Circuits and Systems
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ens.ewi.tudelft.nl/

CAS-2021-4373715

M.Sc. Thesis

Population Step Forward Encoding Algorithm

L. de Gelder

Abstract

Conversion from digital information to spike trains is needed for Spiking Neural Networks. Moreover, it is one of the most important steps for Spiking Neural Networks. This conversion could lead to much information loss depending on which encoding algorithm is used. Another major problem that can occur in a specific use-case is the limited bandwidth for the spikes that get generated through the encoding algorithm.

In this thesis, we propose population Step Forward Encoding algorithm. This algorithm takes the signal encoding accuracy of Step Forward encoding algorithm and makes it into a population, generating multiple spike trains. This allows a higher threshold to encode a large part of the signal, increasing the efficiency. We show that population Step Forward Encoding algorithm doesn't just work good for the signal encoding accuracy, but also for the classification accuracy. Moreover, population Step Forward Encoding algorithm does not only have a high efficiency with a low spike count, it can also achieve higher efficiency with higher spike count. Thus, population Step Forward can make most use of a limited bandwidth of spikes.

TU Delft

Faculty of Electrical Engineering, Mathematics and Computer Science

Delft University of Technology

Population Step Forward Encoding Algorithm

Improving the signal encoding accuracy and efficiency of spike encoding algorithms

Thesis

submitted in partial fulfillment of the
requirements for the degree of

Master of Science

in

Embedded Systems

by

L. de Gelder
born in Katwijk, The Netherlands

This work was performed in:

Circuits and Systems Group
Department of Microelectronics & Computer Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Delft University of Technology

Delft University of Technology
Department of
Microelectronics & Computer Engineering

The undersigned hereby certify that they have read and recommend to the Faculty of Electrical Engineering, Mathematics and Computer Science for acceptance a thesis entitled **"Population Step Forward Encoding Algorithm"** by **L. de Gelder** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: 8th February 2021

Chairman: ______________________________

prof.dr.ir. René van Leuken

Committee Members: ______________________________

prof.dr.ir. Sorin Cotofana

____

dr.ir. Sumeet Kumar

____

dr. Amir Zjajo

Abstract

Conversion from digital information to spike trains is needed for Spiking Neural Networks. Moreover, it is one of the most important steps for Spiking Neural Networks. This conversion could lead to much information loss depending on which encoding algorithm is used. Another major problem that can occur in a specific use-case is the limited bandwidth for the spikes that get generated through the encoding algorithm.

In this thesis, we propose population Step Forward Encoding algorithm. This algorithm takes the signal encoding accuracy of Step Forward encoding algorithm and makes it into a population, generating multiple spike trains. This allows a higher threshold to encode a large part of the signal, increasing the efficiency. We show that population Step Forward Encoding algorithm doesn't just work good for the signal encoding accuracy, but also for the classification accuracy. Moreover, population Step Forward Encoding algorithm does not only have a high efficiency with a low spike count, it can also achieve higher efficiency with higher spike count. Thus, population Step Forward can make most use of a limited bandwidth of spikes.

Acknowledgments

I would like to thank my supervisors, Amir, René and Sumeet, for their comments and constructive criticism.

I would also like to thank my colleagues at Innatera for their support and advice.

I would further like to thank my fellow Master students for their presence and their help before and during the pandemic.

Thanks to my family and fiends, who provided me with relief, I was able to persevere in this challenging time.

Thank you, God! For giving me the opportunity of the life that has brought me to this point.

L. de Gelder
Delft, The Netherlands
8th February 2021

Contents

Acronyms

AER Address-Event-Representation.

AFR Average Fire Rate.

AI Artificial Intelligence.

ANN Artificial Neural Network.

backprop backward propagation of errors.

BSE Ben's Spiker Encoding algorithm.

FIR Finite Impulse Response.

HSE Hough Spiker Encoding algorithm.

LE Latency Encoding algorithm.

MWE Moving Window Encoding algorithm.

p-SFE population Step Forward Encoding algorithm.

p-TE population Threshold Encoding algorithm.

RMSE Root-Mean-Square Error.

ROE Rank Order Encoding algorithm.

SER Signal-to-Error Ratio.

SFE Step Forward Encoding algorithm.

SNN Spiking Neural Network.

SNR Signal-to-Noise Ratio.

SVM Support Vector Machine.

TBR Threshold-Based Representation.

TCE Temporal Contrast Encoding algorithm.

TE Threshold Encoding algorithm.

List of Figures

List of Tables

Introduction 1

Artificial Intelligence (AI) tries to re-create the intelligence of humans and animals, for instance, detecting and classifying objects of interest. Within the AI-space there are many methods and algorithms, all with their own advantages and disadvantages. This makes it for a designer more difficult to find the best fitting method for their task.

Spiking Neural Network (SNN) is one of the methods within the AI name-space, which try to achieve a highly energy-efficient intelligent behavior. To that end, research into the source of intelligence, the brain, was done. By mimicking the biological behavior of a brain we can create a system, which acts intelligent.

1.1 Motivation

In the society of today saving power is of utmost importance. Computing chips are limited on the power they can use, which is especially prominent in mobile phones and applications. Moreover, transitioning to more renewable energy sources only gets us so far, we should also be using as little power as possible. As AI is getting more accessible and more widely used, making sure it is power efficient is getting more important as well. SNNs are one of the most power efficient methods of AI, thus is it a very interesting and rewarding research field.

1.2 Problem definition

As more spatio-temporal signals are used, use SNNs more inputs and bigger networks as well. Temporal signals can be seen as a variable which changes with time. The spatio-temporal signals are a multitude of temporal signals, which are related to one another. With the more input signals, while keep encoding the same amount of information for each of them, are the number of spikes from encoding algorithm inevitably increased as well.

With the increasing number of input spikes, bandwidth will become an obstacle. The encoding algorithm forms a bridge between the digital spatio-temporal signals and the analog SNN. As the algorithm deals with the digital values, is it (generally) needed to be a digital system itself. Sending the spikes from the encoder to the SNN is usually done with the digital communication of Address-Event-Representation (AER). This digital communication has per definition a limited bandwidth, which should limit the number of spikes created by the encoding algorithm.

There exists a trade-off between the signal encoding accuracy and the signal accuracy per spike. Usually, you could say that with a more accurate signal encoding more information can be used by the SNN allowing the SNN to perform the best. However, with a bandwidth limit on spikes getting the most signal accuracy per spike would help to make sure that the highest signal accuracy is achieved by the spikes that will get through. Moreover, having fewer spikes could make sure that less power is consumed by the SNN. These to cases show that pure signal encoding accuracy might not be best and a trade-off with the signal accuracy per spike should be made.

1.3 Objectives

In light of this problem, we propose with this thesis a new encoding algorithm, population population Step Forward Encoding algorithm (p-SFE). This encoding algorithm take the signal accuracy of Step Forward Encoding algorithm (SFE) and expands its efficiency, which allows it to increase both the maximum achievable signal accuracy and signal accuracy per spike. To show the potential of p-SFE it needs to be compared with other encoding algorithms. The comparison of encoding algorithms is done on three categories: signal encoding accuracy, efficiency and classification potential.

1.4 Contributions

- Design/Introduction of new encoding algorithm, p-SFE.
- MATLAB functions for encoding signals with the following algorithms: BSE, LE, TE, TCE, MWE, SFE, p-TE, p-SFE
- MATLAB scripts to find configurations for the algorithms for both signal encoding accuracy and efficiency, while BSE and LE will have a Genetic Algorithm search within the design space and for the other algorithms the design space can be swept.
- MATLAB scripts to compare encoding algorithms and check their classification potential.
- Code analysis of the encoding algorithms.

1.5 Thesis overview

Firstly, in chapter 2 we will give a short introduction about SNN and explain the other encoding algorithms that p-SFE will be compared to. Secondly, in chapter 3 we will define the metrics and methods used to compare the encoding algorithms and to find the best configurations of the encoding algorithms for these comparisons. Thirdly, in chapter 4 we will give a simple analysis of the codes of those encoding algorithms. Fourthly, in chapter 5 the datasets that are used for the comparison will be given and analyzed as well as the pre-processing steps will be given. Fifthly, in chapter 6 the results from each encoding algorithm will be given and comparisons will be made. Lastly, in chapter 7 will show that the comparisons are in favor of p-SFE and give some open endings for future work.

2 Background and related work

In this chapter we will establish the field of research and explore the related research in encoding algorithms. First we will summarize the research leading up to the Spiking Neural Networks. Secondly we will skim through all the major subjects regarding Spiking Neural Networks. Thirdly, we will go into depth with the encoding of spike trains used by the Spiking Neural Networks.

2.1 Artificial Intelligence

Artificial Intelligence (AI) is getting increasingly more important as it gets increasingly more used in daily life. One of the most obvious examples are voice recognition like Alexa and Siri [32] as well as recommendations by YouTube [10]. There are also examples that are not as easily noticeable, like recommending video's or post in YouTube, twitter or other social media's. These platforms also implement AI to flag uploads containing content against their policies. If we look at Tesla and other electric cars, we see that self-driving cars are becoming more and more a reality [2]. This could even evolve to a point that everyone drives a self-driving car [30]. Not only is AI being used in our daily life, it is also getting more integrated into it.

To define what AI is, we should first look at what Intelligence is. S. Legg and M. Hutter have defined it as “Intelligence measures an agent's ability to achieve goals in a wide range of environments.” [29]. Building on this definition we can say that AI is an artificial agent able to achieve a specific goal within a wide range of environments. This is a very abstract definition. To make it more concrete we can even subdivide the goal as: classification, regression, clustering, optimization, ranking and generation [9]. From this you can see that the goals can be very diverse. This has as a consequence that there are many methods for AI [7], each tailored to certain goals.

2.2 Neural Networks

Neural Networks is one of these methods for AI. This method is specifically good at regression and classification. Neural Networks are based on how brains work. To be more specific the neurons that are the building blocks of our brains. By simulating these neurons and connecting them to one another we create a network for a very specific task, which can perform regression and classification with about the same performance or even better than humans. Neural Networks can generally achieve their goal faster than a human can, making it very attractive to replace humans with Neural Networks where possible.

As Neural Networks try to mimic neurons, we need to know how those neurons work. To understand the human neurons we started to look at the neurons of other animals, such as squids [17] and frogs [20, 21]. As a result, we found that neurons are built as shown figure 2.1. These neurons receive signals through their dendrites from the axon terminals of other neurons. Each pair of axon terminal and dendrite is called a synapse. The axon terminals release neurotransmitters that can open certain receptors in the dendrites. These open receptors let specific ions flow into and out of the neuron, either increasing or decreasing the voltage potential of the neuron. When the voltage potential of a neuron exceeds a certain threshold it will fire and release its own neurotransmitters through the synapses of its axon. The interaction of the synapses can be modeled as an electrical pulse send from one neuron to another.

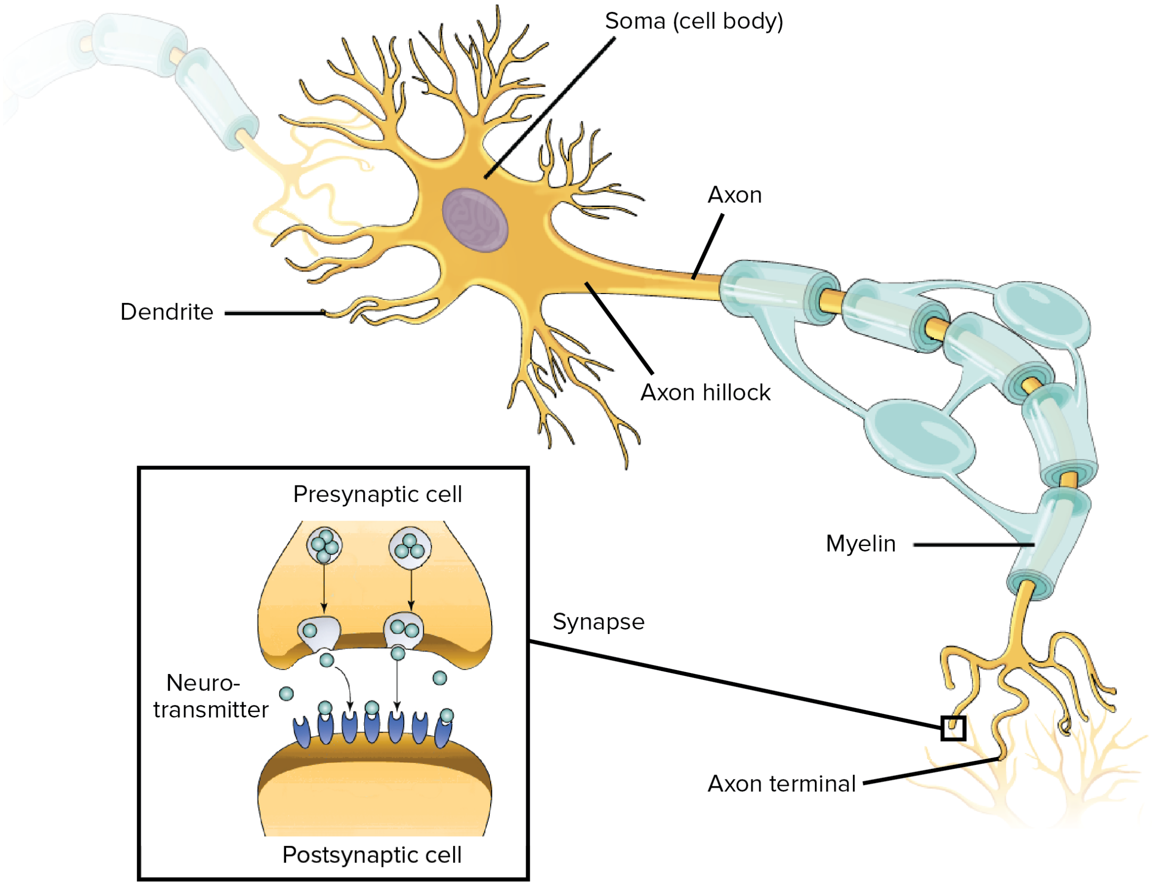

Figure 2.1: Schematic image of a biological neuron. Image taken from “Neuron: Function and its important types” (https://simplestudynet.blogspot.com/2018/01/neuron-function-and-its-important.html) on 2020-11-05.

Through the years we can distinguish three different generations of Neural Networks [31, 14]. The first generation of neural networks is based on the first artificial neuron model from Warren S. McCulloch and Walter Pitts [33]. This generation uses a step function with the threshold: below the threshold 0 as output (no spike) and above the threshold 1 (spike). The second generation introduces a differentiable activation function instead of a 'rigid' step function for the threshold. This allows learning algorithms that are based on gradient descent, like backward propagation of errors (backprop) [38]. Another advantage to the non-binary output is that it can be more easily work with analog systems.

2.3 Spiking Neural Networks

The first two generations have a disadvantage, the state of the neurons do not change with time. As new inputs are presented to a neuron, the output changes. However, changing the order in which those inputs are presented does not change their respective outputs (outside of training). This is not efficient with respect to temporal data, where information is partially stored in time. Changing the order of inputs matters and should therefore give different outputs of their respective inputs. Thus, the neurons of the first two generations don't have a temporal aspect.

Even without a temporal aspect in the neurons, there are networks with a temporal aspect. These networks create a feedback loop, where outputs of neurons will be added to the network inputs for the next time an input is supplied, for example recurrent networks. Such an architecture will ensure that information of past inputs are partially retained. Changing the order of the inputs, will now result in different outputs. A temporal aspect can thus be achieved with such network.

Still, the temporal aspect of these networks don't compare with the temporal aspect of the third generation [31]. In the third generation the neuron model is made more inline with the biological neuron. This makes for more complex models, as time is taken into consideration. These models generate spikes at a certain times, making the times these spikes occur very important (relative to the other spikes). This time dependency of the third generation neurons make for a very strong temporal aspect. As the neuron models themselves already contain a temporal aspect, do networks from these neurons also contain this temporal aspect. For temporal signal the recurrence within a network is thus not necessarily needed.

This third generation has been given a special name, Spiking Neural Network (SNN). The neuron models create series of spikes, also called spike trains, with specific times for each spike. These spikes are central to the generation, as it is the only way information is sent and processed. This makes that the spike times relative to the other spikes are essential to these neural networks. Therefore, Spiking Neural Network is a very fitting name.

Unfortunately, with the more complex neuron models, the method for backprop of the second generation does not apply anymore. This is because the error is not differentiable in the same way. Time needs to be taken into consideration with the backprop. In which case we get for example SpikeProp [5]. This learning method is based on solving the same differential, but with different intermediate steps. However, this method makes some assumptions, like each neuron spikes exactly only once. This restricts the neuron models it can work with. Thus, there have been many neuron models proposed [22, 3] and many learning algorithms accompany them [13, 47, 16, 11].

2.4 Encoding algorithms

As stated in the introduction, encoding and decoding is needed to convert the digital data we normally work with to the trains of spikes the neurons can use. Just like the neuron models are based on findings of research into the biological neurons, are the encoding algorithms based on similar research. On the other hand, the decoding algorithms are designed to invert their respective encoding algorithm. However, as there are multiple neuron models from research, so are there multiple encoding algorithms from research. We will differentiate two groups of encoding algorithms: rate encoding and temporal encoding [8], see figure 2.2. Beside those groups can be a modification of an encoding algorithm, population encoding [35], which transforms information of a single input into multiple spike trains. We will first briefly discuss the rate encoding, before going into more detail of the temporal encoding algorithms. Which will bring us towards two population encoding algorithms, containing the proposed population Step Forward Encoding algorithm.

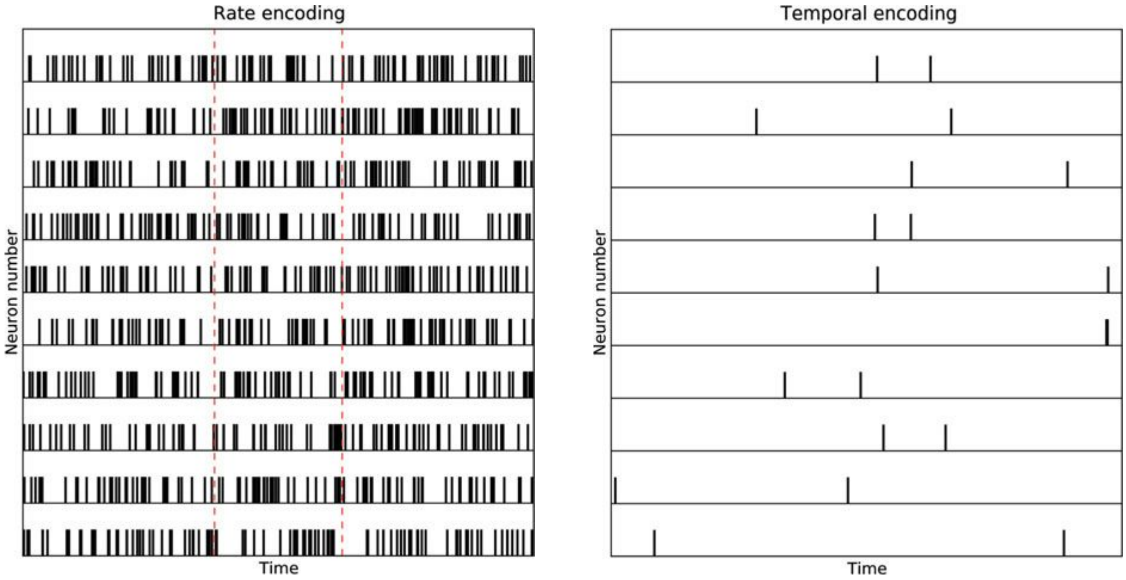

Figure 2.2: Rate encoding and temporal encoding examples. Image taken from [25] p.130.

2.4.1 Rate encoding

Earlier studies of the brain have indicated that they likely work based on the firing rate of the neurons [4]. In SNNs this is implemented as Rate encoding. Rate Encoding transforms each new input value to a frequency of spikes, basically a frequency modulation.

The biggest advantage of Rate Encoding is that the SNNs can be converted from a neural network of the second generation. The numbers that the second generation neurons output can be seen as the firing rate of that neuron. The synaptic weights can therefore be translated to work with SNN neurons. This means that an ANN can be trained and then transformed into a SNN with roughly the same performance [28].

This conversion is a big advantage as we can make use of the backprop that the second generation has, which is simpler to implement.

This encoding schema has unfortunately an inherent downfall. To get the information of frequency, several periods need to have elapsed. This means that multiple spikes are needed to encode a single number, which makes it less efficient than other encoding algorithms per definition, for example temporal encoding [8]. Furthermore, as multiple periods need to have elapsed, the response time is generally much larger, thus less information can be processed in the same time frame. To summarize Rate Encoding is generally not a great option and therefore will not be compared to other algorithms in this thesis.

2.4.2 Temporal encoding

2.4.2.1 Rank Order Encoding algorithm

For Rank Order Encoding algorithm (ROE) [41] the main concept is that systems will need to act on the information which is provided the earliest. The first information received is thus the most important. Therefore, later received information should be given less attention. To do this ROE uses an extra weight, where each input will also get multiplied with this extra weight to the power of the rank in arrival time. This extra weight must be between 0 and 1. For ROE the potential of a neuron i at time t is given by:

$$potential(i,t) = \sum_{j \in [1,m]} w_r^{order(a_j)} w_{j,i} \tag{2.1}$$

where there are m neurons with a dendrite to neuron i with a weight of $w_{j,i}$ and rank of arrival is noted as $order(a_j)$. The extra weight is given by w_r.

ROE has quite an advantage over Rate Encoding as it doesn't need several periods, but it still does have its own limitations. With the introduction of ROE, the theoretical information per neuron and spike was calculated. Here it was found that Rank Order would outperform Rate Encoding, but not Temporal Encoding. It was argued that It would still be "a good second best" [41]. This still means that ROE is inferior to Temporal Encoding with respect to efficiency.

ROE will also not be compared in this thesis, for two reasons. The first reason is that it is, as just stated, theoretically inferior. The second reason that it is not as much used for encoding as it is used for decoding in a learning rule [43]. This means that transforming a floating-point number to a spike train with Rank Order is not trivial [12]. These two reasons show that implementing these algorithms will be tough, while they are unlikely to perform better than others.

2.4.2.2 Ben's Spiker Encoding algorithm

Ben's Spiker Encoding algorithm (BSE) [39] is based on the Hough Spiker Encoding algorithm (HSE) [19]. These algorithms use a filter with the idea that the signal will also directly be filtered. These algorithms generally work by checking if the filter fits below the last values of the temporal signal. If so, a spike will be generated and the filter will be subtracted.

BSE improved on HSE by even allowing it to spike if the error between the filter and the signal is not too big instead of the filter fitting below the signal. This is done by checking the difference between two sum of the absolutes; the signal values and the error between the signal and the filter. The algorithm will spike if the difference between these values are below a certain threshold.

The improvement comes with a cost however. BSE uses an extra threshold besides the filter. This extra variable complicates the process of finding an optimal configuration of this algorithm. More computational power will thus be needed.

Unfortunately, designing a filter is not so straight forward. This is because the filter is not used as a regular FIR filter, as the values of the signal are not being convoluted with the filter. This would have given floating-point values again instead of the needed spike train. The act of checking whether the filter fits below the signal will thus throw away a lot of information. Finding a filter which highlights the most important information is therefore done by using a Genetic Algorithm. Using a Genetic Algorithm however does not guarantee that an optimal filter will be found, not to mention the most optimal filter.

2.4.2.3 Latency Encoding algorithm

Latency Encoding algorithm (LE) [35] encodes each input value of the signal to a relative spike time. This relativity is with respect to the moment the value was supplied. With a low value the spike should be generated very quickly, while for a high value the spike should be generated much later. However, in the case of a later spike you still want it to spike before the next input arrives lest it interferes with the next input, so a maximum spike time should be set. In the case of a real-time application, there is also some delay between input arrival and for it to be transformed to a spike time, thus a minimum should also be set. We will combine these minimum and maximum into a spike window. To transform an input value to the spike window, we can then also use a signal window to map from one to another.

The advantage and disadvantage of this algorithm lies in the theoretical near infinite signal encoding accuracy. If all values are within the signal window, they can all map perfectly into the spike window. To decode these spikes we can map them back with perfection, theoretically. However, if the spikes deviate slightly, the decoded value will also deviate from the original value. This deviation in spike time will limit the maximum signal accuracy achieved with this algorithm. As the hardware/software that creates and reads the spikes gets more accurate, so does the signal accuracy achieved through this algorithm. Thus, the hardware/software will determine the signal accuracy of the algorithm, which in theory could be a perfect accuracy.

2.4.2.4 Threshold Encoding algorithm

With Threshold Encoding algorithm (TE) the temporal signal is being approximated a bit similar to BSE. A threshold is specified and when the new value of the signal exceeds the threshold, while the previous value did not, a positive spike is generated.

If the signal than get beneath the threshold, while the signal was previously above that threshold, a negative spike is generated.

Because of its simplicity it has a big advantage, but also a disadvantage. With this method you only need to remember whether the previous spike was positive or not. This amounts to only a single bit needed in the memory for encoding. Unfortunately, the simplicity has a disadvantage during decoding. We only know that the signal is between the threshold and either positive or negative infinity. Thus determining the original value is theoretically impossible. However, with a bounded upper and lower limit, the best estimation of the original value would be the middle of the threshold and either the upper or lower limit (depending on whether a positive or negative spike was last), making it possible to estimate the original values and calculating the signal encoding accuracy. Still, these bounds need to be defined based on the structure of the input values, if even possible. It should also be noted that even with the bounds, only two values can be encoded, while bigger datasets usually (also) contain non-boolean data. To summarize, the simplicity has a low implementation cost, but creates an inaccurate signal reconstruction.

2.4.2.5 Temporal Contrast Encoding algorithm

Although Temporal Contrast Encoding algorithm (TCE) [37][1] also uses a threshold, is that threshold used differently. TCE doesn't really encode the signal directly. It takes the difference with the previous input value and spikes if that difference exceeds the set threshold. This could be a positive spike if the difference is higher than the positive of the threshold, or a negative spike if it is lower than the negative of the threshold. Unlike TE, TCE spikes for every time the difference is either higher than the positive or lower than the negative. As TCE only looks at the difference, is it actually encoding the differential of the input signal.

The fact of encoding the differential and not the actual signal is a big disadvantage of this algorithm. Reconstruction of the original signal can prove to be more and more inaccurate in time. For example assume a signal with a slight drift, smaller than the threshold. In that case TCE will not spike for that drift. When decoding the resulting spike train, the drift will thus barely be present in the decoded signal. This means that information is lost, but it not only happens with the drift. This information loss happens in general, as all differences below the threshold is discarded. The accumulated information discarded this way can really add up to a large decree over time.

A small advantage of this encoding algorithm lies in the easy, straightforward implementation. Only the previous value needs to be remembered to calculate the difference. Moreover, there are only three operations needed; calculate difference, check positive threshold and check negative threshold. This is much less than BSA, where each filter value has several operations.

[1] In the referenced paper TCE is called Threshold-Based Representation (TBR). Although it does use a threshold, so does the Step Forward encoding Algorithm. However, with TCE the contrast, the difference, is central. Moreover, without threshold in its name, is it more distinctly different from the described Threshold encoding Algorithm.

2.4.2.6 Moving Window Encoding algorithm

Using a Moving Window Encoding algorithm (MWE) [24] is very similar to TCE. However, with MWE the previous values are subjected to a moving window filter. The difference is then calculated between the new value and the filtered previous values. If we look at the drift issue again, we see that the filtered previous value is likely to be further from the new input value, thus it is more likely to generate a spike. Through this algorithm it is slightly more likely to decode correctly.

The drawback of this algorithm is likely to be more problematic than the added signal encoding accuracy. To calculate the average of the moving window, all values within that window need to be remembered. Of course, this can be done with a shifting memory array, but it will still be multiple times bigger than the memory needed for TCE, where only one value needs to be stored in the memory. The area and energy (hardware implementation) or computation time (software implementation) will be bigger than TCE.

2.4.2.7 Step Forward Encoding algorithm

With Step Forward Encoding algorithm (SFE) [24] the implementation is again similar to TCE, and thus MWE as well. Instead, now the difference with the previous value is taken with a baseline, which is remembered and not changed if no spike is generated. When the difference between the current signal value with the baseline exceeds a given threshold, a spike is generated and the threshold is added to the baseline. Just like TCE, this goes both positive and negative, so if the difference is bigger than the positive threshold a positive spike is generated, while a negative spike occurs for a difference smaller than the negative threshold. With this algorithm the actual signal is encoded instead of the differential.

The resulting advantage of SFA unfortunately also causes a slight disadvantage. The advantage is that the signal encoding accuracy is much better as the decoded signal usually don't deviate more than the threshold value. Even if it does, the next new value will likely fix it. Even though the signal can now be much better reconstructed, the number of spikes that are encoded also increases. This could have as disadvantage that the information per spike on average, the encoding efficiency, becomes less.

2.4.3 Population encoding

As stated before, some encoding algorithms have a population variant. The main idea with population encoding is that instead of a single spike train per input signal, each input signal is transformed to multiple trains. This can have the advantage that it will be easier for the first layer of the SNN to distinguish between certain aspects of the input(s).

However, population encoding has a common disadvantage. With encoding multiple spike trains, generally more spikes are generated as well. Even though the spatially separated spikes help with training, they can also be less information efficient as the spikes added might exceed the information that is added and reduce the average information per spike. Moreover, it will generally also take more resources and time to generate spikes for each spike train, which needs to be taken into consideration.

In this thesis there are two population variants explored. First variant is from TA, the population Threshold Encoding algorithm (p-TE) [35]. The second explored variant is the proposed encoding algorithm, the population Step Forward Encoding algorithm (p-SFE). p-TE will be discussed in this section, while p-SFE will be discussed in section 3.1.

2.4.3.1 population Threshold Encoding algorithm

The population variant from TE uses multiple thresholds instead of only one. Each threshold creates its own spike train and is checked with each new value. These checks are independent of each other and can thus be done parallel each other.

The advantage from p-TE does not only lie in the multiple spike trains, but also in the improved decoding. As we found with TE, it has the problem that the upper and lower bounds can go to infinity. On the other hand with p-TE, the multiple thresholds make sure that when a threshold is passed, there is generally a threshold bigger and/or smaller. This means that the signal needs to be between those thresholds, thus the resulting decoded value will be much more accurate and much less ambiguous.

Methods 3

In this chapter, the methods used in the thesis are discussed except for encoding algorithms other than population Step Forward Encoding algorithm (p-SFE) which are discussed in 2.4. First we will talk about the proposed encoding algorithm, population Step Forward Encoding algorithm. Secondly, we will define the methods to compare configurations and different algorithms. Thirdly, we will show how the most optimal configurations are found.

3.1 population Step Forward Encoding algorithm

USPTO patent number: 63/146,587

The algorithm proposed in this thesis, p-SFE, uses multiple thresholds instead of one, each with its own spike train. The multiple thresholds is where p-SFE gets the population from its name, similar to p-TE. The thresholds should be put in descending order. Then from the biggest onward the difference between the baseline and the new value should be checked. When the difference exceeds the threshold a spike should be generated and the threshold should be added to the baseline, just like SFE. Only after the bigger threshold is checked should the smaller thresholds be checked, because the baseline could have changed. An example for how p-SFE works is given in figure 3.1. As the spikes of the smaller thresholds get combined into a single spike of the bigger threshold, are the number of spikes reduced while the same information is retained, thus is the efficiency increased compared to SFE.

A disadvantage of p-SFE is that the thresholds need to be checked sequentially. p-TE can parallelize the checking of the thresholds, because they are independent of each other. For p-SFE this is not possible, as each threshold can change the baseline, can the spike of one threshold make sure the next threshold won't spike. As parallelization is not possible, using more compute-units to accelerate the algorithm is not possible, where with p-TE that is possible. The number of thresholds that can be check is thus limited by the speed of the compute-units.

Although you could choose the thresholds for p-SFE arbitrarily, a simplified version of p-SFE makes it easier to choose the thresholds. The simplified version will assume a specific relation between the different thresholds. The simplification makes sure that only the base threshold and the number of thresholds is needed to be specified, thus can the configurations be swept, otherwise more difficult methods for finding an optimal configuration is needed, such as genetic algorithms. Simplified p-SFE is also used in this thesis, where the thresholds are set to half of the previous threshold. In the case of example figure 3.1, would the configuration amount to two thresholds with the base threshold of four.

