Faculty of Engineering Sciences

Heidelberg University

Institute of Computer Engineering (ZITI)

Master's Thesis

Optimized Calibration for Analog Computations Targeting Deep Neural Networks on the Example of BrainScaleS-2

by

Eric Matthias Kern

Supervisor:	Prof. Holger Fröning
Reviewer:	Dr. Johannes Schemmel
Advisors:	Hendrik Borras and Bernhard Klein
Field of Study:	Computer Engineering
Submission Date:	06.10.2023

Acknowledgments

First I would like to express my gratitude to the people at ZITI. I'd like to thank Holger Fröning, not only for his valuable input and guidance but also for his genuine interest in this research. Of course, this gratitude also extends to my advisors, Bernhard and Hendrik, who enabled valuable discussions during this thesis. A special acknowledgment goes to Hendrik, who did an outstanding job as an advisor. I highly appreciated your experience, your ability to pay attention to important details and your motivating and positive nature. Likewise, I thank all of my other colleagues and fellow students at ZITI for the great time together.
Furthermore, I owe my thanks to the Electronic Vision(s) Group, both for providing the special hardware and for answering all of my questions about it.
Finally, I sincerely thank my family and friends for their support during this thesis. A special thanks goes out to my friend Nico, who always motivates me to grow beyond myself and gain a new perspective. My friends James and Daniel also deserve my gratitude for their moral support and the great evenings, during which I learned to appreciate much more than just marjoram. And last but not least I'm deeply grateful for the continuous trust and support of my parents.

Abstract

Machine learning is pervasive today, but as more complex models are developed, their application is becoming increasingly costly. This work explores analog computing as a scalable and energy-efficient alternative to the typically used digital computations.
Leveraging the BrainScaleS-2 system as an analog matrix multiplication accelerator, we optimize for typical imperfections like noise and saturation effects that measurably degrade accuracy. While training with hardware in the loop can partially recover lost accuracy, the gap can usually not be fully bridged, deterring potential users. Our research aims to narrow this gap through calibration parameter adjustments and algorithmic optimizations.
We examine how the translation of operands to the hardware affects training and analyze the effects of calibration parameters to develop an overall better solution that is tuned for neural networks to ultimately enhance their accuracy. As the primary contribution, we find that higher circuit time constants, which lead to a higher average load on the analog amplifiers, allow for decreased amplifier gain at similar output amplitudes while improving noise behavior.
By using this custom calibration together with statically scaled input operands, we achieve approximately 41 % accuracy improvement before retraining and 7 % afterward. Notably, we identify that the varying matrix sizes due to the varying number of neurons between layers require gain adjustments, which can potentially further increase accuracy.
Our work addresses common challenges in the efficient execution of artificial neural networks on BrainScaleS-2, providing a path toward competitive usage. These findings have significant implications for cost-effective and energy-efficient machine learning model applications.

Maschinelles Lernen ist heute allgegenwärtig, doch mit der Entwicklung immer komplexerer Modelle wird ihre Anwendung immer kostspieliger. Diese Arbeit erforscht analoge Berechnungen als skalierbare und energieeffiziente Alternative zu den üblicherweise verwendeten digitalen Berechnungen.
Wir nutzen das BrainScaleS-2-System als Beschleuniger für analoge Matrixmultiplikation und optimieren es bezüglich typischer Unvollkommenheiten wie Rauschen und Sättigungseffekte, die die Genauigkeit messbar beeinträchtigen. Obwohl das Hardware-in-the-Loop Training die verlorene Genauigkeit teilweise wiederherstellen kann, bleiben oft noch kleine Genauigkeitseinbußen, was potenzielle Nutzer abschreckt. Unsere Forschung zielt darauf ab, diese Lücke durch Anpassungen der Kalibrierungsparameter und algorithmische Optimierungen zu verkleinern.
Wir untersuchen, wie sich die Übersetzung von Operanden auf die Hardware auf das Training auswirkt, und analysieren die Auswirkungen von Kalibrierungsparametern, um eine insgesamt bessere Lösung zu entwickeln, die auf neuronale Netze abgestimmt ist, um letztendlich deren Genauigkeit zu verbessern. Als Hauptbeitrag stellen wir fest, dass höhere Zeitkonstanten, die zu einer höheren durchschnittlichen Auslastung der Analogverstärker führen, einen geringeren Verstärkungsfaktor bei ähnlichen Ausgangsamplituden ermöglichen und gleichzeitig das Rauschverhalten verbessern.
Durch die Verwendung dieser angepassten Kalibrierung zusammen mit statisch skalierten Eingangsoperanden erreichen wir eine Verbesserung der Genauigkeit um etwa 41 % vor und 7 % nach dem erneuten Training. Insbesondere stellen wir fest, dass die unterschiedlichen Matrixgrößen aufgrund der variierenden Anzahl von Neuronen zwischen den Schichten Verstärkungsanpassungen erfordern, was die Genauigkeit potenziell weiter erhöhen kann.
Unsere Arbeit befasst sich mit allgemeinen Herausforderungen bei der effizienten Ausführung künstlicher neuronaler Netze auf BrainScaleS-2 und bietet einen Weg zu einer konkurrenzfähigen Nutzung. Unsere Erkenntnisse haben erhebliche Auswirkungen auf die kostengünstige und energieeffiziente Anwendung von Modellen des maschinellen Lernens.

Contents

Chapter 1

Introduction

In the last decades, chip designs for general-purpose and high-performance computing have been predominantly digital. However, a very imminent problem of machine learning has begun to reshape this perspective, challenging the long-standing supremacy of purely digital designs.
As neural networks continue to establish themselves as state-of-the-art tools across a wide range of applications, including computer vision and natural language processing, their model sizes and computational requirements have continued to grow. In the ongoing search for better models, the computational demands of the most advanced ones have reached a point where their training has become incredibly costly, involving significant expenses in terms of energy consumption, time, and hardware resources. This trend of continually expanding models is expected to persist [1]. To bridge this growing disparity, researchers revisited the fundamental approach to computations, leading to a resurgence of interest in analog computations.
Analog computing offers increased energy efficiency, especially at low precision. However, it is accompanied by inherent challenges, such as noise, temporal drift, non-linear behaviors, and saturation effects. Even though calibration routines and intelligent circuit design try to compensate for these imperfections, in practice they cannot be entirely avoided. Consequently, applications utilizing analog accelerators must find strategies to minimize the detrimental effects of these imperfections.
Given that the multiply and accumulate operation (MAC) dominates in these increasingly massive machine learning architectures, there has been a push for the development of analog accelerators optimized specifically for these operations.
One specific implementation is the BrainScaleS-2 system developed by the Electronic Vision(s) Group at the Kirchhoff-Institute for Physics at Heidelberg University in Germany. It is a neuromorphic mixed-signal system with an analog chip at its core, that serves as a research platform for spiking neural networks and as an analog matrix multiplication accelerator at the same time. It provides a sophisticated software interface so that existing software is easily portable and calibration routines, that compensate for many of the analog hardware imperfections.
Previous research has demonstrated that, with the provided calibration, machine learning tasks can achieve high accuracy while maintaining high energy efficiency on this accelerator. However, highest accuracy is attainable only through hardware-in-the-loop training in which the model parameters are optimized in the presence of the hardware's imperfections so that models can to some extent account for them. Although previous studies have indicated that training with hardware in the loop can partially replace aspects of the calibration process, it remains unclear whether a calibration specifically designed for hardware-in-the-loop training can further enhance accuracy.
In this work, we take a critical step towards addressing this question. We aim to provide an updated overview of the peculiarities of the analog matrix multiplication and how calibration parameters as well as input operands affect these. With this analysis, we try to get a deeper understanding of the chip's imperfections so that we can develop strategies to achieve an overall improved operational behavior for the execution of neural networks. To achieve this we have to trade off parameter effects like noise, saturation and gain.

Besides optimizations targeting the calibration, we also investigate algorithmic adaptations primarily focusing on the required quantization scaling. However, we also consider traditional model compression techniques like pruning to attenuate the effects of analog imperfections. To verify the validity of our approaches, we train a small Multilayer Perceptron (MLP) with the mentioned optimizations and compare the achieved accuracy to the default calibration.
The following work is structured in four chapters. We start by recapitulating a few fundamentals of deep learning followed by an introduction to the BrainScaleS-2 system, its functioning principles, and an exploration of the calibrated components involved in the analog matrix multiplication. We conclude the background chapter with a review of related literature.
In the next chapter, we analyze the peculiarities of the analog matrix multiplication and investigate parameters influencing runtime, static variations, noise and saturation.
Subsequently, the following chapter focuses on the optimized execution of artificial neural networks. Here we develop optimization strategies that are afterward validated by assessing the accuracy of a network trained on the analog hardware.
Finally, we wrap up our findings in the concluding chapter with an outlook on future research and a concise summary of the obtained insights.

Chapter 2

Background

At the start of this work, we want to provide some background information regarding the fundamental principles of deep learning, the BSS-2 system, and previous research especially focusing on artificial neural networks on BSS-2.

2.1 Fundamentals

2.1.1 Multilayer Perceptron

The Multilayer Perceptron (MLP) is one of the most fundamental architectures of feedforward Deep Neural Networks (DNNs). It originated from the Perceptron [2] by Frank Rosenblatt in 1958. However, the single-layer version was limited in its modeling capabilities and for the multilayer version there was no feasible training algorithm at that time. Only in 1982, Paul Werbos [3] applied backpropagation as we use it today to train such multilayer networks.
A Multilayer Perceptron, as the name implies, consists of several layers. By concatenating these layers bigger networks so-called Deep Neural Networks can be built, which can approximate arbitrary functions [4], [5]. In MLPs each of these layers consists of a so-called linear (or dense or fully connected) layer followed by an arbitrary non-linear function. Examples of non-linear functions are ReLU, tanh, or sigmoid, but over the years the machine learning community proposed many more. These networks can be trained or fitted to approximate a certain unknown function by using backpropagation: an algorithm that calculates the gradient of each adjustable parameter with respect to a cost function by using the chain rule. The cost function compares the network's output with the reference from the training set and calculates a score of how close the network's output was to the ground truth. The derivative of this score can then be used as a starting point to calculate the gradients of all adjustable parameters by using the aforementioned chain rule.
This cost function is minimized with an algorithm called Gradient Descend (or a derivative of it) in which the negative gradients (obtained with backpropagation) are used to update the parameters so that after some updates these parameters reach values that minimize the cost function. While this algorithm often leads only to a locally optimal result in the global parameter space, in practice the local optima are often sufficiently close to the global optimum and very good results can be achieved.
Machine learning approaches that compare the predicted result with training data and update their parameters based on this comparison are part of a subclass called supervised learning. A central part of the computations in MLPs are matrix multiplications (multiple weighted sums) of the input features with so-called weights. These operations are integral to every linear layer within MLPs, however, the exact implementations can vary. For example the optional addition of a bias vector after the matrix product can be absorbed into the matrix multiplication or executed separately afterwards.

In the context of machine learning, specific terminology is often used to describe matrix multiplications in terms of input operands and their dimensions. In this work, we also make use of this terminology. Figure 2.1 shows some of the most relevant terms used in machine learning that eventually refer to plain matrix multiplication.

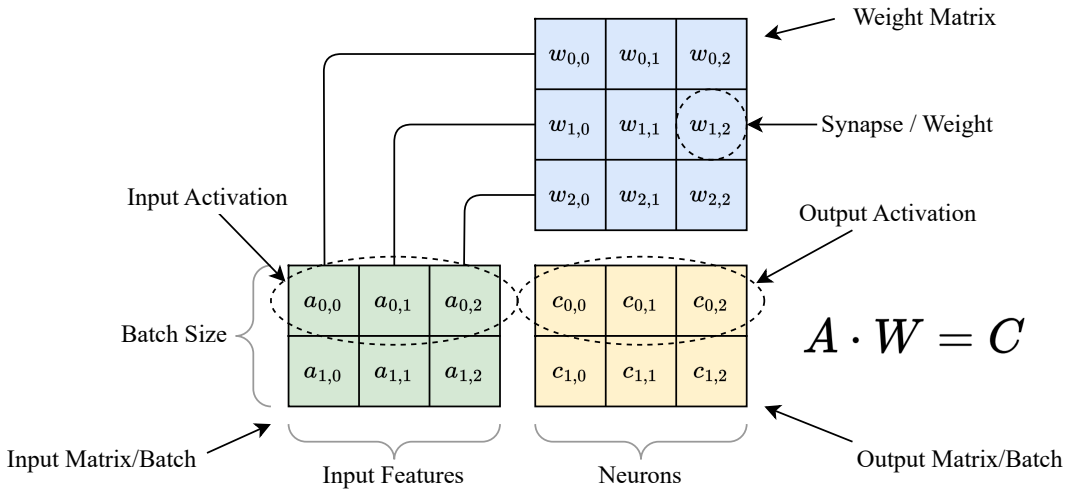

Figure 2.1: Matrix multiplication with the typical nomenclature of machine learning.

The input to a layer of an MLP is called an input activation. Unfortunately, the word *activation* is used quite loosely in the machine learning community and can be used for a single input/output element, a whole vector of these elements or in batched execution for the whole matrix. Throughout this work, we consistently use the word *activation* for an input vector consisting of multiple elements that are called input features. In the first layer of the network, these features can be pixel intensities of an image, amplitude values of an audio sequence, or any other data that contains the relevant information for the desired prediction task.

This input activation or feature vector is then multiplied with a weight matrix. The magnitude of each weight decides the importance of an input feature in the weighted sum (dot product) in each column of the weight matrix. Due to the analogy to a biological neuron, the weights are often referred to as synapses. Since every column of the weight matrix results in a dot product of the input vector with different, independent weights, the width dimension of the matrix corresponds to the number of neurons in that layer. The output vector comprising all dot product results is called the output activation. Its width only depends on the number of neurons in that layer. The output activation undergoes an elementwise non-linear activation function before being passed to the subsequent layer as the input activation. Which operations are applied to the output activation and also their order, before it becomes the input activation of the next layer, depends on the concrete network architecture. As mentioned there can also be the addition of a bias vector and possibly a normalization layer like batch normalization (see Section 2.1.2).

While this is the fundamental view on the computation of a single sample, in practice training requires the processing of thousands of samples. It is quite common to concatenate multiple input vectors to a matrix, a so-called batch of input activations. The batch size represents the number of input vectors and, consequently, the number of output vectors in that batch. This is often done to improve the computation speed because the batched execution sometimes allows a better memory access pattern or as in the case of the BrainScaleS-2 system allows amortization of setup costs. Other typical reasons are better pipelining, more concurrency to better utilize parallel processors but also training-related effects like a better estimate of the gradient for the whole training set.

If we consider batched execution, the three parameters batch size, number of input features, and the number of neurons fully describe all dimensions during the matrix multiplication.

2.1.2 Batch Norm

Batch normalization was introduced by Ioffe and Szegedy in [6] as a fundamental technique to stabilize and accelerate the training of DNNs. During training each layer adapts to the input distribution of its previous layer. But since all layers update their parameters during training also their output distributions change, so that each layer beside the first layer has to adapt to a continuously varying input distribution.
Ioffe and Szegedy call this the *internal covariate shift* and propose batch normalization as a way to reduce this effect. It allows normalization of activations not only at the input of the network but also normalization of internal activations.
As the name implies batch normalization tries to normalize the distribution of each feature in a batch during training. This means each feature has a variance (σ^2) of 1 and a mean (μ) of 0 along the batch dimension.
If we want to apply batch normalization to the output batch C of Figure 2.1 with a batch size i and j elements per output activation, we have to compute:

$$\text{BatchNorm}(c_{i,j}) = \frac{c_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}\gamma_j + \beta_j \tag{2.1}$$

$$\mu_j = \frac{1}{i}\sum_{k=0}^{i-1} c_{k,j} \tag{2.2}$$

$$\sigma_j^2 = \frac{1}{i}\sum_{k=0}^{i-1}(c_{k,j} - \mu_j)^2 \tag{2.3}$$

The parameters γ and β are trainable parameters to restore the modeling capabilities of the network. They offer the network the option to reintroduce a controllable mean and variance, but their utilization is optional. The constant ϵ is for numerical stability to avoid division by 0. For inference, the batch normalization layer keeps exponential moving averages of the mean and variance.
Even though batch normalization is an effective method to improve the training of neural networks, there are still areas where its behavior is not fully understood. For example Santurkar, Tsipras, Ilyas, *et al.* [7] show that the effectiveness of batch normalization is not caused by the reduced internal covariate shift. Instead, they claim that batch normalization smoothens the optimization landscape and therefore improves the learning dynamics.
The work of Li, Chen, Hu, *et al.* [8] is another example of ongoing research. In their work, they analyze the negative behavior of *Dropout* in combination with batch normalization and propose new methods to circumvent this problem.
Another example much closer related to this work is the research of Borras, Klein, and Fröning [9] in which they analyze the effect of batch normalization on the robustness of networks trained with noise injection. In their work, they find that the use of batch normalization increases the robustness against noise for both additive and multiplicative noise. In case of multiplicative noise, batch normalization seems to augment weight binarization during training, which makes networks extremely robust against noise.

2.1.3 Quantization

Quantization is a model compression technique in machine learning that allows for the reduction of the model's memory footprint and an increase in inference speed. Prior work has shown that neural networks can often preserve their prediction accuracy even if internal computations are executed at lower precision [10], [11]. This already shows the fundamental idea behind quantization. In quantization, the commonly used 32-bit floating point data type is replaced with a smaller one. Most of the time integer data types are used since their arithmetic operations are faster compared

with their floating-point counterpart. The smaller data types automatically lead to a smaller memory consumption of the model parameters.

In general, arithmetic operations are faster on smaller data types than on larger ones, however, this is only the case if the processor supports the special execution of smaller data types. Often small data types are packed into a bigger one that corresponds to the native word size on the system. A single operation on this native data type then implicitly results in the parallel processing of multiple of the smaller packed data types. If this is not possible and there are no special units to execute low-precision operations in parallel, quantization has only the advantage of reduced memory consumption. This however can also lead to a speed-up if the computation is limited by bandwidth constraints.

For very strong quantization to small bit widths, the use of FPGAs becomes attractive because they allow the implementation of custom hardware operations on these untypically small data types, which can result in very high throughput [12]. However, the quantization also introduces errors in the computation. This error becomes bigger for smaller bit widths since the number of representable values decreases.

For quantization in neural networks, there are fundamentally two approaches. The first one is post training quantization (PTQ) [10] in which the quantization is done after the initial training. In this approach, the quantization is applied right after the training of the floating-point model, yet the quantization error is not compensated with additional training steps. This makes PTQ a challenging approach since finding quantization strategies that maintain accuracy is non-trivial.

The second one is quantization aware training (QAT) [13, p. 10+18], [14] in which training happens with quantization introduced in the forward pass. This allows the model to adjust to the introduced quantization error so that often the accuracy can be recovered. The quantization aware training can happen after an initial full precision training or the quantized network can be trained from scratch. The former is sometimes seen as a combination of PTQ and QAT, however, the community is very inconsistent in this regard.

Training with quantization, however, also introduces challenges. Some operations used in quantization, such as rounding or clipping, often have parts that are non-differentiable or result in gradients that are zero most of the time. To use backpropagation together with quantization, a gradient approximation technique called straight-through estimator (STE) [15] is used. Fundamentally the STE separates the forward and backward operations so that sensible yet arbitrary functions in the backward pass can be used. This solves the difficulties with the quantization operations. For these functions, the backward pass is often approximated with the identity function but there are also much more advanced methods [16].

In this work, we have to use quantization since the BSS-2 system performs analog operations on quantized integer inputs (5-bit unsigned activations, 7-bit signed weights). Since we focus in this work on improving accuracy after hardware-in-the-loop training (see Section 2.1.4), our methodology implicitly leads to QAT.

Besides general quantization approaches, we also present concrete quantization schemes. For the following explanation of these, we rely on the information in [13] and [17].

In quantization, we want to map a real value $x \in \mathbb{R}$ from the range $[x_{\min}, x_{\max}]$ to the quantized integer range $[q_{\min}, q_{\max}]$. We start with a very simple approach, that we also use in the main part of this work.

Scale Quantization

A straightforward way of quantizing the input values is to apply a uniform transformation. This means that every value of the input range is scaled with the same scaling factor. After scaling we have to round each value so that the results are integers and clip any value outside the target

range. Equation 2.4 shows this quantization approach.

$$q = \text{clip}(\text{round}(s \cdot x), q_{\min}, q_{\max}) \tag{2.4}$$

$$\text{with} \qquad \text{clip}(x, l, u) = \begin{cases} l, & x < l \\ x, & l \leq x \leq u \\ u, & x > u \end{cases} \tag{2.5}$$

If we need to recover the real-valued result from the quantized representation, we can dequantize it with:

$$x = \frac{q}{s} \tag{2.6}$$

If we have a signed quantization range, then $q_{\min} = -2^{b-1}$ and $q_{\max} = 2^{b-1} - 1$ where b is the number of available bits in the quantization range. For an unsigned range the values change to $q_{\min} = 0$ and $q_{\max} = 2^b$. In both cases, it is useful if the input range matches the symmetry or asymmetry of the quantized range. Only then there are no values of the quantization range that are unused, which leads to a minimized quantization error. For a signed quantization range this means ideally $x_{\min} = -x_{\max}$ while for an unsigned quantization range $x_{\min} = 0$.
This becomes more clear if we consider a possible computation of the scaling factor s. If we want to map the full input range to the output range, we compute

$$s_{\text{signed}} = \frac{2^{b-1} - 1}{\max(|x_{\min}|, |x_{\max}|)} \tag{2.7}$$

$$s_{\text{unsigned}} = \frac{2^b}{x_{\max}} \tag{2.8}$$

For Equation 2.7 this means that we never use $q_{\min} = -2^{b-1}$ in our quantized range. This is commonly done in favor of symmetry but for a very small b this is problematic e.g. for $b = 2$ we lose 25 % of the representable values if we never use $q_{\min}$. Also, the mentioned symmetry in the input range is important for a good utilization of the quantized value range. If we look at the denominator of Equation 2.7, we see that the mapping depends on the bigger one of the absolute minimum or maximum value of the input range. If those are vastly different either positive or negative quantization ranges are poorly used. Similarly for Equation 2.8, if $x_{\min}$ is much bigger than 0 a lot of the quantized range is unused.
With the symmetric quantization, all we need to fully describe the input values is their quantized representation and a single scaling factor. In the context of matrix multiplication, we can quantize the operants separately. This means activation and weight matrix can have different scaling factors. For the BSS-2 system this is particularly sensible since the quantization ranges differ for weights (signed 7-bit) and activations (unsigned 5-bit).
Also, it is possible to compute the scaling factor for each activation, each batch, each column of the weight matrix or even statically with statistics for the whole training set. In the main part of this work, we use different approaches to evaluate their performance in the presence of noise introduced by the analog hardware.
If we use the matrix multiplication of Figure 2.1 as an example and use a single scaling factor for each the input batch $A = (a_{i,p}) \in \mathbb{R}^{i \times p}$ and the weight matrix $W = (w_{p,j}) \in \mathbb{R}^{p \times j}$ with A^* and W^* denoting the quantized versions, we get:

$$C = A \cdot W \tag{2.9}$$

$$s_C C^* = s_A A^* \cdot s_W W^* \tag{2.10}$$

$$s_C c^*_{i,j} = s_A s_W \sum_{q=0}^{p-1} a^*_{i,q} \cdot w^*_{q,j} \tag{2.11}$$

This shows that the computationally intensive part, the sum of products, can be done fully in integer arithmetic.

For the computation of the sum of products, it is important to avoid overflows. Therefore often an accumulator with a higher bit width is used. The required bit width can be determined using a worst-case analysis. If we assume $a^*_{i,q} \cdot w^*_{q,j}$ are unsigned and quantized to 5 bit, their product requires in the worst case a bit width of 10. Further, if we assume that we have to accumulate $64 = 2^6$ of these products, the result would be in the worst-cases $2^{10} \cdot 2^6 = 2^{16}$ so the accumulator should have at least 16 bit.
In case the next operation requires the operand to have a lower bit width a dequantization followed by a quantization can be applied. This would preserve the total range of the computation but potentially waste a lot of the range for very high or very low outliers. An alternative is bit shifting where the scaling factor is accordingly modified followed by a clipping operation. How far to shift the elements can be computed from statistics if only inference is done after the quantization. For QAT these steps are simulated in floating point precision so that overflow protections and matching bit widths are unnecessary.
In the previously described quantization scheme, we determined the scale based on the absolute maximum of the input range. However, there are alternatives like using a certain percentile of the input range [13, p. 6].

Scale and Shift Quantization

As mentioned a significant disadvantage of the simple scale quantization is the requirement that the input distribution must match the symmetry of the quantization range to not waste quantization levels.
An alternative to that is the scale and shift quantization (often called affine quantization).
Here an additional quantization parameter is used, the so-called zero-point. In this scheme, the scale s and zero-point z are calculated as

$$s = \frac{q_{\max} - q_{\min}}{x_{\max} - x_{\min}} \tag{2.12}$$

$$z = -\text{round}(x_{\min} \cdot s) - q_{\min} \tag{2.13}$$

The rounding operation is introduced to accurately represent the input value 0 so that after dequantization its original value is preserved. With these parameters, the quantized values can be computed as

$$q = \text{clip}(\text{round}(s \cdot x + z), q_{\min}, q_{\max}) \tag{2.14}$$

Fundamentally this results in a better utilization of the output range than with the plain scale quantization. However, because of the additive quantization parameter z, the matrix multiplication becomes more complex:

$$C = A \cdot W \tag{2.15}$$

$$s_C(C^* + z_Z) = s_A(A^* + z_A) \cdot s_W(W^* + z_W) \tag{2.16}$$

$$s_C(c^*_{i,j} + z_Z) = s_A s_W \sum_{q=0}^{p-1} (a^*_{i,q} + z_A) \cdot (w^*_{q,j} + z_W) \tag{2.17}$$

As shown by Equation 2.17 the zero-points of the weight and activation cannot be distributed outside the summation. This makes the computation slightly more demanding than Equation 2.11. Since this quantization scheme is very common but not used in this work, we refer for further steps to [17].

Both described quantization schemes show two fundamental approaches for quantization but in practice, the specific implementations vary. As we mentioned there are different options for choosing the input range but there are many more adaptations.

A very fundamental question is how to choose the quantization parameters during training. Sakr, Dai, Venkatesan, *et al.* [14] mention for example static approaches that initialize the parameters after the initial full precision training and then keep the parameters constant during training. Others recompute the parameters for each batch or even make them part of the learnable parameters.
In this work, besides others, we use a hybrid form of the static and dynamic approaches that updates the scaling factor based on an exponential moving average (EMA). We compute the exponential moving average v_{EMA} as

$$v_{\text{EMA}} \leftarrow d \cdot v_{\text{EMA}} + (1 - d) \cdot v_{\text{new}} \tag{2.18}$$

where v_{new} is a new observation and d is the decay rate, which is usually close to 1. Sometimes the term $(1 - d)$ is called momentum and hence the formula is written in a slightly different form. However, the term should not be confused with the term used for neural network optimizers although they have similar effects.

2.1.4 Hardware-in-the-Loop Training

Hardware-in-the-loop training takes on an important role in this work as it allows to compensate for static imperfections like static offsets or non-linearities of hardware. In this training method, the pre-trained model is retrained with the actually used hardware in the training loop, very similar to the previously mentioned quantization aware training. By that, it allows good model performance even with imperfect and variable hardware. Besides static imperfections hardware-in-the-loop training can also compensate for dynamic imperfections like noise but only to a limited degree. Although we use it in this work to improve the model performance of artificial neural networks, its effectiveness has also been shown on the first BrainScaleS system in the spiking operation [18].
Fundamentally, hardware-in-the-loop training uses the inherent capability of neural networks to model arbitrary functions. By including the hardware in the training loop its characteristics become part of the function to be modeled. This should lead to similar performance as with ideal hardware, yet there are some constraints.
Although neural networks are theoretically capable of modeling arbitrary functions there are limitations when it comes to hardware-in-the-loop training. Neural networks are large and complex systems that require sensible updates during training to successfully model complex functions. As we mentioned in Section 2.1.1, modern machine learning relies on backpropagation to compute the parameters' gradients, which are essential for the update step. A requirement for backpropagation is the partial derivative of every step in the forward pass. Although the derivative does not have to be extremely precise, it should be at least a sensible approximation. The success of the STE [15], a technique used in quantization to approximate the gradient of for example the rounding function, is a confirmation of that. For hardware-in-the-loop training, this means that either the partial derivatives of the hardware must be known and deterministic or at least a close approximation of it must be available during training.
The closer the approximation of the hardware derivatives is to the ground truth the better, however, it is unclear how big the difference can be before the training fails. Based on the remarkable error tolerance of neural networks in the past (see Section 2.3), we also assume some robustness in this regard.
Another quite relevant aspect of hardware-in-the-loop training is the speed of training. Since the hardware is part of the forward pass, it can also become the new bottleneck for the computation. This is typically the case if small and energy-efficient devices are the designated target platform. In our work, this is a significant problem as it increases training time drastically. Still, if the used hardware represents a time-invariant system, this price must be paid only once during training. A solution addressing this problem is digital modeling of the imperfect hardware [19] (more details in Section 2.3).

2.2 BrainScaleS-2 and the HICANN-X Chip

The hardware used in this work is the latest version of the BrainScaleS-2 hardware, the HICANN-X chip in its third generation (HICANN - High Input Count Analog Neural Network). It is a mixed signal chip with digital I/O and an analog core. Currently, it is a research platform combining efforts in neurology, computer science, and physics. It has a very versatile architecture and supports both spiking neural networks and conventional artificial neural networks [20]. In the spike-based mode, the system tries to model the behavior of biological neurons with their event-based, asynchronous communication. The current BSS-2 system implements the Adaptive-Exponential Integrate-and-Fire (AdEx) neuron model [21], a more complex mathematical model of a biological neuron than the plain Leaky Integrate-and-Fire (LIF) model [22].
However, in this work, we focus on a different operation mode and hence discard a lot of the system's complexity. We use the rate-based mode, sometimes called HAGEN mode (Heidelberg analog evolvable Neural Network). The name is a reference to an earlier project of the group that designed the BSS-2 system. In this mode, we can use the system to perform analog matrix multiplication, which is a central operation in artificial neural networks (see Section 2.1.1).
Besides the analog core, the ASIC has a digital I/O and control part. It consists of two (one per hemisphere) microprocessors based on the Power Instruction Set Architecture including a SIMD extension [23, p. 133]. This microprocessor is often referred to as Plasticity Processing Unit (PPU) since its intended purpose is to perform weight updates to implement synaptic plasticity models from biology. The PPU can be used as an on-chip controller for the analog core but also external control is possible.
In the current setup, there is a field-programmable gate array (FPGA) that manages communication with the BSS-2 ASIC and enables real-time control. It buffers data from the host computer or the chip while coordinating real-time experiments. Additionally, the on-chip processors can access external memory connected to the FPGA to support more extensive programs and data storage [24, p. 3].
If we omit the digital part, the analog core can be described in a simplified form as seen in Figure 2.2. In HAGEN mode the most relevant components of that analog core are the synapse drivers, the synapses, the neurons and the columnwise parallel analog-digital converter (CADC).

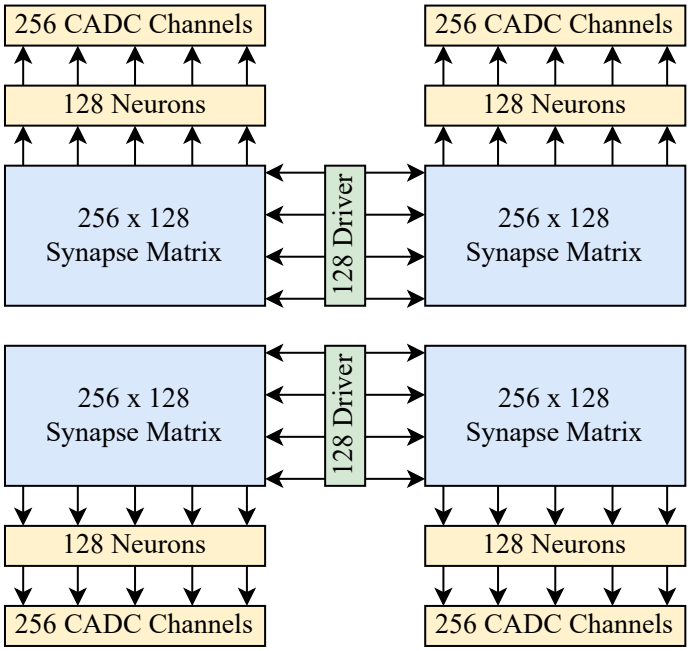

Figure 2.2: Reduced Topology of the HICANN-X chip.

The HICANN-X chip consists of two mostly independent hemispheres that are functionally equiva-

lent but can differ in their imperfections. However, in the context of this work, we see the chip as a single instance without distinguishing hemispheres or quadrants. In total, it contains 512 neurons and 128k synapses. If we need positive and negative weights, we have to combine two synapse rows so that one represents the positive weight and the other one represents the negative weight. With that, we can execute a parallel vector-matrix multiplication with matrices up to a size of 128 rows and 512 columns in singed mode. Respectively 256 rows and 512 columns if we omit the sign. By combining the results of multiple chip executions also bigger input operands can be processed.
The BSS-2 system provides a sophisticated software stack to program the chip. The most relevant packages for artificial neural networks on the BSS-2 system are `hxtorch` [25], [26], [27] and `calix` [28].
The `hxtorch` package is a Python extension to the widely used machine learning framework Pytorch [29]. It allows an easy integration of the neuromorphic hardware into existing projects. By providing replacements for commonly used layers, `hxtorch` provides an easily usable programming interface. Additionally, rounding to integer values and range checks are done by `hxtorch`. The in- and outputs of the matrix multiplication are floating point tensors that are internally converted to the corresponding hardware resolution with $a_j \in [0, 31]$ being the unsigned values of the input activation, $w_{ij} \in [-63, 63]$ the signed elements of the weight matrix and $c_j \in [-128, 127]$ the signed elements of the output activation. Also, the partitioning of larger matrices that do not fit on a single chip is handled by `hxtorch`. Besides the layer replacements, other functions are provided like addition on the ASIC's digital microprocessor (PPU) or the so-called `ConvertingReLU`, which applies the ReLU function to the input operand and performs a shifting operation so that the outputs of the analog matrix multiplications are scaled to the value range of the inputs. A great example that shows in only 14 lines of code the fundamental use of the `hxtorch` package can be found in [24, p. 17 Fig. 13].
Another relevant feature of `hxtorch` is the mock mode, which provides a way to simulate some of the hardware's characteristics without the need for an actual hardware instance. The mock mode models the noise by adding samples drawn from a Gaussian distribution to the output of the digital matrix multiplication. The intrinsic gain is represented by a single global scaling factor. Both the standard deviation of the Gaussian distribution and the scaling factor can be freely adjusted but also a measuring function to retrieve values of a specific hardware instance is provided. With that, the rough behavior of the chip can be reproduced, however many imperfections like saturation and differences among components are not considered.
The `calix` package [28] provides a high-level programming interface for the calibration process of the chip. This calibration is a central part of the BSS-2 system as it allows reconfiguring the hardware for a certain operation mode. During the calibration, hardware parameters are adjusted to values that are beneficial for the current execution mode. Besides that, the calibration routines try to compensate for imperfections in the parallel analog units so that they exhibit uniform behavior. The `calix` package provides functions to generate default calibrations for each operation mode but the user has also the opportunity to provide target values for certain hardware parameters. With that users can create application-specific calibrations that might lead to a more beneficial behavior of the chip than the default calibration. Currently, the default calibration tries to make the analog matrix multiplication behave as similar as possible to its digital counterpart.
In this work, we use the calibration to configure the chip in a way that results in a behavior that is better suited for the execution of neural networks.

2.2.1 Operation Principle

Since we want to optimize the chip's behavior with the calibration, we need an understanding of the operation principle. Therefore we use this section to explain the low-level principles of the analog matrix multiplication and the involved parameters.
Since matrix multiplication is a parallel dot product, the chip architecture resembles this parallelism in the replication of similar neuron compartments and synapse columns. Because the fundamental

functioning principle is similar for all neurons, we focus on a single one. In Figure 2.3 a simplified schematic of this neuron circuit with its synaptic inputs can be seen.

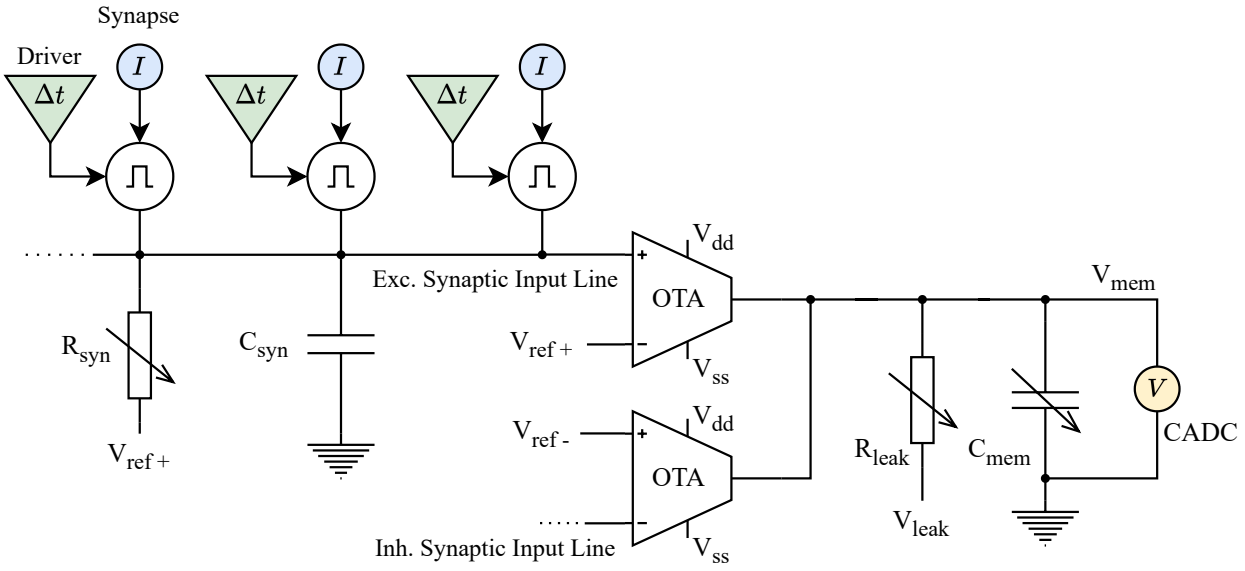

Figure 2.3: Simplified schematic of a single neuron and its synaptic input[1] in HAGEN mode. Input activations are represented by the synapse drivers, weights by the synapses, and the output activation by V_{mem}.

