Convolutional Autoencoder for Compression and Feature Extraction of Neural Data

Philip Kroll

March 31, 2023

Abstract

This report discusses the use of Convolution Autoencoders (CAE) to address two main problems in the field of pain research: Spike Detection and Spike Sorting in Microneurography (MNG) data. MNG records the activity of single C-nociceptors in patients with neuropathic pain, providing insights into the mechanisms of pathological discharges in these peripheral nerve fibers. However, the low signal-to-noise ratio and high variability in spike shapes in MNG data make it challenging to accurately detect and sort spikes. CAEs are a type of autoencoder commonly used in computer vision tasks to detect features in images. The report proposes the use of CAEs to detect and compress spikes in MNG data, which can be further analyzed to understand the neural coding underlying pain perception.

1 Introduction

Pain is a multifaceted sensory and emotional sensation. It serves as the body's alarm system, warning us of impending damage or harm and encouraging preventive activities. However, when pain persists for an extended time, it can develop into a debilitating condition with adverse effects on a person's quality of life. Neuroscience, psychology, and medicine are just a few of the numerous fields involved in the study of pain [18, 12]. In recent years, there has been an increase in interest in using neuroimaging techniques to study the brain mechanisms underlying pain perception and the development of chronic pain conditions [19, 10]. The technique of Microneurography (MNG) records the activity of single C-nociceptors in patients with neuropathic pain, providing an approach to investigate the underlying mechanisms of pathological discharges in these peripheral nerve fibers [21, 22]. MNG employs a thin electrode placed within a peripheral nerve to record action potentials from neighboring fibers, but sorting these signals to single nerve fibers is necessary to understand neural coding. However, standard shape-based sorting algorithms, successful in in-vitro data, cannot be reliably used in MNG due to low signal-to-noise ratio and high variability in spike shapes. Thus, there are two problems: 1. Spike Detection and 2. Spike Sorting. In this report, we are going to introduce *Convolution Autoencoders (CAE)* to help with the aforementioned problems. A CAE is a special type of autoencoder, which is commonly used in computer vision tasks to detect features in images, as they can detect patterns in input data [14, 13, 8]. Because a CAE is capable of detecting the most important parts of the input data, the hope is that spikes get detected and appropriately compressed (Encoder) and decompressed (Decoder). Further, spike detection and further analysis can be done on the encoded data. For that to work reasonably the features in the signal in form of the spikes have to be detected and preserved by a used CAE. The workflow of this project can be seen in Section 1.

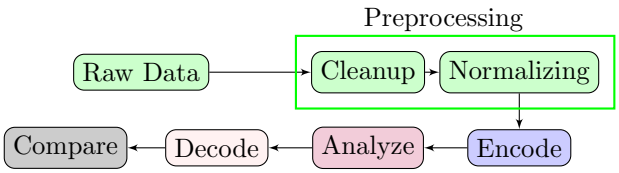


Figure 1: Workflow for using CAEs to compress and extract features of neural data.

2 Background

In the following, we are going to present the theoretical principles of *(Convolutional) Autoencoders*. Further, the *Compression Rate* of a given a is introduced, which is later used to distinguish between different architectures of used convolutional autoencoders.

2.1 Autoencoder

An autoencoder is a neural network usually used for unsupervised learning as it aims to recreate the input rather than to classify it under a certain class [6]. The network is trained on a set of input data and learns to compress the data into a lower-dimensional representation (this process is known as encoding). The compressed representation is then reconstructed back to the same dimension as the input data (this process is known as decoding). Autoencoders consist of two main components:

1. Encoder: Takes the input data and maps it into the latent representation. Or in other words, the first half of the autoencoder, where the dimension of data is progressively decreased.

2. Decoder: Takes the latent representation and maps it back to the original data dimension. Here the dimension of data is increased.

In autoencoders, the hidden layers have progressively fewer neurons until the *bottleneck* is reached. The bottleneck is a layer in the neural network, which has the least amount of neurons. Starting from the bottleneck the number of neurons per layer increases progressively until the output layer is reached. The goal of an autoencoder is to reconstruct the input data as closely as possible using the encoded representation while also learning useful features of the input data in the process. An important part of an autoencoder is the bottleneck. As it is the layer with the smallest amount of neurons in the whole network it restricts the flow of information from the input layer to the output layer. Thus, the network has to learn to let the most vital information pass through, such that the Decoder can reconstruct the input data as closely as possible. The abstract architecture of an autoencoder can be seen in Figure 2.

2.1.1 Convolutional autoencoder

A convolutional autoencoder (CAE) is a type of autoencoder that uses convolutional layers for both the encoding and decoding stages. Convolutional layers are commonly used in computer vision tasks to detect features in images, and in a CAE, these layers are used to detect patterns in the input data [14, 13, 8]. The convolution operation involves sliding a small filter or kernel over the input data and computing the dot product between the filter and the corresponding input values at each position. This produces a single output value, which is then stored in the output feature map. By sliding the filter over the entire input, the convolutional layer can generate a set of output feature maps that capture different patterns and features of the input data. Convolutional layers typically

have several parameters that can be adjusted to control the behavior of the layer. These include the size and stride of the filter, the number of filters in the layer, and the padding used around the input data. These parameters are defined and explained in the following [17, 5].

- *Filters*: Refers to the number of distinct convolutions performed on the input data. Each filter generates a different output feature map. Filters enable the extraction of different features from the input data, which can help improve the accuracy of the autoencoder.
- *Kernel size*: Refers to the dimension of the filters used to perform the convolution operation. The size of the kernel influences the level of detail that can be captured.
- *Stride*: Refers to the number of data points that the filter is moved by across the input data. A stride of 2 means that the filter moves 2 data points at a time. This influences the degree of overlap between adjacent regions.
- *Padding*: Refers to the addition of extra pixels around the edges of the input data. This is done to ensure that the output size of the convolution is the same as the input size.

Using filters decreases the number of connections between the layers compared to a fully connected layer. Compared to a general autoencoder, a CAE is particularly useful for processing data with a certain structure. A general autoencoder typically uses fully connected layers for the encoding and decoding stages, which are not able to capture the structural relationships between different parts of the signal. A CAE, on the other hand, is able to capture these relationships by using convolutional layers. The signal structures which can be detected particularly also include temporal features like action potentials or spikes in neural data. The Compression Rate (CR) is the ratio of the size of the original data to the size of the compressed data. Formally, the compression rate is defined in Equation (1). A higher compression rate corresponds to less information being able to pass the latent representation. This, in turn, forces the CAE to learn the features of the data more efficiently. Different Architectures of CAE can have different compression rates.

$$\text{CR} := \frac{\text{Original data size}}{\text{Encoded data size}} \tag{1}$$

3 Proposed Solutions

As mentioned in Section 2 a convolutional autoencoder can be used to compress and reconstruct data, particularly also neural data. The compression reduces the dimension of the data of interest and thus makes spike detection and further analysis possible. For that to work reasonably the features, in form of the spikes have to be detected, encoded and decoded correctly. In the following, it is presented how every step as shown in Section 1, except for the *Analysis* is performed.

3.1 Preprocessing

As described in Section 1, MNG is a technique used to record electrical activity by which we obtain the neural data of interest. In an in-vitro experiment in which C-fibers are stimulated from the skin of a mouse, the neural activity is recorded. The stimulation times were used to find the electrical artifacts and were replaced by a piece of raw signal (noise). In the following, the cleaned neural signal is being referred to as *data(set)*, is it is the solely used dataset. The data is normalized to the range $[0, 1]$ by using the formula Equation (2) to replace a value x with x_{scaled}. The normalization

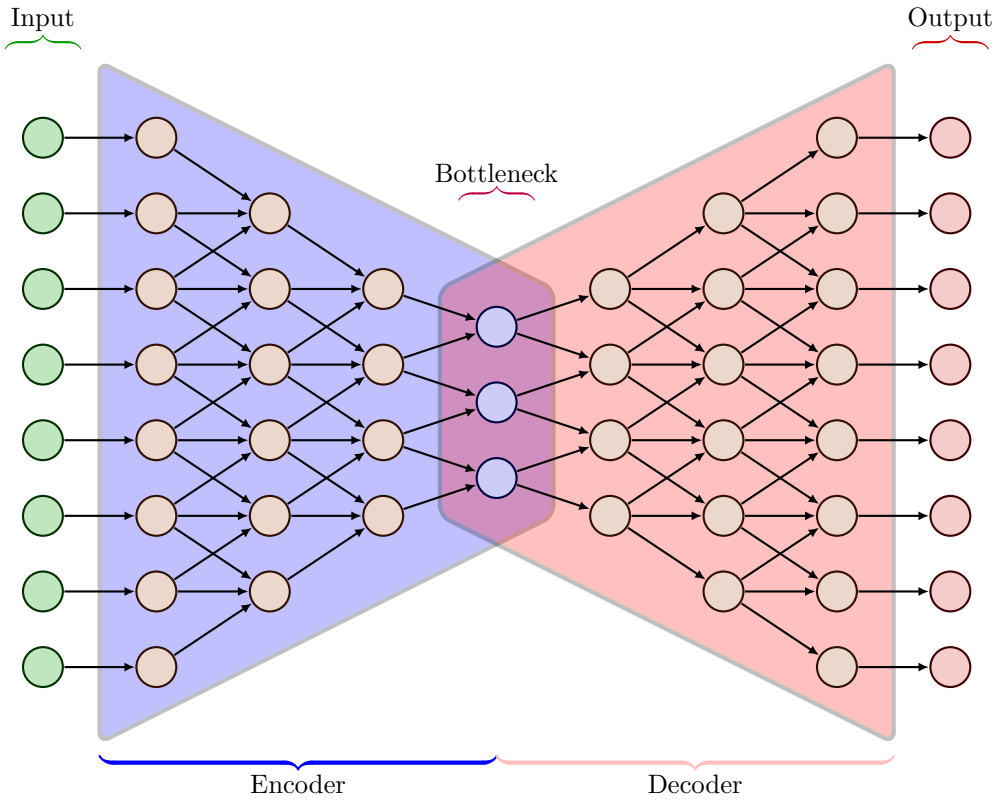


Figure 2: Abstract Architecture of an autoencoder. Notice that the Input and the Output have the same dimension and that each of the layers only connect to the respective next one. Further note that a neuron of one layer is not connected to every other neuron in the next layer.

step has a significant effect during training and testing. It guarantees stable convergence of weights and biases. Without the normalization step the training would be more complicated and slower [7].

$$\begin{aligned} x_{scaled} &:= \frac{(x - min)}{max - min} \\ min&\text{: Minimum } y\text{-value in dataset,} \\ max&\text{: Maximum } y\text{-value in dataset} \end{aligned} \tag{2}$$

This would reduce the dimension of the data of interest and thus make analysis and spike detection feasible. For that to work reasonably the features, in form of the spikes have to be detected, encoded and decoded correctly. The dataset is then split into continuous segments of length 1024. The sorted set of those segments is then split into a train and a test subset, where the train subset consists of 80% of the whole dataset and the test subset contains the rest of the segments. The segments in the train subset are used to train the CAE and the test subject is used for the respective evaluation. The order of the segments in the test subset is then randomized, this is done to prevent the model from memorizing the order of data. Shuffling the data ensures that the model is exposed to different patterns and variations in the data during training, which helps it to learn a more generalized representation of the data. This improves the model's ability to make accurate predictions on new,

unseen data [14, 16].