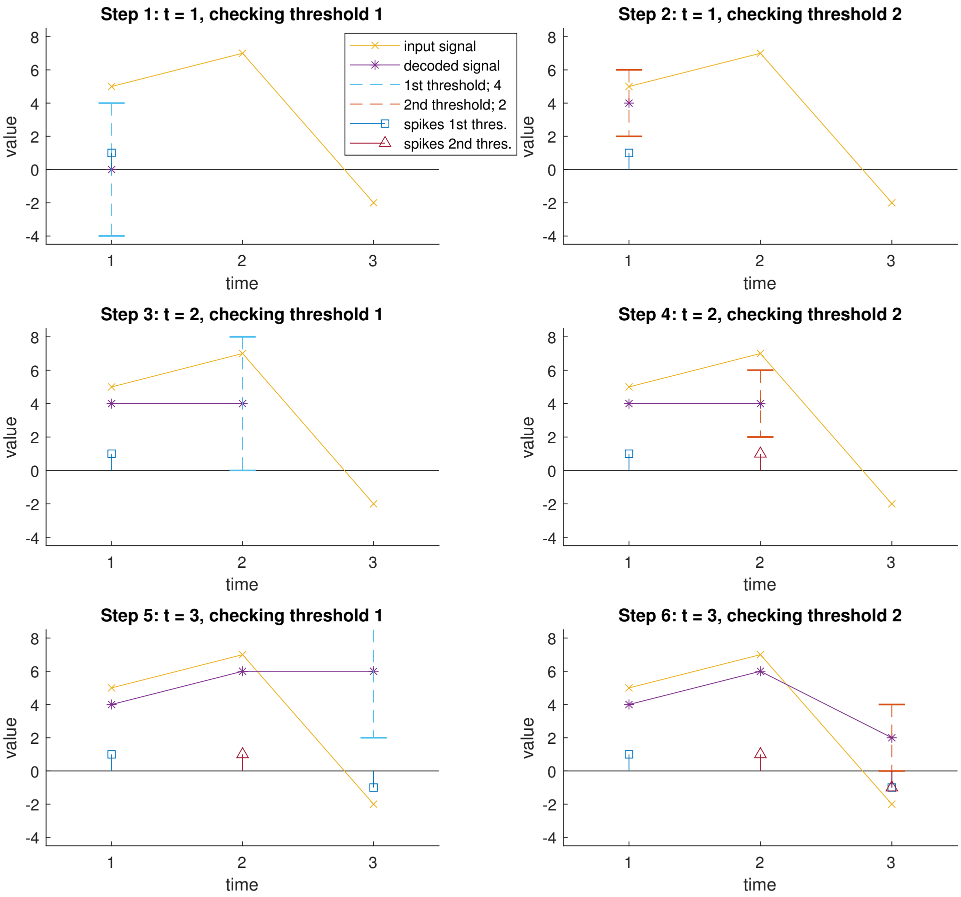

Figure 3.1: An example for p-SFE, where the input signal is supplied with the following values; 5, 7 and -2. In addition, p-SFE is configured with two thresholds; 4 and 2.

3.2 Metrics

In this section the metrics we will use are discussed. These metrics will be used for Encoding Accuracy, Efficiency and Classification Potential. These metrics are needed, because the spike trains from the encoding algorithms are not intuitive for us, as we generally are used to numbers. Therefore, after encoding to spike trains will the trains directly be decoded back to a signal consisting of regular values (in our case floats), see figure 3.2. The reconstructed signals can be compared with the original signals and the number of spikes to measure the signal Encoding Accuracy, Efficiency and Classification Potential.

3.2.1 Signal encoding accuracy

In [37] several metrics for determining signal encoding accuracy are used, and we will refine their choices. Using multiple metrics can give redundancy, but can also confuse

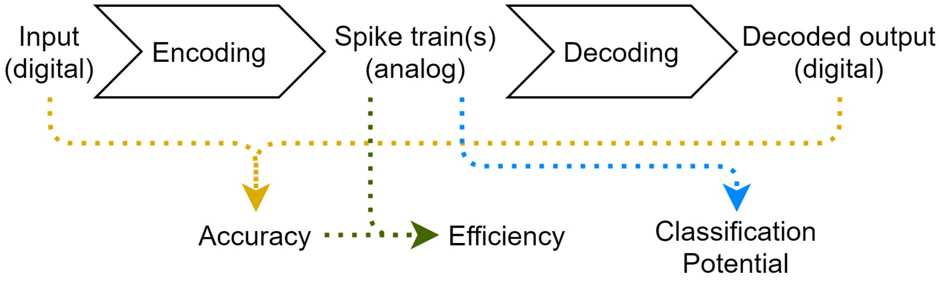

Figure 3.2: The signal encoding accuracy metric will be based on the input signal and the decoded output signal. The efficiency will be based on that signal accuracy combined with the generated spike train(s). On the other hard, the Classification Potential is based on the generated spike train(s) alone.

someone of which metric is *the* indicator. Thus, our goal is to establish one metric to be used for signal (encoding) accuracy. First we will evaluate the metrics from [37], see equations 3.2, 3.3 and 3.4. Unfortunately, we will see some small issues with each of them. Thus, we will slightly change the equation 3.4 and use that as the metric for signal accuracy.

The first metric with an issue is the coefficient of determination, equation 3.2. This metric is usually called R-squared and describes how much of the variance can be explained. The definition of this metric is given in equation 3.1. In this equation SS stands for Squared Sum and RC for Remaining Components. Moreover, the R in SSR stands for Remainder, the T in SST is for Total and the E in SSE for Error. Usually, the RC cancel out one another and the equation can be written as in the last step is shown. The problem with the metric in [37] is with this simplification.

$$R^2 = \frac{SSR}{SST} = \frac{SST - SSE - RC}{SST} = 1 - \frac{SSE}{SST} \tag{3.1}$$

As can be seen in equation 3.2, is the simplified R-squared metric used. The simplification assumes that RC equals to zero, but from tests with the datasets and the encoding algorithms we find that this assumption might not be valid. This is most likely caused by the fact that the time samples depend on one another, while this metric is usually used for independent measurements. As the simplification is wrongly applied, equation 3.2 does not correspond to the original intention of the metric. On the other hand, this doesn't mean that the metric shouldn't be used, but there is a more trustworthy metric as we will see later.

$$R^2 = 1 - \frac{\sum_t [signal_{input}(t) - signal_{decoded}(t)]^2}{\sum_t [signal_{input}(t) - mean(signal_{input})]^2} \tag{3.2}$$

The second metric with a slight problem is the RMSE, see equation 3.3. It roughly describes the average error between the input and the decode signal. The normalization (average) is done through the length of the signal, the number of time samples. There lies the slight problem, because the length of the signal can be arbitrary. For example if we extend the signal with noise, some encoding algorithms will be able to encode the noise much better than others, while they could encode the spoken audio much worse.

Resulting in skewed metric values, because of the normalization with the signal length. Though the described problem might not prop up, there is still a metric without such an issue.

$$RMSE = \sqrt{\frac{\sum_t [signal_{input}(t) - signal_{decoded}(t)]^2}{N_{time-samples}}} \tag{3.3}$$

The last metric used in [37] is Signal-to-Noise Ratio (SNR), see equation 3.4, and will be the base of the metric used in this thesis. This is also a widely known metrics where both the signal and noise should be known. In this case the noise is defined as the error of the decoded signal. However, in that case it should be called Signal-to-Error Ratio (SER) to prevent the confusion that the error could be defined as Gaussian noise, as usually is the case. The biggest advantage over the other two equations 3.2 and 3.3 is that this metric is not influenced by the length of the signal, not even through the mean of the signal.

$$SNR = 20 \cdot log(\frac{\sum_t [signal_{input}(t)]^2}{\sum_t [signal_{input}(t) - signal_{decoded}(t)]^2})[dB] = SER_{input} \tag{3.4}$$

Even tough so far we have seen that equation 3.4 would be the best choice, there is one last potential improvement to make. For this improvement we will take the decoded signal as a reference instead of the input signal. With a higher SER is the signal even more prominent with respect to the error, which is thus better. In the case that the signal input is used, the numerator stays the same for each encoding algorithm. If instead the decoded signal is used, the numerator is bigger for a higher amplitude signal. Combined with the need for a bigger SER, the encoding algorithms are being forced to encode a bigger signal. As most encoding algorithms tend to encode a signal below the input, will the bigger signal undoubtedly be closer to the input. Thus, would this create a second component in the metric which encourages an encoding closer to the input, creating a bigger contrast.

$$SER_{decoded} = 20 \cdot log(\frac{\sum_t [signal_{decoded}(t)]^2}{\sum_t [signal_{input}(t) - signal_{decoded}(t)]^2})[dB] \tag{3.5}$$

3.2.2 Encoding efficiency

To define a metric for encoding efficiency, we first need to establish what efficiency we talk about. In the introduction we showed that the bandwidth for sending spikes can become a problem. In this case the most information per spike is preferred, as that would mean that the most information is encoded with the limited number of spikes. Therefore, the metric for efficiency we look for is information per spike.

For the spikes part we can take a look at another metric from [37], the Average Fire Rate (AFR). In [37] this AFR is the number of spikes divided by the number of time samples. This should actually be called the probability to spike, P$_{spike}$, see equation 3.6. If that spike probability is then divided with the sample time, would it give the actual AFR, because that would amount to the total number of spikes divided by the total elapsed time, see equation 3.7.

$$P_{spike} = \frac{\sum_t spikes}{\sum timesamples} \tag{3.6}$$

$$AFR = \frac{\sum_t spikes}{t} = \frac{\sum_t spikes}{T_{sample} * \sum timesamples} = \frac{P_{spike}}{T_{sample}} \tag{3.7}$$

Combine the spike probability with the SER for signal encoding accuracy we can make an estimation for efficiency. We can say that the SER for signal accuracy is also similar to the information that is encoded. As the decoded signal is more similar to the input signal, caries the decoded signal more of the same information as the input signal. Unfortunately, the SER is a normalized metric for the entire signal, so we need to transform it to a metric per spike. We can use the AFR for that. Moreover, we can make it even simpler with the dataset we use, because it has a constant sampling rate for all signals. As the sampling rate is constant, and we are only interested in a comparison, can we use the spike probability. The resulting metric we will use for encoding efficiency is thus given by

$$EncodingEfficiency = \frac{information}{spike} \sim \frac{accuracy}{\Sigma AFR} \sim \frac{SER}{\Sigma P_{spike}} \tag{3.8}$$

where the AFR is summed, because population encoding algorithms would create multiple spike trains, each with their own AFR.

3.2.3 Classification potential

So far we have looked at how much of the input signal is encoded, however we should also look at the spike trains themselves. As the spike trains will be used by the network, are the spike trains themselves important to look at. Especially for classification problems, since only a small portion of the encoded signal is usually already enough for correct classification.

The biggest problem however is that the spike trains aren't easily comprehensible for us, so for classification problem we need a special method to test the algorithm capabilities. The straightforward way would be to directly test the classification by training an architecture and evaluate the classification accuracy. Unfortunately, this does pose a potential biasing problem, as some architecture training methodologies might have a bias towards certain encoding algorithms as they generally are designed with a specific encoding algorithm in mind. Because of this biasing problem, we will propose a method to estimate potential for classification problems.

3.2.3.1 Accuracy

There are multiple ways to calculate the 'distance' between different spike trains [44, 42, 26, 27], so spike trains belonging to similar classes should have smaller distances than spike trains from different classes. These spike train distances can be used to find a distribution density for each class combination. If the similar class spike trains indeed have a smaller distance, then the distribution density for spike trains of the same class would have more mass towards zero than the other distribution densities. The points below which the same class densities clearly have the most mass, can be used to calculate the theoretical optimal classification accuracy.

Unfortunately, using such a method did not work in this case. We found that the distribution densities were overlapping very much with one another, regardless where it were same class densities or not. This proved finding areas were the same class densities exceeded other (relevant) densities difficult if even possible. When such areas were found, were they small and usually not towards zero. Whether such areas would describe the classification potential with such spike trains, is therefore questionable.

So we turn to other AI methods to make a rough estimation on the classification potential. These other methods are not as temporally based as SNNs are, thus the temporal aspect of the signal will be transformed to a dimensional aspect usable by the other AI methods. This transformation is done by regarding each time sample as a separate feature, creating one vector for each data signal with many dimensions. However, the fact that spikes only mean something relative to one another causes a problem. Shifting these spikes in time equally should thus not impact the outcome. While, for other AI methods this shift can make a huge difference in the outcome. To minimize this issue we will normalize the spike trains in time relative to *all* other spike trains (of the same encoding algorithm), see subsection 5.1.3.

To work with these highly dimensional data samples, we choose to use the Support Vector Machine (SVM). The reason behind this is that SVM have proven to work extremely well with highly dimensional datasets [6]. An SVM creates a hyperspace as boundary, while trying to maximize the margin between that boundary and the samples of the classes. To do this the boundary is described with even more dimensions, so it becomes very flexible.

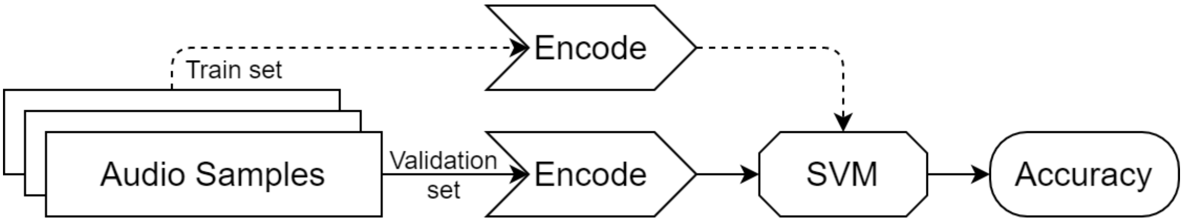

Figure 3.3: Overview of the classification methodology, where SVM stands for Support Vector Machine (SVM). This SVM is trained on the spike trains of the train set. Then the spike trains of the validation set are used to calculate the classification accuracy of the SVM on the spike trains.

Using the SVM is as normal and thus very straightforward, see figure 3.3. We separate the audio samples into sets (train, test and validation). The data from the sets is encoded into spike trains. Then all spike trains are converted such that they can be used by the SVM effectively. From the sets are the spike trains of the train set first used to train an SVM. After which, the validation set can then be classified with the SVM, resulting in a confusion plot and a classification accuracy. This classification accuracy will show us the potential for classification of these spike trains. An algorithm which would score higher in this accuracy is more likely to work better for classification problems.

3.2.3.2 Noise resistance

Though checking the classification potential doesn't end with just this; the resistance to noise should also be checked. Noise is unavoidable in any system, as even every resistor creates some noise. As every component adds some noise, combined they could even overtake the signal of interest in power. For classification could this mean that the noise would impact the class prediction more than the signal we should be using for the prediction.

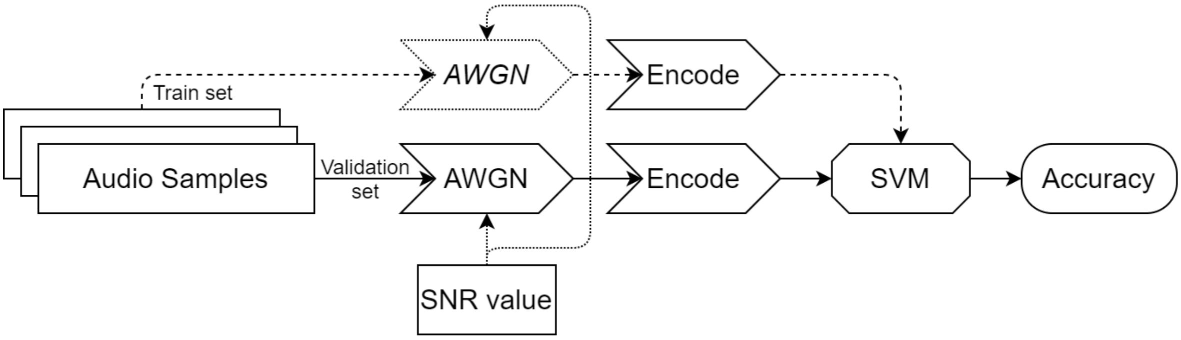

Figure 3.4: Overview of the classification methodology with noise. Here AWGN stands for 'add white Gaussian noise'. If no noise is added to the train set, you speak of miss-matched training, which is the worst case scenario. If noise is also added there, then you speak of multi-condition training.

There are two ways to take noise into consideration; miss-matched and multi-condition, see figure 3.4. The first way, miss-matched, you assume that the noise can not be learned with. This could be caused by not knowing the exact noise during training. It would be the worst case and can be modeled by adding noise to the input signal of the validation set, before encoding and validation. The second way, multi-condition, you assume the noise is known while training. In this case the noise is also added during training, which should mean that the trained network can take the noise into consideration as well, making the network more robust to noise. In this thesis multi-condition is used as it is commonly used.

Besides noise in these condition, the classification 'accuracy' with only noise supplied is also quite useful. The classification accuracy with only noise as in put shows how good the SVM can guess the classes without any information. The SVM could than just randomly guess or always guess the same class. In both cases the probability of guessing correctly would be the same (if all classes occur equally). If you set the classification accuracy achieved through this method as a baseline for the achieved classification accuracy for miss-matched or multi-condition training, you can find at which Signal-to-Noise ratio the SVM degrades to random guesses, for an example see figure 6.2.

3.3 Encoding algorithm optimization

From the related work we have seen that every encoding algorithm has its own configuration that needs to be set. In most cases we talk about a threshold that needs to be set, but there are also cases where that is not enough. For example MWE needs a filter length as well and BSE even needs an entire filter to be specified. On the other hand some encoding algorithm configurations don't even need a threshold, but other values need to get specified. For example LE, where the spike window and value window are needed. To find the most optimal configuration of these encoding algorithms different methods will be needed.

3.3.1 Configuration sweeping

For most algorithms only one or two values need to be configured, these can easily be swept for the most optimal configuration. Sweeping just amounts to first setting bounds, then picking points within those bounds (usually linearly spaced) and lastly testing which of those points result in the best configuration. If the bounds are chosen correctly, you see that around the points closest to the most optimal configuration are only slightly worse and better than the points further from the most optimal configuration. If the most optimal configuration is near a specified bound(s), it is a good idea to stretch those bounds further as to check whether there might be a better configuration is that direction.

However, other algorithms have more than two values to configure for which another method is preferred. Sweeping with only one value puts the number of points to search in a line. If two values need to be swept, the search points are put in a square. With three values are the search points put in the volume of a cube, while for even more values the points are space within a hyperspace. This means that the number of search points grow exponentially with the number of configure values. Thus, the computational complexity grows with the increasing configurable values.

3.3.2 Genetic algorithms

As an alternative, we use genetic algorithms [46] for configurations of more than two values. A genetic algorithm views the configuration values as genes of a chromosome, such that every chromosome represents a configuration. At the start, a population of (semi) random chromosomes is created. Each chromosome of the population should be evaluated (just like described in section 3.2), and from that a value should be given of how good those individual chromosomes perform in the desired task, the fitness. Chromosomes with a better fitness will get a higher chance to propagate to the next generation as well as becoming a parent to the next generation. This will mean that some chromosomes won't propagate, while the population size should be maintained, so new chromosomes need to be generated for which parents will be used. To generate these new chromosomes operations will take parts from these parents and slightly change them. These operations try to combine the best parts of the parents or randomly slightly change values, in search for better fitting chromosomes. Combining the operations with the selective nature of fitness, makes that a new population contain chromosomes with a good chance to achieve a higher fitness. Eventually, a new population won't or barely increase the fitness, at which point an optimal configuration would be found by the best fitting chromosome of that last generation.

We need to design the operations which will create new chromosomes. Unfortunately, these operations need to be designed for each genetic algorithm separately, because the interpretation of (certain) genes can be different. Some gene interpretations might have a limit or other limitations, which might need to be taken into consideration during these operations. However, these operations aren't arbitrarily designed. They should and are inspired by biological operations, which happen in DNA. These operations can for example be: mutation, crossover and inversion. In this thesis operations are designed for genetic algorithms of both BSE and LE.

3.3.2.1 Ben's Spiker Encoding algorithm

The chromosomes for BSE will be constructed with one gene for the threshold and an 'arbitrary' number of genes for the filter. This will mean that the first gene should not mingle with the latter genes, as the ranges of the values don't compare with one another. For this genetic algorithm several operations were defined; crossover, length extension/subtraction, mutation. Multiple operations can act on the same child.

- The crossover operation will need two parents. These two parents can have different filter lengths. Therefore, a random number between 0 and 1 is picked as crossover-point. The crossover-point will be multiplied by the filter length of both parents and then those values will be rounded to get the indexes to divide the filters of both parent in two parts. The filter of the child would have the first filter part of the first parent and the second part of the second parent. The crossover-point will also influence the threshold, as it would be used as a linear interpolation between the threshold of the first and the second parent.

- For length extension/subtraction the number of filter values to add or remove is picked. Then either adding or subtracting is picked with same probabilities. Removing filter values is done fairly simple by selecting indexes at random and discarding the filter values at those indexes. When adding filter values, it will help to add values in a similar range as the rest of the filter values, thus the mean and the standard deviation of the filter values is calculated. Based on the mean and standard deviation random values with a normal distribution will be added at the back of the filter.

- Mutation of the chromosomes is done fairly simple as well by adding 'noise'. This noise also has a normal distribution with a standard deviation based on the current filter values, but with a mean of zero. This standard deviation is first also multiplied with a factor, which should be kept between 0 and 1 to avoid adding so much that the noise becomes dominant (resulting in no converting solution). The noise added to the threshold is done slightly differently; another factor is used and a random value between the negative and the positive of that factor is picked with a uniform probability distribution.

To get the configurations for BSE the probabilities for these operations have been set among other settings. The crossover operation has a probability of 0.3, while mutation has a probability of 0.95. For length extension/subtraction there is a probability of changing the size by either one or two, with those probabilities being 0.1 and 0.05 respectively. Tough the size has also been limited to a minimum of 3 and a maximum of 30, as smaller could hardly be called a filter and bigger would probably need a bit too much computation. Another limitation is the number of generations have been limited to 1000 with twenty chromosomes each generation. To make sure that the population will not degrade the best three fitting chromosomes will automatically propagate tot the next generation.

3.3.2.2 Latency Encoding algorithm

Unlike the 'arbitrary' chromosome length of BSE, the chromosome length of LE is always 4. The first two values describe the signal window with a minimum and a maximum, while the last two values describe the spike window also with a minimum and maximum. If the signal sampling time is constant, the values for the spike window can be normalized. Then the spike window can be bounded by 0 (start of the signal sample) and 1 (end of the signal sample). It is also preferred that the signal window values are bounded by the minimum and maximum the signal can take, because it does not make sense that these bounds are beyond the minimum and maximum of the signal, as this would get the same result as shortening the spike window. Another difference with BSE is that there are only two operations used; crossover and mutation.

- Where crossover with BSE is at an arbitrary point, with LE takes the signal window of the first parent and the spike window of the second parent.
- For mutation random values are added. These random values are based on the size of the windows. A factor (between 0 and 1) is multiplied with those window sizes and the random value is chosen with a uniform probability distribution between the negative and the positive of those factorized sizes.

The settings used to get the configurations for LE are mostly similar to those for BSE with only a few differences. The number of generations is reduced to 500, because there are fewer number of variables which make the configuration converge faster. Thus, fewer generations are needed. Besides that, the size changing is not needed, so there is no probability for that. This also mean that no limit on the size is needed, however a limitation on the values of the windows is very useful. With the spike window bounded by zero and one, while the signal window can be limited by the maximum and minimum the signal could take.

Code analysis 4

In this chapter we will analyze the code used for the encoding algorithms into more depth. These MATLAB functions will be transformed into pseudo-codes as an intermediate step. Next, these pseudo-codes are analyzed for which operation are used, how much and how they scale. The operations used by the encoding algorithms will cost a certain amount of energy and limit how much can be encoded within a given time, thus the operations can be seen as a cost of the encoding algorithms.

4.1 Pseudo-codes

Each of the pseudo-codes, see figures 4.1 to 4.8, have been created with a similar structure. They are made for a single input stream, as the computational complexity scales linearly with the input streams, because the input streams can be encoded in parallel. It is assumed that the signal input will not contain NaN values, because these values take generally much less computation to process and will thus not take part in the limit of encoding values within a specified time.

The MATLAB codes for BSE, see A.1.1 and A.1.2, can be changed to the pseudo-code given in figure 4.1. We can assume that the cutoff will either not cost any computations or be insignificant in the case of a significantly longer signal than the filter. First, we need to note that the last L minus 1 values need to be remembered. Then, to calculate the errors L subtractions, two times L absolute operations as well as L minus one times 2 summations. After that, in the worst case one comparison, L subtractions and L plus one writing operations are still needed. However, in this equation the differences between the 'latestSignal' and the filter is calculated twice, but this could also only be done ones which would mean that L subtractions less are needed and L values need to be temporarily remembered.

```
Input: signal → a T-length vector containing the signal values in chronological order
       filter → a L-length vector containing the BSA filters, where L is the length
       for the filter
       threshold
Output: spikes → a T-length boolean vector containing at which signal values to
        spike
foreach T → t do
    if t < L then
        cutoff filters to length t, so L = t;
    end
    latestSignal = signal(t-L+1 ... t);
    E1 = $\sum_L$ ||latestSignal - filter||;
    E2 = $\sum_L$ ||latestSignal||;
    if E1 $\leq$ E2 - threshold then
        spikes(t) = true;
        signal(t-L+1 ... t) = latestSignal - filter;
    else
        spikes(t) = false;
    end
end
```

Figure 4.1: Ben's Spiker Encoding algorithm pseudo-code

The MATLAB codes for LE, see A.2.1 and A.2.2, can be changed to the pseudo-code given in figure 4.2. With respect to the memory, the value for warped will likely need to be remembered for a short while. With regard to other operations, firstly the calculation of the warped value two subtractions and one division is needed, then two comparisons and a boolean operation is need. Secondly, in the worst case one more subtraction as well as a multiplication and addition is needed. Lastly, one writing operation is needed. However, it should be noted that a relative time is returned that might need more computations before it can be used to generate the spike which depends on the used hardware.

```
Input: signal → a T-length vector containing the signal values in chronological order
       signal window
       spike window
Output: relative fire time → a T-length vector of the relative fires times respective
        to their signal moments and normalized to the sample time
foreach T → t do
    warped = $\frac{signal(t) - signal\ window_{min}}{signal\ window_{max} - signal\ window_{min}}$;
    if warped < 0 OR warped > 1 then
        relative fire time(t) = NaN;
    else
        relative fire time(t) = (warped * (spike window$_{max}$ - spike window$_{min}$)) +
          spike window$_{min}$;
    end
end
```

Figure 4.2: Latency Encoding algorithm pseudo-code

The MATLAB codes for TE, see A.3.1 and A.3.2, can be changed to the pseudo-code given in figure 4.3. In this pseudo-code we can already see that every input stream will have a single boolean as a memorized value. For every signal value in the worst case four comparisons, two boolean operations and two writing operations are needed.

```
Input: signal → a T-length vector containing the signal values in chronological order
       threshold
Output: spikes → a T-length vector containing at which signal values to spike
        (either positively or negatively)
memory = false;
foreach T → t do
    if memory = false AND signal(t) ≥ threshold then
        spikes(t) = 1;
        memory = true;
    else if memory = true AND signal(t) < threshold then
        spikes(t) = -1;
        memory = false;
    else
        spikes(t) = NaN;
    end
end
```

Figure 4.3: Threshold Encoding algorithm pseudo-code

The MATLAB codes for TCE, see A.4.1 and A.4.2, can be changed to the pseudo-code given in figure 4.4. With this algorithm the diff value needs to be remembered for only a short while, but the previous signal value need to be remembered for a longer time. Besides that we would need one subtraction operation, two comparisons, one inversion and one write operation.

```
Input: signal → a T-length vector containing the signal values in chronological order
       threshold
Output: spikes → a T-length vector containing at which signal values to spike
        (either positively or negatively)
foreach T → t do
    diff = signal(t) - signal(t - 1);
    if diff $\geq$ threshold then
        spikes(t) = 1;
    else if diff $\leq$ -threshold then
        spikes(t) = -1;
    else
        spikes(t) = NaN;
    end
end
```

Figure 4.4: Temporal Contrast Encoding algorithm pseudo-code

The MATLAB codes for MWE, see A.5.1 and A.5.2, can be changed to the pseudo-code given in figure 4.5. Besides the short term memory for diff, should the signal values within the window be remembered for a longer time as well as their sum and number. Furthermore, next to the one subtraction, two comparisons, one inversion and one write operation, we will need and extra comparison, a subtraction or addition and a division.

```
Input: signal → a T-length vector containing the signal values in chronological order
       window size = W
       threshold
Output: spikes → a T-length vector containing at which signal values to spike
        (either positively or negatively)
windowSum = 0;
size = 0;
foreach T → t do
    windowSum += signal(t);
    if size = window size then
        windowSum -= signal(t - window size);
    else
        size += 1;
    end
    diff = signal(t) - $\frac{windowSum}{size}$;
    if diff ≥ threshold then
        spikes(t) = 1;
    else if diff ≤ -threshold then
        spikes(t) = -1;
    else
        spikes(t) = NaN;
    end
end
```

Figure 4.5: Moving Window Encoding algorithm pseudo-code

The MATLAB codes for SFE, see A.6.1 and A.6.2, can be changed to the pseudo-code given in figure 4.6. In this algorithm, we don't need to remember the signal values, but we do have two memorized values we need for both short term and long term. Next, there are two subtraction, two comparisons, one inversion and two write operations needed in the worst case.

```
Input: signal → a T-length vector containing the signal values in chronological order
       threshold
Output: spikes → a T-length vector containing at which signal values to spike
        (either positively or negatively)
memory = 0;
foreach T → t do
    diff = signal(t) - memory;
    if diff $\geq$ threshold then
        spikes(t) = 1;
        memory += threshold;
    else if diff $\leq$ -threshold then
        spikes(t) = -1;
        memory -= threshold;
    else
        spikes(t) = NaN;
    end
end
```

Figure 4.6: Step Forward Encoding algorithm pseudo-code

The MATLAB codes for p-TE, see A.7.1, can be changed to the pseudo-code given in figure 4.7. In comparison to TE is it mostly the same, but with an extra FOR-loop and a memorized boolean for each threshold. This means that the operations would amount to four times H comparisons, two times H boolean operations and two times H writing operations. However, the FOR-loop can be done in parallel which would mean that the latency would be not much worse than the latency of TE, but more (hardware) resources would be needed to enable parallel computing.

```
Input: signal → a T-length vector containing the signal values in chronological order
       thresholds → a H-length vector containing the thresholds
Output: spikes → a T*H-sized matrix containing at which signal values to spike
        (either positively or negatively)
memory(1 ... H) = false;
foreach T → t do
    foreach H → h do
        if memory = false AND signal(t) $\geq$ thresholds(h) then
            spikes(t, h) = 1;
            memory = true;
        else if memory = true AND signal(t) $<$ thresholds(h) then
            spikes(t, h) = -1;
            memory = false;
        else
            spikes(t) = NaN;
        end
    end
end
```

Figure 4.7: population Threshold Encoding algorithm pseudo-code

The MATLAB codes for p-SFE, see A.8.1, can be changed to the pseudo-code given in figure 4.8. Just like p-TE, is p-SFE not much different then SFE, while having added long term memory values of each threshold and a FOR-loop. Thus, does it amount to two times H subtraction, two times H comparisons, H inversions and two times H write operations. On the other hand does p-SFE not allow for parallel computing.

```
Input: signal → a T-length vector containing the signal values in chronological order
       thresholds → a H-length vector containing the thresholds
Output: spikes → a T*H-sized matrix containing at which signal values to spike
        (either positively or negatively)
memory = 0;
foreach T → t do
    foreach H → h do /* order is important and can't be done parallel */
        diff = signal(t) - memory;
        if diff $\geq$ thresholds(h) then
            spikes(t) = 1;
            memory += thresholds(h);
        else if diff $\leq$ -thresholds(h) then
            spikes(t) = -1;
            memory -= thresholds(h);
        else
            spikes(t) = NaN;
        end
    end
end
```

Figure 4.8: population Step Forward Encoding algorithm pseudo-code

4.2 Operations

Analyzing the pseudo-codes from the previous section, we will arrive at the operation summation seen in table 4.1 and some more info regarding the memory. All inputs except the signal will need to be remembered for the entire encoding time, thus need a long term memory. Whereas, the short term memory only needs to be remembered for a short while and does not need to leave the cache of the processing unit. With respect to the read operations, they are only needed for the long term memory to get those bytes from memory back into the cache. This distinction between the two different memory types can be important, as the short term memory will likely not need to leave the cache whereas the long term memory will have to be put into the main memory, which is likely to cost more.

Table 4.1: Overview of the operations used by each encoding algorithm for every new signal value supplied.

	add. op.	sub. op.	inv. op.	abs. op.	mult. op.	div. op.	bool. op.	comp. op.	short mem.	long mem.	wr. op.
BSE	2L-2	2L		2L					1	2L	L+1
LE	1	1			1	1	1	2	2	4	2
TE							2	4	1	2	2
TCE		1	1					2	1	2	2
MWE		2	1			1		3	1	W+4	4
SFE		2	1					2	1	2	3
p-TE							2H	4H	1	2H	2H
p-SFE		2H	H					2H	1	H+1	3H

We would prefer to change these operations into an energy and area cost needed to implement, however it has proven to be difficult. In the paper [18] an exceptional attempt is made to establish energy numbers for the multiply- and the add-operations. This paper shows that the energy per operation has such a high overhead, that it is the predominant factor for determining the operation cost and makes the energy cost per operation roughly 70 pJ regardless of the operation type. However, the paper [18] also shows the most energy will be preserved when a dedicated solution is taken, which is thus the likely route taken for the encoding algorithms, so the encoding algorithm would not negate the energy efficiency of the SNN.

Unfortunately, the dedicated solutions is where the comparisons become difficult and/or time-consuming. Either estimations for the cost per operation can be used to transform the values in table 4.1 to an area and energy consumption or all of the encoding algorithms can be implemented into a dedicated solution and tested for area and energy consumption. The later option would consume a lot of extra time, making it outside the scope for this thesis. The first option proofs to be difficult to get an accurate comparison, as the exact implementation can have a big impact. For example, in paper [18] the operation energy depends on the data type used, moreover the ratio between the energy cost for an add operation and a multiply operation is also different depending on the data type. Even more challenging is finding energy cost for all operations as defined in table 4.1. This will most likely take several sources which might not even be comparable. Such a method to make a comparison for the energy cost would also take a lot of time. So, an exact implementation would be preferred as it is more dependable. Unfortunately, for this thesis would be making an exact implementation too much extra work to do for all encoding algorithms and thus we will leave it at the results from table 4.1.

To end the code analysis, one last observation should be made. p-SFE of a singular input can not be done in parallel as the thresholds act on the same memory. Where with p-TE more resources can be used to speed up the encoding process, for p-SFE is it only possible for the different inputs. This means that the number of thresholds limits the time in which the encoding can take place, and consequently the maximum allowed time for encoding will limit the number of thresholds that can be used by p-SFE.

