Iter 3/5 epochs | Iter 4 |
| Learning Rate | 0.0001 | 0.001 | 0.001 |  |
| Encoding dim | 32 | 32 | 128 |  |
| Window size | 512 | 512 | 512 |  |
| Train /Val loss | Test loss: 0.003830 | Test loss: 0.002765 | Test loss: 0.007637 |  |
| EVM | 14% | 11.29% | 15.26% |  |
| Note | The constellation diagram is not exactly recognizable compared to input Reconstructed waveform looks good with visible variability | Overfitting problem, train loss saturates very fast and doen’t improve further with epochs | Better recontuction of constellation and the real valued waveform. Still validation loss doesn’t show much improvement but better performance to avoid overfitting problem as train loss shows constant deca and not very fast decay |  |

Looking at the loss graph from your autoencoder signal processing model, there are several interesting and somewhat unconventional behaviors that catch my attention:

1. **Validation loss lower than training loss**: Throughout all epochs, the validation loss (orange line) is consistently lower than the training loss (blue line). This is contrary to the conventional pattern where validation loss is typically higher than training loss.
2. **Initial gap between losses**: There's a significant initial gap between training and validation loss at epoch 0, which suggests fundamental differences in the data distributions between these sets.
3. **Rapid initial decrease in training loss**: The training loss shows a steep decline in the first epoch (from ~0.063 to ~0.04), followed by another significant drop by epoch 1 (to ~0.018), then a much more gradual decrease afterward.
4. **Smooth validation loss curve**: The validation loss decreases much more gradually and steadily throughout training without sharp drops.
5. **No overfitting signs**: There's no point where validation loss starts increasing while training loss continues to decrease, which would indicate overfitting.

Possible theoretical issues explaining these observations:

1. **Data splitting methodology**: Your code uses np.split() on data after applying a sliding window approach. This could create correlation between training and validation sets if the windows overlap across these splits, particularly with your 0.4 overlap ratio. This correlation might explain why validation loss is unnaturally low.
2. **Noise and regularization**: Your model includes GaussianNoise layers and Dropout (0.3) which are only active during training but not during validation/testing. This makes the training task deliberately harder, contributing to higher training loss compared to validation.
3. **BatchNormalization behavior**: Your model uses several BatchNormalization layers, which behave differently in training versus inference modes. During training, they use batch statistics, while during validation they use running averages, potentially leading to different loss behaviors.
4. **Validation set easier than training set**: Despite random shuffling, you might have ended up with an easier validation set due to chance. This is particularly possible with signal data where some sections may be cleaner than others.
5. **Huber loss function**: You're using Huber loss which is more robust to outliers. If your training data contains more outliers than your validation data, this could explain part of the gap.
6. **Small validation set**: With a 60/20/20 split and possibly limited total data, your validation set may not be representative enough to capture the true generalization error.

To address these issues, you might consider:

* Implementing a more rigorous train/validation split that ensures no window overlap between sets
* Using a larger validation set
* Experimenting with different regularization parameters
* Monitoring additional metrics beyond just the loss function

The behavior isn't necessarily "wrong" but rather indicates specific characteristics of your model architecture and data that are causing this unconventional loss pattern.