UNIVERSITÄT DUISBURG-ESSEN
FAKULTÄT FÜR INGENIEURWISSENSCHAFTEN
ABTEILUNG INFORMATIK UND ANGEWANDTE KOGNITIONSWISSENSCHAFT

Bachelorarbeit

Implementation and Evaluation of an Autoencoder-based Approach for Feature Extraction of Spike Data

David Lewakis
Matrikelnummer: 3080397
Angewandte Informatik (Bachelor)

Fachgebiet Eingebettete Systeme, Abteilung Informatik
Fakultät für Ingenieurwissenschaften
Universität Duisburg-Essen

19. Juli 2021

Erstgutachter: Prof. Dr. Gregor Schiele
Zweitgutachter: Prof. Dr-Ing. Torben Weis
Zeitraum: 19 April 2021 - 19 Juli 2021

Abstract

Spike data consists of time series signals of neuron activations recorded extracellularly in the brain. Spike sorting is used to filter and distinguish between these activations from different neurons. One step in this procedure is to extract the most relevant features of the data. This thesis proposes the use of machine learning for this feature extraction of spike data using an autoencoder. The autoencoder follows the deep temporal clustering design approach. The training process was performed using a standard reconstruction loss for the autoencoder and additionally with the custom loss also proposed for the deep temporal clustering, where reconstruction and clustering performance are combined. This approach was implemented in an existing spike sorting pipeline. Evaluation of various simulations showed potential for the autoencoder, as it was sometimes able to outperform principal component analysis at compression ratios no greater than 6, even when the number of distinct spikes exceeded 10. However, this was not consistent across all simulations tested, with principal component analysis outperforming the autoencoder even at smaller compression ratios for some. In the future, these shortcomings could be addressed and the technique tested on real data.

Contents

Chapter 1

Introduction

1.1 Motivation

Much about how the human brain works is still unknown today. In recent years, advances in extracellular implants in the brain have made it possible to record more and more neurons in a region simultaneously. This opened up the possibility of understanding networks of neurons and how they work together. The resulting insights can potentially be used for treatments of paralysis, epilepsy, cognitive loss, and memory loss [5].

The challenge in recording data from multiple neurons is to determine when which neuron is active. The process of making this decision is called spike sorting. To simplify the decision of which inputs to mark as the same neuron, the most relevant features can be extracted from the data. Improving this process could potentially provide better spike sorting results and provide more information about the roles of the neurons. In addition, a robust feature extraction technique could make sorting more resistant to noise or changes in neuron connections in the brain. Furthermore, a smaller data size could save resources for the clustering task or save bandwidth when the data is sent away to be clustered in the cloud, for example.

Until recently, this feature extraction step was mainly performed using linear transformation techniques such as principal component analysis or discrete wavelet transforms. Newer approaches using machine learning with autoencoders offer several advantages, such as non-linear representations and the ability to perform multiple small transformations instead of one large one [4].

1.2 Task

In this thesis, the goal is to use machine learning to extract the most relevant features and thereby reduce the dimension of a spike for easier and more efficient clustering. This will be done using a neural network called an autoencoder. This autoencoder

will be integrated into an existing spike sorting pipeline and then evaluated on various simulations of spike data.

1.3 Structure

In the next chapter, the basics of the spike data and spike sorting are explained, as well as some feature extraction and clustering techniques used for these tasks. In the related work, knowledge about spike sorting with and without autoencoders and clustering of similar data is collected. Then, this knowledge is used to design and then implement an autoencoder to perform the feature extraction task. The performance of the autoencoder is evaluated in the next chapter on several spike datasets compared to a standard feature extraction algorithm. Finally, the thesis is concluded and a look at possible future work is presented.

Chapter 2

Basics

In this chapter first the process of spike sorting is described in more detail. After that principal component analysis as a standard approach for feature extraction, as well as autoencoders and how they can be used for this task as well are presented. Lastly the process of clustering data, for example the features extracted from PCA or the autoencoder, is explained together with the widely used unsupervised k-means clustering algorithm.

It will be assumed that the basics of neural networks are familiar, for more information see the Deep Learning book from Goodfellow et al. [6].

2.1 Spike Data

A time series is a set of data ordered in time. Spike data are a type of time series data because they are a signal recorded over a period of time. They are recorded via extracellular (outside the cell) electrodes approximately 13-80 µm in diameter implemented in the brain [5]. These electrodes then record the neurons in their surrounding, typically about 2 to 20. When a neuron is activated, it generates an action potential, which is a spike with a specific shape. For spike sorting, it is generally assumed that all recorded neurons have different action potentials and therefore generate spikes with different shapes. [13]

Figure 2.1 shows an example of how different spikes in a brain might be detected. The neurons in area A are close enough to be detected. The red, blue, and green neurons are very active and therefore can be sorted, while the gray neurons fire sparsely and may be detected so infrequently that they cannot be identified. All neurons in area B are recorded, but not well enough to be sorted individually. Neurons outside B are part of the background noise.

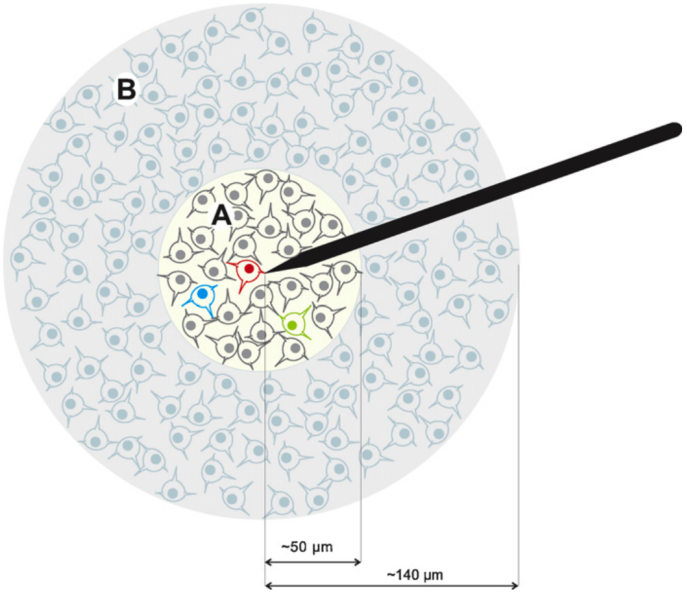

Figure 2.1: Simplified two-dimensional representation of how an electrode might be implanted in the brain tissue between neurons, with each dot representing a neuron. [13]

2.2 Spike Sorting

The process of sorting spike data consists of a pipeline of 6 steps [5].

The first step is now to identify those where an action potential is present. This is done using a threshold that decides what level of activation change should correspond to a potential firing neuron and thus a spike to be found. In Figure 2.2 this can be seen in the raw input, whenever there are tips there is likely a spike present.

The second step is to align the detected spikes according to a relative point of the spike waveforms. Here a filter can be used to remove noise from the spikes.

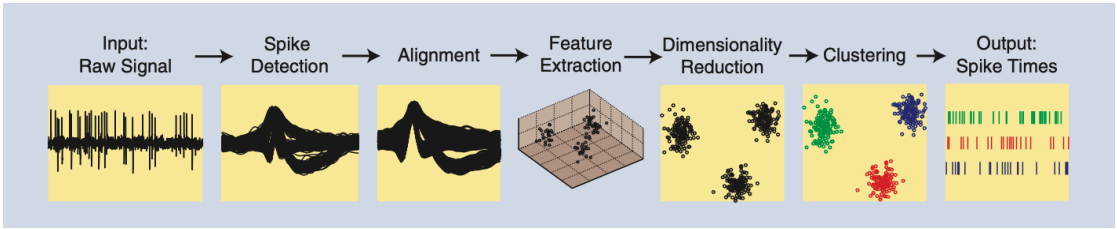

Figure 2.2: Spike Sorting pipeline from Gibson et al. [5]. Each picture showing the result after the labeled step.

Third, feature extraction is performed, in Figure 2.2 it can be seen how 3 features are extracted from each spike.

Fourth, the dimension is further reduced, in the example from 3 to 2 dimensions.

Next, this reduced representation is then clustered to identify spikes that come from the same neuron. For example, 3 different clusters as can be seen in the figure, so there are three different neurons firing 3 different types of spikes.

In the final step, this information can be used to output which neurons fired when, these are called the spike times of the neurons.

2.3 Principal Component Analysis

One of the most common approaches for feature extraction on time series data is principal component analysis (PCA). It produces as many so-called principal components as there are input dimensions in the data, with value differences decreasing with each component, meaning the first component has the most information, the second has the second most, and so on. Depending on how many are selected from these components, the data will be reduced to this size. [3]

The PCA is calculated in 5 main steps [9]. First, the values are normalized to ensure that all input variables have the same contribution, even if they are in different units of measure, for example. To calculate a normalized value z, the original value is subtracted from the mean of its input variables and then divided by the standard deviation of these variables.

$$z = \frac{value - means}{standarddeviation}$$

Second, a covariance matrix is created in which the covariance between all different variables is calculated. An example of this, calculated for 3 input variables, can be found in Figure 2.3. Since the covariance of a variable with itself is just its variance, all variances can be found on the diagonal. Furthermore, since the covariance of x with y is equal to that of y with x, the matrix is symmetric on the diagonal. The values themselves are positive if the two variables are correlated and negative if they are inversely correlated (one decreases as the other increases).

The third step is to use the covariance matrix and calculate the eigenvectors, for each variable an eigenvector is calculated. The eigenvectors of the covariance matrix then indicate the direction of the axes with the greatest variance, these are the principal components. To decide which component has the greatest variance, the eigenvalues of the corresponding eigenvectors can be calculated and sorted in descending order.

$$\begin{bmatrix} Cov(x,x) & Cov(x,y) & Cov(x,z) \\ Cov(y,x) & Cov(y,y) & Cov(y,z) \\ Cov(z,x) & Cov(z,y) & Cov(z,z) \end{bmatrix}$$

Figure 2.3: Covariance Matrix for 3 dimensional data. [9]

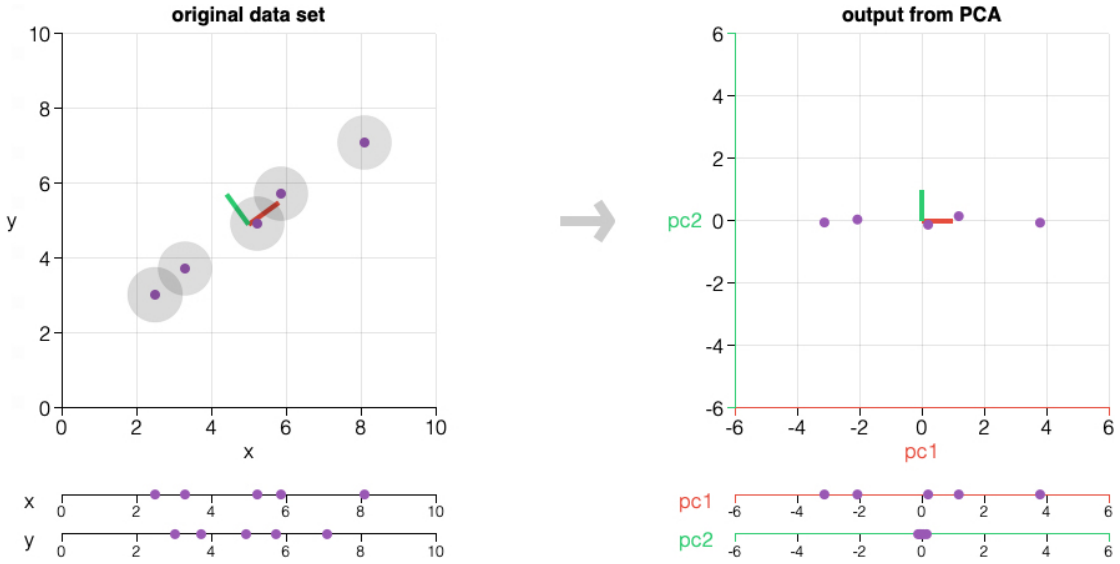

Figure 2.4: Two dimensional example data plotted in a coordinate system, as well as x and y values for each point. The same for both principal components after PCA was performed. [14]

In the fourth step, the feature vector is created depending on the number of components to be used. The eigenvectors of the components that are retained are simply combined to create the feature vector.

