Master Thesis

Full Integer Arithmetic Online Training for Spiking Neural Networks

Ismael Gomez
i6334270

Thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science of Artificial Intelligence
at the Department of Advanced Computing Sciences
of the Maastricht University

Thesis Committee:

DACS	Guangzhi Tang
DACS	Enrique Hortal Quesada

Maastricht University
Faculty of Science and Engineering
Department of Advanced Computing Sciences

March 14, 2025

Abstract

Spiking Neural Networks (SNNs) offer substantial advantages for neuromorphic computing due to their biological plausibility and energy efficiency. However, traditional training methods such as Backpropagation Through Time (BPTT) and Real Time Recurrent Learning (RTRL) are computationally intensive and difficult to implement efficiently on neuromorphic hardware. This thesis proposes a novel integer-only, online training algorithm tailored specifically for SNNs, leveraging a mixed-precision approach and integer-only arithmetic to significantly enhance computational efficiency and reduce memory usage.

The developed learning algorithm integrates gradient-based methods inspired by Real-Time Recurrent Learning (RTRL) and spatiotemporal backpropagation, using local eligibility traces to approximate gradients. The algorithm employs integer-only arithmetic, replacing costly floating-point operations with computationally efficient bit-shift operations, making it particularly suitable for deployment on specialized neuromorphic hardware. Three error propagation strategies (Feedback, Final, and Direct) are evaluated, with the Direct approach emerging as optimal for integer-based training due to its stability and lower computational complexity. Additionally, the integer training algorithm is successfully extended beyond fully connected SNNs to Convolutional Spiking Neural Networks (CSNNs) and Recurrent Spiking Neural Networks (RSNNs), demonstrating versatility across architectures.

Extensive experiments are conducted on two widely used datasets: MNIST for static vision tasks and the Spiking Heidelberg Digits (SHD) dataset for temporal neuromorphic tasks. Results indicate that mixed-precision configurations, particularly those using 16-bit shadow weights and 8- or 12-bit inference weights, achieve comparable or superior accuracy relative to full-precision floating-point implementations, reducing memory usage by over 60% and computational energy by more than an order of magnitude. Despite limitations in multi-layer training and ultra-low precision configurations, the method achieves robust performance, matching or exceeding baseline methods like BPTT.

In conclusion, the proposed integer-only online learning algorithm presents an effective solution for efficiently training SNNs, enabling deployment on resource-constrained neuromorphic hardware without sacrificing accuracy. Future research lines include achieving stability in deeper networks, exploring more advanced neuron models and quantization techniques, and testing sparsity methods to enhance efficiency and hardware applicability.

Contents

Chapter 1

Introduction

Over the past decade, artificial intelligence (AI) has experienced a remarkable surge in both popularity and capability, transitioning from a specialized research field to the cornerstone of a new technological revolution [1, 2]. This AI revolution has been propelled largely by advances in machine learning and deep neural networks, which have demonstrated unprecedented performance on tasks ranging from image recognition and natural language processing to autonomous vehicles. The transformative impact of these methods has led to a proliferation of AI applications across diverse industries, including healthcare, finance, robotics, and entertainment. As AI systems become more prevalent and integrated into daily life, the need to explore energy-efficient, and robust solutions has grown significantly.

In response to these evolving demands, neuromorphic machine learning has gained attention as an emerging paradigm that aims to emulate key aspects of biological neural processing in order to achieve efficiency, adaptability, and resilience [1, 3]. By leveraging architectures that mirror the structure and function of the human brain, neuromorphic systems promise to operate with lower power consumption and greater computational efficiency than conventional deep learning methods. A central concept within this domain is the spiking neural network (SNN), where neurons communicate via discrete "spikes" rather than continuous activation values.