3.2 Used CAE Architectures

To investigate how a CAE performs on neural data, we are going to compare different architectures which in turn realize different CRs as defined in Equation (1). As an activation function for each layer, except for the very last one of the Decoder, *leaky relu*, as defined in Equation (3) with $\alpha := 0.3$ was chosen. It was shown that this prevents the *vanishing gradient problem* [4]. The last layer of the autoencoder was chosen to have the sigmoid activation function, as defined in Equation (4), as this forces the output value to the range $[0, 1]$, which is equal to the range of the input values. The Encoder is built by a sequence of 1D-Convolution layers, each with a different amount of filters. The Decoder is built by a sequence of 1D-Transposed convolution layers, each with a different amount of filters. A transposed convolution layer can be seen as a *deconvolution*, as it is going in the opposite direction of normal convolution [14].

$$f(x) := \begin{cases} \alpha \cdot x, & \text{if } x < 0 \\ x, & \text{if } x \geq 0 \end{cases} \tag{3}$$

$$f(x) := \frac{1}{1 + e^{-x}} \tag{4}$$

No pooling layers were used, but instead, each layer in both the Encoder and the Decoder has a stride of 2 to respectively decrease or increase the dimension of the data. This was done as a simplification and it was shown that this does not decrease the performance of the neural network [20]. The Decoder and the Encoder have the same amount of layers for each of the different compression rates. As a further simplification, the *kernel size* of every layer was chosen to be 8 and the input is zero-padded before convolution. The input layer is chosen to have a width of $1024 = 2^{10}$, as this coincides with the length of the segments of the data, as described in Section 3.1. This was chosen in combination with a stride of 2 as this halves the width of the next layer and thus works well with a width that is a power of 2. The general architectures of the CAE for each of the used CRs are presented in Figure 3. A more specific overview can be found in Figure 10. The adaptive moment estimation *(Adam)* algorithm [15] with an initial learning rate of 0.001, 100 training epochs, and a batch size of 128 was used to train the network. The loss function to minimize was chosen the be the Mean-Squared Error *(MSE)* between the Input of the Encoder and the Output of the Decoder, as defined in Equation (5).

$$\begin{aligned} MSE &:= \sum_{1}^{n} (y_i - \tilde{y_i})^2 \\ n &: \text{Number of data points,,} \\ y_i &: y\text{-value of the } i\text{-th input value of the Encoder,} \\ \tilde{y_i} &: y\text{-value of the } i\text{-th output value of the Decoder} \end{aligned} \tag{5}$$

Compression Rate (CR)	Number of layers in Encoder/Decoder	Number of filters in last Encoder Layer
2	3	4
4	3	2
8	5	4
16	5	2
32	6	2
64	6	1
128	7	1

Figure 3: CAE Architecture parameters for each CR

3.3 Performance Metrics

To evaluate the performance of a used CAE several metrics were used. This evaluation is always done as a comparison between the normalized input signal and the output of the Decoder, unless stated otherwise. The following metrics are used:

- MSE, as defined in Equation (5). This reflects the basic differences between the signals. A lower MSE is expected for a quality compression.
- *Signal-to-Noise Ratio (SNR)*, as defined in Equation (6). A high SNR is expected for a quality compression [11].
- *Threshold Analysis*, as defined in Section 3.3.1.

$$\begin{aligned} SNR &:= 10 \cdot \log\left(\frac{\sum_1^n (y_i - M)^2}{\sum_1^n (y_i - \tilde{y_i})^2}\right) \\ y_i &: y\text{-value of the } i\text{-th input value,} \\ M &: \text{mean of the input data,} \\ \tilde{y_i} &: y\text{-value of the } i\text{-th output value of the Decoder} \end{aligned} \tag{6}$$

3.3.1 Threshold Analysis

While MSE and SNR are metrics that help with the general evaluation, they do not give any insights about how well spikes in neural data get detected, encoded, and decoded. To do this we introduce the notion of *Thresholds*. A threshold is a constant value, and we detect how often and where the given signal crosses this value. To detect a crossing, we calculate the x-value (time) for a given constant. The conditions necessary for a crossing are described formally in Equation (7).

$$\begin{aligned} 1.&: y_i < t \leq y_{i+i}, \\ 2.&: y_{i+1} < t \leq y_i, \\ t &: \text{Threshold,} \\ y_i &: y\text{-value of the } i\text{-th data point,} \\ y_{i+1} &: y\text{-value of the } (i+1)\text{-th data point} \end{aligned} \tag{7}$$

Crossings can be identified in the original signal, as well as the decoded signal for every compression rate. This enables us to count the total number of crossings for every signal of interest. Further, this

enables the notion of *preserved crossings*, which is defined as a crossing at the same x-value in the original signal, as well as in the decoded signal for a given compression rate. Lastly, we introduce the *percentage of correctly preserved crossings*, as defined in Equation (8), where for a given threshold we inspect the ratio of the number of preserved crossings and crossings in the original data. For a well working autoencoder we expect a high percentage of correctly preserved crossings.

$$\textit{Correctly Preserved Crossings[\%]} := \frac{\#\text{preserved crossing}}{\#\text{original crossings}} \cdot 100 \tag{8}$$

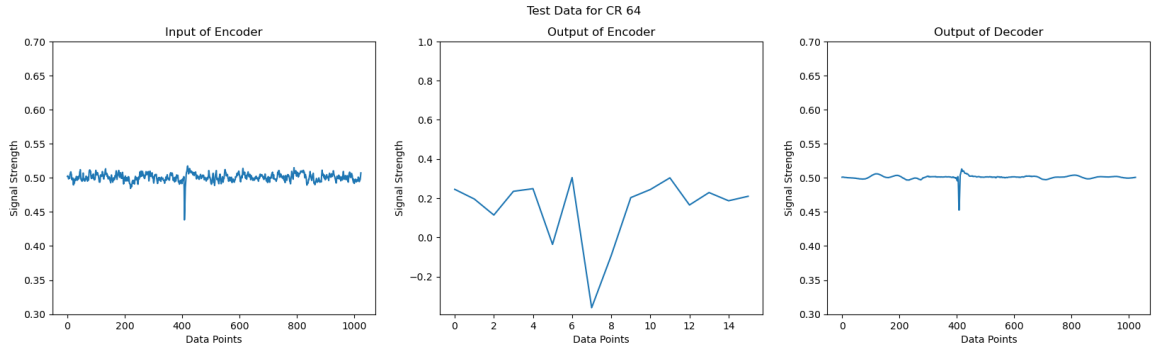
3.4 Environment Setup

Python 3.10 with the libraries Keras [9] and Tensorflow [3] is used to create and train the autoencoders. The experiments are performed on a PC with AMD Ryzen 7 5800X 8x 3.80GHz CPU, 32GB RAM and a AMD Radeon RX 6900 XT 16GB. The actual implementation was run in a docker container, which fixes the used python version and its libraries. This container can be found under [1]. Note that training of a neural network is highly optimized for training on a GPU. All files, that were used during this project can be found under [2].

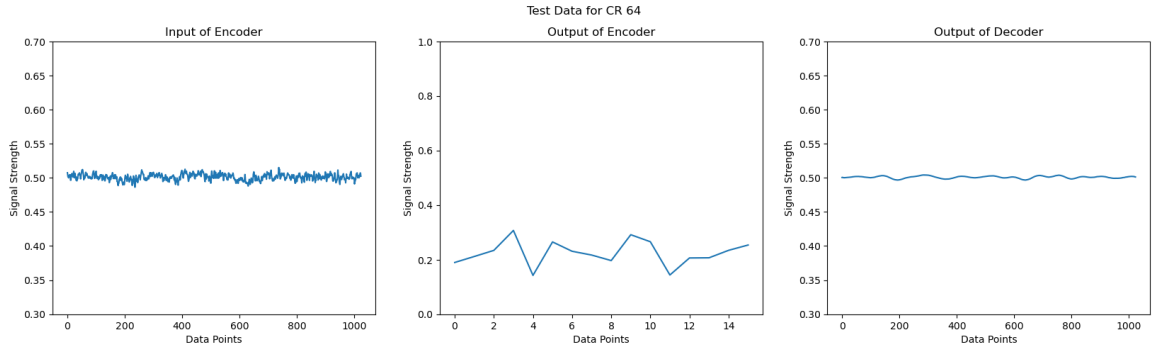
4 Results and Discussion

4.1 Compression Rate 64

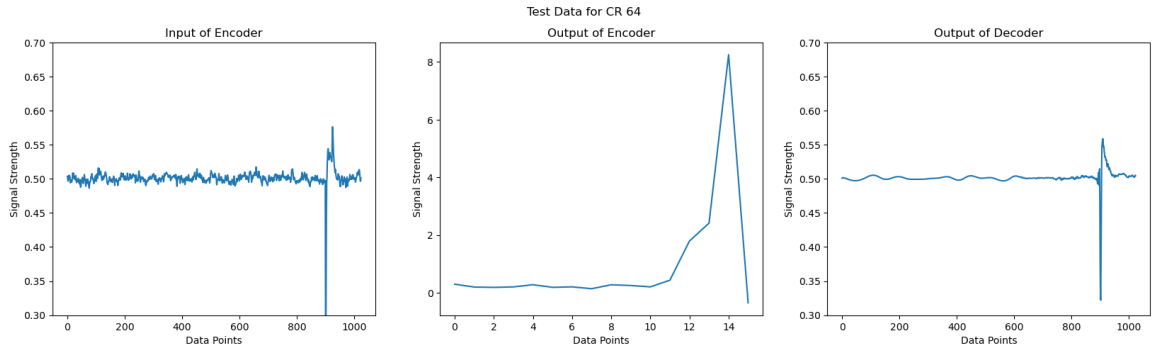
To evaluate the results of the project in detail, we will consider the compression rate 64, as this is the highest compression rate with reasonable results. In Figure 4, we can see 3 example segments of the signal. In Figure 4a, we can observe that a spike is preserved and that the noise in the signal is visibly reduced. This implies that the autoencoder has the ability to correctly recognise a spike in the original data. In Figure 4b, we see a segment of original data with no spikes and only noise. We can observe that output of the Decoder shows a significant smoothing of the signal. Therefore, we can postulate that the autoencoder can remove noise. Lastly, looking at Figure 4c, we can see how the autoencoder behaves, when the signal shows spikes of significant signal strength. While the noise is still vastly removed, we can also observe that the spike becomes smaller regarding the signal strength. It should also be noted that the output of the Encoder does not necessarily have to be in the range $[0, 1]$. We also observe that the output of the Encoder does not fully reflect the shapes that can be found in both the input of the Encoder and the output of the Decoder.