Finally, to create the final data set, the transposed normalized data is multiplied by the transposed feature vector. This reduces the size of the data dimension to the size of the components, as this vector multiplication produces data the size of the number of components.

An example can be found in Figure 2.4. Since the data points all lie around a hypothetical diagonal, it can be assumed that one value would be sufficient to obtain most of their information. When PCA is then applied, the second principal component (pc2) shows virtually no difference between the various inputs, while the first component (pc1) contains the information necessary to distinguish the data. Therefore, pc2 could be omitted and the data reduced to one dimension.

2.4 Autoencoder

Autoencoders are a type of neural network consisting of an encoder and a decoder component. The encoder receives an input and generates a code. The decoder should be able to recover the original input from the code as best it can. So, basically, it should "copy" the data. The encoder and decoder can have any number of layers, the simplest versions have a synchronous encoder and decoder.

Like other neural networks, autoencoder layers have a number of nodes, with nodes from one layer connected to nodes from the next. The weights between these connections are adjusted during a training phase using a set of training data. During training, an attempt is made to minimize a loss function, for example, in the case of autoencoders, the mean square error (MSE) between the input and output differences. Often, an autoencoder is designed in such a way that it is nearly impossible to reproduce the input completely, since the goal is to obtain additional information from the process.

Since autoencoders, like most neural networks, require a large amount of training, the time required for this should not be underestimated. Additionally, this means that the amount of training data must be relatively high. Furthermore, autoencoders are often very specialized on their input data, as they need to adapt to the specific data in order to replicate the input as well as possible.

Autoencoders can differ in four basic principles [4], which can be changed depending on the application.

First, the **size of encoded data** in the hidden layer, called h, can be different. Depending on the code size h, there are three different types of autoencoders. If the h is larger than the input, it is called **overcomplete**, if the h size is the same, it is called **complete** (see Figure 2.5). For **undercomplete** autoencoders, the size of h is smaller than the input. For these, this is also called a bottleneck, since the information must be compressed to the size of this layer.

Second, the **number of hidden layers** is variable, but must be at least one to have an autoencoder at all. Figure 2.6 shows one with three hidden layers, one for encoders and one for decoders. Autoencoders with more layers are also called deep autoencoders.

Third, the **number of nodes per layer**. This is constrained in most autoencoders by h, which decides whether the number per layer in the encoder part goes up, down, or neither toward the middle layer.

Fourth, the **loss function** on which the autoencoder is trained, which can be the MSE mentioned earlier, but also another estimator for the deviation between input or output, or any other value calculated over the input and output of an autoencoder.

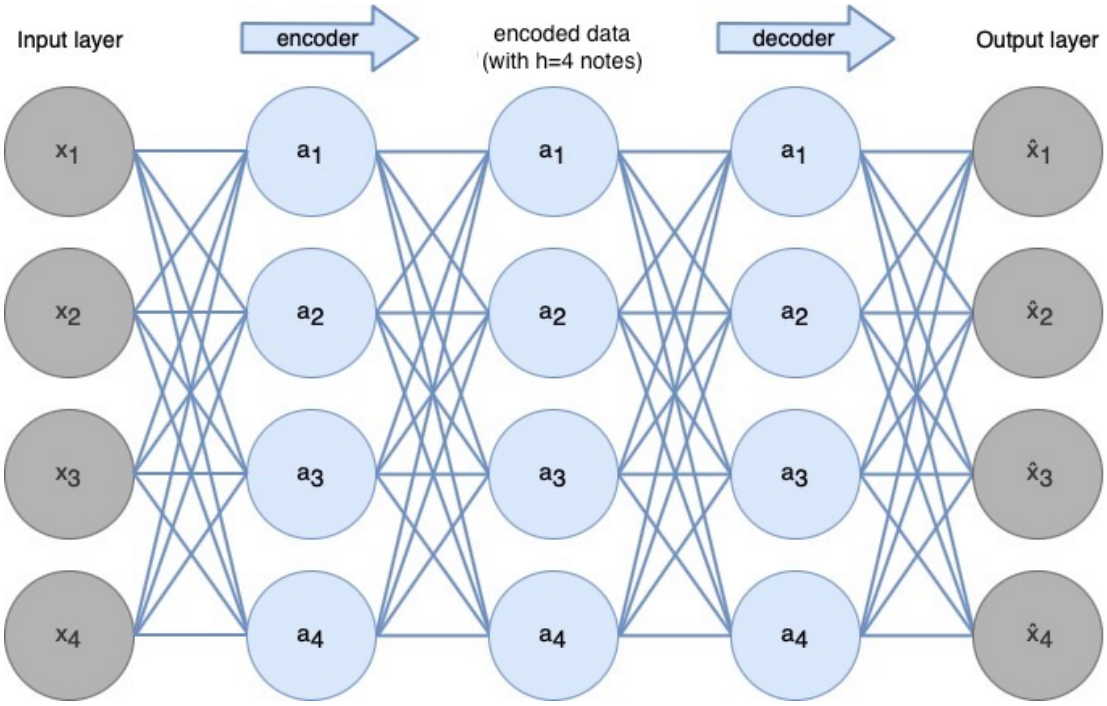

Figure 2.5: A complete autoencoder with one en-/decode layer

2.4.1 Undercomplete Autoencoders

This type of autoencoder is explained in more detail because it can be used for feature extraction. As described above, undercomplete autoencoders have a node size h that is smaller than the input size. Typically, hidden layers also have a node size between the input and h, as shown in Figure 2.6.

This smaller h generously forces the neural network to filter out the most important features from the input in order to reconstruct the data as well as possible.

The number of encoder and also decoder layers should not be so large that they only perform the copying task by representing the train data in their layers. This would only lead to good performance on the train data and perhaps validation data, but would not help to gain knowledge from the data and would not be useful after training. This so-called **overfitting** is a common problem with neural networks and for this reason the evaluation must be performed on different data.

In addition, the size of h must be small enough not to represent too much of the input data, but also not so small that it cannot represent all the important features of the data. [4]

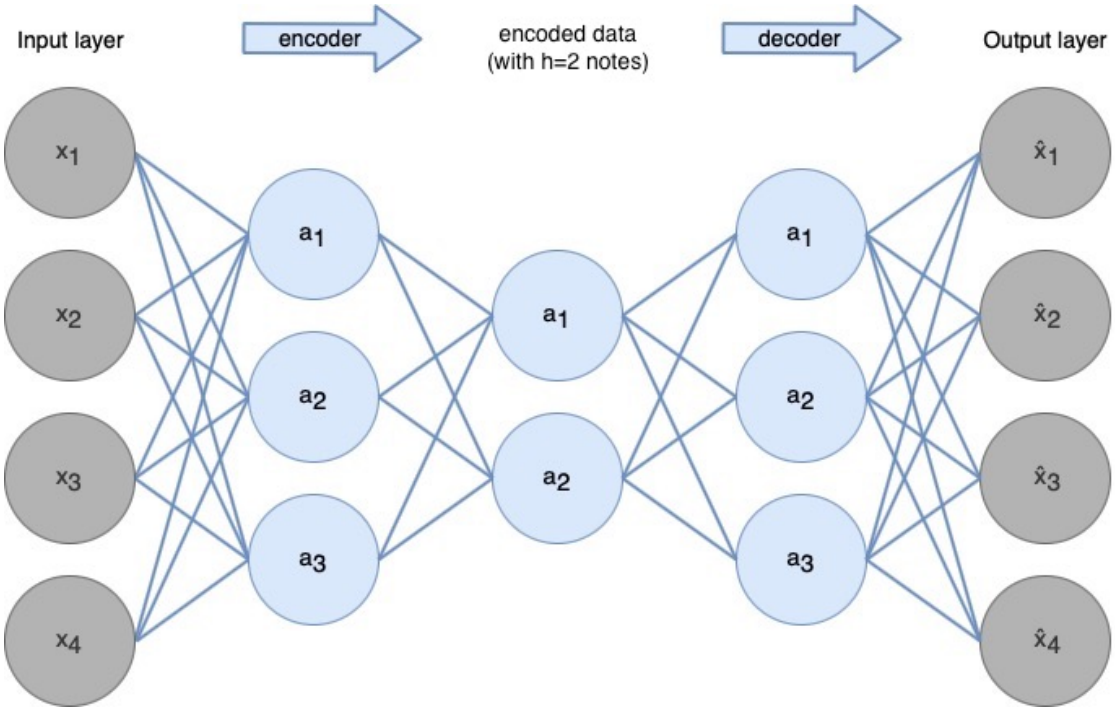

Figure 2.6: An undercomplete autoencoder with one en/decoder layer. The input consists of four input data points, as does the output. The first layer brings them down to three and the encoded representation consists of only two nodes. The decoder brings this back down to three and then to the four output points to resemble the input.

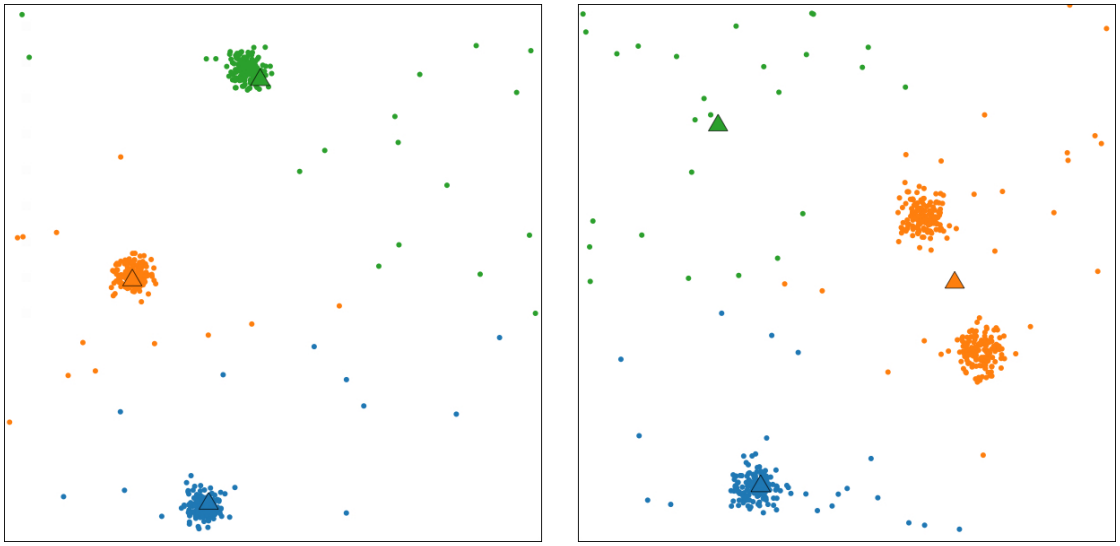

Figure 2.7: K-means performed on two different data sets with 3 centroids visualized as triangles. Data points associated with the same cluster are shown in the same color as their centroid. Visualized with a k-means simulation by Karanveer Mohan [12].

When performed well, undercomplete autoencoders can be used to reduce the dimension of the data by extracting only the most relevant features, since their limitations prevent them from storing the full data. The data represented in the middle layer can then be used for other tasks, like clustering, not just reconstruction. This reduced content is also called **latent representation**.

2.5 K-Means

K-means is an unsupervised clustering algorithm, meaning there is no target value, therefore accuracy cannot be measured and other metrics are required to evaluate performance. Basically, clustering is the process of dividing a dataset into different groups (or clusters) based on certain similarities they share. [17]

In k-means, k stands for the number of clusters and thus the so-called centroids, each centroid is basically the center of a cluster. To train (or fit) k-means, a training dataset is needed.

First, each centroid is randomly selected from the data set, then for all data points their cluster is calculated depending on the centroids. Then, the centroid of all points in the same cluster is computed, which becomes the new centroid. Using the new centroids, the data is reassigned to the different clusters and the centroids can be updated again. This

process continues until either the centroids or the cluster assignments do not change, or the number specified as the maximum number of iterations is reached. [17]

After the training process, the cluster of a given data point can be predicted using all cluster centroids and computing the closest one.

Figure 2.7 on the left shows how k-means can work very well in the optimal case. However, on the right side, 3 different clusters can be seen, but the algorithm assigns values from two clusters to one. Reasons for this can be too much noise, too similar clusters, or bad initial values of the centroids.