Datasets 5

In this chapter, we will firstly define the datasets we use as well as give some reasons for these datasets. Secondly, some preprocessing of the datasets are also taking into consideration, so these preprocessing techniques are described as well. Thirdly, analyzing the datasets a little, gives an advantage while looking for the best configuration by limiting the search space for each parameter.

Spoken audio is chosen for datasets because of its 'small' samples, while containing a wide variety of information. A simple word can be spoken within a second, if that word is recorded with a sampling rate of a few thousand, then the resulting record is also 'only' a few thousand samples. Moreover, the recorded stream is only 1 dimensional contrary to for example video. Video contains a picture at each sample time, where thus each pixel will be a separate stream that needs encoding. Even though, there is only one stream of data, that single data stream has a big variety of information it contains. The human hearing works in the range of about 50 Hz to 22 kHz, which is quite a big range. However, speech doesn't cover the entire range, it is usually assumed that the majority of the speech information is within the 500 Hz to 4 kHz bandwidth [1].

In this thesis two datasets are used of which the Free Spoken MNIST [23] is the first one. This dataset consists of spoken digits (0 to 9), thus there are ten classes. Each digit is spoken 50 times by each speaker. The used dataset was created from four speakers. However, the newest version of the dataset has increased to six in the duration of the thesis (still all men unfortunately). This amounts to 2000 and 3000 audio samples, respectively. All of these audio samples are recorded with a sampling rate of 8000 Hz. Nyquist theorem shows that the usable audio then ranges up to 4 kHz. The relatively small number of audio samples makes is faster to find and evaluate configurations for encoding algorithms, making quick comparisons possible.

The second datasets, Speech Commands [45], is much larger which gives it several advantages over the simpler Free Spoken MNIST. The first advantage is the increased number of classes; in the Speech Commands dataset 35 words are spoken including the digits. The bigger number of classes increases the difficulty for classification, which should widen the gap between difficult to separate spike trains and the easier ones to separate. Moreover, we can take up to 1500 samples for each word instead of only 200. More samples make sure that a single incorrect classification has less impact on the estimated classification accuracy, so the estimation of the classification accuracy will be more accurate. Another advantage of the Speech Commands dataset is the sampling frequency of 16 kHz, making the usable audio range up to 8 kHz according to the Nyquist theorem. Although most of the speech information is up to 4 kHz, the frequency band of 4 kHz to 8 kHz could increase the intelligibility with a few percent [1]. One last difference is that the Speech Commands dataset already has audio samples with different specific noises added, adding more difficulty which could also widen the gaps between how well encoding algorithms work. These added difficulties might also decrease classification accuracies so far, that the differences between encoding algorithms can become difficult to see, so using both datasets is still preferred.

These two datasets will be split into three subsets to counteract overfitting. The three subsets will be used for training, testing and validating. The test and validate subsets will consist of 20% of the dataset, which is 200 and 5262 samples for the Free Spoken MNIST and Speech Commands dataset, respectively. This leaves 1600 and 41820 samples for the training set for Free Spoken MNIST and Speech Command dataset, respectively. The samples will be divided over the subsets semi-randomly where each subset has the same number of samples from each class. With a separate set for testing and validating, overfitting on only the training samples will not reflect in the results form either sets, as the samples are different. Similarly, will overfitting on the training and test samples combined reflect on the results from the validation samples.

5.1 Preprocessing

There are three preprocessing methods used; audio normalization, using a bank of bandwidth filters, and spike train normalization. The first two preprocessing steps are meant to give an alternative to the raw audio data, such that different dataset attributes/characteristics are covered. The last preprocessing step ensures that the classification of the SVM will be representable as much as possible, as is already described for a bit in section 3.2.3.1. The normalization step and the steps for the bandwidth filtering are shown in the figure 5.1.

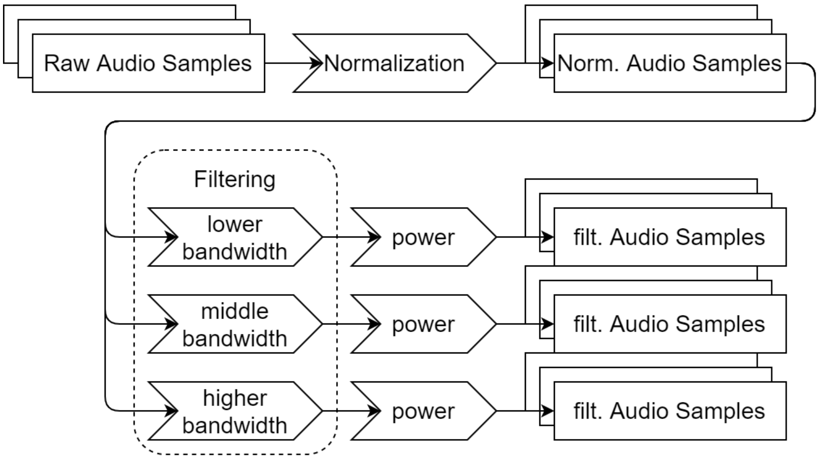

Figure 5.1: The overview of the preprocessing steps to transform the datasets into new ones: the normalized variant and the filtered variants.

5.1.1 Audio normalization

In both datasets are the recordings not completely consistent with one another. Some recordings are more pronounced than others, which is similar to how the real world behaves. There are people who have a very loud voice, while others can hardly be heard. Moreover, depending on how far you are from the microphone, is the recording more clear or barely distinguishable. This can make it difficult for some encoding algorithms to find a good configuration, so normalizing the recordings can give a more even playing ground, but it should be noted that both normalized and non-normalized cases are interesting to compare the encoding algorithms.

There are two audio normalization methods tested; maximum amplitude and signal power. The first method finds the maximal value of the absolute of the audio recordings, the maximum amplitude, then multiplies that audio sample such that the maximum amplitude becomes 1. The second method works a bit similar, but calculates the signal power, the squared Root-Mean-Square, and lets that become 1. Both methods were used to determine the classification potential and both gave similar results, so either methods worked just as good for normalization. However, it can be argued that using the maximum amplitude is more like reality, because audio recording usually have a bounded range that the values can take. Thus, it would make more sense that the setup would be such that the recordings will be just within the range bounds, but with different signal powers. Therefore, the normalization based on maximum amplitude is chosen.

5.1.2 Bank of bandwidth filters

Beside raw audio and normalized audio, we will look at audio filtered through several band filters. The reason is that bank filters are just like how the biology works, as well as it provides opportunities for testing other circumstances for encoding. Much research has been done to find filter-banks that would resemble the ones found in the human ear [15, 48, 36, 34], but we will use the MEL-scale [40]. Besides resembling biology, will it also allow us to test a different type of input. With a cochlear filter bank usually, the output of the filters is put through an extra operation to get the power of the filtered signal, which transforms the oscillating audio signal into a strictly positive signal. Bank filtering the signal will give an entirely different environment for the encoding algorithms be compared in.

The biggest problem with using a bank filter is that the data is multiplied several times. Each filter will create a separate signal stream which needs a separate configuration for each encoding algorithm. This means that the computation needed scales linearly with the number of filters used both for finding configurations and for assessing the metrics. Moreover, for training the SVM scales the size of the input linearly with the number of filters, however this will likely cause an exponential increase in the training time instead of the linear computation time increase of the other parts.

Because of the increasing computation time, is it preferred to use a minimal number of filters for which we will use the MEL-scale. The MEL-scale transforms the frequencies to a scale within which the pitch is linear, for example a frequency duplication does not correspond to a pitch duplication, while a MEL duplication does. This means that we can split any frequency range in other ranges with equal pitch distances, which can be done by transforming the frequencies to the MEL-scale, then linearly spacing the crossing points for the bandwidth filters, and lastly transforming those MEL values back to frequencies. This method lets us create arbitrary number of filters contrary to other research which define specific bandwidths to use for the bank filtering. Thus, for this thesis we use three filters with the cutoff frequencies as defined in table 5.1 and a filter order of ten.

Table 5.1: Frequency for the filters of the filter banks. These Frequencies are given in both Hz and MEL for the Free Spoken MNIST dataset (FS MNIST) and the Speech Command dataset (SC).

	FS MNIST		SC	
	MEL	Hz	MEL	Hz
lowest F	77.755	50	77.755	50
low to mid F_{cutoff}	767.19	682.7	998.51	997.8
mid to high F_{cutoff}	1456.6	1849.3	1919.3	3143.3
highest F	2146.1	4000	2840.0	8000

Getting the power of the signal in time can also be seen as a filtering step. Getting the power in time, we take the power of a small portion of the signal and move this small portion in time. The window taken for this portion is set to 1 ms, which would amount to 8 samples for the Free Spoken MNIST dataset and 16 samples for the Speech Commands dataset. In MATLAB a moving Root-Mean-Square (RMS) is than applied with that many samples. The result from this moving RMS is then squared to get the resulting filtered signals as can be seen in figure 5.2.

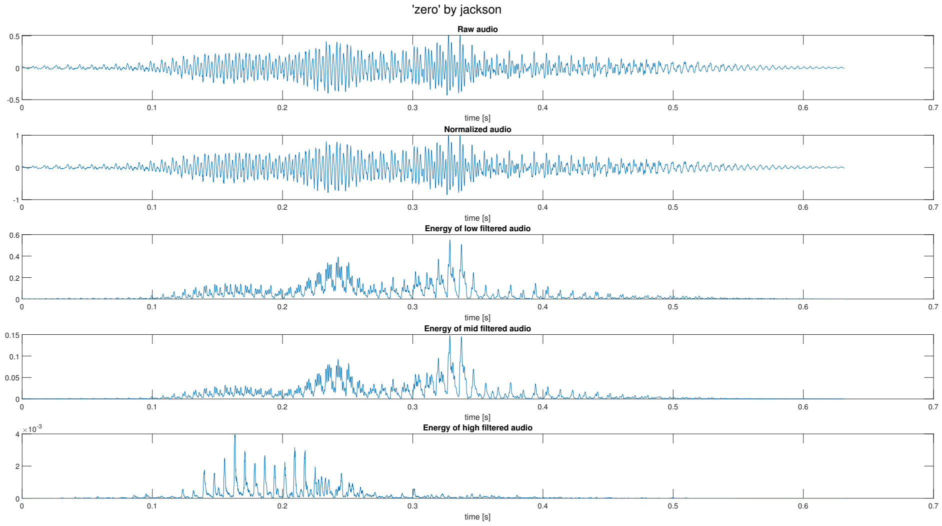

Figure 5.2: Example for the five different signals gotten from preprocessing. The example is taken from a recording of 'Jackson' saying 'zero' from the Free Spoken MNIST dataset.

5.1.3 Spike train normalization

As described in section 3.2.3.1, will an SVM classify the spike trains, but it will need a spike train normalization to simulate the temporal classification potential more accurately. In temporal classification are spikes important relative to one another, so a shift in time for all spikes equally shouldn't matter. This time shift occurs in the dataset, because the starting times of the audio signals are not consistent; some audio samples are cutoff at the start, while others are cutoff at the end. However, for the SVM this amounts to shifting values through the dimensions. To minimize this dimension shifting, we want to find a reference point for all spike trains such that the spike train times align with one another.

We have explored two ways for this normalization; take the first spike as reference and take the mean of the spikes as reference. In both cases were the classification accuracies increased with a similar degree, indicating that spike train normalization was important, but no clear advantage for either method was found. Thus, the simpler method was chosen the first spike as reference. This method is simpler and less computational expensive as only the first and last spike need to be taken into account, while for the mean of the spikes all spike timings need to be checked. Moreover, the mean of the spikes can create the need for 'longer' spike trains. If we have two trains, where the first spike train has many spikes at the front of the audio sample and a single one at the back (thus the mean is much closer to the front), while the second spike train has the opposite (and thus a mean much closer to the back). When those spike time means are aligned with one another, you need to shift the first spike train much later in time, thus a much larger time window is needed. On the other hand, normalizing spike trains according the first spike is done by truncating time samples at the front which can even lead to a smaller time window. Therefore, normalizing spike trains based on the first spike time will be more computational efficient, while have unnoticeable different impacts on the classification accuracy than using the mean spike time.

5.2 Analysis

Analyzing the maximum amplitudes and maximum differences of the audio samples will help us find the optimal configurations easier and faster. Most of the encoding algorithms encode the new signal value or the difference. If we know the maximum of the amplitude and the difference, will it allow us to know the absolute limit the thresholds that the encoding algorithms can use. If we use the distribution, we can even make an educated guess as to what the threshold for the most optimal configuration will be. Thus, the search place for thresholds will be made smaller, making the search for the most optimal configuration easier.

From figures 5.3 and 5.4, we can see that the differences of raw and normalized audio can reach a twice as high as the maximum amplitude, which is not strange as the signals can be within the range of the negative and positive of the amplitude. We can also see that the shapes of the differences' distributions of the two datasets are very similar, indicating that the datasets are similar which can be expected.

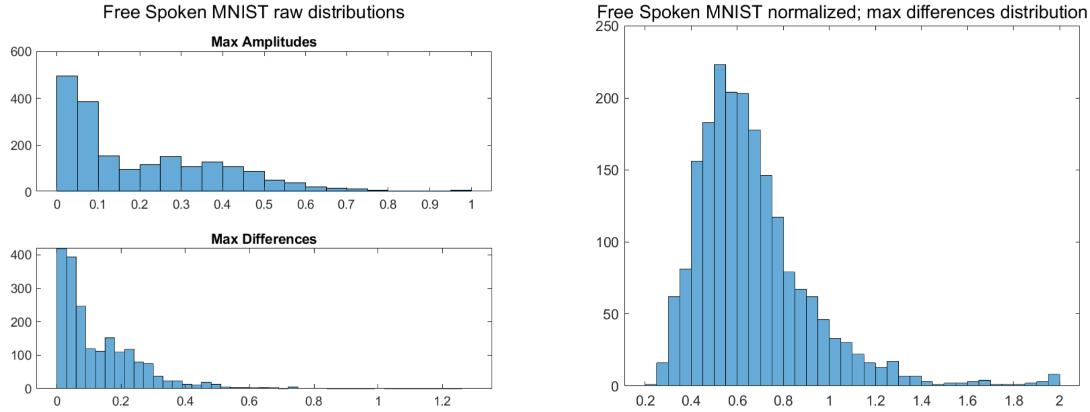

Figure 5.3: The distributions for the Free Spoken MNIST dataset of the maximum amplitudes and maximum differences from the raw and normalized audio samples. The absolute maxima of the raw audio samples from the dataset are 1 and 1.2849 for the amplitude and differences respectively. In the case of the normalized audio samples do all the audio samples have a maximum amplitude of 1 and is the absolute maximum of the differences 1.9915.

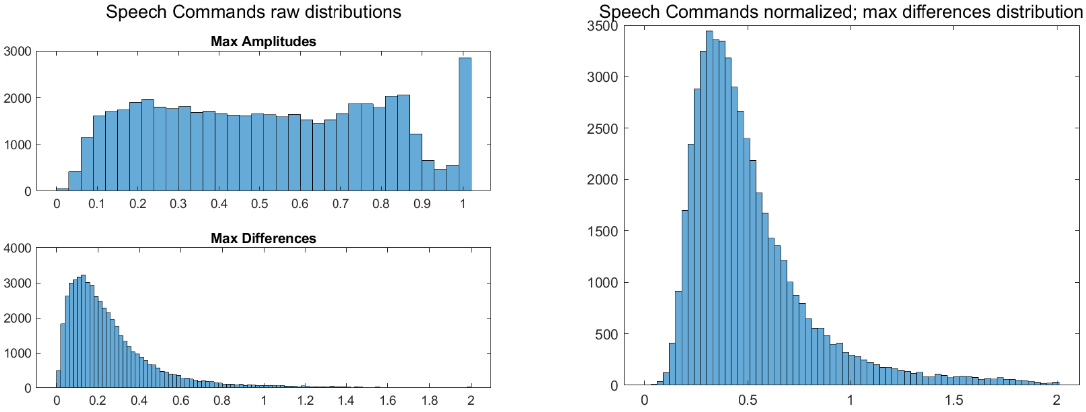

Figure 5.4: The distributions for the Speech Command dataset of the maximum amplitudes and maximum differences from the raw and normalized audio samples. The absolute maxima of the raw audio samples from the dataset are 1 and 2 for the amplitude and differences respectively. In the case of the normalized audio samples do all the audio samples have a maximum amplitude of 1 and is the absolute maximum of the differences 2.

In figures 5.5, 5.6 and 5.7 the distributions for the filtered audio samples are given. From these figures we can see that the shapes for the maximum amplitude distributions for the Speech Commands dataset is similar to those of the Free Spoken MNIST dataset, again indicating that the datasets are similar.

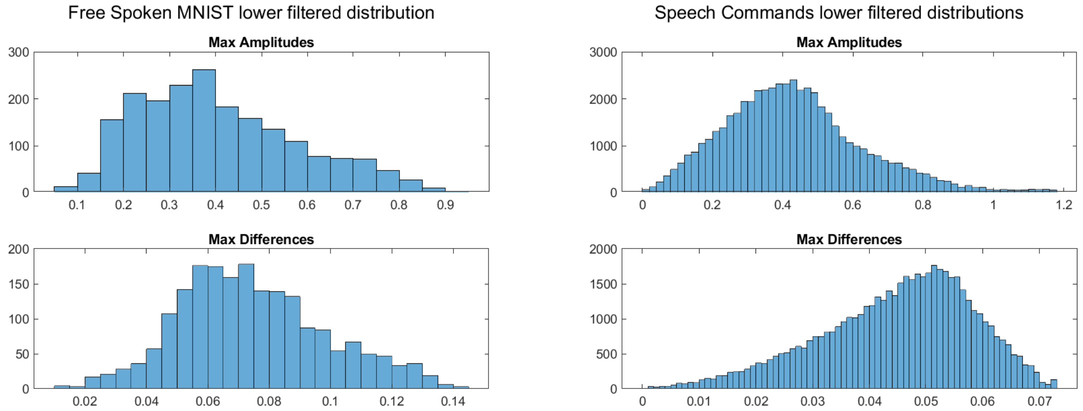

Figure 5.5: The distributions of the maximum amplitudes and maximum differences from the lower bandwidth filtered audio samples for both the Free Spoken MNIST dataset and the Speech Command dataset. The absolute maxima for the amplitude and differences are for the lower bandwidth filtered audio of the Free spoken MNIST dataset 0.9244 and 0.1420 respectively, and of the Speech Commands are they 1.1621 and 0.0726 respectively.

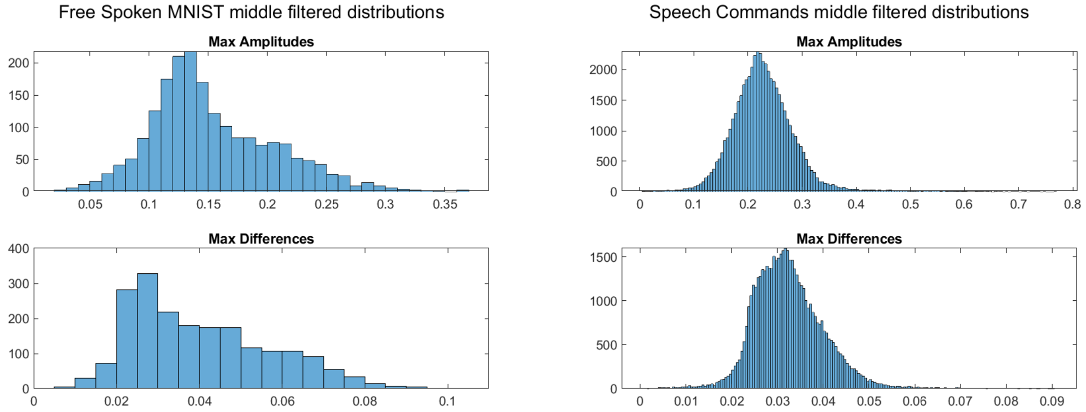

Figure 5.6: The distributions of the maximum amplitudes and maximum differences from the middle bandwidth filtered audio samples for both the Free Spoken MNIST dataset and the Speech Command dataset. The absolute maxima for the amplitude and differences are for the middle bandwidth filtered audio of the Free spoken MNIST dataset 0.3685 and 0.1026 respectively, and of the Speech Commands are they 0.7688 and 0.0905 respectively.

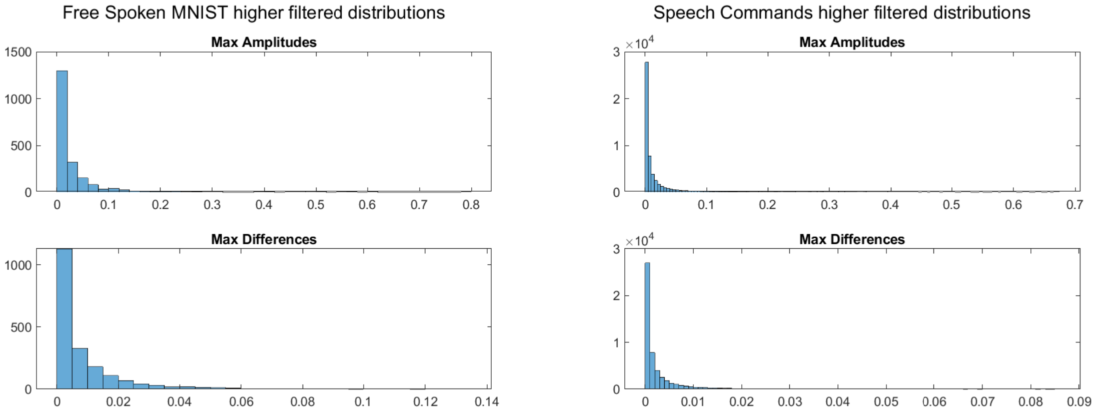

Figure 5.7: The distributions of the maximum amplitudes and maximum differences from the higher bandwidth filtered audio samples for both the Free Spoken MNIST dataset and the Speech Command dataset. The absolute maxima for the amplitude and differences are for the higher bandwidth filtered audio of the Free spoken MNIST dataset 0.7925 and 0.1349 respectively, and of the Speech Commands are they 0.6750 and 0.0854 respectively.

Results and evaluation 6

In this chapter the results will be given as well as discussed. First the signal encoding accuracy is covered. Secondly, the encoding efficiency is extensively explained. Thirdly, the classification potential is estimated by first the rough classification accuracy and then the resistance to noise. Lastly, some limitations and question regarding these specific results are discussed.

6.1 Signal encoding accuracy

In tables 6.1 to 6.10 the results for signal encoding accuracy are shown. The validation set is used to get these results, thus the mean and standard deviation of the results from the validation set are put in the table, as the figures of the distributions would take even more space. For each table one of the p-SFE is chosen to be compared it the other encoding algorithms, and printed in bold. The chosen p-SFE is the configuration with the least thresholds while exceeding the most encoding algorithms in the mean of the signal encoding accuracy. The signal accuracies of the other encoding algorithms that exceed the signal accuracies of the chosen p-SFE are also printed in bold.

The first issue that should be addressed is the difference between SER_{input} and $\text{SER}_{decoded}$. For the majority are these two SERs not too different, with $\text{SER}_{decoded}$ generally being lower than SER_{input}. However, for p-TE and MWE is it the opposite, where $\text{SER}_{decoded}$ is higher than SER_{input}. For p-TE we can see that the difference between the two SERs is less with more thresholds added. Another consistent exception to the majority is TE, which has a significantly lower $\text{SER}_{decoded}$ then SER_{input}. All these differences don't give an advantage to p-SFE and can even give a disadvantage or are insignificant compared to p-SFE. Thus do these differences between SER_{input} and $\text{SER}_{decoded}$ not impact the comparison with p-SFE.

Another short issue that should be addressed is the lack of signal encoding accuracy for BSE in the filtered signal cases. The issue lies with the difficulty of the filter design which indirectly causes the algorithm to not spike for the validation set. As told before, designing a filter is non-trivial and using a genetic algorithm is practically necessary. Even with randomly splitting the datasets in three parts to prevent overfitting, we see that overfitting still happens in the cases of the filtered signals, while overfitting doesn't happen for the audio cases. This shows the big downside of BSE, which is fragile in design.

We can compare the maximum signal encoding accuracy reached by p-SFE with the signal accuracies reached by the other encoding algorithms. In generally we can see that the signal accuracy reached by p-SFE is higher than the signal accuracy of the other encoding algorithms. There is a significant exception, LE manages to achieve a much better signal accuracy for the raw audio and normalized audio, making it technically the better option in these cases. Still, in the cases of the filtered signals we see a drastic drop in the signal accuracy of LE. This decrease is even to such a degree that p-SFE achieves better signal accuracies with only two or three thresholds, similar to the other algorithms.

Another reason to disregard LE for signal encoding accuracy is its weakness to spike time inaccuracy. For other encoding algorithms as the spike only need to occur somewhere within sample time, can those spikes be perfectly decoded as long as they are within half a sample time from the moment they should spike. On the other hand, with LE will a slight deviation of the exact spike moment be reflected in the decoded output. Thus, an error in spike time would get reflected into a similar error of the decoding, which is not modeled in the results of tables 6.1 to 6.10. This kind of error could have a significantly negative impact on the signal encoding accuracy, such that p-SFE could become a comparable or perhaps even better option even for the raw and/or normalized audio cases.

We can also compare the p-SFE with different number of thresholds. Here we see that both $\mathrm{SER}_{decoded}$ and SER_{input} increase in the same way. However, we also see that $\Sigma\ \mathrm{P}_{spike}$ (see equation 3.8) increases. This trend is visible in both the means and the standard deviation. Thus, with more thresholds added we can expect a growth in signal accuracy at the cost of more spikes generated. However, we can see that the growth is much quicker than p-TE, which allows p-SFE to only need two or three thresholds to standout above the other encoding algorithms with respect to the signal accuracy.

Table 6.1: The mean *signal encoding accuracy* of the results for the encoding algorithms from the *raw* audio of the *Free Spoken MNIST* dataset. The parentheses behind the population encoding algorithms show the number of thresholds used, while the parenthesis behind the mean values are the standard deviation.

	SER$_{input}$ [dB]	SER$_{decoded}$ [dB]	Σ P$_{spike}$
BSE	7.335 (± 5.116)	4.474 (± 9.024)	0.06035 (± 0.07052)
LE	**467.4 (± 248.6)**	**467 (± 249.4)**	0.9977 (± 0.006742)
TE	1.89 (± 1.647)	-18.05 (± 7.295)	0.02677 (± 0.02732)
TCE	5.915 (± 3.897)	-0.3927 (± 8.613)	0.1314 (± 0.1416)
MWE	-42.81 (± 15.54)	-0.07945 (± 0.4387)	0.3335 (± 0.2427)
SFE	6.722 (± 4.485)	0.1052 (± 9.14)	0.1311 (± 0.141)
p-TE (2)	-20.03 (± 9.278)	1.482 (± 0.9156)	0.05067 (± 0.03502)
p-TE (3)	-23.73 (± 13.51)	2.329 (± 1.863)	0.06311 (± 0.04859)
p-TE (4)	-14.28 (± 18.16)	4.015 (± 3.884)	0.07053 (± 0.05841)
p-TE (5)	1.184 (± 7.602)	7.177 (± 5.411)	0.1176 (± 0.09425)
p-TE (6)	1.624 (± 8.182)	8.497 (± 6.57)	0.1194 (± 0.0961)
p-TE (7)	3.584 (± 10.46)	9.642 (± 7.837)	0.1207 (± 0.09776)
p-TE (8)	8.076 (± 9.958)	11 (± 9.682)	0.141 (± 0.1208)
p-TE (9)	10.17 (± 10.55)	11.87 (± 10.83)	0.1652 (± 0.1457)
p-TE (10)	11.68 (± 12.02)	13.02 (± 11.97)	0.1712 (± 0.1531)
p-SFE (2)	12.35 (± 7.504)	8.758 (± 9.951)	0.3358 (± 0.299)
p-SFE (3)	**19.48 (± 11.32)**	**17.49 (± 13.07)**	**0.3212 (± 0.2804)**
p-SFE (4)	25.25 (± 11.9)	23.54 (± 13.69)	0.8567 (± 0.5707)
p-SFE (5)	35.43 (± 14.09)	34.84 (± 14.6)	0.6703 (± 0.4461)
p-SFE (6)	44.42 (± 14.54)	44.1 (± 14.83)	0.9978 (± 0.5659)
p-SFE (7)	54.01 (± 15.95)	53.82 (± 16.13)	1.265 (± 0.6124)
p-SFE (8)	63.04 (± 16.9)	62.92 (± 17.05)	1.668 (± 0.6761)
p-SFE (9)	71.97 (± 18.61)	71.88 (± 18.76)	2.101 (± 0.7165)
p-SFE (10)	84.29 (± 20.33)	84.24 (± 20.4)	2.288 (± 0.6739)

Table 6.2: The mean *signal encoding accuracy* of the results for the encoding algorithms from the *normalized* audio of the *Free Spoken MNIST* dataset. The parentheses behind the population encoding algorithms show the number of thresholds used, while the parenthesis behind the mean values are the standard deviation.

	SER_{input} [dB]	$\text{SER}_{decoded}$ [dB]	Σ P_{spike}
BSE	11.09 (± 4.122)	12.17 (± 5.529)	0.157 (± 0.07883)
LE	**396.5 (± 277.2)**	**396.4 (± 277.3)**	0.9998 (± 0.0003924)
TE	3.222 (± 0.9074)	-12.94 (± 2.593)	0.02674 (± 0.01135)
TCE	11.38 (± 3.521)	8.469 (± 4.677)	0.1468 (± 0.04876)
MWE	-49.33 (± 14.59)	0.08398 (± 0.1784)	0.1111 (± 0.0445)
SFE	12.3 (± 3.646)	9.139 (± 5.06)	0.1461 (± 0.04813)
p-TE (2)	-9.54 (± 3.98)	-0.6231 (± 0.649)	0.02989 (± 0.01236)
p-TE (3)	3.387 (± 3.05)	6.796 (± 1.735)	0.08697 (± 0.02926)
p-TE (4)	7.107 (± 2.522)	10.27 (± 1.533)	0.1659 (± 0.0528)
p-TE (5)	14.36 (± 2.104)	13.67 (± 2.606)	0.187 (± 0.05963)
p-TE (6)	17.04 (± 2.154)	18.57 (± 2.118)	0.1931 (± 0.0609)
p-TE (7)	19.24 (± 1.655)	21.02 (± 1.443)	0.2923 (± 0.08545)
p-TE (8)	21.6 (± 2.227)	22.62 (± 2.036)	0.3005 (± 0.08819)
p-TE (9)	23.53 (± 2.157)	24.34 (± 2.084)	0.3493 (± 0.1028)
p-TE (10)	25.37 (± 1.573)	26.2 (± 1.48)	0.4622 (± 0.136)
p-SFE (2)	22.69 (± 4.944)	21.37 (± 5.643)	0.3205 (± 0.09235)
p-SFE (3)	**31.34 (± 4.148)**	**30.77 (± 4.428)**	**0.4269 (± 0.1181)**
p-SFE (4)	41.18 (± 5.935)	40.86 (± 6.163)	0.7099 (± 0.1838)
p-SFE (5)	50.26 (± 8.482)	50.05 (± 8.686)	1.063 (± 0.2492)
p-SFE (6)	58.17 (± 12.05)	58.01 (± 12.23)	1.459 (± 0.3149)
p-SFE (7)	64.81 (± 16.21)	64.68 (± 16.38)	1.886 (± 0.368)
p-SFE (8)	70.27 (± 20.65)	70.16 (± 20.8)	2.329 (± 0.4026)
p-SFE (9)	74.92 (± 25.24)	74.82 (± 25.38)	2.78 (± 0.433)
p-SFE (10)	79 (± 29.87)	78.9 (± 30)	3.246 (± 0.467)

Table 6.3: The mean *signal encoding accuracy* of the results for the encoding algorithms from the *low filtered* audio of the *Free Spoken MNIST* dataset. The parentheses behind the population encoding algorithms show the number of thresholds used, while the parenthesis behind the mean values are the standard deviation.

	SER$_{input}$ [dB]	SER$_{decoded}$ [dB]	Σ P$_{spike}$
BSE	NaN	NaN	0 (± 0)
LE	31.94 (± 9.4)	31.5 (± 9.915)	0.2728 (± 0.135)
TE	7.133 (± 1.913)	-4.922 (± 3.902)	0.01839 (± 0.01197)
TCE	19.02 (± 5.782)	18.8 (± 5.069)	0.1278 (± 0.086)
MWE	-56.59 (± 10.24)	0.2443 (± 0.2204)	0.3053 (± 0.1458)
SFE	15.76 (± 5.848)	16.23 (± 4.419)	0.1258 (± 0.08578)
p-TE (2)	10.05 (± 2.683)	6.365 (± 4.798)	0.04711 (± 0.02049)
p-TE (3)	14.27 (± 5.097)	16.57 (± 3.018)	0.05153 (± 0.02363)
p-TE (4)	17.37 (± 4.679)	19.14 (± 3.258)	0.07674 (± 0.03615)
p-TE (5)	21.28 (± 5.864)	21.92 (± 4.926)	0.08758 (± 0.04303)
p-TE (6)	25.94 (± 3.996)	26.1 (± 4.19)	0.117 (± 0.04856)
p-TE (7)	27.75 (± 4.412)	28.52 (± 3.922)	0.1183 (± 0.04969)
p-TE (8)	29.45 (± 4.439)	30.12 (± 4.056)	0.1406 (± 0.06098)
p-TE (9)	31.51 (± 2.996)	32.06 (± 2.845)	0.141 (± 0.06102)
p-TE (10)	33.16 (± 3.699)	33.53 (± 3.567)	0.1485 (± 0.06626)
p-SFE (2)	27.69 (± 6.84)	27.87 (± 6.396)	0.2073 (± 0.1357)
p-SFE (3)	**39.2 (± 7.256)**	**39.19 (± 7.186)**	**0.3256 (± 0.19)**
p-SFE (4)	50.28 (± 7.905)	50.26 (± 7.906)	0.471 (± 0.2441)
p-SFE (5)	61.32 (± 8.72)	61.3 (± 8.731)	0.6515 (± 0.3039)
p-SFE (6)	72.37 (± 9.536)	72.35 (± 9.545)	0.8657 (± 0.3667)
p-SFE (7)	83.52 (± 10.33)	83.51 (± 10.33)	1.115 (± 0.4337)
p-SFE (8)	94.64 (± 11.52)	94.64 (± 11.52)	1.408 (± 0.5058)
p-SFE (9)	105.4 (± 13.96)	105.4 (± 13.96)	1.75 (± 0.5839)
p-SFE (10)	116.4 (± 16.12)	116.4 (± 16.12)	2.11 (± 0.6601)

Table 6.4: The mean *signal encoding accuracy* of the results for the encoding algorithms from the *mid filtered* audio of the *Free Spoken MNIST* dataset. The parentheses behind the population encoding algorithms show the number of thresholds used, while the parenthesis behind the mean values are the standard deviation.