The right side of the schematic shows the neuron with the membrane capacitor C_{mem}, the leak term R_{leak} and both excitatory and inhibitory synaptic input operational transconductance amplifiers (OTAs). On the left side, it shows only the excitatory synaptic input circuit. The inhibitory circuit is omitted since it is identical to its excitatory counterpart. Their only difference is the polarity of the signals at the OTA inputs. In case of the excitatory synaptic input, the input events cause a positive current onto C_{mem} and increase the voltage while the events on the inhibitory synaptic input cause a negative current and decrease the voltage.
The triangles at the top represent the synapse drivers. Each of them encodes the magnitude of a single input feature in the activation vector. Next to the driver is a synapse that is connected to the row of the synapse driver. A single synapse encodes the magnitude of a weight. Together, both the synapse driver and synapse emit a rectangular current pulse. The synapse driver controls the temporal aspect of this pulse, specifically its width, while the synapse determines the magnitude or strength of the current. However, a synapse can only create positive currents. Its connection to either the excitatory or inhibitory input decides whether the contribution increases or decreases the voltage on C_{mem}. This structure allows the implementation of signed weights by using two synapses. Both synapses are connected to the same synapse driver but depending on the sign only one of them is used to create a current pulse on either the excitatory or inhibitory synaptic input line. Effectively this increases the 6-bit unsigned resolution of a single synapse to a 7-bit signed resolution of a synapse pair.
On the circuit level, a synapse is a digital-analog converter (DAC), which converts the digitally encoded weight into a proportional current. The synapse driver controls for how long the current of the DAC is enabled with a trigger signal. How this trigger signal is created is explained in more detail in the following calibration section.
The conversion of the digital weight value to a current is done with parallel, binary-weighted current sources that are enabled in different combinations to create the desired current. Since the weight

[1] We assume that the synapse current increases the voltage on the synaptic input line. In the actual system, synapse drivers decrease the voltage on the synaptic input line and hence the polarity of the OTAs must be switched so that excitatory and inhibitory OTA still have the same effect. In our opinion, this adaptation simplifies the schematic because both current and signal flow are from top to bottom, respectively from left to right.

of a single synapse has a 6-bit resolution (unsigned), there are 6 parallel current sources. As the strength of these current sources is designed to increment in powers of 2, the binary representation of the weight can be used to select the required current sources. A more detailed description of the synapse, yet with a focus on the spiking operating mode, can be found in [23, p. 129].
The current pulse causes the accumulation of a certain amount of charge $Q = \Delta t \cdot I$ on one of the synaptic input lines and hence represents the element-wise multiplication of the dot product. This charge causes a voltage increase on C_{syn}, which is picked up by the connected OTA. An OTA can be most simply described as a voltage-controlled current source. The conversion factor (gain) is called transconductance, which gives the OTA its name. Since the dot product requires the accumulation of the elementwise products, R_{leak} is configured high so that the OTA's output current is integrated on the membrane capacitance C_{mem}.
After the input pulses of all synapse-driver pairs have been sent, the accumulated charge on C_{mem} results in a voltage proportional to the dot product of the input activation with a synapse column. This voltage is then converted to a digital signal by one of the CADC's channels. Fundamentally this describes the dot product but there are some relevant details for a complete understanding of the circuit.
The input current pulses (or events) are sent sequentially because the voltage on C_{syn} must not exceed a certain threshold. Above this threshold the behavior of the OTA becomes (too) non-linear [30, p. 9] and if we further increase the voltage, the output current saturates. This makes the contribution of the input events to the final output voltage unfair and hence introduces a variable bias to the computational result. The variable gain of the OTA has been analyzed in more detail in [21, Fig. 2 (b)].
To further improve the behavior of the chip, the designers have introduced another feature in that context. If an input feature with a value of zero is sent, the input event is skipped. In the analog execution, an input feature with a value of zero would lead to an extremely narrow current pulse, although it should not affect the output at all. Therefore the synapse driver with an activation of zero is simply deactivated and the next driver immediately continues with the next current pulse. This also has the positive side effect that the overall accumulation time is reduced for every zero element in the activation so that no interfering signals can be induced during this hypothetical idle time.
Because the voltage at the input of the OTA depends on the activation and weight magnitude and due to the event serialization also on their order, the effect of the variable bias is hardly predictable in the precision required to compensate for it after the computation. Hence the non-linear behavior should be avoided.
This can be achieved with a small R_{syn}. By sending the input events sequentially with a time interval between the input events the charge on C_{syn} is periodically reduced because of a current through R_{syn} and thus the voltage decays as well. The time between successive input current pulses is controlled with the parameter `wait_between_events`. By increasing it, the voltage decays further, which leads to a better mitigation of the dynamic synaptic-input saturation. However, it also increases the time required for the total accumulation.
Since R_{syn} directly affects the synaptic input time constant $\tau_{\text{syn}} = R_{\text{syn}} \cdot C_{\text{syn}}$, it also controls how fast the voltage on the synaptic input line decays. By choosing a small R_{syn} and hence a small τ_{syn} the saturation mittigating effect of `wait_between_events` can be enhanced. However, τ_{syn} has also another fundamental effect. If the voltage on the synaptic input line decays faster, the OTA emits overall less current. This leads to a reduced voltage on the membrane capacitor hence the finally digitized result appears smaller. This means that decreasing τ_{syn} also reduces the overall gain of the operation.
The OTA-gain (transconductance) also influences the overall gain of the operation. By adjusting a bias current to the OTA it can be adjusted. This can be done independently for each OTA, however a similar transconductance in both the excitatory and inhibitory OTA leads to a better common mode rejection.
A last and very flexible method to adjust the gain of the operation is to repeat input events

multiple times. This proportionally increases the voltage amplitude on C_{mem} and hence the gain. This might be necessary if only small input matrices are multiplied resulting in a limited number of input events and small voltages on C_{mem}. In the current implementation, the whole input pattern is repeated instead of single drivers enabling the synapses multiple times in close succession [30, p. 31]. The parameter that controls how often input events are sent is called `num_sends`. Similar to `wait_between_events` an increase also leads to a longer total accumulation time.
The remaining important parameters of Figure 2.3 are R_{leak} and C_{mem}. As mentioned R_{leak} is set high to avoid leakage of the accumulation result. Unfortunately, this also has a disadvantage. The high resistance makes V_{mem} very sensitive to constant offset currents from the OTAs. The voltage can drift if the reference voltages of the OTAs are not calibrated precisely enough. Adjusting R_{leak} is also quite simple since it is implemented as another OTA yet with negative feedback [30, p. 8]. This means that an increasing V_{mem} increases the current out of the membrane capacitor C_{mem} and into the OTA. Therefore it behaves similarly to a simple resistor. After the digitization of the output voltage, the leak OTA is also used to reset the voltage to a reset potential. During that time R_{leak} is set very low.
The last adjustable parameter is C_{mem}. It can be varied by connecting or disconnection multiple parallel capacitors. Since the current on C_{mem} is controlled by the OTAs, the emitted charge does not change if the voltage on C_{mem} rises. If we consider that $V_{\text{mem}} = C_{\text{mem}} \cdot Q$, this means that C_{mem} is another parameter that affects the gain of the overall operation.

2.2.2 Calibration

As we described at the start of the previous section, the BSS-2 system provides the `calix` package to create a global chip calibration. Besides the mentioned equalization of the components, it sets suitable values for the just described parameters R_{syn}, OTA gain, R_{leak} and C_{mem}.
The default calibration tries to make the analog matrix multiplication universally applicable, by having a uniform behavior of all components. Especially in the context of machine learning, this allows the reuse of models trained on fast, purely digital hardware so that BSS-2 is only used for efficient inference. However, some imperfections remain even after calibration. Without any retraining, this leads to a decrease in accuracy compared with results on purely digital hardware [31, p. 8], [32, p. 40], [19]. Retraining however allows the model to adjust to the remaining imperfections and to recover accuracy.
Even though the default calibration has achieved good results in previous studies, we assume that the heavy focus on uniformity is not necessary with hardware-in-the-loop training. The analysis of J. Weis [30, p. 65] has confired that assumption. Instead, there might be other configurations that are more beneficial for improved accuracy after hardware-in-the-loop training.
To have a better understanding of the calibration of a few important components this process is described in the following. A more detailed description can be found in [30].

Synapse Driver

For each element of the digital input vector, there is a synapse driver. Its purpose is to convert the digital input into a pulse width. Simplified, it represents the time component in the formula $Q = \Delta t \cdot I$. Internally the conversion from the digital input to the pulse width happens in a few steps. First, the digital input is converted to a voltage V_{Thresh}, which is then compared with an internally generated voltage ramp. When both voltages are equal an event is triggered that enables the current flow. The current is later disabled when the ramp reaches its end. In Figure 2.4 this process is visualized.
Generating a pulse width takes in total 8 ns, which corresponds to 1 clock cycle and sets an upper bound on the number of pulses a synapse driver can produce. In the first 2 ns the driver is in a discharge phase in which the ramp voltage is lowered to a defined initial potential. In the following precharge phase, the ramp voltage already starts increasing and there is the option to connect an additional current source to the ramp, which is only active during this adjustment

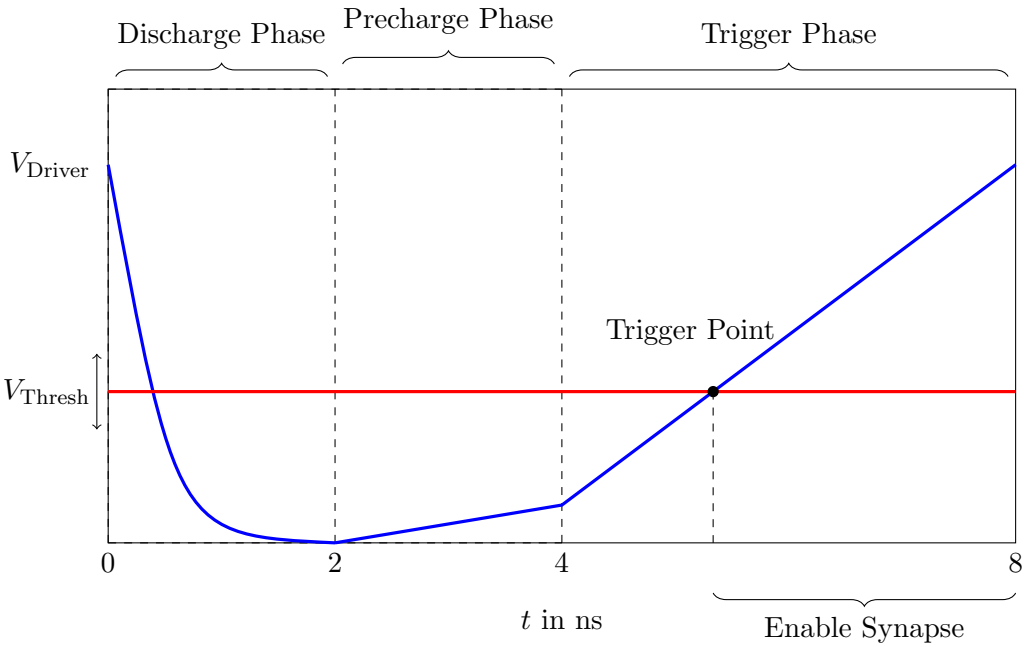

Figure 2.4: Trigger cycle of a synapse driver.

phase. This allows the starting voltage of the ramp to be shifted for the trigger phase to allow for offset calibrations to achieve better uniformity among drivers as the trigger points might be slightly different due to transistor level mismatch [30, pp. 23–24]. This adjustment procedure was not very precise in the first generation of the chip so an improvement in the second generation of the HICANN-X chip additionally allowed an offset of the threshold voltage to compensate for driver variations. This means for the synapse driver the primary calibration is done to get a uniform behavior among all drivers.

CADC

The columnar analog-digital converter (CADC) is responsible for the digitization process after the analog matrix multiplication. It reads the analog voltage of each neuron in parallel with a single slope approach. The working principle is similar to the previously seen synapse driver.

For each quadrant (two per hemisphere), there is a single voltage ramp that is the input to a comparator in each channel of the CADC. The other input of the comparator is connected to the voltage of the neuron's membrane. At the start of the voltage ramp, a counter is reset and later read when the comparator finds both voltages to be equal. The result of the counter is then used as the digitized output of the neuron voltage.

Again there are imperfections among the comparators resulting in different triggerpoints for similar voltages. The default calibration again tries to find similar trigger points for equal voltages on all neurons by adjusting offsets, but also the dynamic range of the CADC is adjusted to the lower and upper limits of the neuron's membrane voltage. Setting up the dynamic input range of the CADC also involves tuning the slope of the voltage ramp so that the upper limit can be reached before the counter overflows. Here we see another trade-off.

Increasing the slope of the reference voltage ramp can make digitization of the output faster but for too steep voltages the maximum voltage is reached before the counter reaches its maximum. This means that the most significant bit and hence the resolution of the CADC is not fully used.

Calibrating the CADC is done at the beginning of the calibration routine so it can be used to find good calibration parameters of other components.

Neuron

In the non-spiking mode, the neuron's calibration is comparably simple since only a subset of the available functionality is used. The purpose of a neuron is the accumulation of the charge produced by each synapse in a column of the array. The accumulation is done on a capacitor (C_{mem} in Figure 2.3) that is in the context of neuromorphic hardware often referred to as the *membrane*. The capacitor should have as little leakage during accumulation as possible but must be discharged between accumulations. For this purpose, an OTA (represented in Figure 2.3 by the adjustable R_{leak}) is used, which connects the membrane to a reference potential V_{leak}. In the non-spiking mode, it is used as a switch that toggles between high-resistance mode during accumulation and high-conductance mode during the reset phase.
Besides the membrane, the primary part of a neuron is its synaptic input. Concerning the synaptic input time constant, we already mentioned that a high value would allow more charge to reach the membrane because the voltage difference on the synaptic input line decays more slowly. At the same time, it would require a higher `wait_between_events` because the voltage of the synaptic input line would reach saturation faster if many pulses are sent in quick succession. Here the calibration again has to find a trade-off. The default calibration sets the time constant to the lower end of the possible range to avoid saturation and decrease the necessary accumulation time.
Nevertheless, more charge reaching the synaptic input would be beneficial to increase the signal-to-noise ratio on the synaptic input line, thereby reducing the impact of noise. Hence the strength of the maximum current of each synapse is set to the maximum value. The strength is controlled by a single instance per quadrant (the synapse bias current), which leads to differences among quadrants.
These differences among others are absorbed in the calibration of the OTAs. Their gain is adjusted so that similar input events lead to similar voltage differences on all neurons.
Additionally, the reference voltage of each OTA is calibrated so that no current is produced if there are no events from the synaptic array. As mentioned this is a very important step since even small constant currents lead to a drift of the membrane potential because of the high R_{leak}. Unfortunately, the calibration of the OTA gain also changes the offset of the reference voltage, requiring additional recalibration steps.
A full calibration of the chip takes around 4 minutes and is done every night to compensate for long-term deviations.

2.3 Related Work

In this section, we offer a concise summary of significant findings within the scope of prior research in our field of study. Our objective is to provide a brief yet insightful overview of research about error tolerance and noise in neural networks. Moreover, we place a specific emphasis on exploring the research conducted on the BrainScaleS-2 system regarding artificial neural networks as it is the energy-efficient yet noise-injecting target platform used in this work.

Error Tolerance

A remarkable property of neural networks especially in the context of analog computations is the ability to tolerate hardware imperfections like noise, non-linearities, static variations or even the failure of internal components. In our work, this is of major importance because this allows us to reach quite high accuracy with simple models and with very energy-efficient yet imperfect hardware. However, this knowledge is not new. Already in the 90s there has been quite involved research on the topic of how to make neural networks more robust against hardware imperfections and how to deal with them in general. In 1994 Murray and Edwards investigate in [33] the effects of noise on the synapses of an MLP and conclude that

> ... the injection of stochastic variations (noise) into the synaptic arithmetic during MLP training enhances the trained network's fault tolerance, generalization performance and learning trajectory [33, p. 801].

Closely related is the capability of neural networks to adapt to the complete failure of neurons described in [34]. Sequin and Clay show that by simple retraining the network can recover from these failures but beyond that they find that with special training the robustness against failures even without retraining can be improved. This technique in which neurons are randomly disabled during training became 22 years later known under the name *Dropout* [35] but as a regularization technique in a different context.
A more recent survey of different fault-tolerance techniques can be found in the work of Torres-Huitzil and Girau [36]. They claim that, beyond energy efficiency and high-performance, novel computing principles like neuromorphic or brain-inspired computing also fundamentally require fault-tolerance. Even though their work covers a much wider view on fault tolerance than the work of Sequin and Clay they come to a very similar conclusion. They state that fault tolerance is not inherent to neural networks but must be built into the model [36, p. 17338]. Yet this does not mean that the neural architecture must be designed for fault tolerance, rather special training techniques or retraining in general result in more robust models.
While the assumption of the necessity for designed fault tolerance seems to be widely accepted for artificial neural networks, the view on that is slightly different for brain-inspired computing systems. In [37] Schemmel, Fieres, and Meier present the FACETS system, a project to explore the modeling of large-scale, spike-based neural networks in analog CMOS circuits and the project before the first BrainScaleS system. In that paper Schemmel, Fieres, and Meier describe biological neural networks as inherently fault tolerant [37, p. 431] with a reference to the work of Kalampokis, Kotsavasiloglou, Argyrakis, *et al.* [38] in which they present a computational model for biological neural networks that can tolerate a neuron connection loss of 70 % - 80 % before a steep neural functionality loss. After that Schemmel, Fieres, and Meier also describe their hardware as fault-tolerant because of the reconfigurability of the system, which allows disabling defective components. Besides that, the authors also point out that many spike-based applications do not rely on all of the neurons and synapses [37, p. 432]. With these insights, the fault tolerance of artificial and spiking neural networks should be treated differently and should not be mixed up. In our work, we only focus on artificial neural networks, but executed on the neuromorphic, analog BrainScaleS-2 system.

Artificial Neural Networks on BrainScaleS-2

Even though the chip's primary strength is in the modeling of spiking neural networks, there have been several studies that are concerned with specifically artificial neural networks on the chip.
To make the chip easily accessible Spilger, Müller, Emmel, *et al.* [27] provide a user-friendly interface, simplifying the integration and utilization of the analog chip. In their work, they present how they integrated the chip's functionality into the widely used machine learning framework Pytorch [29]. Besides high-level replacement options for linear layers, convolution layers, and others they also provide functions for plain matrix multiplication and easy initialization of the chip with a specific calibration.
To handle arbitrary matrix dimensions they also develop a partitioning scheme that tries to execute as many operations as possible in BSS-2's analog core, however, the pre- and post-processing still require the use of digital hardware.
Their implementation appears as a mature and optimized solution that puts a lot of effort into the parallelization of tasks and overlapping execution where possible. A notable example is the presentation of a specialized 1D convolution, which makes better use of the hardware by repeating the filter in offset positions of different matrix columns. The paper includes a performance evaluation of the matrix multiplication for different matrix dimensions and for both chip versions at that time. The maximum throughput of the second chip version is measured to be at 2×10^9 MAC/s.

In [27] also tests with an artificial neural network were conducted. A small 3-layer network with a convolution at the input followed by two dense layers is used to classify human activity from the acceleration data of a smartphone sensor. While the accuracy for the quantized and with artificial noise trained model is at 92.7 %, it significantly drops after the plain transfer to the hardware. However, 1 epoch of retraining on the analog hardware restores the accuracy to 82.3 %, which is good but still around 10 % below the accuracy on digital hardware. While the authors mention the static variations and non-linearities to be detrimental to accuracy yet compensable by retraining they do not provide more details about the remaining characteristics that cause the 10 % gap. In their conclusion, they expect an improved performance for the second chip generation and they declare their intentions to expand the `pytorch` extension with high-level constructs to model spiking neural networks.
The work is extended in much more detail in [39] and [26]. In [39] more details about the mock mode and a hardware-optimized layer initialization can be found. Additionally, it provides more examples of the execution of convolutional neural networks on BSS-2. In [26] more details about the partitioning, the signal-graph execution and more extensive performance measurements can be found.
In our work, we use the provided solution extensively for measurements of the chip behavior but also for the integration of the chip into our network architecture.

A rather recent study was conducted by F. Ebert. In [32] they focus on fast image classification with a mobile system that uses BSS-2 as an energy-efficient accelerator for analog matrix multiplication. They use the MNIST handwritten digit database [40], a widely used image classification task, in combination with a two-layer MLP. The network is that small because of the desired energy efficiency and classification speed. Especially, the latter is a central optimization goal in their work. Nevertheless, the small network is capable of classifying the images with around 96 % accuracy on digital hardware. Similar results were achieved by using the digitally emulated hardware in the forward pass (the so-called *mock mode*), which introduces additive white Gaussian noise and quantization to the computation.
Yet the model achieved only 20 % accuracy after transferring the mock-trained parameters to the actual hardware and only 62 % accuracy after 10 epochs of retraining with the hardware in the forward pass (hardware-in-the-loop training). F. Ebert attributes the poor performance to a problem with the chip's calibration and confirms the exceptional non-uniform behavior for some neurons with an experiment. While this problem and possibly other actual imperfections of the analog hardware explain the highly reduced initial accuracy, the remaining gap of more than 30 % accuracy to the mock mode result is really surprising. We would assume that for a comparably simple classification task like MNSIT, retraining will result in higher accuracies even with the increased non-uniform behavior of a faulty calibration. The small number of only 10 retraining epochs might be a problem, but from the provided accuracy trace in [32, p. 40] further growth with more training time appears unlikely. Besides the retraining length also the network size could limit the network's recovery capabilities. A small network designed for inference speed might not have the necessary number of parameters to adjust to the hardware's imperfections.
However, similar to the author, we assume that an improved calibration could result in a better alignment of the chip's operational characteristics with the demands of neural networks. In our work, we want to address this problem and find ways to improve the behavior of the chip for neural networks. Even though we assume that hardware-in-the-loop training can compensate for many of the hardware's imperfections, we still believe that an application-specific calibration can further enhance the achievable accuracy. This would further decrease the gap between the accuracy of the digital execution and the analog one.

Besides the calibration, there might also be algorithmic adjustments that can improve the behavior of the chip. A first step in that direction has been made by L. Kuhn. In their thesis [41] they train a three-layer MLP on a keyword spotting task. It reaches 78 % accuracy in floating point precision

on digital hardware.
They also use hardware-in-the-loop training and conduct several experiments to find out at which step of the deployment the accuracy is lost. The choice for a quite simple model was made to simplify the analysis of the analog hardware and its special characteristics. In their analysis of the effects of different quantization schemes with and without retraining they find that retraining can fully restore accuracy for a bit width of 7 and above. The analysis is done for the affine quantization scheme of Google's `gemmlowp` library [42] and the uniform symmetric quantization in `hxtorch` [27]. For a bit width of 6, the accuracy decreases only slightly but for smaller bit widths the accuracy is significantly reduced. Also with the mock mode training, they reach 79 % accuracy and hence outperform the floating point baseline. This means that the effects of quantization do not limit the classification performance. However, the retrained models on the analog hardware always perform about 8 % or 9 % worse. These results were only achievable by tuning the hardware parameters `num_sends` and `wait_between_events` for each layer. They are not part of the static global calibration but similarly affect the hardware execution. For bad parameter choices, the accuracy was close to random guessing accuracy.
This was an important contribution since it shows the importance of these parameters even though only the effects on accuracy were analyzed. The effects of both parameters on the hardware imperfections have not been analyzed in more detail and unfortunately, no algorithm has been discovered that enables the efficient search for the layer-specific parameters with out the need for a grid search.
Further, L. Kuhn used the mock mode to verify that the output range can be reduced without losing accuracy and that the matrix multiplication of BSS-2 is more linear in a smaller output range. By using a modified loss function that penalizes outputs above 50 and below -50 the accuracy was further increased to 73 % (still with hardware in the loop). Fundamentally the adaption of the training (here the loss function) increased the accuracy. However by the analysis in [41] it remains unclear which behavioral improvements caused by the reduced output range eventually lead to the increase in accuracy. In their work, L. Kuhn suggests a more linear behavior of the hardware as a possible reason, but since hardware-in-the-loop training should be capable of adjusting to the investigated static non-linearities, other reasons are also possible. Later in this work, we will analyze the noise of the analog matrix multiplication for different output values (Figure 4.3) and find that the noise also changes depending on the output value.
In any case, the work of L. Kuhn describes a fundamental step towards an optimized execution of artificial neural networks on the BSS-2 system.

The work of Klein, Kuhn, Weis, *et al.* [19] is not so much concerned with direct improvement strategies for the execution of neural networks on analog hardware, instead it addresses a methodological challenge. While hardware-in-the-loop training is an important requirement for high-accuracy networks on analog hardware, it often consumes a lot of time.
The adjustment of a network to static variations and non-linear effects requires several epochs of retraining and especially if the analog accelerator is designed for high energy efficiency and not for high throughput, the retraining can take a lot of time. In this context also availability is an important factor. During training, the analog hardware is occupied and cannot be used by other users for other urgent tasks. Especially the analog imperfections make each chip unique. Hence for some applications, the reuse of a specific chip can be very important. In [19] Klein, Kuhn, Weis, *et al.* present a solution to that problem on the example of the BSS-2 system.
By creating a digital clone of the analog hardware (and especially its imperfections) the training of neural networks can happen on fast digital hardware without occupying the actual analog hardware resource. In case of sufficiently available digital computing resources, even multiple training runs can happen simultaneously.
To create the digital representation a combination of splines and lookup tables is used to model the deterministic hardware behavior from a full range measurement. During that measurement, all possible input combinations and input vector sizes are executed and recorded on the real hardware.

Additionally, the non-deterministic noise is modeled by using an additive zero-mean Gaussian distribution.
To evaluate the performance of models trained with the digital hardware representation, the paper shows a comparison of training time and accuracy for different hardware representations that include more and more hardware characteristics. The keyword spotting task and 3-layer MLP in the work of L. Kuhn [41] are also used in the performance comparison with the digital hardware representation. However, the achievable accuracy slightly differs from the results before. This can be caused by a slightly different training procedure (e.g. using a learning-rate scheduler, other loss function, or other hyperparameters), but the paper does not provide further details on that.
The new floating point precision is at 80.8 % and the transfer accuracy without hardware-in-the-loop training is significantly reduced similar to all other presented related works. The use of the digital representation for training can increase the plain transfer accuracy from 12.3 % to 41.0 %, however, the accuracy is still notably reduced. Interestingly the use of the digital hardware representation in combination with hardware-in-the-loop training can increase the achievable accuracy from 66.8 % without the digital representation to 70.1 % with the digital clone. By considering the decrease in total retraining time from 652 min to 201 min the results become even more impressive. The primary drawback of the digital representation, however, lies in the time it takes to record the output for all possible input combinations of the analog chip. Also, the internal processes of the network during training with the hardware representation remain unclear.
Despite not utilizing the hardware representation, our work uses the same keyword spotting task and input data preprocessing with a slightly modified 3-layer MLP (see Figure 4.1).

Another contribution that holds considerable significance for our research is the work of J. Weis [30]. They describe the fundamental operation principles of the BSS-2 system as well as hardware imperfections and the current calibration algorithms. Although a major part of their work is concerned with the calibration for the spiking operation, we are primarily interested in the analog matrix multiplication mode (HAGEN mode). Besides the calibration of the different components, they describe trade-offs between calibration parameters. An example is the trade-off between the synaptic input time constant and the synapse bias current. While reducing the value for each of the parameters can reduce saturation effects, in both cases this would also decrease the strength of the input signal. A shorter synaptic input time constant however allows a faster integration.
To test the performance of the calibration J. Weis also trains an artificial neural network. Similar to F. Ebert they use the MNIST dataset, yet with two different models: a small convolutional model and an MLP. For both models, they conduct experiments in which they analyze the sensitivity of the models to quantization on purely digital hardware and continue with experiments that show the accuracy before and after retraining on the hardware. Because of the high runtime of chip version 1, retraining was only done for 10 epochs and because of the early development stage of chip version 2, no retraining was conducted on the newer chip generation. While the test accuracy barely changes for different quantizations, it significantly drops for both models without retraining. In-the-loop training however restores the accuracy. Overall it seems like the trained models on the MNIST dataset are very robust to any kind of distortion because the highest drop ever observed is from an initial 98.3 % to 92.1 %. This drop is extremely low considering the previous results of other related work.
Besides the retraining, J. Weis also analyzes the performance of the MLP right after transferring the digitally trained model to the hardware. To improve the calibration, they assess the impact of specific calibration parameters on accuracy without hardware-in-the-loop training. Overall they find a high repetition parameter for the analog dot product (`num_sends`) to be very beneficial for the accuracy and decreasing the membrane capacitance to have an almost similar effect.
In our research, we use the results of J. Weis to get an intuition of which parameters might also increase the performance after hardware in the loop training. However, the presented results are quite outdated since they show the behavior of the first chip generation whereas our research uses the third one. Therefore we do not fully rely on the precise results but only use them to get an

impression of the general effect of a certain parameter change.
Another interesting contribution of J. Weis is the analysis with respect to the capability of the dense network to adjust to static variations among chip components. In an experiment they set all calibration parameters gradually closer to the median value of similar parameters in the initial calibration and measure the model performance before and after retraining.
Gain and offsets among neurons can be vastly different after equalization of the parameters, however, retraining can always restore the initial accuracy. The accuracy without retraining, in contrast, gradually decreases. This result is expected and shows the strength of hardware-in-the-loop training to adjust to even quite high static differences among neurons.
In our work, we see the results as an opportunity for optimization. Since static variations can be compensated by training, we are willing to sacrifice uniform behavior after calibration if it enables more accurate results not achievable through hardware-in-the-loop training.

Chapter 3

Peculiarities of Analog Matrix Multiplication

This chapter presents a few characteristics of the analog matrix multiplication on the BSS-2 system that might surprise users unfamiliar with analog hardware. On one side, we want to use this opportunity to give the reader an overview of the most significant peculiarities that usually do not occur in digital computations. On the other side, we want to use it to document the results of the current chip generation.

3.1 Execution Time

In this first section, we look at factors that affect the execution time of the matrix multiplication. Among these factors are the matrix dimensions, which determine the computational effort necessary to compute the result, but also the parameters `num_sends` and `wait_between_events`, which control the execution on the chip. The exact purpose of `num_sends` and `wait_between_events` is described in Section 2.2.1. In this section only their impact on the execution time is relevant.
We start with the impact of the matrix dimensions. Our initial assumptions regarding the execution time arise from an algorithmic perspective. Referring to the nomenclature introduced in Figure 2.1 in matrix multiplication, the number of output elements is the product of the input matrix's height (the batch size) and the weight matrix's width [43, p. 19]. Each output element is the result of a dot product between two vectors with a length equal to the common dimension of both matrices (the number of input features). Therefore we assume that the computational effort necessary for the result scales linearly with all three variable dimensions of the input operands [44, p. 1218].
If we assume square matrices with n rows and columns, the naive matrix multiplication has a worst-case runtime complexity of $O(n^3)$. Even though there are practical algorithms that can perform matrix multiplications with a slightly better worst-case runtime complexity (see [44, p. 85]), for simplicity during our assumptions we expect the behavior of the naive version. Accordingly, we assume that also on the BSS-2 chip the runtime time linearly increases with the dimensions of the input operands.
In Figure 3.1 we see how the batch size and the matrix width affect the total execution time. For stable results, we average the runtime results over 200 iterations and show the standard deviation as error bars. For Figure 3.1 (a) and (b) we choose the number of input features to be 128, which corresponds to the number of available synapse drivers. The batch size in Figure 3.1 (a) can be chosen arbitrarily since it does not correspond to a physical hardware component. We choose a batch size of 128 here since it corresponds to the batch size we also use in our machine learning training in chapter 4. In our opinion, this is a moderate batch size and close to other typically used values. In Figure 3.1 (b) the weight matrix width of 512 corresponds to the total number of neurons on the chip. Since both Figure 3.1 (a) and (b) cover a wide range of values, we use a log-log plot.

In both cases, we see that reducing the batch size or weight matrix width at some point does not further decrease the execution time. We attribute this offset to initializations and setup costs. According to Amdahl [45] this is the "sequential component" that is always present and that cannot be sped up by the parallel processing capabilities of the hardware. But this is not specific to analog hardware, also digital hardware suffers from this problem.
In Figure 3.1 (a) we see that the execution time starts to increase for a matrix width greater than 2^9. This is plausible since the chip features $2^9 = 512$ neurons. Below this matrix width, the chip is underutilized and for bigger matrices, the runtime linearly increases. There is a transition window for matrices up to a width of 2^{11} where the growth rate gradually approaches linear growth but after that, the execution time almost doubles when doubling the matrix width.
For the batch size in Figure 3.1 (b) we see a similar trend but there is not a clear batch size after which the execution time steeply increases. Also, the transition window to linear growth is much wider. This is to some extent also explainable with the chip topology. Since the chip executes matrix-matrix multiplication as a series of vector-matrix multiplications there is no exact start for the linear growth but rather a gradual trend. Because of this execution model, there is no sharp transition from a state with unused hardware resources to a state of full utilization. The setup time can be amortized so that for batches bigger than 2^{11} almost linear growth can be seen. The reason for a slightly smaller growth might be the matrix width of 512 which could be slightly too small for a sufficient amortization. But also the previously mentioned "sequential component" of the batched execution might grow with a growing batch size, thereby limiting the amortization. For even bigger batch sizes than shown in Figure 3.1 (b) a slightly closer approximation to linear growth could be possible.

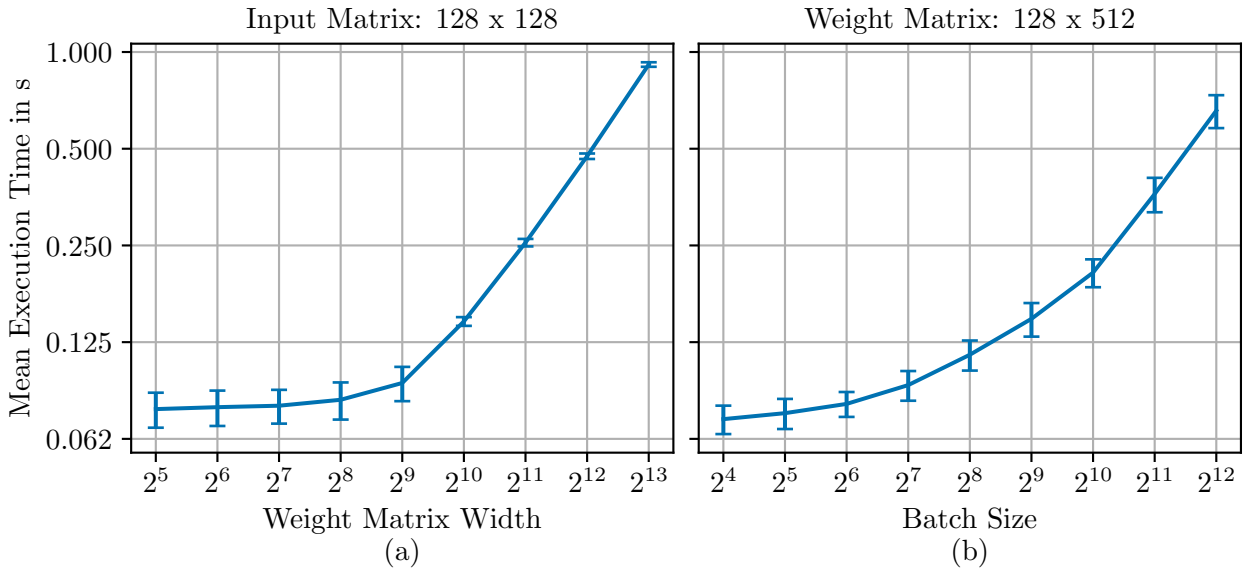

Figure 3.1: Mean execution time of the analog matrix multiplication. With 128 features per activation, `num_sends`=1, `wait_between_events`=1. For more details on the used nomenclature see Figure 2.1.

In Figure 3.2 the runtime dependence on the remaining feature dimension is depicted. Instead of the previous log-log plot, we now use linear axes. This is because the wide range displayable with log-log plots is not necessary in this case to show both the linear dependence and the transition window from constant to linear time. Also, it allows a better view of another peculiarity of the execution model. Since the hardware has only 128 feature inputs in signed mode, the results of multiple hardware vector-matrix multiplications are digitally accumulated if the number of input features exceeds this limit [27].
This execution model explains the shape of the plot. For feature vector sizes that are a multiple of 128, the execution time increases linearly and the chip utilization is high. For any additional

features, another chip execution is necessary, which abruptly increases the execution time. Further features increase the execution time unnoticeably until another multiple of 128 is reached.

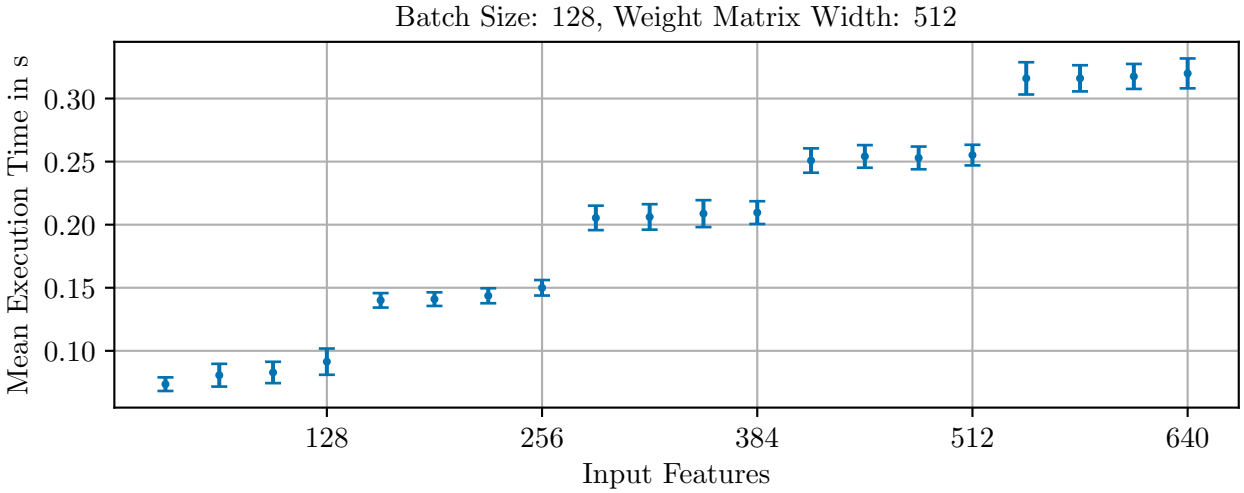

Figure 3.2: Mean execution time of the analog matrix multiplication over increasing input features.

Besides some effects related to the execution model, the hardware behaves quite similarly to digital hardware when it comes to scaling with the operand dimensions. Because of the chip's size and the energy-efficient design, we cannot expect the throughput of modern GPUs. However, in our opinion, a smaller required batch size for efficient execution would be desirable. Since the convergence and the behavior of certain components in Artificial Neural Networks (ANNs) (e.g. batch normalization) are sensitive to the batch size a more flexible choice without sacrifices to efficiency would be great.

In the next step, we look at the impact of the parameters `num_sends` and `wait_between_events`. Both parameters control the analog execution of the matrix multiplication on the chip that we have explained in more detail in Section 2.2.1.
In Figure 3.3 we repeat the experiment of Figure 3.1 (b) but this time we vary the aforementioned parameters.

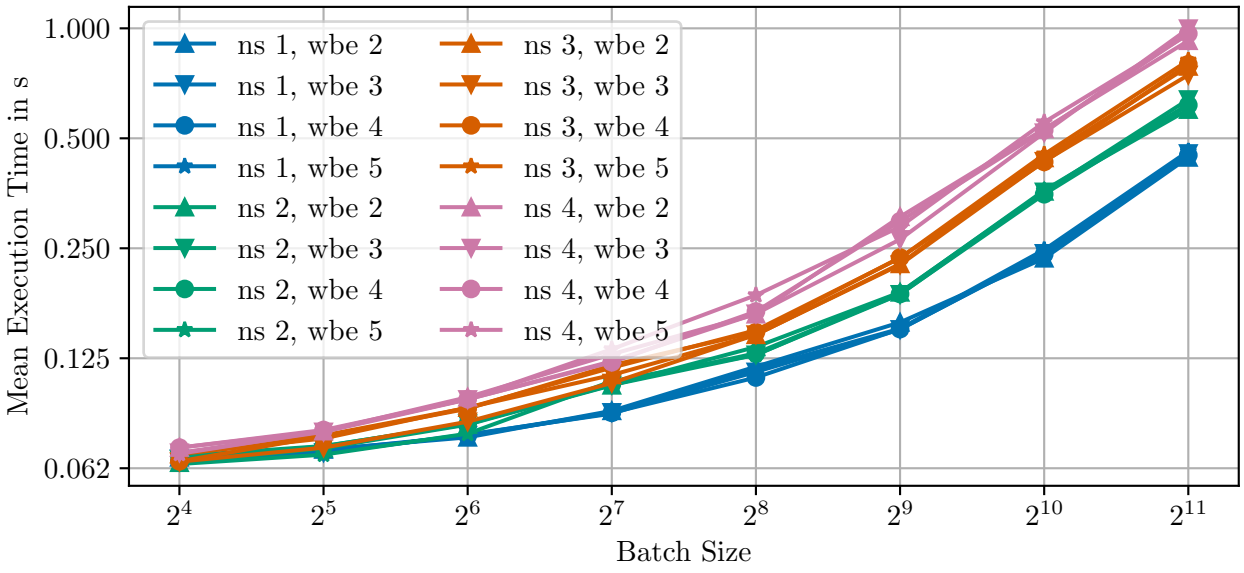

Figure 3.3: Mean execution time of the analog matrix multiplication with 128 features per activation and a matrix width of 512.

The trend of each curve is similar to the previous results, but we see an offset among the curves. It seems like `num_sends` has a significant impact on the runtime while `wait_between_events` has only a small or almost no effect. Since the results are shown in a log-log plot the offsets imply a different growth coefficient and not just a constant runtime difference. Also, it can be seen that the offset among the curves diminishes for small batch sizes. It is important to mention that the plots show only curves with `wait_between_events` greater than 1. For a value of 1 the execution time was visibly smaller. This was an exceptional behavior, which is included in Figure 3.4.
To see the effect of the two parameters on the execution time more clearly, we include Figure 3.4. Here we pivot the plot so that `num_sends` (ns) respectively `wait_between_events` (wbe) are on the horizontal axis at a batch size of 128.