The performance of clustering can be measured as the distance between data points in the same cluster, this is called *inertia*. Since clustering in this thesis is performed on a reduced dimension, the evaluation of the performance of clustering in the Evaluation chapter 5 is done on the data before its reduction.

Chapter 3

Related Work

This chapter will illustrate which research led to the conclusion that using an undercomplete autoencoder for feature extraction on spike data is a promising idea. Additionally inspiration on how to design and train the autoencoder is gathered. In addition, another research that shows how the training of time series data in general could be improved is examined in detail.

The first of this researches is from Thies et al. [18], who used an autoencoder to compress spike data. Radmanesh et al. [15] also used an autoenocder, but for clustering the spike data.

Huang et al. [7] used PCA rather than autoencoders for feature extraction before the clustering of spike data, but combined these two tasks.

Lastly Madiraju et al. [10] combined an autoencoder with the clustering task, but for time series data in general.

3.1 Compact and low-power neural spike compression using undercomplete autoencoders

Thies et al. [18] used an undercomplete autoencoder to efficiently compress spike data to send over the air and predict the spikes after reconstruction.

The reasons they suggest for the suitability of undercomplete autoencoders for spike data are that spikes tend to have similar shapes with only a few different variations that can be represented in a lower dimension, and autoencoders are well suited when the data falls into a few classes. In addition, they chose an autoencoder over PCA to have more resistance to variations and noise in the data.

As a reconstruction loss function, they used the signal-to-noise distortion ratio (SNDR), which is basically a metric to measure signal quality, between the input and output of the autoencoder. This is combined with a weight regulation coefficient to ensure that the weights in the model are not too large, as this loss increases when the model contains

large weights. This is necessary because the goal is to put part of the autoencoder on hardware with small memory.

After training the autoencoder, they report that they can infer from the weights of the autoencoder that it has learned to identify the peaks of the spikes. In their experiments, the autoencoders performed better than discrete wavelet transform (DWT), another feature extraction technique for time series data that uses low-pass and high-pass filters to obtain a latent representation of the data.

With the autoencoder finished, they designed a hardware architecture on which the encoder could run and compress the data. The tests on this gave promising results, but they expressed concerns that the spike shapes might change over time in a real scenario and mention that increasing the robustness of the autoencoder compression is still an open task.

3.2 Online spike sorting via deep contractive autoencoder

Radmanesh et al. [15] used the latent representation of an autoencoder to cluster spike data. They say that spike sorting lags behind the growing ability to record more neurons simultaneously, and that current spike sorting techniques are unable to handle the capability of modern electrodes, and that newly proposed techniques are only capable of sorting data offline, after the recording is completed.

To improve spike sorting, they presented their so-called deep contractive autoencoder, which has symmetric encoder and decoder components, each consisting of two layers, followed by a *dropout* layer after each one. A *dropout* layer changes a certain percentage of the random values from the input values to zero, this is done to prevent overfitting of the data since the neural network cannot rely on specific connections and must use as many as possible. Then, the latent spatial representation of the data is clustered using k-means.

With this approach they archived a accuracy of 90% on the test data. Additionally, the accuracy of using this representation to classify the data was compared to using the input data directly. This was done with three different base classifiers and with the data first sent through the encoder, which always performed better. The classifiers were the linear support vector machine, Naive Bayes, and k-nearest neighbors. They also showed that adding noise to the training data only slightly decreased the accuracy. In addition, they tested the accuracy on small datasets to prove that their approach is feasible as an online approach that can be trained on the go.

3.3 A Unified Optimization Model of Feature Extraction and Clustering for Spike Sorting

Huang et al. [7] performed spike sorting by training feature extraction and clustering together, rather than each separately. They name three reasons why performing feature extraction and clustering separately on spike data can lead to unsatisfactory results. First, there is no link between feature extraction and clustering, so they are not aligned. Second, the extracted features may be sensitive to noise and finally, the unsupervised clustering methods are sensitive to poor initial values.

They present a complete spike sorting pipeline. For the feature extraction they use PCA and as their unsupervised clustering algorithm k-means, which is initialized with the *k-means++* initializer. To combine these two algorithms, they are first formulated as a decision matrix and then combined into an equation using what is called the alternating direction method to ensure that the equation converges.

They tested their method on various data sets and achieved 100% accuracy when there were no overlapping spikes, outperforming most of the methods compared. Their performance was even better on data sets with overlapping spikes, where the other approaches could not keep up, with accuracy mostly above 99.5% across the different simulations.

3.4 Deep Temporal Clustering: Fully unsupervised learning of Time-Domain Features

This approach is not focused on spike data, but instead represents an interesting technique for clustering time series data with autoencoders in general.

In 2018, Madiraju et al. [10] presented their so-called deep temporal clustering (DTC). The system consists of an undercomplete autoencoder and a temporal clustering component to cluster time series from the latent representation of the data it receives from the encoder, as shown in Figure 3.1.

This system is trained completely unsupervised by combining the loss functions from the decoder output as the difference from the input data using the mean squared error (MSE) and Kullback-Leibler divergence (KL divergence) from the temporal clustering, called **reconstruction loss** and **clustering loss**, respectively. Training both components together produced results superior to separate training, as the area under the curve was approximately 5% better.

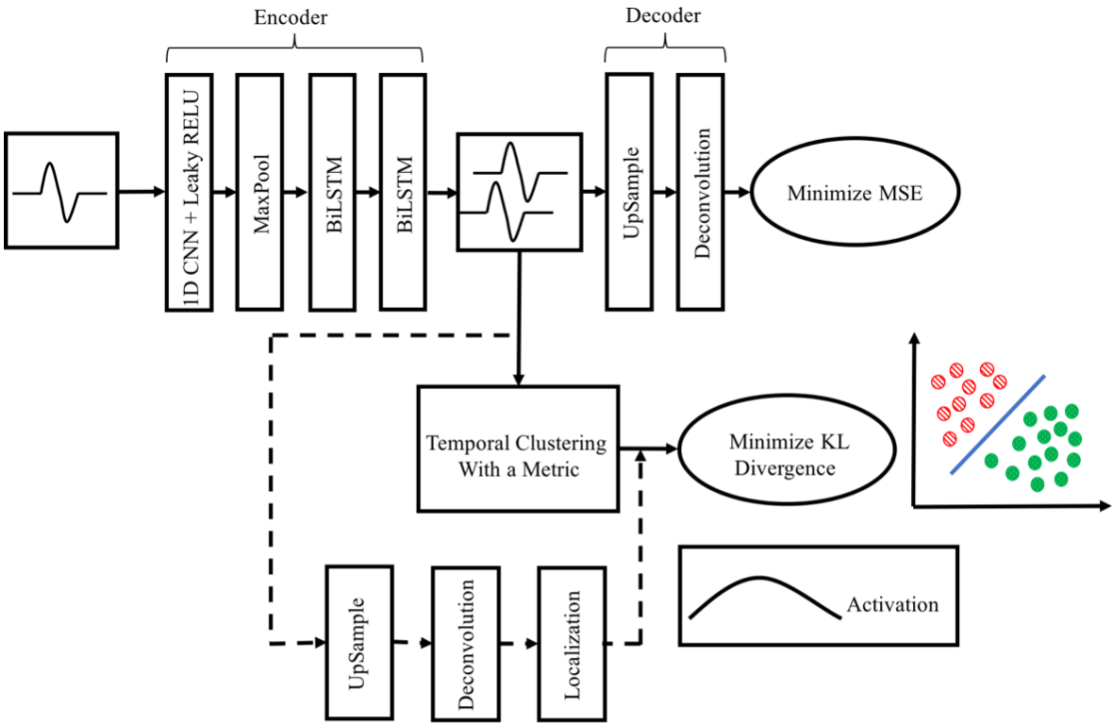

Figure 3.1: The Deep Temporal Clustering overview by Madiraju et al. [10]. The two circles show the loss functions and the dashed part represents the generation of the heatmap.

3.4.1 Temporal Autoencoder

The first layer of their proposed encoder is a one-dimensional *convolutional* layer. A *convolutional* layer consists of several kernels, also called filters. The size of a kernel determines the amount of input it receives from all channels at once. Each channel input is then multiplied by the corresponding kernel channel. All channels are combined into one output by addition. In doing so, the kernel iterates over the entire input data. For example, a size 3 filter receiving a size 20 input with 2 input channels would generate an output for the first 3 inputs from both channels, so would have 6 input values, then the next output from the second to the fifth value from both channels, and so on. This would then generate 28 output values.

This is followed by a *max pooling*. *Max pooling* receives a number of values as input and simply outputs the largest value from them. Thus, with a *max pooling* of 2, an input would be halved by iterating over the input in increments of 2 and always keeping the larger value of the 2 input values. This step mainly reduces long sequences while preserving most features, which prevents poor performance on the next layers.

This is followed by two *bidirectional long-short term memory* (BiLSTM) layers as the second layer. *LSTMs* are used when learning connections in sequential data. Bidirectional ones receive the input data from beginning to end and in the opposite direction, which can help the learning process. Thus, the goal is to preserve the connections over the time scale while further reducing the dimensions. This reduced dimension is called latent representation and is used for clustering.

3.4.2 Temporal Clustering

Temporal clustering is the third layer and is performed by first obtaining a latent representation from the encoder after pre-training. This is divided into k clusters, the average of each cluster is the initial value of its centroid. The centroid represents the mean vector of the cluster. Each new input is compared to each centroid and the data is sorted into the most likely cluster. Then the centroids are updated, giving more weight to the assignments with the highest confidence.

3.4.3 Heatmap

To gain insight into the black box of unsupervised training, a visualization is generated to show which data features are most relevant to the cluster assignments. This is archived by training a supervised convolutional network with the same input as the temporal clustering component and comparing the labels from the clustering with its classification.

3.4.4 Conclusion

The results presented by Madiraju et al. [10] present in their evaluation suggest that for the task of clustering time series data using an undercomplete autoencoder, the combined training of cluster component and autoencoder is generally preferable to separate training. This conclusion can be drawn from their comparison on 13 different datasets [19], where DTC always performed best. However, as mentioned at the beginning of this section, their tested datasets do not contain spike data, so no final conclusion can yet be drawn about the usefulness of this approach for spike sorting.

3.5 Conclusion

Thies et al. [18] showed that an undercomplete autoencoder is generally able to learn important features from spike data and outperform the classic feature extraction technique DWT. Madiraju et al. [10] demonstrated that in general clustering the latent representation gained from an autoencoder of time series data can be successful, especially when trained combined with the clustering. From Huang et al. [7] it is known that training PCA combined with clustering on Spike data yields better results. Additionally Radmanesh et al. [15] found that using an autoencoder to reduce the dimension can be better than using the original data.

Concluding from this using the latent representation of an undercomplete autoencoder for spike clustering seems promising. Further is could be tested if using a combined training as proposed by Madiraju et al. [10] also yields better results for spike sorting, as this type of data was not in their test data sets, but was already used successful similar with PCA on spike data.

Chapter 4

Design and Implementation

This chapter describes how the undercomplete autoencoder is designed and implemented as well as the PCA for comparison and a k-means for the clustering. Moreover a heatmap is implemented to see which features of the spike data are most relevant to determining its cluster. The majority of the code is implemented using the programming language **Python**(3.8.2)[1].

4.1 Spike Alignment

Before the alignment the Savitzky-Golay [16] filter is applied to the data. As Azami et al. [2] point out this filter can de-noise and smooth signals while keeping important features like relative maxima and minima, for example in contrast to the similar moving average.

The filter is used with a order of 4 and a frame length of 15. With a frame length of 15 for calculating each filtered data point, the point itself and the 7 before and after it are used. For the edges a matrix multiplication is performed for getting the same amount of data points for each side. Using the data points a polynomial function of the given order is calculated and the retrieved data point form this is the filtered data point. The filter is implemented using the available implementation from Matlab [2].

Each detected spike lies in a windows of 60 time steps and is aligned into a window of 48. Meaning that the autoencoder and PCA both will have an input of 48 data points. It can be seen in Figure 4.1, how some spikes that were detected late are now more in order and the spike centers are aligned together. However some errors still occur and can be seen as deviations in the last third, after the alignment.