	SER_{input} [dB]	$\text{SER}_{decoded}$ [dB]	Σ P_{spike}
BSE	NaN	NaN	0 (± 0)
LE	34.9 (± 11.78)	34.4 (± 12.6)	0.3054 (± 0.1145)
TE	6.774 (± 2.015)	-6.143 (± 4.696)	0.01561 (± 0.008679)
TCE	18.85 (± 6.07)	17.93 (± 6.765)	0.1083 (± 0.05091)
MWE	-55.74 (± 10.04)	0.2236 (± 0.1999)	0.2939 (± 0.1179)
SFE	16.58 (± 5.002)	15.59 (± 5.961)	0.1218 (± 0.05565)
p-TE (2)	9.902 (± 3.31)	6.937 (± 4.995)	0.04324 (± 0.01867)
p-TE (3)	13.86 (± 4.317)	15.95 (± 2.791)	0.04649 (± 0.01959)
p-TE (4)	16.79 (± 3.579)	18.26 (± 2.48)	0.04684 (± 0.01962)
p-TE (5)	20.37 (± 2.873)	21.22 (± 2.825)	0.06843 (± 0.02943)
p-TE (6)	24.53 (± 4.201)	24.55 (± 4.376)	0.07611 (± 0.03213)
p-TE (7)	25.99 (± 4.409)	26.79 (± 3.929)	0.0769 (± 0.03246)
p-TE (8)	27.54 (± 4.409)	28.24 (± 4.037)	0.1022 (± 0.04354)
p-TE (9)	23.97 (± 6.152)	25.51 (± 5.064)	0.1022 (± 0.04354)
p-TE (10)	24.5 (± 6.491)	25.94 (± 5.368)	0.1073 (± 0.04515)
p-SFE (2)	26.94 (± 6.566)	26.44 (± 7.124)	0.2146 (± 0.09143)
p-SFE (3)	**36.04 (± 8.627)**	**35.72 (± 8.997)**	**0.3291 (± 0.1289)**
p-SFE (4)	44.43 (± 11.22)	44.22 (± 11.47)	0.4709 (± 0.1718)
p-SFE (5)	52.41 (± 14.13)	52.27 (± 14.3)	0.6425 (± 0.2224)
p-SFE (6)	60.27 (± 17.15)	60.17 (± 17.28)	0.8445 (± 0.2783)
p-SFE (7)	68.04 (± 20.55)	67.96 (± 20.64)	1.084 (± 0.3404)
p-SFE (8)	75.88 (± 24.06)	75.82 (± 24.13)	1.351 (± 0.4077)
p-SFE (9)	81.31 (± 28.98)	81.25 (± 29.06)	1.691 (± 0.4851)
p-SFE (10)	89.59 (± 32.88)	89.55 (± 32.94)	2.018 (± 0.5509)

Table 6.5: The mean *signal encoding accuracy* of the results for the encoding algorithms from the *high filtered* audio of the *Free Spoken MNIST* dataset. The parentheses behind the population encoding algorithms show the number of thresholds used, while the parenthesis behind the mean values are the standard deviation.

	SER$_{input}$ [dB]	SER$_{decoded}$ [dB]	Σ P$_{spike}$
BSE	NaN	NaN	0 (± 0)
LE	14.97 (± 10.33)	10.45 (± 15.26)	0.1185 (± 0.1061)
TE	4.258 (± 2.215)	-15.53 (± 10.48)	0.01127 (± 0.0101)
TCE	9.136 (± 8.666)	6.047 (± 10.72)	0.08282 (± 0.08047)
MWE	-52.37 (± 19.53)	0.1066 (± 1.546)	0.1452 (± 0.1336)
SFE	9.993 (± 7.93)	4.888 (± 12.84)	0.1463 (± 0.1148)
p-TE (2)	-22.84 (± 19.85)	0.2549 (± 4.385)	0.003956 (± 0.005551)
p-TE (3)	-23.03 (± 20.02)	3.313 (± 3.234)	0.004304 (± 0.006451)
p-TE (4)	-22.99 (± 20.09)	3.698 (± 3.758)	0.004335 (± 0.00654)
p-TE (5)	-21.68 (± 21.45)	3.881 (± 4.482)	0.005397 (± 0.008951)
p-TE (6)	-21.45 (± 21.84)	3.974 (± 4.821)	0.005506 (± 0.009357)
p-TE (7)	-21.41 (± 21.9)	4.062 (± 4.968)	0.005517 (± 0.009397)
p-TE (8)	-21.37 (± 22)	4.12 (± 5.223)	0.005522 (± 0.009417)
p-TE (9)	-21.34 (± 22.05)	4.132 (± 5.28)	0.005532 (± 0.00946)
p-TE (10)	-21.32 (± 22.1)	4.148 (± 5.37)	0.005542 (± 0.009504)
p-SFE (2)	**16.08 (± 10.66)**	**12.14 (± 14.84)**	**0.2559 (± 0.1987)**
p-SFE (3)	22.06 (± 12.88)	19.12 (± 16.48)	0.3373 (± 0.2575)
p-SFE (4)	27.67 (± 15.89)	25.63 (± 18.31)	0.4308 (± 0.3109)
p-SFE (5)	33.59 (± 18.28)	31.75 (± 20.73)	0.5736 (± 0.3739)
p-SFE (6)	41.45 (± 19.98)	40.43 (± 21.45)	0.6166 (± 0.3845)
p-SFE (7)	48.07 (± 23.25)	47.06 (± 24.84)	0.8331 (± 0.4584)
p-SFE (8)	54.69 (± 27.05)	53.72 (± 28.58)	1.09 (± 0.5342)
p-SFE (9)	60.63 (± 31.54)	59.65 (± 33.05)	1.417 (± 0.6197)
p-SFE (10)	67.49 (± 36.14)	66.59 (± 37.52)	1.745 (± 0.6879)

Table 6.6: The mean *signal encoding accuracy* of the results for the encoding algorithms from the *raw* audio of the *Speech Commands* dataset. The parentheses behind the population encoding algorithms show the number of thresholds used, while the parenthesis behind the mean values are the standard deviation.

	SER$_{input}$ [dB]	SER$_{decoded}$ [dB]	Σ P$_{spike}$
BSE	7.378 (± 4.488)	6.485 (± 8.389)	0.05487 (± 0.05593)
LE	**364.1 (± 279.5)**	**363.4 (± 280.4)**	0.9973 (± 0.01099)
TE	2.1 (± 1.27)	-18.02 (± 6.502)	0.009264 (± 0.01169)
TCE	11.25 (± 6.238)	7.188 (± 9.327)	0.06981 (± 0.06137)
MWE	-54.25 (± 17.19)	-0.03276 (± 0.9651)	0.1469 (± 0.09621)
SFE	11.48 (± 6.084)	7.357 (± 9.29)	0.06974 (± 0.06131)
p-TE (2)	-34.08 (± 13.05)	0.1377 (± 0.287)	0.01006 (± 0.01278)
p-TE (3)	-4.89 (± 5.68)	4.125 (± 2.931)	0.0247 (± 0.02859)
p-TE (4)	-5.091 (± 6.258)	5.873 (± 3.986)	0.02525 (± 0.0298)
p-TE (5)	2.057 (± 7.812)	7.657 (± 5.984)	0.02811 (± 0.03451)
p-TE (6)	3.659 (± 7.83)	10.06 (± 5.35)	0.05233 (± 0.05601)
p-TE (7)	7.054 (± 9.401)	11.6 (± 6.6)	0.05527 (± 0.06038)
p-TE (8)	13.75 (± 5.447)	15.72 (± 7.737)	0.07962 (± 0.08287)
p-TE (9)	16.22 (± 5.886)	17.27 (± 8.407)	0.08666 (± 0.09156)
p-TE (10)	18.34 (± 7.407)	18.85 (± 9.436)	0.08765 (± 0.09359)
p-SFE (2)	20.36 (± 8.793)	18.4 (± 10.65)	0.1429 (± 0.1146)
p-SFE (3)	**28.26 (± 10.87)**	**27.16 (± 12)**	**0.2478 (± 0.1754)**
p-SFE (4)	36.23 (± 12.61)	35.55 (± 13.43)	0.3557 (± 0.227)
p-SFE (5)	43.25 (± 15.15)	42.73 (± 15.9)	0.5287 (± 0.2989)
p-SFE (6)	51.13 (± 17.31)	50.76 (± 17.93)	0.7002 (± 0.3583)
p-SFE (7)	57.67 (± 20.72)	57.33 (± 21.31)	0.9522 (± 0.4379)
p-SFE (8)	65.97 (± 23.33)	65.71 (± 23.82)	1.198 (± 0.5005)
p-SFE (9)	72.47 (± 27.45)	72.22 (± 27.92)	1.536 (± 0.5779)
p-SFE (10)	78.74 (± 31.86)	78.49 (± 32.31)	1.914 (± 0.6493)

Table 6.7: The mean *signal encoding accuracy* of the results for the encoding algorithms from the *normalized* audio of the *Speech Commands* dataset. The parentheses behind the population encoding algorithms show the number of thresholds used, while the parenthesis behind the mean values are the standard deviation.

	SER_{input} [dB]	$\text{SER}_{decoded}$ [dB]	Σ P_{spike}
BSE	9.795 (± 3.243)	3.949 (± 6.622)	0.1277 (± 0.08479)
LE	**602.1 (± 5.148)**	**602.1 (± 5.148)**	1 (± 0)
TE	2.746 (± 0.9099)	-15.12 (± 3.867)	0.009437 (± 0.007479)
TCE	14.2 (± 6.012)	11.43 (± 7.827)	0.08414 (± 0.04507)
MWE	-57.14 (± 15.68)	0.03028 (± 0.4827)	0.06543 (± 0.04047)
SFE	14.82 (± 5.693)	12.01 (± 7.567)	0.08332 (± 0.04472)
p-TE (2)	-8.753 (± 5.113)	0.301 (± 1.985)	0.1455 (± 0.07577)
p-TE (3)	-10.07 (± 6.524)	2.852 (± 1.883)	0.1512 (± 0.07611)
p-TE (4)	-2.025 (± 5.739)	5.403 (± 2.271)	0.1787 (± 0.08797)
p-TE (5)	10.07 (± 3.567)	12.3 (± 1.91)	0.2061 (± 0.1035)
p-TE (6)	13.81 (± 3.686)	14.61 (± 2.665)	0.2169 (± 0.1076)
p-TE (7)	14.8 (± 4.178)	16.64 (± 3.136)	0.218 (± 0.1079)
p-TE (8)	17.24 (± 4.553)	18.52 (± 3.779)	0.2313 (± 0.1141)
p-TE (9)	19.8 (± 3.827)	20.75 (± 3.284)	0.2706 (± 0.1428)
p-TE (10)	23.03 (± 2.74)	23.67 (± 2.434)	0.3098 (± 0.1731)
p-SFE (2)	24.59 (± 8.383)	23.19 (± 9.672)	0.1853 (± 0.09435)
p-SFE (3)	**33.15 (± 10.42)**	**32.34 (± 11.36)**	**0.2929 (± 0.1408)**
p-SFE (4)	39.65 (± 13.6)	39.04 (± 14.4)	0.4551 (± 0.2024)
p-SFE (5)	46.57 (± 16.36)	46.11 (± 17.01)	0.631 (± 0.263)
p-SFE (6)	53.2 (± 19.33)	52.84 (± 19.88)	0.8435 (± 0.3299)
p-SFE (7)	59.77 (± 22.51)	59.48 (± 22.97)	1.094 (± 0.4013)
p-SFE (8)	66.44 (± 25.89)	66.19 (± 26.29)	1.383 (± 0.4738)
p-SFE (9)	73.31 (± 29.45)	73.1 (± 29.8)	1.709 (± 0.5429)
p-SFE (10)	77.72 (± 33.95)	77.51 (± 34.28)	2.106 (± 0.6125)

Table 6.8: The mean *signal encoding accuracy* of the results for the encoding algorithms from the *low filtered* audio of the *Speech Commands* dataset. The parentheses behind the population encoding algorithms show the number of thresholds used, while the parenthesis behind the mean values are the standard deviation.

	SER_{input} [dB]	$\text{SER}_{decoded}$ [dB]	Σ P_{spike}
BSE	NaN	NaN	0 (± 0)
LE	33.09 (± 11.61)	32.48 (± 12.65)	0.1526 (± 0.1058)
TE	6.573 (± 2.213)	-6.53 (± 5.696)	0.003703 (± 0.002887)
TCE	21.73 (± 10.75)	22.4 (± 8.008)	0.0447 (± 0.03693)
MWE	-58.09 (± 13.58)	0.1839 (± 0.3809)	0.3171 (± 0.228)
SFE	18.73 (± 8.297)	19.27 (± 6.678)	0.05554 (± 0.04394)
p-TE (2)	7.391 (± 5.499)	5.745 (± 4.704)	0.01064 (± 0.005552)
p-TE (3)	10.79 (± 7.53)	14.27 (± 4.032)	0.01149 (± 0.006199)
p-TE (4)	13.91 (± 7.345)	16.48 (± 4.683)	0.01716 (± 0.009323)
p-TE (5)	18.13 (± 5.293)	19.84 (± 3.809)	0.02372 (± 0.01184)
p-TE (6)	22.63 (± 6.846)	23.32 (± 5.591)	0.02583 (± 0.01316)
p-TE (7)	24.01 (± 7.455)	25.27 (± 5.937)	0.02608 (± 0.01345)
p-TE (8)	26.14 (± 6.413)	27.16 (± 5.193)	0.02614 (± 0.01345)
p-TE (9)	27.93 (± 5.994)	28.79 (± 5)	0.03277 (± 0.01681)
p-TE (10)	29.57 (± 6.129)	30.28 (± 5.261)	0.03751 (± 0.01954)
p-SFE (2)	30.26 (± 10.02)	30.64 (± 8.992)	0.08658 (± 0.06504)
p-SFE (3)	**41.46 (± 10.82)**	**41.62 (± 10.3)**	**0.1279 (± 0.08855)**
p-SFE (4)	52.62 (± 10.93)	52.67 (± 10.71)	0.1937 (± 0.1237)
p-SFE (5)	63.53 (± 11.27)	63.55 (± 11.18)	0.2715 (± 0.1625)
p-SFE (6)	74.65 (± 11.49)	74.65 (± 11.47)	0.3733 (± 0.2116)
p-SFE (7)	85.79 (± 11.87)	85.79 (± 11.87)	0.4941 (± 0.2674)
p-SFE (8)	96.73 (± 12.66)	96.73 (± 12.68)	0.6419 (± 0.3336)
p-SFE (9)	108.4 (± 12.92)	108.4 (± 12.93)	0.7749 (± 0.3904)
p-SFE (10)	120 (± 13.69)	120 (± 13.71)	0.9591 (± 0.4677)

Table 6.9: The mean *signal encoding accuracy* of the results for the encoding algorithms from the *mid filtered* audio of the *Speech Commands* dataset. The parentheses behind the population encoding algorithms show the number of thresholds used, while the parenthesis behind the mean values are the standard deviation.

	SER$_{input}$ [dB]	SER$_{decoded}$ [dB]	Σ P$_{spike}$
BSE	NaN	NaN	0 (± 0)
LE	37.15 (± 8.949)	36.83 (± 9.641)	0.1512 (± 0.09565)
TE	7.063 (± 1.796)	-4.618 (± 3.626)	0.003604 (± 0.002457)
TCE	22.56 (± 9.366)	23.09 (± 7.423)	0.03932 (± 0.02675)
MWE	-60.41 (± 11.55)	0.1585 (± 0.3404)	0.2911 (± 0.2088)
SFE	-25.14 (± 8.163)	1.288 (± 1.506)	9.994e-05 (± 0.000417)
p-TE (2)	8.984 (± 3.58)	7.305 (± 3.407)	0.01055 (± 0.005065)
p-TE (3)	12.06 (± 5.171)	15 (± 3.235)	0.01131 (± 0.005484)
p-TE (4)	16.37 (± 5.159)	18.36 (± 3.9)	0.01699 (± 0.008363)
p-TE (5)	19.08 (± 6.402)	20.28 (± 5.251)	0.01899 (± 0.009505)
p-TE (6)	22.84 (± 4.795)	23.52 (± 4.237)	0.01905 (± 0.009505)
p-TE (7)	24.27 (± 4.821)	24.82 (± 4.423)	0.02579 (± 0.01276)
p-TE (8)	24.86 (± 5.102)	25.78 (± 4.538)	0.02596 (± 0.01296)
p-TE (9)	25.98 (± 5.394)	26.8 (± 4.885)	0.03058 (± 0.01548)
p-TE (10)	26.91 (± 5.79)	27.63 (± 5.311)	0.03349 (± 0.01716)
p-SFE (2)	32.27 (± 7.418)	32.37 (± 7.135)	0.08755 (± 0.05498)
p-SFE (3)	**43.24 (± 7.828)**	**43.25 (± 7.742)**	**0.1329 (± 0.07875)**
p-SFE (4)	53.83 (± 8.116)	53.83 (± 8.099)	0.1933 (± 0.1083)
p-SFE (5)	64.09 (± 8.555)	64.08 (± 8.56)	0.2704 (± 0.1443)
p-SFE (6)	74.24 (± 9.292)	74.23 (± 9.3)	0.3653 (± 0.1871)
p-SFE (7)	83.29 (± 11.46)	83.28 (± 11.47)	0.4932 (± 0.2429)
p-SFE (8)	94.67 (± 11.63)	94.66 (± 11.63)	0.6101 (± 0.2919)
p-SFE (9)	103.7 (± 14.86)	103.7 (± 14.86)	0.7769 (± 0.3599)
p-SFE (10)	114.4 (± 16.57)	114.4 (± 16.58)	0.9468 (± 0.4264)

Table 6.10: The mean *signal encoding accuracy* of the results for the encoding algorithms from the *high filtered* audio of the *Speech Commands* dataset. The parentheses behind the population encoding algorithms show the number of thresholds used, while the parenthesis behind the mean values are the standard deviation.

	SER$_{input}$ [dB]	SER$_{decoded}$ [dB]	Σ P$_{spike}$
BSE	NaN	NaN	0 (± 0)
LE	17.29 (± 14)	10.9 (± 21.96)	0.07746 (± 0.08808)
TE	4.06 (± 2.47)	-18.74 (± 14.77)	0.002487 (± 0.00309)
TCE	9.131 (± 12.61)	8.706 (± 10.62)	0.06067 (± 0.07946)
MWE	-68.47 (± 19.7)	0.08067 (± 0.7351)	0.1305 (± 0.1612)
SFE	7.23 (± 11.84)	6.496 (± 10.2)	0.07132 (± 0.0866)
p-TE (2)	-20.9 (± 15)	2.837 (± 2.839)	0.0008471 (± 0.002293)
p-TE (3)	-20.69 (± 15.45)	3.539 (± 3.909)	0.0008619 (± 0.002352)
p-TE (4)	-18.82 (± 16.92)	3.681 (± 4.818)	0.001083 (± 0.003139)
p-TE (5)	-18.58 (± 17.41)	3.779 (± 5.205)	0.001122 (± 0.003341)
p-TE (6)	-18.51 (± 17.57)	3.907 (± 5.545)	0.001129 (± 0.003385)
p-TE (7)	-18.16 (± 17.9)	3.866 (± 5.837)	0.001562 (± 0.004745)
p-TE (8)	-18.07 (± 18.07)	3.89 (± 6.003)	0.001695 (± 0.005342)
p-TE (9)	-18.02 (± 18.17)	3.911 (± 6.107)	0.00176 (± 0.00569)
p-TE (10)	-18.02 (± 18.17)	3.911 (± 6.107)	0.00176 (± 0.00569)
p-SFE (2)	13.27 (± 13.74)	13.27 (± 12.1)	0.1143 (± 0.15)
p-SFE (3)	**19.37 (± 15.19)**	**19.03 (± 14.6)**	**0.1659 (± 0.2144)**
p-SFE (4)	25.51 (± 17.15)	25.41 (± 16.64)	0.1967 (± 0.2568)
p-SFE (5)	32.06 (± 18.77)	31.67 (± 18.99)	0.2532 (± 0.3124)
p-SFE (6)	38.81 (± 20.7)	38.41 (± 21.12)	0.3069 (± 0.3583)
p-SFE (7)	45.26 (± 22.92)	44.58 (± 23.94)	0.4111 (± 0.4342)
p-SFE (8)	53.28 (± 24.81)	52.86 (± 25.51)	0.4513 (± 0.4545)
p-SFE (9)	60.52 (± 27.53)	60.05 (± 28.36)	0.5665 (± 0.5211)
p-SFE (10)	68.13 (± 30.61)	67.69 (± 31.43)	0.6923 (± 0.5855)

6.2 Encoding efficiency

In tables 6.11 to 6.20 the results for encoding efficiency are shown. These results are the mean and standard deviation from the validation set results, just like for the tables of the signal encoding accuracy. In addition, the best efficiency with the least number of thresholds is printed in bold as well as the efficiencies that exceed it, similar to the tables of signal accuracy. Further, the p-SFE with the highest efficiency and a Σ P$_{spike}$ similar to the other encoding algorithms is underlined. The efficiencies that are higher than the underlined efficiency of p-SFE are also underlined.

Although the differences between SER$_{input}$ and SER$_{decoded}$ for efficiency is mostly similar to the differences for signal accuracy, there are some notable changes. Firstly, for BSE is SER$_{decoded}$ higher than SER$_{input}$ in all cases, where with the signal accuracy it was the opposite. Secondly, for p-TE with the signal accuracy the SER$_{decoded}$ is sometimes lower than the SER$_{input}$, but for efficiency SER$_{decoded}$ is always significantly higher than SER$_{input}$. Thirdly, for p-SFE in the cases with the filtered signals we see that SER $_{decoded}$ becomes higher than SER$_{input}$ for fewer thresholds. This would give p-SFE an advantage with respect to other encoding algorithms, thus using SER$_{decoded}$ becomes questionable.

Where BSE didn't spike with the signal accuracy test, it now does sometimes spike for the efficiency. There are only two cases left where BSE doesn't spike: the low and high filtered signals. So BSE is less fragile while designing for efficiency, but is still fragile enough to fail to spike in two cases. However, it should be noted that the efficiency of BSE achieves much better results than the signal accuracy relative to the other algorithms. It shows that BSEs specialty is being efficient, even though it is fairly fragile.

Comparing the maximum efficiency achieved by p-SFE to that of other encoding algorithms we see a better result than for the signal accuracy test. LE has been reduced very significantly with respect to the signal accuracy. The result is that even for raw and normalized audio the p-SFE achieves better efficiencies. This makes p-SFE unchallenged for the maximum efficiency with one exception; p-TE achieves higher SER$_{decoded}$ / Σ P$_{spike}$ for the high filtered signals on the Speech Commands dataset, see table 6.20. However, as stated before, using SER$_{decoded}$ for efficiency is questionable. Moreover, looking at the rate that the maximum efficiency increases for each threshold added to p-SFE shows that with eleven or twelve thresholds p-SFE would be likely to also achieve higher efficiency in that specific case. This means that p-SFE is strongly in the lead with respect to signal accuracy.

Unfortunately, achieving the highest efficiency doesn't show that it is necessarily best for the intended purpose. The efficiency metric should show that with a spike bandwidth limit, because of the AER communication, the highest signal encoding accuracy would be achieved even while dropping/forgetting spikes. However, this only truly holds for roughly similar Σ P$_{spike}$ as some spikes could have a bigger impact on the signal accuracy than others. This could lead to an exponential drop of signal accuracy relative to the number of spikes dropped. Therefore, we should compare encoding algorithms with roughly similar Σ P$_{spike}$. The maximum efficiency for p-SFE is achieved by those with the most number of thresholds in all cases, which also have the highest Σ P$_{spike}$. So to get a better comparison we should look at the lesser performing p-SFE with only a few thresholds.

Comparing p-SFE with fewer thresholds to the other encoding algorithms, we see that p-SFE doesn't get the highest efficiency anymore. Firstly, BSE achieves almost always better efficiency. Secondly and similarly, TE also almost always achieves better efficiency, but in two of the cases TE only has a higher SER$_{input}$ / Σ P$_{spike}$ by more than 5. Thirdly, in half the cases TCE achieves a better efficiency; for the mid and high filtered signals only with about 2, while for the normalized audio of the Speech Commands it reaches a significant 18. Fourthly, LE achieves a higher SER$_{input}$ / Σ P$_{spike}$ only for the high filtered signals. If we look at SER$_{decoded}$ / Σ P$_{spike}$, we see that p-SFE achieves better efficiency relative to the other encoding algorithms, while MWE and p-TE achieve a higher efficiency than p-SFE.

Table 6.11: The mean *encoding efficiency* of the results for the encoding algorithms from the *raw* audio of the *Free Spoken MNIST* dataset. The parentheses behind the population encoding algorithms show the number of thresholds used, while the parenthesis behind the mean values are the standard deviation.

	SER$_{input}$/Σ P$_{spike}$ [dB]	SER$_{decoded}$/Σ P$_{spike}$ [dB]	Σ P$_{spike}$
BSE	45.59 (± 11.63)	47.01 (± 5.721)	0.002281 (± 0.004252)
LE	28.49 (± 12.53)	15.9 (± 6.317)	0.1082 (± 0.1072)
TE	43.61 (± 17.62)	12.49 (± 10.41)	0.03012 (± 0.02943)
TCE	31.99 (± 12.74)	25.78 (± 9.983)	0.09434 (± 0.1098)
MWE	-24.04 (± 24.85)	24.85 (± 17.24)	0.1043 (± 0.1261)
SFE	30.88 (± 11.71)	24.55 (± 10.78)	0.1109 (± 0.1249)
p-TE (2)	-9.306 (± 8.603)	41.4 (± 18.69)	0.03091 (± 0.02979)
p-SFE (2)	37.43 (± 7.133)	33.14 (± 3.879)	0.0994 (± 0.1147)
p-SFE (3)	37.97 (± 6.007)	34.53 (± 4.759)	0.2209 (± 0.2113)
p-SFE (4)	38.8 (± 6.438)	37.34 (± 7.174)	0.3803 (± 0.3077)
p-SFE (5)	42.17 (± 8.581)	41.5 (± 9.151)	0.5719 (± 0.3985)
p-SFE (6)	**47.41 (± 10.74)**	**47.07 (± 11.04)**	**0.8295 (± 0.5006)**
p-SFE (7)	38.53 (± 26.15)	37.25 (± 27.88)	1.88 (± 0.8104)
p-SFE (8)	42.35 (± 30.54)	41.09 (± 32.22)	2.259 (± 0.8889)
p-SFE (9)	69.85 (± 16.53)	69.79 (± 16.6)	1.739 (± 0.6281)
p-SFE (10)	78.59 (± 18.75)	78.55 (± 18.81)	2.093 (± 0.6836)

Table 6.12: The mean *encoding efficiency* of the results for the encoding algorithms from the *normalized* audio of the *Free Spoken MNIST* dataset. The parentheses behind the population encoding algorithms show the number of thresholds used, while the parenthesis behind the mean values are the standard deviation.

	$\text{SER}_{input}/\Sigma\ \text{P}_{spike}$ [dB]	$\text{SER}_{decoded}/\Sigma\ \text{P}_{spike}$ [dB]	$\Sigma\ \text{P}_{spike}$
BSE	52.15 (± 9.456)	52.35 (± 4.155)	0.001404 (± 0.001843)
LE	38.23 (± 8.316)	21.3 (± 3.968)	0.02209 (± 0.01686)
TE	43.91 (± 5.776)	22.36 (± 4.877)	0.009231 (± 0.005413)
TCE	56 (± 6.095)	35.59 (± 4.268)	0.002113 (± 0.002373)
MWE	19.16 (± 23.12)	62.02 (± 7.967)	0.0003124 (± 0.0008252)
SFE	63.16 (± 3.525)	36.21 (± 4.488)	0.0007745 (± 0.0004323)
p-TE (2)	20 (± 5.286)	40.93 (± 5.64)	0.01001 (± 0.0057)
p-SFE (2)	63.16 (± 3.525)	36.21 (± 4.488)	0.0007745 (± 0.0004323)
p-SFE (3)	39.12 (± 3.848)	38.55 (± 4.091)	0.4269 (± 0.1181)
p-SFE (4)	44.5 (± 5.346)	44.18 (± 5.569)	0.7099 (± 0.1838)
p-SFE (5)	50.01 (± 7.878)	49.8 (± 8.083)	1.063 (± 0.2492)
p-SFE (6)	55.12 (± 11.49)	54.97 (± 11.67)	1.459 (± 0.3149)
p-SFE (7)	59.49 (± 15.72)	59.36 (± 15.89)	1.886 (± 0.368)
p-SFE (8)	63.08 (± 20.26)	62.97 (± 20.41)	2.329 (± 0.4026)
p-SFE (9)	**66.17 (± 24.94)**	**66.06 (± 25.08)**	**2.78 (± 0.433)**
p-SFE (10)	68.89 (± 29.63)	68.79 (± 29.76)	3.246 (± 0.467)

Table 6.13: The mean *encoding efficiency* of the results for the encoding algorithms from the *low filtered* audio of the *Free Spoken MNIST* dataset. The parentheses behind the population encoding algorithms show the number of thresholds used, while the parenthesis behind the mean values are the standard deviation.

	SER$_{input}$/Σ P$_{spike}$ [dB]	SER$_{decoded}$/Σ P$_{spike}$ [dB]	Σ P$_{spike}$
BSE	50.68 (± 8.608)	53.97 (± 7.59)	0.002896 (± 0.003772)
LE	38.14 (± 6.939)	30.59 (± 3.735)	0.04126 (± 0.04057)
TE	45.3 (± 6.371)	33.02 (± 6.264)	0.01517 (± 0.01119)
TCE	49.62 (± 7.295)	63.54 (± 7.197)	0.001428 (± 0.002684)
MWE	3.764 (± 28.07)	50.93 (± 11.25)	0.002191 (± 0.005416)
SFE	48.24 (± 6.596)	68.09 (± 7.55)	0.0007473 (± 0.001649)
p-TE (2)	37.31 (± 3.562)	42.17 (± 8.076)	0.01552 (± 0.01123)
p-TE (3)	36.03 (± 4.066)	46.64 (± 5.659)	0.01584 (± 0.01169)
p-TE (4)	36.03 (± 4.065)	46.67 (± 5.645)	0.01584 (± 0.01169)
p-SFE (2)	48.44 (± 6.74)	68.05 (± 7.8)	0.000802 (± 0.001793)
p-SFE (3)	46.75 (± 7.134)	64.23 (± 10.17)	0.002053 (± 0.003895)
p-SFE (4)	58.31 (± 6.192)	58.29 (± 6.209)	0.4539 (± 0.2359)
p-SFE (5)	66.21 (± 7.514)	66.2 (± 7.531)	0.6425 (± 0.3003)
p-SFE (6)	74.27 (± 9.453)	74.25 (± 9.466)	0.8761 (± 0.3714)
p-SFE (7)	**83.08 (± 10.48)**	**83.08 (± 10.49)**	**1.126 (± 0.4358)**
p-SFE (8)	91.64 (± 12.64)	91.63 (± 12.64)	1.43 (± 0.5133)
p-SFE (9)	100.6 (± 14.94)	100.6 (± 14.95)	1.764 (± 0.5889)
p-SFE (10)	109.9 (± 17.31)	109.9 (± 17.31)	2.121 (± 0.6626)

Table 6.14: The mean *encoding efficiency* of the results for the encoding algorithms from the *mid filtered* audio of the *Free Spoken MNIST* dataset. The parentheses behind the population encoding algorithms show the number of thresholds used, while the parenthesis behind the mean values are the standard deviation.

	$\text{SER}_{input}/\Sigma\ \text{P}_{spike}$ [dB]	$\text{SER}_{decoded}/\Sigma\ \text{P}_{spike}$ [dB]	$\Sigma\ \text{P}_{spike}$
BSE	54.23 (± 6.867)	57.21 (± 3.817)	0.00192 (± 0.001663)
LE	40.09 (± 6.495)	39.21 (± 7.067)	0.1619 (± 0.06955)
TE	49.81 (± 5.617)	35.38 (± 5.43)	0.006549 (± 0.004485)
TCE	47.72 (± 7.238)	63.96 (± 6.92)	0.001156 (± 0.001708)
MWE	-4.145 (± 29.62)	49.19 (± 11.22)	0.003476 (± 0.005539)
SFE	46.82 (± 6.436)	68.2 (± 6.865)	0.0006462 (± 0.001138)
p-TE (2)	37.9 (± 5.061)	49.57 (± 8.437)	0.006654 (± 0.004446)
p-TE (3)	36.69 (± 5.536)	51.21 (± 7.46)	0.006654 (± 0.004446)
p-SFE (2)	46.82 (± 6.331)	68.49 (± 6.661)	0.0006038 (± 0.001082)
p-SFE (3)	46.46 (± 7.843)	66.53 (± 8.6)	0.001014 (± 0.001712)
p-SFE (4)	39.51 (± 6.957)	50.61 (± 9.466)	0.008239 (± 0.007703)
p-SFE (5)	58.58 (± 13.25)	58.47 (± 13.38)	0.6087 (± 0.2122)
p-SFE (6)	64.15 (± 16.32)	64.07 (± 16.42)	0.813 (± 0.2687)
p-SFE (7)	**69.97 (± 19.62)**	**69.91 (± 19.7)**	**1.049 (± 0.3318)**
p-SFE (8)	73.74 (± 24.47)	73.68 (± 24.55)	1.351 (± 0.4077)
p-SFE (9)	80.04 (± 28.04)	79.99 (± 28.1)	1.653 (± 0.4746)
p-SFE (10)	86.89 (± 31.61)	86.85 (± 31.67)	1.979 (± 0.5445)

Table 6.15: The mean *encoding efficiency* of the results for the encoding algorithms from the *high filtered* audio of the *Free Spoken MNIST* dataset. The parentheses behind the population encoding algorithms show the number of thresholds used, while the parenthesis behind the mean values are the standard deviation.