(a) Example 1



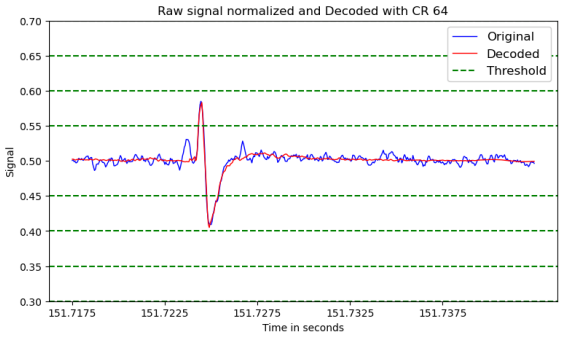
(b) Example 2



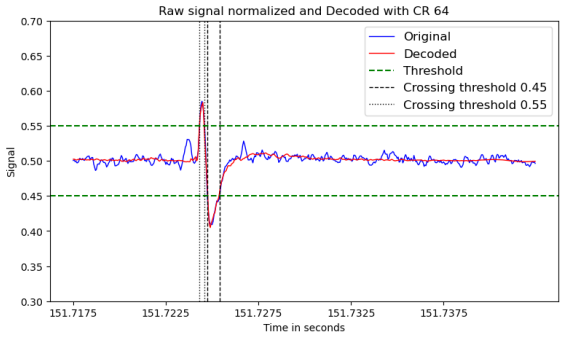
(c) Example 3

Figure 4: Every row of images shows how the autoencoder processes a given segment of the normalized raw signal (for a specific time section). For a compression rate 64, the amount of input and output data points is equal, as described in Section 2. Thus the output of the Encoder consists of $\frac{1024}{64} = 16$ data points.

In Figure 5, a section of the original signal and the decoded signal can be observed. Figure 5a shows a few possible thresholds to evaluate the crossing point analysis, as described in Section 3.3.1. In Figure 5b, we can see an example for detection of crossings for 2 specific thresholds 0.45 and 0.55. The vertical lines indicate a crossing with the 2 chosen thresholds. On the basis of this information, we can infer the spike position in the original and decoded dataset. For this specific compression rate and example, you can see that the crossings happen in the same place, and thus we have preserved crossings.



(a) Overview of a few possible thresholds. We choose increments of 0.05 between the thresholds. The threshold 0.5 is omitted, as it is too close to the mean of the signal and thus does not reflect any meaningful information.

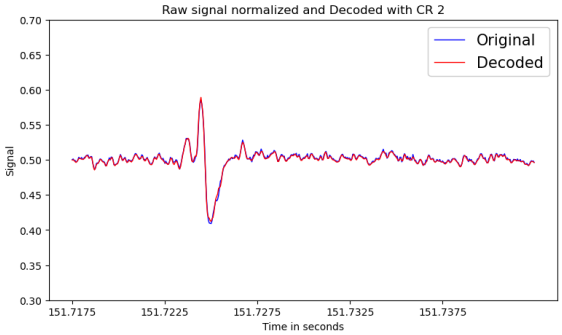


(b) Example for the crossing points analysis with the thresholds 0.45 and 0.55.

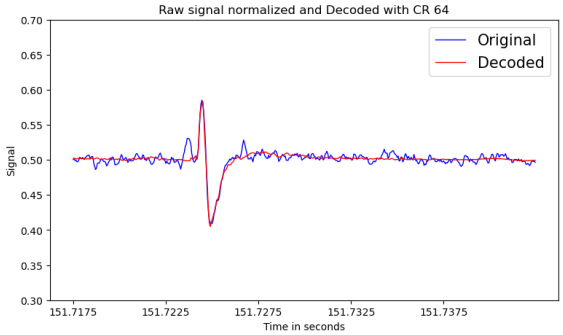
Figure 5: Section of the raw normalized signal overlaid with the output of the autoencoder with CR 64 for the same segment of signal.

4.2 Comparison of Compression Rates

In the following we evaluate the influence of compression rates on the results, as described in Figure 3. In Figure 6, we can see the differences in the decoded signal between compression rate 2 and compression rate 64 for the same section of input data. We can observe that, for compression rate 2 the shape of the original data is very well preserved (see Figure 6a). This includes both the spike and the noise in the signal. For compression rate 64 (see Figure 6b), we can see that the preservation is visibly worse, while the spike shape is being maintained, the structures that form the noise get simplified and smoothened. This is in line with our expectations regarding different compression rates for the used autoencoders.



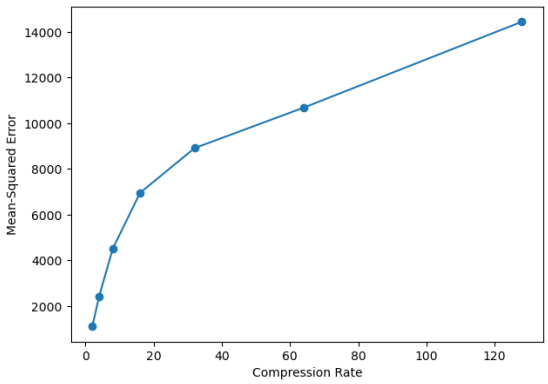
(a) Example with compression rate 2



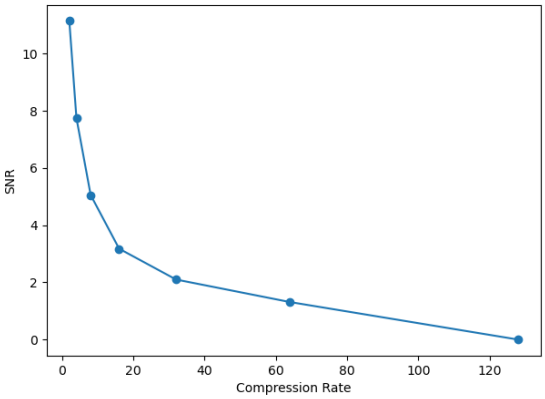
(b) Example with compression rate 64

Figure 6: A section of the raw normalized signal overlaid with the output of autoencoder with different compression rates

The observation that higher compression rates preserve the structures in the original signal worse is also supported by Figure 7a and Figure 7b. Here, we can see that the MSE grows approximately logarithmically with increasing compression rate. We can also see that the SNR decreases with an increase in compression rate. This is also in line with our expectations, as a higher compression rate results in less information being able to pass through the bottleneck layer.



(a) MSE, as defined in Equation (5), for the different used compression rates.

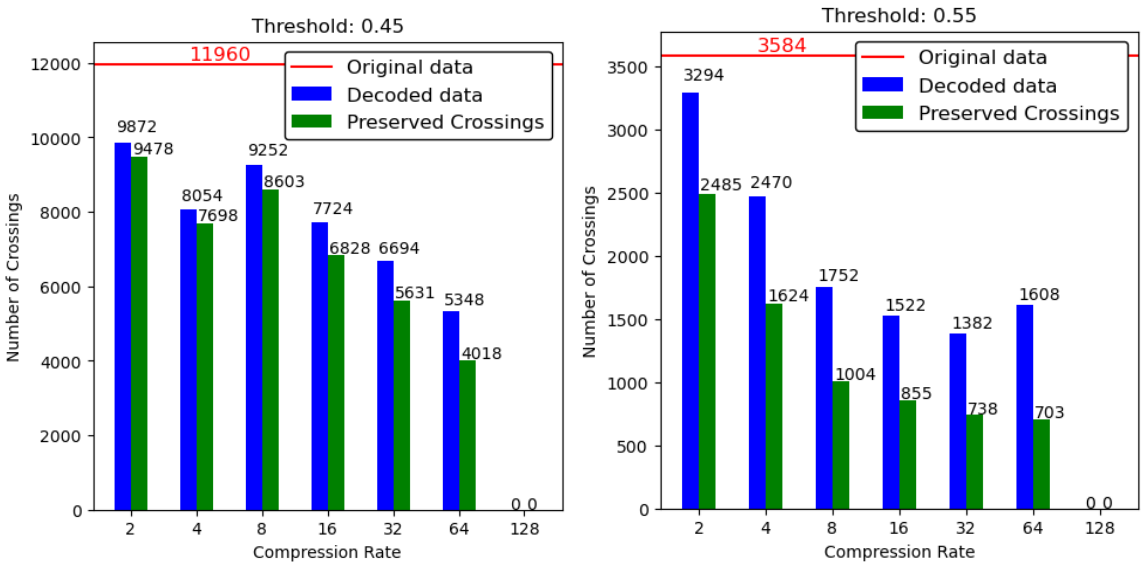


(b) SNR, as defined in Equation (6), for the different used compression rates.

Figure 7: The MSE and the SNR is calculated between the raw input and the output signal of the respective autoencoders implementing the shown compression rates for the whole test subset

In Figure 8, we can observe the calculated number of crossings for two different exemplary thresholds for the whole test subset. The number of preserved crossings can help us understand the quality of spike preservation and thus spike detection of the respective autoencoder. Note that the crossings detected in the decoded data are not always the same as the ones in the original data. This is reflected in the difference between the number of decoded data crossings and preserved crossings. Firstly, we observe that the original data always has a higher number of crossings than any decoded data. This suggests that the use of autoencoders in our case leads to a loss of information about the

signal. In Figure 8a, we can see that for threshold 0.45 compression rate 4 yields worse results than the next higher evaluated compression rate of 8. This is a surprising observation. However, it can be generally said that with an increase in compression rate, we achieve worse results regarding the number of decoded crossings, as well as preserved crossings. For threshold 0.55 (see Figure 8b), the decreasing number of crossings for increasing compression rates is maintained. This is also consistent with other thresholds, as can be seen in Figure 11. Also note that compression rate 64 is the highest compression rate which yields sensible results. Compression rate 128 does not include any crossings and thus does not preserve any information from the original dataset.



(a) Example for the threshold 0.45

(b) Example for the threshold 0.55

Figure 8: Bar charts for different compression rates illustrating the number of crossings in the original data, the decoded data and the number of preserved crossings, as described in Section 3.3.1

.

Lastly, we analyze the percentage of correctly preserved crossings for different thresholds and compression rates, as seen in Figure 9. Generally, it can be observed that a higher compression rate yields a lower percentage of correctly preserved crossings, with the autoencoder implementing compression rate 128 maintaining 0 correctly preserved crossings for every threshold. It is important to notice that the behavior of the autoencoder implementing compression rate 4 shows surprising character. Specifically, for thresholds less than 0.5, it yields worse results than expected. The percentages of preserved crossings are lower than the ones for compression rate 8 for thresholds less than 0.5. On the other hand, the autoencoder for compression rate 4 outperforms the one for compression rate 8 for thresholds bigger than 0.5. Generally, it can again be said, that with an increase in compression rate, the autoencoder performs worse. A further observation is that for thresholds close to the edges of the interval $[0, 1]$ our respectively used autoencoders yield worse results. This is expected, as we have fewer crossings in total with such extreme values. Further, this supports our earlier observation in Section 4.1 that spikes become smaller regarding the signal strength when processed with the used autoencoders. This claim is also supported by the threshold analysis images provided in Figure 11.