[1] Python: `https://www.python.org`

[2] MathWorks Savitzky-Golay filtering: `https://www.mathworks.com/help/signal/ref/sgolayfilt.html`

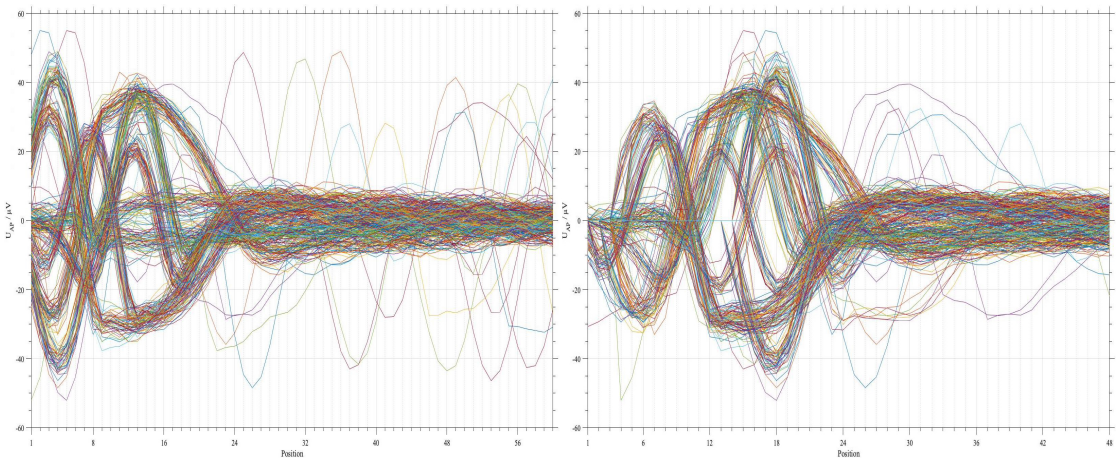

Figure 4.1: Generated 10 minute spike data set with 7 different spike types before alignment on the left and after alignment using Savitzky-Golay as the filter on the right.

The aligned spikes are read out from the Matlab code using the Matlab engine for python provided by Matlab. The python code saves the aligned spikes as **NumpPy**(R2020b)[3] arrays in the NumpPy file format.

4.2 Autoencoder

The autoencoder is split into an encoder and a decoder component, where the complete autoencoder puts the input data through the encoder and its output then through the decoder. The autoencoder uses the machine learning framework **PyTorch**(1.8.1)[4].

The visualized architecture of the proposed autoencoder can be seen in Figure 4.2 with an example latent representation of size 12. Each bar represent one channel. The red squares show the input and the green ones the output and the dotted lines represent padding. The latent representation can be found between the encoder and decoder component with the size of h. It can bee seen that the outputs of both *LSTMs* depends on h, this will be described in more detail in the encoder section.

4.2.1 Encoder

Most of the layers used in the encoder were also used by Madiraju et al. [10], and therefore explained in section 3.4.1.

[3]NumpPy: `https://numpy.org`
[4]PyTorch: `https://pytorch.org`

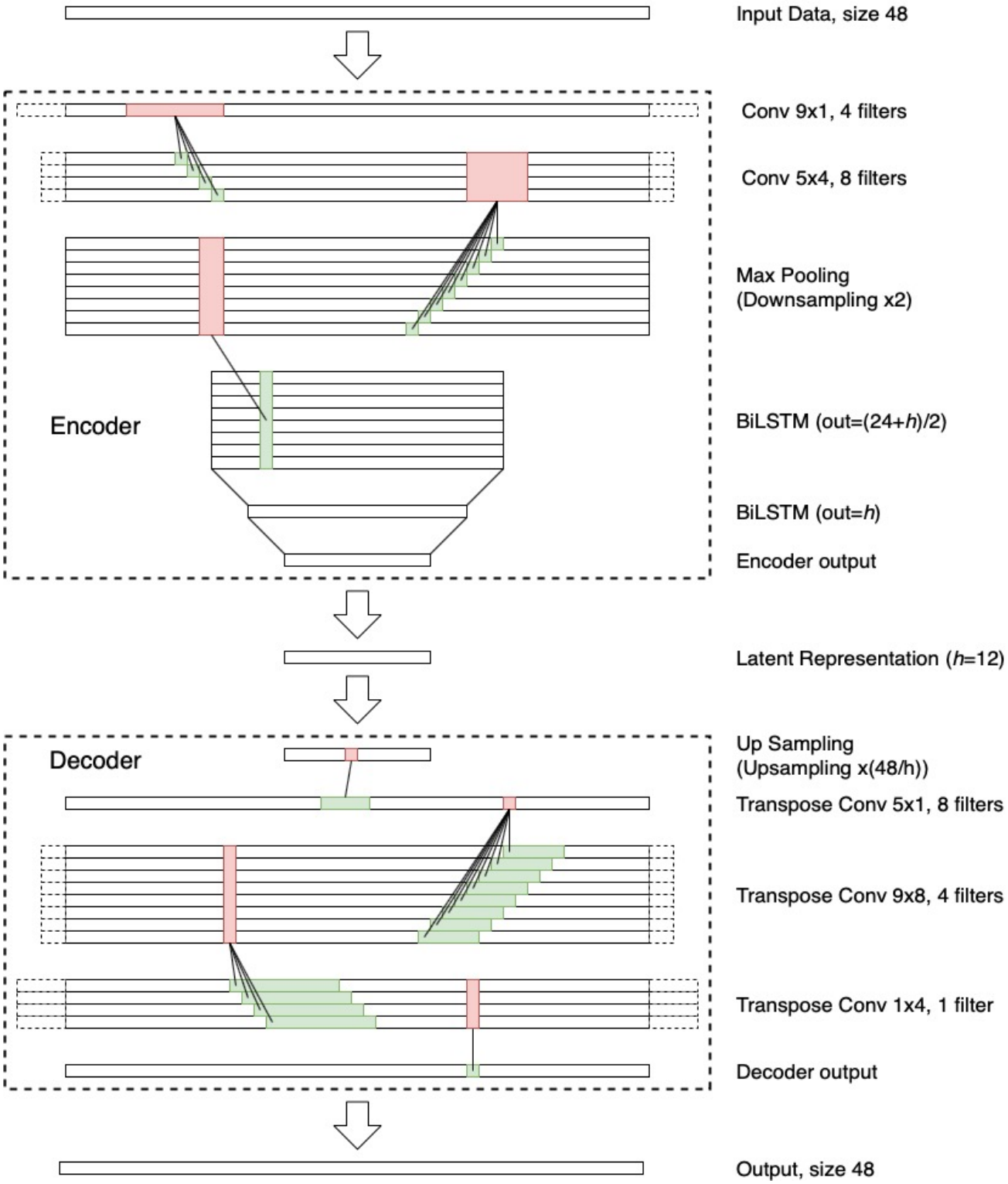

Figure 4.2: Autoencoder architecture with a latent representation of size 12.

The filter dimensions will be given as width times channels (width x channels), as the spike data is one dimensional there is no height.

The first *convolutional* layer used has 4 kernels of size 9 (9x1) each. A padding of size 4, which replicates the outer most values, is applied. Padding in *convolutional* layers is used to preserve the size of the input data, as a kernel bigger than size 1 uses multiple values to generate one output value.

This is followed by a *Leaky Rectified Linear Unit* (Leaky ReLU) activation function. The *ReLu* activation maps all positive values to themselves, but all negative values to zero. The *Leaky ReLu*, is similar, but instead of mapping the negative values to zero it uses a coefficient to reduce the gradient for negative values. For example with a coefficient of 0.25, -1 would be -0.25, while 1 would remain 1.

A second *convolutional* layer follows with a smaller kernel size of 5 and the 4 input channels (5x4), but with a higher number of 8 kernels. The padding size of 2 is needed this time to keep the data size.

Again the output is activated by a *Leaky ReLU*.

The dimension is then cut in half by a *max pooling* of 2 in each channel, leaving a dimension size of 24 in 8 channels.

The encoder is completed with two *bidirectional long short-term memory* (BiLSTM) layers.

The size between those layers is determined by the size of the embedded dimension, being in the middle between the embedded dimension and 24. For example an embedded dimension of 8 would lead to a output size of 16, 8 for each direction. Because each direction has an output, odd values are not possible and decreased by one, so the size for an embedded size of 18 would be 20 instead of 21. The first BiLSTM also combines all 8 channels into 1.

Finally the second *BiLSTM* outputs the latent representation, each half consisting of one direction, therefore the embedded dimension must be odd as well.

This architecture aims to use the first filters to learn the longer time trends, then with the smaller, but more filters the more different short term trends. This is different to the deep temporal clustering design with only one *convolution*. The *max pooling* after this two layers should get the more significant values from the *convolutions*. The target of the *BiLSTMs* is then to combine and compress this information into the aimed smaller dimension.

4.2.2 Decoder

The decoder starts with a simple *up sampling* layer to bring the latent representation back to the original input dimension. An *up sampling* layer just replicates the input values. When the input to the decoder is of size 12, each value would be used four times to get back to size 48.

Following that a first *transposed convolution* is performed with size 5 and 8 kernels with padding 2, exactly like the second convolution. A *transpose convolution* gets one input and produces as many outputs as its kernel size for each kernel. So when early 5 values were convoluted into one value, now one value is "de-convoluted" into 5 values. To keep the data size the padding on *transpose convolutions* removes the borders of the output.

A second *transposed convolution* replicates the first convolution with 4 kernels each of size 9 and a padding of 4.

As the output of this has the dimension of the original input, but with 4 channels from the 4 kernels, a third *transpose convolution* is performed using only one kernel with the size of one to get back the original autoencoder input size with only one channel.

The *up sampling* layers is the equivalent to the two *BiLSTMs* and the *max pooling*, while the two *transpose convolution* layers should invert the two *convolution* layers from the encoder.

4.3 Principal Component Analysis

To compare the results of the autoencoders to another feature extraction technique, principal component analysis is implemented as well.

The PCA is initialized with a size of the reduced dimension (principal components) and then fitted on the trainings data. After that it can be used so reduce dimension of data to the number of components. For this implementation PCA another machine learning library called **scikit-learn**(0.24.2)[5] is used.

4.4 Clustering

For the clustering a standard k-means clustering implementation also from the **scikit-learn** library is used. The number of components corresponds to the number of different clusters, so the amount of different spikes. It is fitted to the preprocessed training data and can then be used to determine the cluster number of a given input.

[5]scikit-learn: `https://scikit-learn.org/`

The k-means is always initialized with the *k-means++* [1] initializer, as proposed by Huang et al. [7]. With *k-means++* the initial cluster centroids are not picked randomly from the data, but instead just one is picked to be the first centroid, after that the data points which is the furthest away is chosen as the second centroid. Then, one centroid at a time is selected, always being the data point farthest from all other centroids, until the complete amount is reached. After being fitted, the cluster of a data point can be predicted.

4.5 Heatmap

Following the approach of Hwang et al. [8] and the proposed integration with Deep Temporal Clustering by Madiraju et al. [10], a neural network is implemented for the purpose of visualizing which parts of an input are most important in determining its cluster. This network is a supervised and uses the labels generated by clustering after feature extraction with the trained autoencoder.

The network starts of with the same layers as the decoder. The output from this part then goes to the classification layer as well as the localization layer. The classification part contains a *linear* layer with an output size as large as the number of clusters. *Linear* layers compute a linear transformation to the output form. The localization part starts with a *convolutional* layer, where the number of cores is equal to the number of clusters. This is done because when a *global average pooling* is applied afterwards, each cluster is given one value. This is because *global average pooling* returns the average value of each channel. After both parts, the *softmax* function is applied. This function rescales the input so that each value is between 0 and 1 and the sum of the values equals one. Applied to the given outputs, each of which represents a cluster, this basically assigns a probability to each.

For training, the *multi label soft margin loss* [6] is computed for both classification and localization. The loss is initially composed of 90% classification loss and 10% localization loss. After 60 epochs, these percentages are switched to train the mapping kernels after the shared layers have been sufficiently trained by the classification.

To display the activation after training to get a heatmap, the output of the kernel corresponding to the label of the input data is displayed after the *ReLu* activation.