	$\text{SER}_{input}/\Sigma\ \text{P}_{spike}$ [dB]	$\text{SER}_{decoded}/\Sigma\ \text{P}_{spike}$ [dB]	$\Sigma\ \text{P}_{spike}$
BSE	56.26 (± 8.813)	57.17 (± 6.865)	0.0028 (± 0.007455)
LE	45.72 (± 8.589)	39.36 (± 8.831)	0.009941 (± 0.02141)
TE	49.85 (± 9.476)	31.19 (± 12.89)	0.006069 (± 0.007683)
TCE	42.59 (± 8.385)	53.48 (± 11.7)	0.007783 (± 0.01767)
MWE	-13.68 (± 26.79)	40.06 (± 15.26)	0.01687 (± 0.02976)
SFE	38.44 (± 6.852)	54.91 (± 15.22)	0.009866 (± 0.02126)
p-TE (2)	34.9 (± 11.75)	58 (± 15.66)	0.003956 (± 0.005551)
p-TE (3)	34.01 (± 11.79)	60.93 (± 11.24)	0.004228 (± 0.006257)
p-TE (4)	33.96 (± 11.76)	61.17 (± 10.79)	0.004242 (± 0.006297)
p-TE (5)	33.99 (± 11.81)	61.23 (± 10.71)	0.004247 (± 0.006314)
p-TE (6)	33.99 (± 11.81)	61.24 (± 10.71)	0.004247 (± 0.006314)
p-TE (7)	34 (± 11.82)	61.24 (± 10.71)	0.004258 (± 0.00635)
p-TE (8)	34.15 (± 11.94)	61.3 (± 10.66)	0.004294 (± 0.006464)
p-TE (9)	34.16 (± 11.95)	61.3 (± 10.66)	0.004294 (± 0.006464)
p-TE (10)	35.16 (± 12.11)	61.34 (± 10.55)	0.004849 (± 0.008105)
p-SFE (2)	39.59 (± 6.198)	55.89 (± 13.27)	0.01021 (± 0.02285)
p-SFE (3)	40.24 (± 6.202)	56.32 (± 12.67)	0.01001 (± 0.02117)
p-SFE (4)	40.03 (± 6.507)	56.84 (± 12.82)	0.008623 (± 0.01884)
p-SFE (5)	40.36 (± 6.403)	56.44 (± 12.62)	0.009805 (± 0.02023)
p-SFE (6)	40.87 (± 6.469)	55.92 (± 12.56)	0.01121 (± 0.02239)
p-SFE (7)	40.69 (± 6.412)	56.12 (± 12.56)	0.0108 (± 0.02182)
p-SFE (8)	57.41 (± 26.88)	56.65 (± 28.23)	1.006 (± 0.5058)
p-SFE (9)	61.22 (± 31.32)	60.45 (± 32.62)	1.322 (± 0.59)
p-SFE (10)	**65.29 (± 36.02)**	**64.52 (± 37.28)**	**1.678 (± 0.6705)**

Table 6.16: The mean *encoding efficiency* of the results for the encoding algorithms from the *raw* audio of the *Speech Commands* dataset. The parentheses behind the population encoding algorithms show the number of thresholds used, while the parenthesis behind the mean values are the standard deviation.

	$\text{SER}_{input}/\Sigma\ \text{P}_{spike}$ [dB]	$\text{SER}_{decoded}/\Sigma\ \text{P}_{spike}$ [dB]	$\Sigma\ \text{P}_{spike}$
BSE	69.45 (± 11.58)	60.31 (± 6.674)	3.615e-05 (± 0.000332)
LE	39.06 (± 11.24)	27.29 (± 6.007)	0.0282 (± 0.03503)
TE	52.65 (± 9.559)	30.75 (± 6.361)	0.003389 (± 0.006187)
TCE	47.06 (± 8.927)	41.26 (± 6.476)	0.01719 (± 0.02431)
MWE	6.591 (± 30.85)	60.5 (± 14.67)	0.0007642 (± 0.005005)
SFE	48.11 (± 9.036)	41.7 (± 6.226)	0.01358 (± 0.02078)
p-TE (2)	35.56 (± 9.514)	60.11 (± 18.38)	0.004247 (± 0.008235)
p-TE (3)	35.53 (± 9.531)	60.54 (± 17.92)	0.004247 (± 0.008235)
p-SFE (2)	13.75 (± 11.64)	4.041 (± 17.01)	0.5236 (± 0.2473)
p-SFE (3)	21.8 (± 16.05)	17.28 (± 19.49)	0.5773 (± 0.3076)
p-SFE (4)	35.19 (± 18.34)	33.5 (± 20.17)	0.559 (± 0.3185)
p-SFE (5)	50.13 (± 16.75)	49.61 (± 17.55)	0.5287 (± 0.2989)
p-SFE (6)	53.29 (± 20.02)	52.84 (± 20.76)	0.7447 (± 0.3765)
p-SFE (7)	60.97 (± 20.99)	60.69 (± 21.52)	0.9071 (± 0.4207)
p-SFE (8)	67.06 (± 23.36)	66.84 (± 23.8)	1.152 (± 0.4852)
p-SFE (9)	**71.75 (± 27.29)**	**71.54 (± 27.71)**	**1.481 (± 0.5629)**
p-SFE (10)	78.87 (± 30.01)	78.7 (± 30.37)	1.803 (± 0.6215)

Table 6.17: The mean *encoding efficiency* of the results for the encoding algorithms from the *normalized* audio of the *Speech Commands* dataset. The parentheses behind the population encoding algorithms show the number of thresholds used, while the parenthesis behind the mean values are the standard deviation.

	$\text{SER}_{input}/\Sigma\ \text{P}_{spike}$ [dB]	$\text{SER}_{decoded}/\Sigma\ \text{P}_{spike}$ [dB]	$\Sigma\ \text{P}_{spike}$
BSE	49.07 (± 10.23)	50.77 (± 5.906)	0.00351 (± 0.006029)
LE	47 (± 8.827)	30.26 (± 5.445)	0.008786 (± 0.01119)
TE	56.35 (± 7.731)	32.51 (± 6.17)	0.002429 (± 0.003654)
TCE	72.8 (± 6.619)	46.36 (± 6.396)	0.0004408 (± 0.001964)
MWE	25.71 (± 26.99)	70.47 (± 12.15)	0.0002195 (± 0.001725)
SFE	54.55 (± 6.288)	44.12 (± 5.995)	0.003139 (± 0.00395)
p-TE (2)	44.23 (± 8.126)	52.12 (± 9.989)	0.002968 (± 0.003417)
p-TE (3)	44.01 (± 8.172)	52.2 (± 9.767)	0.003031 (± 0.003429)
p-SFE (2)	54.54 (± 6.267)	44.15 (± 5.947)	0.00314 (± 0.003969)
p-SFE (3)	44.63 (± 5.258)	40.95 (± 5.532)	0.01839 (± 0.01354)
p-SFE (4)	50.96 (± 12.73)	50.52 (± 13.37)	0.3921 (± 0.1781)
p-SFE (5)	55.43 (± 15.16)	55.12 (± 15.68)	0.5597 (± 0.2364)
p-SFE (6)	58.78 (± 18.81)	58.5 (± 19.28)	0.7888 (± 0.3108)
p-SFE (7)	63.4 (± 21.85)	63.18 (± 22.25)	1.035 (± 0.3826)
p-SFE (8)	68.34 (± 25.11)	68.15 (± 25.46)	1.321 (± 0.4561)
p-SFE (9)	**73.64 (± 28.54)**	**73.48 (± 28.84)**	**1.645 (± 0.5273)**
p-SFE (10)	77.12 (± 33.1)	76.96 (± 33.39)	2.035 (± 0.5984)

Table 6.18: The mean *encoding efficiency* of the results for the encoding algorithms from the *low filtered* audio of the *Speech Commands* dataset. The parentheses behind the population encoding algorithms show the number of thresholds used, while the parenthesis behind the mean values are the standard deviation.

	$\text{SER}_{input}/\Sigma\ \text{P}_{spike}$ [dB]	$\text{SER}_{decoded}/\Sigma\ \text{P}_{spike}$ [dB]	$\Sigma\ \text{P}_{spike}$
BSE	NaN	NaN	0 (± 0)
LE	47.01 (± 8.734)	37.98 (± 5.846)	0.01587 (± 0.03388)
TE	60.14 (± 6.902)	46.65 (± 6.154)	0.002248 (± 0.002292)
TCE	53.34 (± 8.762)	55.12 (± 7.186)	0.02401 (± 0.02241)
MWE	11.41 (± 31.61)	63.2 (± 11.79)	0.0004748 (± 0.002307)
SFE	57.01 (± 8.561)	81.01 (± 8.286)	0.0001933 (± 0.0006746)
p-TE (2)	43.44 (± 6.574)	60.16 (± 12.18)	0.00231 (± 0.002292)
p-TE (3)	42.72 (± 6.818)	61.88 (± 10.97)	0.002321 (± 0.002335)
p-TE (4)	42.72 (± 6.816)	61.88 (± 10.97)	0.002321 (± 0.002335)
p-SFE (2)	57.14 (± 8.361)	82 (± 7.255)	0.0001417 (± 0.0005609)
p-SFE (3)	55.72 (± 8.929)	77.61 (± 10.35)	0.0003926 (± 0.001115)
p-SFE (4)	49.85 (± 8.721)	65.98 (± 12.19)	0.001766 (± 0.003005)
p-SFE (5)	76.33 (± 7.93)	76.35 (± 7.883)	0.2715 (± 0.1625)
p-SFE (6)	**84.55 (± 8.571)**	**84.55 (± 8.563)**	**0.3668 (± 0.2082)**
p-SFE (7)	93.12 (± 9.451)	93.11 (± 9.46)	0.4941 (± 0.2674)
p-SFE (8)	102.1 (± 9.959)	102.1 (± 9.975)	0.6109 (± 0.319)
p-SFE (9)	111.6 (± 10.84)	111.6 (± 10.86)	0.7749 (± 0.3904)
p-SFE (10)	121.4 (± 11.98)	121.4 (± 12)	0.9591 (± 0.4677)

Table 6.19: The mean *encoding efficiency* of the results for the encoding algorithms from the *mid filtered* audio of the *Speech Commands* dataset. The parentheses behind the population encoding algorithms show the number of thresholds used, while the parenthesis behind the mean values are the standard deviation.

	SER$_{input}$/Σ P$_{spike}$ [dB]	SER$_{decoded}$/Σ P$_{spike}$ [dB]	Σ P$_{spike}$
BSE	67.4 (± 9.425)	69 (± 6.092)	0.0006416 (± 0.001962)
LE	49.42 (± 7.534)	48.87 (± 8.163)	0.1119 (± 0.07611)
TE	60.27 (± 6.317)	47.77 (± 6.104)	0.002283 (± 0.001877)
TCE	60.59 (± 8.043)	80.1 (± 6.23)	0.0001883 (± 0.0005403)
MWE	37.51 (± 31.6)	73.39 (± 12.23)	0.0001644 (± 0.001603)
SFE	58.12 (± 7.639)	84.55 (± 3.98)	9.994e-05 (± 0.000417)
p-TE (2)	40.89 (± 4.113)	57 (± 7.461)	0.002345 (± 0.001877)
p-TE (3)	40.65 (± 4.211)	58.19 (± 6.762)	0.002387 (± 0.002049)
p-TE (4)	40.64 (± 4.206)	58.2 (± 6.753)	0.002387 (± 0.002049)
p-SFE (2)	58.12 (± 7.639)	84.55 (± 3.982)	9.995e-05 (± 0.000417)
p-SFE (3)	58.23 (± 7.639)	84.49 (± 4.117)	0.0001045 (± 0.0004464)
p-SFE (4)	52.21 (± 8.69)	71.4 (± 10.04)	0.0007344 (± 0.001414)
p-SFE (5)	76.8 (± 6.189)	76.8 (± 6.192)	0.2589 (± 0.1389)
p-SFE (6)	84.02 (± 7.637)	84.01 (± 7.648)	0.3653 (± 0.1871)
p-SFE (7)	**92.41 (± 7.981)**	**92.4 (± 7.989)**	**0.465 (± 0.2306)**
p-SFE (8)	99.87 (± 10.51)	99.86 (± 10.52)	0.6101 (± 0.2919)
p-SFE (9)	108.4 (± 12.15)	108.4 (± 12.15)	0.7605 (± 0.3531)
p-SFE (10)	115.7 (± 15.91)	115.7 (± 15.92)	0.9468 (± 0.4264)

Table 6.20: The mean *encoding efficiency* of the results for the encoding algorithms from the *high filtered* audio of the *Speech Commands* dataset. The parentheses behind the population encoding algorithms show the number of thresholds used, while the parenthesis behind the mean values are the standard deviation.

	$\text{SER}_{input}/\Sigma\ \text{P}_{spike}$ [dB]	$\text{SER}_{decoded}/\Sigma\ \text{P}_{spike}$ [dB]	$\Sigma\ \text{P}_{spike}$
BSE	NaN	NaN	0 (± 0)
LE	61.18 (± 11.14)	24.2 (± 17.43)	0.000513 (± 0.001456)
TE	60.49 (± 10.41)	40.34 (± 16.56)	0.001653 (± 0.00266)
TCE	50.95 (± 9.044)	63.32 (± 13.28)	0.006709 (± 0.02099)
MWE	-28.57 (± 34.32)	43.56 (± 17.8)	0.01825 (± 0.06079)
SFE	46.93 (± 7.212)	64.03 (± 15.16)	0.005725 (± 0.01896)
p-TE (2)	27.83 (± 22.27)	**75.16 (± 15.39)**	0.0008201 (± 0.002046)
p-TE (3)	25.98 (± 21.39)	**76.76 (± 12)**	0.0008597 (± 0.002148)
p-TE (4)	25.97 (± 21.39)	**76.85 (± 11.86)**	0.000864 (± 0.002169)
p-TE (5)	25.97 (± 21.39)	**76.86 (± 11.85)**	0.000864 (± 0.002169)
p-TE (6)	25.97 (± 21.39)	**76.86 (± 11.85)**	0.0008659 (± 0.002184)
p-SFE (2)	48.06 (± 7.32)	65.33 (± 13.73)	0.005672 (± 0.02046)
p-SFE (3)	48.56 (± 7.892)	65.35 (± 13.19)	0.006287 (± 0.0216)
p-SFE (4)	48.87 (± 8.297)	65.31 (± 13.11)	0.006438 (± 0.02116)
p-SFE (5)	48.79 (± 8.379)	65.58 (± 13.2)	0.006071 (± 0.01988)
p-SFE (6)	48.67 (± 8.105)	66.64 (± 13.46)	0.004333 (± 0.01552)
p-SFE (7)	48.79 (± 8.38)	65.58 (± 13.2)	0.006071 (± 0.01988)
p-SFE (8)	**65.57 (± 25.46)**	**65.25 (± 26.4)**	**0.4247 (± 0.4349)**
p-SFE (9)	69.51 (± 29.13)	69.08 (± 30.07)	0.5535 (± 0.5122)
p-SFE (10)	74.46 (± 32.62)	74.03 (± 33.5)	0.6878 (± 0.5824)

6.3 Classification potential

6.3.1 Classification accuracy

Figure 6.1 shows the classification accuracies for the encoding algorithms on the different input types of the FS MINST dataset. For classification potential we also added an extra input type, the combination of the spike trains from the three filtered signals. This means that the SVM is trained with the spike trains of all three filtered power-analyzed signals. In appendix C the classification accuracies for the different number of thresholds are given, while in figure 6.1 only the best performing configuration from appendix C are used.

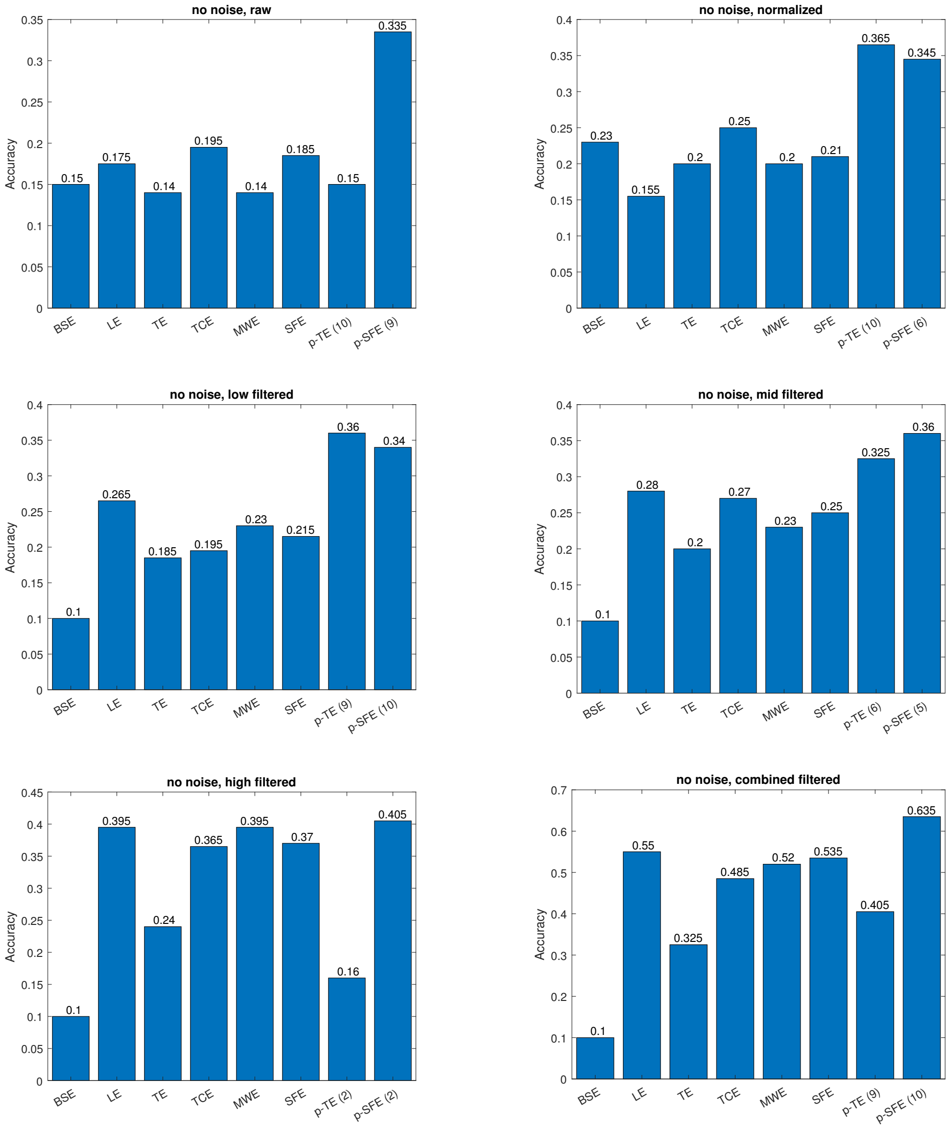

Figure 6.1: The classification accuracy achieved with an SVM on the validation set from the FS MNIST dataset.

From figure 6.1 we can see that p-SFE achieves the highest classification accuracy in almost all cases. The only two cases where p-SFE doesn't achieve the highest classification accuracy is for the normalized audio and the lower filtered signal. Even in these cases the classification accuracy of p-SFE is only 0.02 less than p-TE. Which means that p-SFE either achieves the highest classification accuracy or comparable to the highest classification accuracy, making it the best option in this regard.

6.3.2 Noise resistance

In figure 6.2 an example is shown for how the noise resistance is determined. In this figure the classification accuracy for the signal with noise is given as well as the classification accuracy for only the noise. When the classification accuracy of only the noise exceeds the classification accuracy of the signal with noise, the signal does not attribute to classifying correctly. The highest SNR where this happens is the crossing point where the noise becomes destructive for the classification, thus this SNR will be called the SNR_{cross}. The example in figure 6.2 shows a SNR_{cross} of 12dB.

In table 6.21 the SNR_{cross} is given for each encoding algorithm. The SNR_{cross} shows how much noise is needed to be disruptive for the classification. As the noise power is in the numerator, indicates a higher SNR_{cross} that less noise is needed to impact the accuracy of the prediction. Signals with an SNR higher than the corresponding SNR_{cross} from the table would give an 'accurate' prediction by the SVM. On the other hand, if the SNR is lower than the corresponding SNR_{cross}, the prediction will become inaccurate. So, having a lower SNR_{cross} is better and gives a higher resistance to the noise.

From table 6.21 we can see that the resistance to noise for p-SFE is worse than other encoding algorithms. Most of the noise resistances for p-SFE are zero or above, while for the other encoding algorithms only SFE, BSE and TCE have mostly a noise resistance above zero. This means that most of the other encoding algorithms have a lower, a better SNR_{cross}.

6.4 Discussion

In case of efficiency, we see for p-TE that only results for a few thresholds are given, while there should have been tests for up to ten thresholds. This is because at a certain point adding thresholds to p-TE would not result to better efficiency. Thus taking configurations with more thresholds were omitted and not taken into consideration.

There have been several decisions made to limit the computation time as many comparisons already require a lot of computation. For example, the population encoding algorithms were limited to ten thresholds. This would give enough samples to see the growth of the metrics with the number of thresholds used. Another example would be that only Free Spoken MNIST was tested for the classification accuracy, which was mainly because training for the spike trains from the Speech Command needed a lot of RAM memory and time. Moreover, the saved files needed became a bit too big to comfortably work with. One more example would be the number of thresholds used by the population encoding algorithms for the classification of the combined spike trains.

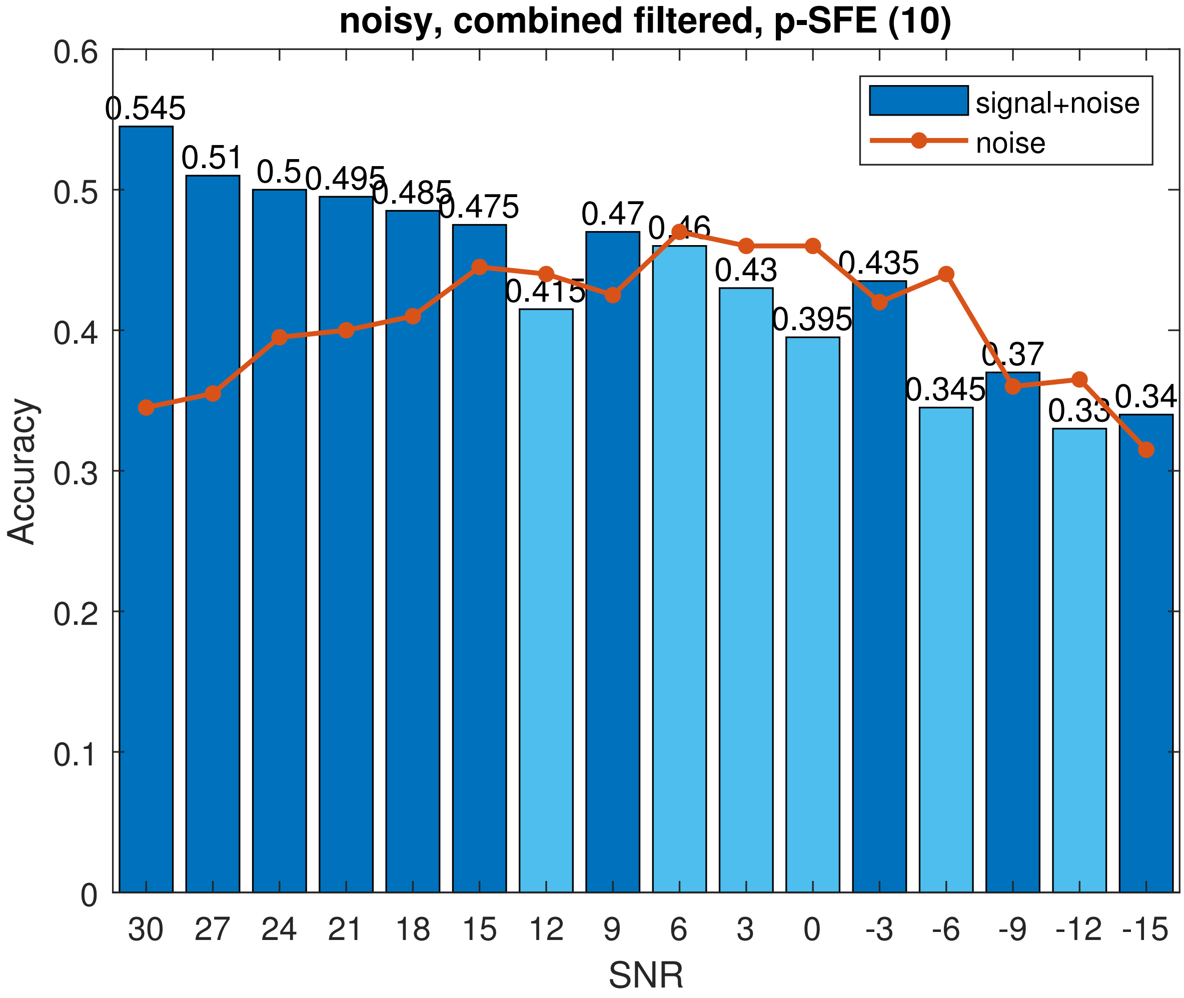

Figure 6.2: Classification accuracies for different Signal-to-Noise values of p-SFE with ten thresholds on the spike trains of the three filtered signals combined. Lighter blue bars are the SNR at which the classification of only noise outperforms the classification of the signal with noise.

For both population algorithms, the configurations with many thresholds are used even though for the higher filtered case fewer thresholds give better results. This is because for the combined spike trains the spike trains of the same amount of thresholds were only combined, otherwise the number of combinations that would need to be tested for these population encoding algorithms would be 9^3 instead of just 9.

In the case with only noise added to the SVM we would expect that the classification accuracy would be around the same classification accuracy for randomly guessing the class. However, from figure 6.2 we see that the classification of the noise reaches almost 50% which is much higher than the expected 10%, as there are 10 classes in the Free Spoken MNIST dataset. As we use white Gaussian noise is there no aspect of the noise that can be used for classification except for the power of the signal. So we can see that the power of the signal becomes a big factor for determining the class.

Table 6.21: Lowest SNR values in dB for which the algorithms failed to be more accurate with the signal with noise data over data from only noise. These SNR values can be from 30dB to -15dB. In some cases the signal with noise always achieved better accuracy, then ’-’ is put there.

	raw	normalized	low filtered	mid filtered	high filtered	combined filtered
BSE	-15	-9	30	30	30	30
LE	30	30	-15	-	-	-12
TE	3	-6	-15	-	-12	-6
TCE	-3	-3	12	9	6	6
MWE	-9	-15	-6	-6	-12	3
SFE	0	-6	9	-3	0	6
p-TE (2)	15	-9	-12	-	-3	-
p-TE (3)	15	-	-	-	-9	-
p-TE (4)	15	-	-	-	-9	-
p-TE (5)	15	-	-	-	-12	-
p-TE (6)	-3	-	-	-12	-12	-15
p-TE (7)	15	-12	-	-	-9	-
p-TE (8)	15	-	-	-	-9	-
p-TE (9)	15	-15	-	-	-9	-
p-TE (10)	15	-9	-	-	-9	-
p-SFE (2)	-9	-6	6	6	3	6
p-SFE (3)	6	-6	9	3	3	12
p-SFE (4)	6	0	12	3	6	12
p-SFE (5)	6	0	9	-3	12	12
p-SFE (6)	3	0	9	6	3	12
p-SFE (7)	0	0	9	12	9	12
p-SFE (8)	9	0	9	6	12	6
p-SFE (9)	9	3	12	6	21	12
p-SFE (10)	9	3	15	18	18	12

7 Conclusion

A new encoding algorithm is proposed to encode temporal digital data into spike trains usable by Spiking Neural Network (SNN), population Step Forward Encoding algorithm (p-SFE). The idea is to take the signal encoding accuracy of SFE and improve the efficiency by using a population of thresholds, which would allow the bigger thresholds to encode multiple spikes of the smaller thresholds. This should allow for better signal accuracy when there is a limiting bandwidth of spikes. The proposed algorithm, p-SFE is tested and compared for encoding signal accuracy, efficiency and classification potential.

Firstly, the signal encoding accuracy of p-SFE was compared. p-SFE had a higher maximum signal accuracy than almost all encoding algorithms, with only an exception to LE. However, this only held for raw and normalized audio signals. Moreover, LE is highly sensitive to spike time noise, which could make p-SFE a better solution even for those cases. Thus, p-SFE is shown to be universally one of the best if not the best choice regarding signal accuracy.

Secondly, the efficiency of the encoding algorithms were compared. The maximum efficiency of p-SFE is higher than all other encoding algorithms for all cases. Even though p-TE manages to have a higher efficiency in one case, the results suggest that with only one or two thresholds added p-SFE would be likely to have a better efficiency. However, a downside was noticed, which was the number of spikes needed to achieve the best efficiency. The spikes needed was significantly higher, which could have a very negative impact on the case of a limited bandwidth for spikes. When looking at the p-SFE configurations with a spike count similar to the other algorithms, we could see that the efficiency is lower, but similar to those of other encoding algorithms. This shows that p-SFE can get the best efficiency and even when limited by the number of spikes would it still be a good choice.

Thirdly, the classification potential was explored. When looking at the raw classification accuracy p-SFE was a rarely beaten champion and performed in all cases either the best or second best. Even when placing second best, p-SFE proved to be almost just as good. However, p-SFE showed its weakness when the resistance to noise was tested. The results show that p-SFE is relatively weak to the noise and even the best resistances of p-SFE are among the worst resistances compared with the other algorithms.

Looking back at the intention for p-SFE, it does not achieve the best efficiency when the spike count is limited. This could mean that it would not achieve the best signal encoding accuracy when the spike bandwidth is limited severely. However, p-SFE would not be far behind the best efficiency of the other encoding algorithms. Moreover, when the limitation would be slackened more p-SFE could easily make use of it and increase the signal accuracy much faster than the other encoding algorithms could. This makes p-SFE the encoding algorithm which is universally a good option if not the best.

This doesn't only apply to regression problems as tested with the signal encoding accuracy and efficiency, but also to classification problems. In which it also showed universally one of the best potentials, *if* the noise doesn't get too high.

7.1 Future work

The simplified version for p-SFE could get extended to be used with an arbitrary factor. In this thesis the factor between thresholds is set to 2. However, it could be that using another factor would give better results for certain datasets. This would add only one extra variable and doesn't add much complexity to the search for an optimal configuration. The advantage for choosing a factor of 2 is that it can be done with a simple shift operation.

The more complex version of p-SFE can also be explored. If the thresholds are chosen arbitrarily, is the design space much bigger, because each threshold will add its own dimension. With such a bigger design space is it more likely to contain better configurations. Though finding these better configurations will be much harder, as the design space grows with the number of thresholds. Thus is searching within that design space likely best done by an evolutionary algorithm. Moreover, each threshold change will have an impact on the behavior of the other thresholds; changing one threshold would force you to change other thresholds as well.

One of the weak points is the method for determining the classification potential. Using an SVM instead of an SNN makes the results inaccurate, as it deviates from how the spike trains will be used. Still, the advantage or disadvantage should apply to all encoding algorithms evenly and not change the comparison made in this Thesis. However, finding a method for determining classification potential without prejudice for any encoding algorithm is preferred. This could be achieved by either finding a spike train distance that would give sensible results or prove that a certain SNN classification method doesn't have a prejudice to any encoding algorithm.

Tough it was theorized that dropping spikes could drastically reduce the signal encoding accuracy and thus the efficiency, more research could be used. The rate of signal accuracy and/or efficiency loss with dropping spike is useful to determine how many spikes can be dropped without significant impact. This can be used to find how close to the spike limit the encoding algorithms can be configured. This could also show that dropping spike might affect certain encoding algorithms differently.

Comparing the encoding algorithms for the computational complexity is not really straight forward. The main cause for this is that the exact implementation can have a big impact on the complexity. Moreover, A direct hardware implementation for p-SFE has not been made yet. Fortunately, in appendix D a first hardware implementation draft for the simplified p-SFE is given. This implementation shows an implementation with an even smaller footprint than that as given in chapter 4, as it is based on the simplified p-SFE and only two threshold need to be remembered. From this example we can see that a specific implementation could have a huge impact on the computational complexity. Thus, finding an efficient hardware implementation for p-SFE is important.