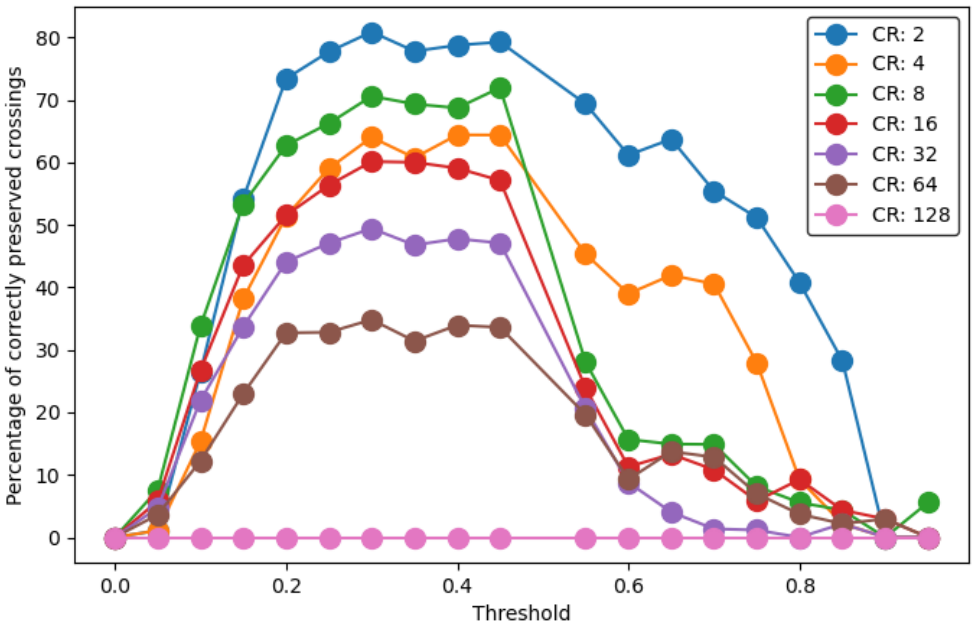


Figure 9: Line graph showing the percentage of correctly preserved crossings, as defined in Equation (8). Compression rate values are represented by different colors

.

5 Conclusion

In this report, we have introduced microneurography, a technique used to generate neural data. When analyzing this neural data there are 2 problems, namely spike detection and spike sorting. The low signal-to-noise ratio of the signal and the high variability of spike shapes is problematic, because of that standard sorting and clustering algorithms for this data can not be used. We have introduced the notion of autoencoders, particularly the special case of a convolutional autoencoder. The compression rate of an autoencoder has been defined. It influences the flow of information in an autoencoder. To check the feasibility of using autoencoders to eliminate noise in the neural data, as well as detecting spikes we have implemented 7 autoencoders implementing different compression rates. To effectively evaluate the performance of these autoencoders we have used standard metrics such as MSE and SNR, as well as introduced the notion of thresholds. Thresholds allow us to count the crossings of a signal, which is the number of times this signal crosses the given threshold, as well as detect the exact location of the crossings. Using this, we define the notion of a preserved crossing, which is a crossing that happens in the input signal, and the output signal of the autoencoder, at the same position in the signal. Empirically, as well as using these metrics we have seen that the used convolutional autoencoders can filter out the noise and detect spikes. We have presented that with an increase in compression rate, the original signal gets reconstructed less closely by the respective autoencoder. Additionally, using the total number of crossings and the number of preserved crossings, we have shown that the used autoencoders are not able to reconstruct every

spike. Consistently, the number of crossings in the decoded data was less than the number of crossings in the original data. With an increase in compression rate, the number of crossings, as well as the number of preserved crossings decreases.

6 Future Work

What is left to show is how the used autoencoders perform on human data and how the presented noise reduction and spike preservation assist in spike detection and spike sorting. We use the MSE between the input data and the output data to train and optimize the used autoencoders. It might be possible to define a more meaningful metric in terms of what the goal of the project is, to train the autoencoders. Further, we shuffled the data segments used for training. While this approach is typically done in the context of using a CAE, it is a non-trivial decision when working with time series data and requires further investigation. As presented in Section 3.2 we have used several simplifications. Changing the used number of filters, the kernel size, the stride, or the padding for any layer may increase the performance. Finally, a larger number of compression rates and architectures might have to be tested to conclude what the best-performing autoencoder is.

References

[1] Amd development container. `https://hub.docker.com/layers/rocm/tensorflow/rocm5.2.0-tf2.7-dev/images/sha256-8e2d6cb3bf997574c87dc9faf021bd03bcf753b6ebd64ace69eced1f59dba62e?context=explore`.

[2] Rwth-gitlab repository. `https://git.rwth-aachen.de/Philip.Kroll/ConvAutoencoder`.

[3] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[4] Forest Agostinelli, Matthew Hoffman, Peter Sadowski, and Pierre Baldi. Learning activation functions to improve deep neural networks. *arXiv preprint arXiv:1412.6830*, 2014.

[5] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. Understanding of a convolutional neural network. In *2017 international conference on engineering and technology (ICET)*, pages 1–6. Ieee, 2017.

[6] Pierre Baldi. Autoencoders, unsupervised learning, and deep architectures. In Isabelle Guyon, Gideon Dror, Vincent Lemaire, Graham Taylor, and Daniel Silver, editors, *Proceedings of ICML Workshop on Unsupervised and Transfer Learning*, volume 27 of *Proceedings of Machine Learning Research*, pages 37–49, Bellevue, Washington, USA, 02 Jul 2012. PMLR.

[7] Jason Brownlee. *Deep learning with Python: develop deep learning models on Theano and TensorFlow using Keras*. Machine Learning Mastery, 2016.

[8] Zhengxue Cheng, Heming Sun, Masaru Takeuchi, and Jiro Katto. Deep convolutional autoencoder-based lossy image compression. In *2018 Picture Coding Symposium (PCS)*, pages 253–257. IEEE, 2018.

[9] François Chollet et al. Keras. `https://keras.io`, 2015.

[10] Joyce T Da Silva and David A Seminowicz. Neuroimaging of pain in animal models: a review of recent literature. *Pain reports*, 4(4), 2019.

[11] Kimia Dinashi, Ali Ameri, Mohammad Ali Akhaee, Kevin Englehart, and Erik Scheme. Compression of emg signals using deep convolutional autoencoders. *IEEE Journal of Biomedical and Health Informatics*, 26(7):2888–2897, 2022.

[12] Christopher Eccleston, SJ Morley, and AC de C Williams. Psychological approaches to chronic pain management: evidence and challenges. *British journal of anaesthesia*, 111(1):59–63, 2013.

[13] Chi Geng and JianXin Song. Human action recognition based on convolutional neural networks with a convolutional auto-encoder. In *2015 5th International Conference on Computer Sciences and Automation Engineering (ICCSAE 2015)*, pages 933–938. Atlantis Press, 2016.

[14] Aurélien Géron. Hands-on machine learning with scikit-learn and tensorflow: Concepts. *Tools, and Techniques to build intelligent systems*, 2017.

[15] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[16] Qi Meng, Wei Chen, Yue Wang, Zhi-Ming Ma, and Tie-Yan Liu. Convergence analysis of distributed stochastic gradient descent with shuffling. *Neurocomputing*, 337:46–57, 2019.

[17] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015.

[18] Hannah Robins, Victoria Perron, Lauren C Heathcote, and Laura E Simons. Pain neuroscience education: state of the art and application in pediatrics. *Children*, 3(4):43, 2016.

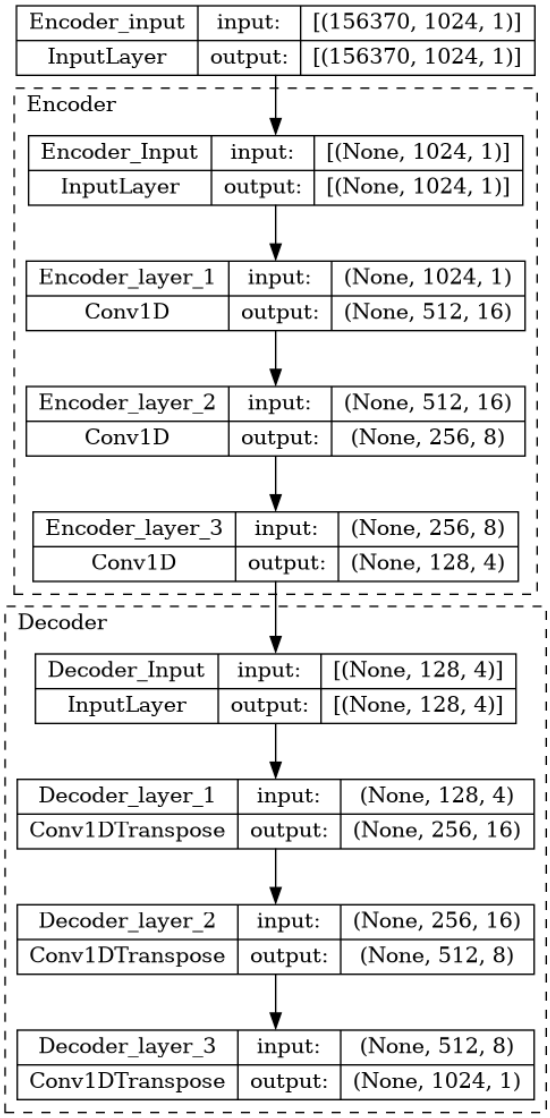
[19] Tobias Schmidt-Wilcke. Neuroimaging of chronic pain. *Best practice & research Clinical rheumatology*, 29(1):29–41, 2015.

[20] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*, 2014.

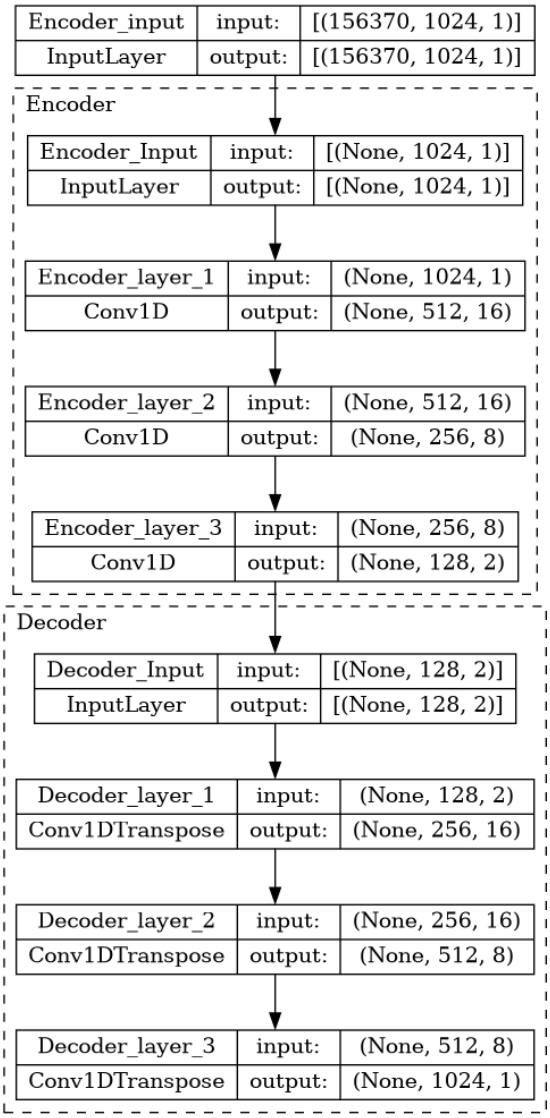
[21] Ake B Vallbo, Karl-Erik Hagbarth, and B Gunnar Wallin. Microneurography: how the technique developed and its role in the investigation of the sympathetic nervous system. *Journal of applied physiology*, 96(4):1262–1269, 2004.