[6] MultiLabelSoftMarginLoss: `https://pytorch.org/docs/stable/generated/torch.nn.MultiLabelSoftMarginLoss`

Chapter 5

Evaluation

In this chapter, the different autoencoders are compared against PCA as a feature extraction step in the spike sorting pipeline. The autoencoder trained with clustering (combined autoencoder) and the separately trained one (separate autoencoder) are used. These comparisons are made on different simulations with different numbers of spike types as well as different sizes of latent dimensions.

For the evaluation of the clustering, the Kullback-Leibler divergence (KL divergence) is calculated for every two spikes in the same cluster. The KL divergence actually calculates the difference between two probability distributions, but can also be used to calculate the difference between two time series, in this case two in the same cluster. The value between the time series before feature extraction is calculated because it is not relevant whether the latent representation is similar, but the spikes are. A smaller value means that the two time series are more similar, which is desirable since the time series from the same clusters are compared. As negative values are not defined, each spike is evaluated by the amount of its lowest point plus a minimum to avoid zero values. The KL divergence is then calculated using the entropy function from **SciPy**(3.8.2)[1] library.

Since there is no unique numbering for the different clusters, to determine the accuracy, each permutation where each cluster is given a number must be evaluated, and the one with the largest overlap gives the accuracy. Considering that the number of permutations can be calculated using $n!$, where n is the magnitude of the values (here, the number of clusters), this metric cannot be applied for a higher number of clusters due to the high growth rate of the factorial function.

5.1 Spike Data Simulations

Own simulations were performed using a given implementation in the **Matlab**(R2020b)[2] programming language. Records of 2 up to 7 different spike types were generated over

[1] SciPy: `https://www.scipy.org`
[2] MathWorks: `https://www.mathworks.com`

a period of 500 minutes.

Every spike is fired at an average rate of 5 Hz. For each spike, some values are first initialized randomly. The Matlab function *rand* generates a matrix of the given size with random numbers uniformly distributed between 0 and 1. The function *randn* also generates a matrix, but with random numbers taken proportional from the standard normal distribution. The *gaussmf* stands for the gaussian membership function and *heaviside* for the heaviside step function. In Matlab the *find*[3] function applied on an array returns only the indices with non zero values.

$$
\begin{aligned}
n &= 0.6 + 0.3 * \text{mean}(\text{rand}(1,3)) \\
\tau &= 100e-6 * (1 + \text{mean}(\text{randn}(1,5))) \\
f &= \lfloor 450 * (1 + \text{mean}(\text{randn}(1,100))) \rceil \\
U_{SA} &= 60e-6 * (1 + \text{mean}(\text{rand}(1,2))) \\
U_0 &= (n-1) * (e^{\frac{t}{2*\tau}} - 1) \\
A &= \text{find}(U_0 <= n-1) \\
U_0(A) &= 0
\end{aligned}
$$

Using these random values, depending on the spike type the corresponding function is used to generate the spike data, as a time series.

$$
\begin{aligned}
f_1 &= U_{SA} * e^{\frac{-t}{\tau}} \\
f_2 &= -U_{SA} * e^{\frac{-t}{\tau}} \\
f_3 &= U_{SA} * (n * e^{\frac{-t}{\tau}} - (1-n) * \sin(2 * \pi * f * t)) \\
f_4 &= -U_{SA} * (n * e^{\frac{-t}{\tau}} - (1-n) * \sin(2 * \pi * f * t)) \\
f_5 &= U_{SA} * n * \text{gaussmf}(t, [0.6 * \tau, \frac{t(\text{end})}{2}]) \\
f_6 &= U_{SA} * (n * \text{gaussmf}(t, [\frac{\tau}{2}, \frac{t(\text{end})}{2}]) - (1-n) * \sin(2 * \pi * f * t)) \\
f_7 &= U_{SA} * (U_0 + e^{-\frac{(t-t(A(1)))}{\tau}} .* \text{heaviside}(t - t(A(1))) - (1-n) * \sin(2 * \pi * f * t))
\end{aligned}
$$

An average of the data generated by each function can be seen after the alignment in Figure 5.1. When using less than 7 different spikes, the functions are used in the order of their numeration.

[3] Matlab find function: `https://de.mathworks.com/help/matlab/ref/find.html`

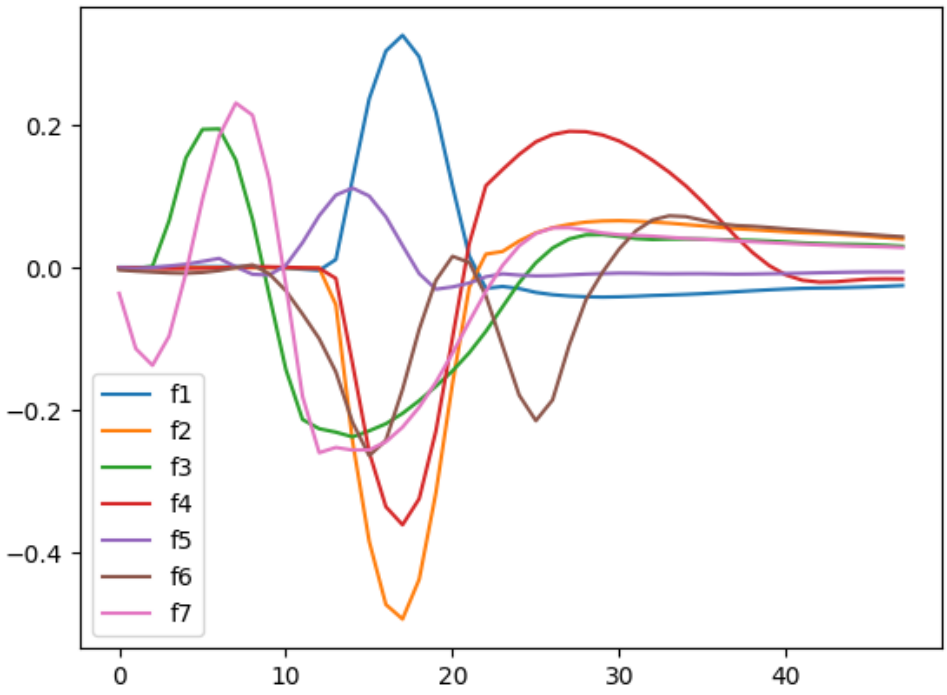

Figure 5.1: The average value of the generated spikes with their corresponding spike function after alignment.

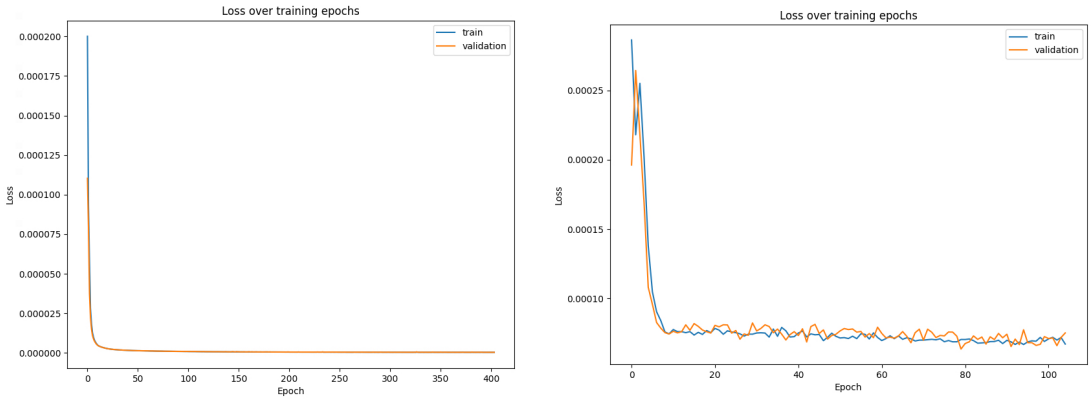

Figure 5.2: The training of just the autoencoder on the left and the combined training on the right with an embedded dimension size of 12 on the first simulation from Pedreira et al. [13].

5.2 Autoencoder Training

Before training, the data is split into 70% training data and 15% each for validation and test data.

Training of the autoencoder is theoretically performed up to 500 epochs, but with an early stopping after 25 epochs in which the validation loss has not improved by more than 0.5%. A data batch of 128 is trained together.

In normal training, each batch of data is fed into the autoencoder and the difference between the input and the output is calculated as the reconstruction loss.

When trained together with the clustering, a k-means is created and fitted to each batch after its data is reduced with the encoder part of the autoencoder being trained. The KL divergence for each cluster is then calculated as the clustering loss.

For the simulations to be evaluated, one autoencoder of each type was trained on every second embedded dimension between 2 and 24, corresponding to a compression ratio between 24 and 2.

Figure 5.2 shows that the validation loss for the autoencoder did not improve after about 400 epochs, after slowly decreasing. The combined training shows that the loss is not as constant as the reconstruction loss and does not improve after about 100 epochs.

5.3 Performance on own Simulations

Figure 5.3 and 5.4 show that for just two different spike types all approaches archive the same KL divergence and a perfect accuracy.

With an additional cluster, the combined autoencoder needs at least a size of 10 to achieve the same divergence as the PCA and the separate archives it only at dimension sizes 12 and 16. Looking at accuracy, a latent representation of size 6 is sufficient to archive the same accuracy. However, the separate shows poor performance at 10, 20 and 22, while it shows the best performance at 14 and 24.

In the simulation with four different spikes, the KL divergence for the combined autoencoder is stable from dimension size 8 to 16, while lower and higher sizes show poorer performance. The separate one here is quite similar with an outlier at a latent size of 16 (and 6, where the combined one is already quite good). Comparing this to the accuracy of the larger sizes, only size 18 stands out, as the combined autoencoder has a slight drop-off there. Interestingly, the outlier in the KL divergence shows no significant drop in accuracy for the separate autoencoder.

Looking at performance on 5 clusters, the KL divergence remains constant along with accuracy at a minimum size of 10. The combined autoencoder already performs quite well at size 8, and the separate one has an outlier here with poor divergence and accuracy. This is the first size where PCA needs more than a latent representation of 2 to perform as well as without preprocessing.

With 6 spike types, the combined autoencoder archives the same KL divergence as PCA and no feature extraction above a size of 8, with the exceptions of 14 and 18. The separate autoencoder gets the best performance only at a size of 8, 10, and 22. The accuracy of the autoencoder trained in both ways is better than PCA and no feature extraction starting at a size larger than 14, and for the combined training also at 10, indicating that the sizes with lower performance in KL divergence do not show significantly worse performance in accuracy here.

The simulation with the most different spikes shows consistently slightly worse performance from both autoencoders than PCA and no feature extraction. With the notable exception that both achieve higher accuracy with a latent representation of only 2. Additionally, the separate autoencoder again has an outlier with size 10.

Figure 5.5 shows that for a dimension size of more than 2, the combined autoencoder basically always gets the same performance as PCA, and the separately trained one most of the time. For a dimension size of only two, the autoencoder can only get the same performance on 2 clusters.