A MATLAB code

A.1 Ben's Spiker Encoding algorithm

A.1.1 BSA_encode

```
function [spikes, signal_rest, threshold] = BSA_encode(signal, filter,
    varargin)
%BSA_encode     Ben's Spiker encoding Algorithm - encode
%   Encodes a time varying analog signal into spike trains using Ben's
%   Spiker encoding Algorithm.
%   The writen algorithm is based on the pseudo code as given in the
    paper
%   "BSA, a Fast and Accurate Spike Train Encoding Scheme"
%   (https://doi.org/10.1109/IJCNN.2003.1224019) by Benjamin Schrauwen
    and
%   Jan Van Campenhout.
%
%   [spikes, threshold, signal_rest] = BSA_encode(signal, filter,
    varargin)
%
%   - The input 'signal' should be a T-by-n matrix. Where 'T' is the
    number
%   of time samples and 'n' is the number of input signals. 'n' can even
    be
%   a matrix.
%
%   - The input 'filter' should be a F-by-m matrix or a 1-by-F matrix,
%   depending on the shape of the input 'signal'. This specifies the
%   filter used in the algorithm. Where 'F' is the size of the filter and
%   'm' is either 1 or equal to 'n', the number of input signals.
%
%   Optional inputs are 'fraction', 'threshold', 'nonCausal' and 'Causal
    '.
%
%   - The option 'fraction' should be followed by a float between the
%   values 0 and 1. This will calculate a theshold value based on the
    input
%   filter(s). It will have a default value of 0.1, if not set.
%
%   - The option 'threshold' should be followed by a float. This will set
%   the threshold to the float value. If the threshold is set the 'factor
    '
%   option will be ignored.
%
%   - The option 'nonCausal' will make sure that the filter will be
%   subtracted from the following time samples.
%
%   - The option 'Causal' will make sure that the filter will be
   subtracted
%   from the previous time samples.
%
%   From the options 'nonCausal' and 'Causal', the last options in the
%   input arguments will be applied. The default option is 'Causal'.
%
%   - The output 'spikes' will contain the spike trains in the same
   format
%   as the 'signal' input.
%
%   - The output 'signal_rest' will return the leftover from the signal,
%   which was not allowed to be sutracted by the filter.
%
%   - The output 'threshold' will return the used threshold.
%
%   To decode the spike train back to roughly the original signal use the
%   function filtered_decode().
%
%   Author: Luuk de Gelder

    % default values
    spikes = NaN;
    threshold = NaN;
    signal_rest = NaN;
    useFilterCausally = true;
    useMEX = false;
    fraction = 0.1;

    % input parsing
    if nargin >= 2

        % variable input parsing
        Narg = numel(varargin);
        for i = 1:Narg
            if ischar(varargin{i})
                str = varargin{i};
                nextIsFl = false;
                if i + 1 <= Narg
                    nextIsFl = isfloat(varargin{i + 1});
                end

                if nextIsFl
                    if strcmp(str, 'fraction')
                        fraction = varargin{i + 1};
                    elseif strcmp(str, 'threshold')
                        threshold = varargin{i + 1};
                    end
                end

                if strcmp(str, 'nonCausal')
                    useFilterCausally = false;
                elseif strcmp(str, 'Causal')
                    useFilterCausally = true;
                elseif strcmp(str, 'MEX')
                    useMEX = true;
                end
            end
        end

        Ss = size(signal);
        Sf = size(filter);

        % calculate threshold if not set
        if isnan(threshold)
            threshold = fraction * sum(abs(filter));
        end

        startFromCell = iscell(signal);
        if startFromCell
            spikes = cell(Ss);
            signal_rest = cell(Ss);
            Ns = prod(Ss);
        else
            spikes = cell(1);
            signal_rest = cell(1);
            Ns = 1;
            signal = {signal};
        end

        for i = 1:Ns
            signalM = signal{i};
            SsM = size(signalM);

            % check input shape
            correctFilterSize = Sf(2) == 1 || sameVectors(Sf(2:end), SsM(2:end));

            % construct some temporary variables
            if ~correctFilterSize
                if Sf(1) == 1 || Sf(1) == SsM(2)
                    filter = filter';
                else
                    error('Filter size does not correspont to signal size.');
                    %return
                end
            end

            % reshape if necesary
            if numel(SsM) > 2
                signalM = reshape(signalM, [SsM(1) prod(SsM(2:end))]);
            end
            if numel(Sf) > 2
```

```
                filter = reshape(filter, [Sf(1) prod(Sf(2:end))]);
            end

            % run BSA (in separate file for MEX optimization)
            if useFilterCausally
                if useMEX
                    if isa(signalM, 'single')
                        [spikesM, signal_restM] = BSA_causal_mex_single(
                           signalM, filter, threshold);
                    else
                        [spikesM, signal_restM] = BSA_causal_mex(signalM,
                            filter, threshold);
                    end
                else
                    [spikesM, signal_restM] = BSA_causal(signalM, filter,
                        threshold);
                end
            else
                if useMEX
                    if isa(signalM, 'single')
                        [spikesM, signal_restM] =
                           BSA_noncausal_mex_single(signalM, filter,
                           threshold);
                    else
                        [spikesM, signal_restM] = BSA_noncausal_mex(
                           signalM, filter, threshold);
                    end
                else
                    [spikesM, signal_restM] = BSA_noncausal(signalM,
                       filter, threshold);
                end
            end

            % reshape back
            if numel(SsM) > 2
                signal_restM = reshape(signal_restM, SsM);
                spikesM = reshape(spikesM, SsM);
            end

            spikes(i) = {spikesM};
            signal_rest(i) = {signal_restM};
        end

        if ~startFromCell
            spikes = spikes{1};
            signal_rest = signal_rest{1};
        end
    end
end
```

A.1.2 BSA_causal

```
function [spikes, signal_rest] = BSA_causal(signal, filters, threshold)
```

```
    Ss = size(signal);
    Sf = size(filters);
    Flen = Sf(1);
    filters_windowed = filters;
    spikes = nan(Ss);

    % loop the signal values
    for t = 1:Ss(1)
        % cutoff the filter if necessary
        if t <= Sf(1)
            Flen = t;
            filters_windowed = filters(end-Flen+1:end, :);
        end
        T = (t-Flen+1:t);

        % no signal, no spike
        if isnan(signal(t, i))
            continue
        end

        % check if the filter fits within the signal, then spike
        for i = 1:Ss(2)
            % pick corresponding filter
            if Sf(2) ~= 1
                filter = filters_windowed(:, i);
            else
                filter = filters_windowed;
            end

            % check for spike
            E1 = sum(abs(signal(T, i) - filter), 1, 'omitnan');
            E2 = sum(abs(signal(T, i)), 1, 'omitnan');
            if E1 <= (E2 - threshold)
                % apply spike, remove filter from the signal
                spikes(t, i) = 1;
                signal(T, i) = signal(T, i) - filter;
            end
        end
    end
    signal_rest = signal;
end
```

A.2 Latency Encoding algorithm

A.2.1 LA_encode

```
function spikesRT = LA_encode(signal, windowReach, varargin)
%LA_encode     Latency encoding Algorithm - encode
    windowFrame = [0 1];
    useMEX = false;

    % input parsing
    if nargin >= 2

    % variable input parsing
    Narg = numel(varargin);
    for i = 1:Narg
        if ischar(varargin{i})
            str = varargin{i};
            nextIsFl = false;
            if i + 1 <= Narg
                nextIsFl = isfloat(varargin{i + 1});
            end

            if nextIsFl
                if strcmp(str, 'frame')
                    windowFrame = varargin{i + 1};
                end
            end

            if strcmp(str, 'MEX')
                useMEX = true;
            end
        end
    end

    Ss = size(signal);

    startFromCell = iscell(signal);
    if startFromCell
        spikesRT = cell(Ss);
        Ns = prod(Ss);
    else
        spikesRT = cell(1);
        Ns = 1;
        signal = {signal};
    end

    for i = 1:Ns
        signalM = signal{i};
        SsM = size(signalM);

        % reshape if necesary
        if numel(SsM) > 2
            signalM = reshape(signalM, [SsM(1) prod(SsM(2:end))]);
        end

        % run LA (in separate file for MEX optimization)
        if useMEX
            if isa(signalM, 'single')
                spikesRTM = rangeTransform_mex_single(signalM,
                    windowReach, windowFrame);
            else
                spikesRTM = rangeTransform_mex(signalM, windowReach,
                    windowFrame);
            end
            else
                spikesRTM = rangeTransform(signalM, windowReach,
                    windowFrame);
            end

            % reshape back
            if numel(SsM) > 2
                spikesRTM = reshape(spikesRTM, SsM);
            end

            spikesRT(i) = {spikesRTM};
        end


        if ~startFromCell
            spikesRT = spikesRT{1};
        end
    end
end
```

A.2.2 rangeTransform

```
function transformed = rangeTransform(signal, signalRange, transformRange
    )
    % Warp the signal to the domain of [0 1] with respect to its
        specified
    % range.
    warped = (signal - signalRange(1)) / (signalRange(2) - signalRange(1)
        );
    % Remove values outside of the signal range.
    warped(warped < 0 | warped > 1) = NaN;
    % Transform the warped signal to the specified transform range.
    transformed = (warped * (transformRange(2) - transformRange(1))) +
        transformRange(1);
end
```

A.3 Threshold Encoding algorithm

A.3.1 TA_encode

```
function spikes = TA_encode(signal, thresholds, varargin)
    % default values
    spikes = NaN;
    useMEX = false;

    % input parsing
    if nargin >= 2

        % variable input parsing
        Narg = numel(varargin);
        for i = 1:Narg
            if ischar(varargin{i})
                str = varargin{i};
```

```
%                 nextIsFl = false;
%                 if i + 1 <= Narg
%                     nextIsFl = isfloat(varargin{i + 1});
%                 end
%
%                 if nextIsFl
%                 end

                if strcmp(str, 'MEX')
                    useMEX = true;
                end
            end
        end

        % check input shape
        Ss = size(signal);
        Nthres = numel(thresholds);

        startFromCell = iscell(signal);
        if startFromCell
            spikes = cell(Ss);
            Ns = prod(Ss);
        else
            spikes = cell(1);
            Ns = 1;
            signal = {signal};
        end

        for i = 1:Ns
            signalM = signal{i};
            SsM = size(signalM);

            % reshape if necesary
            if numel(SsM) > 2
                signalM = reshape(signalM, [SsM(1) prod(SsM(2:end))]);
            end

            % run TA (in separate files for MEX optimization)
            if Nthres > 1
                if useMEX
                    if isa(signalM, 'single')
                        spikesM = populationTA_encode_mex_single(signalM,
                            thresholds);
                    else
                        spikesM = populationTA_encode_mex(signalM,
                           thresholds);
                    end
                else
                    spikesM = populationTA_encode(signalM, thresholds);
                end
            else
                if useMEX
                    if isa(signalM, 'single')
```

```
                        spikesM = regularTA_encode_mex_single(signalM,
                           thresholds);
                    else
                        spikesM = regularTA_encode_mex(signalM,
                           thresholds);
                    end
                else
                    spikesM = regularTA_encode(signalM, thresholds);
                end
            end

            % reshape back
            if numel(SsM) > 2
                if Nthres > 1
                    spikesM = reshape(spikesM, [SsM Nthres]);
                else
                    spikesM = reshape(spikesM, SsM);
                end
            elseif Nthres == 1
                spikesM = squeeze(spikesM);
            end

            spikes(i) = {spikesM};
        end


        if ~startFromCell
            spikes = spikes{1};
        end
    end
end
```

A.3.2 regularTA_encode

```
function spikes = regularTA_encode(signal, threshold)
    Ss = size(signal);
    mem = zeros([1 Ss(2)], 'logical');
    spikes = nan(Ss);
    for t = 1:Ss(1)
        spike_pos = signal(t, :) >= threshold & ~mem;
        spike_neg = signal(t, :) < threshold & mem;
        mem = xor(mem, spike_neg) | spike_pos;
        spikes(t, spike_pos) = 1;
        spikes(t, spike_neg) = -1;
    end
end
```

A.4 Temporal Contrast Encoding algorithm

A.4.1 TCA_encode

```
function [spikes, threshold] = TCA_encode(signal, varargin)
%TCA_encode     Temporal Contrast encoding Algorithm - encode
```

```
%   Encodes a time varying analog signal into spike trains using a
%   Temporal Contrast encoding Algorithm.
%   The writen algorithm is based on the pseudo code as given in the book
%   "Time-Space, Spiking Neural Networks and Brain-inspired Artificial
%   Intelligence" by Nikola K. Kasabov.
%
%   [spikes, threshold] = TCA_encode(signal, varargin)
%
%   - The input 'signal' should be a T-by-n matrix. Where 'T' is the
  number
%   of time samples and 'n' is the number of input signals. 'n' can even
  be
%   a matrix.
%
%   Optional inputs are 'factor' and 'threshold'.
%
%   - The option 'factor' should be followed by a float. This will
%   calculate a theshold value based on the input signal(s). It will have
    a
%   default value of 1, if not set.
%
%   - The option 'threshold' should be followed by a float. This will set
%   the threshold to the float value. If the threshold is set the 'factor
  '
%   option will be ignored.
%
%   - The output 'spikes' will contain the spike trains in the same
  format
%   as the 'signal' input.
%
%   - The output 'threshold' will return the used threshold.
%
%   To decode the spike train back to roughly the original signal use the
%   function generic_decode().
%
%   Author: Luuk de Gelder

    % default values
    spikes = NaN;
    threshold = NaN;
    factor = 1;
    useBase = true;
    useMEX = false;

    % input parsing
    if nargin >= 1

        % variable input parsing
        Narg = numel(varargin);
        for i = 1:Narg
            if ischar(varargin{i})
                str = varargin{i};
                nextIsFl = false;
        if i + 1 <= Narg
            nextIsFl = isfloat(varargin{i + 1});
        end

        if nextIsFl
            if strcmp(str, 'factor')
                factor = varargin{i + 1};
            elseif strcmp(str, 'threshold')
                threshold = varargin{i + 1};
            end
        end

        if strcmp(str, 'MEX')
            useMEX = true;
        elseif strcmp(str, 'withBase')
            useBase = true;
        elseif strcmp(str, 'noBase')
            useBase = false;
        end
    end
end

% check input shape
Ss = size(signal);

% set threshold if not specified
if isnan(threshold)
    dif = diff(signal, 1);
    threshold = mean(dif, 'omitnan') + factor * std(dif, 'omitnan');
end

startFromCell = iscell(signal);
if startFromCell
    spikes = cell(Ss);
    Ns = prod(Ss);
else
    spikes = cell(1);
    Ns = 1;
    signal = {signal};
end

for i = 1:Ns
    signalM = signal{i};
    SsM = size(signalM);

    % reshape if necesary
    if numel(SsM) > 2
        signalM = reshape(signalM, [SsM(1) prod(SsM(2:end))]);
    end

    % run TCA (in separate files for MEX optimization)
    if useBase
```

```
                if useMEX
                    if isa(signalM, 'single')
                        spikesM = TCA_base_mex_single(signalM, threshold)
                           ;
                    else
                        spikesM = TCA_base_mex(signalM, threshold);
                    end
                else
                    spikesM = TCA_base(signalM, threshold);
                end
            else
                if useMEX
                    if isa(signalM, 'single')
                        spikesM = TCA_nobase_mex_single(signalM,
                           threshold);
                    else
                        spikesM = TCA_nobase_mex(signalM, threshold);
                    end
                else
                    spikesM = TCA_nobase(signalM, threshold);
                end
            end

            % reshape back
            if numel(SsM) > 2
                spikesM = reshape(spikesM, SsM);
            end

            spikes(i) = {spikesM};
        end


        if ~startFromCell
            spikes = spikes{1};
        end
    end
end
```

A.4.2 TCA_nobase

```
function spikes = TCA_nobase(signal, threshold)
    Ss = size(signal);

    spikes = nan(Ss);
    % Loop the signal values
    for t = 2:Ss(1)
        % Calculate the diffrence and check for spikes
        Diff = signal(t, :) - signal(t - 1, :);
        spikes(t, Diff >= threshold) = 1;
        spikes(t, Diff <= -threshold) = -1;
    end
end
```

A.5 Moving Window Encoding algorithm

A.5.1 MWA_encode

```
function [spikes, threshold] = MWA_encode(signal, windowSize, varargin)
%MWA_encode     Moving Window encoding Algorithm - encode
%   Encodes a time varying analog signal into spike trains using a
%   Moving Window encoding Algorithm.
%   The writen algorithm is based on the description as given in the book
%   "Time-Space, Spiking Neural Networks and Brain-inspired Artificial
%   Intelligence" by Nikola K. Kasabov.
%
%   [spikes, threshold] = MWA_encode(signal, varargin)
%
%   - The input 'signal' should be a T-by-n matrix. Where 'T' is the
  number
%   of time samples and 'n' is the number of input signals. 'n' can even
  be
%   a matrix.
%
%   - The input 'windowSize' should be an integer value. This specifies
  the
%   size for the moving window which is used in the algorithm.
%
%   Optional inputs are 'factor' and 'threshold'.
%
%   - The option 'factor' should be followed by a float. This will
%   calculate a theshold value based on the input signal(s). It will have
   a
%   default value of 1, if not set.
%
%   - The option 'threshold' should be followed by a float. This will set
%   the threshold to the float value. If the threshold is set the 'factor
  '
%   option will be ignored.
%
%   - The output 'spikes' will contain the spike trains in the same
  format
%   as the 'signal' input.
%
%   - The output 'threshold' will return the used threshold.
%
%   To decode the spike train back to roughly the original signal use the
%   function generic_decode().
%
%   Author: Luuk de Gelder

    % default values
    spikes = NaN;
    threshold = NaN;
    factor = 1;
    useRegular = true;
    useBase = true;
    useMEX = false;

    % input parsing
    if nargin >= 2

        % variable input parsing
        Narg = numel(varargin);
        for i = 1:Narg
            if ischar(varargin{i})
                str = varargin{i};
                nextIsFl = false;
                if i + 1 <= Narg
                    nextIsFl = isfloat(varargin{i + 1});
                end

                if nextIsFl
                    if strcmp(str, 'factor')
                        factor = varargin{i + 1};
                    elseif strcmp(str, 'threshold')
                        threshold = varargin{i + 1};
                    end
                end

                if strcmp(str, 'MEX')
                    useMEX = true;
                elseif strcmp(str, 'withBase')
                    useBase = true;
                    useRegular = false;
                elseif strcmp(str, 'noBase')
                    useBase = false;
                    useRegular = false;
                elseif strcmp(str, 'regular')
                    useRegular = true;
                end
            end
        end

        % check input shape
        Ss = size(signal);

        % set threshold if not specified
        if isnan(threshold)
            dif = diff(signal, 1);
            threshold = mean(dif, 'omitnan') + factor * std(dif, 'omitnan
               ');
        end

        startFromCell = iscell(signal);
        if startFromCell
            spikes = cell(Ss);
            Ns = prod(Ss);
        else
            spikes = cell(1);
            Ns = 1;
            signal = {signal};
        end

        for i = 1:Ns
            signalM = signal{i};
            SsM = size(signalM);

            % reshape if necesary
            if numel(SsM) > 2
                signalM = reshape(signalM, [SsM(1) prod(SsM(2:end))]);
            end

            % run MWA (in separate files for MEX optimization)
            if useRegular
                if useMEX
                    if isa(signalM, 'single')
                        spikesM = MWA_regular_mex_single(signalM,
                            windowSize, threshold);
                    else
                        spikesM = MWA_regular_mex(signalM, windowSize,
                            threshold);
                    end
                else
                    spikesM = MWA_regular(signalM, windowSize, threshold)
                       ;
                end
            elseif useBase
                if useMEX
                    if isa(signalM, 'single')
                        spikesM = MWA_base_mex_single(signalM, windowSize
                            , threshold);
                    else
                        spikesM = MWA_base_mex(signalM, windowSize,
                            threshold);
                    end
                else
                    spikesM = MWA_base(signalM, windowSize, threshold);
                end
            else
                if useMEX
                    if isa(signalM, 'single')
                        spikesM = MWA_nobase_mex_single(signalM,
                            windowSize, threshold);
                    else
                        spikesM = MWA_nobase_mex(signalM, windowSize,
                            threshold);
                    end
                else
                    spikesM = MWA_nobase(signalM, windowSize, threshold);
                end
            end
```

```
            % reshape back
            if numel(SsM) > 2
                spikesM = reshape(spikesM, Ss);
            end

            spikes(i) = {spikesM};
        end


        if ~startFromCell
            spikes = spikes{1};
        end
    end
end
```

A.5.2 MWA_regular

```
function spikes = MWA_regular(signal, windowSize, threshold)
    Ss = size(signal);

    spikes = nan(Ss);
    windowSum = signal(1, :);
    windowMean = signal(1, :);
    windowN = double(~isnan(windowSum));
    % Loop the signal values
    for t = 2:Ss(1)
        % Calculate the mean of the window
        for i = 1:Ss(2)
            if t > windowSize + 1 && ~isnan(signal(t - windowSize - 1, i)
                )
                windowSum(i) = windowSum(i) - signal(t - windowSize - 1,
                    i);
                windowN(i) = windowN(i) - 1;
            end
            if isnan(signal(t - 1, i))
                continue
            end
            if isnan(windowSum(i))
                windowSum(i) = signal(t - 1, i);
            else
                windowSum(i) = windowSum(i) + signal(t - 1, i);
            end
            windowN(i) = windowN(i) + 1;
            windowMean(i) = windowSum(i) / windowN(i);
        end

        % Calculate the diffrence and check for spikes
        Diff = signal(t, :) - windowMean;
        spikes(t, Diff >= threshold) = 1;
        spikes(t, Diff <= -threshold) = -1;
    end
end
```

A.6 Step Forward Encoding algorithm

A.6.1 SFA_encode

```
function spikes = SFA_encode(signal, thresholds, varargin)
%SFA_encode     Step Forward encoding Algorithm - encode
%   Encodes a time varying analog signal into spike trains using a
%   Step Forward encoding Algorithm. When multiple thresholds are
   supplied
%   a population variant is applied.
%   The writen algorithm is based on the description as given in the book
%   "Time-Space, Spiking Neural Networks and Brain-inspired Artificial
%   Intelligence" by Nikola K. Kasabov.
%   The population variant is thought up by the author of this code.
%
%   [spikes] = SFA_encode(signal, thresholds)
%
%   - The input 'signal' should be a T-by-n matrix. Where 'T' is the
   number
%   of time samples and 'n' is the number of input signals. 'n' can even
   be
%   a matrix.
%
%   - The input 'thresholds' should be a 1D vector of floats in
   descending
%   order (length of 1 is allowed). This will be the threshold values
   used
%   by the algorithm.
%
%   - The output 'spikes' will contain the spike trains in the same
   format
%   as the 'signal' input.
%
%   To decode the spike train back to roughly the original signal use the
%   function SFA_decode().
%
%   Author: Luuk de Gelder

    % default values
    spikes = NaN;
    useMEX = false;

    % input parsing
    if nargin >= 2

        % variable input parsing
        Narg = numel(varargin);
        for i = 1:Narg
            if ischar(varargin{i})
                str = varargin{i};

                if strcmp(str, 'MEX')
                useMEX = true;
            end
        end
    end

    % check input shape
    Ss = size(signal);
    Nthres = numel(thresholds);

    startFromCell = iscell(signal);
    if startFromCell
        spikes = cell(Ss);
        Ns = prod(Ss);
    else
        spikes = cell(1);
        Ns = 1;
        signal = {signal};
    end

    for i = 1:Ns
        signalM = signal{i};
        SsM = size(signalM);

        % reshape if necesary
        if numel(SsM) > 2
            signalM = reshape(signalM, [SsM(1) prod(SsM(2:end))]);
        end

        % run SFA (in separate files for MEX optimization)
        if Nthres > 1
            if useMEX
                if isa(signalM, 'single')
                    spikesM = populationSFA_encode_mex_single(signalM
                        , thresholds);
                else
                    spikesM = populationSFA_encode_mex(signalM,
                        thresholds);
                end
            else
                spikesM = populationSFA_encode(signalM, thresholds);
            end
        else
            if useMEX
                if isa(signalM, 'single')
                    spikesM = regularSFA_encode_mex_single(signalM,
                        thresholds);
                else
                    spikesM = regularSFA_encode_mex(signalM,
                        thresholds);
                end
            else
                spikesM = regularSFA_encode(signalM, thresholds);
            end
            end

            % reshape back
            if numel(SsM) > 2
                spikesM = reshape(spikesM, SsM);
            end

            spikes(i) = {spikesM};
        end


        if ~startFromCell
            spikes = spikes{1};
        end
    end
end
```

A.6.2 regularSFA_encode

```
function spikes = regularSFA_encode(signal, threshold)
    Ss = size(signal);
    spikes = nan(Ss);
    base = zeros([1 Ss(2)]);
    for t = 1:Ss(1)
        D = signal(t, :) - base;
        spike_p = D >= threshold;
        spikes(t, spike_p) = 1;
        spike_n = D <= -threshold;
        spikes(t, spike_n) = -1;
        base = base + threshold * (spike_p - spike_n);
    end
end
```

A.7 population Threshold Encoding algorithm

A.7.1 populationTA_encode

```
function spikes = populationTA_encode(signal, thresholds)
    Ss = size(signal);
    Nthres = numel(thresholds);
    mem = zeros([Ss(2) Nthres], 'logical');
    spikes = nan([Ss Nthres]);
    for t = 1:Ss(1)
        for Tind = 1:Nthres
            threshold = thresholds(Tind);
            spike_pos = signal(t, :) >= threshold & ~mem(:, Tind);
            spike_neg = signal(t, :) < threshold & mem(:, Tind);
            mem(:, Tind) = xor(mem(:, Tind), spike_neg) | spike_pos;
            spikes(t, spike_pos, Tind) = 1;
            spikes(t, spike_neg, Tind) = -1;
        end
    end
end
```

A.8 population Step Forward Encoding algorithm

A.8.1 populationSFA_encode

```
function spikes = populationSFA_encode(signal, thresholds)
    Ss = size(signal);
    Nthres = numel(thresholds);
    spikes = nan([Ss Nthres]);
    base = zeros([1 Ss(2)]);
    for t = 1:Ss(1)
        for p = 1:Nthres
            D = signal(t, :) - base;
            spike_p = D >= thresholds(p);
            spikes(t, spike_p, p) = 1;
            spike_n = D <= -thresholds(p);
            spikes(t, spike_n, p) = -1;
            base = base + thresholds(p) * (spike_p - spike_n);
        end
    end
end
```

Supplementary data B

B.1 Signal encoding accuracy

Table B.1: Supplementary data to the results of *signal encoding accuracy* for the encoding algorithms from the *raw* audio of the *Free Spoken MNIST* dataset, see table 6.1. The parentheses behind the population encoding algorithms show the number of thresholds used, while the parenthesis behind the mean values are the stnadard deviation.

	T$_{exe}$	P$_{noSp}$	P$_{posSp}$	P$_{mixSp}$	ratio
BSE	0.3055	0.23	0.77	0	NaN
LE	0.05214	0	1	0	NaN
TE	0.1425	0	0.315	0.685	1.082 (± 0.1871)
TCE	0.1256	0.19	0	0.81	0.9997 (± 0.002377)
MWE	0.1404	0.015	0	0.985	1.11 (± 0.4137)
SFE	0.1064	0.19	0	0.81	0.9998 (± 0.002663)
p-TE (2)	0.2033	0.03	0	0.97	1 (± 0)
p-TE (3)	0.1443	0	0.03	0.97	1.05 (± 0.1301)
p-TE (4)	0.1843	0	0.03	0.97	1.05 (± 0.1302)
p-TE (5)	0.2305	0	0.03	0.97	1.073 (± 0.2114)
p-TE (6)	0.2781	0	0.03	0.97	1.11 (± 0.3171)
p-TE (7)	0.3135	0	0.03	0.97	1.11 (± 0.3171)
p-TE (8)	0.3611	0	0.03	0.97	1.109 (± 0.3173)
p-TE (9)	0.404	0	0.03	0.97	1.144 (± 0.4235)
p-TE (10)	0.4447	0	0.03	0.97	1.18 (± 0.5295)
p-SFE (2)	0.2493	0.015	0	0.985	1.01 (± 0.03506)
p-SFE (3)	0.1987	0.015	0	0.985	1.023 (± 0.05479)
p-SFE (4)	0.2672	0	0	1	1.017 (± 0.04318)
p-SFE (5)	0.325	0	0	1	1.028 (± 0.06145)
p-SFE (6)	0.3887	0	0	1	1.026 (± 0.06153)
p-SFE (7)	0.45	0	0	1	1.027 (± 0.062)
p-SFE (8)	0.5229	0	0	1	1.022 (± 0.05655)
p-SFE (9)	0.5855	0	0	1	1.023 (± 0.05602)
p-SFE (10)	0.6459	0	0	1	1.021 (± 0.05557)

Table B.2: Supplementary data to the results of *signal encoding accuracy* for the encoding algorithms from the *normalized* audio of the *Free Spoken MNIST* dataset, see table 6.2. The parentheses behind the population encoding algorithms show the number of thresholds used, while the parenthesis behind the mean values are the stnadard deviation.

	T_{exe}	P_{noSp}	P_{posSp}	P_{mixSp}	ratio
BSE	0.4874	0	1	0	NaN
LE	0.0515	0	1	0	NaN
TE	0.1457	0	0	1	1.03 (± 0.01382)
TCE	0.1293	0	0	1	0.9998 (± 0.002672)
MWE	0.1426	0	0	1	0.9428 (± 0.184)
SFE	0.1053	0	0	1	0.9997 (± 0.003374)
p-TE (2)	0.1606	0	0	1	1.026 (± 0.01191)
p-TE (3)	0.1379	0	0	1	1.009 (± 0.003389)
p-TE (4)	0.1854	0	0	1	1.009 (± 0.00353)
p-TE (5)	0.2284	0	0	1	1.008 (± 0.003065)
p-TE (6)	0.2739	0	0	1	1.011 (± 0.004369)
p-TE (7)	0.322	0	0	1	1.007 (± 0.002747)
p-TE (8)	0.3626	0	0	1	1.007 (± 0.002686)
p-TE (9)	0.3996	0	0	1	1.008 (± 0.003071)
p-TE (10)	0.4504	0	0	1	1.008 (± 0.003085)
p-SFE (2)	0.2039	0	0	1	1.011 (± 0.03164)
p-SFE (3)	0.1975	0	0	1	1.023 (± 0.0561)
p-SFE (4)	0.2636	0	0	1	1.022 (± 0.05626)
p-SFE (5)	0.3344	0	0	1	1.019 (± 0.05297)
p-SFE (6)	0.3925	0	0	1	1.018 (± 0.05462)
p-SFE (7)	0.4578	0	0	1	1.017 (± 0.05342)
p-SFE (8)	0.5222	0	0	1	1.016 (± 0.0527)
p-SFE (9)	0.5929	0	0	1	1.015 (± 0.05149)
p-SFE (10)	0.6532	0	0	1	1.015 (± 0.05072)

Table B.3: Supplementary data to the results of *signal encoding accuracy* for the encoding algorithms from the *low filtered* audio of the *Free Spoken MNIST* dataset, see table 6.3. The parentheses behind the population encoding algorithms show the number of thresholds used, while the parenthesis behind the mean values are the stnadard deviation.

	T_{exe}	P_{noSp}	P_{posSp}	P_{mixSp}	ratio
BSE	0.4886	1	0	0	NaN
LE	0.08245	0	1	0	NaN
TE	0.183	0	0	1	1 (± 0)
TCE	0.1841	0	0	1	1.002 (± 0.006776)
MWE	0.2202	0	0	1	2.022 (± 0.533)
SFE	0.1615	0	0	1	1.009 (± 0.008126)
p-TE (2)	0.1618	0	0	1	1 (± 0.002946)
p-TE (3)	0.1423	0	0	1	1 (± 0.002143)
p-TE (4)	0.1866	0	0	1	1 (± 0.003536)
p-TE (5)	0.2311	0	0	1	1 (± 0.002828)
p-TE (6)	0.2825	0	0	1	1 (± 0.0034)
p-TE (7)	0.3156	0	0	1	1 (± 0.003245)
p-TE (8)	0.3583	0	0	1	1 (± 0.003893)
p-TE (9)	0.4084	0	0	1	1.006 (± 0.005238)
p-TE (10)	0.4464	0	0	1	1.006 (± 0.004762)
p-SFE (2)	0.245	0	0	1	0.9898 (± 0.05233)
p-SFE (3)	0.2014	0	0	1	0.9834 (± 0.06718)
p-SFE (4)	0.2712	0	0	1	0.9842 (± 0.07035)
p-SFE (5)	0.33	0	0	1	0.9821 (± 0.0702)
p-SFE (6)	0.3955	0	0	1	0.9822 (± 0.06853)
p-SFE (7)	0.4666	0	0	1	0.9827 (± 0.06466)
p-SFE (8)	0.5306	0	0	1	0.9837 (± 0.06305)
p-SFE (9)	0.6074	0	0	1	0.9857 (± 0.05859)
p-SFE (10)	0.6645	0	0	1	0.9839 (± 0.05816)

Table B.4: Supplementary data to the results of *signal encoding accuracy* for the encoding algorithms from the *mid filtered* audio of the *Free Spoken MNIST* dataset, see table 6.4. The parentheses behind the population encoding algorithms show the number of thresholds used, while the parenthesis behind the mean values are the stnadard deviation.