[22] Åke Bernhard Vallbo. Microneurography: how it started and how it works. *Journal of neurophysiology*, 120(3):1415–1427, 2018.

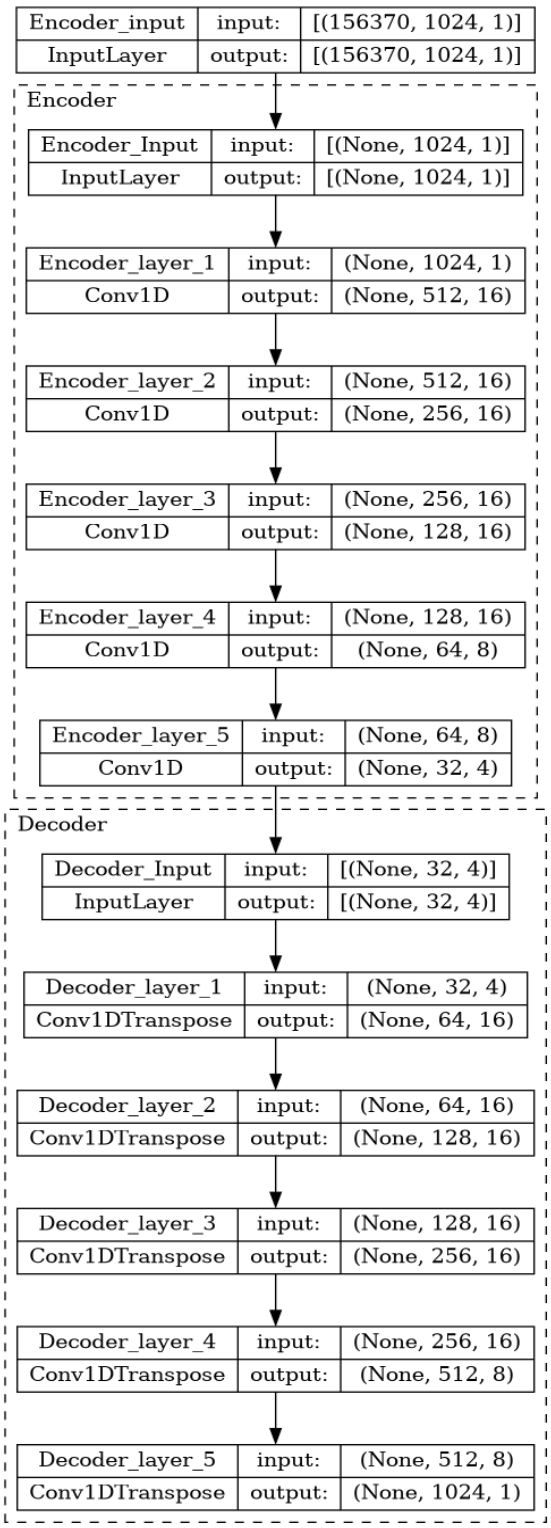
Appendices


Encoder_input | input: | [(156370, 1024, 1)]
InputLayer | output: | [(156370, 1024, 1)]
Encoder
Encoder_Input | input: | [(None, 1024, 1)]
InputLayer | output: | [(None, 1024, 1)]
Encoder_layer_1 | input: | (None, 1024, 1)
Conv1D | output: | (None, 512, 16)
Encoder_layer_2 | input: | (None, 512, 16)
Conv1D | output: | (None, 256, 8)
Encoder_layer_3 | input: | (None, 256, 8)
Conv1D | output: | (None, 128, 4)
Decoder
Decoder_Input | input: | [(None, 128, 4)]
InputLayer | output: | [(None, 128, 4)]
Decoder_layer_1 | input: | (None, 128, 4)
Conv1DTranspose | output: | (None, 256, 16)
Decoder_layer_2 | input: | (None, 256, 16)
Conv1DTranspose | output: | (None, 512, 8)
Decoder_layer_3 | input: | (None, 512, 8)
Conv1DTranspose | output: | (None, 1024, 1)

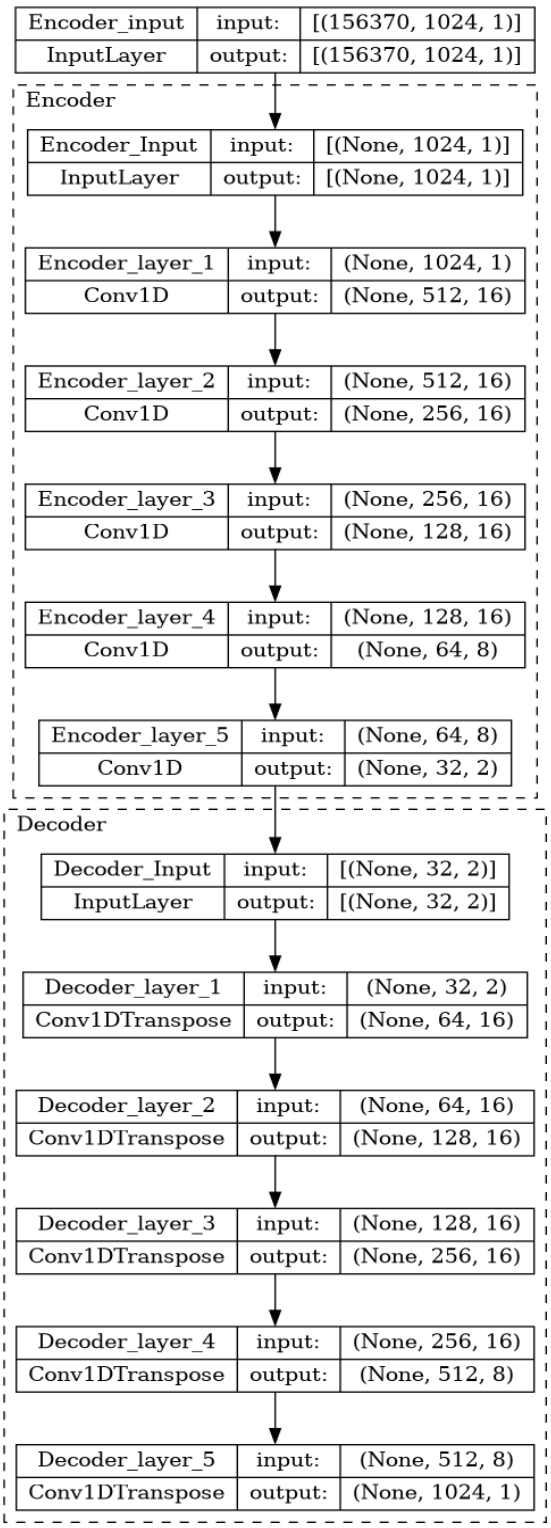
(a) CR 2


Encoder_input | input: | [(156370, 1024, 1)]
InputLayer | output: | [(156370, 1024, 1)]
Encoder
Encoder_Input | input: | [(None, 1024, 1)]
InputLayer | output: | [(None, 1024, 1)]
Encoder_layer_1 | input: | (None, 1024, 1)
Conv1D | output: | (None, 512, 16)
Encoder_layer_2 | input: | (None, 512, 16)
Conv1D | output: | (None, 256, 8)
Encoder_layer_3 | input: | (None, 256, 8)
Conv1D | output: | (None, 128, 2)
Decoder
Decoder_Input | input: | [(None, 128, 2)]
InputLayer | output: | [(None, 128, 2)]
Decoder_layer_1 | input: | (None, 128, 2)
Conv1DTranspose | output: | (None, 256, 16)
Decoder_layer_2 | input: | (None, 256, 16)
Conv1DTranspose | output: | (None, 512, 8)
Decoder_layer_3 | input: | (None, 512, 8)
Conv1DTranspose | output: | (None, 1024, 1)