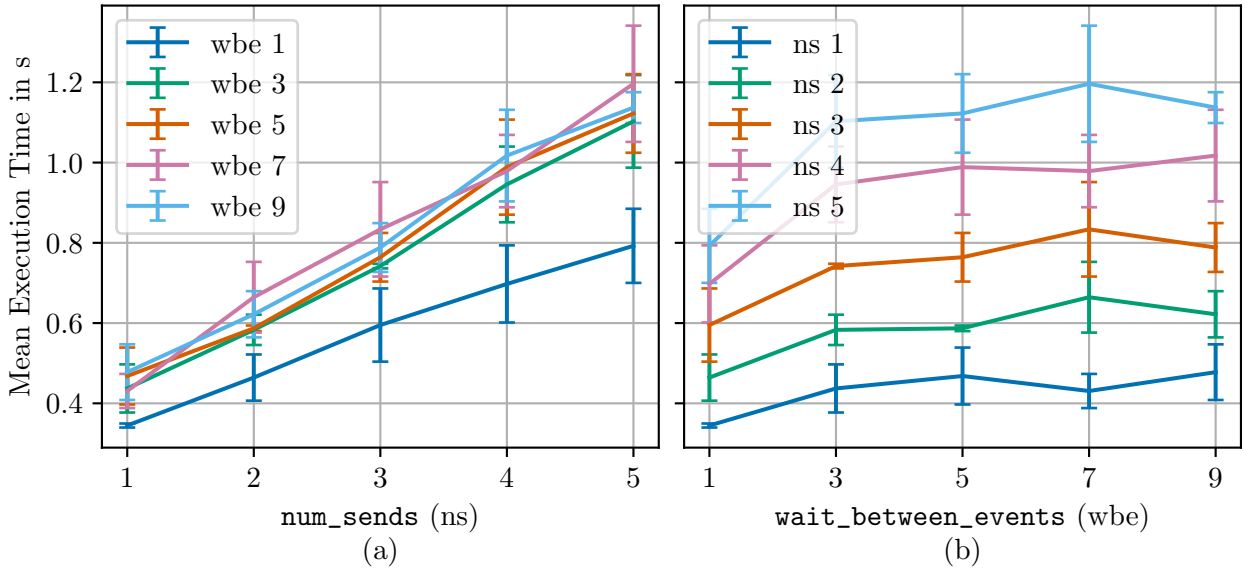

Figure 3.4: Mean execution time of the analog matrix multiplication when varying the parameters `num_sends` and `wait_between_events`. Input Matrix: 128 x 128, Weight Matrix: 128 x 512.

In Figure 3.4 (a) the linear increase of time with a growing `num_sends` can be seen quite clearly and also the exceptional behavior of `wait_between_events` for a value of 1. Figure 3.4 (b) confirms that `wait_between_events` does not increase the runtime except for the transition from a value of 1 to higher values.
We see that by changing `num_sends` we can significantly change the overall runtime of the matrix multiplication. By increasing `num_sends` also the runtime increases. In applications that require maximum energy efficiency, `num_sends` should be chosen as small as possible to reduce the runtime and hence energy consumption. The results of Figure 3.4 (a) indicate that for the application machine learning both training and inference speed significantly depend on the parameter `num_sends`. By changing its value from 1 to 4 the execution time of the matrix multiplication roughly doubles.

3.2 Spatial Dependency

As in any other chip, there are variations among the physical components, which are ideally identical. In digital computations, these variations are hidden by quantization. As long as the produced signal is sufficiently far away from the quantization threshold, slight variations in the signal still result in the same digital representation. However, in analog computations, these variations cause the neurons to behave differently and hence produce slightly different results. In the present case not only manufacturing differences of the components themselves influence the computation but also their physical location on the chip. Therefore we call this location-dependent phenomenon the *spatial dependency*. As we have already described in Section 2.2.2 the calibration aims at reducing

these differences to a minimum. Yet certain cases exaggerate these differences to a degree where the calibration cannot compensate them anymore.
This primarily happens when very small input activations and very large weights are used in the matrix multiplication. But also parameters like `num_sends` and `wait_between_events` can have an impact.
To measure this effect we execute a matrix multiplication with a weight matrix of 512 neurons and 128 weights per neuron and set all weights to a value close to the possible min/max value. In this case, we use a value of 50. As an input matrix, we use a batch of 200 activation vectors so we can average the results. Each activation vector consists of 65 non-zero values, which are set to a value close to the minimum activation value (here set to 2). The rest of the 128 vector elements are set to 0 because the accumulated result of more partial sums would exceed the output value range. The results of the experiment can be seen in Figure 3.5.

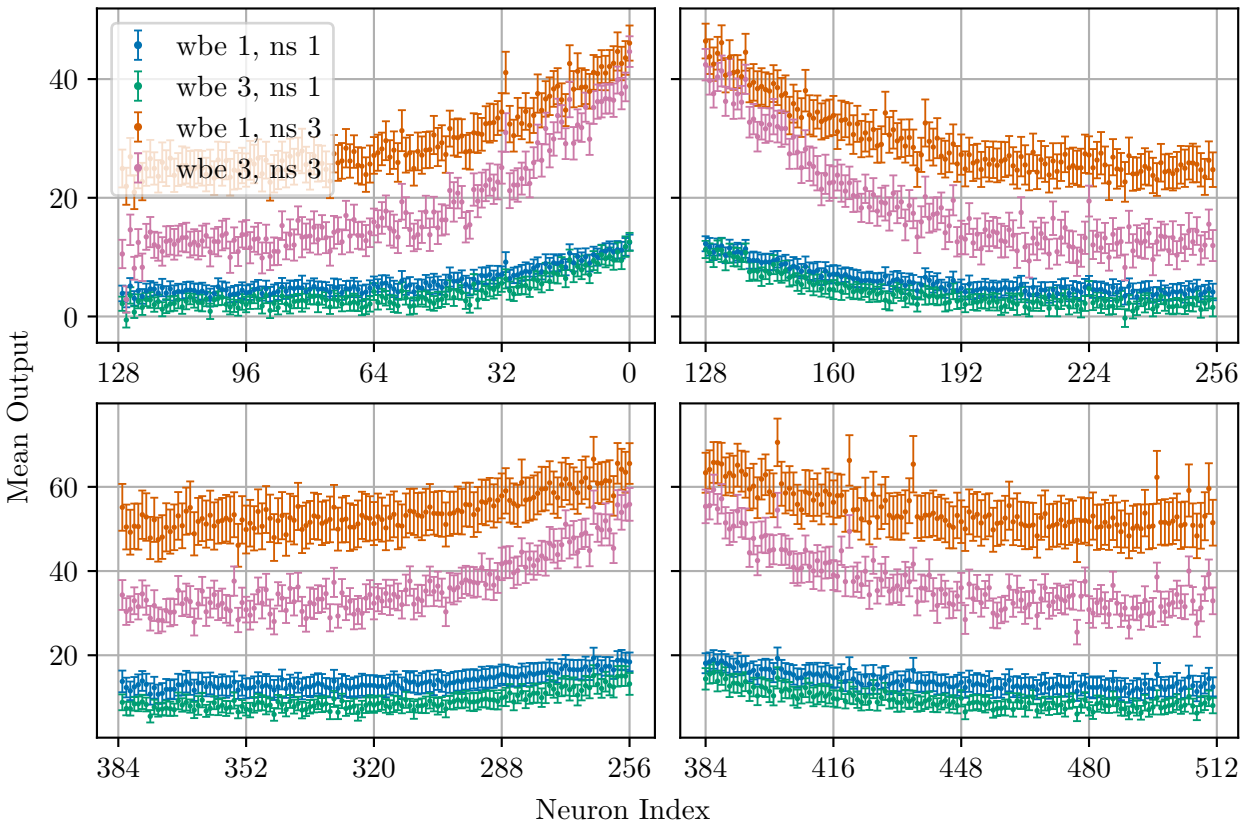

Figure 3.5: Spatial gain dependence for small activations in combination with high weights.

The neuron index is on the horizontal axis while the mean value of each dot product is on the vertical axis with the standard deviation as error bars. The neuron enumeration is done in a way that resembles the layout of the chip with a quadrant in each corner and the lower index of each quadrant in the middle of the chip. The different colors indicate different parameter combinations of `num_sends` and `wait_between_events`.
We see that in each quadrant the neurons closer to the middle of the chip (smallest indices of each quadrant) have a higher mean value and that this effect decreases toward the chip's edge. This trend appears with all parameter combinations, but the difference among neurons is much stronger with an increased `num_sends`. Apparently, this behavior only appears when very small activations in the range of 1 to 5 are multiplied with high weights above 45. The severity of the difference changes gradually and the value range in which this behavior is visible changes with different parameter combinations. For weight-activation combinations outside this range, the neurons behave gradually more uniformly.
In the plot, the spatial dependence looks different in the upper and lower half of the chip. Addi-

tionally, we see that the value range is also different. This means that the chip hemispheres have different gain factors and hence can have different value ranges for the weight-activation combination that makes the spatial dependence visible.
This trend implies that the inherent gain factor of the analog matrix multiplication differs for each neuron. Since the spatial dependence is much less severe for higher activation values the gain also varies for different weight-activation combinations. In the context of machine learning, we assume these offsets to be of minor importance. As we explained in Section 2.1.4 retraining on the hardware should compensate for the static offsets and also the variable gain.

3.3 Noise

The error bars in the previous plot have already indicated that the results of a vector matrix multiplication can vary. This shows the presence of noise in the computation, which is typical for analog devices. We have also seen in Figure 3.5 that the error bars are wider for the parameter combination `num_sends`=3 and `wait_between_events`=1 compared with `num_sends`=1 and `wait_between_events`=1. This implies that there are variable noise sources. In this section, we want to get a better understanding of the noise and find affecting parameters so that we can avoid situations that cause a lot of noise or even optimize parameters so that the resulting noise is particularly low.
Firstly, we want to find out which type of noise affects the result of the matrix multiplication. There are fundamentally two options. The noise might be static and independent of the values used in the computation. Then we would assume the noise is an additive term. The other option is that the noise changes depending on the used values in the computation. A possible behavior could be that the noise increases with higher values in the weight matrix or the activation batch. In this case, we would assume the noise to be a multiplicative term. Of course, both types of noise can appear at the same time and might overlap
In the work of J. Weis [30, p. 36+85] they explain that the noise increases for a longer integration time and that they assume the noise to be a random walk process. This means that the mean distance from the start grows proportionally to the square root of the number of steps [46]. In the present case, the steps are the integration time t, hence the noise should increase proportionally to $\sqrt{t}$. This is however just an assumption without any proof, therefore we conduct an extensive analysis of the overall noise measurable in the results of the analog matrix multiplication.

As a first test, we look at the default calibration and measure the standard deviation of multiple successive executions of the matrix multiplication. To get a measure for the whole chip, we average the standard deviations of each neuron. We do this for different weight-activation combinations to find out whether the noise is sensitive to the input operands. Also, we measure the mean output value of all neurons to check if the output is still in the output range. If the output exceeds this range the noisy result would be clipped resulting in a deterministic output with a standard deviation close to zero. For simplicity, each matrix consists of elements that are identical within their respective matrix but differ between the two matrices. In Figure 3.6 the results of the described experiment can be seen.
In the upper graph, the mean value is plotted while in the lower graph, the averaged standard deviation can be seen. Data points that have a mean output value above 100 and below -100 are excluded to primarily show the behavior without clipping. Even though, the output range is slightly bigger we use this smaller range because some neurons reach the limits earlier than others. Since we plot the average standard deviation this metric starts decreasing when the first neurons start clipping.
We see that the average standard deviation reaches a minimum of 1.1 for a weight of zero independent of the used activation. This represents the additive noise term. In addition, we see that increasing the absolute weight also increases the noise. For positive weights at around 20 and for negative weights at around -18 the noise starts oscillating. If we follow the upper peak values of

each curve, the growth behavior might be quadratic. However, if we follow the lower peak values, it might also increase just linearly. We can definitely see a data-dependent noise term that increases with increasing weights. A possible explanation of the curves' oscillating pattern might be related to the hardware component that produces the synapse currents. The current of a 6-bit weight is created with a combination of 6 binary weighted current sources [23]. An increase of the weight value by 1 can cause a completely different bit pattern and in that case, completely different current sources are used. These variations of the used hardware components might lead to jumps in the curves. If this is the case, we would see similar but vertically scaled patterns for different activations but that behavior is not visible in Figure 3.6. The current sources may react differently on the length of the trigger pulse, with some of them behaving more sluggishly than others.
Even though the growth behavior is not quite clear, it seems like a higher activation makes the noise grow a bit faster. The exception to that is the curve with an activation of 4. Here the standard deviation grows faster than any other curve. The curve with an activation of 10 also grows very fast but only at medium and high weights. It seems like a setting with very high weights and very low activations not only causes the spatial dependence from Section 3.2 to become visible, but it also causes a disproportionate increase in noise.
Another generally observable trend is the different behavior for negative and positive weights. Although in both cases the noise increases for higher absolute weights the increase is slightly smaller for the negative weights. Also, we see the mean values, especially for an activation of 22, bend close to the clipping range at -100. This indicates that some neurons have already reached their lower limit and this also explains why some data points in the standard deviation plot reach lower values at high negative weights compared with the minimum standard deviation at a weight of zero. Since this behavior is not visible for positive weights, we conclude that the negative output range without limiting effects is smaller than the positive output range.

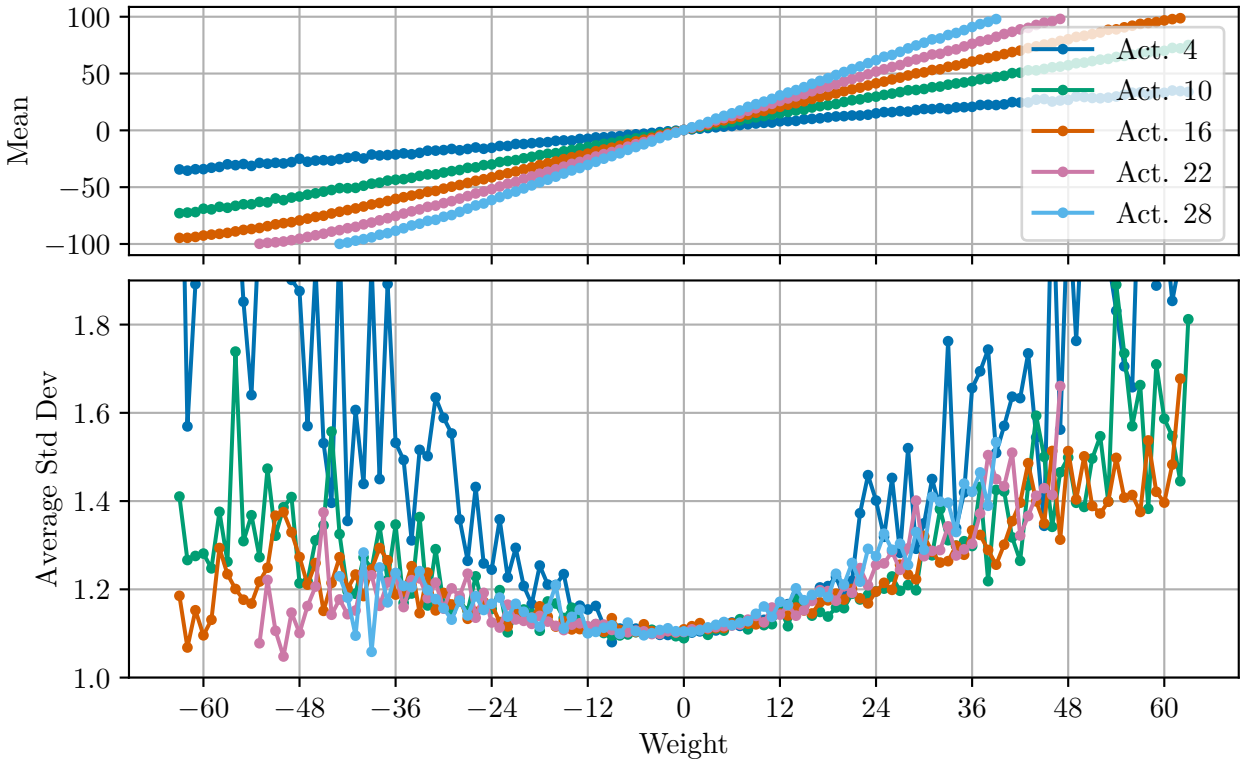

Figure 3.6: Mean and standard deviation over weight. Recorded with the default calibration, 64 input features and `num_sends`=1, `wait_between_events`=3.

In Figure 3.7 we pivot the plot and put the activation values on the horizontal axis with different weights indicated by differently colored lines. In this representation, we can see the special behavior for small activations and high weights even clearer.
The activations start at a value of 1 because all activations with a value of 0 are skipped in the execution so that no activity on the chip can be observed. We mentioned this behavior already on page 13 during the explanation of the functional principles. Similar to the previous plot we exclude data points where the output exceeds a value of 100 and in this representation we only use positive weights. The behavior, however, is very similar for negative weights.

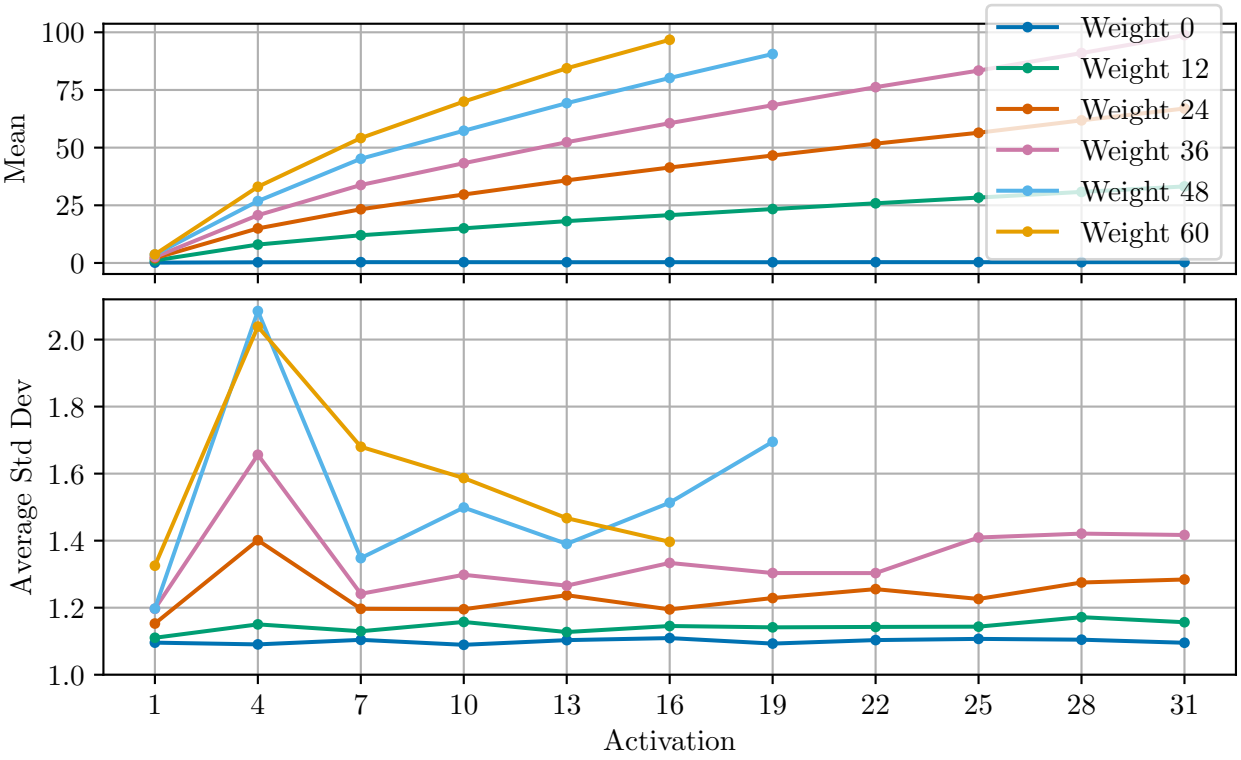

Figure 3.7: Mean and standard deviation over activation. Recorded with the default calibration, 64 input features and `num_sends`=1, `wait_between_events`=3.

At an activation value of 4 all curves with weights greater than 12 have a peak in the averaged standard deviation. However, the standard deviation instantly decreases again at an activation value of 7. The only exception is the yellow curve with a weight value of 60. There the standard deviation decreases up to an activation value of 16. In general, the trend of higher standard deviation for higher activation values is only visible for weights greater than 24 and for an activation interval where the anomaly for these special weight-activation pairs has decreased. Compared to Figure 3.6 the weight's impact on the noise is much stronger than the activations' impact. This strange behavior of small noisy input activations could be related to the inverse behavior of the synapse drivers that we explained in Figure 2.4. Since small activations require a high trigger voltage for the current pulse, it is possible that these very high voltages cause the increased noise.

3.3.1 Additive Component

The previous analysis covered the noise behavior when varying the input operands of the matrix multiplication. Next, we want to find out how the timing-relevant parameters affect the noise. Therefore we change `num_sends`, `wait_between_events`, and the number of input features. As we explained in Section 2.2.1 these three parameters have a significant impact on the total number of required clock cycles for the accumulation. The number of input features $N_{\text{input\_features}}$ determines

how many current pulses must be sent during an accumulation. Each current pulse requires one clock cycle followed by the decay time set by `wait_between_events`. With `num_sends` the process of sending all current pulses can be repeated. Therefore the minimum required number of clock cycles is

$$t_{\text{send}} = (1 + \texttt{wait\_between\_events}) \cdot N_{\text{input\_features}} \cdot \texttt{num\_sends} \tag{3.1}$$

For the complete accumulation, we have to add a small wait period after t_{send} to allow a fair contribution from the last sent current pulses. Besides that, there might be other contributions to the overall accumulations that we don't know about. However, we assume t_{send} to be the dominant part of the overall integration time.
Based on Equation 3.1, we assume that all three parameters linearly increase the time needed for the calculation of the dot product. Hence, with the assumption of the noise as a random walk, we would expect at least a growth rate that follows a square root pattern. However, as we have seen in the previous analysis the weight also influences the noise so there might be additional factors that contribute to the noise and the random walk assumption might not hold. Since we do not know for sure which components actually contribute to the final output noise, we analyze each parameter separately.
We start with an analysis of the additive noise term since it can be easily isolated by setting the values in the weight matrix to zero. As we have seen in Figure 3.6 the noise at a weight of 0 does not change with increasing activations. Therefore we conduct the next experiment with a weight value of 0 and activations of 1 to avoid skipping the input signals. The results of the experiment can be seen in Figure 3.8.

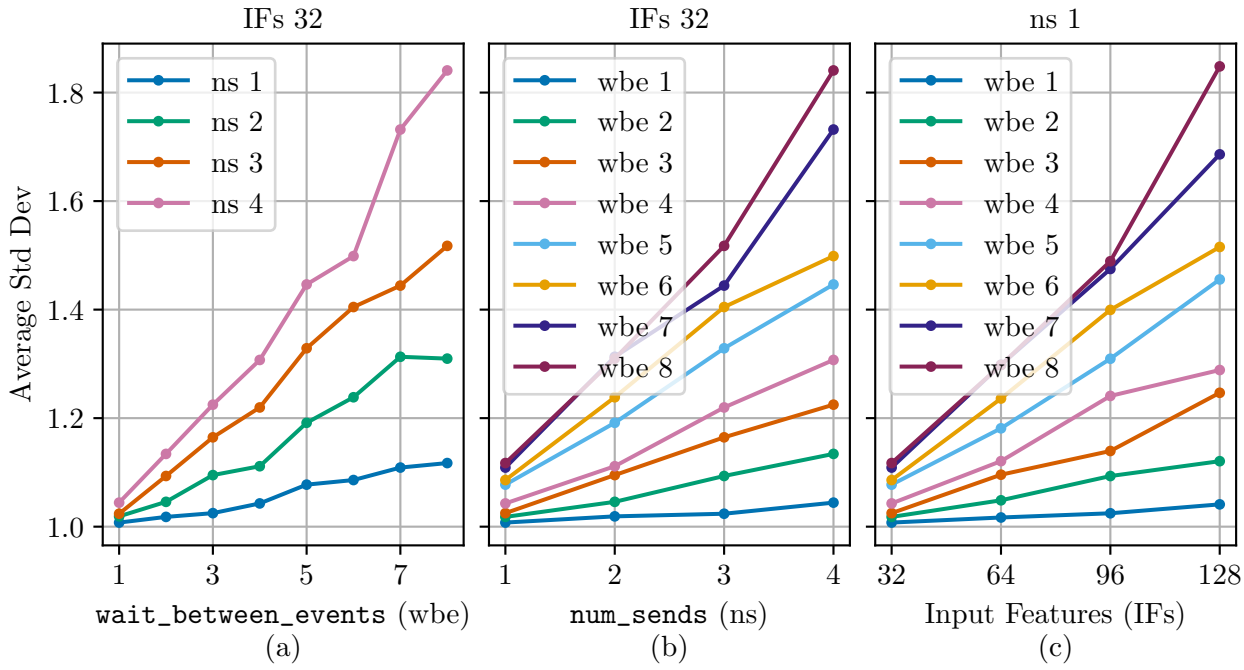

Figure 3.8: Noise over `wait_between_events`, `num_sends`, and the number of input features. Recorded with the default calibration, weights of 0 and activations of 1.

In Figure 3.8 (a) the average standard deviation over `wait_between_events` is depicted. Differently colored lines indicate different `num_sends`. We can see that each curve increases linearly with an increasing `wait_between_events`. Increasing `num_sends` increases the slope of each curve. Because it is not quite clear whether `num_sends` increases the slope linearly, we again pivot the plot so that we have in Figure 3.8 (b) `num_sends` on the horizontal axis and different `wait_between_events` as differently colored lines. We see that the curve approximately increases linearly. Since Figure 3.8 (c) closely resembles Figure 3.8 (b) we conclude that the activation vector width behaves very similarly to `num_sends` and also affects the noise linearly.

Before we continue, we want to justify the parameter choice of an activation vector with only 32 elements. This is only one-fourth of the possible input vector size, but we chose this because we want to compare the effect of `num_sends` with the effect of the number of activation vector elements on the noise. In theory, if we send an input vector of 32 elements twice, the result should be quite similar compared to a single vector sent with 64 elements. Therefore we want to verify that the noise also behaves similarly. To compare more than two values, we choose an activation vector with 32 elements, so we can send this vector up to four times and compare it to an input vector with a length of up to 128 elements. Because of the very similar behavior of Figure 3.8 (b) and (c), we conclude that the parameter `num_sends` and the number of input features indeed have a very similar effect not only on the expected accumulation value but also on the noise. We have seen that all three parameters increase the noise so that part of our assumption is correct. However, instead of square root growth, we see a linear increase in the standard deviation.
We can see this trend even clearer for all parameter combinations if we compute a time score by computing the product of all three metrics and then plotting the standard deviation over this time score. We call this metric time score because it closely resembles the number of clock cycles it takes for the inner product. It is not the exact time because we do not consider the conversion from clock cycles to an actual time and we omit the additional time at the end of an accumulation that we previously mentioned. Since we omit the constant offset at the end we also discard the time in which the drivers are active so that the time score only consists of the product of all three time-influencing parameters.

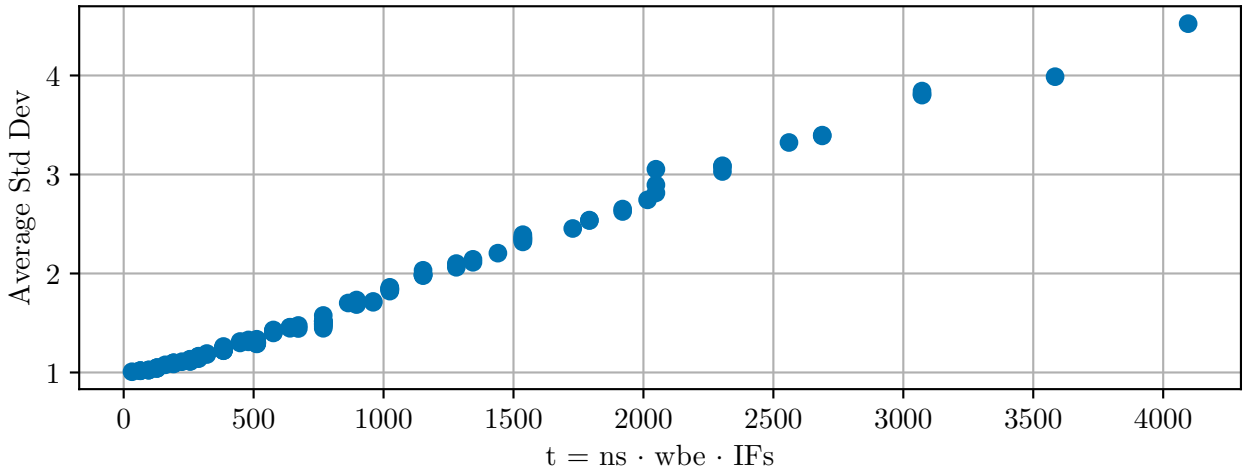

Figure 3.9: Noise over time score. Recorded with the default calibration, weights of 0 and activations of 1.

In Figure 3.9 we see the standard deviation's dependence on the time score. At this point, we want to remind the reader that the average standard deviation is the standard deviation over repetitive matrix multiplications for each individual neuron, which is then averaged over all neurons. It is not the standard deviation among different neurons. This means that the increase in standard deviation over time is not explainable by membrane potential drifts among neurons (this would indicate less uniformity).

3.3.2 Multiplicative Component

Up to now, the linear dependence is only confirmed for the additive noise term. Therefore further experiments are necessary to explore the noise behavior with more chip activity. Unfortunately measuring the multiplicative noise is more complex and by the time of writing, we have not found as clear regularities as for the additive noise. Nonetheless, we want to share our current findings.
With weights different from zero, we want to introduce a slightly modified version of the plot shown in Figure 3.9. Besides the average standard deviation, we also want to include information about

the mean output because, compared with the previous situation, the output will clearly differ for different `num_sends` and different numbers of input features. Because of that, we distinguish different `num_sends` and input feature combinations by using different color and marker combinations in the plot. Markers connected with a line represent points that only differ in the chosen value for `wait_between_events`.

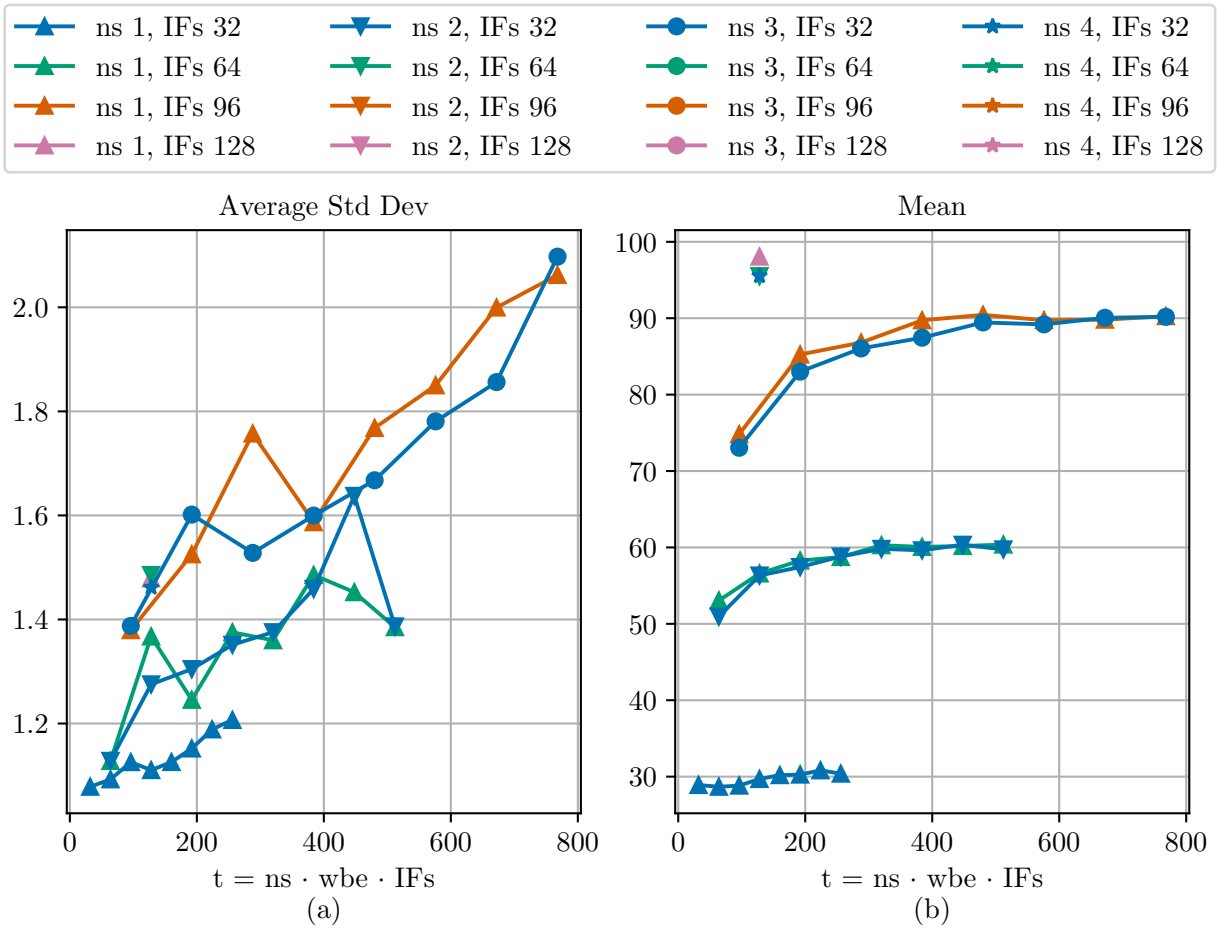

Figure 3.10: Noise and mean over time score. Recorded with the default calibration, weights of 40 and activations of 13.

Our first experiment results with a weight different from zero can be seen in Figure 3.10. In Figure 3.10 (b) we now see the mean output value of all neurons and all repetitions. Similar to Figure 3.8 (b) and (c), we see that sending a vector with 32 elements twice compared to sending a vector with 64 elements once results in very similar mean outputs. The mean output in this scenario also reveals another peculiarity of the analog matrix multiplication. We see that increasing `wait_between_events` also increases the output until it reaches a stable state. This only happens for configurations that result in a medium to high output value and is caused by saturation effects which we will analyze in the following section.

With respect to the standard deviation in Figure 3.10 (a), we see that there is still the general trend of the noise increasing with the time score. In this case, however, we see that different combinations of `num_sends` and the number of input features are offset even though, they share similar time scores. Yet similar to before, combinations where the product of `num_sends` and the number of input features match, behave quite similarly regarding the noise. Because this offset was not present in our experiments with a weight of zero, it must be caused by the newly introduced activity on the chip. The given data provides two possible reasons for the increased noise.

Firstly, increasing the product of `num_sends` and the number of input features increases both the

total time in which the drivers are active and the total decay time between events. The parameter `wait_between_events` would only increase the total decay time. Possibly the time intervals in which the drivers are active allow other noise sources to inject charge to the membrane and thereby increase the overall noise.
Secondly, the increase in output amplitude might force the OTAs to a different operation point. After the first events have increased the membrane potential, the OTAs have to drive the same current per event at steadily increasing voltages. This is the typical application for OTAs, but even though the produced current on average might be the same, the noise of this current signal possibly increases for higher output potentials.
The second theory is contradicted to some extent by the values for `wait_between_events`=1 (the left end of each connected curve). In the standard deviation plot the points with [ns=2, IFs=32] and [ns=1, IFs=64] are extremely close to the [ns=1, IFs=32] curve. But the mean outputs and hence the output potentials are clearly different. The same can be seen for the data points at (180|87) in Figure 3.10 (b). In the mean plot, they are quite far apart from other curves but in the standard deviation plot, they are very close to other data points at coordinates (180|1.5). The data points with `wait_between_events`=1 seem to behave quite differently in some cases.
Also, it is not clear how the saturation affects the noise. We would assume that the saturation forces the OTAs to high output currents, which might lead to higher noise in its signal, but we cannot see any signs of an increase in the standard deviation for low `wait_between_events`. In some cases, it seems like the opposite happens.

As we've seen the linear dependence over the time score no longer holds when we use weights different from zero. But with our previous assumption that the noise consists of a superposition of additive and multiplicative noise, we can simply subtract the additive part and try to find patterns in the multiplicative part.
From Figure 3.9 we estimate the additive noise to be a linear function of the form:

$$g(t) = a \cdot t + c \tag{3.2}$$

$$\text{with } a = \frac{1.9 - 1}{1000} \quad \text{and} \quad c = 1$$

We then subtract the additive noise as a function of the time score from the actual standard deviation with non-zero weights. To avoid the hardly predictable behavior seen in the last figure, we only use activations above 10. If we then again analyze how the noise behaves depending on the different time parameters, we get the results shown in Figure 3.11.
In Figure 3.11 (a) we see that the multiplicative noise doesn't depend on `wait_between_events`. The slightly negative trend is caused by the linear fit to the additive noise, which predicts slightly too high values for very small time scores. However, we see that an increase in weights causes an offset in the additive noise.
If we consider the results in Figure 3.11 (b) and (c), we see that the multiplicative noise also increases with `num_sends` and the input feature count. Higher weights then increase the slope of the noise. Considering this information and that we have in Figure 3.11 (a) 32 input features and `num_sends` of 1 the offset increase with bigger weights might also be deduced to only `num_sends` and the number of input features.

For completeness, we also show the results of varying the activation with constant weights. This can be seen in Figure 3.12.
The general trend is very similar to the trend seen in Figure 3.11. Here as well, the parameter `wait_between_events` seems to have no impact on the multiplicative noise while the other two parameters increase it. In both cases the growth rate is not clear and might be somewhere between linear and quadratic and in both cases there are outliers to the trend that cannot be explained with the current knowledge.

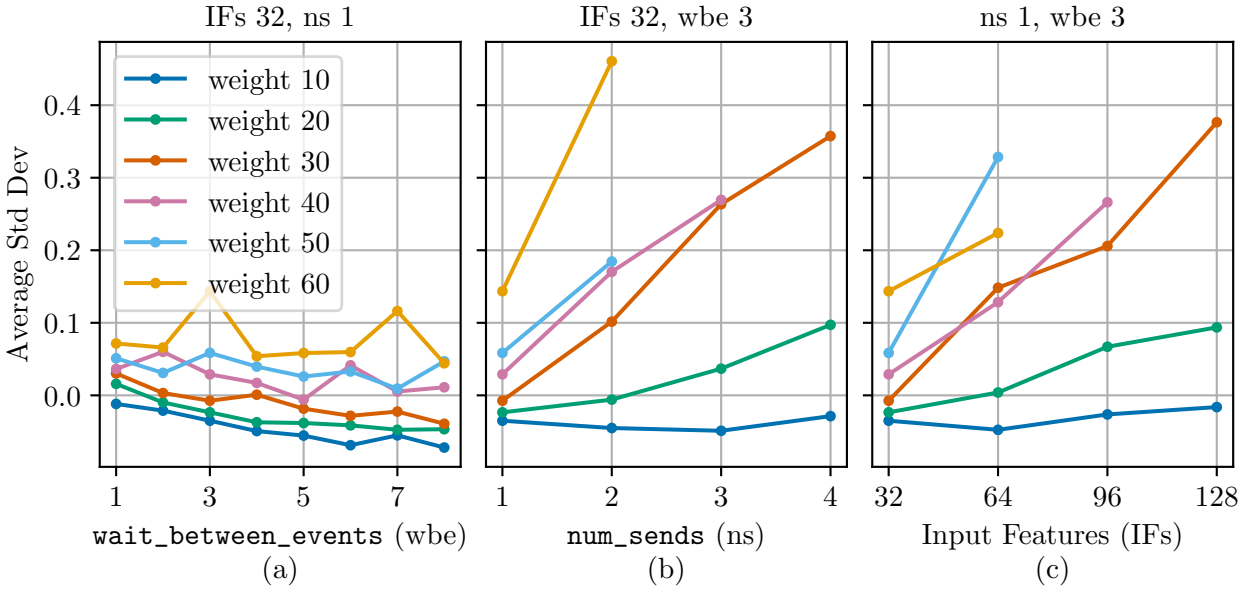

Figure 3.11: Noise over `wait_between_events`, `num_sends`, and the number of input features. Recorded with the default calibration, and activations of 16.