	T_{exe}	P_{noSp}	P_{posSp}	P_{mixSp}	ratio
BSE	0.3021	1	0	0	NaN
LE	0.0494	0	1	0	NaN
TE	0.1353	0	0	1	1 (± 0)
TCE	0.1206	0	0	1	1.002 (± 0.006775)
MWE	0.1419	0	0	1	2.183 (± 0.6448)
SFE	0.1047	0	0	1	1.008 (± 0.007413)
p-TE (2)	0.1581	0	0	1	1 (± 0.003928)
p-TE (3)	0.1383	0	0	1	1 (± 0.003928)
p-TE (4)	0.1825	0	0	1	1.018 (± 0.01169)
p-TE (5)	0.2259	0	0	1	1.013 (± 0.01029)
p-TE (6)	0.2686	0	0	1	1.011 (± 0.008893)
p-TE (7)	0.3143	0	0	1	1.011 (± 0.008912)
p-TE (8)	0.3719	0	0	1	1.009 (± 0.008288)
p-TE (9)	0.3986	0	0	1	1.009 (± 0.008288)
p-TE (10)	0.4489	0	0	1	1.008 (± 0.007742)
p-SFE (2)	0.2225	0	0	1	0.9881 (± 0.03192)
p-SFE (3)	0.204	0	0	1	0.9792 (± 0.04728)
p-SFE (4)	0.2875	0	0	1	0.9744 (± 0.05291)
p-SFE (5)	0.3445	0	0	1	0.9714 (± 0.05757)
p-SFE (6)	0.3943	0	0	1	0.9695 (± 0.05872)
p-SFE (7)	0.4602	0	0	1	0.9697 (± 0.05876)
p-SFE (8)	0.5323	0	0	1	0.9707 (± 0.05604)
p-SFE (9)	0.596	0	0	1	0.9711 (± 0.05381)
p-SFE (10)	0.6621	0	0	1	0.9709 (± 0.05407)

Table B.5: Supplementary data to the results of *signal encoding accuracy* for the encoding algorithms from the *high filtered* audio of the *Free Spoken MNIST* dataset, see table 6.5. The parentheses behind the population encoding algorithms show the number of thresholds used, while the parenthesis behind the mean values are the stnadard deviation.

	T_{exe}	P_{noSp}	P_{posSp}	P_{mixSp}	ratio
BSE	0.2807	1	0	0	NaN
LE	0.05574	0.08	0.92	0	NaN
TE	0.1425	0.145	0	0.855	1.001 (± 0.01315)
TCE	0.1194	0.035	0.01	0.955	1.008 (± 0.04518)
MWE	0.1401	0.04	0.035	0.925	3.014 (± 2.705)
SFE	0.1158	0.015	0.015	0.97	1.026 (± 0.1075)
p-TE (2)	0.1608	0	0.495	0.505	1.276 (± 0.3226)
p-TE (3)	0.1378	0	0.495	0.505	1.274 (± 0.3242)
p-TE (4)	0.1867	0	0.495	0.505	1.274 (± 0.3243)
p-TE (5)	0.2302	0	0.495	0.505	1.266 (± 0.3288)
p-TE (6)	0.2761	0	0.495	0.505	1.266 (± 0.329)
p-TE (7)	0.3203	0	0.495	0.505	1.266 (± 0.329)
p-TE (8)	0.3606	0	0.495	0.505	1.266 (± 0.329)
p-TE (9)	0.4074	0	0.495	0.505	1.266 (± 0.329)
p-TE (10)	0.4408	0	0.495	0.505	1.266 (± 0.329)
p-SFE (2)	0.2056	0.01	0.005	0.985	0.99 (± 0.04464)
p-SFE (3)	0.1941	0.01	0.005	0.985	0.9669 (± 0.04553)
p-SFE (4)	0.2635	0	0.01	0.99	0.962 (± 0.08923)
p-SFE (5)	0.3245	0	0.005	0.995	0.9605 (± 0.08958)
p-SFE (6)	0.3869	0	0	1	0.9585 (± 0.1011)
p-SFE (7)	0.4543	0	0	1	0.9525 (± 0.05637)
p-SFE (8)	0.51	0	0	1	0.9533 (± 0.05306)
p-SFE (9)	0.5856	0	0	1	0.9562 (± 0.05028)
p-SFE (10)	0.6575	0	0	1	0.9591 (± 0.04893)

Table B.6: Supplementary data to the results of *signal encoding accuracy* for the encoding algorithms from the *raw* audio of the *Speech Commands* dataset, see table 6.6. The parentheses behind the population encoding algorithms show the number of thresholds used, while the parenthesis behind the mean values are the stnadard deviation.

	T_{exe}	P_{noSp}	P_{posSp}	P_{mixSp}	ratio
BSE	38.82	0.0361	0.9639	0	NaN
LE	1.826	0	1	0	NaN
TE	10.85	0	0.136	0.864	1.05 (± 0.1321)
TCE	8.504	0.01577	0	0.9842	0.9998 (± 0.01587)
MWE	8.019	0.00627	0.00342	0.9903	1.062 (± 1.214)
SFE	7.825	0.01653	0	0.9835	0.9999 (± 0.01423)
p-TE (2)	13.56	0	0.1229	0.8771	1.047 (± 0.1303)
p-TE (3)	20.3	0	0.05643	0.9436	1.027 (± 0.1009)
p-TE (4)	25.91	0	0.05643	0.9436	1.054 (± 0.2017)
p-TE (5)	30.79	0	0.05643	0.9436	1.053 (± 0.2019)
p-TE (6)	36.24	0	0.00836	0.9916	1.034 (± 0.1583)
p-TE (7)	42.2	0	0.00836	0.9916	1.044 (± 0.2111)
p-TE (8)	47.91	0	0.00551	0.9945	1.028 (± 0.171)
p-TE (9)	53.85	0	0.00551	0.9945	1.028 (± 0.1711)
p-TE (10)	59.98	0	0.00551	0.9945	1.028 (± 0.1711)
p-SFE (2)	18.25	0.00266	0.00019	0.9971	1.001 (± 0.0318)
p-SFE (3)	27.33	0.00057	0	0.9994	1.002 (± 0.04334)
p-SFE (4)	34.97	0.00038	0	0.9996	1.003 (± 0.05069)
p-SFE (5)	42.48	0	0.00038	0.9996	1.003 (± 0.05282)
p-SFE (6)	50.38	0	0.00019	0.9998	1.004 (± 0.06124)
p-SFE (7)	58.91	0	0.00019	0.9998	1.003 (± 0.05367)
p-SFE (8)	67.2	0	0	1	1.003 (± 0.05263)
p-SFE (9)	75.43	0	0	1	1.003 (± 0.05004)
p-SFE (10)	83.81	0	0	1	1.002 (± 0.04747)

Table B.7: Supplementary data to the results of *signal encoding accuracy* for the encoding algorithms from the *normalized* audio of the *Speech Commands* dataset, see table 6.7. The parentheses behind the population encoding algorithms show the number of thresholds used, while the parenthesis behind the mean values are the stnadard deviation.

	T_{exe}	P_{noSp}	P_{posSp}	P_{mixSp}	ratio
BSE	44.34	0	1	0	NaN
LE	1.804	0	1	0	NaN
TE	10.45	0.00095	0.00019	0.9989	1 (± 0.01423)
TCE	8.196	0	0	1	1 (± 0.001939)
MWE	7.506	0.00019	0.00057	0.9992	1.043 (± 1.312)
SFE	7.676	0	0	1	1 (± 0.004257)
p-TE (2)	13.39	0	0.00019	0.9998	1.001 (± 0.001188)
p-TE (3)	20.05	0	0.00019	0.9998	1.002 (± 0.001909)
p-TE (4)	25.53	0	0.00019	0.9998	1.001 (± 0.001386)
p-TE (5)	30.83	0	0.00019	0.9998	1.002 (± 0.001538)
p-TE (6)	36.2	0	0.00019	0.9998	1.002 (± 0.001242)
p-TE (7)	43.07	0	0.00019	0.9998	1.002 (± 0.001553)
p-TE (8)	48.22	0	0.00019	0.9998	1.003 (± 0.001729)
p-TE (9)	53.96	0	0.00019	0.9998	1.003 (± 0.001607)
p-TE (10)	59.98	0	0.00019	0.9998	1.003 (± 0.00169)
p-SFE (2)	26.06	0	0	1	1.001 (± 0.02925)
p-SFE (3)	38.86	0	0	1	1.002 (± 0.04214)
p-SFE (4)	50.76	0	0	1	1.002 (± 0.0466)
p-SFE (5)	64.84	0	0	1	1.003 (± 0.04976)
p-SFE (6)	77.76	0	0	1	1.003 (± 0.05113)
p-SFE (7)	89.36	0	0	1	1.003 (± 0.05081)
p-SFE (8)	100.7	0	0	1	1.003 (± 0.04933)
p-SFE (9)	114.8	0	0	1	1.003 (± 0.04771)
p-SFE (10)	128.7	0	0	1	1.002 (± 0.04541)

Table B.8: Supplementary data to the results of *signal encoding accuracy* for the encoding algorithms from the *low filtered* audio of the *Speech Commands* dataset, see table 6.8. The parentheses behind the population encoding algorithms show the number of thresholds used, while the parenthesis behind the mean values are the stnadard deviation.

	T_{exe}	P_{noSp}	P_{posSp}	P_{mixSp}	ratio
BSE	42.4	1	0	0	NaN
LE	5.704	0.00266	0.9973	0	NaN
TE	10.62	0.0513	0.00038	0.9483	1 (± 0.005289)
TCE	8.726	0.0019	0.00076	0.9973	1.001 (± 0.0351)
MWE	8.486	0	0.00019	0.9998	3.468 (± 13.92)
SFE	8.444	0.00171	0.00095	0.9973	1.006 (± 0.02905)
p-TE (2)	13.98	0.00171	0.00038	0.9979	1.001 (± 0.01029)
p-TE (3)	21.47	0.00171	0.00019	0.9981	1.001 (± 0.02272)
p-TE (4)	26.89	0.00171	0.00019	0.9981	1.001 (± 0.02983)
p-TE (5)	31.27	0.00057	0.00019	0.9992	1.001 (± 0.0355)
p-TE (6)	36.25	0.00057	0.00019	0.9992	1.001 (± 0.02876)
p-TE (7)	42.28	0.00057	0.00019	0.9992	1.001 (± 0.008191)
p-TE (8)	48.08	0	0.00076	0.9992	1.008 (± 0.01528)
p-TE (9)	54	0	0.00076	0.9992	1.006 (± 0.01506)
p-TE (10)	59.86	0	0.00076	0.9992	1.006 (± 0.01495)
p-SFE (2)	24.54	0.00057	0.00114	0.9983	0.9913 (± 0.04115)
p-SFE (3)	39.33	0	0.00057	0.9994	0.9834 (± 0.05472)
p-SFE (4)	48.86	0	0	1	0.9804 (± 0.06041)
p-SFE (5)	63.42	0	0	1	0.9788 (± 0.0604)
p-SFE (6)	72.62	0	0	1	0.979 (± 0.06159)
p-SFE (7)	83.57	0	0	1	0.9795 (± 0.06183)
p-SFE (8)	94.89	0	0	1	0.9802 (± 0.06112)
p-SFE (9)	107.9	0	0	1	0.9809 (± 0.06052)
p-SFE (10)	119.9	0	0	1	0.9818 (± 0.05933)

Table B.9: Supplementary data to the results of *signal encoding accuracy* for the encoding algorithms from the *mid filtered* audio of the *Speech Commands* dataset, see table 6.9. The parentheses behind the population encoding algorithms show the number of thresholds used, while the parenthesis behind the mean values are the stnadard deviation.

	T_{exe}	P_{noSp}	P_{posSp}	P_{mixSp}	ratio
BSE	40.93	1	0	0	NaN
LE	5.295	0.00019	0.9998	0	NaN
TE	10.61	0.00627	0.00019	0.9935	1.001 (± 0.01478)
TCE	8.646	0	0.00038	0.9996	1.001 (± 0.01125)
MWE	8.592	0	0.00019	0.9998	3.687 (± 13.95)
SFE	8.476	0.09253	0.8655	0.04199	1.579 (± 0.4256)
p-TE (2)	13.62	0	0.00019	0.9998	1.001 (± 0.02844)
p-TE (3)	21.32	0	0.00019	0.9998	1.001 (± 0.006417)
p-TE (4)	26.75	0	0.00019	0.9998	1.001 (± 0.00623)
p-TE (5)	31.5	0	0.00019	0.9998	1.001 (± 0.006077)
p-TE (6)	36.66	0	0.00019	0.9998	1.009 (± 0.01005)
p-TE (7)	42.94	0	0.00019	0.9998	1.007 (± 0.009352)
p-TE (8)	48.9	0	0	1	1.007 (± 0.013)
p-TE (9)	55	0	0	1	1.006 (± 0.01357)
p-TE (10)	61.08	0	0	1	1.006 (± 0.01437)
p-SFE (2)	23.64	0	0	1	0.9883 (± 0.02875)
p-SFE (3)	36.55	0	0	1	0.9797 (± 0.04166)
p-SFE (4)	46.76	0	0	1	0.9765 (± 0.04812)
p-SFE (5)	60.57	0	0	1	0.9754 (± 0.0521)
p-SFE (6)	71.69	0	0	1	0.975 (± 0.05293)
p-SFE (7)	81.26	0	0	1	0.9763 (± 0.05219)
p-SFE (8)	92.89	0	0	1	0.9762 (± 0.05273)
p-SFE (9)	107.6	0	0	1	0.9775 (± 0.05135)
p-SFE (10)	119.3	0	0	1	0.9784 (± 0.05036)

Table B.10: Supplementary data to the results of *signal encoding accuracy* for the encoding algorithms from the *high filtered* audio of the *Speech Commands* dataset, see table 6.10. The parentheses behind the population encoding algorithms show the number of thresholds used, while the parenthesis behind the mean values are the stnadard deviation.

	T_{exe}	P_{noSp}	P_{posSp}	P_{mixSp}	ratio
BSE	45.41	1	0	0	NaN
LE	5.366	0.1161	0.8839	0	NaN
TE	10.83	0.2681	0.00019	0.7317	1.001 (± 0.02195)
TCE	8.966	0.03971	0.01121	0.9491	1.009 (± 0.09343)
MWE	8.429	0.03306	0.04294	0.924	8.271 (± 22.75)
SFE	8.393	0.02926	0.02869	0.942	1.029 (± 0.122)
p-TE (2)	13.64	0.6738	0	0.3262	1.002 (± 0.03629)
p-TE (3)	21.19	0.6738	0	0.3262	1.001 (± 0.02968)
p-TE (4)	26.4	0.6738	0	0.3262	1.001 (± 0.01521)
p-TE (5)	31.12	0.6738	0	0.3262	1.001 (± 0.01426)
p-TE (6)	36.15	0.6738	0	0.3262	1.001 (± 0.01426)
p-TE (7)	42.15	0.6738	0	0.3262	1.001 (± 0.01674)
p-TE (8)	48	0.6738	0	0.3262	1.001 (± 0.01452)
p-TE (9)	54.49	0.6738	0	0.3262	1.001 (± 0.0135)
p-TE (10)	54.5	0.6738	0	0.3262	1.001 (± 0.0135)
p-SFE (2)	24.03	0.019	0.01824	0.9628	1.004 (± 0.1057)
p-SFE (3)	35.92	0.00893	0.01368	0.9774	0.9838 (± 0.08319)
p-SFE (4)	45.99	0.00665	0.00779	0.9856	0.9744 (± 0.08786)
p-SFE (5)	59.01	0.0038	0.00361	0.9926	0.9665 (± 0.08581)
p-SFE (6)	69.6	0.00171	0.00323	0.9951	0.9615 (± 0.06498)
p-SFE (7)	82.21	0.00057	0.00114	0.9983	0.9615 (± 0.05252)
p-SFE (8)	93.25	0	0.00076	0.9992	0.9598 (± 0.04825)
p-SFE (9)	107.2	0	0.00019	0.9998	0.9615 (± 0.04308)
p-SFE (10)	116.8	0	0	1	0.9636 (± 0.0403)

B.2 Encoding efficiency

Table B.11: Supplementary data to the results of *encoding efficiency* for the encoding algorithms from the *raw* audio of the *Free Spoken MNIST* dataset, see table 6.11. The parentheses behind the population encoding algorithms show the number of thresholds used, while the parenthesis behind the mean values are the stnadard deviation.

	T_{exe}	P_{noSp}	P_{posSp}	P_{mixSp}	ratio
BSE	0.3602	0.61	0.39	0	NaN
LE	0.05474	0.12	0.88	0	NaN
TE	0.1429	0	0.285	0.715	1.061 (± 0.1367)
TCE	0.125	0.26	0	0.74	0.9998 (± 0.003031)
MWE	0.1408	0.295	0.015	0.69	0.847 (± 0.4238)
SFE	0.1042	0.235	0	0.765	0.9997 (± 0.003334)
p-TE (2)	0.1657	0	0.285	0.715	1.046 (± 0.09876)
p-SFE (2)	0.2071	0.255	0	0.745	1.013 (± 0.05057)
p-SFE (3)	0.1984	0.045	0	0.955	1.018 (± 0.06553)
p-SFE (4)	0.2618	0	0	1	1.027 (± 0.06773)
p-SFE (5)	0.3187	0	0	1	1.026 (± 0.06604)
p-SFE (6)	0.3882	0	0	1	1.025 (± 0.06433)
p-SFE (7)	0.4569	0	0	1	1.009 (± 0.042)
p-SFE (8)	0.5153	0	0	1	1.012 (± 0.03861)
p-SFE (9)	0.5904	0	0	1	1.015 (± 0.05292)
p-SFE (10)	0.6463	0	0	1	1.02 (± 0.05467)

Table B.12: Supplementary data to the results of *encoding efficiency* for the encoding algorithms from the *normalized* audio of the *Free Spoken MNIST* dataset, see table 6.12. The parentheses behind the population encoding algorithms show the number of thresholds used, while the parenthesis behind the mean values are the stnadard deviation.

	T$_{exe}$	P$_{noSp}$	P$_{posSp}$	P$_{mixSp}$	ratio
BSE	0.3875	0.385	0.615	0	NaN
LE	0.05309	0.015	0.985	0	NaN
TE	0.1415	0	0.01	0.99	1.124 (± 0.1605)
TCE	0.1266	0.025	0	0.975	1 (± 0)
MWE	0.1636	0.745	0.095	0.16	0.1145 (± 0.5329)
SFE	0.1092	0	0	1	1 (± 0)
p-TE (2)	0.1672	0	0.01	0.99	1.221 (± 0.2465)
p-SFE (2)	0.2043	0	0	1	1 (± 0)
p-SFE (3)	0.2049	0	0	1	1.023 (± 0.0561)
p-SFE (4)	0.2615	0	0	1	1.022 (± 0.05626)
p-SFE (5)	0.3356	0	0	1	1.019 (± 0.05297)
p-SFE (6)	0.394	0	0	1	1.018 (± 0.05462)
p-SFE (7)	0.4538	0	0	1	1.017 (± 0.05342)
p-SFE (8)	0.5263	0	0	1	1.016 (± 0.0527)
p-SFE (9)	0.5879	0	0	1	1.015 (± 0.05149)
p-SFE (10)	0.6572	0	0	1	1.015 (± 0.05072)

Table B.13: Supplementary data to the results of *encoding efficiency* for the encoding algorithms from the *low filtered* audio of the *Free Spoken MNIST* dataset, see table 6.13. The parentheses behind the population encoding algorithms show the number of thresholds used, while the parenthesis behind the mean values are the stnadard deviation.

	T_{exe}	P_{noSp}	P_{posSp}	P_{mixSp}	ratio
BSE	0.4023	0.29	0.71	0	NaN
LE	0.05161	0.09	0.91	0	NaN
TE	0.1424	0.02	0	0.98	1 (± 0)
TCE	0.1309	0.225	0.06	0.715	1.055 (± 0.2088)
MWE	0.1586	0.64	0.325	0.035	23.2 (± 22.03)
SFE	0.1142	0.265	0.58	0.155	1.451 (± 0.3567)
p-TE (2)	0.1613	0	0.02	0.98	1.095 (± 0.1598)
p-TE (3)	0.1394	0	0.02	0.98	1.095 (± 0.1599)
p-TE (4)	0.1862	0	0.02	0.98	1.095 (± 0.1599)
p-SFE (2)	0.2002	0.255	0.585	0.16	1.446 (± 0.3563)
p-SFE (3)	0.1939	0.115	0.525	0.36	1.359 (± 0.331)
p-SFE (4)	0.2639	0	0	1	0.9834 (± 0.07291)
p-SFE (5)	0.3242	0	0	1	0.9808 (± 0.06819)
p-SFE (6)	0.3922	0	0	1	0.9825 (± 0.06782)
p-SFE (7)	0.4593	0	0	1	0.984 (± 0.06742)
p-SFE (8)	0.5279	0	0	1	0.9852 (± 0.06039)
p-SFE (9)	0.5893	0	0	1	0.9858 (± 0.05853)
p-SFE (10)	0.6646	0	0	1	0.9858 (± 0.05717)

Table B.14: Supplementary data to the results of *encoding efficiency* for the encoding algorithms from the *mid filtered* audio of the *Free Spoken MNIST* dataset, see table 6.14. The parentheses behind the population encoding algorithms show the number of thresholds used, while the parenthesis behind the mean values are the stnadard deviation.

	T_{exe}	P_{noSp}	P_{posSp}	P_{mixSp}	ratio
BSE	0.4904	0.17	0.83	0	NaN
LE	0.0515	0	1	0	NaN
TE	0.1436	0.065	0	0.935	1 (± 0)
TCE	0.1186	0.13	0.11	0.76	1.038 (± 0.1816)
MWE	0.1402	0.465	0.375	0.16	16.42 (± 15.43)
SFE	0.1088	0.155	0.67	0.175	1.571 (± 0.3723)
p-TE (2)	0.1599	0	0.075	0.925	1.186 (± 0.2226)
p-TE (3)	0.14	0	0.075	0.925	1.186 (± 0.2226)
p-SFE (2)	0.2051	0.165	0.675	0.16	1.58 (± 0.3712)
p-SFE (3)	0.1973	0.1	0.64	0.26	1.413 (± 0.3054)
p-SFE (4)	0.2643	0	0.1	0.9	1.18 (± 0.2285)
p-SFE (5)	0.3236	0	0	1	0.9698 (± 0.05991)
p-SFE (6)	0.3941	0	0	1	0.9684 (± 0.05857)
p-SFE (7)	0.4494	0	0	1	0.97 (± 0.05766)
p-SFE (8)	0.5252	0	0	1	0.9707 (± 0.05604)
p-SFE (9)	0.581	0	0	1	0.9701 (± 0.05716)
p-SFE (10)	0.6563	0	0	1	0.9692 (± 0.05575)

Table B.15: Supplementary data to the results of *encoding efficiency* for the encoding algorithms from the *high filtered* audio of the *Free Spoken MNIST* dataset, see table 6.15. The parentheses behind the population encoding algorithms show the number of thresholds used, while the parenthesis behind the mean values are the stnadard deviation.

	T_{exe}	P_{noSp}	P_{posSp}	P_{mixSp}	ratio
BSE	0.4056	0.565	0.435	0	NaN
LE	0.06569	0.62	0.38	0	NaN
TE	0.1395	0.31	0	0.69	1.002 (± 0.02128)
TCE	0.1241	0.38	0.045	0.575	1.031 (± 0.1376)
MWE	0.162	0.405	0.175	0.42	4.726 (± 5.857)
SFE	0.1081	0.27	0.28	0.45	1.212 (± 0.3145)
p-TE (2)	0.1604	0	0.495	0.505	1.276 (± 0.3226)
p-TE (3)	0.1377	0	0.495	0.505	1.274 (± 0.3239)
p-TE (4)	0.1816	0	0.495	0.505	1.274 (± 0.324)
p-TE (5)	0.2271	0	0.495	0.505	1.274 (± 0.324)
p-TE (6)	0.2694	0	0.495	0.505	1.274 (± 0.324)
p-TE (7)	0.3115	0	0.495	0.505	1.274 (± 0.324)
p-TE (8)	0.3618	0	0.495	0.505	1.274 (± 0.3241)
p-TE (9)	0.3991	0	0.495	0.505	1.274 (± 0.3241)
p-TE (10)	0.4426	0	0.495	0.505	1.271 (± 0.3258)
p-SFE (2)	0.2262	0.255	0.265	0.48	1.225 (± 0.387)
p-SFE (3)	0.2046	0.24	0.27	0.49	1.206 (± 0.3811)
p-SFE (4)	0.2647	0.27	0.28	0.45	1.202 (± 0.384)
p-SFE (5)	0.3248	0.24	0.27	0.49	1.205 (± 0.3817)
p-SFE (6)	0.3891	0.23	0.26	0.51	1.146 (± 0.3025)
p-SFE (7)	0.453	0.23	0.26	0.51	1.193 (± 0.3506)
p-SFE (8)	0.5188	0	0	1	0.9481 (± 0.05407)
p-SFE (9)	0.5828	0	0	1	0.952 (± 0.05362)
p-SFE (10)	0.6581	0	0	1	0.9556 (± 0.05102)

Table B.16: Supplementary data to the results of *encoding efficiency* for the encoding algorithms from the *raw* audio of the *Speech Commands* dataset, see table 6.16. The parentheses behind the population encoding algorithms show the number of thresholds used, while the parenthesis behind the mean values are the stnadard deviation.

	T_{exe}	P_{noSp}	P_{posSp}	P_{mixSp}	ratio
BSE	54.32	0.9514	0.04864	0	NaN
LE	3.131	0.1313	0.8687	0	NaN
TE	9.937	0.3454	0	0.6546	1 (± 0.001603)
TCE	8.091	0.156	0	0.844	1 (± 0.004058)
MWE	7.638	0.8136	0.03895	0.1474	1.436 (± 3.622)
SFE	7.657	0.1898	0	0.8102	1 (± 0.005286)
p-TE (2)	13.47	0	0.3221	0.6779	1.092 (± 0.1844)
p-TE (3)	19.89	0	0.3221	0.6779	1.092 (± 0.1844)
p-SFE (2)	18.25	0	0.00038	0.9996	1 (± 0.01179)
p-SFE (3)	27.13	0	0.00038	0.9996	1.001 (± 0.02615)
p-SFE (4)	35.25	0	0.00038	0.9996	1.002 (± 0.0408)
p-SFE (5)	43.05	0	0.00038	0.9996	1.003 (± 0.05282)
p-SFE (6)	51.03	0	0.00019	0.9998	1.003 (± 0.05996)
p-SFE (7)	59.23	0	0.00019	0.9998	1.003 (± 0.05613)
p-SFE (8)	68.69	0	0	1	1.003 (± 0.05322)
p-SFE (9)	77.35	0	0	1	1.003 (± 0.05077)
p-SFE (10)	84.5	0	0	1	1.003 (± 0.04895)

Table B.17: Supplementary data to the results of *encoding efficiency* for the encoding algorithms from the *normalized* audio of the *Speech Commands* dataset, see table 6.17. The parentheses behind the population encoding algorithms show the number of thresholds used, while the parenthesis behind the mean values are the stnadard deviation.

	T_{exe}	P_{noSp}	P_{posSp}	P_{mixSp}	ratio
BSE	39.03	0.0532	0.9468	0	NaN
LE	3.035	0.02204	0.978	0	NaN
TE	9.964	0.06289	0.00019	0.9369	1 (± 0.01487)
TCE	8.129	0.01767	0	0.9823	0.9995 (± 0.02809)
MWE	7.635	0.7823	0.07486	0.1429	0.7822 (± 1.65)
SFE	7.685	0	0.00038	0.9996	1 (± 0.009856)
p-TE (2)	13.62	0	0.04123	0.9588	1.102 (± 0.1663)
p-TE (3)	20.26	0	0.04104	0.959	1.099 (± 0.1621)
p-SFE (2)	24.66	0	0.00038	0.9996	1 (± 0.02995)
p-SFE (3)	36.36	0	0.00019	0.9998	1.001 (± 0.03729)
p-SFE (4)	47.69	0	0	1	1.003 (± 0.0516)
p-SFE (5)	58.99	0	0	1	1.003 (± 0.05406)
p-SFE (6)	71.58	0	0	1	1.003 (± 0.05337)
p-SFE (7)	82.99	0	0	1	1.003 (± 0.05238)
p-SFE (8)	93.29	0	0	1	1.003 (± 0.05089)
p-SFE (9)	106.3	0	0	1	1.003 (± 0.0488)
p-SFE (10)	118.6	0	0	1	1.002 (± 0.04635)

Table B.18: Supplementary data to the results of *encoding efficiency* for the encoding algorithms from the *low filtered* audio of the *Speech Commands* dataset, see table 6.18. The parentheses behind the population encoding algorithms show the number of thresholds used, while the parenthesis behind the mean values are the stnadard deviation.

	T_{exe}	P_{noSp}	P_{posSp}	P_{mixSp}	ratio
BSE	57.03	1	0	0	NaN
LE	5.277	0.1642	0.8358	0	NaN
TE	10.96	0.1414	0.00019	0.8584	1.001 (± 0.01585)
TCE	8.724	0.00304	0.00209	0.9949	1.003 (± 0.05159)
MWE	8.426	0.7374	0.2607	0.0019	0 (± 0)
SFE	8.394	0.2459	0.5605	0.1936	1.45 (± 0.3603)
p-TE (2)	13.6	0	0.1416	0.8584	1.123 (± 0.1886)
p-TE (3)	21.19	0	0.1414	0.8586	1.123 (± 0.1864)
p-TE (4)	26.46	0	0.1414	0.8586	1.123 (± 0.1864)
p-SFE (2)	24.4	0.2915	0.5599	0.1486	1.485 (± 0.3537)
p-SFE (3)	37.35	0.1668	0.4782	0.3549	1.404 (± 0.3561)
p-SFE (4)	47.52	0.05795	0.1991	0.7429	1.186 (± 0.2477)
p-SFE (5)	61.64	0	0	1	0.9788 (± 0.0604)
p-SFE (6)	70.39	0	0	1	0.9789 (± 0.06189)
p-SFE (7)	82.2	0	0	1	0.9795 (± 0.06183)
p-SFE (8)	93.98	0	0	1	0.9797 (± 0.06177)
p-SFE (9)	106.8	0	0	1	0.9809 (± 0.06052)
p-SFE (10)	117	0	0	1	0.9818 (± 0.05933)

Table B.19: Supplementary data to the results of *encoding efficiency* for the encoding algorithms from the *mid filtered* audio of the *Speech Commands* dataset, see table 6.19. The parentheses behind the population encoding algorithms show the number of thresholds used, while the parenthesis behind the mean values are the stnadard deviation.

	T_{exe}	P_{noSp}	P_{posSp}	P_{mixSp}	ratio
BSE	58	0.1514	0.8486	0	NaN
LE	5.612	0.00038	0.9996	0	NaN
TE	11.22	0.02033	0.00019	0.9795	1 (± 0.007349)
TCE	9.07	0.07638	0.2457	0.6779	1.025 (± 0.1581)
MWE	8.544	0.8573	0.1423	0.00038	0 (± 0)
SFE	8.307	0.09253	0.8655	0.04199	1.579 (± 0.4256)
p-TE (2)	13.66	0	0.02052	0.9795	1.11 (± 0.1435)
p-TE (3)	21.13	0	0.02052	0.9795	1.109 (± 0.1431)
p-TE (4)	26.74	0	0.02052	0.9795	1.109 (± 0.1431)
p-SFE (2)	24.12	0.09253	0.8655	0.04199	1.574 (± 0.4264)
p-SFE (3)	35.15	0.08303	0.87	0.04693	1.572 (± 0.417)
p-SFE (4)	47.01	0.00969	0.2451	0.7452	1.374 (± 0.332)
p-SFE (5)	59.07	0	0	1	0.9744 (± 0.05279)
p-SFE (6)	69.21	0	0	1	0.975 (± 0.05293)
p-SFE (7)	82.03	0	0	1	0.9751 (± 0.05402)
p-SFE (8)	92.93	0	0	1	0.9762 (± 0.05273)
p-SFE (9)	104.3	0	0	1	0.9772 (± 0.0519)
p-SFE (10)	120	0	0	1	0.9784 (± 0.05036)

Table B.20: Supplementary data to the results of *encoding efficiency* for the encoding algorithms from the *high filtered* audio of the *Speech Commands* dataset, see table 6.20. The parentheses behind the population encoding algorithms show the number of thresholds used, while the parenthesis behind the mean values are the stnadard deviation.

	T_{exe}	P_{noSp}	P_{posSp}	P_{mixSp}	ratio
BSE	53.2	1	0	0	NaN
LE	5.473	0.708	0.292	0	NaN
TE	11.17	0.4178	0.00019	0.582	1.001 (± 0.02126)
TCE	8.64	0.3789	0.04826	0.5729	1.051 (± 0.1917)
MWE	8.237	0.4144	0.1708	0.4148	14.29 (± 25.08)
SFE	8.094	0.4102	0.1524	0.4374	1.18 (± 0.2902)
p-TE (2)	13.75	0	0.674	0.326	1.26 (± 0.3376)
p-TE (3)	21.45	0	0.6738	0.3262	1.257 (± 0.3382)
p-TE (4)	26.89	0	0.6738	0.3262	1.257 (± 0.3382)
p-TE (5)	31.55	0	0.6738	0.3262	1.257 (± 0.3382)
p-TE (6)	36.45	0	0.6738	0.3262	1.257 (± 0.3382)
p-SFE (2)	24.08	0.4102	0.1524	0.4374	1.173 (± 0.2935)
p-SFE (3)	36.83	0.3768	0.1503	0.4729	1.165 (± 0.2929)
p-SFE (4)	48.31	0.369	0.1457	0.4853	1.168 (± 0.3011)
p-SFE (5)	60.27	0.3768	0.1503	0.4729	1.164 (± 0.2944)
p-SFE (6)	73.89	0.4465	0.1533	0.4002	1.178 (± 0.3046)
p-SFE (7)	82.34	0.3768	0.1503	0.4729	1.164 (± 0.2946)
p-SFE (8)	93.82	0.00019	0.00114	0.9987	0.9588 (± 0.05062)
p-SFE (9)	108	0	0.00019	0.9998	0.9614 (± 0.04353)
p-SFE (10)	122.1	0	0	1	0.9635 (± 0.04049)

C Classification accuracy additional results

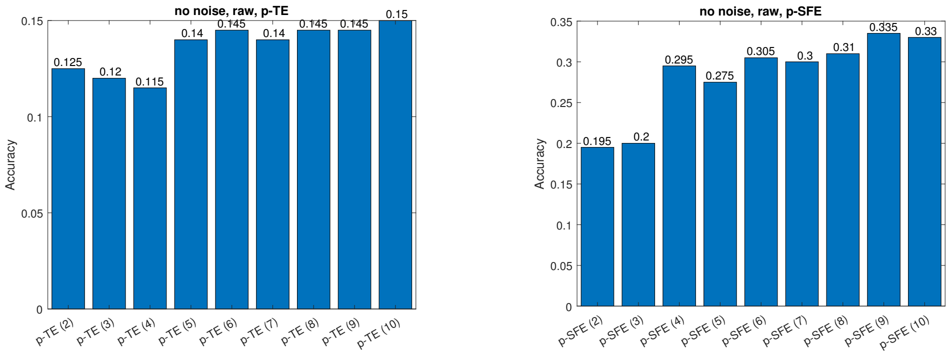

Figure C.1: The classification accuracy achieved with an SVM on the raw audio of the validation set from the FS MNIST dataset for the different number of thresholds of the p-TE and the p-SFE.