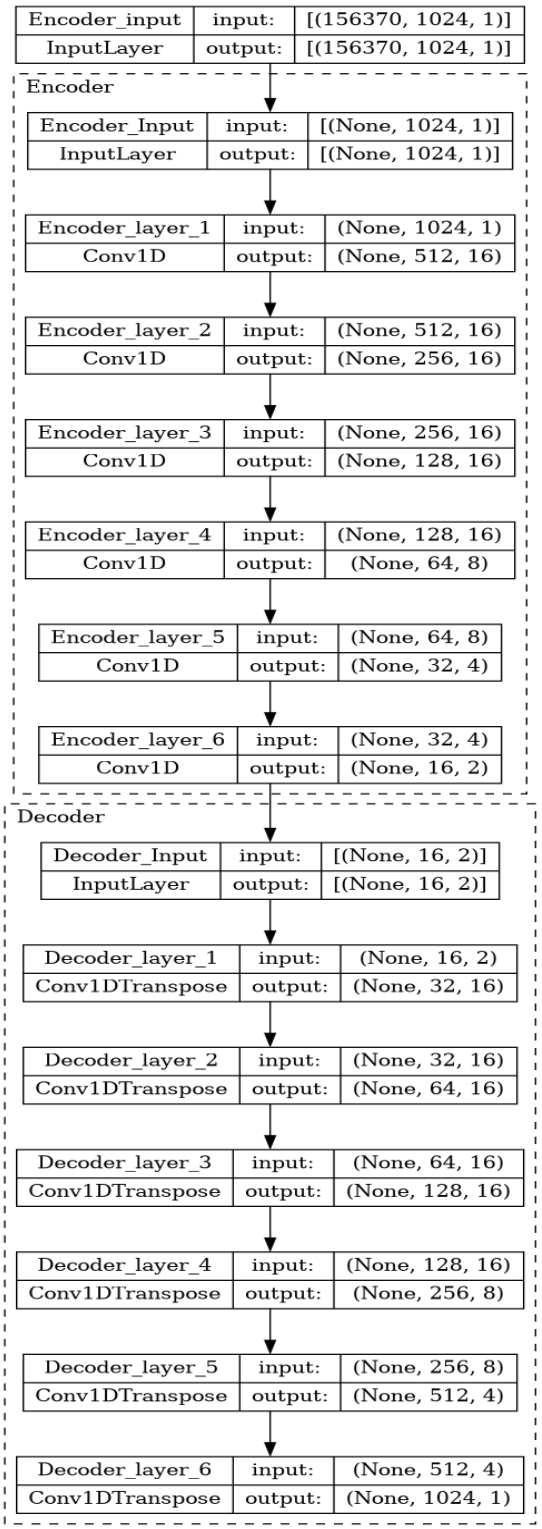
(b) CR 4


Encoder_input | input: | [(156370, 1024, 1)]
InputLayer | output: | [(156370, 1024, 1)]
Encoder
Encoder_Input | input: | [(None, 1024, 1)]
InputLayer | output: | [(None, 1024, 1)]
Encoder_layer_1 | input: | (None, 1024, 1)
Conv1D | output: | (None, 512, 16)
Encoder_layer_2 | input: | (None, 512, 16)
Conv1D | output: | (None, 256, 16)
Encoder_layer_3 | input: | (None, 256, 16)
Conv1D | output: | (None, 128, 16)
Encoder_layer_4 | input: | (None, 128, 16)
Conv1D | output: | (None, 64, 8)
Encoder_layer_5 | input: | (None, 64, 8)
Conv1D | output: | (None, 32, 4)
Decoder
Decoder_Input | input: | [(None, 32, 4)]
InputLayer | output: | [(None, 32, 4)]
Decoder_layer_1 | input: | (None, 32, 4)
Conv1DTranspose | output: | (None, 64, 16)
Decoder_layer_2 | input: | (None, 64, 16)
Conv1DTranspose | output: | (None, 128, 16)
Decoder_layer_3 | input: | (None, 128, 16)
Conv1DTranspose | output: | (None, 256, 16)
Decoder_layer_4 | input: | (None, 256, 16)
Conv1DTranspose | output: | (None, 512, 8)
Decoder_layer_5 | input: | (None, 512, 8)
Conv1DTranspose | output: | (None, 1024, 1)

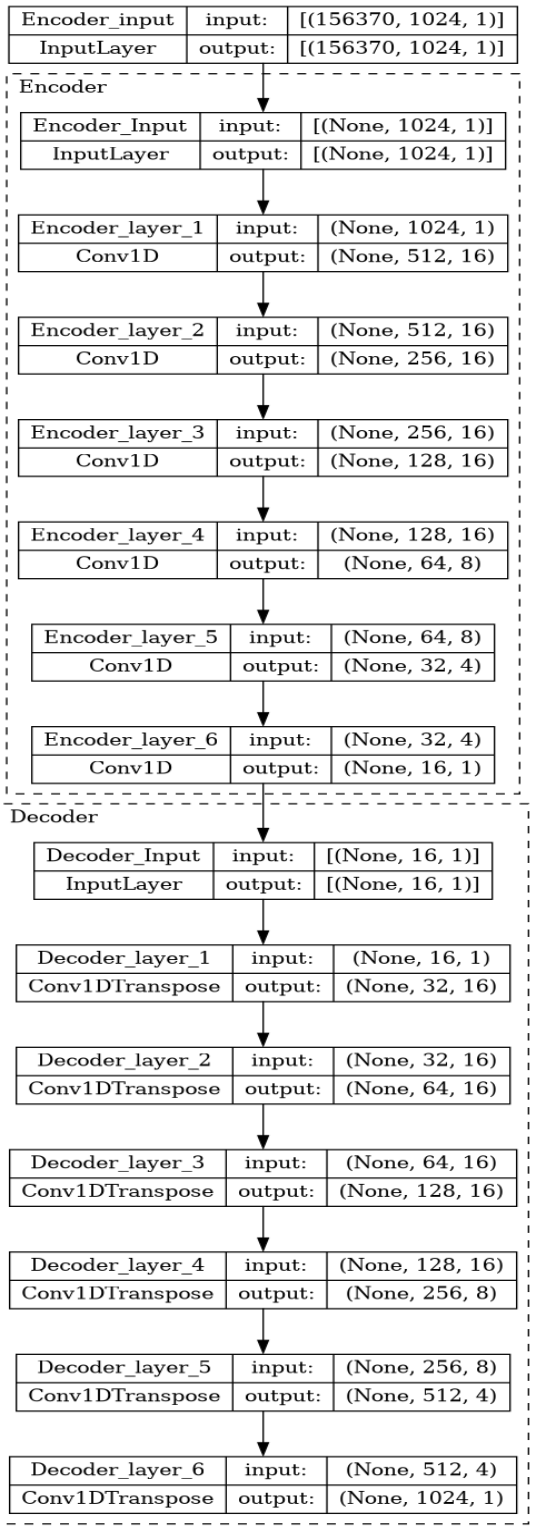
(c) CR 8


Encoder_input | input: | [(156370, 1024, 1)]
InputLayer | output: | [(156370, 1024, 1)]
Encoder
Encoder_Input | input: | [(None, 1024, 1)]
InputLayer | output: | [(None, 1024, 1)]
Encoder_layer_1 | input: | (None, 1024, 1)
Conv1D | output: | (None, 512, 16)
Encoder_layer_2 | input: | (None, 512, 16)
Conv1D | output: | (None, 256, 16)
Encoder_layer_3 | input: | (None, 256, 16)
Conv1D | output: | (None, 128, 16)
Encoder_layer_4 | input: | (None, 128, 16)
Conv1D | output: | (None, 64, 8)
Encoder_layer_5 | input: | (None, 64, 8)
Conv1D | output: | (None, 32, 2)
Decoder
Decoder_Input | input: | [(None, 32, 2)]
InputLayer | output: | [(None, 32, 2)]
Decoder_layer_1 | input: | (None, 32, 2)
Conv1DTranspose | output: | (None, 64, 16)
Decoder_layer_2 | input: | (None, 64, 16)
Conv1DTranspose | output: | (None, 128, 16)
Decoder_layer_3 | input: | (None, 128, 16)
Conv1DTranspose | output: | (None, 256, 16)
Decoder_layer_4 | input: | (None, 256, 16)
Conv1DTranspose | output: | (None, 512, 8)
Decoder_layer_5 | input: | (None, 512, 8)
Conv1DTranspose | output: | (None, 1024, 1)

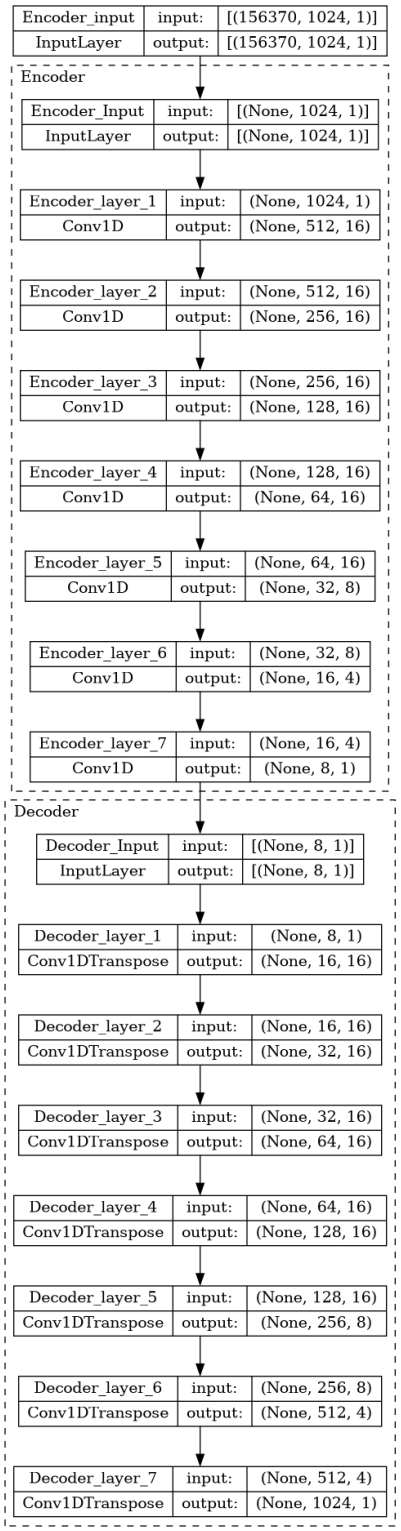
(d) CR 16


Encoder_input | input: | [(156370, 1024, 1)]
InputLayer | output: | [(156370, 1024, 1)]
Encoder
Encoder_Input | input: | [(None, 1024, 1)]
InputLayer | output: | [(None, 1024, 1)]
Encoder_layer_1 | input: | (None, 1024, 1)
Conv1D | output: | (None, 512, 16)
Encoder_layer_2 | input: | (None, 512, 16)
Conv1D | output: | (None, 256, 16)
Encoder_layer_3 | input: | (None, 256, 16)
Conv1D | output: | (None, 128, 16)
Encoder_layer_4 | input: | (None, 128, 16)
Conv1D | output: | (None, 64, 8)
Encoder_layer_5 | input: | (None, 64, 8)
Conv1D | output: | (None, 32, 4)
Encoder_layer_6 | input: | (None, 32, 4)
Conv1D | output: | (None, 16, 2)
Decoder
Decoder_Input | input: | [(None, 16, 2)]
InputLayer | output: | [(None, 16, 2)]
Decoder_layer_1 | input: | (None, 16, 2)
Conv1DTranspose | output: | (None, 32, 16)
Decoder_layer_2 | input: | (None, 32, 16)
Conv1DTranspose | output: | (None, 64, 16)
Decoder_layer_3 | input: | (None, 64, 16)
Conv1DTranspose | output: | (None, 128, 16)
Decoder_layer_4 | input: | (None, 128, 16)
Conv1DTranspose | output: | (None, 256, 8)
Decoder_layer_5 | input: | (None, 256, 8)
Conv1DTranspose | output: | (None, 512, 4)
Decoder_layer_6 | input: | (None, 512, 4)
Conv1DTranspose | output: | (None, 1024, 1)