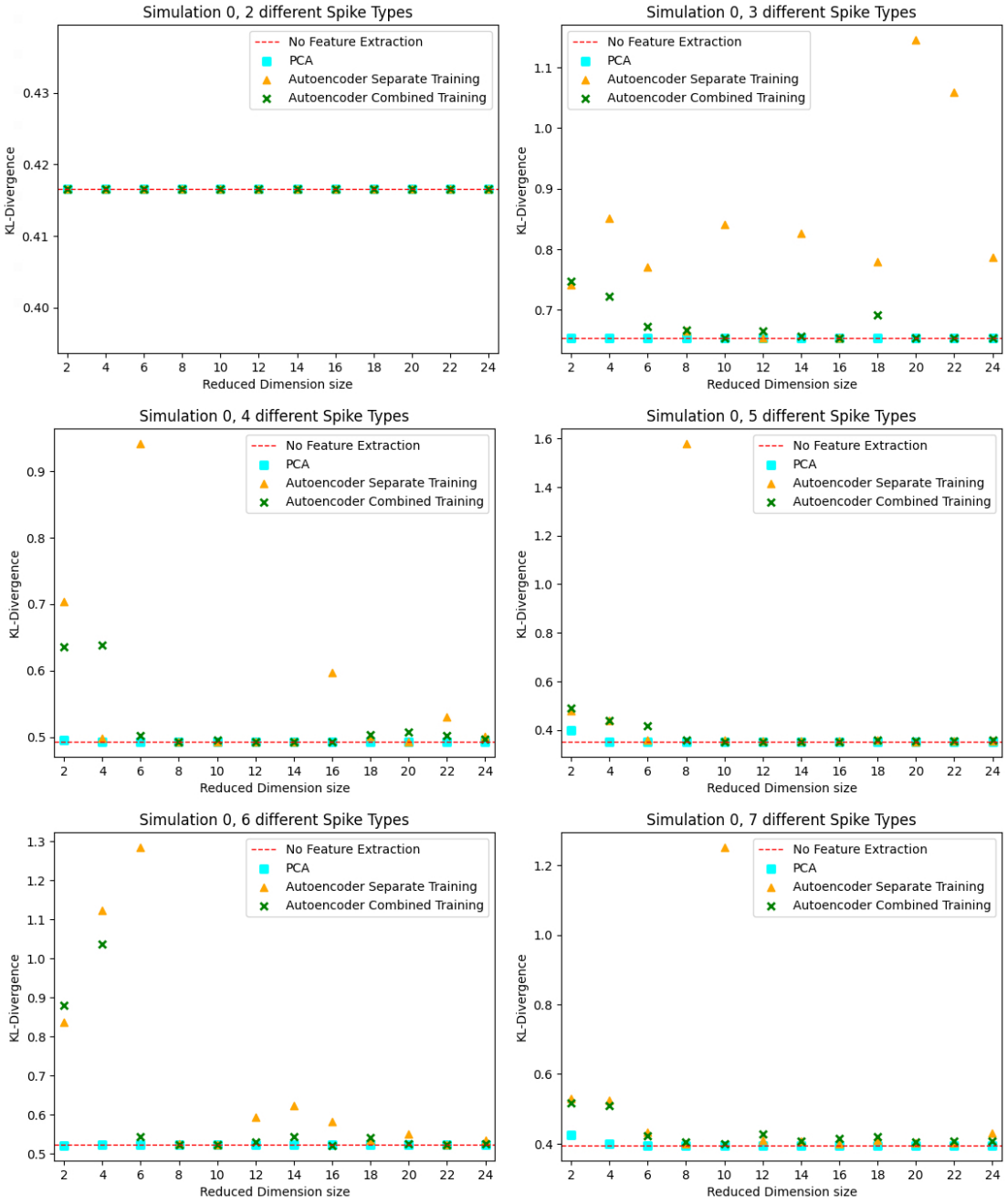

Figure 5.3: The KL divergence on different numbers of clusters for the autoencoder trained with and without combined clustering loss compared to PCA with the data reduced to different dimension sizes and the data clustered with no feature extraction as baseline..

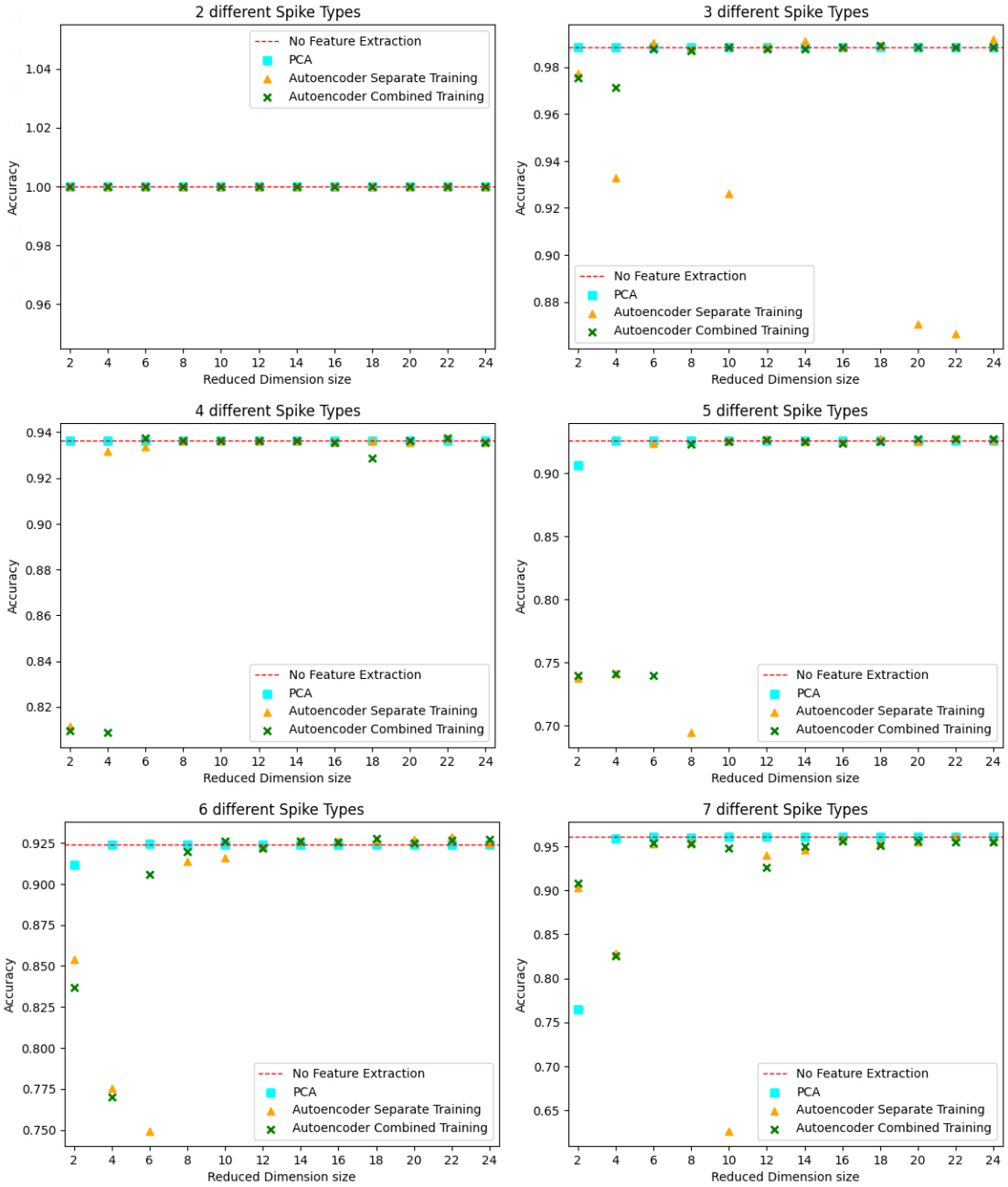

Figure 5.4: The accuracy on different numbers of clusters or the autoencoder trained with and without combined clustering loss compared to PCA with the data reduced to different dimension sizes and the data clustered with no feature extraction as baseline..

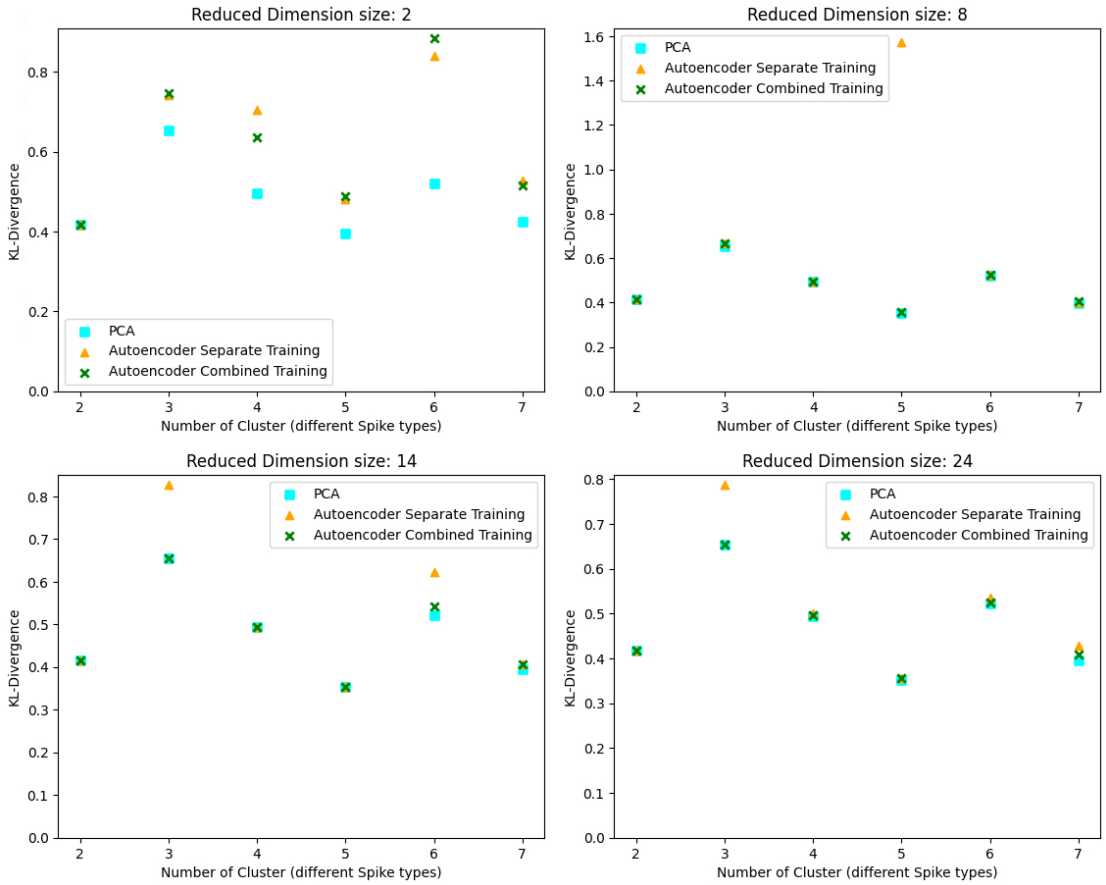

Figure 5.5: KL divergence of PCA and autoencoder trained with and without clustering with different number of spikes averaged on different latent representation sizes.

5.4 Performance on Simulations from Pedreira et alii

Pedreira et al. [13] provide spike data simulations of 2 up to 20 different spike types. For this, they used the mean values of the recorded spikes from a database. Since some simulations have a high number of different spikes, the accuracy could not be calculated because the required computational power is too high, as described at the beginning of this chapter.

For dimensions larger than 6, the autoencoders trained with combined loss get similar results to PCA, sometimes slightly worse and in a few cases slightly better. The autoencoders with separate training perform inconsistently especially for simulation 0 and 1. For a very small dimension size of 2, PCA always performs better than both autoencoders.

PCA and the autoencoders performed better in simulation 1 and 2 than when using the data without feature extraction. In Simulation 3, both autoencoders performed very similarly and outperformed PCA for dimension sizes greater than 10, but Simulation 4 always showed better results with PCA. With two exceptions, the combined autoencoders performed better in Simulation 5 when they had a latent representation of at least 8. The separated autoencoder archived this as well, but not consistently across all dimensions.

These simulations show that the autoencoders can archive good performance even with a high number of distinct spikes, but require a larger latent representation to do so compared to PCA.

5.5 Performance on Simulations from Martinez et alii

Martinez et al. [11] created spike simulations with 2 single unit activations and one multi unit, so three different clusters. To account for the small size of the training data, the batch size for training was set to 32 for these simulations.

As in the last section, Figure 5.7 shows the KL divergence of the different simulations with the different latent representation sizes and Figure 5.8 shows the accuracy of the same.

Simulation 0 shows perfect accuracy for PCA on all dimension sizes and for both autoencoders with a size greater than 8. However, the combined autoencoder shows an outlier at size 18 on the KL divergence, which is lower for the other autoencoders than for PCA.

For Simulation 1, a size of 4 is already sufficient to obtain perfect accuracy and the same divergence as PCA, which requires only 2.