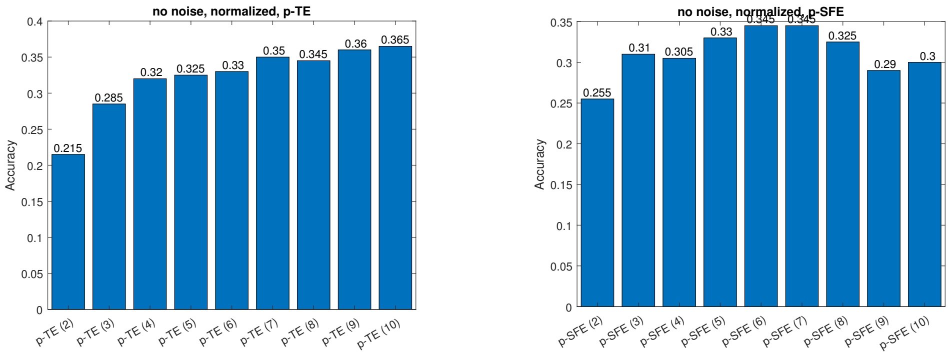

Figure C.2: The classification accuracy achieved with an SVM on the normalized audio of the validation set from the FS MNIST dataset for the different number of thresholds of the p-TE and the p-SFE.

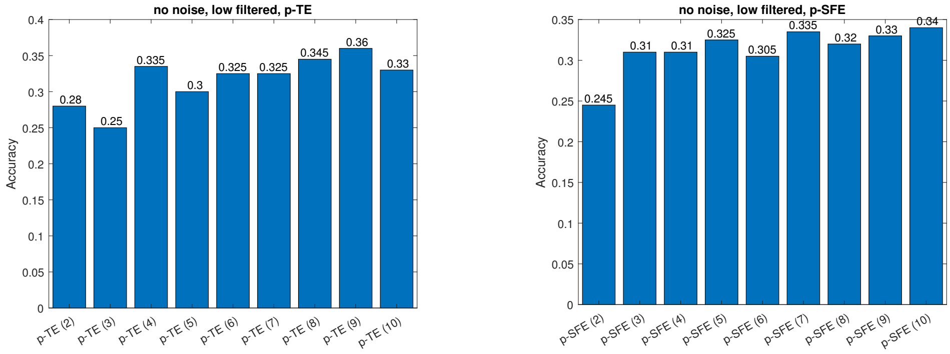

Figure C.3: The classification accuracy achieved with an SVM on the low filtered signal of the validation set from the FS MNIST dataset for the different number of thresholds of the p-TE and the p-SFE.

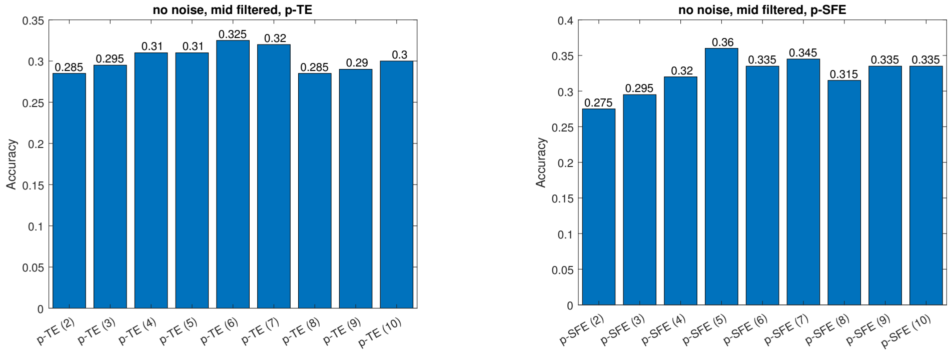

Figure C.4: The classification accuracy achieved with an SVM on the mid filtered signal of the validation set from the FS MNIST dataset for the different number of thresholds of the p-TE and the p-SFE.

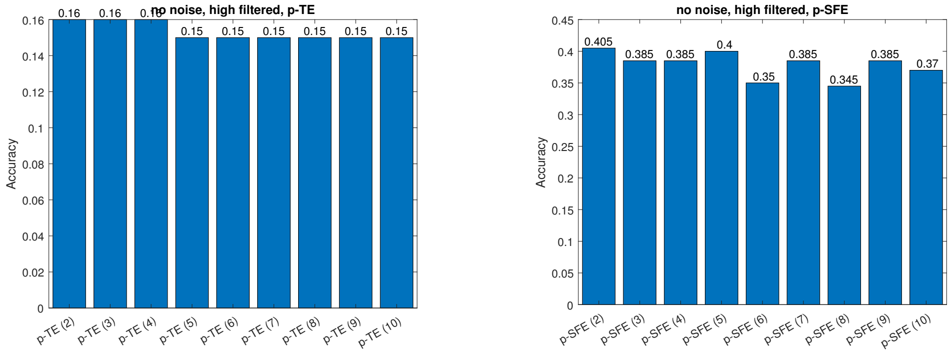

Figure C.5: The classification accuracy achieved with an SVM on the high filtered signal of the validation set from the FS MNIST dataset for the different number of thresholds of the p-TE and the p-SFE.

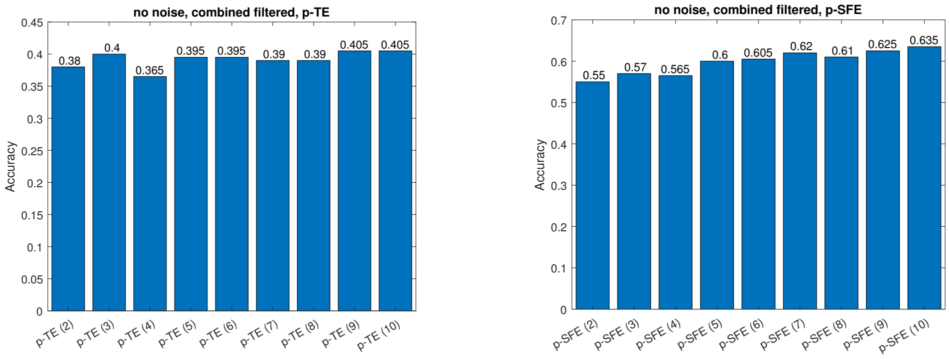

Figure C.6: The classification accuracy achieved with an SVM on the combined spike trains of the filtered signal of the validation set from the FS MNIST dataset for the different number of thresholds of the p-TE and the p-SFE.

D Hardware implementation proposal

State descriptions:
State = 000; wait for input
State = 001; buffer := difference [buffer + input]
State = 010; buffer := abs(difference) [invert buffer]
State = 011; check for spikes and update the difference if needed
State = 100; reconstruct difference sign [invert buffer]
State = 101; reconstruct encoded signal [buffer + input]
State = 110; preprocess for next loop [invert buffer]
State = 111; unused, should not be reached

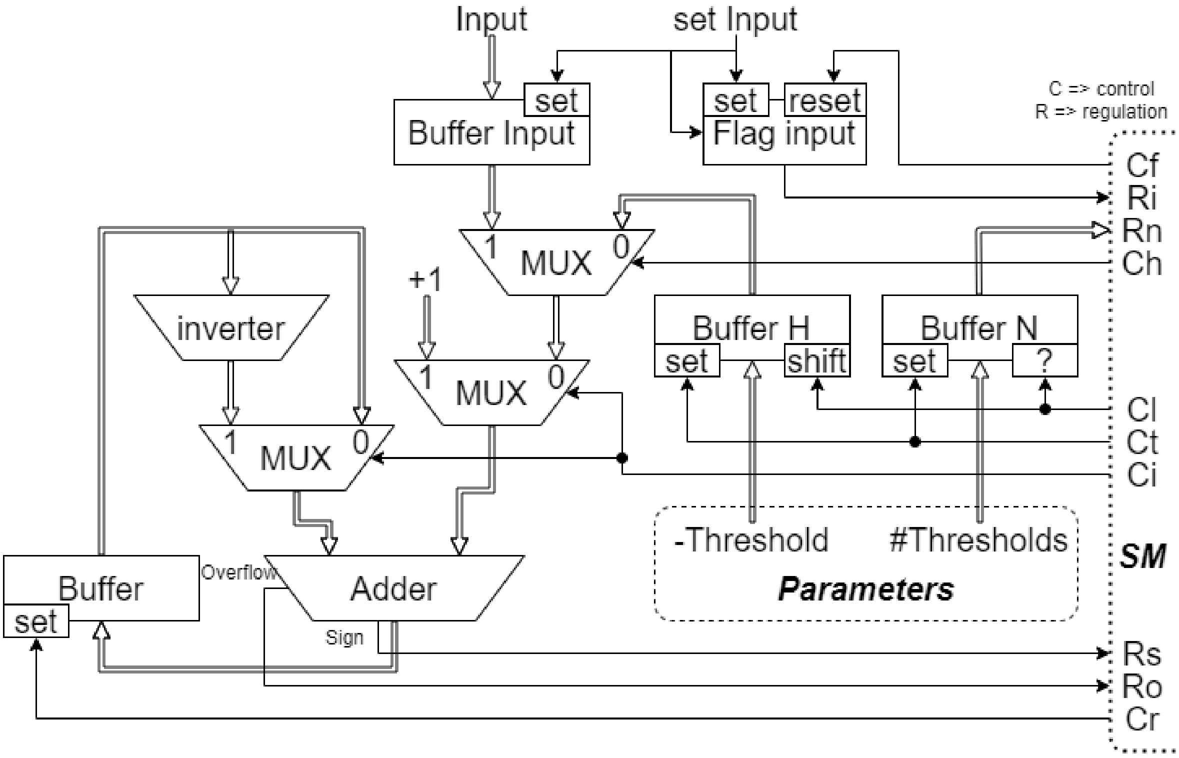

Figure D.1: Block diagram schematic of the computation part from the proposed simplified p-SFE hardware implementation.

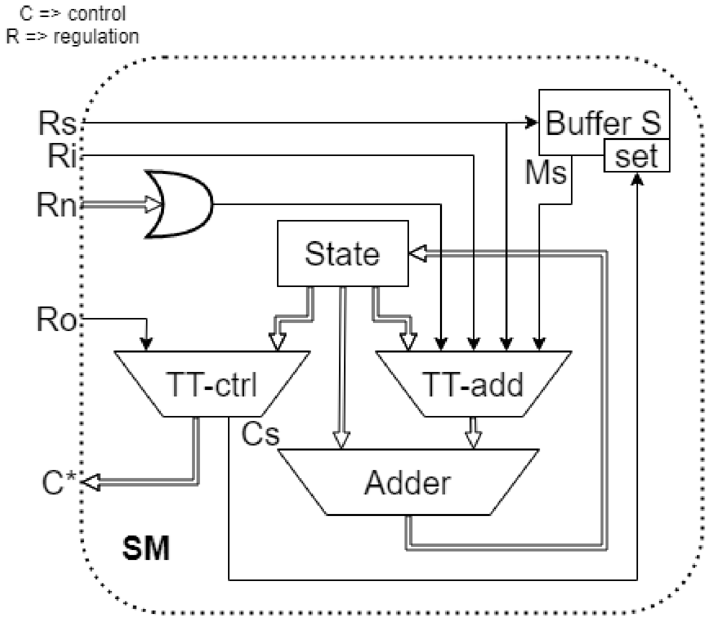

Figure D.2: Block diagram schematic of the controller part from the proposed simplified p-SFE hardware implementation.

Table D.1: Truth-table for the control from the state, TT-ctrl. This truth-table should be implemented in the "TT-ctrl sub-component of the controller, see figure D.2. The bold control bit on the other hand are essential to the function of the proposed hardware implementation. The italic control bits don't have an impact on the function of the implementation and can be chosen arbitrary, but have been set to these values so as little logic gates would be needed. Though, they might need to be changed for adding stability into controller, e.g. a bit flip would not impact the result of the encoding.

S2	S1	S0	Ci	Cs	Ct	Ch	Cl	Cr	Cf
0	0	0	*0*	*0*	*0*	*1*	*1*	**0**	**0**
0	0	1	**0**	**1**	**1**	**0**	**0**	**1**	**1**
0	1	0	**1**	**1**	**1**	*0*	**0**	**1**	**0**
0	1	1	**0**	**0**	**0**	**1**	**1**	**Ro**	**0**
1	0	0	**1**	*1*	*1*	*0*	*0*	**1**	**0**
1	0	1	**0**	*1*	*1*	**0**	*0*	**1**	**0**
1	1	0	**1**	*1*	*1*	*0*	*0*	**1**	**0**
1	1	1	**0**	*0*	*0*	*1*	*1*	**0**	**0**

Table D.2: Truth-table for the addition to the state, TT-add. This truth-table should be implemented in the "TT-add" sub-component of the controller, see figure D.2.

S2	S1	S0	Ri	Rs	Rn	Ms	A1	A0
0	0	0	0	*	*	*	0	0
0	0	0	1	*	*	*	0	1
0	0	1	*	0	*	*	1	0
0	0	1	*	1	*	*	0	1
0	1	0	*	*	*	*	0	1
0	1	1	0	*	0	*	0	0
0	1	1	*	*	1	0	1	0
0	1	1	1	*	*	0	1	0
0	1	1	*	*	1	1	0	1
0	1	1	1	*	*	1	0	1
1	0	0	*	*	*	*	0	1
1	0	1	*	*	*	*	0	1
1	1	0	*	*	*	*	1	0
1	1	1	*	*	*	*	0	1

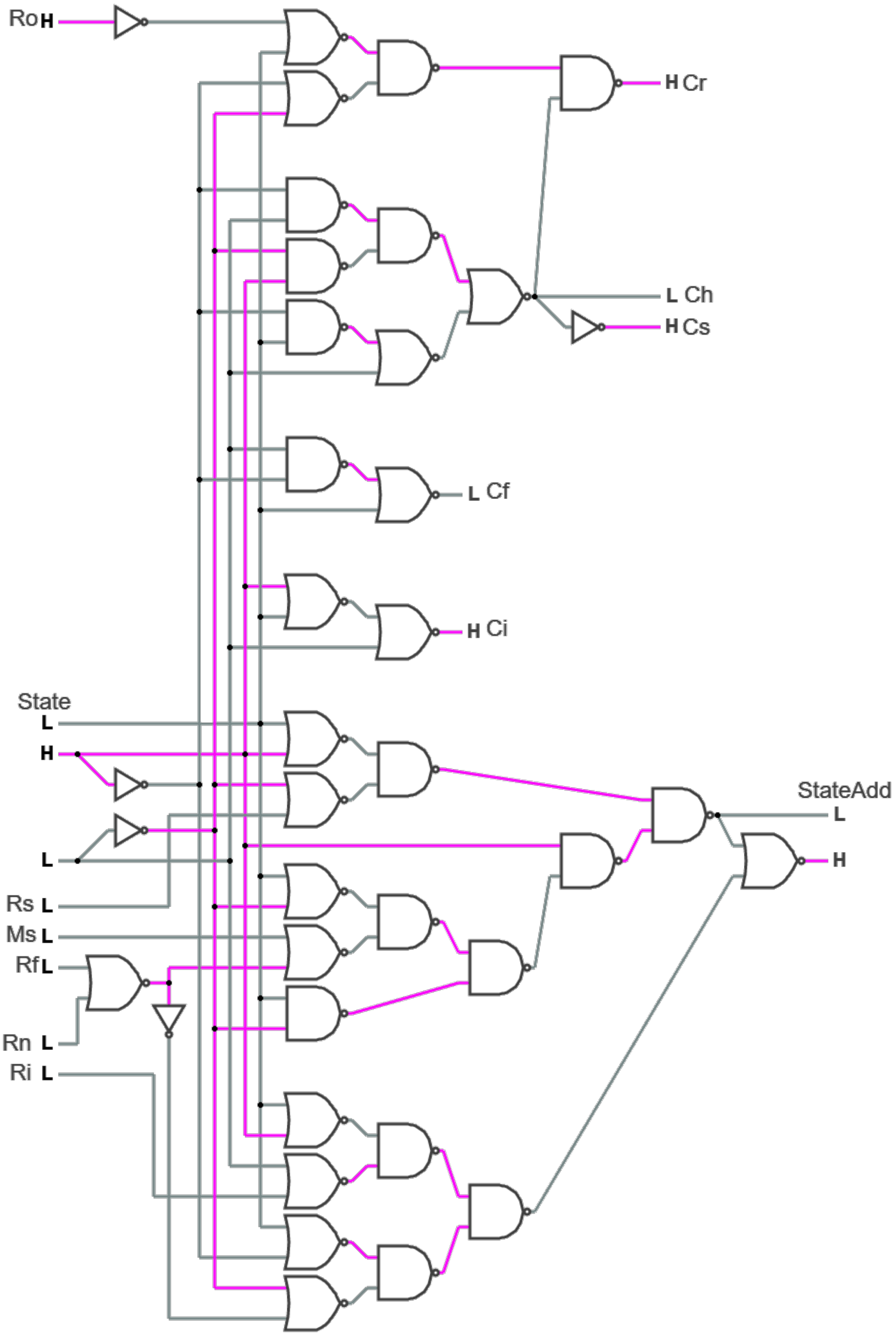

Figure D.3: Logical gates to implement the truth-tables for both TT-ctrl and TT-add, as defined in tables D.1 and D.2 respectively.

Bibliography

[1] Facts about speech intelligibility: human voice frequency range. URL https://www.dpamicrophones.com/mic-university/facts-about-speech-intelligibility.

[2] Mobility, public transport and road safety: Self-driving vehicles, 2016. URL https://www.government.nl/topics/mobility-public-transport-and-road-safety/self-driving-vehicles.

[3] Ahmed A. Abusnaina and Rosni Abdullah. Spiking Neuron Models: A Review. International Journal of Digital Content Technology and its Applications, 8(3):14–21, 2014. URL https://www.researchgate.net/publication/317579637{_}Spiking{_}Neuron{_}Models{_}A{_}Review.

[4] E. D. Adrian and Yngve Zotterman. The impulses produced by sensory nerve-endings: Part II. The response of a Single End-Organ. The Journal of Physiology, 61(2):151–171, 4 1926. ISSN 14697793. doi: 10.1113/jphysiol.1926.sp002281. URL https://physoc-onlinelibrary-wiley-com.tudelft.idm.oclc.org/doi/full/10.1113/jphysiol.1926.sp002281https://physoc-onlinelibrary-wiley-com.tudelft.idm.oclc.org/doi/abs/10.1113/jphysiol.1926.sp002281https://physoc-onlinelibrary-wiley-com.tudelft.idm.oclc.org/doi/10.1113/jphysiol.1926.sp002281.

[5] Sander M. Bohte, Joost N. Kok, and Han La Poutré. Error-backpropagation in temporally encoded networks of spiking neurons, 10 2002. ISSN 09252312.

[6] Andreas Ch Braun, Uwe Weidner, and Stefan Hinz. Classification in high-dimensional feature spaces-assessment using SVM, IVM and RVM with focus on simulated EnMAP data. IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing, 5(2):436–443, 2012. ISSN 19391404. doi: 10.1109/JSTARS.2012.2190266.

[7] César Bravo, Luigi Saputelli, Francklin Rivas, Anna Gabriela Pérez, Michael Nikolaou, Georg Zangl, Neil De Guzmán, Shahab Mohaghegh, and Gustavo Nunez. State of the art of artificial intelligence and predictive analytics in the E&P industry: A technology survey. SPE Journal, 19(4):547–563, aug 2014. ISSN 1086055X. doi: 10.2118/150314-pa. URL http://onepetro.org/SJ/article-pdf/19/04/547/2099035/spe-150314-pa.pdf.

[8] Romain Brette. Philosophy of the Spike: Rate-Based vs. Spike-Based Theories of the Brain. Frontiers in Systems Neuroscience, 9:151, 11 2015. ISSN 1662-5137. doi: 10.3389/fnsys.2015.00151. URL http://journal.frontiersin.org/Article/10.3389/fnsys.2015.00151/abstract.

[9] Eliya Elon. AI Problem Types and Their Definitions, 2018. URL https://www.razor-labs.com/ai-problem-types-and-their-definitions/.

[10] Covington Paul;Adams Jay;Sargin Emre. Deep Neural Networks for YouTube Recommendations. In Proceedings of the 10th ACM Conference on Recommender Systems, New York, New York, USA, 2016.

[11] Răzvan V. Florian. The chronotron: A neuron that learns to fire temporally precise spike patterns. PLoS ONE, 7(8):40233, 8 2012. ISSN 19326203. doi: 10.1371/journal.pone.0040233. URL www.plosone.org.

[12] Francesco Galluppi and Steve Furber. Representing and decoding rank order codes using polychronization in a network of spiking neurons. In Proceedings of the International Joint Conference on Neural Networks, pages 943–950, 2011. ISBN 9781457710865. doi: 10.1109/IJCNN.2011.6033324.

[13] Andrey V. Gavrilov and Konstantin O. Panchenko. Methods of learning for spiking neural networks. A survey. In 2016 13th International Scientific-Technical Conference on Actual Problems of Electronic Instrument Engineering, APEIE 2016 - Proceedings, volume 2, pages 455–460. Institute of Electrical and Electronics Engineers Inc., 7 2016. ISBN 9781509040698. doi: 10.1109/APEIE.2016.7806372.

[14] Samanwoy Ghosh-Dastidar and Hojjat Adeli. Third Generation Neural Networks: Spiking Neural Networks. In Advances in Intelligent and Soft Computing, volume 61 AISC, pages 167–178. Springer, Berlin, Heidelberg, 2009. ISBN 9783642031557. doi: 10.1007/978-3-642-03156-4_17. URL http://link.springer.com/10.1007/978-3-642-03156-4{_}17.

[15] Brian R. Glasberg and Brian C.J. Moore. Derivation of auditory filter shapes from notched-noise data. Hearing Research, 47(1-2):103–138, aug 1990. ISSN 03785955. doi: 10.1016/0378-5955(90)90170-T.

[16] Robert Gütig and Haim Sompolinsky. Tempotron Learning. In Encyclopedia of Computational Neuroscience, pages 1–3. Springer New York, 2014. doi: 10.1007/978-1-4614-7320-6_685-1. URL https://link-springer-com.tudelft.idm.oclc.org/referenceworkentry/10.1007/978-1-4614-7320-6{_}685-1.

[17] A. L. Hodgkin and B. Katz. The effect of sodium ions on the electrical activity of the giant axon of the squid. The Journal of Physiology, 108(1):37–77, 3 1949. ISSN 14697793. doi: 10.1113/jphysiol.1949.sp004310. URL https://physoc-onlinelibrary-wiley-com.tudelft.idm.oclc.org/doi/full/10.1113/jphysiol.1949.sp004310https://physoc-onlinelibrary-wiley-com.tudelft.idm.oclc.org/doi/abs/10.1113/jphysiol.1949.sp004310https://physoc-onlinelibrary-wiley-com.tudelft.idm.oclc.org/doi/10.1113/jphysiol.1949.sp004310.

[18] Mark Horowitz. 1.1 Computing's energy problem (and what we can do about it). In Digest of Technical Papers - IEEE International Solid-State Circuits Conference, volume 57, pages 10–14, 2014. ISBN 9781479909186. doi: 10.1109/ISSCC.2014.6757323.

[19] Michael Hough, Hugo de Garis, Michael Korkin, Felix Gers, and Norberto Eiji Nawa. SPIKER: Analog waveform to digital spiketrain conversion in ATR's artificial brain (cam-brain) project. International Conference on Robotics and Artificial Life, 1999. URL https://www.researchgate.net/publication/2462190{_}SPIKER{_}Analog{_}Waveform{_}to{_}Digital{_}Spiketrain{_}Conversion{_}

[20] A. F. Huxley and R. Stämpfli. Direct determination of membrane resting potential and action potential in single myelinated nerve fibres. The Journal of Physiology, 112(3-4):476–495, 2 1951. ISSN 00223751. doi: 10.1113/jphysiol.1951.sp004545. URL http://doi.wiley.com/10.1113/jphysiol.1951.sp004545.

[21] A. F. Huxley and R. Stämpfli. Effect of potassium and sodium on resting and action potentials of single myelinated nerve fibres. The Journal of Physiology, 112(3-4):496–508, 2 1951. ISSN 14697793. doi: 10.1113/jphysiol.1951.sp004546. URL https://physoc-onlinelibrary-wiley-com.tudelft.idm.oclc.org/doi/full/10.1113/jphysiol.1951.sp004546https://physoc-onlinelibrary-wiley-com.tudelft.idm.oclc.org/doi/abs/10.1113/jphysiol.1951.sp004546https://physoc-onlinelibrary-wiley-com.tudelft.idm.oclc.

[22] Eugene M. Izhikevich. Which model to use for cortical spiking neurons? IEEE Transactions on Neural Networks, 15(5):1063–1070, 9 2004. ISSN 10459227. doi: 10.1109/TNN.2004.832719.

[23] Zohar Jackson, César Souza, Jason Flaks, Yuxin Pan, Hereman Nicolas, and Adhish Thite. Jakobovski/free-spoken-digit-dataset: v1.0.8. aug 2018. doi: 10.5281/ZENODO.1342401. URL https://zenodo.org/record/1342401.

[24] Nikola Kasabov, Nathan Matthew Scott, Enmei Tu, Stefan Marks, Neelava Sengupta, Elisa Capecci, Muhaini Othman, Maryam Gholami Doborjeh, Norhanifah Murli, Reggio Hartono, Israel Espinosa-Ramos, Lei Zhou, Fahad Bashir Alvi, Grace Wang, Denise Taylor, Valery Feigin, Sergei Gulyaev, Mahmoud Mahmoud, Zeng-Guang Hou, and Jie Yang. Evolving spatio-temporal data machines based on the NeuCube neuromorphic framework: Design methodology and selected applications. Neural Networks, 78:1–14, 2016. doi: 10.1016/j.neunet.2015.09.011. URL http://dx.doi.org/10.1016/j.neunet.2015.09.011.

[25] Nikola K. Kasabov. Time-Space, Spiking Neural Networks and Brain-Inspired Artificial Intellige volume 7 of Springer Series on Bio- and Neurosystems. Springer Berlin Heidelberg, Berlin, Heidelberg, 2019. ISBN 978-3-662-57713-4. doi: 10.1007/978-3-662-57715-8. URL http://link.springer.com/10.1007/978-3-662-57715-8.

[26] Thomas Kreuz, Julie S. Haas, Alice Morelli, Henry D.I. I Abarbanel, and Antonio Politi. Measuring spike train synchrony. Journal of Neuroscience Methods, 165(1):151–161, sep 2007. ISSN 01650270. doi: 10.1016/j.jneumeth.2007.05.031. URL https://linkinghub.elsevier.com/retrieve/pii/S0165027007002671http://inls.ucsd.edu/.

[27] Thomas Kreuz, Daniel Chicharro, Martin Greschner, and Ralph G. Andrzejak. Time-resolved and time-scale adaptive measures of spike train synchrony. Journal of Neuroscience Methods, 195(1):92–106, jan 2011. ISSN 01650270. doi: 10.1016/j.jneumeth.2010.11.020. URL http://neurodatabase.org/,https://linkinghub.elsevier.com/retrieve/pii/S0165027010006564.

[28] Alexander Kugele, Thomas Pfeil, Michael Pfeiffer, and Elisabetta Chicca. Efficient Processing of Spatio-Temporal Data Streams With Spiking Neural Networks. Frontiers in Neuroscience, 14:439, 5 2020. ISSN 1662-453X. doi: 10.3389/fnins.2020.00439. URL https://www.frontiersin.org/article/10.3389/fnins.2020.00439/full.

[29] Shane Legg and Marcus Hutter. A Collection of Definitions of Intelligence. Frontiers in Artificial Intelligence and Applications, 157(1):17–24, 2007. URL http://arxiv.org/abs/0706.3639.

[30] Todd Alexander Litman. Autonomous Vehicle Implementation Predictions: Implications for Transport Planning. Technical report, Victoria Transport Policy Institute, 6 2020. URL www.vtpi.org.

[31] Wolfgang Maass. Networks of spiking neurons: The third generation of neural network models. Neural Networks, 10(9):1659–1671, 12 1997. ISSN 08936080. doi: 10.1016/S0893-6080(97)00011-7. URL https://linkinghub.elsevier.com/retrieve/pii/S0893608097000117.

[32] Bernard Marr. Are Alexa And Siri Considered AI?, 2020. URL https://bernardmarr.com/default.asp?contentID=1830.

[33] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. The Bulletin of Mathematical Biophysics, 5(4):115–133, 12 1943. ISSN 0007-4985. doi: 10.1007/BF02478259. URL http://link.springer.com/10.1007/BF02478259.

[34] Brian C.J. Moore and Brian R. Glasberg. Suggested formulae for calculating auditory-filter bandwidths and excitation patterns. Journal of the Acoustical Society of America, 74(3):750–753, sep 1983. ISSN NA. doi: 10.1121/1.389861. URL http://asa.scitation.org/doi/10.1121/1.389861.

[35] Zihan Pan, Jibin Wu, Malu Zhang, Haizhou Li, and Yansong Chua. Neural Population Coding for Effective Temporal Classification. In 2019 International Joint Conference on Neural Networks (IJCNN), volume 2019-July, pages 1–8. IEEE, 7 2019. ISBN 978-1-7281-1985-4. doi: 10.1109/IJCNN.2019.8851858. URL https://ieeexplore.ieee.org/document/8851858/.

[36] Roy D. Patterson and Mike H. Allerhand. Time-domain modeling of peripheral auditory processing: A modular architecture and a software platforma. Journal of the Acoustical Society of America, 98(4):1890–1894, oct 1995. ISSN NA. doi: 10.1121/1.414456. URL http://asa.scitation.org/doi/10.1121/1.414456.

[37] Balint Petro, Nikola Kasabov, and Rita M. Kiss. Selection and Optimization of Temporal Spike Encoding Methods for Spiking Neural Networks. IEEE Transactions on Neural Networks and Learning Systems, 31(2):358–370, 2 2020. ISSN 2162-237X. doi: 10.1109/TNNLS.2019.2906158. URL https://ieeexplore.ieee.org/document/8689349/.

[38] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. Nature, 323(6088):533–536, 1986. ISSN 00280836. doi: 10.1038/323533a0. URL https://www-nature-com.tudelft.idm.oclc.org/articles/323533a0.

[39] Benjamin Schrauwen and I. Van Campenhout. BSA, a fast and accurate spike train encoding scheme. In Proceedings of the International Joint Conference on Neural Networks, 2003., volume 4, pages 2825–2830. IEEE, 2003. ISBN 0-7803-7898-9. doi: 10.1109/IJCNN.2003.1224019. URL http://ieeexplore.ieee.org/document/1224019/.

[40] S. S. Stevens, J. Volkmann, and E. B. Newman. A Scale for the Measurement of the Psychological Magnitude Pitch. Journal of the Acoustical Society of America, 8(3):185–190, jan 1937. ISSN NA. doi: 10.1121/1.1915893. URL http://asa.scitation.org/doi/10.1121/1.1915893.

[41] Simon Thorpe and Jacques Gautrais. Rank Order Coding. In Computational Neuroscience, pages 113–118. Springer US, Boston, MA, 1998. doi: 10.1007/978-1-4615-4831-7_19. URL http://link.springer.com/10.1007/978-1-4615-4831-7{_}19.

[42] M. C.W. Van Rossum. A novel spike distance. Neural Computation, 13(4):751–763, apr 2001. ISSN 08997667. doi: 10.1162/089976601300014321.

[43] Rufin Van Rullen, Jacques Gautrais, Arnaud Delorme, and Simon Thorpe. Face processing using one spike per neurone. In BioSystems, volume 48, pages 229–239. Elsevier, 11 1998. doi: 10.1016/S0303-2647(98)00070-7. URL https://www.sciencedirect.com/science/article/abs/pii/S0303264798000707.

[44] Jonathan D. Victor and Keith P. Purpura. Metric-space analysis of spike trains: theory, algorithms and application. Network: Computation in Neural Systems, 8(2):127–164, jan 1997. ISSN 0954-898X. doi: 10.1088/0954-898X_8_2_003. URL https://www.tandfonline.com/doi/full/10.1088/0954-898X{_}8{_}2{_}003.

[45] Pete Warden. Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition. arXiv, apr 2018. URL http://arxiv.org/abs/1804.03209.

[46] Darrell Whitley. A genetic algorithm tutorial. Statistics and Computing, 4(2):65–85, jun 1994. ISSN 09603174. doi: 10.1007/BF00175354. URL https://link-springer-com.tudelft.idm.oclc.org/article/10.1007/BF00175354.

[47] Simei Gomes Wysoski, Lubica Benuskova, and Nikola Kasabov. On-line learning with structural adaptation in a network of spiking neurons for visual pattern recognition. In Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), volume 4131 LNCS - I, pages 61–70. Springer Verlag, 2006. ISBN 3540386254. doi: 10.1007/11840817_7. URL http:www.kedri.info.

[48] E. Zwicker and I. E. Terhardt. Analytical expressions for critical-band rate and critical bandwidth as a function of frequency, nov 1980. ISSN NA. URL http://asa.scitation.org/doi/10.1121/1.385079.