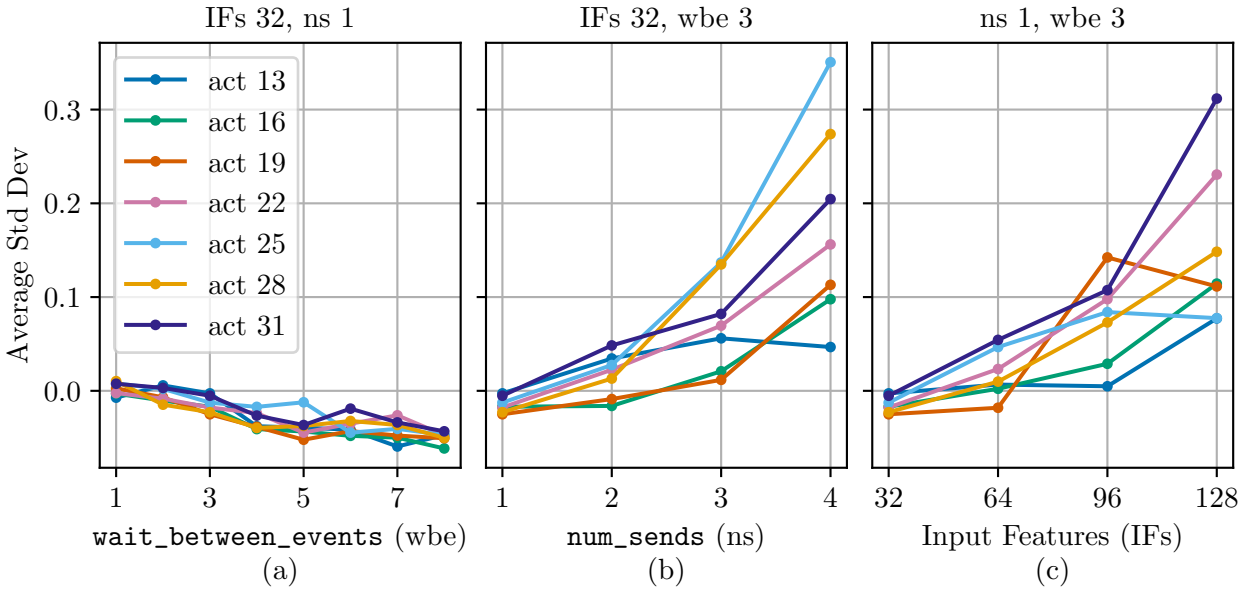

Figure 3.12: Noise over `wait_between_events`, `num_sends`, and the number of input features. Recorded with the default calibration, weights of 18.

Another type of noise can be found again in a scenario with very low activations and high weights. In the previous experiments when varying weights and activations, we have seen that low activations do not produce very reliable output results, especially in combination with high weights as seen in Figure 3.7. When varying the timing parameters this behavior is even more unstable.

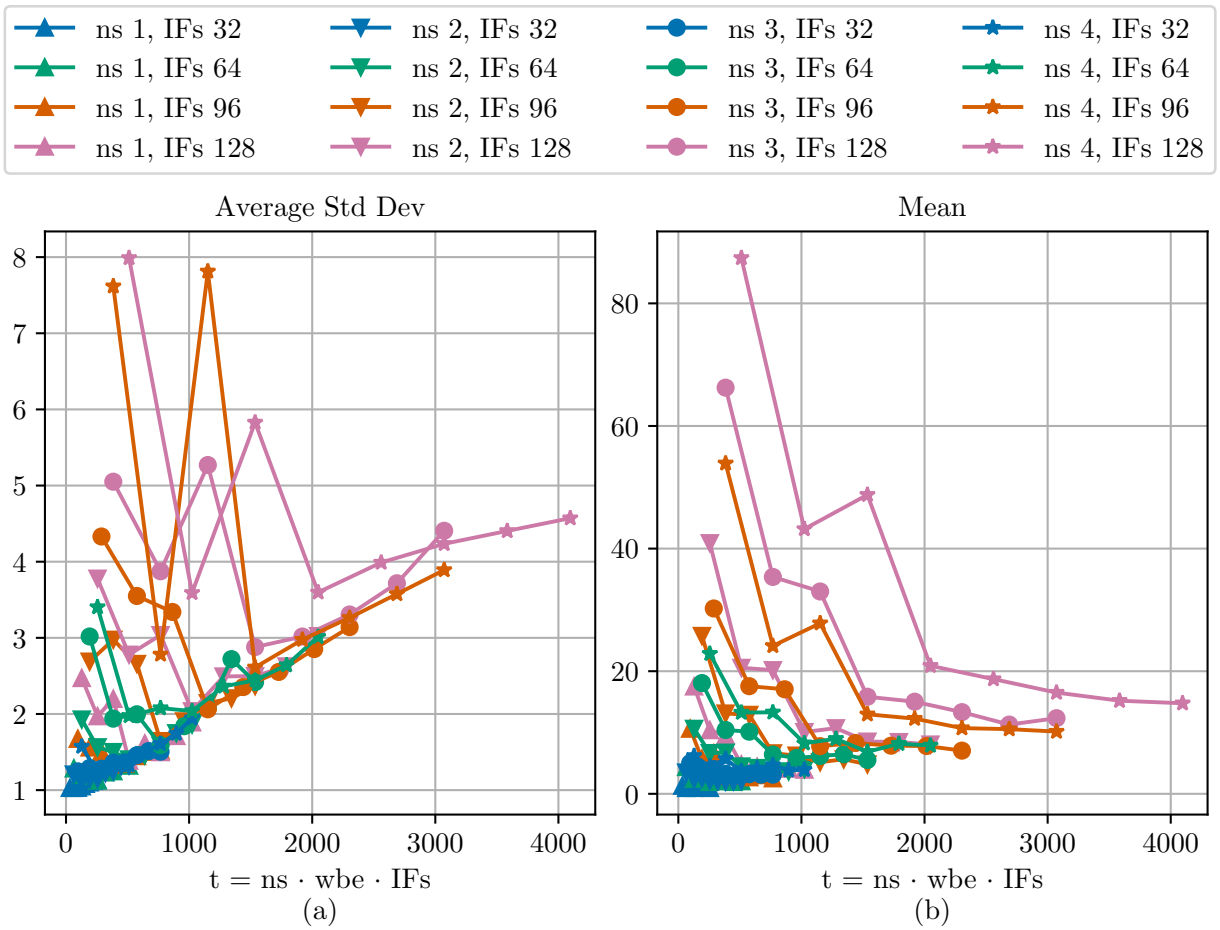

Figure 3.13: Noise and mean over time score. Recorded with the default calibration, weights of 40 and activations of 1.

In Figure 3.13 the results with weights of 40 and activations of 1 can be seen. What is special about this result is that a high `wait_between_events` can significantly reduce the mean output and at the same time reduce noise so that it gets closer to the linear trend that we have seen already in Figure 3.9.
This is quite unintuitive. In general, we would assume that very long `wait_between_events` can reduce the output because the time increases and leakage on the membrane can become relevant. But a decrease of the mean output by the amounts seen in Figure 3.13 is unlikely caused by leakage. It is more likely that a very low `wait_between_events` causes the disproportionately high outputs. The reason for that is unclear.
For very low activations the synapse drivers activate the synapses only for extremely short time intervals. Also because of the inverse design of the synapse drivers, small activations require high trigger voltages that could further increase the instability of the trigger signal. In combination with high weights, this instability becomes even more visible because a missing event with high weights causes a bigger difference in the output than a missing event with small weights. But since the `wait_between_events` parameter should only affect the wait time in between the events, there is no clear reason why low values increase the mean output and standard deviation that much.

3.3.3 Noise Takeaways

After the series of experiments, we want to recap what we just found and which possibly good parameter configurations we see in the context of machine learning from a noise perspective.
We have seen that all three time-relevant parameters `wait_between_events`, `num_sends`, and the number of input features increase the additive noise linearly with a constant offset that cannot be undercut. Both constant and linear parts of the additive noise contribute significantly to the overall noise.
For the multiplicative noise, both weights and activations increase this part in combination with the parameter `num_sends`, and the input feature count. Therefore it would make sense to keep all three time-relevant parameters as low as possible to reduce the noise to a minimum.
However, the number of activations is given by the network architecture and therefore hardly adjustable. But with the current execution model of the chip, which skips input vector elements of zero, this property could be used to increase the number of zero elements in the input activation to reduce the noise. This can be leveraged for example by using the ReLU as an activation function that automatically sets a great part of all activations to zero. A possible extension can be pruning techniques that set whole output activations to zero. Activation pruning based on magnitude has also the advantage of avoiding or attenuating the problem with low activations combined with high weights.
Low `num_sends` seems to be similarly effective as a low number of elements in an activation. This parameter seems to be the most flexible. The main constraint is that the output gain must be sufficient so that small activation changes cause a measurable change in the output activation. This gain can be raised with an increase of `num_sends`.
From the noise perspective, it would also make sense to use small activations and weights to reduce the multiplicative noise. Yet to avoid big quantization errors, we want to use a wide value range. Therefore using only small weights and activations is not a feasible modification.
The `wait_between_events` parameter seems to affect only the additive noise and should also be chosen as small as possible. The reason to choose higher values than one is to avoid saturation of the synaptic inputs and to avoid unpredictable behavior when using low activations with high weights.
Overall we see that there is no obvious solution since there are good reasons against just using the minimal possible value for the mentioned parameters.

3.4 Saturation

Another peculiarity that we have already seen in the last section is the saturation of the synaptic input. It occurs when the voltage difference of the synaptic input becomes too large, leading to a change of the OTA gain. In Figure 3.14 the effect of the synaptic input time constant and `wait_between_events` is depicted. For smaller time constants the voltage difference decays faster so that the decay interval determined by `wait_between_events` can be chosen smaller without the risk of saturation. As described in Section 2.2.1 small time constants also decrease the overall gain of the operation so it cannot be configured arbitrarily small. If the synaptic input time constant is increased to raise the overall gain, `wait_between_events` must be increased as well. Otherwise, saturation will occur if no other countermeasures (like smaller inputs) are being taken.

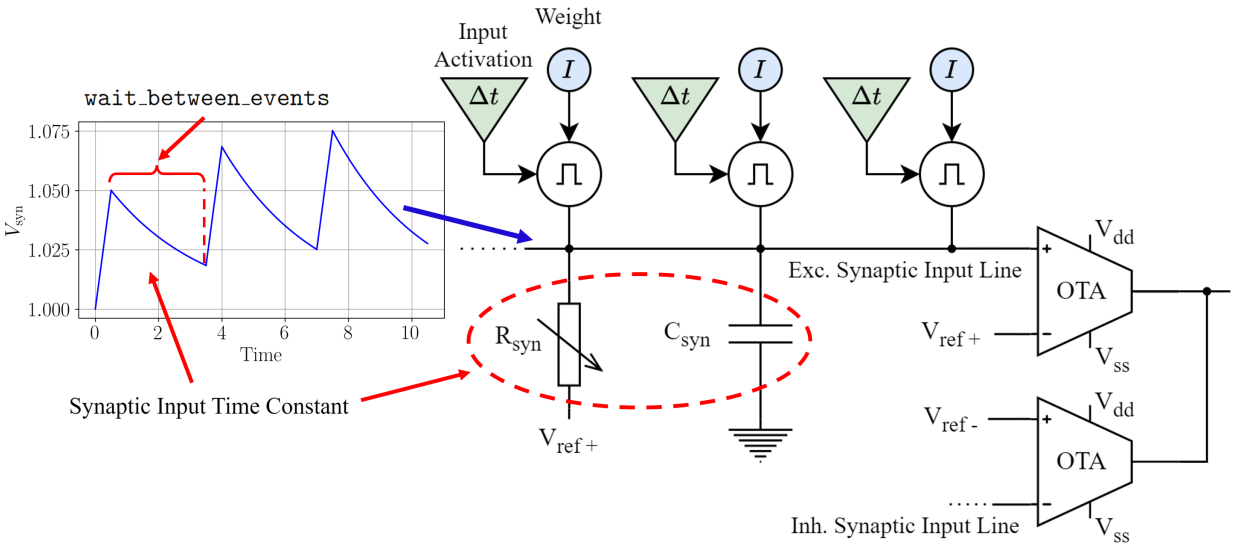

Figure 3.14: Simplified schematic with the effects of `wait_between_events` and the synaptic input time constant on saturation[1].

According to [30, p. 9], there are two reasons for the saturation. Firstly, the synapse current decreases if the voltage on the synaptic input line reaches a voltage of 1.0 V, and secondly the emitted output current of the OTA saturates. In both cases, not enough charge reaches the membrane capacitor making the absolute result of the accumulation too small. This means the saturation depends on the magnitude and accumulation order of the partial sums in the dot product.
Unfortunately, the magnitude of a partial sum depends on the specific weight-activation pairs. Therefore saturation might occur only with certain activation-weight patterns. For example, if the activation vector contains many high values that appear in dense clusters it is more likely that saturation will appear than if the values are more evenly distributed. Alternatively, a pattern in the input vector that interleaves the high-valued clusters with a few small values would allow the voltage on the synaptic input line to decay and thereby attenuate saturation.
In [30, p. 9] J. Weis already describes tradeoffs to mitigate this problem. Both presented options involve the reduction of the signal on the synaptic input line. The first option is a decrease of the overall current from the synapses to lower the amplitude of the voltage pulse on the line. The second option is the reduction of the synaptic time constant so that each voltage pulse decays faster. Both options reduce the amount of current emitted by the OTA and hence reduce the membrane voltage. This is unfavorable because we want a strong output signal clearly distinguishable from background noise. If the output signal decreases, the signal-to-noise ratio (SNR) decreases too thereby making it harder to distinguish actual signals from noise.

[1] Similar to Figure 2.3 the synapse current actually decreases the voltage on the synaptic input line. Additionally, the voltage trace is not measured but only a qualitative model.

Saturation has the disadvantage that the results change for certain input combinations and the appearance of saturation is hardly predictable. As a compromise, the current calibration chooses the synapse current as high as possible while the synaptic time constant is chosen very small. This has the small advantage that the voltage after the last event decays faster. This allows for reduced waiting periods and thereby speeds up the overall execution.
Because it is unclear at which value saturation appears and how severe the impact of the saturation is, we conduct experiments to measure its effect. To get an estimate of the saturation's severity, we want to measure it in the worst-case scenario with maximal saturation and the best-case scenario with saturation as low as possible. Even though the saturation depends on both the weight and the activation in our experiments, we only vary the activation vector to create different settings. This makes the results more easy to interpret since only one variable changes.
Fundamentally it is possible to produce the desired scenarios with any of the two vectors of a dot product, but since it is faster to send a bigger batch than many different weight matrices, we choose to keep the weight matrix constant.
If the weight matrix consists of equal values, the dot product formula for a single neuron output:

$$c = \sum_{i=0}^{n-1} a_i \cdot w_i \tag{3.3}$$

simplifies to:

$$c = w \cdot \sum_{i=0}^{n-1} a_i \tag{3.4}$$

where n is the length of the activation row and the weight column. Therefore the output is proportional to the sum of the activation values and the single weight value.
In the following experiments, we steadily increase the activation sum, however for different activation vector patterns and different values of `wait_between_events`. The weight magnitude is mostly constant and only adjusted to get an impression of how severe saturation can become and for which weight-activation pairs it occurs.

3.4.1 Dependence on Input Order

In a digital scenario, any permutation of the activation vector that has the same vector sum will produce the same output result (associativity of the addition) but in our analog setting with saturation, this assumption is no longer valid.
In the worst-case scenario, the activation vector is filled from one side with high values without any gaps that would allow the voltage on the synaptic input line to decay. Figure 3.15 illustrates this activation pattern. In the upper row, the activation vector is barely filled. Most of the elements are zero and we assume no saturation. Now we increase the filling degree and fill the activation vector from left to right. In the middle row, the activation vector is partly filled and possibly saturation starts to occur while in the lower row, the activation vector is almost completely filled and the activations are most likely affected by saturation.
Even though the activation vector's elements can have values of up to 31, we fill the elements only to a value of 20 to mimic an average activation magnitude that produces moderate outputs.
If we linearly fill the activation vector in the described fashion we would assume the output to increase linearly until we reach saturation and the slope decreases.
In the best-case scenario, we create an activation pattern similar to the one shown in Figure 3.16. Instead of the activation vector being filled from left to right, we divide the vector into groups that are then equally filled from left to right. This interleaves the high-valued clusters with gaps that allow the voltage on the synaptic input line to decrease. Because the chip skips vector elements of zero in the accumulation, we have to insert minimal values in the gaps so that there really is a time interval in which the voltage can decay. In the worst-case scenario, we don't do this because there is no following cluster that could benefit from a decay.

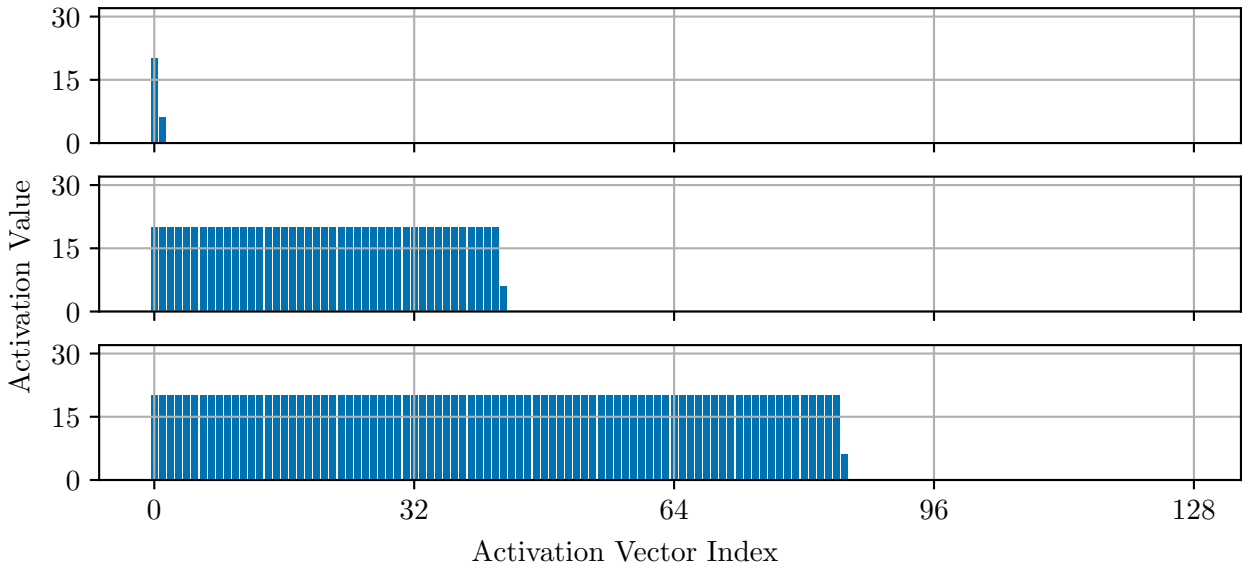

Figure 3.15: Activation pattern to create maximum saturation.

The pattern has similarities to a pulse-width modulation (PWM) signal. The number of vector elements in a group corresponds to the PWM period, while the filling degree corresponds to the duty cycle. The amplitude of the signal is the upper limit of the filling value.
With that analogy in mind, the worst-case scenario describes a PWM signal with a maximal period while the best-case scenario resembles a signal with a very small period. It is unclear whether the smallest possible period is the best option to avoid saturation or if slightly bigger gaps achieve better saturation suppression. The smallest possible period would be a group size of one, meaning that each vector element would be increased at the same time so there are again no more gaps.

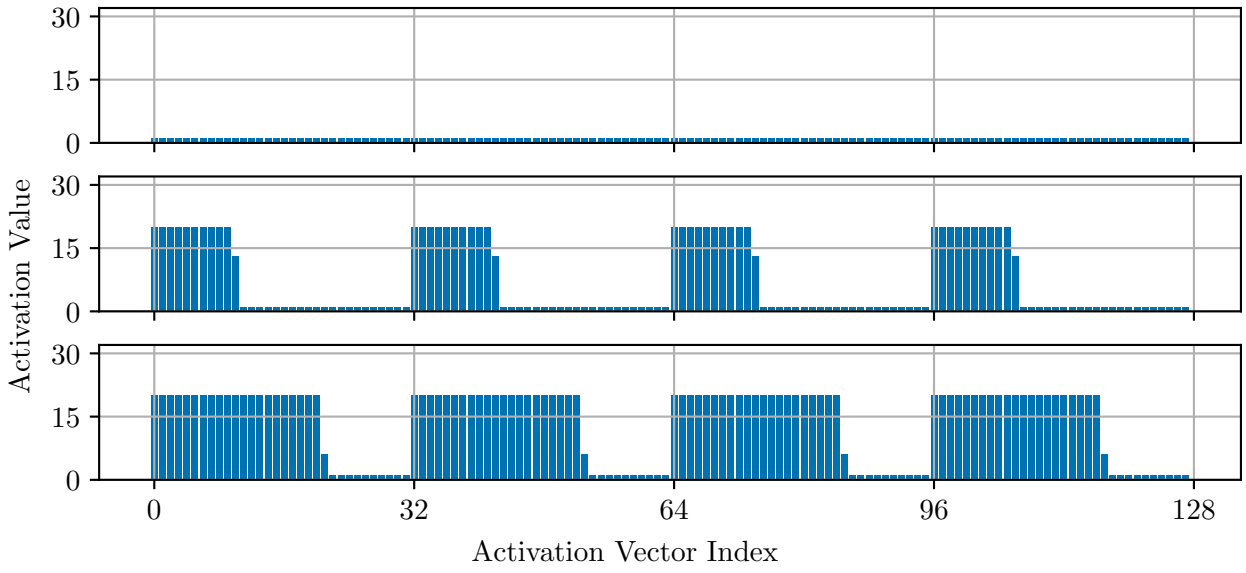

Figure 3.16: Activation pattern with a group size of 32 to spread activations and avoid saturation.

To assess the impact of the group size on saturation, we conducted experiments where we varied the group size but during the experiments, we noticed a limitation of our experiment that made the interpretation of the results slightly harder.
In Figure 3.17 we see the result of our experiment in a scenario where no saturation occurs. The reason for that is the small synaptic input time constant of the default calibration in combination

with a sufficiently big `wait_between_events` and sufficiently small weights and activations.
In this scenario, we use different very small group sizes and steadily increase the filling degree of the activation vector (which makes the sum of vector elements bigger). As a reference, we also included the pattern of the worst-case scenario with a group size of 128.

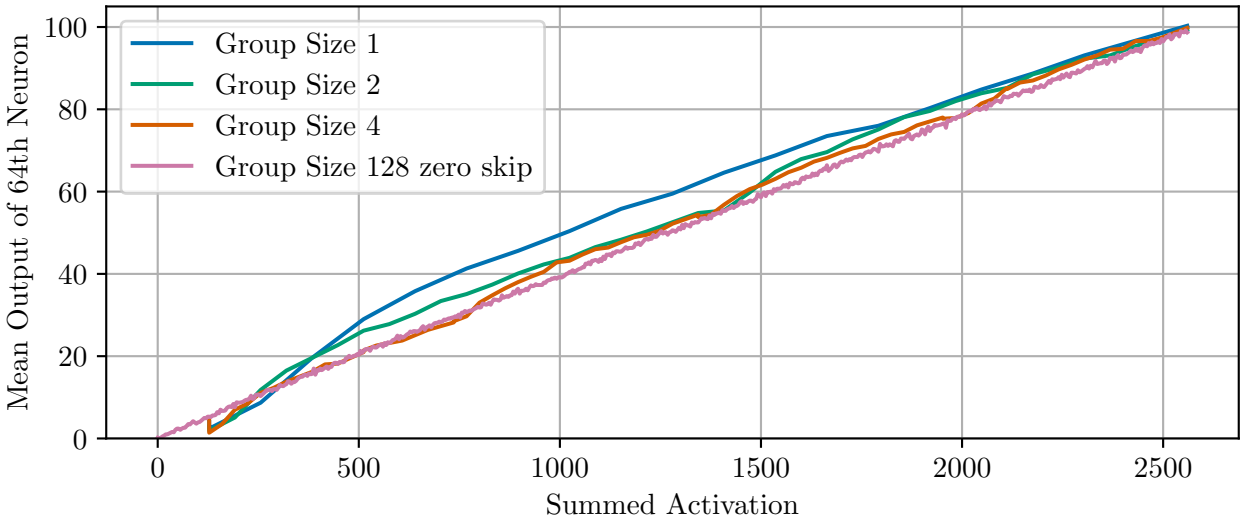

Figure 3.17: Saturation effect for different group sizes (differently interleaved input orderings). Recorded with `weight`=28, `num_sends`=1, `wait_between_events`=2 and the default calibration.

Firstly, we see that the curve with a group size of 128 starts at a summed activation of 0 while the other curves start at a summed activation of 128. This is caused by the fact that we fill the activation vectors with a group size smaller than 128 with at least a value of 1 to avoid skipping the values in the desired gaps. This means that the minimum value with that pattern is 128.
Secondly, we see that at the start the curves with a small group size produce very small outputs below the output of an activation with a group size of 128 and skipping of zeros. The reason for this is that activations of 1 are so small they do not correctly trigger a synaptic event.
When further increasing the filling degree, we see that a lower group size achieves higher results. But we also see a pattern in the curve shape. The curve with a group size of 1 always runs above the curve with a group size of 128 and only touches it at the end where the activation patterns match. It forms a bump over the other curves. In the same range the curve with a group size of 2 forms 2 bumps and the curve with a group size of 4 forms 4 bumps. Two in the upper half and 2 barely visible ones in the lower half.
What we see here is the non-linearity of the activations that we have already seen in more detail in Figure 3.7. The start of each bump represents a new element in the group that starts linearly increasing to the maximum group value. For small group sizes (short PWM periods), we have many groups and thereby many small values that increase at the same time, thus the non-linear increase becomes visible in the output. This means the number of vector elements in a group determines how many non-linear increases (or bumps) we see in the output and the smaller the group size the more severe the non-linear increase.
To avoid mixing up static non-linearities with dynamic saturation, we further consider only group sizes of 8 and above since the bumps are already barely visible for a group size of 4.
With this consideration, we repeat the experiment with bigger group sizes and with a configuration in which saturation actually occurs. Our assumptions however remain the same. We assume the curve with a group size of 128 to be the worst-case scenario where we reach a certain activation sum after which the slope of the curve decreases, which indicates the presence of saturation. With smaller group sizes we assume the point after which the saturation occurs to increase and possibly the slope reduction to decrease. Also, we still assume a zero-biased result for small activation sums if we fill the gap values with ones.

In Figure 3.18 we repeat the experiment with an increased weight and small `wait_between_events`. We now see that the curves have a similar slope in the interval between 0 and 400, but then the slope of some curves decreases. The slope is less strongly decreased for smaller group sizes. This confirms our assumption and shows that more gaps between high-valued clusters in the activation vector are beneficial to avoid saturation. Also, it shows that the smallest group size of 8 achieves the best saturation mitigation, but the difference between a group size of 8 and 16 is very small. If we look for the biggest difference among the curves, we see that differences of up to 25 at a summed activation of approximately 1200 occur. However, we have to consider that these results have been produced by using the maximum value for the weights with a quite small `wait_between_events` and purely positive weights. If we assume the weights are evenly distributed and around half of the values are positive while the other half is negative, the saturation effect would be reduced because the charge is distributed over both synaptic input lines.

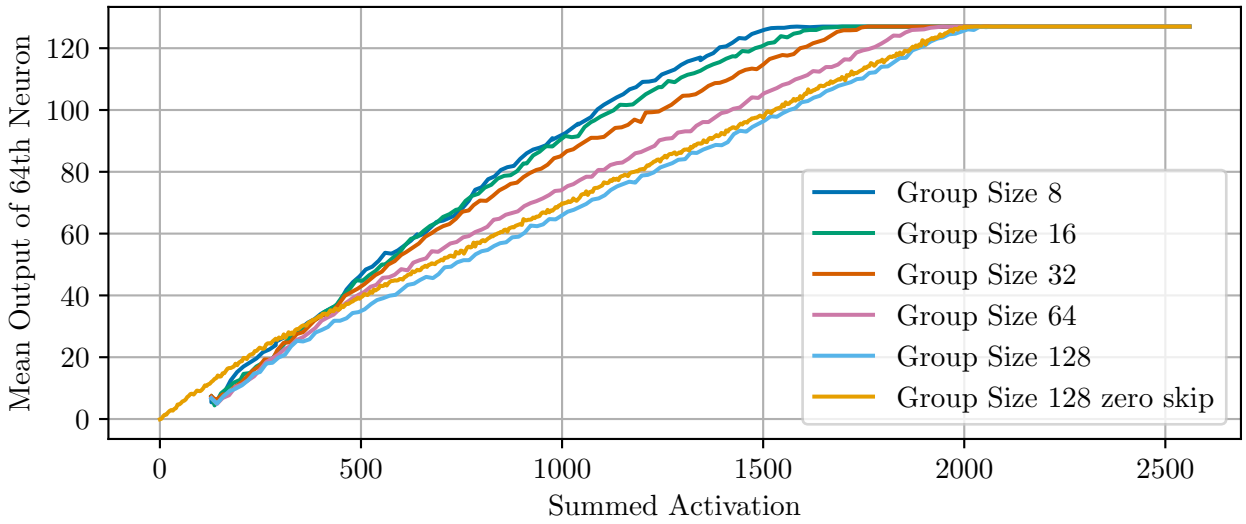

Figure 3.18: Saturation effect with different group sizes (differently interleaved input orderings). Recorded with `weight`=63, `num_sends`=1, `wait_between_events`=2 and the default calibration.

3.4.2 Dependence on `wait_between_events`

Besides the activation pattern and weight, the most relevant parameter that influences saturation is `wait_between_events`. Its main purpose is to control the time interval between successive activation events so that the voltage on the synaptic input line has enough time to decay and no saturation occurs.

However, we have seen in the last section that at least the additive noise term increases proportionally to `wait_between_events`. To also get an impression of the effect of `wait_between_events` on the saturation we conduct further experiments in which we vary this parameter.

Figure 3.19 combines the results with 4 different weights in combination with `wait_between_events` between 1 and 5. In Figure (a) the worst-case scenario is depicted while Figure (b) shows the best-case scenario with a group size of 8.

The lowest set of curves with variable `wait_between_events` is recorded with a weight of 3 and shows no sign of saturation. Because of the low weight only very small output values can be reached. In both cases, all curves are extremely close together.

The next highest set of curves is recorded with a weight of 13. Again both cases look very similar but the curve with a `wait_between_events` of 1 is slightly below all other curves. Also, we can see in both cases that the results with a higher `wait_between_events` achieve a higher output.

The third set of curves from the bottom is recorded with a weight of 28 and shows moderate saturation. Again in both cases, the curves with a `wait_between_events` of 1 are further apart

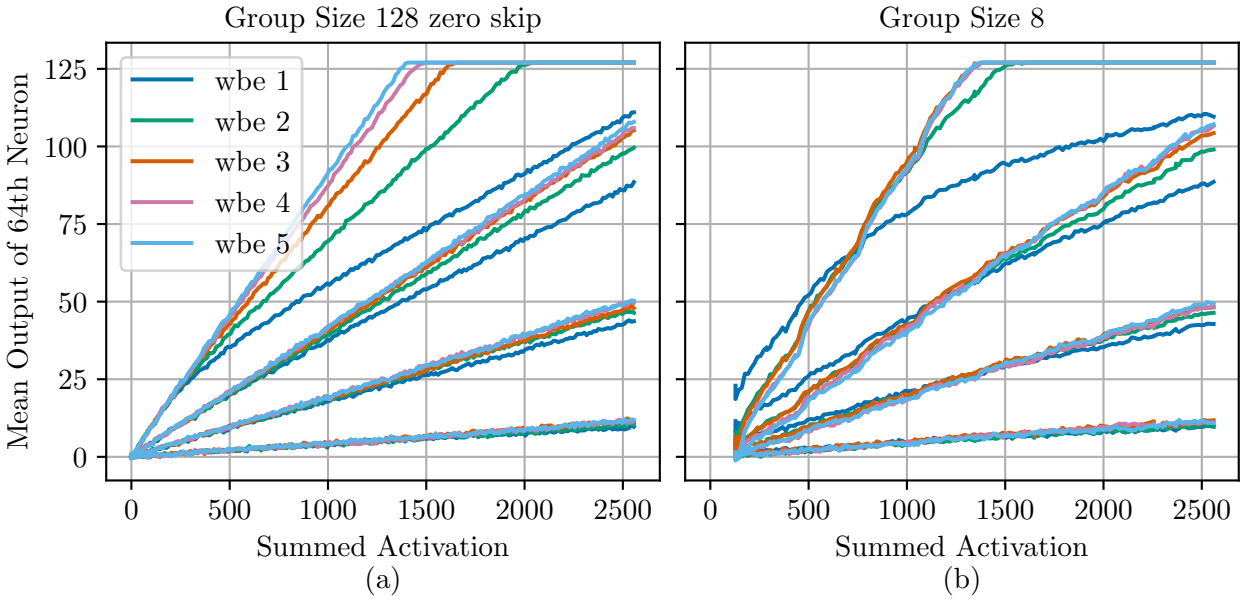

Figure 3.19: Worst-case vs. best-case saturation for the weights {3, 13, 28, 63} (higher weight ⇒ higher initial slope) with default calibration.

from all other curves with a higher `wait_between_events`. With this weight, we see a significant difference between the two cases. For a group size of 128, the curves start separating for summed activations between 500 and 1000, while in the best-case scenario with a group size of 8, this interval has been shifted to an increased range between 1000 and 1500.

For very small summed activations we see that the curve with a small `wait_between_events` has significantly higher values than the other curves with a higher `wait_between_events`. This is an anomaly that we have seen already in the mean plot of Figure 3.13. Also, the wave pattern that we previously deduced to the non-linear influence of growing activations is slightly visible for high weight values even with a group size of 8.

If we increase the weights one last time to a value of 63, we see that all curves of the group clearly separate in the worst-case scenario. The separation point is at a summed activation of around 300 and we can even see that the curve with `wait_between_events` of 1 is parallel to the curve of the next lower weight with `wait_between_events` of 1. We assume that we reached maximum saturation in both cases where only the increase in accumulation time caused by the additional non-zero activation element allows further growth of the output. However, we can observe that even in the worst-case scenario a `wait_between_events` of 5 can almost completely mitigate the saturation since the curve exhibits a smooth transition without any significant change in gradient. It reaches the upper output limit at a similar summed activation as the best case.

In the best case, almost all curves except the curve with a `wait_between_events` of 1 stay close together and reach the upper output limit at similar points. This means that barely any saturation occurs before reaching the output limit and implies that the necessary number of `wait_between_events` can be reduced by accumulating the partial sum of each dot product in a beneficial order. Unfortunately in the context of machine learning, this doesn't seem like a trivial task. We would have to estimate the size of the partial sums and based on that estimate resort the rows of the weight matrix in the same way as the columns of the activation vector.

Even then it is not clear whether an optimum for all neurons in a single matrix multiplication can be found. While the reordering of the weight and activation pairs might reduce the saturation of one neuron, the same permutation could increase the saturation of another neuron. Since the chip reaches full utilization only with matrices of 512 at least columns, splitting the matrix to reduce the reordering dependence is unfeasible as well due to the resulting runtime increase.

3.4.3 Saturation Takeaways

After the saturation experiments, we provide a brief wrap-up of the most important findings. Fundamentally we found that measuring saturation quantitatively is difficult because the severity of it can be vastly different and other non-linear effects might interfere.
Regarding the input order, we found that the activation pattern can have a noticeable impact on saturation. For our constructed worst-case ordering saturation effects occur for `wait_between_events` below 4, however for the best-case ordering `wait_between_events`=2 is almost enough to completely compensate for saturation effects.
Rearranging the inputs, however, appears as a difficult strategy to avoid saturation since an optimal reordering for one matrix column might result in a bad pattern for other columns.
As a general result, we found that the default calibration is very robust against saturation and even small values for `wait_between_events` attenuate saturation quite well. This also applies to very bad input orderings.
Our results show that there is barely any saturation for moderate weight and activation values. Since we expect on average many small weights and activations for typical neural networks, we assume saturation to barely affect the prediction performance.

Chapter 4

Artificial Neural Networks on BSS-2

In this chapter, we focus on the execution of ANNs on BSS-2. In particular, we want to find good calibration parameters for the hardware. Besides that, we explore different mapping strategies for the inputs of the matrix multiplications inside our network to the integer inputs used by the analog matrix multiplication. We verify our approaches by training a small MLP on the hardware so we can compare the achieved accuracy.

4.1 Considerations for Optimized ANNs

If we want to optimize the execution of Deep Neural Networks (DNNs), we have to consider what they want to achieve. Fundamentally DNNs are universal function approximators [4]. They are used to predict a certain property or behavior from an input that contains the relevant information for the prediction task. However, the function that translates the input to the prediction is unknown and should be learned by the network. Hence their aim is to reach a good approximation of an arbitrary underlying function so that the model learns an accurate and generalized prediction rule. We measure the prediction quality with the accuracy metric that counts how many of all samples in an unseen data set have been classified correctly. Ideally, we reach an accuracy of 100 % but there are many factors that limit the achievable accuracy. Typical difficulties are the choice of a good model architecture, good hyperparameter choices, sufficiently big and accurate datasets, etc. In the context of this work, we see the imperfections of the hardware as a limiting factor of the achievable accuracy. We identify noise and saturation effects as the possibly strongest limiting factors of high accuracy. In contrast, we assume other imperfections of the hardware to be of minor importance. The low resolution of the operation does not seem to be too problematic since the work of L. Kuhn [41] already showed a very small accuracy loss for a very similar model.
When it comes to the non-uniform behavior of the components that we have seen in Figure 3.5, we assume that the remaining differences after calibration can be compensated by hardware-in-the-loop training (Section 2.1.4). Especially, static offsets and differences in gain should be easy to compensate for. The seen non-linearities appear so small that we assume a linear approximation causes only small differences in the backward pass. The non-linear behavior of the matrix multiplication is not very pronounced and is limited to a slightly reduced gain for large input operands.
The influence of the hardware's temporal drift is an aspect that we haven't covered so far. We noticed a slight deterioration in accuracy weeks later compared to the accuracy immediately after training. Even though a few epochs of retraining can usually restore the accuracy, this is a significant problem. If the accuracy of the model decreases over time, the accuracy must be monitored, which is a waste of energy. Retraining large models might be expensive and prohibits long continuous operation in a production environment without the significant overhead of having a redundant system. In the scope of this work, we do not consider temporal drift as an optimization point and only focus on the achievable accuracy right after training. The effects causing the temporal drift are poorly understood. However, considering the time frame of a master's thesis a meaningful analysis seems out of scope.

With that, the remaining imperfections are noise and saturation. Noise is problematic for the model since it makes the outcome of the matrix multiplication non-deterministic. It makes training the model much harder because an optimization step might be too big, too small or even in the opposite direction because of the influence of noise in the forward pass. The presence of noise in machine learning is not new, as we have described in Section 2.3. Little noise is actually beneficial for training [33] since it acts as a regularizer. There are also approaches that can tolerate noise to some degree [36]. Basically, any optimization algorithm that implements any form of momentum reduces the impact of immediate disturbances caused by noise. However without specific hardening techniques for neural architectures to avoid the interference of noise, it will reduce accuracy. Therefore we want to find ways to reduce the noise in the computation so that its effect is minimal. From the analysis in Section 3.3 it can be seen that the relevant parameters influencing the noise intensity are:

- operand magnitude
- `wait_between_events`
- `num_sends`
- number of non-zero input features

The increase of any of the parameters also increases the noise. From personal communication with J. Weis (also see [30, p. 35]), we were able to find out that the OTAs in the analog circuit are significant contributors to the noise as well. The reduction of the gain could also further reduce the noise. The excitatory and inhibitory OTAs however should still have similar gains. This suppresses common mode interference on the supply rails or the synaptic input line. Based on the observations of the multiplicative noise in Section 3.3 it is also quite likely that a reduced synapse bias current can reduce the noise. By lowering the synapse bias current the maximum emittable current of each synapse is reduced. This would lower the activity on the chip and hence probably further reduce the multiplicative noise.
Reducing all the previously mentioned parameters also has drawbacks. A shared property of almost all parameters is that their reduction would also reduce the output signal of the matrix multiplication.
In our opinion, an optimized calibration tries to reduce the noise by reducing the OTA gain in combination with a low `num_sends` and as low as possible `wait_between_events`. Reducing the operands could be achieved by regularization techniques but since the mentioned changes already reduce the output signal, we don't want to further decrease it. The number of input features to a layer is given by the network architecture and shall remain a flexible parameter. The use of the ReLU function however will likely set many activations to zero and thereby have a beneficial impact on the noise.
Since our proposed noise optimizations also reduce the output signal, we need means to recover its strength. An option is to increase `num_sends` in the hope that the related increase in noise is less detrimental than a high gain. The results of Section 3.3 however indicate that `num_sends` significantly influences noise since it increases both additive and multiplicative noise. This approach thus seems less promising. Instead, we see the increase of the synaptic input time constant as a more promising alternative. Since it is currently set close to its minimal value there is a great potential to boost the output signal. The increase of the synaptic time constant causes the individual voltage pulses of each activation element to decay slower so that the OTA charges the membrane capacitor for a longer time. This will cause a higher output voltage for the same input operands.
Surely this will increase the saturation effects of the synaptic input, which may need to be compensated with an increased `wait_between_events`, but as we have seen in Section 3.4, depending on the input distribution, saturation is compensated quite well in the default calibration. Saturation for slightly smaller input operands might be tolerable.