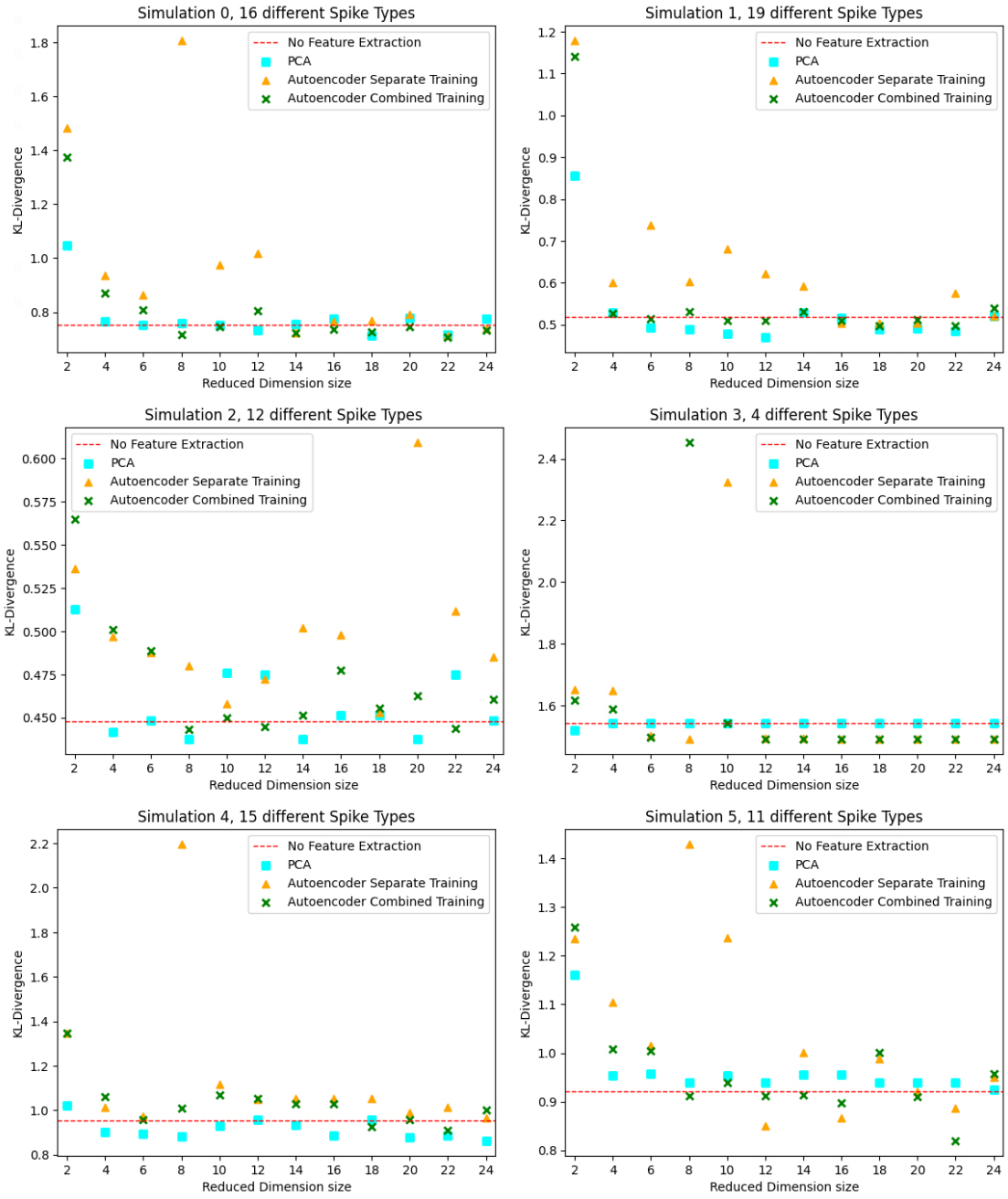

Figure 5.6: The KL divergence for the first four simulations from Pedreira et al. [13] for the autoencoder trained with and without combined clustering loss compared to PCA with the data reduced to different dimension sizes and the data clustered with no feature extraction as baseline..

For simulation 3, except for the combined autoencoder with size 14, all autoencoders with a size of at least 8 get higher accuracy than PCA, which performs better with sizes 2 and 4 and similar with 6. When considering KL divergence, the combined autoencoder performs better than PCA with one exception, even for small dimension sizes. However, the separate autoencoder gives the best result at size 20, but for all other dimensions they were outperformed by the combined autoencoders.

Simulation 4 gives different results for both types of autoencoders. Some manage to archive a better KL divergence than PCA with the same dimension, while others perform significantly worse and PCA even performs best at a size of only 2. Better accuracy is also archived by some autoencoders, the best being the combined one with a size of 24, but again most perform worse than PCA, consistent with a latent representation of at least size 4.

5.6 Different Training and Test Simulations

Additionally the performance is compared by training the feature extraction techniques on one simulation form Martinez and then using the train data of another simulation to fit the k-means after doing the dimension reduction with the already trained autoencoder and PCA. This scenario is equivalent to the feature extraction happening on hardware and the latent representation being send to a server, where now after the neurons around the chip have changed the clustering on the server can be changed to fit the new scenario but the autoencoder or PCA on the hardware not.

When comparing the KL divergence in Figure 5.9, overall autoencoders trained on simulation 0 and 1 and then tested on the other ones, show better performance than PCA, this is especially true for the combined trained ones. With the simulations 3 and 4 trained, PCA performs better when tested on 0 and most of the time equal on simulation 1 (with some autoencoder outliers). When testing 4 on 3 and vice versa, there is mixed performance with the separate autoencoders compared to PCA, but mostly better ones with the combined autoencoders.

From this it can be assumed that autoencoders have the capability to generalize, however the variance is often quit high even though the combined training was able to lower this on some simulations. This correlates with the claim from Thies et al. [18] from chapter 3.1 that the robustness of autoencoders against spike changes is still an open problem.

5.7 Heatmap

The example heatmaps in Figure 5.10 show that the information for clustering is mainly collected where one would expect it, since the activations are mostly around the spikes.

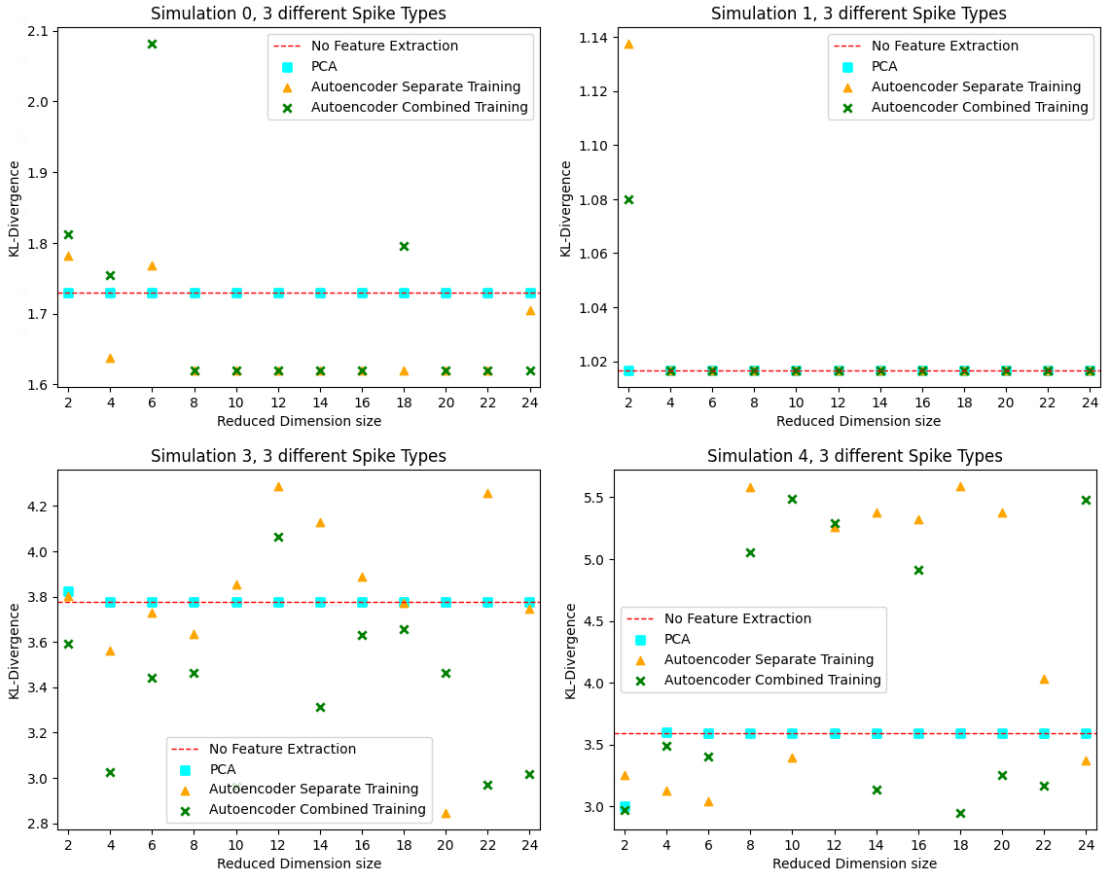

Figure 5.7: The KL divergence on selected simulations from Martinez et al. [11] for the autoencoder trained with and without combined clustering loss compared to PCA with the data reduced to different dimension sizes and the data clustered with no feature extraction as baseline.

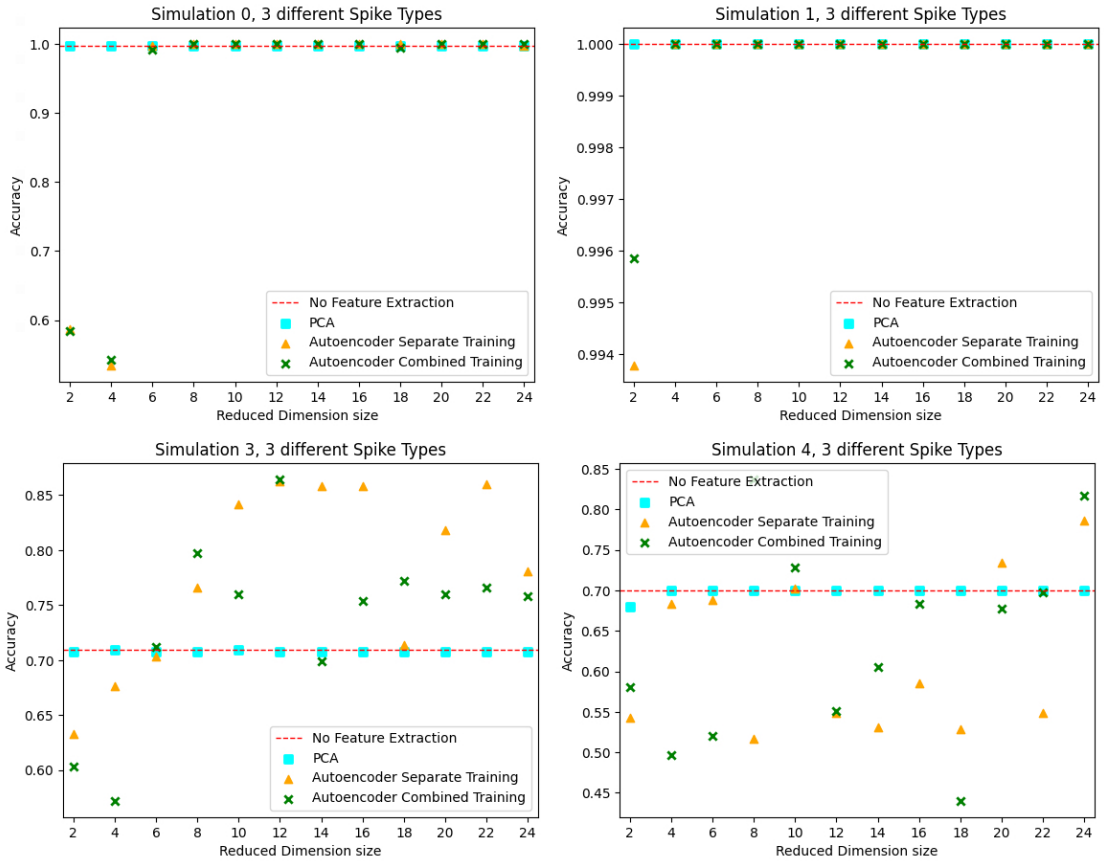

Figure 5.8: The accuracy on selected simulations from Martinez et al. [11] for the autoencoder trained with and without combined clustering loss compared to PCA with the data reduced to different dimension sizes and the data clustered clustered with no feature extraction as baseline.