(e) CR 32

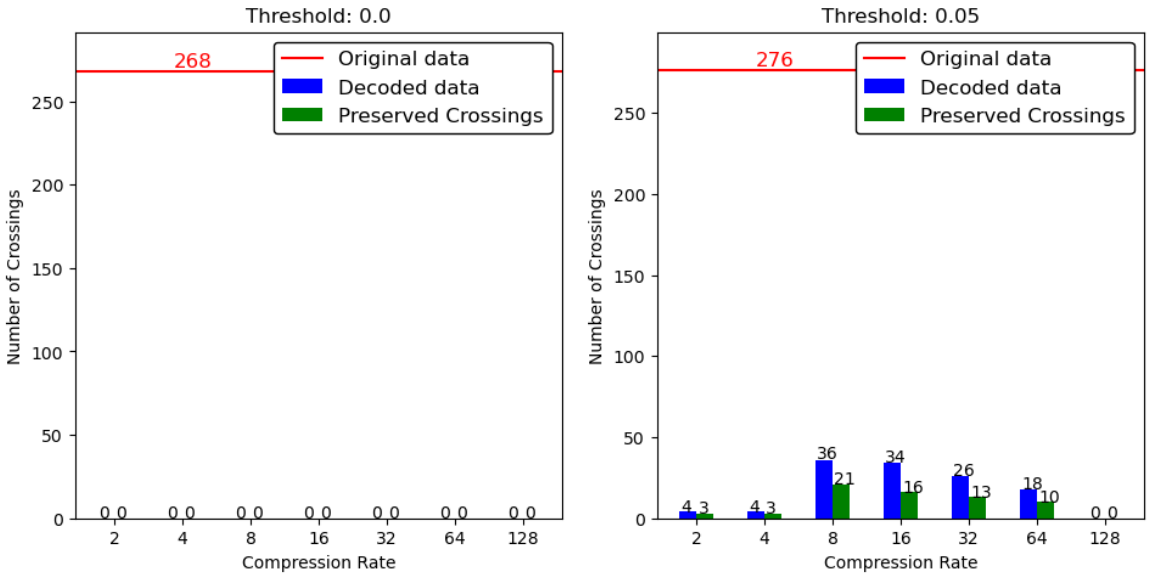

Encoder_input | input: | [(156370, 1024, 1)]
InputLayer | output: | [(156370, 1024, 1)]
Encoder
Encoder_Input | input: | [(None, 1024, 1)]
InputLayer | output: | [(None, 1024, 1)]
Encoder_layer_1 | input: | (None, 1024, 1)
Conv1D | output: | (None, 512, 16)
Encoder_layer_2 | input: | (None, 512, 16)
Conv1D | output: | (None, 256, 16)
Encoder_layer_3 | input: | (None, 256, 16)
Conv1D | output: | (None, 128, 16)
Encoder_layer_4 | input: | (None, 128, 16)
Conv1D | output: | (None, 64, 8)
Encoder_layer_5 | input: | (None, 64, 8)
Conv1D | output: | (None, 32, 4)
Encoder_layer_6 | input: | (None, 32, 4)
Conv1D | output: | (None, 16, 1)
Decoder
Decoder_Input | input: | [(None, 16, 1)]
InputLayer | output: | [(None, 16, 1)]
Decoder_layer_1 | input: | (None, 16, 1)
Conv1DTranspose | output: | (None, 32, 16)
Decoder_layer_2 | input: | (None, 32, 16)
Conv1DTranspose | output: | (None, 64, 16)
Decoder_layer_3 | input: | (None, 64, 16)
Conv1DTranspose | output: | (None, 128, 16)
Decoder_layer_4 | input: | (None, 128, 16)
Conv1DTranspose | output: | (None, 256, 8)
Decoder_layer_5 | input: | (None, 256, 8)
Conv1DTranspose | output: | (None, 512, 4)
Decoder_layer_6 | input: | (None, 512, 4)
Conv1DTranspose | output: | (None, 1024, 1)

(f) CR 64

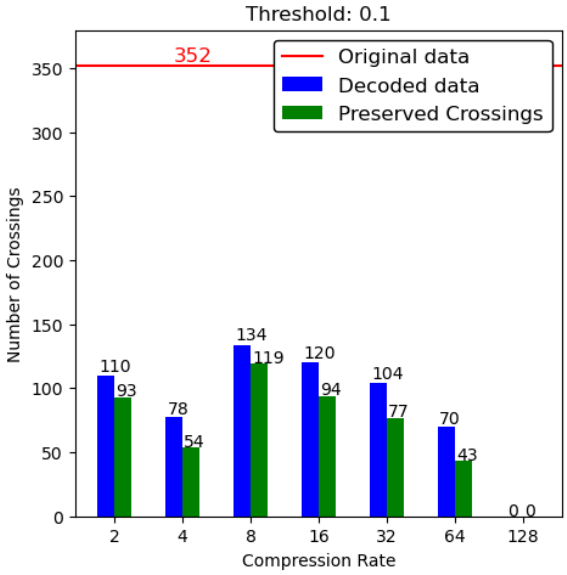


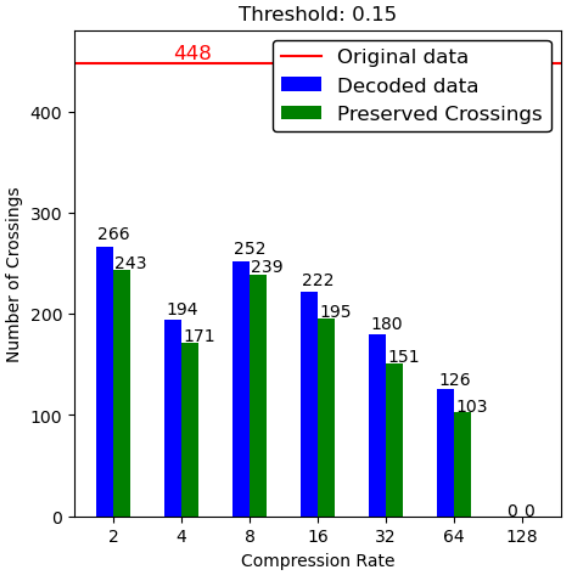
(g) CR 128

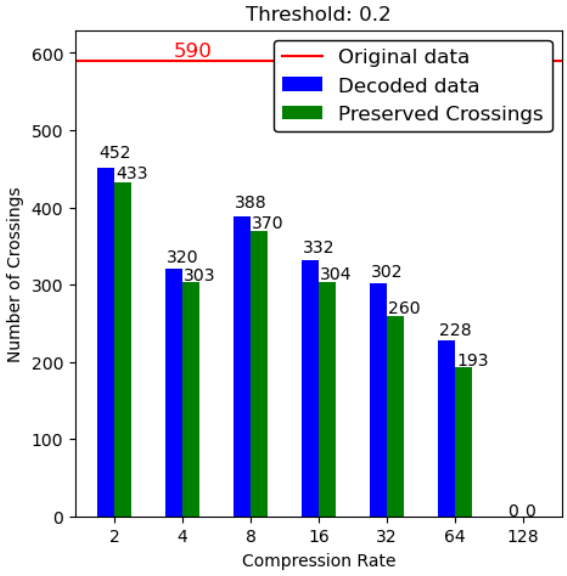
Figure 10: Used Convolutional Autoencoder Architectures

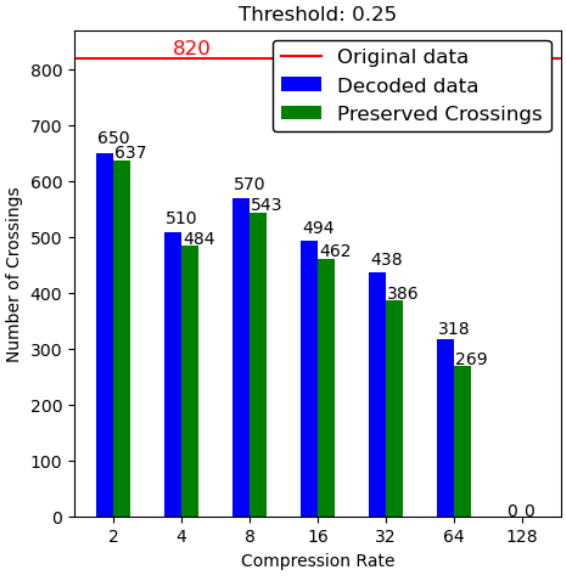

Threshold: 0.0
Original data
Decoded data
Preserved Crossings
268
Number of Crossings
Compression Rate
2 4 8 16 32 64 128
0 0 0 0 0 0 0

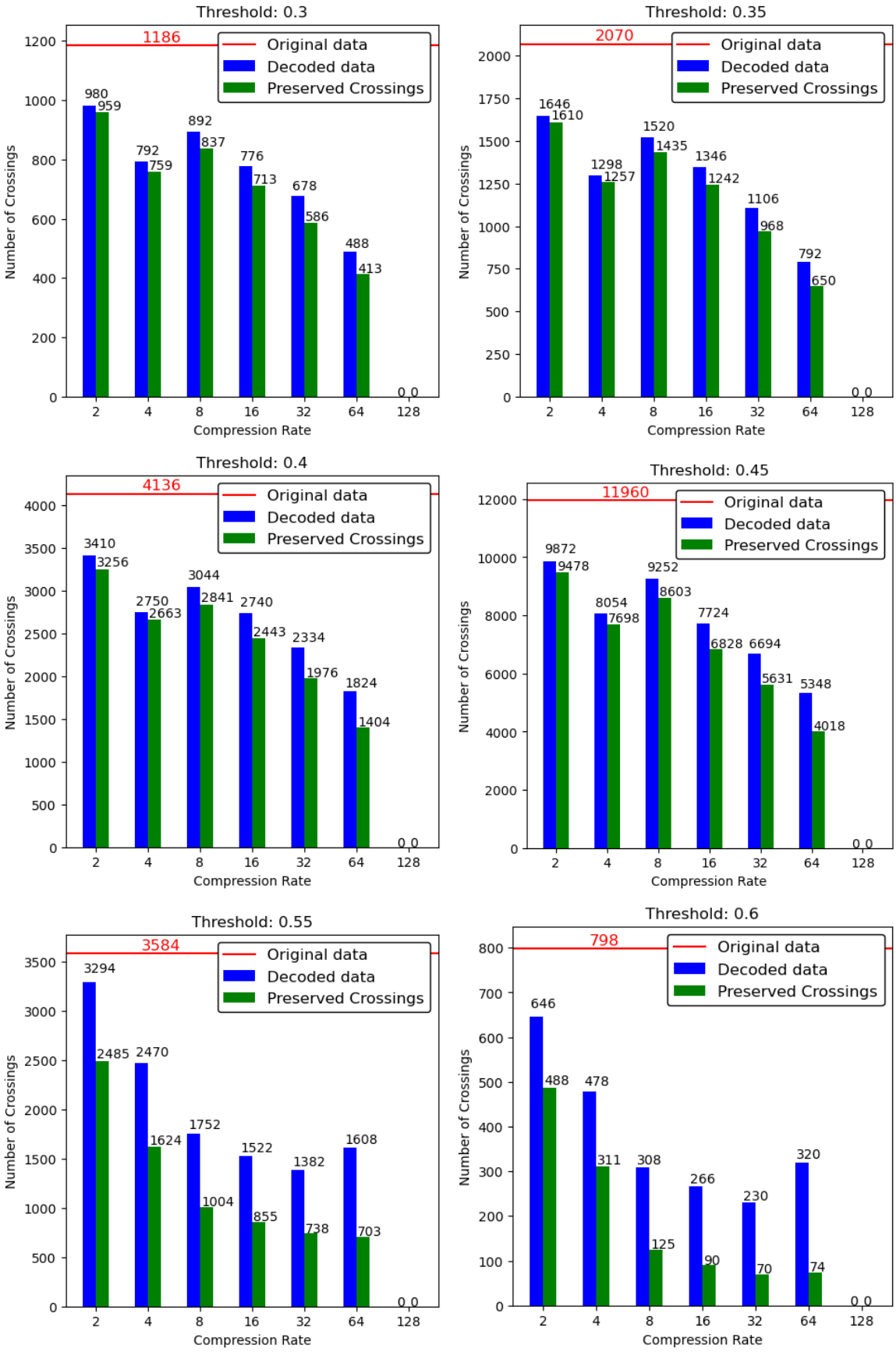
Threshold: 0.05
Original data
Decoded data
Preserved Crossings
276
Number of Crossings
Compression Rate
2 4 8 16 32 64 128
4 3 4 3 36 21 34 16 26 13 18 10 0 0