In the context of machine learning saturation is a very problematic effect, especially for training. Since saturation is a very dynamic effect that appears only for certain input combinations, the network has no means to adjust to the saturation. To the network, saturation appears as a dynamic change of the non-linear behavior of the analog matrix multiplication for every input sample. This makes outputs of different samples hardly comparable so we definitely want to avoid saturation.
Considering that weight and activations often follow an unimodal distribution that is in rough approximation symmetric and centered around zero, we assume in general that the saturation is spread over both excitatory and inhibitory synaptic inputs equally and that most of the weights and activations are close to zero. With machine learning as a workload for the analog matrix multiplication, both these assumptions imply that increasing the synaptic input time constant should be possible without significant saturation effects.
Also in case the output current of the OTA is a saturation-inducing factor, the decrease in gain, which we proposed as an additional optimization, should result in even less severe saturation.

After we presented our thoughts on how an optimized configuration for the chip could look like, we also want to mention the remaining factor, energy consumption. Since energy consumption and the related scaling possibilities are the fundamental reasons to use analog hardware, we want to consider how our optimization approach affects the required energy.
We assume a constant power consumption that is mostly independent of load [30, p. 68]. In [47] the power consumption of the analog chip was measured to be at approximately 200 mW. However, this was a very optimized setting. In [30, p. 68] J. Weis describes the energy consumption as highly dependent on the number of FPGA connections to the chip. Depending on the rate of input events a higher or lower number of active links can be configured so that the chip consumes around 360 mW with all links active. The rate at which input events are sent is dependent on `wait_between_events` so for high values the required number of links could be reduced, but for now we assume the number of links to be constant.
With the power consumption being constant, the used energy primarily depends on the execution time. From this perspective, a low `num_sends` is also beneficial since it has a significant impact on runtime (see Figure 3.3). A higher synaptic input time constant does not increase the execution time of the matrix multiplication directly. Increasing it only makes the voltage decay slower to the reference potential so that successive events may overlap. This is only problematic if the superposition of multiple events causes the synaptic input line to reach too high voltage differences. If we have to increase `wait_between_events` because we reach saturation, this might cause a slight increase in runtime. But as shown in Figure 3.4 only an increase from 1 to 3 results in a visibly increased runtime. For all other values of `wait_between_events`, the runtime is mainly constant.
If we do not consider the power consumption to be constant our optimizations most likely increase the power consumption. Since we are primarily trying to increase the input signal to the OTAs and hence the voltage on the membrane, there is more energy loss during the reset periods in between the vector-matrix multiplications than for smaller input signals. Only the decrease in the OTA gain potentially lowers the signal amplitude on the membrane. Whether this effect outweighs the longer integration of higher input signals caused by the higher synaptic input time constant is unclear.
Nevertheless, it should be noted that if our optimizations are beneficial for training and similar accuracy can be achieved in a shorter time, this would result in significant power savings.

4.2 Training Methodology

In this section, we present our training procedure, which algorithms we use, and what our model architecture looks like.
We are guided by the work of Klein *et al.* [19] and the work of Kuhn [41]. Similarly, we train our model on the SpeechCommandsV1 dataset [48], [49]. We preprocess it with the librosa-library [50] to extract a mel-spectrogram [51] with 32 Mel bands and convert the power spectrogram afterward to decibel (dB) with the reference power equal to the maximum power. If we set the reference

power during the dB conversion to the maximum, this means that the quotient in the logarithm is always smaller than 1 thus we end up with purely negative values in our log-mel-spectrogram. Natively the librosa library limits the range of the dB spectrum to 80 dB so that we have a value range from −80 dB to 0 dB. Because the hardware expects only positive input activations, we have to shift the value range by 80 so that we end up with the log-mel-spectrogram in a range between 0 dB and 80 dB. The shift in the logarithmic domain translates to a scaling in the linear domain but this makes no difference for the classification task. A differently chosen reference during the dB conversion would have a similar effect.
Our preprocessed input data is then fed into our model. Similar to [19] we also use an MLP with 3 linear layers but in this work, we also insert batch normalization layers between each layer. Figure 4.1 shows a representation of our floating point precision model.
Something important to mention is that in this work our linear layers **never** use a bias term. We do this because it makes the outputs of the linear layer equivalent to the output of the analog matrix multiplication. This allows us an easier interpretation. By inserting the batch normalization layers right after the linear layers the learnable offsets for each feature should have a similar effect as a bias term. Our primary reason for inserting the batch normalization layers before the ReLU function is to avoid negative inputs for the next linear layer since this is a requirement of the hardware. The ReLU function sets all negative values to 0 so that only positive values reach the next input.
At the end of the forward pass, we decide on the predicted label by using the *argmax* function.

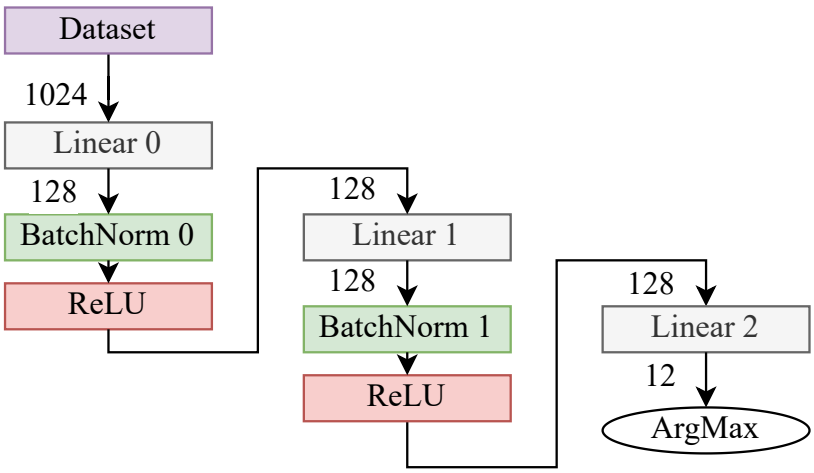

Figure 4.1: Block diagram of the used MLP.

Regarding the optimizer, we use Adam [52] in combination with a CosineAnnealing learning rate scheduler [53], which decays the learning rate so that a learning rate of 0 is reached at the very end of the training. In contrast to the original paper, we use the learning rate scheduler without warm restarts.
After a learning rate sweep, we find the initial learning rate of 1×10^{-3} to work best in our setting. For the training of the floating point precision model on digital hardware, we use a batch size of 128 and train the model for 300 epochs. Although we showed in Figure 3.1 that the chip needs bigger batches to amortize the setup overhead, we use a smaller batch size because the validation accuracy during training is more stable and the final accuracy is slightly better.
Table 4.1 shows our final training results of the floating point model. For reference, we also include the achieved accuracy with a batch size of 1024 and the results if we omit the batch normalization layers. We see that we can reproduce the test accuracy result of [19] for the model without batch normalization and also find a similar learning rate of 1×10^{-4} to work best in that case.
For all following experiments, we stick to a batch size of 128 and a learning rate of 1×10^{-3}.
If we want to use the model on BSS-2, we just have to exchange the linear layers with the replacement `hxtorch.perceptron.nn.Linear`. It extends the interface of pytorch's `torch.nn.Linear`

BatchNorm	yes		no
Batch Size	128	1024	128
Learning Rate	1×10^{-3}	4×10^{-3}	1×10^{-4}
Validation Accuracy	85.65 %	83.90 %	83.35 %
Test Accuracy	84.78 %	83.45 %	81.53 %

Table 4.1: Training results of the floating point MLP on digital hardware on the keyword spotting task.

layer with the runtime parameters `num_sends` and `wait_between_events` and in addition requires two scaling functions, which scale the input activations and weights to the hardware range. Figure 4.2 depicts the relation of the scaling functions.

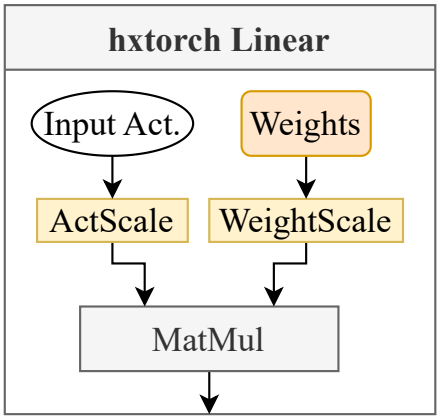

Figure 4.2: Representation of the `hxtorch.perceptron.nn.Linear` layer.

After the scaling of both weights and input activations, a rounding operation followed by a range check is done automatically hidden inside the function for the analog matrix multiplication. We also want to mention that in the network, all digital computations are done in floating-point precision (32 bit). The transition to and from quantized values happens completely inside the function for the analog matrix multiplication. The inputs and outputs of it are float values.
The scaling functions can be any callable that receives the input tensor and returns the scaled output tensor. The input activation scaling can also happen outside the linear layer by passing `None` as a parameter during initialization. Currently, the default for the weight scaling function is an identity function that also clips the weights to the range of the hardware weights. This operation is implemented as an in-place operation so that the floating point weights can never grow bigger than the hardware range.
With the default weight scaling function in mind the `hxtorch`'s linear layer initializes the weights differently than its pytorch counterpart. In [39] the initialization process is described in more detail but the fundamental difference is the weight magnitude. Because of the assumption of unity scaling and considering the inherent gain of the analog matrix multiplication, the weights are initialized with much higher values.
In this work, we do not use this initialization method and rely on `pytorch`'s default uniform initialization in a range between $\pm 1/\sqrt{\text{num\_in\_features}}$ for the initial full precision training on digital hardware. We use our own scaling functions so that the model during retraining does not have to adjust to the sudden weight magnitude change. During hardware-in-the-loop training, we want the model to adapt only to the static offsets, the noise, and the non-linear gain factor of the analog matrix multiplication.
There are multiple strategies for mapping the floating point inputs to the hardware range. Some

of them are described in more detail in Section 4.4. We can achieve a similar effect as the default scaling function by scaling our pre-trained weights by a static factor to the hardware range and clipping weights to the hardware range that might adjust to higher values during training.

4.3 Adjusting the Hardware to ANNs

As described in the consideration section of this chapter, our primary idea of tuning the current default calibration to improve classification accuracy consists of a reduction of the OTA gain in combination with an increased synaptic input time constant. Besides that, we want to choose a minimal `num_sends` of 1 and `wait_between_events` as low as possible. In this section, we present the influence of these changes on the matrix multiplication level. In a way, we extend the peculiarities chapter with the additional experiment results with our new custom calibration.
We start by comparing the noise of the default calibration with our optimized variant. Instead of just comparing the default calibration with the optimized version, we include an intermediate step, in which we only reduce the OTA gain. In Figure 4.3 we compare the default calibration (gm: 450, ts: 0.32) with a calibration that only reduces the OTA gain (gm: 150, ts: 0.32) and a calibration with a reduced OTA gain and an increased synaptic input time constant (gm: 150, ts: 2.27).

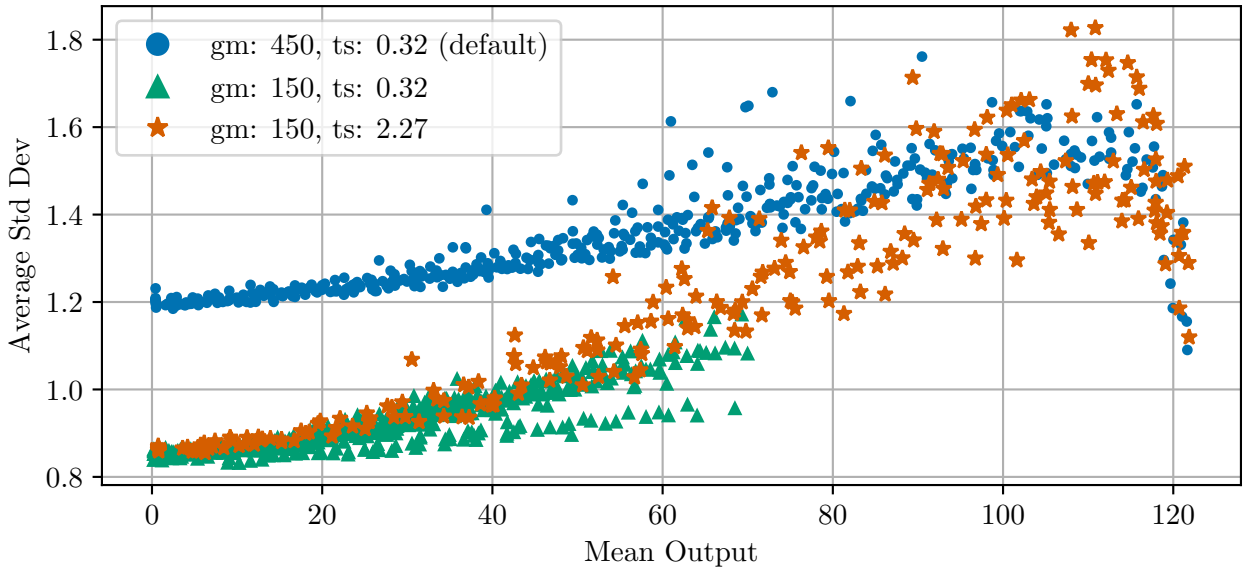

Figure 4.3: Average standard deviation over the mean output of the matrix multiplication sampled for all weight combinations and recorded for the default and custom calibrations.

In this experiment, we record the output of the analog matrix multiplication with a wide range of different input variations. We vary both weight and activation values so we have the results of all variations. To slightly reduce the combinatorial growth, we increase the activations with a step size of 3 from 1 to 31. In the case of the weights, we use all integer values between 0 and 63. For all of these combinations, we compute the average standard deviation over all neurons for 200 repetitions.
In the next step, we filter our data and exclude all results with an activation smaller than 10 because of the drastic increase in noise for combinations of low activations with moderate and high weights (see Figure 3.13). Furthermore, we exclude any data points with a mean output exceeding 122. This is because we see that when the output reaches 120, the standard deviation decreases, indicating that some neurons' outputs are already clipped by the ADC range.
With the remaining data points, we create a scatter plot with the average standard deviation over the mean output. We do this because we want a fair comparison of the noise for different output values. In this context, we see the SNR as a bad metric to compare different calibrations because in

this setting it depends on both, noise and mean output, and both parameters change significantly for different input combinations. Therefore we choose the representation in Figure 4.3. We can compare the noise for different outputs but it is also possible to compare the noise of different calibrations for similar outputs.
The most significant observation in Figure 4.3 is the reduced offset for a mean output of 0. This implies that the OTA gain allows us to influence the term of the additive noise that is independent of the time score in Figure 3.9. The minimal noise of the default calibration is higher than the one in Figure 3.9 because we do not choose a parameter combination that yields the minimal possible time score in the present setting. In this setting, we choose `num_sends`=1, IFs=64, and `wait_between_events`=5 because of the use case we optimize for in this chapter. As mentioned in the considerations section, we use a very small value for `num_sends` because it increases both additive and multiplicative noise. For the number of non-zero elements in the activation vector, we choose 64 because we assume that approximately half of the inputs to the ReLU function are negative so only 64 of the 128 input elements remain as non-zero values. The choice for `wait_between_events`=5 is because we want to make sure to avoid saturation during training. The reduced standard deviation for a mean output of zero is also visible for bigger outputs up to roughly 70.
Further, we see in Figure 4.3 that only reducing the OTA gain is not a feasible option because the output can no longer reach values higher than approximately 70. Our remedy for that is the increase of the synaptic input time constant that increases the gain of the matrix multiplication without increasing the noise of small output values. However, we see that for higher output values we reach an average standard deviation that is similar to the one we got with the default calibration. For very high values around 110, we even surpass the noise of the default calibration. Because most of the orange data points are below the blue ones, we still assume that this calibration can perform better than the default calibration. If we consider that smaller output values occur more often in typical distributions, the reduced noise for these appears even more beneficial.

This custom calibration, however, performs better only if the decrease in noise is not mitigated by the increase in saturation caused by the higher synaptic input time constant. To get an impression of the noise, we repeat the previous experiment where we analyzed the effect of saturation (see Figure 3.19) with our new calibration. The results are shown in Figure 4.4. Similar to before we increase the summed activation vector with an activation pattern that is most likely to create saturation (on the left) with an activation that should attenuate the build-up of saturation (on the right). We do this for different values of the weights and `wait_between_events`.
We can see that in the worst-case scenario not even `wait_between_events` with a value of 5 can avoid saturation of moderate and high weights. Compared to Figure 3.19 the curves separate much more. It is interesting that in the worst case, saturation starts at similar summed activations compared to the default calibration for the weights 28 and 63. This means that during the computation of the dot product, similar partial sums result in higher results without saturation. In the context of machine learning, this implies that individual activations have more impact on the final result and can change the output value in a wider range. In case the final output depends on only a few input features, the output quantization error will decrease.
However, because the increased synaptic time constant also increases the gain of the overall matrix multiplication, we can reach much higher mean output values for similar summed activations before saturation starts. For smaller weights, saturation causes a significant spread of the curves. Compared to the default calibration this spread happens for slightly smaller summed activations implying that saturation starts earlier. But again the achieved output value is higher. If we look at the best-case scenario, we see that resorting the values to a pattern that might cause less saturation is ineffective for small weights. In Figure 4.4 this is the case for the smallest two weights 3 and 13. For higher weights, the best-case scenario again performs better than the worst-case.
To assess whether the new calibration with increased saturation potential is suitable for ANNs, we again have to consider the implications of the workload. If we assume that we use the ReLU activation function, approximately half of the input values will remain at a value of zero. The other

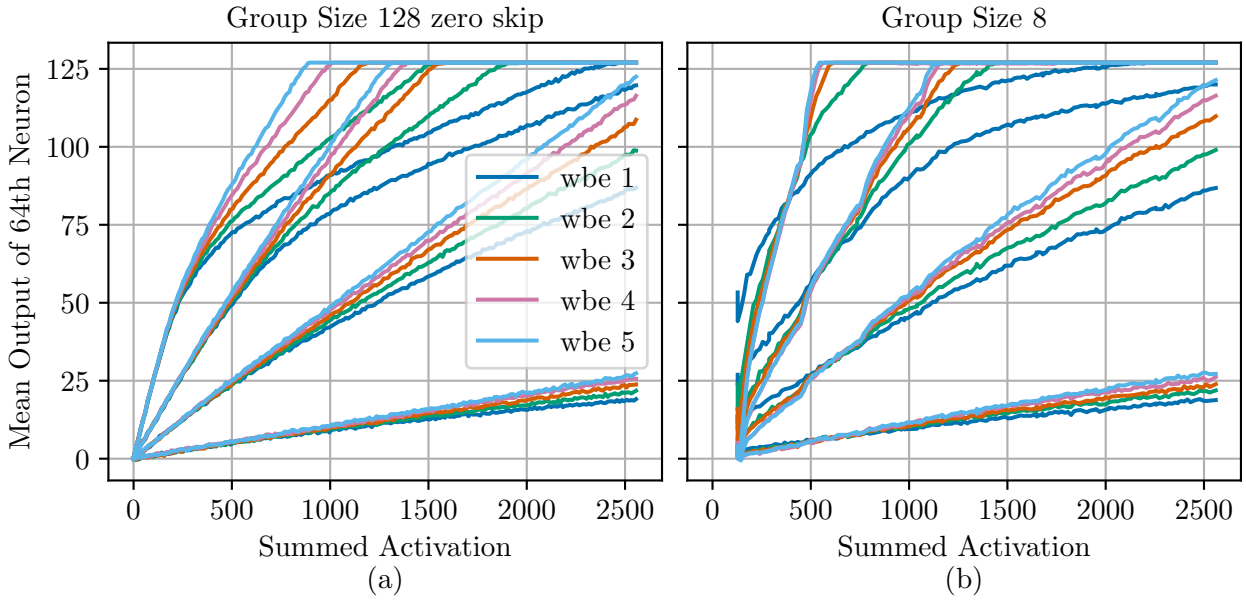

Figure 4.4: Worst-case vs. best-case saturation for the weights {3, 13, 28, 63} (higher weight ⇒ higher initial slope) for custom calibration with high synaptic input time constant and reduced OTA gain.

half does not take on the maximum value but is more likely to have a moderate average value. If we assume that half of the input features on average take a value of 20, which is a conservative estimation in our opinion, then the summed activation in Figure 4.4 will never increase beyond 1280 ($= 64 \cdot 20$).
Similarly, we can assume that not all weights grow to a value of 63. If we assume a roughly symmetric weight distribution with an equal amount of positive and negative weights the influence of saturation will decrease even more. Because there are separate synaptic input lines for positive and negative weights (excitatory and inhibitory input) saturation occurs separately for both lines hence the effect will be even less severe than shown in Figure 4.4.
If we then try to estimate the average value on one of the lines, we get even closer to a real machine learning scenario. Similar to before we assume that on average, the positive weights take on moderate values. In our opinion, the case for a weight of 28 seems to be a good though conservative guess for the average weight value.
With all these assumptions of moderate activations, moderate weights, ReLU activation function and symmetric weight distribution so that exhibitory and inhibitory synaptic inputs saturate separately, saturation might still be unproblematic for the network. However, a slightly increased `wait_between_events` to fully avoid saturation might be necessary.

4.4 Adjusting the Mapping of ANNs to the Hardware

After we verified that our changes to the calibration can be beneficial, the question of how to scale weights and activations to the hardware remains. That is what we want to analyze in this section. Our approaches are guided by the implementations provided by `hxtorch`. On one side, the package suggests statically scaling weights and activations. `Hxtorch`'s `ConvertingReLU` is one example for the activations and the inplace clipping of `hxtorch`'s linear layer is the analogy for the weights. On the other side, `hxtorch` provides dynamic scaling functions that scale weights and activations per batch based on the respective maximum value to the hardware range.
Fundamentally we like the idea of the dynamic maximum scaling per batch since it forces the distribution to fit on the hardware and explicitly avoids clipping on the edges. It also dynamically adjusts to magnitude changes during retraining and solves the problem of finding a suitable static

scaling factor. Yet static scaling gives the network the capability to train higher weights and possibly truncate the pre-trained distribution so that the overall higher weights produce higher signals that can be distinguished better from noise. Because both approaches have beneficial properties, we also suggest a third option that combines both solutions.

Instead of using the maximum of each batch as a reference for the scaling, we track the batch maximum via an exponential moving average (EMA) and use this weighted average as a reference for scaling. This has fundamentally the same behavior as batch maximum scaling if we reached convergence, but during training the reference for scaling is more robust against noise because of the averaging. By choosing a higher or lower decay factor the behavior can be tweaked towards the static scaling or the dynamic scaling. While we are sure that for the per-batch scaling approach without any averaging the scaled values will never exceed the input range of the analog matrix multiplication, the approach using an EMA could overshoot. Therefore we have to clip the result of our scaling function to make sure that the results lie within the required range.

The fact that in the converged state, only the highest value of the input distribution is mapped to the highest value of the input range remains with that approach. Only the sensitivity to outliers of the maximum is reduced and during training the EMA lags behind the current maximum. The introduced lag is a side effect of the EMA since it is only a very simple infinite impulse response filter (IIR).

In our quantized context, this means that in the converged state high values that are possibly important for the network are correctly represented. Yet in typical weight distributions, there are many more small weights than particularly high ones. This raises the question of whether it is really important to distinguish the magnitude of a few very high weights accurately or if it is enough information that their magnitude is higher than all the other ones. If the latter is the case, we would use the available resolution a little bit better by using a certain percentile of the input range as the scaling reference instead of the global maximum. All values that are above this percentile and hence lay outside the input range of the analog matrix multiplication are again clipped to the respective minimum or maximum value. With this approach, we would achieve a fixed percentage of values that are always clipped, but the rest of the distribution is accurately mapped to the available input range.

Another mapping strategy arises from the observed behavior for small activation values. We have seen in Figure 3.7 that the average noise significantly increases for small activations and depending on `wait_between_events` this can grow to huge values (see Figure 3.13). Therefore it might be sensible to avoid small activations at all. A possible approach for that is magnitude-based activation pruning. Since our input range is already quite limited, it is hard to predict good threshold values. If we also assume a bell-shaped activation distribution, magnitude-based activation pruning will affect a significant proportion of the activation values. Also, it is hard to estimate the impact of higher noise on small values compared to a further reduced input range. Our following experiments show whether this mapping is useful.

Finally, a last mapping strategy worth mentioning also originates from the `hxtorch` library. During our initial experiments, we were very surprised that our hardware retraining always resulted in a weight distribution with a strong tendency towards binarization. In other words, starting from the initial bell-shaped weight distribution the hardware retraining made the weights shift towards the minimum or maximum value so that after a sufficient amount of epochs the distribution consisted of mostly these two values. The reason for that is a bug in the per-batch maximum scaling function. Instead of just computing the maximum of the batch and using it as a constant factor in the scaling computation, the retrieval of the maximum became part of Pytorch's computational graph for the backward pass. This means that gradients not only propagate through the scaling factor to all activations or weights that have been scaled by it but also an additional gradient propagates to the maximum values where the maximum originated from. This means that the maximum values that decide the scaling factors have much more impact on the outcome of the computation and hence receive much higher updates.

Because this computation is done per batch, different values receive higher updates and not always

the same ones. Depending on how many epochs the network is trained with that scaling rule, the weights can appear almost fully binarized with only a few values not having taken on the min/max value or in case fewer epochs have been used for retraining only an accumulation of values around the extreme values can be observed (results Figure 4.19).
Binarization in general is a technique that is inherently robust against noise because it relies more on the presence of activations than their actual value. Also in case the binary weights can drive the output of the matrix multiplication into the clipping range, we get very deterministic results due to the clipped noise differences. However, because a network that uses mostly two weight values of the available 127 does not efficiently use its model capacity and because binary networks do not perform generally well on all kinds of data sets, we do not recommend using this scaling approach without additional critical thoughts. Also, the strong dependence between binarization degree and the number of trained epochs makes it a quite unpredictable approach. Since this strategy has been used in earlier approaches unknowingly and because it reaches reasonable training accuracies, we include it in the following training results.

A peculiarity of all previously mentioned scaling approaches is that we scale the inputs to matrix multiplication yet the output is never rescaled. By omitting the rescaling with varying scaling factors we increase the in [6] mentioned internal covariate shift. We do this in accordance with previous work on BSS-2 and thus knowingly differ from typical quantization approaches. However, we have the batch normalization layers that do some sort of rescaling but as we explained in Section 2.1.2 this rescaling is done over the batch dimension for each feature and not globally for all instances of the batch. To some extent, this is okay because the network can adjust to the magnitude differences but in case of our dynamic scaling approaches it is quite likely that we make it significantly harder for the network to adapt to continuously changing input magnitudes. Possibly the use of alternative normalization layers especially the layer norm would be a beneficial adaptation.
For the static scaling approaches, we do not see the missing rescaling as problematic. If we want to consequentially rescale the results or at least consider the scaling factors in the following computations like it is done in other quantization approaches [17], we would find another severe limitation of the analog matrix multiplication. While the input scaling factors to the operation are perfectly known in advance, the inherent scaling factor of the analog matrix multiplication differs among neurons, highly depends on saturation, and because of non-linearities also depends on the input data. This makes the inherent gain of the matrix multiplication a quite fuzzy quantity and prevents the application of mathematically accurate quantization.
Still for the dynamic scaling approaches rescaling the output might be beneficial to improve the learning dynamics. This is not done in the current work and is a possible improvement for future related work.
We end up with the following mapping strategies that we evaluate in the next section:

- Batch Maximum Scaling
- Batch Maximum Scaling with Backprop through the Scaling Factor
- EMA Maximum Scaling
- Static Scaling
- EMA Percentile Scaling
- Activation Thresholding

4.5 Training Results

In this section, we present our training results with the previously mentioned mapping strategies and different calibration approaches. We validate whether our calibration idea can actually yield improved accuracy results and in which range the calibration parameters can be varied without other effects like saturation dominating the computations. The same holds true for the mapping strategies. We compare different approaches and find suitable values for thresholds and scaling percentiles.
Before we start with the presentation of the hardware results, we first show results with the mock mode (see p. 11) to foster an understanding of the two fundamental properties of the hardware, gain and noise. We use this opportunity because we have fine-grained control over these parameters in mock mode and because additional effects like saturation and multiplicative noise do not occur. In Figure 4.5 we see that an increase in the noise's standard deviation results in a steadily deteriorating validation accuracy. That is intuitive and expected, but if we compare by how much the noise has decreased the accuracy, we see that an increasing noise is much more detrimental for the dynamic scaling approaches than for the static scaling. If the noise is quite small the differences are not that big, but for higher noise, the differences among the scaling approaches get increasingly larger.
If we analyze the results of the EMA scaling, we see that this approach is indeed a compromise of both the static scaling and the per batch maximum scaling. We assume that if we decrease the decay rate, we can get even closer to the behavior of the static scaling approach. However, in this work, we do not conduct a hyperparameter analysis of the decay rate. Instead, we use the same value as Pytorch's batch normalization layer, which tracks running estimates of the mean and standard deviation in a similar fashion.

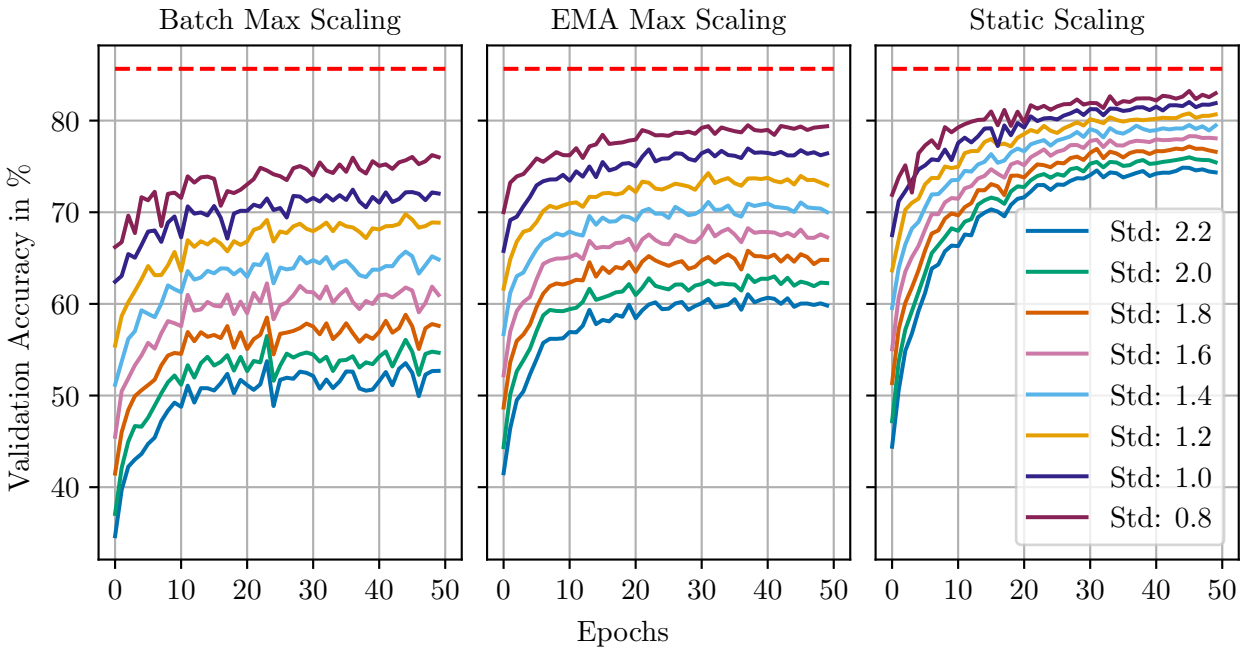

Figure 4.5: Mock mode training with variable additive noise and default gain of 0.002.

In Figure 4.5 we choose a gain factor for the result of the matrix multiplication of 0.002, which corresponds to `hxtorch`'s default. In our experiments with the default calibration, we observe varying values between 1.6×10^{-3} and 2.0×10^{-3} for the default calibration. However, these are not precise values and should only be interpreted as a rough estimate of magnitude. They are measured with hxtorch's `measure_mock_parameter()` function, which does only a quick and relatively rough calculation of the gain.
To see how the gain impacts the general training behavior, we repeat the mock training with the

different scaling approaches, but this time we vary the gain and keep a moderate standard deviation of 1.6, which is in the range of values that we have seen in our previous experiments. Figure 4.6 shows the results of the experiments with variable gain.
We see that decreasing the gain has a similar effect as increasing the noise. A smaller gain factor deteriorates the accuracy. Again the static scaling approach is less sensitive, also to a decrease in gain. The EMA approach again performs better than the per-patch scaling but worse than the static scaling. Compared to the variable noise it seems like a higher gain has a stronger impact than lower noise. While the accuracies do not vary that much for a gain between 7×10^{-3} and 4×10^{-3}, for lower gain factors the difference in accuracy is becoming increasingly more severe. For the variable noise, we cannot observe this behavior.

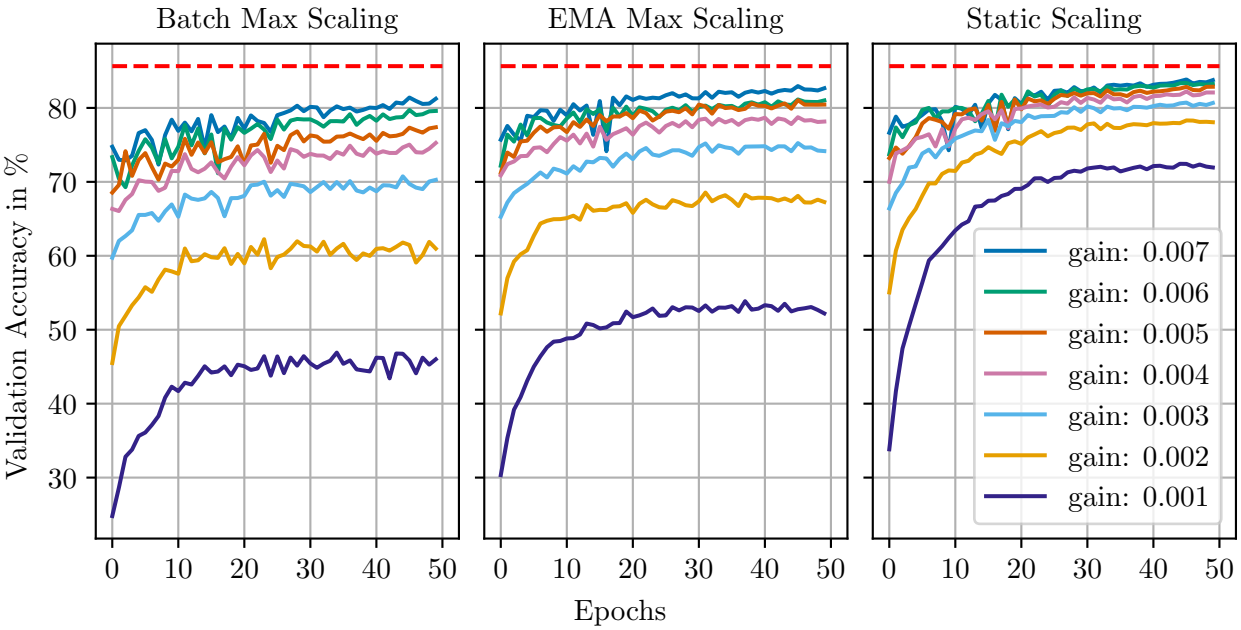

Figure 4.6: Mock mode training with variable gain with noise std of 1.6.

Even though the mock mode is only a rough approximation of the hardware, we assume the noise and the gain to have similar effects on the real hardware. Because we observe that in both of the previous cases, the maximum scaling per batch performs significantly worse than the other approaches, we do not conduct further experiments with that approach. Fundamentally the mapping of the EMA maximum scaling approach should be the same but with better training dynamics and a lower sensitivity to noise so we conduct the following experiments always with an exponential moving average in cases of the dynamic scaling.

4.5.1 EMA Scaling

In the next step, we want to start with the retraining of our model on the real hardware. We start with the EMA maximum scaling approach and use the default values of `hxtorch`'s matrix multiplication `num_sends`=1 and `wait_between_events`=5. Then we train our model for 100 epochs on the analog hardware and vary the parameters for the OTA gain and the synaptic input time constant.
Figure 4.7 shows a sweep over different OTA gain factors. In general, all configurations show very similar training results. Only with the OTA gain at 150 the accuracy is visibly smaller compared with all other curves. In Figure 4.3 we have seen that with such a small OTA gain the output can no longer reach all values of the available output range. Because we scale the output anyway, one could think that this is not that problematic. However, if we do not use the available range, we lose resolution. Since the output of the signed 8-bit resolution is quantized at the input of the next linear layer to an even lower resolution of unsigned 5-bit this appears as an insignificant

problem. Figure 4.3 shows that even for an OTA gain of 150 outputs around 60 can be reached, yet the accuracy with this gain is still below all other curves. What Figure 4.3 does not show is the required input combination to reach these outputs. It is quite likely that only very high weights in combination with high input activations reach these values. The model could just train towards higher values to compensate for the decrease in gain, but this means we effectively lose the resolution in the weights. If all weights have to be close to the maximum possible value to reach sufficiently large outputs, the lower bits of the weight resolution will become meaningless and hence we effectively lose resolution of weights.

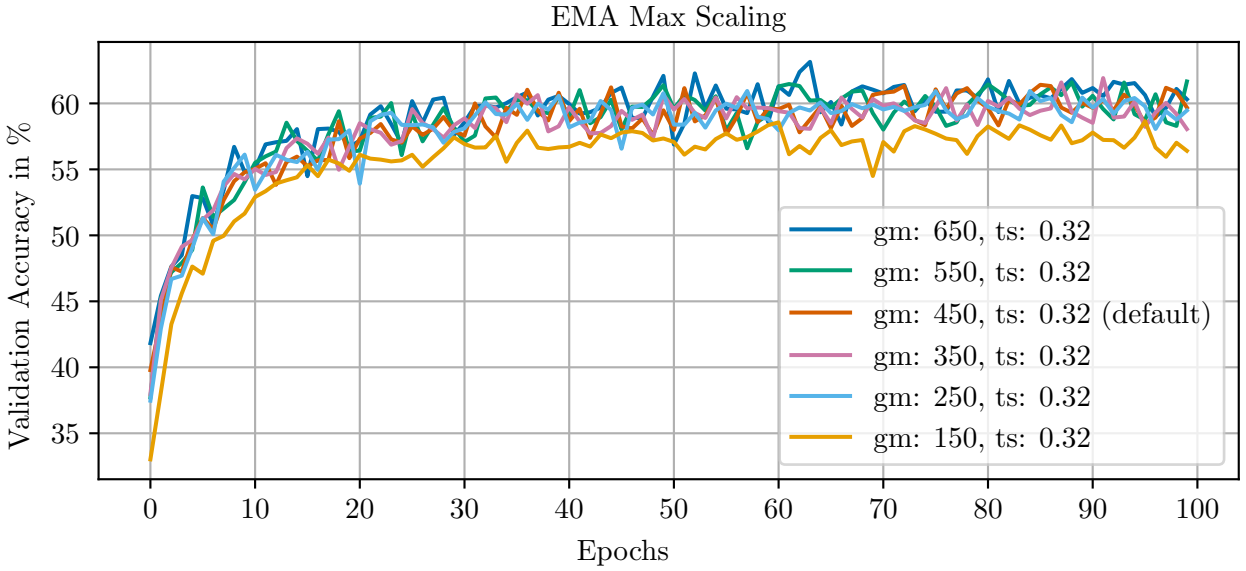

Figure 4.7: Hardware training with variable OTA gain.