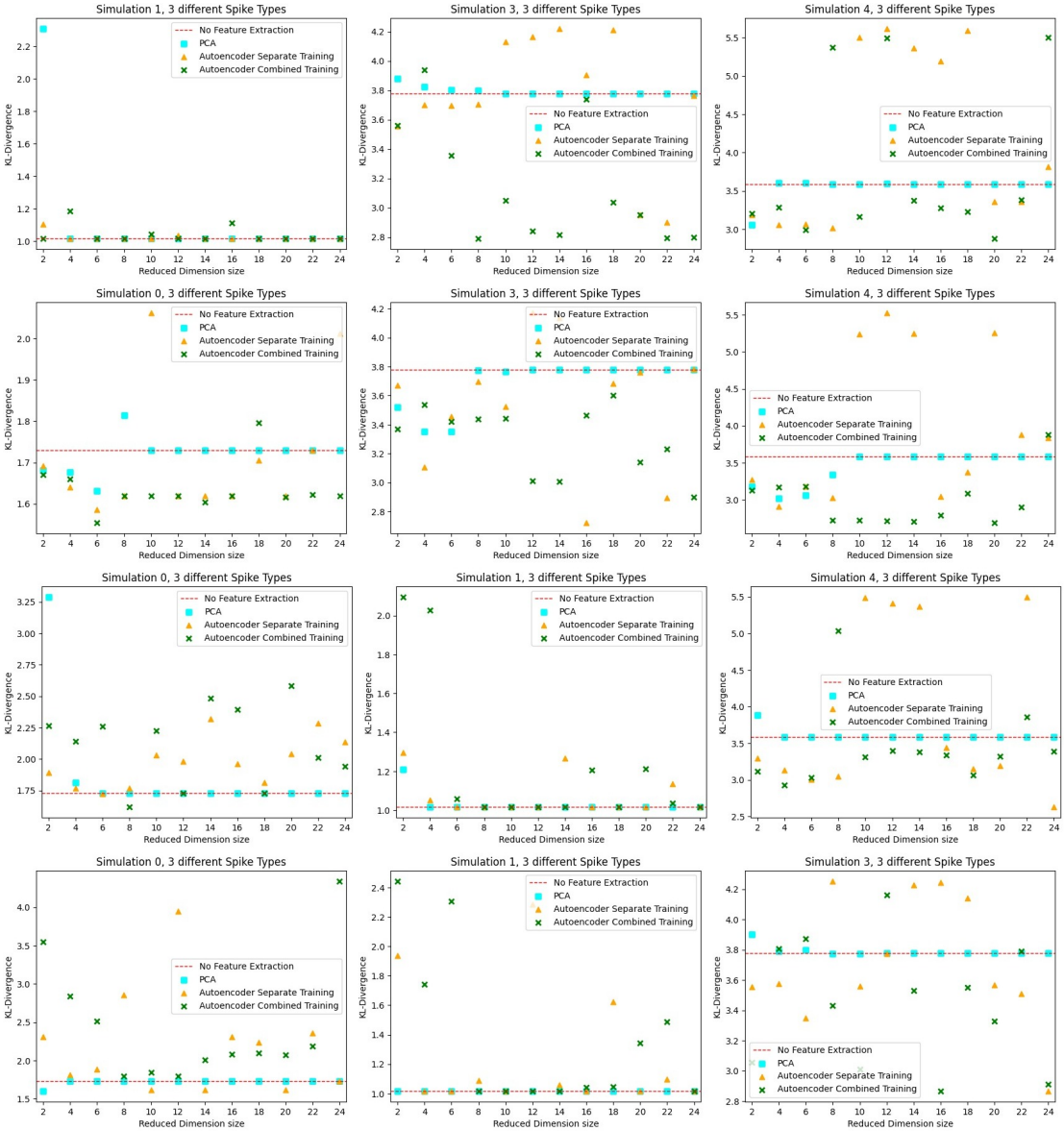

Figure 5.9: KL divergence trained on one Simulation from Martinez et al. [11] and then tested on three other ones. Each row was trained on another simulation, starting from the top 0, 1, 3, and 4.

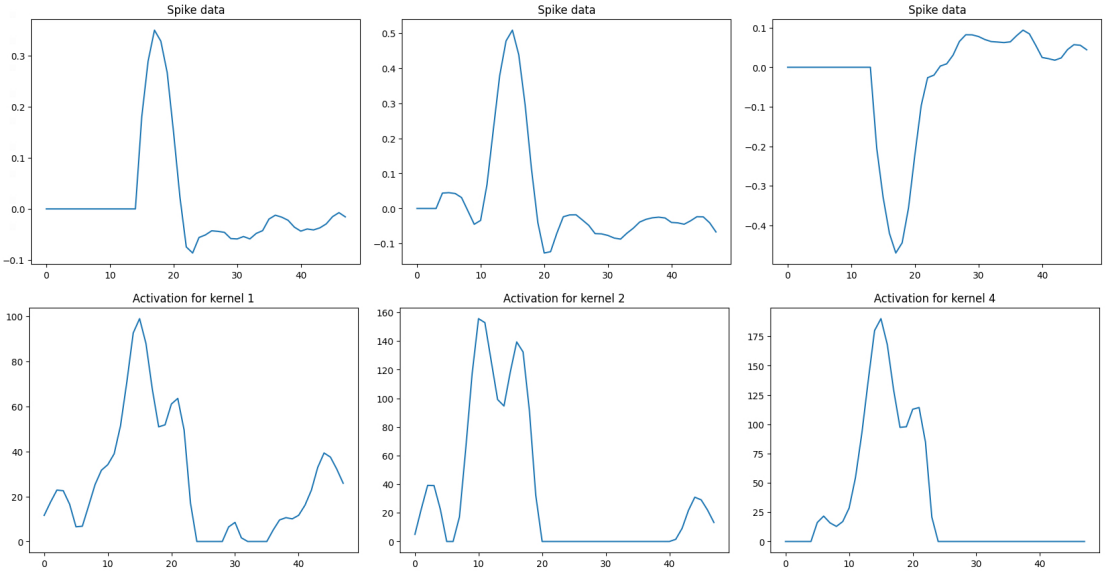

Figure 5.10: The heatmap network applied to 3 spikes form the test data from the own generated data set with 5 different spike types. The used autoencoder was trained combined with a latent representation of size 12. Below the individual spike data, the heatmap of the data can be seen as the output from the kernel corresponding to the label of the data.

Whereas the highest activations are located at the first climbs and smaller ones at the second ones. On the first entry, a medium activation is found at the end of the input, for which there are no clear reasons. This could indicate that this spike type is partially identified by undesirable features.

5.8 Results

In summary, there are scenarios where an autoencoder can achieve better performance when used as a feature extraction technique compared to PCA or no feature extraction at all. Certainly PCA is much more consistent and requires only a fraction of the computational power to train. However, combining reconstruction with clustering loss when training the autoencoder often yields even better results and is more consistent, especially as a preprocessing technique for the clustering task. On the smallest latent representation of size 2 tested, PCA always performed better and when an autoencoder outperformed PCA, it was usually only at a latent size of at least 8. When the number of clusters is taken into account, autoencoders did not show a significant decrease over PCA with more spike types.

The autoencoders showed the potential for generalization, the combined trained autoencoder slightly more so. However, this was not true for all simulations and would need to be improved for a real-world application.

Chapter 6

Conclusion

This chapter summarizes the main points of each chapter and then previews possible future work.

First, the data used in this thesis and the way these spike data is sorted were explained, then a part of the spike sorting pipeline, the feature extraction technique PCA, was presented and it was shown how autoencoders can be used for the same task. For the next step in the pipeline, k-means was presented as an unsupervised clustering technique.

With this knowledge, various works on spike sorting and autoencoders for spike sorting were used for the decision to use an undercomplete encoder for the feature extraction step of the spike sorting pipeline. Another result was to try to train the autoencoder with a more complex loss function that combines the feature extraction and the clustering.

The findings from this investigation led to the design of an autoencoder and its implementation.

Finally, the implemented autoencoders were evaluated with different spike datasets that were fed into the spike sorting pipeline. Here, the autoencoder showed potential with a latent representation, where the compression ratio was no higher than 6. A higher number of different spikes, up to 19, showed no significant drop in performance. Additionally, the technique of training an autoencoder together with clustering loss showed better results than training it separately.

6.1 Future Work

Since the autoencoder was trained in combination with clustering, it can be compared with a PCA that was also trained in combination with both, as done by Huang et al. [7].

To see how the proposed autoencoder performs in the real world, training and testing must be performed on real datasets.

To obtain a complete spike sorting pipeline, the results of clustering must be output as spike times as the last step. This would potentially mean transferring the results back to the Matlab implementation.

Since the results from different simulations for training and testing showed a large variance, the robustness of the autoencoder to spike changes needs to be improved for real-world use.

In virtually all scenarios, PCA outperformed the proposed autoencoder at the highest compression rate of 24. It could be investigated what the reasons are for this poorer performance of the autoencoder and whether it is possible to improve the feature extraction capability at such high compression rates.

Bibliography

[1] David Arthur and Sergei Vassilvitskii. *k-means++: The advantages of careful seeding*. Tech. rep. Stanford, 2006.

[2] Hamed Azami and Saeid Sanei. "Three novel spike detection approaches for noisy neuronal data". In: *2012 2nd International eConference on Computer and Knowledge Engineering (ICCKE)*. IEEE. 2012, pp. 44–49.

[3] Matt Brems. *A One-Stop Shop for Principal Component Analysis*. `https://towardsdatascience.com/a-one-stop-shop-for-principal-component-analysis-5582fb7e0a9c`. Accessed: 2020-04-03.

[4] Sayantini Deb. *How To Perform Data Compression Using Autoencoders?* `https://medium.com/edureka/autoencoders-tutorial-cfdcebdefe37/`. Accessed: 2020-03-29.

[5] Sarah Gibson, Jack W Judy, and Dejan Marković. "Spike sorting: The first step in decoding the brain". In: *IEEE Signal processing magazine* 29.1 (2011), pp. 124–143.

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[7] Libo Huang, Lu Gan, and Bingo Wing-Kuen Ling. "A Unified Optimization Model of Feature Extraction and Clustering for Spike Sorting". In: *IEEE Transactions on Neural Systems and Rehabilitation Engineering* (2021).

[8] Sangheum Hwang and Hyo-Eun Kim. "Self-transfer learning for fully weakly supervised object localization". In: *arXiv preprint arXiv:1602.01625* (2016).

[9] Zakaria Jaadi. *A Step-by-Step Explanation of Principal Component Analysis*. `https://builtin.com/data-science/step-step-explanation-principal-component-analysis`. Accessed: 2020-07-01.

[10] Naveen Sai Madiraju et al. "Deep temporal clustering: Fully unsupervised learning of time-domain features". In: *arXiv preprint arXiv:1802.01059* (2018).

[11] Juan Martinez et al. "Realistic simulation of extracellular recordings". In: *Journal of neuroscience methods* 184.2 (2009), pp. 285–293.

[12] Karanveer Mohan. *Visualizing K-Means Clustering*. `https://stanford.edu/class/engr108/visualizations/kmeans/kmeans.html`. Accessed: 2020-04-05.

[13] Carlos Pedreira et al. "How many neurons can we see with current spike sorting algorithms?" In: *Journal of neuroscience methods* 211.1 (2012), pp. 58–65.

[14] Victor Powell. *Principal Component Analysis Explained Visually*. `https://setosa.io/ev/principal-component-analysis/`. Accessed: 2020-06-22.

[15] Mohammadreza Radmanesh et al. "Online spike sorting via deep contractive autoencoder". In: *bioRxiv* (2021).

[16] Abraham Savitzky and Marcel JE Golay. "Smoothing and differentiation of data by simplified least squares procedures." In: *Analytical chemistry* 36.8 (1964), pp. 1627–1639.

[17] Pulkit Sharma. *The Most Comprehensive Guide to K-Means Clustering You'll Ever Need*. `https://www.analyticsvidhya.com/blog/2019/08/comprehensive-guide-k-means-clustering/`. Accessed: 2020-04-05.

[18] Jameson Thies and Amirhossein Alimohammad. "Compact and low-power neural spike compression using undercomplete autoencoders". In: *IEEE Transactions on Neural Systems and Rehabilitation Engineering* 27.8 (2019), pp. 1529–1538.

[19] *timeseriesclassification.com Dataset listing*. `http://www.timeseriesclassification.com/dataset.php`. Accessed: 2020-04-02.