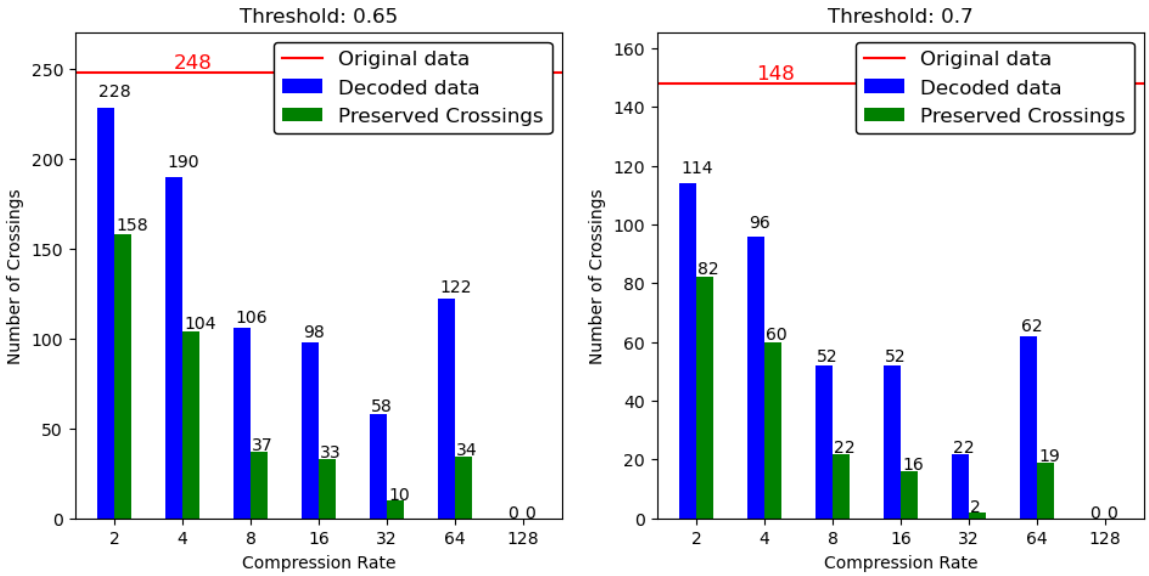

Threshold: 0.1
Original data
Decoded data
Preserved Crossings
352
Number of Crossings
Compression Rate
2 4 8 16 32 64 128
110 93 78 54 134 119 120 94 104 77 70 43 0 0


Threshold: 0.15
Original data
Decoded data
Preserved Crossings
448
Number of Crossings
Compression Rate
2 4 8 16 32 64 128
266 243 194 171 252 239 222 195 180 151 126 103 0 0

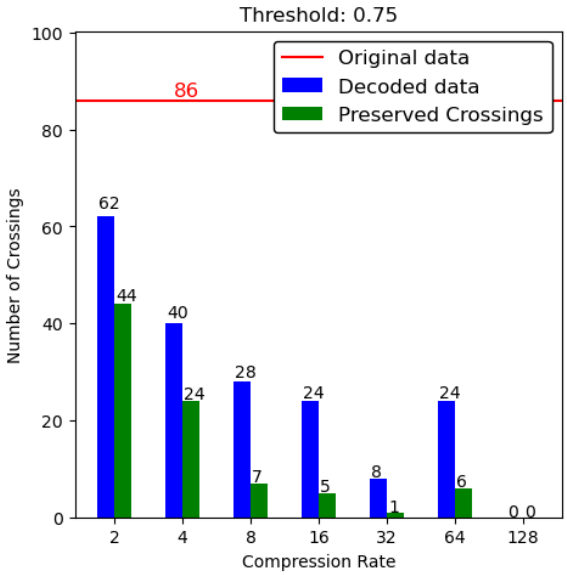

Threshold: 0.2
Original data
Decoded data
Preserved Crossings
590
Number of Crossings
Compression Rate
2 4 8 16 32 64 128
452 433 320 303 388 370 332 304 302 260 228 193 0 0

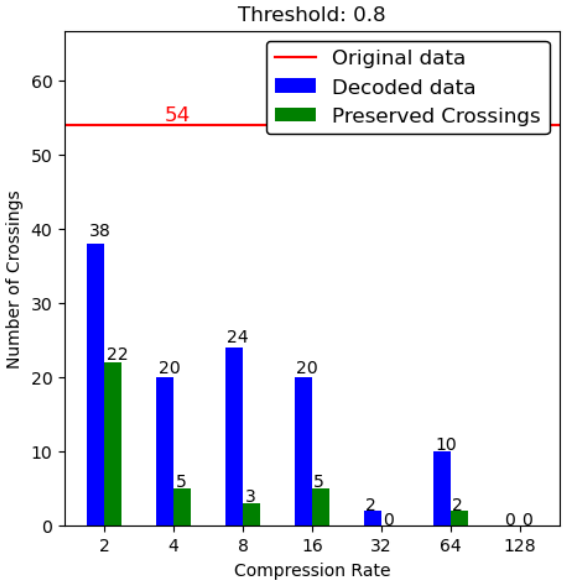

Threshold: 0.25
Original data
Decoded data
Preserved Crossings
820
Number of Crossings
Compression Rate
2 4 8 16 32 64 128
650 637 510 484 570 543 494 462 438 386 318 269 0 0

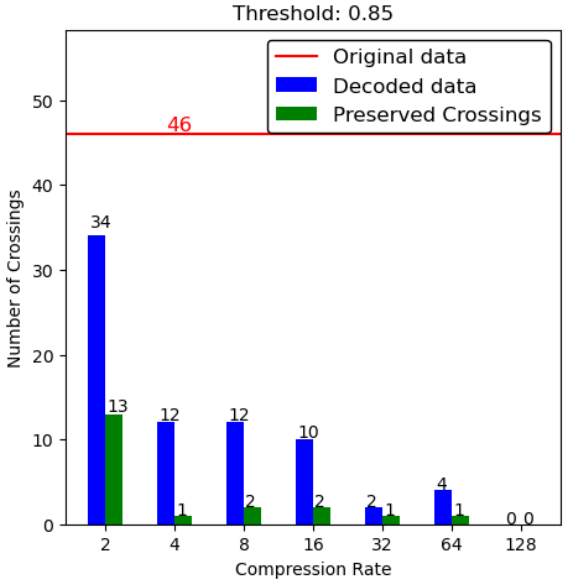

Threshold: 0.3
1186
Original data
Decoded data
Preserved Crossings
Number of Crossings
Compression Rate
980 959
792 759
892 837
776 713
678 586
488 413
0 0
Threshold: 0.35
2070
1646 1610
1298 1257
1520 1435
1346 1242
1106 968
792 650
0 0
Threshold: 0.4
4136
3410 3256
2750 2663
3044 2841
2740 2443
2334 1976
1824 1404
0 0
Threshold: 0.45
11960
9872 9478
8054 7698
9252 8603
7724 6828
6694 5631
5348 4018
0 0
Threshold: 0.55
3584
3294 2485
2470 1624
1752 1004
1522 855
1382 738
1608 703
0 0
Threshold: 0.6
798
646 488
478 311
308 125
266 90
230 70
320 74
0 0

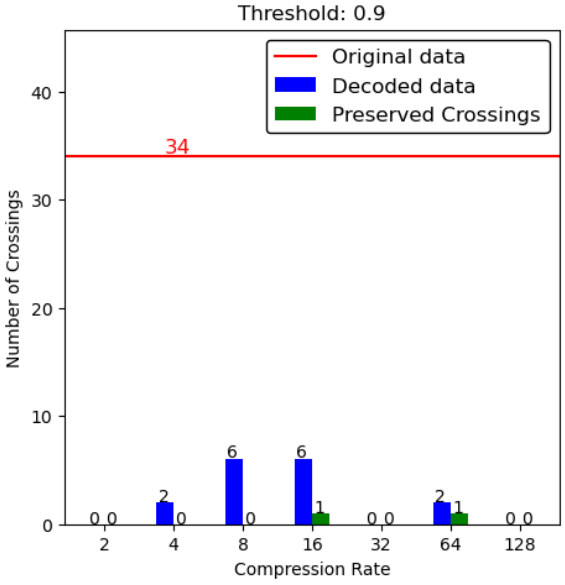

Threshold: 0.65
Original data
Decoded data
Preserved Crossings
248
228
158
190
104
106
37
98
33
58
10
122
34
0 0
Number of Crossings
Compression Rate
2
4
8
16
32
64
128

Threshold: 0.7
Original data
Decoded data
Preserved Crossings
148
114
82
96
60
52
22
52
16
22
2
62
19
0 0
Number of Crossings
Compression Rate
2
4
8
16
32
64
128


Threshold: 0.75
Original data
Decoded data
Preserved Crossings
86
62
44
40
24
28
7
24
5
8
1
24
6
0 0
Number of Crossings
Compression Rate
2
4
8
16
32
64
128


Threshold: 0.8
Original data
Decoded data
Preserved Crossings
54
38
22
20
5
24
3
20
5
2
0
10
2
0 0
Number of Crossings
Compression Rate
2
4
8
16
32
64
128


Threshold: 0.85
Original data
Decoded data
Preserved Crossings
46
34
13
12
1
12
2
10
2
2
1
4
1
0 0
Number of Crossings
Compression Rate
2
4
8
16
32
64
128


Threshold: 0.9
Original data
Decoded data
Preserved Crossings
34
0 0
2
0
6
0
6
1
0 0
2
1
0 0
Number of Crossings
Compression Rate
2
4
8
16
32
64
128

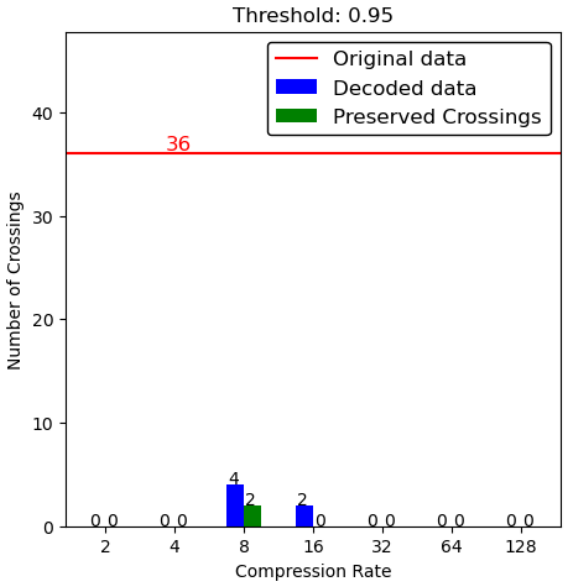


Figure 11: Threshold Analysis