As we explained in Section 4.3, we expect the overall gain to be too small with a very small OTA gain, but our measurements have confirmed a significantly reduced noise. To circumvent that disadvantage, we proposed the increase of the synaptic input time constant. Therefore we continue the parameter sweep, but this time over increasing values of the synaptic input time constant. This increases the gain, but for too high values we expect saturation effects to become relevant. This can have negative implications on the training and the achievable accuracy. In Figure 4.8 we can see a comparable behavior to what we've seen in Figure 4.6.
This indicates that the synaptic input time constant indeed primarily increases the overall gain of the matrix multiplication. Overall we have two significant improvements with an increased synaptic input time constant.
Firstly, the final accuracy increases significantly. The increase is smaller the further we increase the time-constant, but we start with a large accuracy increase of around 10 % between a time constant of 0.32 and 0.71.
Secondly, the initial accuracy, immediately at the start of the retraining, also significantly increases. This implies that an increase in the time constant is in this case effective in two ways. Not only does it increase the achievable accuracy but also the training time might be reduced because of the higher initial accuracy during retraining. Unfortunately, this trend is limited and an increase of the time constant over a value of 2.27 seems to cause barely any accuracy increase anymore.
Still, the question remains whether this increase is also possible without the decrease in the OTA gain and whether the decrease in noise from the OTA was actually beneficial. This question is very hard to answer because the two parameters already involve many cross-dependencies.
The results look indeed very similar therefore we do not include them in this section but they can be found in Figure C.3. However, we want to explain the interwoven dependencies. An increase in the OTA gain implies more output current for a certain input voltage difference. An increase in the time constant means that the voltage difference at the OTA's input exists for a longer time

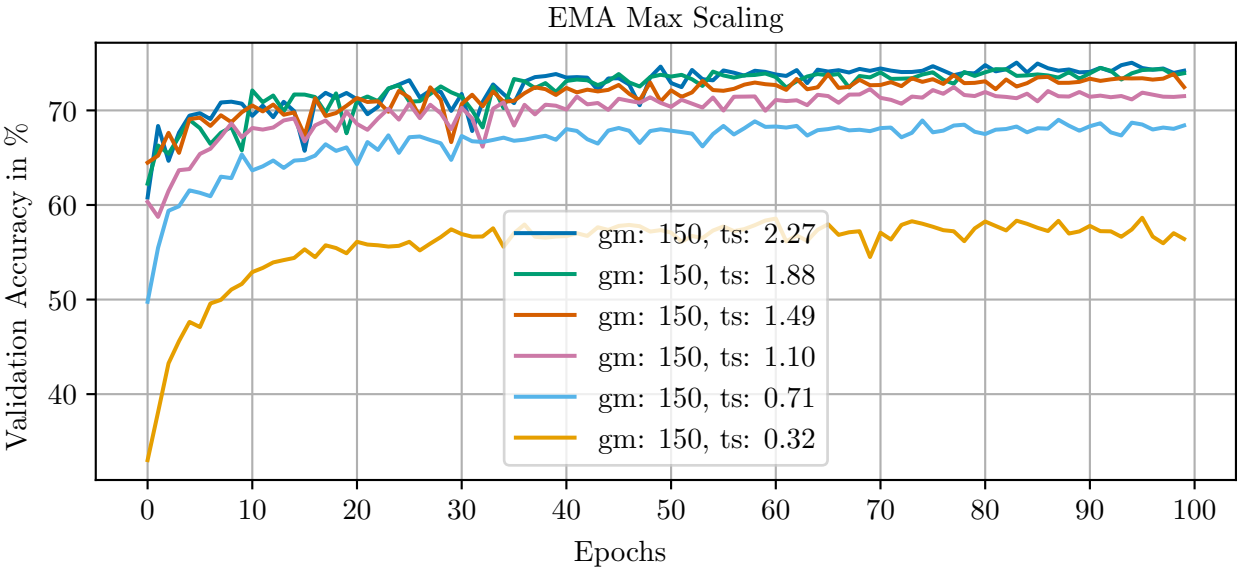

Figure 4.8: Hardware training with variable synaptic input time constant (ts).

and hence the current flows longer. Both of these components influence the accumulated charge multiplicatively. The combined increase of both parameters would therefore increase the overall gain even more. However, problems with noise and saturation still apply. If we increase the OTA's gain, it will reach its output limit earlier and saturation might be relevant for even smaller synaptic input time constants. Overall the higher OTA gain will result in higher noise. If we also consider that higher weights cause more noise, it is quite likely that the slightly different advantages and disadvantages cancel out so that the overall result remains quite similar. If we assume that a high OTA gain in combination with a high synaptic time constant can achieve higher overall gain factors, then the weights even with a small value have a high impact on the final result and hence not so many high weights are necessary. In this case, a lower multiplicative noise is canceled out with a higher constant noise from the OTAs. It is difficult to attribute the accuracy changes to separate factors.

Another very performance-relevant decision is the choice of `num_sends` and `wait_between_events`. In the previous experiments, we used `num_sends` of 1 and `wait_between_events` of 5 because we want to keep runtime and noise low but definitely avoid saturation. Our previous experiments showed that the default calibration is much more robust against saturation than our adjusted calibration. Therefore it is possible that tuning of `wait_between_events` and `num_sends` might also increase the performance. Although it has been proposed in [30] that `wait_between_events` and `num_sends` should be tuned for each layer individually, we only vary the parameters for all layers globally. Even though we agree with the statement, the grid search with hardware-in-the-loop training is a very time-consuming procedure. Therefore we make a global decision similar to the other calibration parameters.
We start with our parameter search with a calibration without increased synaptic time constant. Since all of them performed very similarly, we choose the default calibration and begin with a sweep over `wait_between_events`. The low synaptic input time constant suggests that we might not need a high `wait_between_events` to avoid saturation. Because a lower `wait_between_events` reduces the additive noise, lower values will possibly perform better. Table 4.2 shows the mean validation accuracies of the last 10 training epochs and the average final test accuracies of 5 repetitions of the forward pass after 100 epochs of training. All accuracies are given in percent.
We see that a lower `wait_between_events` indeed increases accuracy. The maximum increase in accuracy is 3.24 % for the test accuracy respectively 4.19 % for the validation accuracy. While a `wait_between_events` of 3 achieves the best test accuracy, the validation accuracy is slightly

smaller than the one achieved with a smaller `wait_between_events`. Considering the standard deviation, the results for `wait_between_events` below 4 lead to very similar results. Still, we are far below the accuracy achieved with an increased synaptic input time constant.

`num_sends`	1						
`wait_between_events`	1	2	3	4	5	8	15
Mean Validation Acc.	64.04	**64.04**	63.27	60.42	59.85	54.72	43.89
Std. Validation Acc.	0.42	0.39	0.96	0.78	1.04	1.31	1.33
Test Acc.	61.97	62.33	**62.43**	59.75	59.19	50.00	40.02

Table 4.2: Varing `wait_between_events` with the default calibration and EMA max scaling.

Hence we also conduct a parameter sweep over `num_sends`. We hope that this increases the gain and hence the accuracy. In Figure 4.9 the results of that experiment are shown.
For both `wait_between_events`=1 and `wait_between_events`=5 we sweep `num_sends` and find a surprising result. The effect of `num_sends` is in the two values of `wait_between_events` completely different.
For `wait_between_events`=1 increasing `num_sends` indeed increases the accuracy by a significant amount. This increase continues for values up to 4 or 5, where the accuracy is definitely in the range of the results with an increased synaptic time constant. After that, the accuracy drops. However, for `wait_between_events`=5 the parameter `num_sends` barely increases accuracy. This is not in line with our expectations.
Since the additive noise is proportional to the product of `wait_between_events` and `num_sends`, it is possible that the positive effects of the increased gain are canceled out by the noise increase. The positive effects of the increased gain are thus only visible if the noise does not grow too much, due to a low `wait_between_events`. With `wait_between_events` at its minimum and an increased `num_sends` the risk of saturation is quite high. However, since the accuracy significantly increased, we derive that saturation is not a problem.
Another possible explanation is based on the effect seen in Figure 3.13 (b). We have seen that the mean output value for low activations significantly increases with a lower `wait_between_events`. In these situations lowering `wait_between_events` seems to have a gain-enhancing effect. Yet this is not a general effect of `wait_between_events` as shown by Figure 3.10 (b). Based on our assumptions during the considerations for the workload machine learning with neural networks, many activations typically have low values. With many low activations, the gain increase caused by `wait_between_events` might be significant.

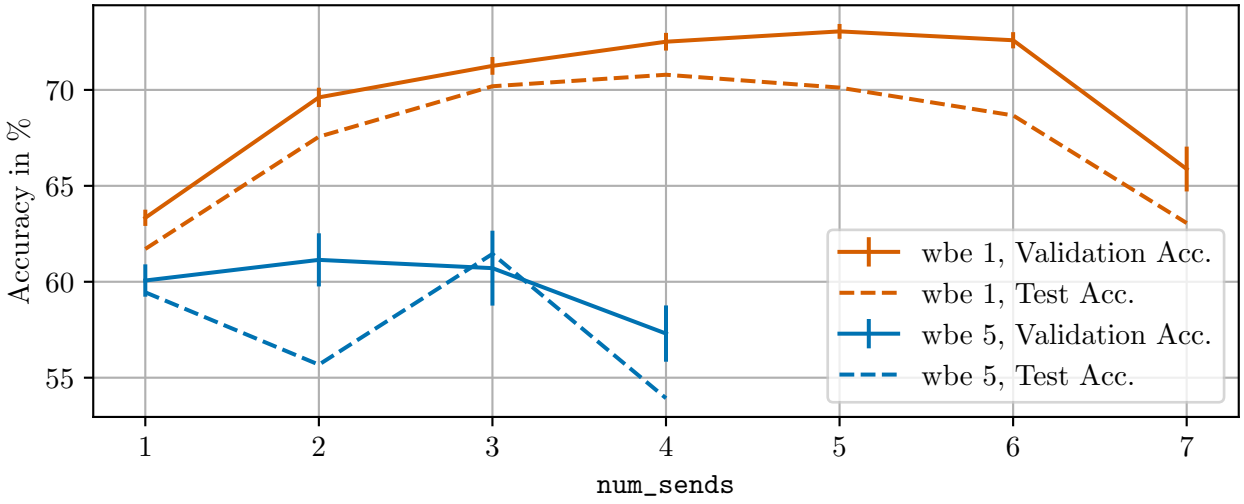

Figure 4.9: Varing `num_sends` with the default calibration and EMA max scaling.

Next, we repeat the analysis of `wait_between_events` and `num_sends` for our calibration with an increased synaptic time constant and a reduced OTA gain.
In Figure 4.10 we see the results of the parameter sweep over `num_sends` with the new calibration. A generally observable trend is that this calibration is much less sensitive to parameter changes of `wait_between_events` and `num_sends`. The difference between the highest and lowest test accuracy is not more than 4 %. This is much less than the 17 % difference we have seen for the default calibration. If we look closer, we see that the best accuracy can be achieved with an opposite parameter configuration compared with the default calibration. Instead of low `wait_between_events` with high `num_sends`, we achieve the best accuracy for a low `num_sends` with a high `wait_between_events`. Increasing `num_sends` in this case always deteriorates the final accuracy. Most likely this is because of the more severe saturation with the increased synaptic input time constant.

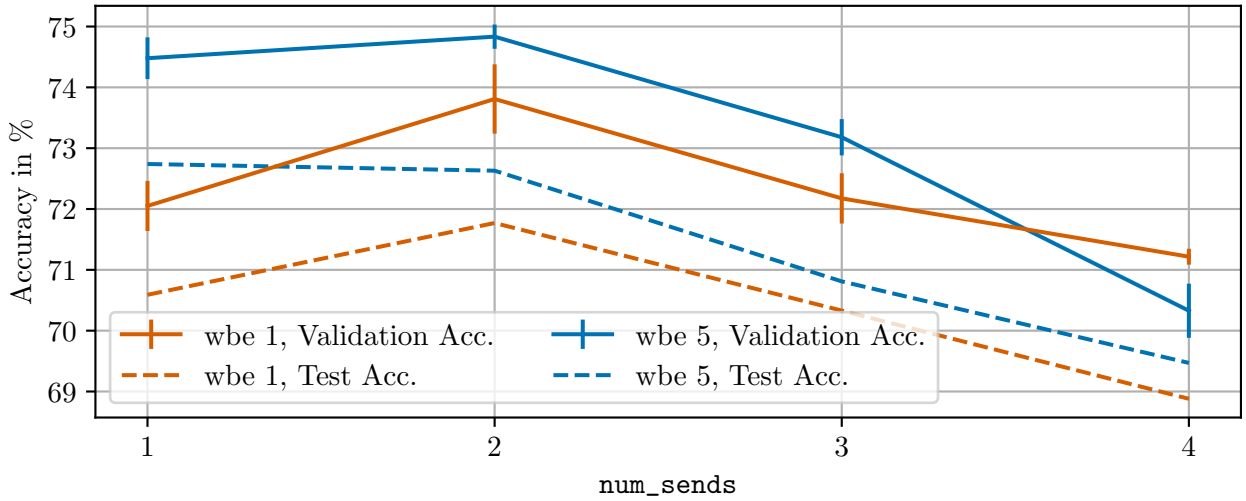

Figure 4.10: Varing `num_sends` with `ts`=2.27, `gm`=150 and EMA max scaling.

If we consider that we have increased the time constant by a quite considerable amount, it is surprising that the network performs that well with `wait_between_events`=1. In our previous analysis, we already saw in Figure 4.4 that there can be a serious amount of saturation.
In Table 4.3 we conduct a sweep also over `wait_between_events` and see that the best accuracy can be achieved for `wait_between_events` of 3. If we further decrease that value, accuracy starts dropping by a few percent most likely because of saturation. However, it seems like saturation is in fact not a very big problem for the network. If we compare the best accuracy of the new calibration at `wait_between_events` of 3 and `num_sends` of 1 with the best accuracy of the default calibration at `wait_between_events` of 1 and `num_sends` of 4, we end up with an improvement of 2.45 % in the test accuracy (the exact values of Figure 4.9 can be found in Table C.1).

`num_sends`	1						
`wait_between_events`	1	2	3	4	5	8	15
Mean Validation Acc.	72.42	73.66	**75.06**	74.63	74.21	73.50	69.45
Std. Validation Acc.	0.20	0.29	0.19	0.31	0.28	0.37	0.98
Test Acc.	71.06	71.63	**73.24**	72.93	73.00	70.83	66.59

Table 4.3: Varing `wait_between_events` with `ts`=2.27, `gm`=150 and EMA max scaling.

However, this was only the case with the EMA maximum scaling. Based on the mock mode results we expect that static scaling results in even better accuracy.

4.5.2 Static Scaling

Even though with the static scaling approach we do not have to decide on a hyperparameter for the decay rate of the exponential moving average, still we have to decide what a good static scaling factor would be. Our solution is closely related to the previous EMA scaling. Instead of manually assigning a certain value, we reused the code of the EMA scaling approach and computed only the forward pass of our network for a whole epoch. During that initialization epoch, the EMA adjusts to the maximum values in every batch but we do not update any parameter. Then after this initialization epoch, we freeze the EMA scaling factors. With that, we have an automatic routine that adjusts to a certain value close to the global maximum value in each batch. Then we start with the actual retraining and similar to before we train our network for 100 epochs but without further adjustments to the scaling factors.

With this static scaling approach, it is very well possible that weights grow to a value that exceeds the hardware input range and must be clipped. If we compute the gradient in a mathematically correct way, the gradient of the clipping function is 0 if the input to it is in a range where clipping occurs. This means that the upstream gradient is set to 0 and all other parameters closer to the input layer that would be influenced by that gradient also receive a gradient of 0. Since we use fully connected layers, the chance remains that the gradient of another neuron that didn't experience clipping can influence the parameter, yet there is also a chance that a parameter gets stuck in the clipping range.

In the previous dynamic scaling approach, this was not a problem since the scaling factor is based on the maximum and hence continuously changes so that the biggest values are the scaling limit just before the clipping range. In a sense, the dynamic scaling based on the maximum value is a self-regulating algorithm that avoids clipping. In the case of the per-batch maximum scaling, clipping is always avoided. For the EMA scaling clipping can occur, but the scaling factor adjusts in a direction that reduces clipping. For static scaling, we do not have this self-adjustment. Therefore we see it as an important addition to the static scaling that we make use of the STE [15] for the clipping function. This means that the backward pass of the clipping function is always the identity function independent of the clipping range.

If we train our network with this implementation, we get the results in Figure 4.11. We still use `num_sends` of 1 but reduce `wait_between_events` to 3 as this was in the previous cases the best value. We recognize that all calibrations achieve an accuracy above 70 %. This corresponds to the trend in the mock mode results. The static scaling is less sensitive to gain and noise compared with the dynamic scaling approaches.

Further, we see a similar trend to before if we vary just the OTA gain. The accuracy does not change much only if we reduce it to a very low value of 150. However, by increasing the synaptic input time constant, we can observe a significant improvement in accuracy, similar to our previous findings. What's novel is the previously unseen intermediate behavior that occurs when a gain of 450 is combined with an increased time constant. During the first epochs, the training behaves similarly to other calibrations with increased synaptic input time constants. The initial accuracy right after the start is significantly increased. However, towards the end of the training, the accuracy is visibly below the accuracies of calibrations with the reduced OTA gain. This confirms that reducing the OTA gain is indeed beneficial. Based on the previous experiments this is caused by the reduced noise. The initially quite high fluctuation of the accuracy for calibrations with an increased synaptic time constant can have two reasons. It could be caused by the high initial learning rate of the cosine annealing learning rate scheduler. Possibly the high initial accuracy requires a smaller learning rate for a smoother training behavior. Another possibility is that during the early epochs of training the distributions shift to situations in which saturation occurs, leading to a decrease in accuracy. A combination of both reasons is also possible.

After another parameter sweep over `wait_between_events` in Table 4.4 we notice that here actually a `wait_between_events` of 4 or 5 is optimal. Compared to the best result of the EMA max scaling approach the static scaling can further increase the accuracy by more than 2 %.

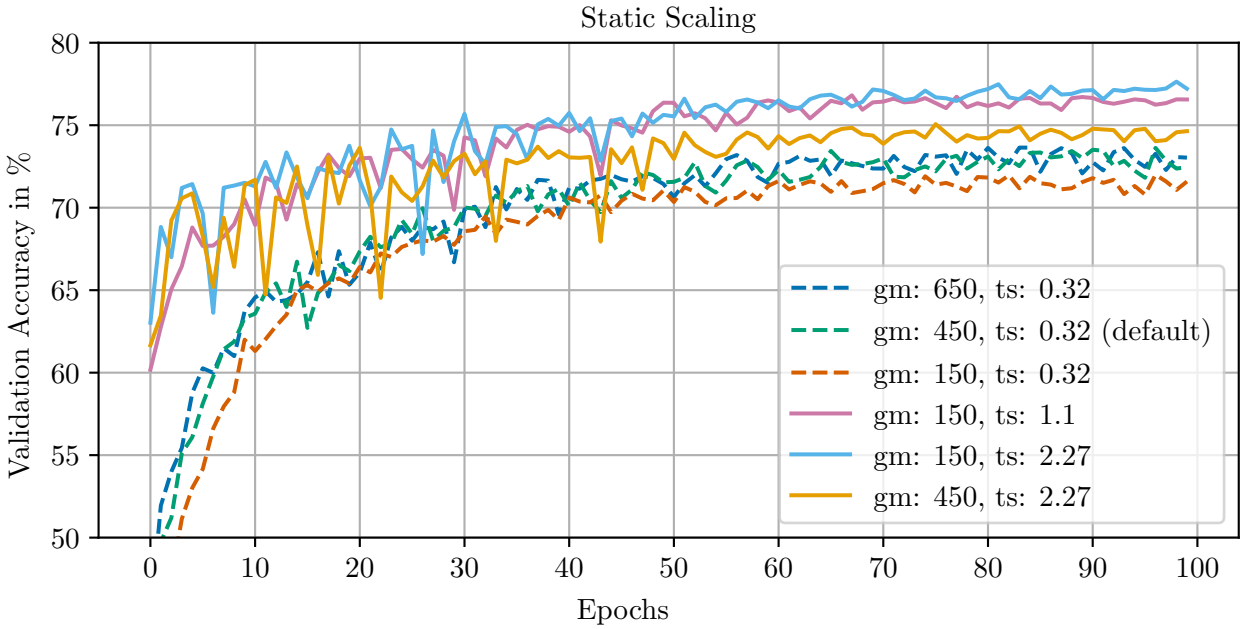

Figure 4.11: Hardware training with different calibrations and static scaling.

`num_sends`	1				
`wait_between_events`	1	2	3	4	5
Mean Validation Acc.	75.02	76.44	76.70	**77.19**	76.96
Std. Validation Acc.	0.30	0.13	0.35	0.19	0.24
Test Acc.	73.80	74.71	74.94	**75.55**	75.20

Table 4.4: Varing `wait_between_events` with `ts`=2.27, `gm`=150 and static max scaling.

The question remains of why the static scaling really performs better. In the section about the different mapping strategies, we already mentioned that dynamic scaling always maps the full input range to the available hardware range. This can result in a distribution that assigns much of the available range to a few very high weights but the majority of the values are small and have a proportionally small remaining value range. Static scaling however allows wider distributions because the scaling reference is not continuously adjusted to the maximum. Essentially, this means that static scaling can result in a mapping with a better resolution of the majority of the weights. Unfortunately, the better use of the available range is only one theory to explain the improved training behavior and there are reasons against this hypothesis. Based on our analysis of the multiplicative noise, higher weights result in higher noise so there is also an accuracy-reducing effect. Yet the actual proportion of the multiplicative noise compared to the rest of the noise is only small. Also, the higher weights make the increased noise for high weights in combination with small activations worse.

Another explanation for why higher weights might be beneficial arises from the sensitivity of the network to gain. If we assume that the matrix multiplication is a linear operation and that the inherent gain of the operation is a global multiplicative constant, then higher weights have a similar effect as a higher global gain in combination with smaller weights with the same distribution. Our hypothesis for why higher gain leads to better accuracy is that it increases the relevance of individual features in the computation. If we assume the global gain to be similar to the mock default at around 0.002 and an average weight of 20, then a difference of a single feature by 1 results in an output difference of 0.04, which is indistinguishable from noise and hidden by quantization error. A much higher number of features must be increased by much higher values to actually see a difference in the output. This however can make the distinction of similar input samples very challenging.

In Figure 4.12 both distributions for all three layers are shown. We see that indeed the static scaling approach trains towards higher values, especially in the first and second linear layers. In the last layer, this trend is just barely visible.

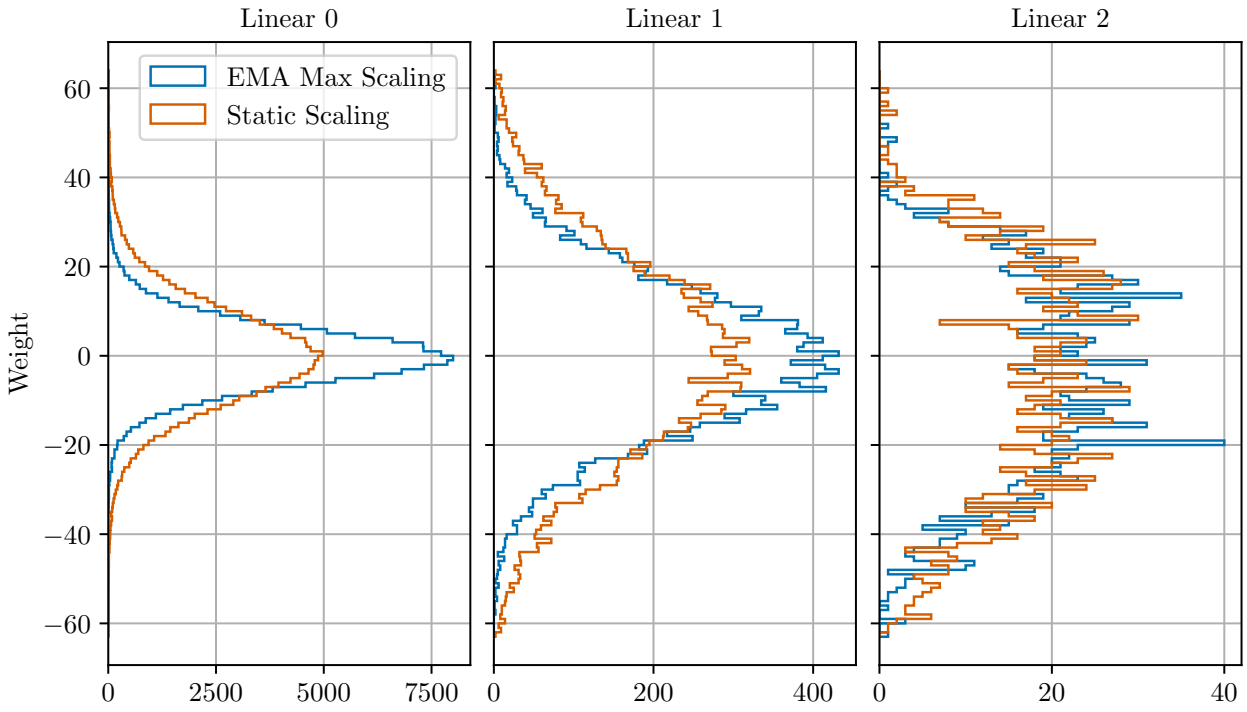

Figure 4.12: Weight distributions of static scaling and EMA maximum scaling.

If we observe the activation distribution of the two approaches in Figure 4.13, we can see a similar trend. Similar to the weights the output activations take on higher values in the static scaling approach. For the EMA maximum scaling approach, the output activations use a quite small part of the available output range. While the vast majority of activations is between -50 and 50 in the first linear layer, in the second layer most values are between -10 and 10 and in the third layer the range is similarly small yet slightly bigger with most values between -20 and 20. Since we rescale the values to the input range of the second layer the value range is not that important from the magnitude perspective. However, if we consider for example *Linear 1*, because of the ReLU function only around 10 discrete output values are mapped to an input range that supports 32 distinct values. In that case, it is quite obvious that the resolution is not used optimally. For *Linear 2* a similar situation can be recognized, although it should be less problematic here since it is the last layer. There is no following ReLU function and no further scaling. Still, this is the only point in the network where the 8-bit range is not limited by a lower following input resolution. Therefore it might be possible that a better-used output range is beneficial for *Linear 2* as well.
While the static scaling causes activations to be spread a bit more widely, only the first linear layer fully utilizes the range. Especially the output of *Linear 1* is still very small.
If we also look at the output distribution of the batch normalization layers in Figure 4.14, we see a quite surprising pattern in the second batch normalization layer. Instead of the quite smooth unimodal distributions from the linear layers and the first batch normalization layer it seems like the unimodal distribution is superposed by a multimodal distribution with at least 7 modes in case of the EMA maximum scaling. The distribution in the case of the static scaling also has these peaks but they are much less pronounced.
Even though one must exercise great caution when interpreting the distributions of neural networks, in this case, we dare to venture a possible interpretation. We assume that the output distribution of *Linear 1* is overly smooth since the scaling factor slightly varies between batches and hence the output distribution slightly varies. For many batches, this creates the impression of a smooth

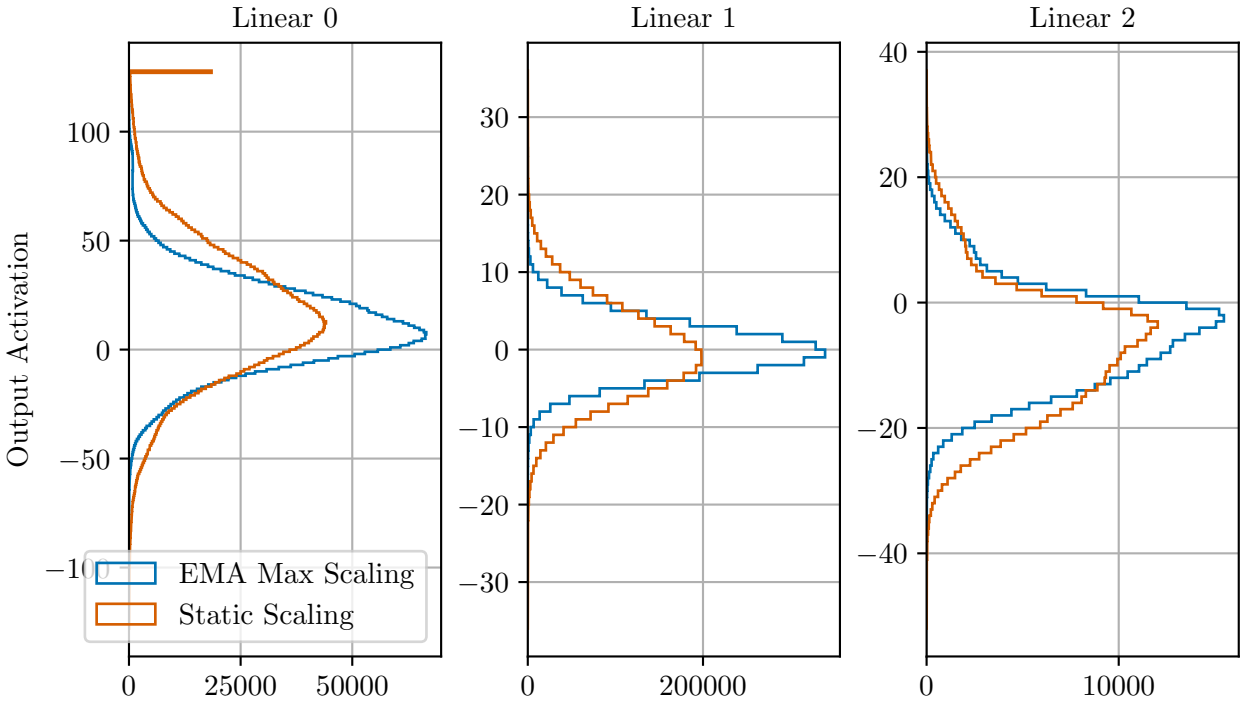

Figure 4.13: Output activation distributions of static scaling and EMA maximum scaling.

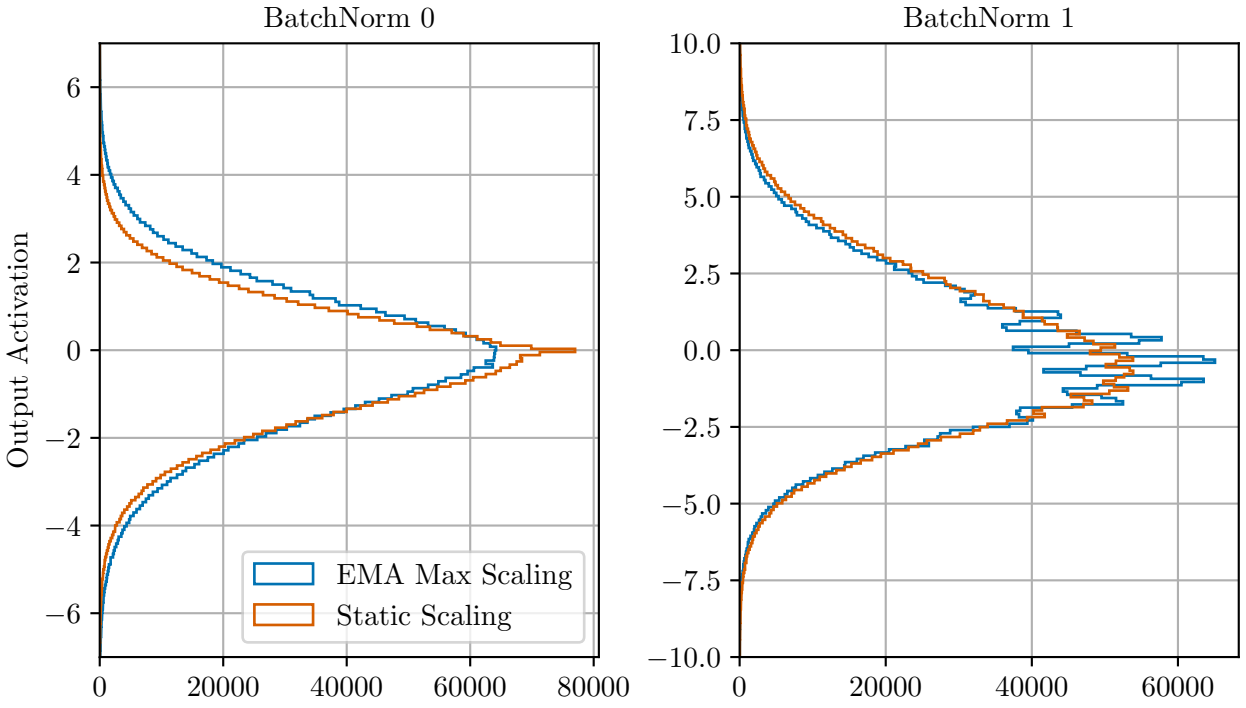

Figure 4.14: Output activation distributions of the batch normalization layers for static scaling and EMA maximum scaling.

unimodal distribution. The batch normalization to some extent also normalizes the effect of the slightly varying scaling factors. Because of the small and discrete output range a possible interpretation of the multi-modal distribution is that the peaks show the different quantization levels of the linear layer's discrete output. The less pronounced appearance of the peaks for a wider output distribution in the case of the static scaling but also the quite regular spacing between the peaks support that hypothesis.

If our interpretation of the multi-modal distribution is true, it is another sign that the output range is not used properly. But what are the reasons for that?
In Figure 4.12 we saw that the weight distribution uses the available range quite well. Also, the output activations of the first linear layer show that the output range is used to a great extent. At the positive end of the output distribution, even some clipping occurs. Likely the distributions could be stretched even wider but then the model's distributions would be truncated even more and hence some modeling capabilities would be lost. This implies that the inputs to the matrix multiplication inside *Linear 1* are already distributed in a rather optimal way, but the output is still way too small. Possibly this is the exact reason why the increase in the overall gain during our experiments with the mock mode is so beneficial for the accuracy. This would also explain why the increased gain caused by the increased synaptic input time constant is so successful. The increase of `num_sends` in combination with low `wait_between_events` in the case of the dynamic scaling has also significantly increased accuracy and is essentially also a method to increase the overall gain of the matrix multiplication.
However, the output of *Linear 0* appears well distributed with the given configuration. The primary reason for that is that *Linear 0* has 8 times more input features in the input activation vectors than *Linear 1*. To compute the result, the analog matrix multiplication must be executed 8 times and each result is then digitally accumulated. This allows the outputs of *Linear 0* to cover a wide value range. Even the weight distribution of *Linear 0* is quite narrow, which indicates that the gain for this layer might be a little too high. Higher weights would result in even more clipping in the output.
Unfortunately, this means that the required gain for the matrix multiplication is a very layer-specific value. It definitely depends on the number of input features into the linear layer, but in addition, it also depends on how many partial sums cancel out during accumulation. This in turn highly depends on the data set and the desired classification. If many partial sums cancel out the output amplitude depends on the contribution of only a few weight-activation pairs and hence requires a high gain. If only a few partial sums cancel out, possibly a lower gain is sufficient. Because this problem requires layer-specific optimizations it is a tremendous limitation.
Currently, viable parameters that can influence the gain on a per-layer basis are only `num_sends` (in some cases in combination with `wait_between_events`) and the number of input features. These are runtime parameters that can be changed quite easily, yet initializing the chip with another calibration (possibly with a different synaptic input time constant) is currently a time-consuming process. Unfortunately, this time increase would not only be present during training. Also during inference reconfiguration between layers would be required. Other options to change the gain involve further intervention in the mapping process so that weights are scaled to higher values. However, the tolerance of the network against clipping might be very limited.
The first linear layer's weight and activation distribution in the case of static scaling, indicates to some extent that weights leading to a high clipping rate of the output activations are detrimental. The question of how to further optimize the gain persists. The previous results indicate that layer-specific optimizations of the gain could be beneficial and we discuss this in Section 5.1.

4.5.3 Activation Pruning

While the previous part of the results is concerned with fundamental mapping strategies and the influence of the calibration on the final accuracy, the next part explores other global mapping strategies that might be a beneficial addition to the current strategies.
As seen in the analysis of the noise, the hardware behaves less predictable for small activations. Additionally, the current execution model completely skips activations of 0 so they have no contribution to the result and hence cannot increase the noise. Because of that behavior, we propose input activation pruning, which basically sets activations below a certain threshold to 0. Because we found that low activations are particularly noisy (see Figure 3.7) and because low activations intuitively indicate lower importance in the weighted sum than high activations, it might be benefi-

cial to set them to 0. If we do this based on the activation magnitude, the problem is that for higher thresholds more and more activations will be affected. Also as we have seen most activations have only a small value so the impact on the classification accuracy is hard to predict. It is especially unclear what a good threshold value could be.
Before presenting the experiment results, we want to mention some important implementation aspects. In case of the EMA maximum scaling, we have to clip the scaled activations to make sure the values fall into the input range. After the clipping, we then use the thresholding function which sets every value below a certain threshold to 0. We already mentioned the importance of the STE in context of the static scaling and in case of thresholding, it is equally important. If the thresholding is done before the clipping, the gradient behavior of the clipping function (without STE) has to be considered again. If we set the lower clipping value to 0 as well as the target value of the thresholding function then depending on the implementation of the clipping function, gradients do not flow through activations that are affected by thresholding. In the case of the current pytorch implementation, values in the **exclusive** range between the lower and upper clipping value receive the unaltered downstream gradient but for all other values, the gradient is set to 0. In our particular case, we want the lower clipping value to be part of the range. This can be circumvented by using a clipping function with STE, by employing the thresholding function after the clipping function, or by adjusting the lower clipping value of the clipping function with the STE so that the thresholding function becomes unnecessary.
Figure 4.15 contains the results of the best EMA maximum scaling approach with activation pruning. We see that up to a threshold of 3, there is barely any change in accuracy. If we further increase the threshold, the accuracy decreases.

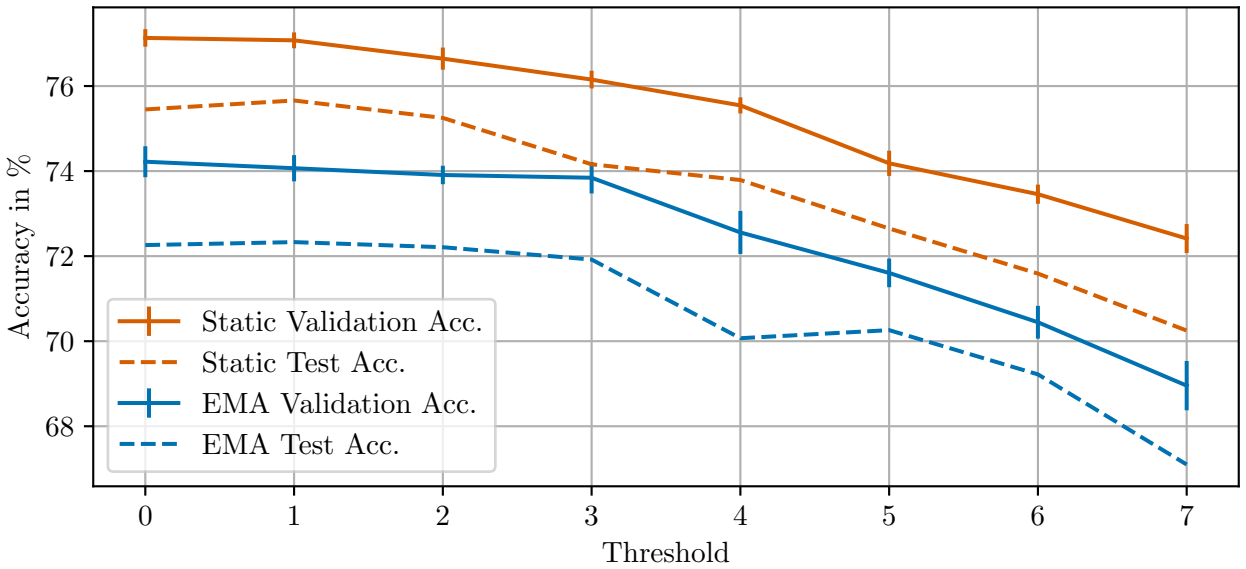

Figure 4.15: Activation pruning results with both static and EMA maximum scaling. The exact values of the data points can be found in Section C.5.

In the case of the static scaling in Figure 4.15 a similar behavior can be observed however the the decrease already starts for a threshold of 2. Because of the noisy training, the final accuracies can be slightly different even if we average the results. The accuracy differences for small thresholds are so small that it is hard to attribute the slight increase in the test accuracy between 0 and 1 to the thresholding.
The lower sensitivity of the EMA scaling to the activation pruning might be related to the previous considerations about the activation distributions and the effective gain of the different scaling approaches. If we assume that the EMA maximum scaling leads to an overall slightly smaller gain, the mentioned insignificance of small activations results in the small sensitivity to magnitude-based pruning for small thresholds. The results of our pruning experiments are yet another indicator

implying that a higher gain (especially for the last two layers) can increase the performance of EMA maximum scaling. This however requires the mentioned layer-specific optimizations.
Contrary to our expectations magnitude-based activation pruning for both static and dynamic scaling does not increase accuracy. We conducted our experiments only to assess whether the noise of small activations outweighs the contribution to the actual computation. Therefore our experiments should not be confused with gradual pruning approaches that focus on model compression. Accuracy results for heavily compressed inputs would be interesting as well since they could also lead to high accuracy. However, this scenario might require even more specialized calibrations with an even higher focus on sufficiently large gain factors due to the sparse input operands.

4.5.4 Percentile Scaling

Besides the thresholding, we also consider manually increasing the weight and activation magnitude with the scaling functions to reach a similar weight distribution as for the static scaling. Instead of using the maximum input value as a scaling reference, we thought of scaling based on a certain percentile of the inputs. We assume that this can make the weight and activation distribution on the hardware for the EMA scaling slightly wider and hence more similar to the static scaling. Besides that, the percentile computation is also less sensitive to outliers than the plain maximum calculation. Even though by using the exponential moving average we already attenuate the effect of outliers, they still contribute to the weighted average. The fundamental disadvantage of this method is that we knowingly clip a certain percentage of large weights/activations and set them to the maximum value. Overall this method is not successful, but we want to share implementation details and experiment results.
For the percentile scaling an important adaption to the EMA maximum scaling must be made. First, we need a more robust scaling computation that avoids division by too small numbers. In case of the maximum scaling, this is not a big issue because we know that the denominator is always bigger than the numerator. The worst case could be that both of them are zero, which can be avoided by a simple conditional expression in advance of the computation. With the addition of the exponential moving average the denominator can become smaller than the numerator, but since the exponential moving average is based on the maximum, its value will be close to the global maximum and hence most of the time bigger than most other values of the numerator. For the percentile scaling this is no longer the case. Since a large part of activations are zero after the ReLU function, the percentile computation is strongly biased towards zero. Hence even for comparably high percentiles, the denominator can become very small so that a stability term usually called ϵ is necessary to avoid overflows.
Of course, we can try to correct the zero bias. Because the scaling factor computation is done purely digitally without any involvement of the analog accelerator, we have full control over every step of the computation. To correct the bias, we can remove all values with a value of 0 from the distribution and compute the percentile from the remaining values. For the activations, this means that we can use only a significantly reduced set of values for the percentile computation. While the filtering makes the pure percentile computation slightly faster, it also makes the result more sensitive to values of the remaining non-zero elements. This can be negative as well because of the involved noise. For the weights, we have positive and negative input values, so similar to before, we only compute the percentile for the absolute values. Asymmetric percentiles for the positive and negative weights are also possible and can potentially improve the clipping behavior for asymmetric weight distributions. The disadvantage of this approach would be a further increase of hyperparameters and more complex rescaling if necessary.
To have a consistent solution, we similarly exclude the zero elements from the weight percentile computation. The scale reference is then computed per batch and with an exponential moving average from all non-zero values after taking the absolute value of the weights or activations.
Since this approach aims at making the weight and activation distribution wider and hence more similar to the results with a static scaling factor, we choose `wait_between_events`=5 to circumvent

saturation. This value corresponds to the best value for the static scaling but for ideal results a parameter sweep over `wait_between_events` would be necessary. We forego this procedure because it is a time-consuming process and not our primary interest. First, we want to find good values for the weight and activation percentiles, the newly introduced hyperparameters with that approach. If the results are promising, we can continue with another sweep over `wait_between_events`.
If we choose the percentiles separately for weights and activations but globally for all layers, we already run into a severe limitation. As we have seen from Figure 4.12 and Figure 4.13 the first layer already chooses a quite narrow weight distribution to avoid extensive clipping of the output activations. The primary layers that appear to have too small weights and activations are *Linear 1* and possibly *Linear 2*.
In Figure 4.16 we see what happens if we choose equal percentiles for all layers and only decrease the weight percentile to 95.

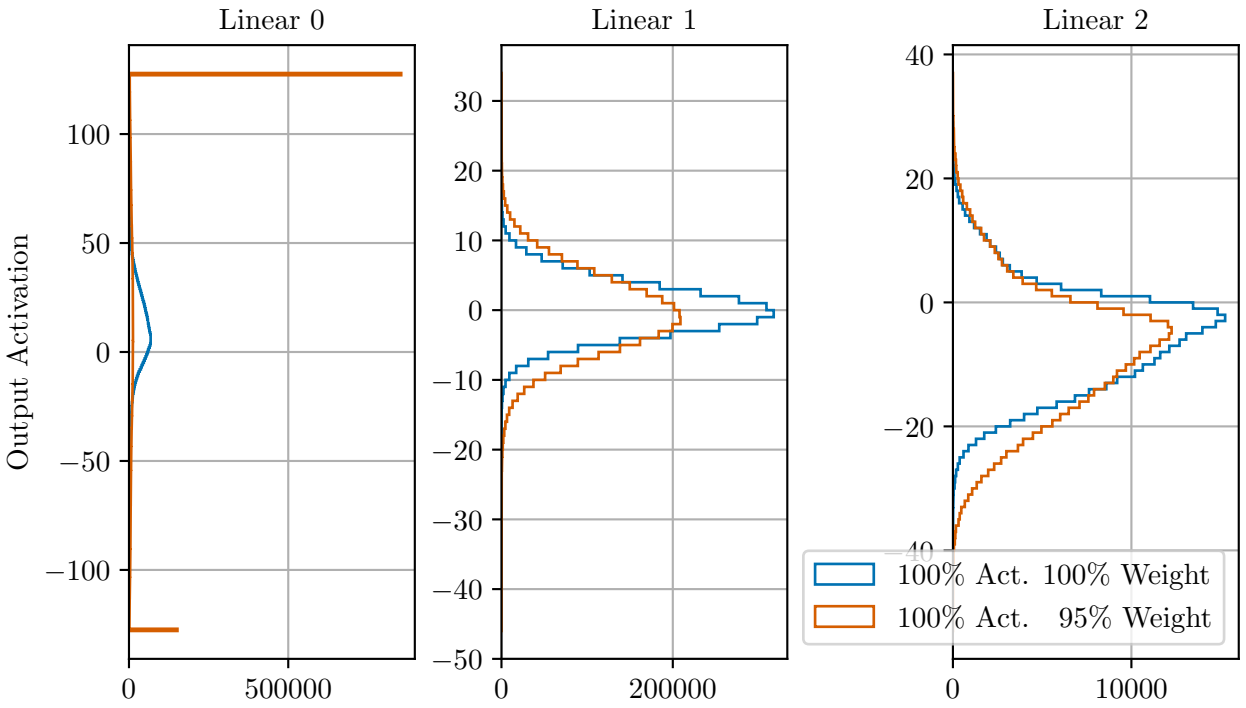

Figure 4.16: Output activation distributions for percentile scaling with equal percentiles on all layers.

The vast majority of all output activations in the first linear layer are clipped to the minimum or maximum value of the output range. While the higher scaling factor causes clipping for most of the activations of *Linear 0* the activation distribution of *Linear 1* covers a visibly wider range of values and also *Linear 2* uses more of the negative output range. That is what we want to achieve with the percentile scaling technique therefore we only apply percentile scaling in the following to *Linear 1* and *Linear 2*. For *Linear 0* we continue the maximum scaling with EMA.

To find suitable values for the last two layers' percentiles, we conduct a grid search and vary the activation and weight percentiles. The results can be seen in Figure 4.17.
The results show that there is indeed a local optimum for the percentiles. For both the validation accuracy in Figure 4.17 (a) and the test accuracy in Figure 4.17 (b) the best results can be achieved for the 95th activation percentile and the 90th weight percentile. In this case, the accuracy is barely higher than the results of EMA maximum scaling and still far below the results of the static scaling. It must be noted that there is further optimization potential by conducting further parameter sweeps over `num_sends` and `wait_between_events`, however, the combinatorial growth in combination with the slow retraining (here ca. 59 hours for the 25 runs) on the hardware makes extensive hyperparameter tuning with this approach very impractical.
Due to the extremely small accuracy increase of 0.05 % in the validation accuracy, we did not conduct further experiments to find optimal parameters for `num_sends` and `wait_between_events`. Assuming a similar distribution as for the static scaling and hence similar behavior as in Table 4.4 an increase of ca. 0.5 % seems to be a realistic order of magnitude.
For further hyperparameter tuning it might also make sense to include the decay factor of the exponential moving average in the search space.
Still, because of the very small accuracy increase and the difficult tuning, we consider this approach to be ineffective.

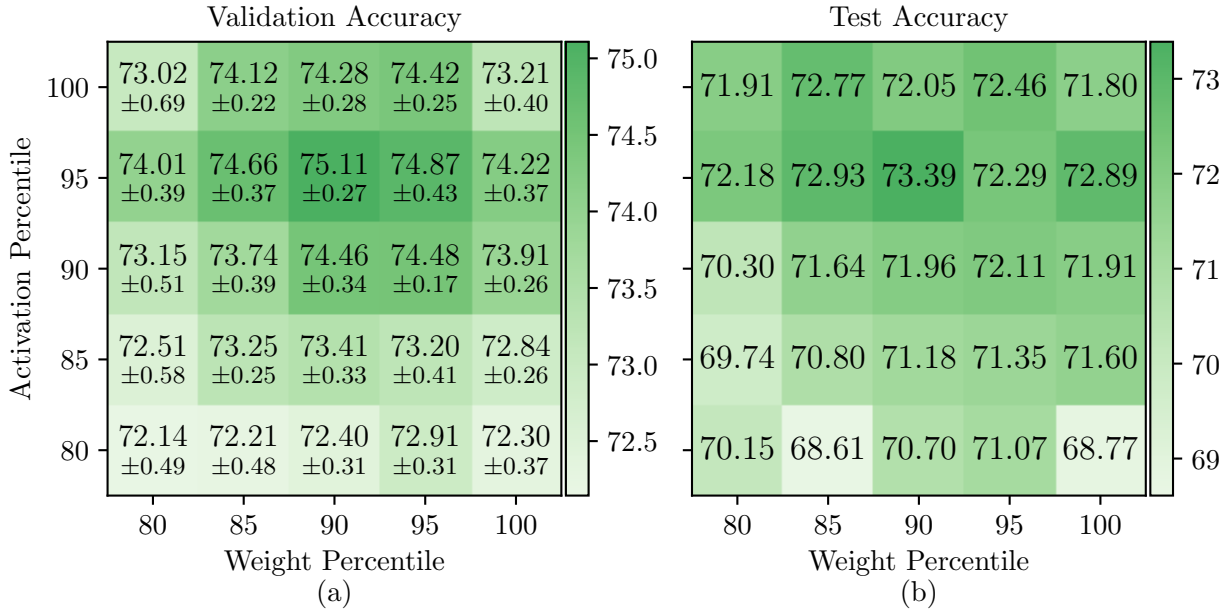

Figure 4.17: Validation accuracy (a) and test accuracy (b) for percentile scaling in the last two layers with zeros excluded from the percentile computation.

The bad behavior might be related to the exclusion of zeros from the percentile computation. A high fluctuation of non-zero elements after the ReLU function can make, the result of the percentile computation equally variably. This is the opposite of our initial intentions and might make it much harder for the network to adjust. However, we did not record the behavior of the scaling factor during training so we cannot be sure. In contrast, if we do not exclude the zeros from the computation a high variability of zero elements could still push the activation percentile more or less toward zero, which would result in similar variations.
It is also possible that the fundamental idea is too invasive to work correctly. By using the percentile as a scaling reference we knowingly clip a significant percentage of the highest weights and activations and set them to the same value. The lack of distinctiveness in these values could revert the benefits of a slightly wider weight distribution. Though the wider distribution should increase the resolution of a majority of small values, intuitively high weights indicate the importance of a

feature so maybe the distinctiveness of these high values is more relevant.
We repeat the experiment without the exclusion of zeros from the percentile computation in Figure 4.18 but because of the zero bias of the activations, we decrease the activation percentile in smaller steps. Due to the inclusion of zeros in the percentile computation, the actual scaling factor should be much smaller for similar percentiles compared to before.
The results show a quite similar trend as in the previous experiment. According to our expectations, the best results can be achieved with a higher percentile compared to the previous results because of the zero bias. However, the best accuracy is below the results of the previous gird search and thereby still smaller than the best accuracy of the static scaling approach and smaller than the best accuracy for the plain EMA maximum scaling. Similar to before a more extensive parameter search could result in a slight accuracy increase. However, our results confirm that including zeros in the percentile computation does not change the general results of percentile scaling.

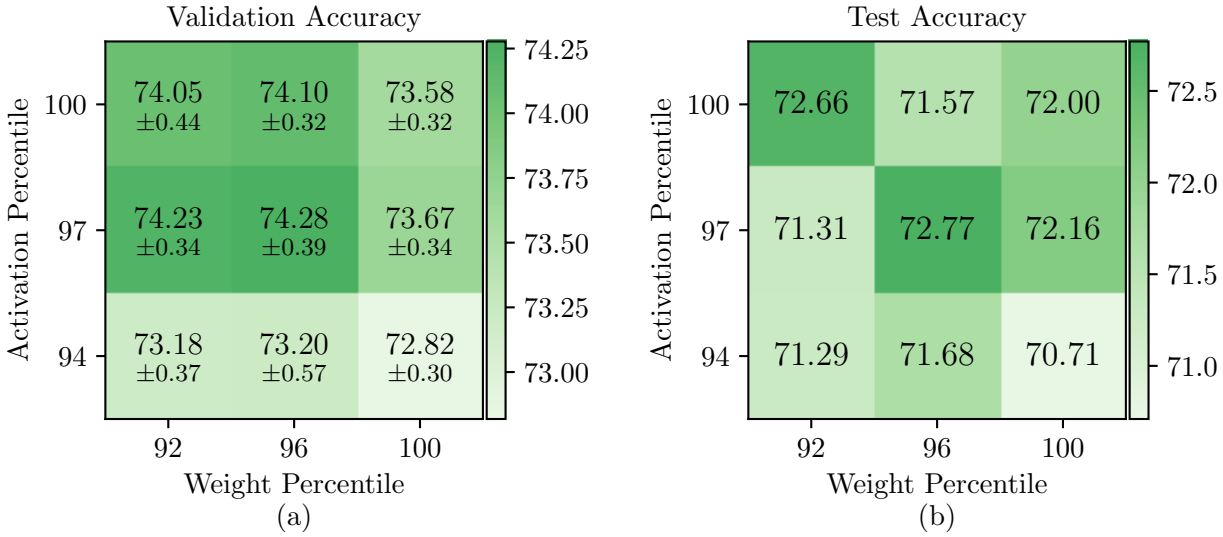

Figure 4.18: Validation accuracy (a) and test accuracy (b) for percentile scaling in the last two layers including zeros in the percentile computation.

Overall the percentile scaling approach turns out to be not only ineffective but also very hard to tune. Ideally, the percentiles should be optimized per layer and in combination with `wait_between_events`. Because there is no analytical procedure to find good value combinations for these hyperparameters without the need for retraining, this approach becomes very unpractical. Since we need hardware-in-the-loop training to adjust to the behavior of the analog accelerator, the training is very slow. This makes hyperparameter tuning with a grid search a very time-consuming process, which becomes immensely worse for additional hyperparameters (here the percentiles) because of the combinatorial growth.

4.5.5 Backprop Through Scale

In this last section, we present the performance of a scaling technique, that we discovered while exploring a bug. It achieves considerable accuracy even with the default calibration and default values for `wait_between_events` and `num_sends`. In this approach, the scaling factor computation is part of the computational graph for the backward computation. Thereby backpropagation significantly exaggerates the importance of the maximum values per batch. This leads to an overall wider distribution and binarization of the weights. Figure 4.19 shows the weight distribution in that setting.
The binarization is clearly visible in all three layers and becomes more and more dominant in deeper layers. If we change the calibration from the default to one with an increased synaptic

input constant, we see only small changes. In general, all layers develop fewer weights that are clipped and in the first linear layer, more values are closer to zero. Because of the increased gain in combination with a high number of input features, the weights of *Linear 0* might not need to be as big as in the default calibration. Why the weight distribution has fewer very high and very low values is however unclear.

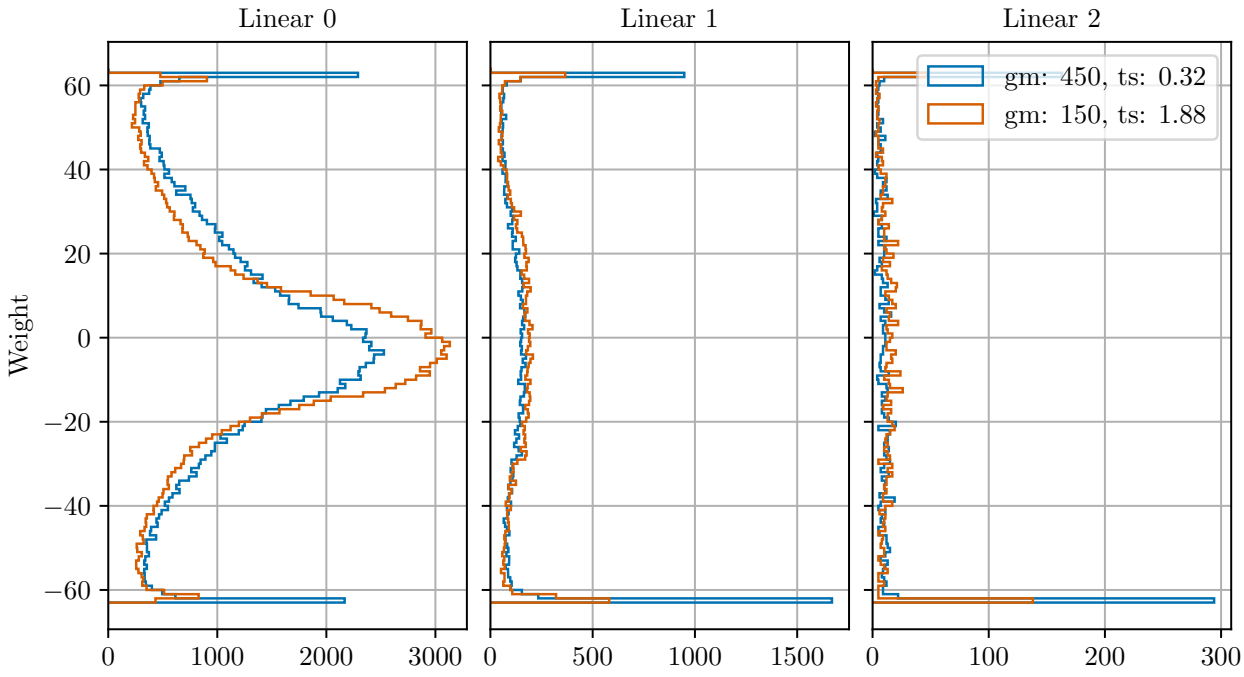

Figure 4.19: Weight distribution with the scaling factor computation as part of backpropagation for default and custom calibration.

In the activation distribution in Figure 4.20 the binarization is not visible. Only in the case of the increased synaptic time constant a huge part of activations in the first layer is clipped. Similar to all other approaches before, the increased synaptic time constant increases gain and hence causes a wider output activation distribution for *Linear 1* and *2*. Again in *Linear 0*, there are more high output activations, but here a very surprising bimodal distribution starts developing.
Both of the previous results are recorded without batch normalization layers between the linear layers. Therefore we also initialize the the model with the floating point reference without batch normalization. The introduction of batch normalization layers makes the training diverge for calibrations with an increased synaptic time constant. Since the histograms for both activations and weights look very similar to the shown results without batch normalization, we omit them at this point. However, they can be found in Section C.6.
The final training results can be seen in Table 4.5. With this scaling approach increasing the global gain by increasing the synaptic input time constant is ineffective. In contrast to our expectations, the training only converges to a similar accuracy as for the default calibration if we omit batch normalization. If we include the batch normalization layers, the training with high synaptic input time constants cannot achieve accuracies anywhere near the results with the default calibration.
Our explanation for that behavior is that the network heavily relies on the binarization aspect. It is quite likely that the wider weight distribution causes saturation on the synaptic input lines. Probably the network trains towards a weight distribution that is robust against noise and saturation because it relies primarily on the presence of features and not so much on their actual magnitude. If we use batch normalization, the binarization of the weights is attenuated and most likely insufficient to tolerate the saturation introduced by the increased synaptic input time constant. Similar to Figure 4.19 the increased synaptic time constant reduces the tendency of the network to develop binary weights. In combination with batch normalization, binarization is almost completely

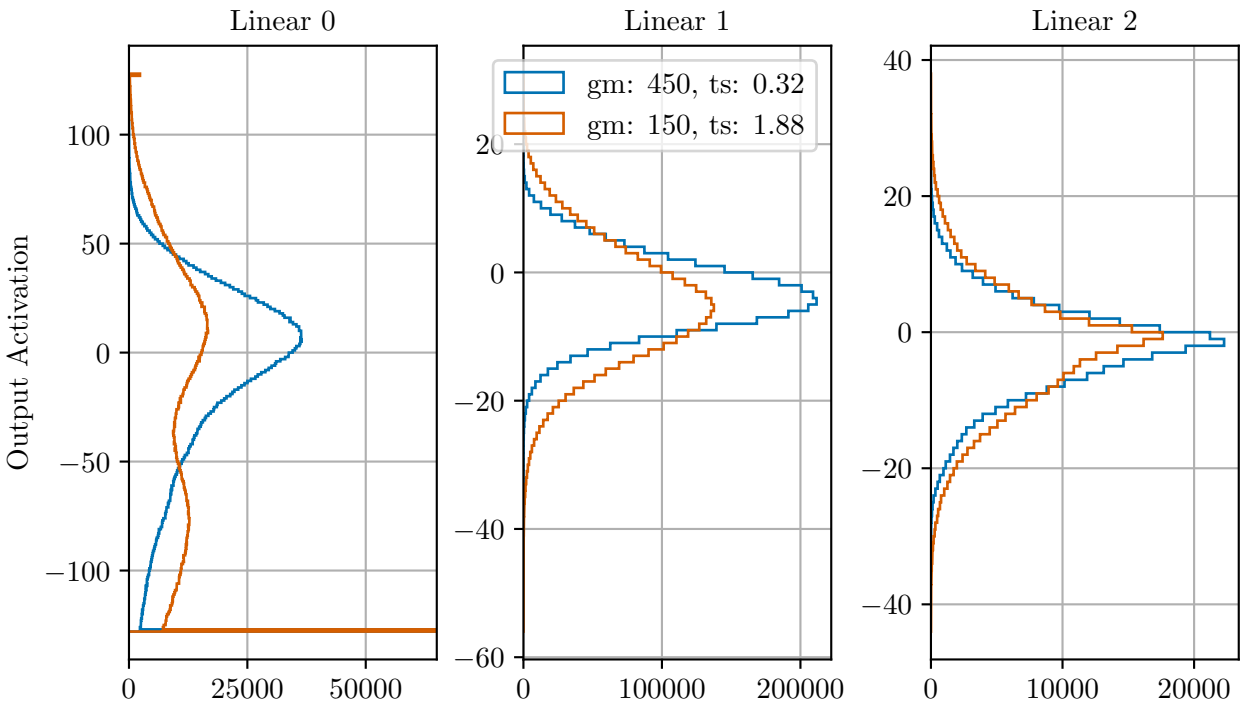

Figure 4.20: Activation distribution with the scaling factor computation as part of backpropagation for default and custom calibration.

avoided in *Linear 1* and *Linear 2* hence the accuracy drops after a few epochs to the values in Table 4.5.
Overall we do not see this scaling technique as a valid approach because it trains a very different weight distribution compared to the floating point initialization. This means it needs a comparably high number of retraining epochs to reach a fully converged state. Additionally, this implies that the binarization degree heavily relies on the number of trained epochs. Since the training only exaggerates the importance of the batch maximum, the process of binarization is quite slow. A training method that targets binarization explicitly is expected to be more effective.
It is also unlikely that all datasets and classification tasks work well with binarized weights because some of the magnitude information is lost. For this application, the results lag behind the achieved results with static scaling and scaling based on the exponential moving average of the batches' maximum. However, the results indicate that binarization can make the hardware imperfections less influential. Increasing the model parameters to compensate for the loss in model capacity caused by binarization might also lead to classification with high accuracy. In that case, there is also optimization potential regarding the chip configuration. Since a fine resolution of activations is not necessary, the slope of the CADC's reference voltage ramp could be increased to its maximum. This lowers the available resolution but speeds up the digitization process.

BatchNorm		no			yes		
OTA gain	150	150	450	150	150	150	450
Synaptic Time Constant	0.32	1.88	0.32	0.32	1.88	2.27	0.32
Mean Validation Acc.	65.91	70.44	68.98	68.56	9.65	8.69	**70.48**
Std. Validation Acc.	0.71	0.48	0.42	0.86	1.64	0.51	1.06
Test Acc.	63.32	67.58	**68.48**	67.70	21.83	11.52	67.84

Table 4.5: Final accuracies for per-batch maximum scaling with the scaling factor computation being part of backprop.

4.5.6 Accuracy Comparison

After several different optimization approaches for better execution of neural networks on the BSS-2 system, we summarize the best accuracies in a conclusive table below.

Scaling	Optimization	Final Test Accuracy	Initial Test Accuracy
Static	Default	68.37	17.20
	ns+wbe	74.50	44.88
	OTA+τ_{syn}+ns+wbe	**75.55**	**57.93**
	pruning+OTA+τ_{syn}+ns+wbe	**75.66**	**58.47**
EMA	Default	59.45	16.06
	ns+wbe	70.79	48.93
	OTA+τ_{syn}+ns+wbe	73.24	58.75
	pruning+OTA+τ_{syn}+ns+wbe	72.85	58.26
	percentile+OTA+τ_{syn}+ns+wbe	73.39	58.39

Table 4.6: Test accuracy comparison of the previous approaches.

While adjustments to `num_sends` and `wait_between_events` can significantly boost the accuracy, adjustments to the static calibration parameters can yield another slight increase. The static scaling approach with our custom calibration and activation pruning reaches the best overall test accuracy. However, the validation accuracy is slightly worse compared to the experiment without pruning (see Section C.5) and the absolute differences are very small. Therefore we still recommend static scaling without activation pruning. A similar result can be seen for the EMA scaling with and without a percentile scaling reference. Due to the increased tuning complexity in the case of percentile scaling, we do not recommend it although the final test accuracies are slightly higher than the ones of EMA scaling with only the custom calibration.
As a general trend static scaling leads to overall better results, yet it is unclear whether static scaling is a feasible option for more complex network architectures. Although the weight distribution during retraining should not change fundamentally, the combined weight changes in multiple layers might lead to a suboptimal operand mapping with static scaling factors. Sakr, Dai, Venkatesan, *et al.* [14, p. 2] also describe this as a fundamental limitation of static quantization scaling. Possibly adjustments to the decay factor of the exponential moving average and a correctly employed rescaling of the output or a different normalization layer after the matrix multiplication would raise the performance of the dynamic scaling to the level of the static approach.
Table 4.6 shows, besides the finally achieved accuracy after the hardware-in-the-loop training, also the accuracies of a plain transfer of the parameters from the floating point model. For the static scaling our optimizations lead to an overall accuracy increase of 41 % before and 7 % after retraining. While we focused in the previous sections only on the final accuracy after hardware-in-the-loop training, we see that the optimization strategies similarly increase the initial accuracy. However, the gap between the final accuracy after retraining is still immense. At the time of writing, we have not explored the potential to reduce the number of retraining epochs, but the remarkably increased initial accuracies indicate that this is possible. This would be a very beneficial result as well since it would reduce the required energy consumption and allow much faster hyperparameter optimization.
From the previous weight distributions we have seen that layer individual optimizations could be more beneficial than globally chosen parameter combinations for the whole network. For the runtime configurable parameters `num_sends` and `wait_between_events` this is already a feasible option. Changing the calibration for every layer, however, is slow and thus impractical in our opinion. Still, we expect that layer-specific optimizations can further increase accuracy.

Chapter 5

Outlook and Summary

After this quite extensive analysis, we use this final chapter to present ideas about further research and propose further steps to make the most out of the presented results. After an outlook into the future with the first section, we will conclude this work with a final summary.

5.1 Future Work

In this section, we present future research directions and potential synergies with previous work done in the context of artificial neural networks on the BSS-2 system.
We start with a paragraph presenting work that continues this thesis and that would allow a broader evaluation of the found results. After that, we present ideas that address challenges slowing down research progress in this field followed by two different ways of future research. Even though we present the two research directions in different paragraphs, we think that neither of these should be continued in isolation. We expect that both of these are important for high accuracy and only a combination of both can eventually reach peak model performance.

Immediate next steps: The final results of the previous chapter indicate that although the custom calibration seems beneficial its potential is limited by the global parameter choice for `num_sends` and `wait_between_events`. To measure the full potential of our proposed custom calibration, layer-specific tuning is required.
Another limitation of this work is that we don't show how our solution generalizes to other datasets. Since the analysis of the HICANN-X chip's behavior and the hardware training and tuning were very time-consuming, we only show results for the three-layer MLP on the SpeechCommands V1 dataset. Therefore it is very important to verify the validity of the shown results on other datasets. Besides that, the scaling methodology of the floating point weights and activations to the hardware should be revised before any new studies are conducted. As we mentioned, we adopt the scaling methodology of Klein, Kuhn, Weis, *et al.* [19], which relies on `hxtorch`'s dynamic scaling functions. However, after the matrix multiplications, the scaling factors are discarded and not considered in further stages. We also mentioned that we assume the used batch normalization to counteract the increased internal covariance shift of that methodology but possibly other normalization techniques like layer normalization would have a better compensation effect. Since we use a classification task based on an *argmax* classification the variable magnitude of the batches might not be a severe problem, but we expect negative effects for other prediction tasks like regression. In that context, we also expect the lack of research on how to treat the varying and often unknown intrinsic scaling factor of the analog matrix multiplication to be problematic.

Improving training and analysis: During our experiments, we noticed that the time-consuming process of hardware-in-the-loop training can slow down research progress quite significantly. Often parameter sweeps are required to find behavioral trends, but for that, the model must be retrained

several times. In contrast to digital training, these training runs must be conducted on the same hardware instance to make results comparable because of the unique imperfections in every chip. This makes training even more time-consuming.
However, our training results suggest that the custom calibration, with its improved initial accuracy, could potentially lead to a significant reduction in training time. To confirm this, further research is necessary that compares achievable accuracies for fewer retraining epochs and possibly adjusted learning rates.
Besides that, we assume that the presented hardware model of Klein, Kuhn, Weis, *et al.* [19] has great potential to significantly boost further research on artificial neural networks on the BSS-2 system. To combine our research with the work of Klein, Kuhn, Weis, *et al.*, we propose the following steps.
Firstly it is important to check whether the training results of the hardware model in [19] have been recorded with the explained bug in the `hxtorch` scaling function. If that is the case, it is important to find out whether the resulting binarization has biased the results.
Then it would be very helpful to determine whether the hardware model can be used for parallelizing parameter sweeps to accelerate research on addressing imperfections introduced by analog hardware. Possibly even the modeling and the recording of the model behavior can be improved with the results of chapter 3.
After that, additional research areas of interest include investigating whether the improved custom calibration in conjunction with the gradual introduction of imperfections by using the hardware model can further increase accuracy.
Other generally very useful contributions for the neural network analysis would be a way to quantify the effects of dynamic saturation during training and a way to accurately capture the intrinsic gain of the analog matrix multiplication. After conducting this research we find both parameters to be very important factors for good model performance.

Adjusting models to the hardware: If we want to further improve model performance with model-specific optimizations, we can analyze the effects for `num_sends` and `wait_between_events` optimized for each layer. We recommend layer-specific optimizations for only these parameters since these are runtime parameters. The other calibration parameters must be set globally during the chip initialization. We assume that finding good values for these parameters on a per-layer basis is possible even without a grid search. However, this requires developing a methodology for that. With the insights of this work, it should become possible to determine suitable parameters based on the noise and gain increase of `num_sends`, `wait_between_events`, and the number of input features. Possibly even the value distributions of the operands can be considered. In that context, it would be very helpful to develop a methodology for determining how increasing the gain affects the output distribution of a layer. Since we found that the number of input features affects the gain and noise, it is possible that also a specific layer design can improve the computation on the analog hardware. In this context, the digital accumulation of `hxtorch`'s partitioned analog matrix multiplication for more than 128 input features is another optimization point. Currently, the accumulation is done in 8-bit, but for many input features a larger accumulator could preserve accuracy and avoid clipping. Since the accumulation is done digitally anyway, this improvement should be straightforward.
A further development step would be to dynamically adjust `num_sends` and `wait_between_events` during training to compensate for dynamic saturation and too-small output amplitudes. This however appears quite complex because of the mentioned lack of measuring techniques for saturation and the effects of it. Also, it seems possible that an advanced scaling concept of the input operands can also compensate for these problems.
Since our work only showed results for optimizations targeting linear layers it would be an important next step to extend our work to the optimization of convolutional neural networks. We expect convolution layers to behave significantly differently because their filter kernels usually contain much fewer weights than linear layers. Hence we expect a need for higher gain. Especially, finding

a good calibration for a network that combines both convolutions and linear layers seems like another involved task.
A more general approach for further optimizations would be to combine the results of this work with the approach used by L. Kuhn [41]. In their work, they improve the model performance by adjusting the training so that the model avoids high output values. Instead of the originally stated reason for better linearity in a smaller output range, we propose a different reason. The results of Figure 4.3 show that smaller outputs usually benefit from smaller noise. With our custom calibration, this effect is even more pronounced so we expect even better results with the technique proposed by L. Kuhn.
Due to the similarly increasing noise for increasing weights, non-uniform quantization approaches might also lead to a more beneficial behavior. However, as mentioned a fundamentally sound scaling technique should be developed first.

Adjusting the hardware to the models: In this work, we only analyzed a subset of the available parameters in the calibration. Based on the results of this work, we expect other parameters to have a significant impact as well. Specifically, we recommend further analysis of the membrane capacitance, the membrane leak resistance and the maximum current of the synapses (synapse bias current).
We expect that the membrane capacitance could be used as well to increase the overall gain of the operation. Since it does not increase the risk of dynamic saturation unlike the synaptic input time constant, it could be a beneficial addition to also tune the membrane capacitance.
For the leak resistance, it would be very interesting to see the model performance for an increased leak current. Although we expect the performance to decrease because the leak current causes an unfair contribution of weight-activation pairs to the overall result, it would be insightful to see how detrimental the effects are and whether retraining can compensate for this. A small leak current could make longer integration times more feasible since it decreases the impact of slightly offset reference voltages on the OTA inputs. Measuring the effect on the overall noise would be a valuable contribution as well.
Similarly measuring the effect of the synapse bias current on the output noise, overall gain and dynamic saturation would further broaden the understanding of the chip's peculiarities.
A fundamentally different component that we have not analyzed is the CADC. Since its reference voltage decides how big the outputs appear, it is also a way to increase the gain globally. Besides that, it allows a trade-off in which we sacrifice one bit of the output resolution for faster digitization by increasing the reference voltage slope. Since the output activations are scaled in successive layers anyway to the lower input resolution this seems like an insignificant loss. The faster digitization however could lead to shorter integration times and our results show that this leads to smaller noise (see Figure 3.9). Since the chip does not feature a sample-and-hold circuit the digitization is part of the integration time.

5.2 Summary

In this work, we use the neuromorphic BrainScaleS-2 system as an analog accelerator for the execution of artificial neural networks.
By using its operation mode for analog matrix multiplication, we analyze the typical imperfections introduced by analog hardware like noise, static-offsets and saturation to find ways to optimize for better execution of neural networks even in the presence of these imperfections.
Fundamentally our optimizations are twofold. On one side, we analyze and improve the behavior of the analog hardware, on the other side, we develop algorithmic adjustments to the imperfections.
The calibration of the BrainScaleS-2 system allows adjustments to the analog execution of the matrix multiplication, hence we see this as an opportunity for application-specific optimizations. By adjusting the calibration we try to align the operational characteristics of the system with the demands of neural networks.

In the first step, we measure the imperfections of the system and find ways to influence them. Because there are many parameters with unknown effects on the imperfections, we decide on a subset of them based on the analysis of previous research. We decide on a deeper analysis of the synaptic input time constant, the gain of the operational transconductance amplifiers and the two runtime parameters `num_sends` and `wait_between_events`, which control the timing and execution behavior of the analog matrix multiplication. Besides that, we conduct experiments to investigate the effects of both input operands. Our results show that all of the mentioned parameters increase the noise however to a different degree and we find that they influence different components of the noise. While we find that `num_sends`, `wait_between_events`, and also the number of input features linearly increase the additive component of the noise, only `num_sends` and the number of input features increase the multiplicative noise. Because of that, our optimized calibration tries to choose `num_sends` and `wait_between_events` to be as small as possible.

Regarding the input operands, we find small activations to be particularly noisy and relate this to the inverse principle of the synapse drivers that require high voltages for small inputs. The weights have compared to the activations a more gradual and overall bigger impact on the noise. Our research shows that the noise strongly varies for different weight values but as a general trend, it grows at least linearly with the weight magnitude.

Regarding the OTA gain our experiments show that by decreasing the gain the noise floor can be reduced. Still, the noise grows for higher output values but the average standard deviation for small output values is generally reduced. However, a too-small OTA gain limits the output values to only a fraction of the available output range.

With this knowledge, we trade off multiple effects to find an overall improved calibration. We use a small OTA gain to reduce the noise and recover the loss in gain by increasing the synaptic input time constant. This increase also raises the potential for dynamic saturation, which alters the results quite significantly and should definitely be avoided.

Further analysis of the dynamic saturation shows that its effects are small even if we increase the synaptic input time constant quite significantly. With a `wait_between_events` of 4 and considering the typical weight and activation distribution of neural networks with many weights and activations close to zero, we expect low saturation effects and verify the improved behavior by training a small 3-layer MLP with hardware in the loop.

On the algorithmic side, we try different quantization scaling approaches and magnitude-based activation pruning to optimize model performance. Because of the increased noise of small activations, we expect activation pruning to be a countermeasure however our experiments show barely any change in accuracy for small pruning thresholds. For higher values, the accuracy starts to decrease.

Regarding scaling techniques, we try both static and dynamic scaling during retraining on the hardware. For the static scaling, we initialize the scaling factors based on the exponential moving average of the maximum in each batch. We determine these values at the start of the training on the hardware in a single epoch where we only execute the forward pass. Then we freeze the scaling factors for the rest of the retraining. With this methodology, we reach the highest accuracy results after retraining.

For the dynamic approach we try maximum scaling for each batch, maximum scaling based on an exponential moving average and percentile scaling based on an exponential moving average. Because of the bad performance of the maximum scaling per batch, we discarded this approach immediately at the start of our evaluation. The other approaches perform better but their accuracy remains below the static scaling approach. The substantial increase in the number of hyperparameters introduced by percentile scaling, along with the long retraining time, makes this approach considerably less appealing.

A deeper analysis of the weights and activations in the different approaches shows that with static scaling the model trains towards slightly higher weights. This seems especially helpful for the second and third layers of our used MLP since the output activations reached only very small values in these layers and hence barely used the available output resolution.

We attribute the small output magnitude to the smaller number of input elements in the second and third layers. Since we use global calibration parameters the outputs of the first layer are well distributed but for the following layers, the gain is too small to reach sufficiently large output values.
Percentile scaling for the second and third layers to manually force the network to higher input operands barely increases output values and hence does not increase the accuracy. This means that either many summands in the dot products of the matrix multiplication cancel out or that the number of input features has a very high impact on the output magnitude. In both cases, a higher intrinsic gain of the matrix multiplication could solve this problem.
Therefore we expect that gain adjustments for layers with fewer inputs can further increase accuracy. Since most calibration parameters are initialized statically and reinitialization is slow only `num_sends` appears as a feasible option to increase the gain of individual layers. Possibly this also requires adjustments to `wait_between_events`. This analysis however is part of future research on this topic.
Still, if we compare the final accuracies of the default calibration with our custom calibration, we reach an accuracy increase of around 7 % after retraining and an accuracy increase of circa 41 % before retraining. This is reduced to 1.1 % respectively 13.1 % if we use the results with optimized `num_sends` and `wait_between_events` as a baseline.
However our best results are still around 10 % worse than the digital floating-point accuracy, but we expect that the mentioned layer-specific tuning can further narrow this gap. Also, we expect that the layer-specific tuning of `num_sends` and `wait_between_events` allows a better evaluation of the potential of the custom calibration since it currently appears to be limited by the mentioned behavior for too few input features. Based on our analysis of the parameter effects, tuning guidelines can be derived.

Overall this work contributes valuable insights into the effects of calibration parameters on accuracy and the imperfections of the analog matrix multiplication of the BrainScaleS-2 system. We provide guidelines for good parameter choices and report accuracy increases for both the accuracy before and after retraining with our custom calibration. Besides that, we also show results for algorithmic optimizations and uncover challenges regarding the design of neural networks when used on this system. We expect this work to foster a broader understanding of the BrainScaleS-2 system and how it can be used efficiently and competitively for the execution of neural networks.

Bibliography

[1] N. C. Thompson, K. Greenewald, K. Lee, and G. F. Manso, *The computational limits of deep learning*, 2022. arXiv: 2007.05558 [cs.LG].

[2] F. Rosenblatt, *The perceptron: A probabilistic model for information storage and organization in the brain.* 1958.

[3] P. J. Werbos, *Applications of advances in nonlinear sensitivity analysis*, Jan. 1982. [Online]. Available: https://link.springer.com/content/pdf/10.1007/BFb0006203.pdf?pdf=inline%20link.

[4] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.

[5] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of control, signals and systems*, vol. 2, no. 4, pp. 303–314, 1989.

[6] S. Ioffe and C. Szegedy, *Batch normalization: Accelerating deep network training by reducing internal covariate shift*, 2015. arXiv: 1502.03167 [cs.LG].

[7] S. Santurkar, D. Tsipras, A. Ilyas, and A. Madry, *How does batch normalization help optimization?* 2019. arXiv: 1805.11604 [stat.ML].

[8] X. Li, S. Chen, X. Hu, and J. Yang, *Understanding the disharmony between dropout and batch normalization by variance shift*, 2018. arXiv: 1801.05134 [cs.LG].

[9] H. Borras, B. Klein, and H. Fröning, *Walking noise: Understanding implications of noisy computations on classification tasks*, 2022. arXiv: 2212.10430 [cs.LG].

[10] W. Sung, S. Shin, and K. Hwang, *Resiliency of deep neural networks under quantization*, 2016. arXiv: 1511.06488 [cs.LG].

[11] C. Sakr, Y. Kim, and N. Shanbhag, "Analytical guarantees on numerical precision of deep neural networks," in *International Conference on Machine Learning*, PMLR, 2017, pp. 3007–3016.

[12] J. Su, N. J. Fraser, G. Gambardella, *et al.*, *Accuracy to throughput trade-offs for reduced precision neural networks on reconfigurable logic*, 2018. arXiv: 1807.10577 [cs.CV].

[13] H. Wu, P. Judd, X. Zhang, M. Isaev, and P. Micikevicius, *Integer quantization for deep learning inference: Principles and empirical evaluation*, 2020. arXiv: 2004.09602 [cs.LG].

[14] C. Sakr, S. Dai, R. Venkatesan, B. Zimmer, W. Dally, and B. Khailany, "Optimal clipping and magnitude-aware differentiation for improved quantization-aware training," in *International Conference on Machine Learning*, PMLR, 2022, pp. 19 123–19 138.

[15] Y. Bengio, N. Léonard, and A. Courville, *Estimating or propagating gradients through stochastic neurons for conditional computation*, 2013. arXiv: 1308.3432 [cs.LG].

[16] S. Chen, W. Wang, and S. J. Pan, "Metaquant: Learning to quantize by learning to penetrate non-differentiable quantization," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[17] B. Jacob, S. Kligys, B. Chen, *et al.*, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 2704–2713.

[18] S. Schmitt, J. Klahn, G. Bellec, *et al.*, "Neuromorphic hardware in the loop: Training a deep spiking network on the BrainScaleS wafer-scale system," in *2017 International Joint Conference on Neural Networks (IJCNN)*, IEEE, May 2017. DOI: 10.1109/ijcnn.2017.7966125.

[19] B. Klein, L. Kuhn, J. Weis, *et al.*, "Towards addressing noise and static variations of analog computations using efficient retraining," in *Machine Learning and Principles and Practice of Knowledge Discovery in Databases*, ser. Communications in Computer and Information Science, M. Kamp, I. Koprinska, A. Bibal, *et al.*, Eds., vol. 1524, Springer International Publishing, 2021, pp. 409–420, ISBN: 978-3-030-93735-5. DOI: 10.1007/978-3-030-93736-2_32.

[20] J. Schemmel, S. Billaudelle, P. Dauer, and J. Weis, *Accelerated analog neuromorphic computing*, 2020. arXiv: 2003.11996 [cs.NE].

[21] S. Billaudelle, J. Weis, P. Dauer, and J. Schemmel, *An accurate and flexible analog emulation of adex neuron dynamics in silicon*, 2022. arXiv: 2209.09280 [cs.NE].

[22] M. Rudolph and A. Destexhe, "Analytical integrate-and-fire neuron models with conductance-based dynamics for event-driven simulation strategies," *Neural computation*, vol. 18, no. 9, pp. 2146–2210, 2006.

[23] S. Friedmann, J. Schemmel, A. Grübl, A. Hartel, M. Hock, and K. Meier, "Demonstrating hybrid learning in a flexible neuromorphic hardware system," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 11, no. 1, pp. 128–142, 2017. DOI: 10.1109/TBCAS.2016.2579164.

[24] E. Müller, E. Arnold, O. Breitwieser, *et al.*, "A scalable approach to modeling on accelerated neuromorphic hardware," *Frontiers in Neuroscience*, vol. 16, 2022, ISSN: 1662-453X. DOI: 10.3389/fnins.2022.884128. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fnins.2022.884128.

[25] Electronic Vision(s) Group, *Hxtorch*, 2023. [Online]. Available: https://github.com/electronicvisions/hxtorch (visited on 2023-10-04).

[26] P. Spilger, "From neural network descriptions to neuromorphic hardware — a signal-flow graph compiler approach," Master's Thesis, Heidelberg University, 2021. [Online]. Available: https://www.kip.uni-heidelberg.de/veroeffentlichungen/details.php?id=4172 (visited on 2023-09-23).

[27] P. Spilger, E. Müller, A. Emmel, *et al.*, "hxtorch: PyTorch for BrainScaleS-2," in *IoT Streams for Data-Driven Predictive Maintenance and IoT, Edge, and Mobile for Embedded Machine Learning*, ser. Communications in Computer and Information Science, J. Gama, S. Pashami, A. Bifet, *et al.*, Eds., Springer International Publishing, 2020, pp. 189–200, ISBN: 978-3-030-66769-6. DOI: 10.1007/978-3-030-66770-2_14. arXiv: 2006.13138 [cs.NE].

[28] Electronic Vision(s) Group, *Calix*, 2023. [Online]. Available: https://github.com/electronicvisions/calix (visited on 2023-10-04).

[29] A. Paszke, S. Gross, F. Massa, *et al.*, *Pytorch: An imperative style, high-performance deep learning library*, 2019. arXiv: 1912.01703 [cs.LG].

[30] J. Weis, "Inference with artificial neural networks on neuromorphic hardware," Master's Thesis, Heidelberg University, 2020. [Online]. Available: https://www.kip.uni-heidelberg.de/veroeffentlichungen/details.php?id=4103 (visited on 2023-08-23).

[31] J. Weis, P. Spilger, S. Billaudelle, *et al.*, "Inference with artificial neural networks on analog neuromorphic hardware," in *IoT Streams for Data-Driven Predictive Maintenance and IoT, Edge, and Mobile for Embedded Machine Learning*, ser. Communications in Computer and Information Science, J. Gama, S. Pashami, A. Bifet, *et al.*, Eds., Springer International Publishing, 2020, pp. 201–212, ISBN: 978-3-030-66769-6. DOI: 10.1007/978-3-030-66770-2_15. arXiv: 2006.13177 [cs.NE].

[32] F. L. Ebert, "Real-time image classification on analog neuromorphic hardware," Bachelor's Thesis, Heidelberg University, 2021. [Online]. Available: https://www.kip.uni-heidelberg.de/Veroeffentlichungen/details.php?id=4230 (visited on 2021-09-09).

[33] A. F. Murray and P. J. Edwards, "Enhanced MLP performance and fault tolerance resulting from synaptic weight noise during training," *IEEE Transactions on Neural Networks*, vol. 5, no. 5, pp. 792–802, 1994, ISSN: 1045-9227. DOI: 10.1109/72.317730.

[34] C. H. Sequin and R. D. Clay, "Fault tolerance in artificial neural networks," in *1990 IJCNN International Joint Conference on Neural Networks*, (San Diego, CA, USA, Jun. 17, 1990), IEEE, 1990, 703–708 vol.1. DOI: 10.1109/IJCNN.1990.137651.

[35] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, *Improving neural networks by preventing co-adaptation of feature detectors*, 2012. arXiv: 1207.0580 [cs.NE].

[36] C. Torres-Huitzil and B. Girau, "Fault and Error Tolerance in Neural Networks: A Review," *IEEE Access*, vol. 5, pp. 17 322–17 341, 2017. DOI: 10.1109/ACCESS.2017.2742698.

[37] J. Schemmel, J. Fieres, and K. Meier, "Wafer-scale integration of analog neural networks," in *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, (Hong Kong, China, Jun. 1, 2008), IEEE, 2008, pp. 431–438, ISBN: 978-1-4244-1820-6. DOI: 10.1109/IJCNN.2008.4633828.

[38] A. Kalampokis, C. Kotsavasiloglou, P. Argyrakis, and S. Baloyannis, "Robustness in biological neural networks," *Physica A: Statistical Mechanics and its Applications*, vol. 317, no. 3-4, pp. 581–590, 2003, PII: S0378437102013407, ISSN: 03784371. DOI: 10.1016/S0378-4371(02)01340-7.

[39] A. Emmel, "Inference with convolutional neural networks on analog neuromorphic hardware," Master's Thesis, Heidelberg University, 2020. [Online]. Available: https://www.kip.uni-heidelberg.de/veroeffentlichungen/details.php?id=4122 (visited on 2023-08-23).

[40] Y. LeCun, C. Cortes, and C. Burges, "Mnist handwritten digit database," *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, vol. 2, 2010.

[41] L. Kuhn, "Quantized neural networks for keyword spotting on neuromorphic hardware," Bachelor's Thesis, Heidelberg University, 2020.

[42] B. Jacob and P. Warden, *Gemmlowp: A small self-contained low-precision gemm library*, 2023. [Online]. Available: https://github.com/google/gemmlowp (visited on 2023-10-04).

[43] L. Papula, *Mathematik für Ingenieure und Naturwissenschaftler Band 2*, German, 14th ed. Wiesbaden: Springer Fachmedien Wiesbaden, 2015, ISBN: 978-3-658-07789-1. DOI: 10.1007/978-3-658-07790-7.

[44] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, eng, Fourth edition. Cambridge, Massachusetts and London, England: The MIT Press, 2022, 1291 pp., ISBN: 978-0262046305.

[45] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities, Reprinted from the afips conference proceedings, vol. 30 (atlantic city, n.j., apr. 18-20), afips press, reston, va., 1967, pp. 483-485, when dr. amdahl was at international business machines corporation, sunnyvale, california," *IEEE Solid-State Circuits Newsletter*, vol. 12, no. 3, pp. 19–20, 2007, ISSN: 1098-4232. DOI: 10.1109/N-SSC.2007.4785615.

[46] O. Leveque, *Random walks: An introduction*, Lecture Notes, 2014. [Online]. Available: https://ipgold.epfl.ch/~leveque/Lecture_Notes/random_walks.pdf (visited on 2023-09-17).

[47] B. Cramer, S. Billaudelle, S. Kanya, *et al.*, *Surrogate gradients for analog neuromorphic computing*, 2021. arXiv: 2006.07239 [cs.NE].

[48] P. Warden, "Speech commands: A public dataset for single-word speech recognition.," *Dataset available from http://download.tensorflow.org/data/speech_commands_v0.01.tar.gz*, 2017.

[49] P. Warden, *Speech commands: A dataset for limited-vocabulary speech recognition*, 2018. arXiv: 1804.03209 [cs.CL].

[50] B. McFee, C. Raffel, D. Liang, *et al.*, "Librosa: Audio and music signal analysis in python," in *Proceedings of the 14th Python in Science Conference*, (Austin, Texas, Jul. 6, 2015), ser. Proceedings of the Python in Science Conference, SciPy, 2015, pp. 18–24. DOI: 10.25080/Majora-7b98e3ed-003.

[51] S. S. Stevens and J. Volkmann, "The relation of pitch to frequency: A revised scale," *The American Journal of Psychology*, vol. 53, no. 3, pp. 329–353, 1940.

[52] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2017. arXiv: 1412.6980 [cs.LG].

[53] I. Loshchilov and F. Hutter, *Sgdr: Stochastic gradient descent with warm restarts*, 2017. arXiv: 1608.03983 [cs.LG].

Appendix A

Abbreviations

AdEx Adaptive-Exponential Integrate-and-Fire

ANN Artificial Neural Network

CADC Columnar Analog-Digital Converter

DAC Digital-Analog Converter

DNN Deep Neural Network

EMA Exponential Moving Average

FPGA Field-Programmable Gate Array

LIF Leaky Integrate-and-Fire

MLP Multilayer Perceptron

OTA Operational Transconductance Amplifier

PPU Plasticity Processing Unit

PTQ Post Training Quantization

PWM Pulse-Width Modulation

SIMD Single Instruction Multiple Data

QAT Quantization Aware Training

SNN Spiking Neural Network

SNR Signal-to-Noise Ratio

STE Straight-Through Estimator

Appendix B

Experiment Environment

For reproducibility, we provide the repository versions of the used software in Table B.1.

Table B.1: Software environment used for all experiments.

Repository	Commit Hash
hxtorch	50235e9a6f945e1947bdda2051084bf19dd5c7da
calix	bc485dad6ad511f88be6d739f16455a033c9ee8f
hxcomm	7580a638b09228b158d1c6eae2c2a31405ae383f
haldls	907408a96bbfdc3cd14e0778ee72c843871b8fdf
grenade	ef8ddca430b317dd6113405ae179e2e4fbcb08c3
code-format	24b533dd390253f5c698708fa735283c2e7282ca
sctrltp	42a988e986906f177102813418d5fd22dd646b44
nhtl-extoll	2d7098e2364141ebc0db2a71982570a46d68c7b8
rant	0d494ce6eedfb74889cf7cee09105258819acb35
hate	28780e4fd56b9994e7f3a815e294a3dfd7c64b55
hwdb	e738de3d2ed818ae9a0e9f397c1b570df85d7ea4
logger	00380efdec521fb08df4a083a1d1f443fef836e4
visions-slurm	8f41ea4f5bd1573d8f4623e9ed698a29f30036a3
flange	298335251daf2d2a02764d888e02566934ad2ec9
lib-rcf	4ac48ea216e1e9026cd0d25f52a4bf683d97a189
bss-hw-params	8e5e18ebbdece63890eebd4f082084111846e965
halco	aae9a05d2f366e809327d112c161caa055eaabb0
fisch	01fdc09285f8c5c71927c1090fcc8d6d64c809a6
libnux	9b335cb56b6faa447ffda27f5ff310e648502071
librma	5159e3c602e0133c74800a33a4042f7aeebfb00f
pywrap	8eda91fcca8bfccb946a0ee5b40ca82b5b15650e
ztl	773660f435e56b1ee7b962e8babfe004ff487cdd
lib-boost-patches	ed89665b4c066629b69617ede2e8b1fbe65822d9
extoll-driver	a0ffdc9ea5517e11bc126c0b9d54e7dca2f1dc07

The experiments of Chapter 3 were conducted on setup 66/0. All other experiments were conducted on setup 63/3. The only reason for that was the constrained availability of the hardware resources since they were shared among multiple users.

Appendix C

Additional Measurements

C.1 Additional Noise Results

Since we show in Figure 4.3 only the noise for positive outputs and since there could be an asymmetry, we also include our measurement results for negative outputs seen in Figure C.1. Similar to Figure 4.3 these results have been recorded for all activations with a step size of 3 from 1 to 31 but now for all negative weight values. We use `wait_between_events`=5, `num_sends`=1 and 64 input features as we did for the positive outputs. Here we filter values with a mean smaller than −122 and and activations with a value below 10.

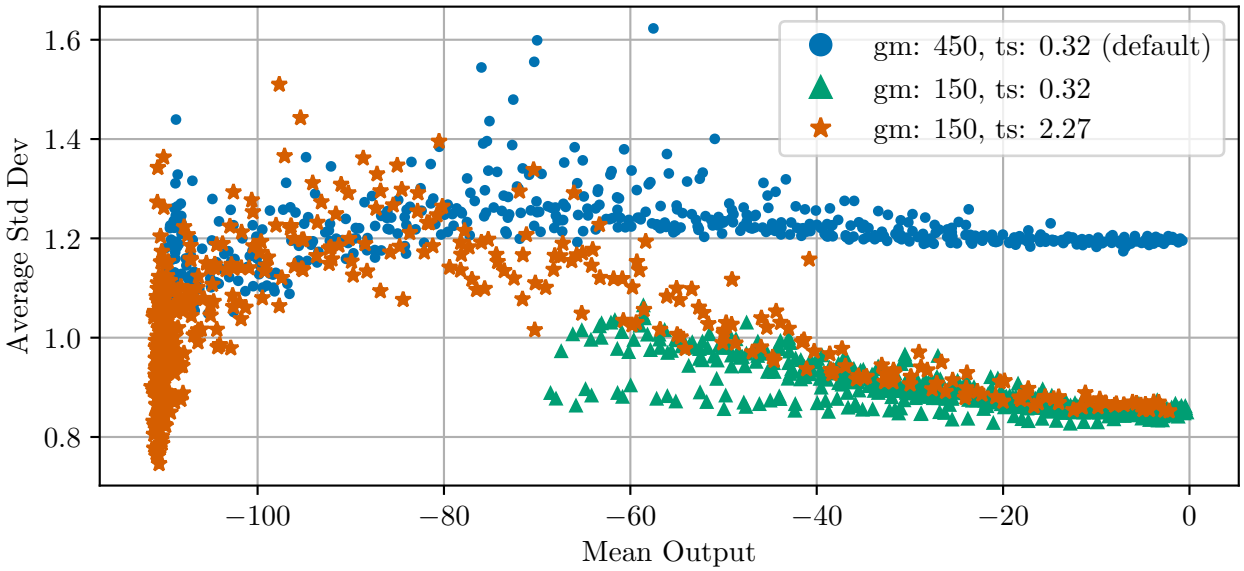

Figure C.1: Average standard deviation over the negative mean output of the matrix multiplication sampled for all weight combinations and recorded for the default and custom calibrations.

C.2 Additional Training Results for Different OTA Gains

Increasing the synaptic input time constant indeed causes very similar accuracy improvements for different OTA gains. Here we show them side by side with a close-up of the final accuracies in Figure C.4. For convenience, we include Figure C.2 from page 58, followed by the mentioned results in Figure C.3.

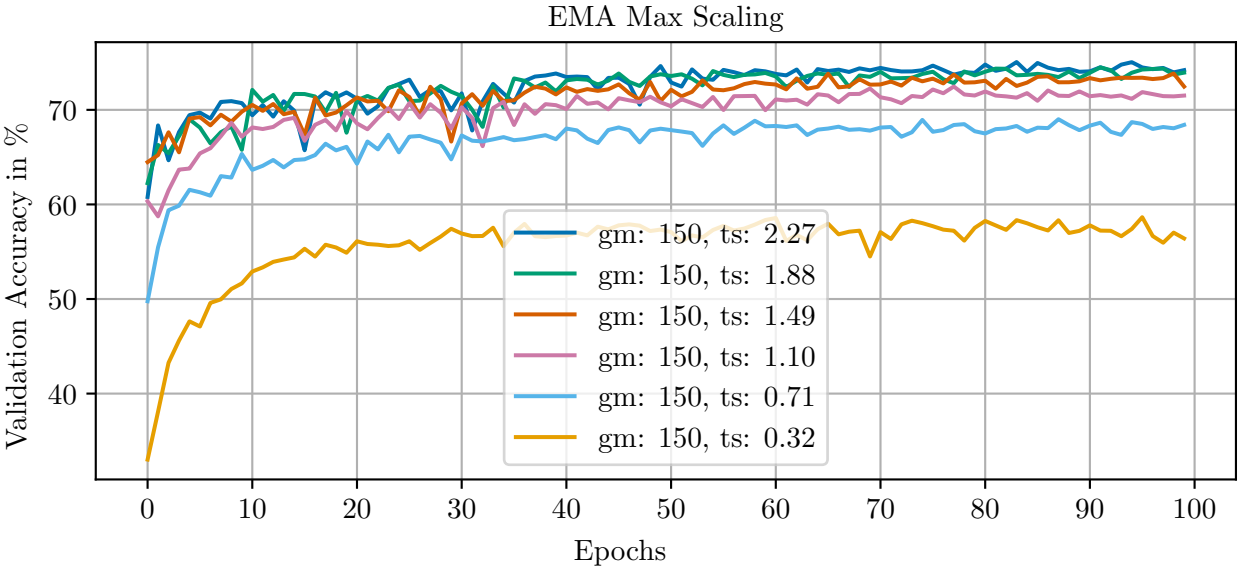

Figure C.2: Hardware training with decreased OTA gain (gm) and variable synaptic input time constant (ts).

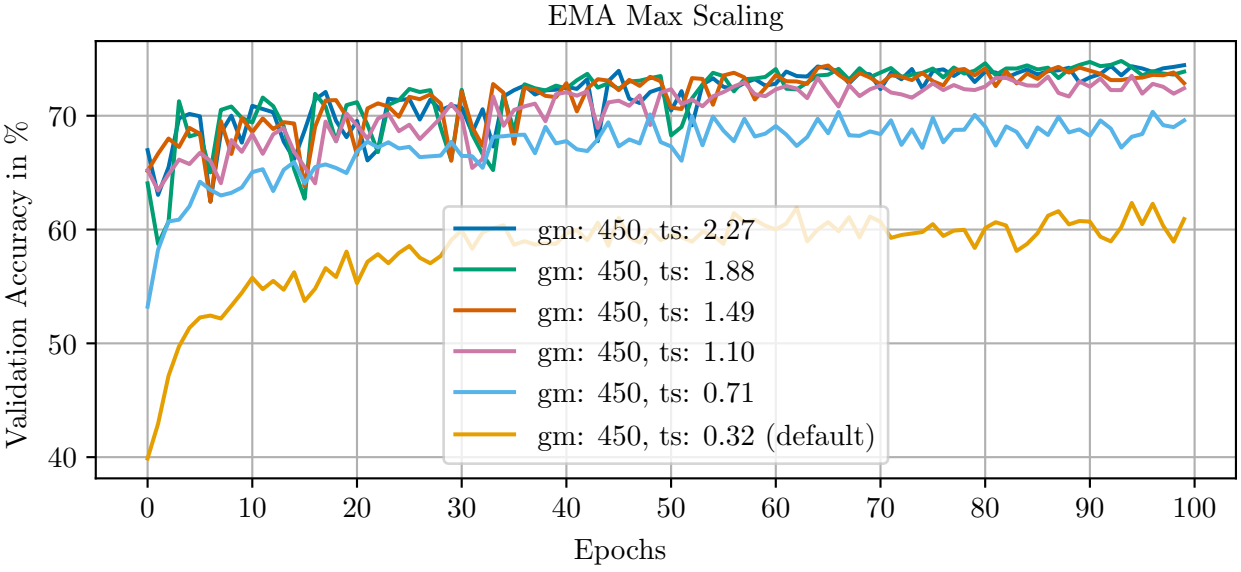

Figure C.3: Hardware training with default OTA gain (gm) and variable synaptic input time constant (ts).

It seems like the reduced OTA gain allows slightly better accuracies. This result is in line with the results of the static scaling however the difference is in this case much smaller.

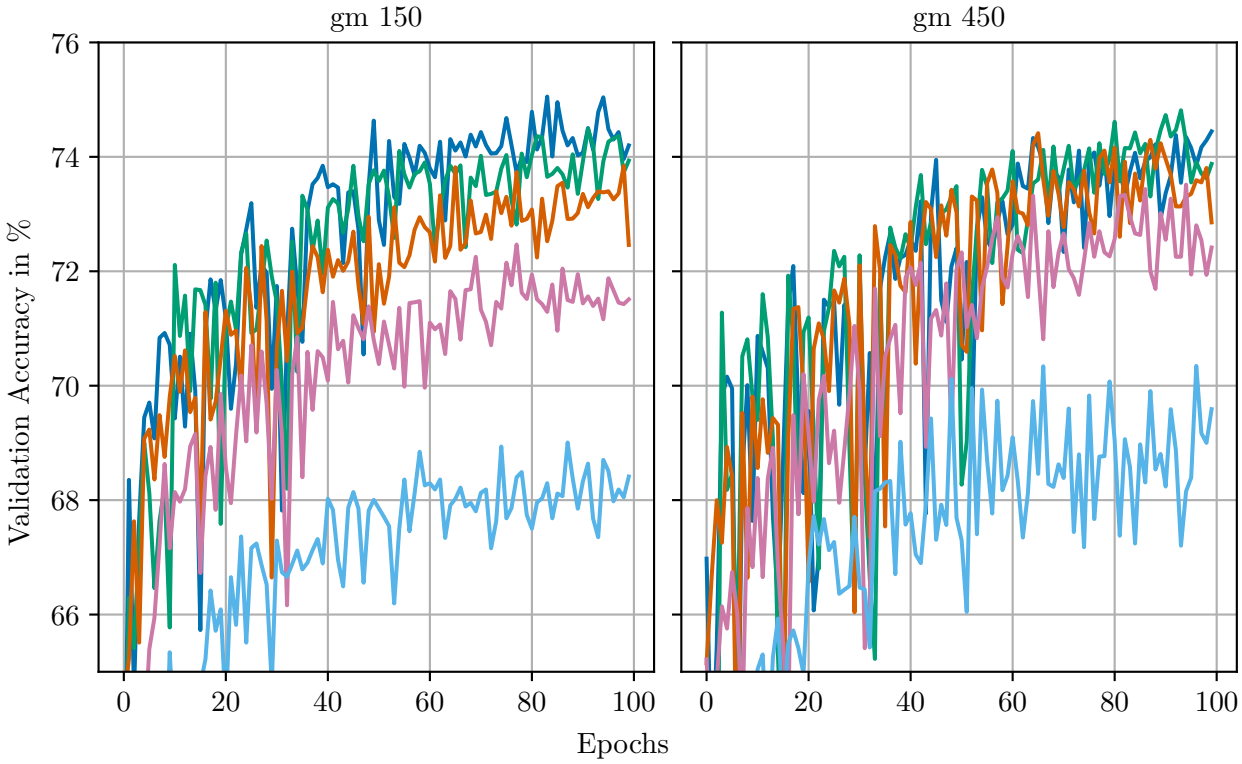

Figure C.4: Close-up comparison of the final accuracy for different OTA gains with increasing synaptic input time constant. Coloring corresponds to Figure C.2 and Figure C.3.

C.3 Training Results for EMA Max Scaling

In the following tables the exact values of the data points in Figure 4.9 and Figure 4.10 can be found. Due to the wider value range, the exact values are hard to obtain just from the plots. The tables show the parameter sweeps of `num_sends` and `wait_between_events` for EMA maximum scaling. Similar to before the mean validation accuracy is averaged over the last 10 epochs of the training and the test accuracy is averaged over 5 repetitions of the forward pass after all 100 epochs of retraining. Again all accuracies are given in percent.

`num_sends`	1	2	3	4	5	6	7
`wait_between_events`	1						
Mean Validation Acc.	63.33	69.61	71.25	72.51	**73.05**	72.59	65.88
Std. Validation Acc.	0.42	0.50	0.46	0.46	0.39	0.42	1.17
Test Acc.	61.71	67.56	70.19	**70.79**	70.12	68.67	63.06

Table C.1: Varing `num_sends` with the default calibration and EMA max scaling.

`num_sends`	1	2	3	4
`wait_between_events`	5			
Mean Validation Acc.	60.07	**61.14**	60.71	57.30
Std. Validation Acc.	0.84	1.38	1.95	1.47
Test Acc.	59.45	55.68	**61.45**	53.93

Table C.2: Varing `num_sends` with the default calibration and EMA max scaling.

`num_sends`	1	2	3	4	1	2	3	4
`wait_between_events`	5				1			
Mean Validation Acc.	74.48	**74.83**	73.18	70.33	72.05	73.81	72.17	71.22
Std Validation Acc.	0.34	0.20	0.30	0.44	0.41	0.57	0.41	0.13
Test Acc.	**72.74**	72.63	70.81	69.47	70.59	71.77	70.33	68.88

Table C.3: Varing `num_sends` with `ts`=2.27, `gm`=150 and EMA max scaling.

C.4 Training Results for Static Scaling

Since the general trends for static scaling with the default calibration are similar to EMA scaling, we omitted them in the main part of this work. For completeness, we provide them at this point.

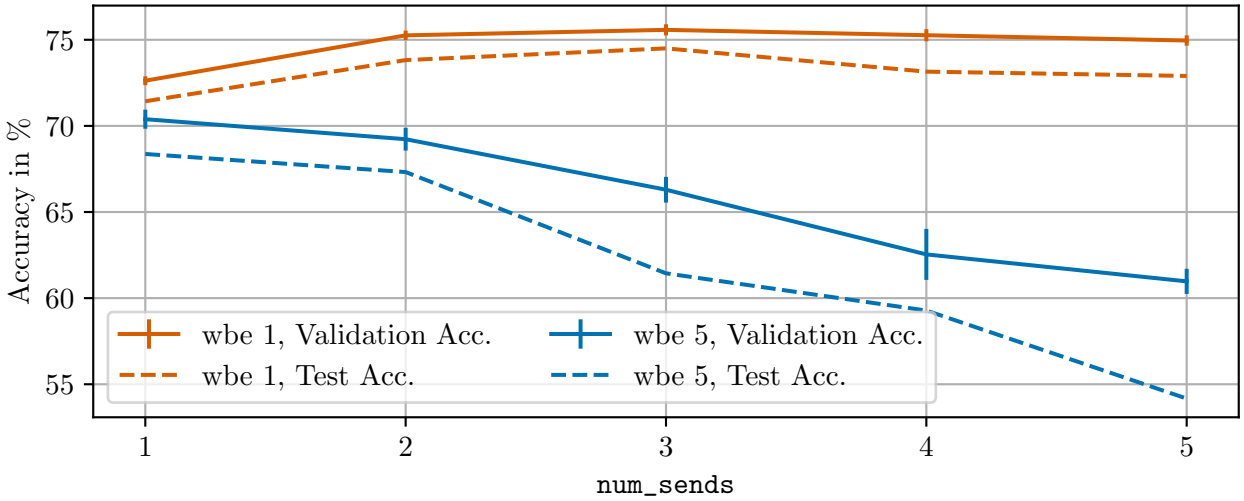

Figure C.5: Varing `num_sends` with the default calibration and static scaling.

`num_sends`	1	2	3	4	5	1	2	3	4	5
`wait_between_events`	5					1				
Mean Validation Acc.	70.39	69.23	66.29	62.54	60.97	72.62	75.26	**75.58**	75.26	74.96
Std. Validation Acc.	0.55	0.66	0.75	1.48	0.73	0.26	0.27	0.32	0.35	0.30
Test Acc.	68.37	67.32	61.44	59.28	54.17	71.43	73.82	**74.50**	73.15	72.90

Table C.4: Varing `num_sends` with the default calibration and static max scaling.

`num_sends`	1						
`wait_between_events`	1	2	3	4	5	8	15
Mean Validation Acc.	72.96	**73.30**	72.73	71.70	70.07	66.76	58.56
Std. Validation Acc.	0.35	0.46	0.42	0.52	0.52	0.85	1.29
Test Acc.	**71.84**	71.72	70.56	69.87	69.04	65.17	51.89

Table C.5: Varing `wait_between_events` with the default calibration and static scaling.

`num_sends`	1	2	3	4	5
`wait_between_events`	1				
Mean Validation Acc.	**74.42**	74.00	72.61	72.95	71.71
Std. Validation Acc.	0.36	0.41	0.37	0.40	0.47
Test Acc.	**72.94**	71.28	71.22	70.68	69.50

Table C.6: Varing `num_sends` with `ts`=2.27, `gm`=150 and static scaling.

C.5 Training Results for Activation Pruning

At this point, we provide the exact values of the data points in Figure 4.15.

Threshold	0	1	2	3	4	5	6	7
Mean Validation Acc.	**74.22**	74.07	73.91	73.84	72.56	71.61	70.45	68.96
Std. Validation Acc.	0.36	0.31	0.22	0.37	0.51	0.34	0.39	0.58
Test Acc.	72.26	**72.33**	72.21	71.92	70.07	70.26	69.22	67.10

Table C.7: EMA maximum scaling with input activation pruning.

Threshold	0	1	2	3	4	5	6	7
Mean Validation Acc.	**77.13**	77.07	76.64	76.15	75.54	74.18	73.46	72.41
Std. Validation Acc.	0.21	0.19	0.26	0.21	0.19	0.30	0.22	0.34
Test Acc.	75.45	**75.66**	75.25	74.16	73.79	72.65	71.59	70.25

Table C.8: Static scaling with input activation pruning.

`num_sends`	1				
`wait_between_events`	1	2	3	4	5
Mean Validation Acc.	75.02	76.44	76.70	**77.19**	76.96
Std. Validation Acc.	0.30	0.13	0.35	0.19	0.24
Test Acc.	73.80	74.71	74.94	**75.55**	75.20

Table C.9: Static scaling without pruning. Repetition of Table 4.4 for comparison.

Pruning can slightly increase the test accuracy of the static scaling approach. However for the validation accuracy there is a slight decrease. Considering the measured standard deviation the results appear equal and activation pruning seems to have a vanishingly small effect.

C.6 Backprop Through Scale Distributions

In the main part of this work, we only show the results without BatchNorm since the training diverges for increased synaptic input time constants in combination with BatchNorm. Most likely due to saturation effects. At this point, we also provide the distributions of the MLP with batch normalization layers included. However, there are no obvious signs that could explain the poor performance with batch normalization and an increased synaptic input time constant.

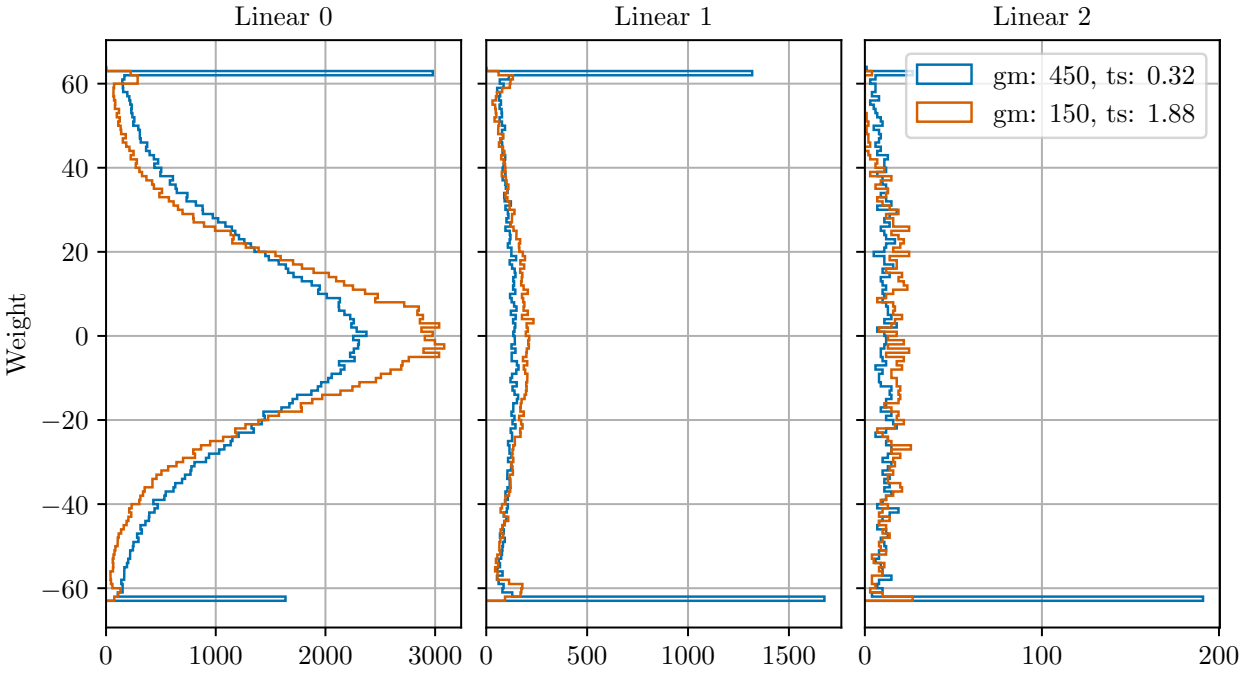

Figure C.6: Weight distribution with BatchNorm in combination with backprop through scale.

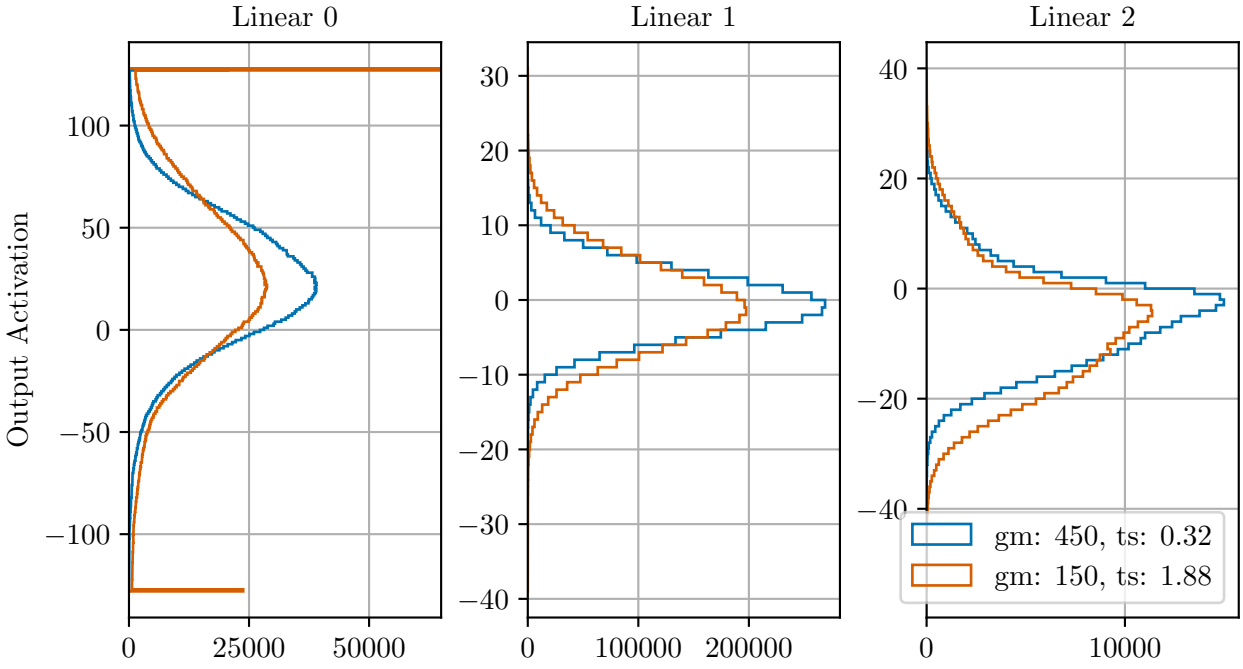

Figure C.7: Output activation distribution with BatchNorm in combination with backprop through scale.

Erklärung

Ich versichere, dass ich diese Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, 06.10.2023

Ort, Datum

E. Kern

Unterschrift