The event-driven nature of SNNs offers intriguing opportunities for parallel processing and highly efficient implementations on specialized hardware platforms [3]. Moreover, neuromorphic computing aligns naturally with the increasing demand for Edge AI and on-device learning, where AI models must operate efficiently on resource-constrained environments such as mobile devices, embedded systems, and autonomous machines. Unlike conventional deep learning models that rely on cloud-based computation, Edge AI enables real-time, local processing, reducing latency, bandwidth requirements, and energy consumption, while improving privacy and adaptability [4]. Neuromorphic processors like Intel Loihi [5] and SpiNNaker [6] exemplify this tendency, as they leverage the sparse and asynchronous nature of SNNs to perform inference and learning at significantly lower power compared to conventional von Neumann architectures. These properties make neuromorphic computing suitable for applications requiring continuous, real-time adaptation.

Despite these advantages, training spiking neural networks remains a significant challenge. Traditional backpropagation methods like Backpropagation Through Time (BPTT) and Real-Time Recurrent Learning (RTRL), while successful for classical deep neural networks, are less straightforward for SNNs due to their discrete and temporally dynamic behavior [7, 8]. As a result, many alternatives are being proposed addressing the limitations of traditional methods by reducing memory usage, computational demand, or both, demonstrating the active and ongoing nature of this research area.

Quantization has long been recognized as an effective strategy for reducing neural networks' computational complexity and memory footprint. Related to quantization, Mixed Quantization Training represents a hybrid approach in which certain elements are quantized during training while others remain at full precision. Furthermore, efforts to reduce computational overhead have led to the exploration of fully integer quantization strategies; utilizing cheap-to-compute integer operations and drastically decreasing the precision requirements of the network parameters [9, 10]

This study focuses on developing an online training method for SNNs based exclusively on integer-only arithmetic. We developed an online training algorithm for SNNs based on state-of-the-art methods and applied mixed-precision training to improve its efficiency while maintaining a competitive performance in the MNIST and SHD datasets [11, 12]. The learning algorithm is extended to Convolutional Spiking Neural Networks (CSNNs) and Recurrent Spiking Neural Networks (RSNNs). Our project aims to answer the following research questions:

- How can quantization and low-precision concepts be applied to improve the efficiency in the learning process of spiking neural networks?
- What trade-offs, if any, are introduced by the new methods in terms of training time, inference speed, and model accuracy?
- How can the learning algorithm be designed while considering realistic neuromorphic hardware processing for real hardware efficiency?

Chapter 2

Related Work

2.1 Spiking Neural Networks

Spiking neural networks (SNNs) are a different class of artificial neural networks that mimic how the biological brain processes information more realistically. The concept of SNNs emerged in the late 1990s and is often referred to as the third generation of neural network models [13]. These networks were inspired by the observation of time-dependent, discrete events (spikes) in biological neurons, which are critical for neural communication in the brain [14].

Unlike traditional ANNs, which use neurons that output a continuous value (activation functions), SNNs utilize a timing-based mechanism where neurons communicate by sending spikes — discrete events in time. This spike-based communication also allows the networks to be structured in layers like traditional ANNs. SNNs are considered an alternative to ANNs because they can potentially handle temporal information more naturally and efficiently. They are capable of processing spatiotemporal patterns in data, making them suitable for tasks involving dynamic and real-time inputs [14, 13].

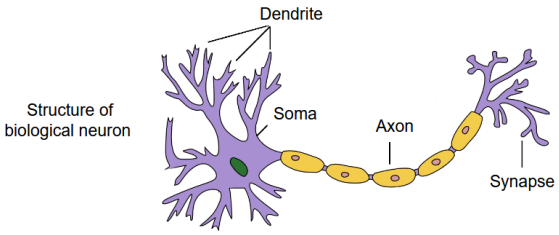

Figure 2.1: Diagram of a biological neuron [14]

SNNs utilize biologically inspired models to encode information in the timing and sequence of spikes, managed by complex neuron models like the leaky integrate-and-fire. SNNs incorporate dynamic synaptic connections, which adjust through learning mechanisms, enabling them to process temporal patterns in data effectively. SNNs offer superior energy efficiency and natural handling of temporal data compared to traditional ANNs, making them ideal for real-time processing tasks like event-based problems [14].

Many different neuron models have been proposed with different numbers of parameters and degrees of complexity. From the complex Hodgkin-Huxley model to the simple Integrate-and-

Fire model, the different neuron implementations exhibit a range of trade-offs between biological realism and computational cost.

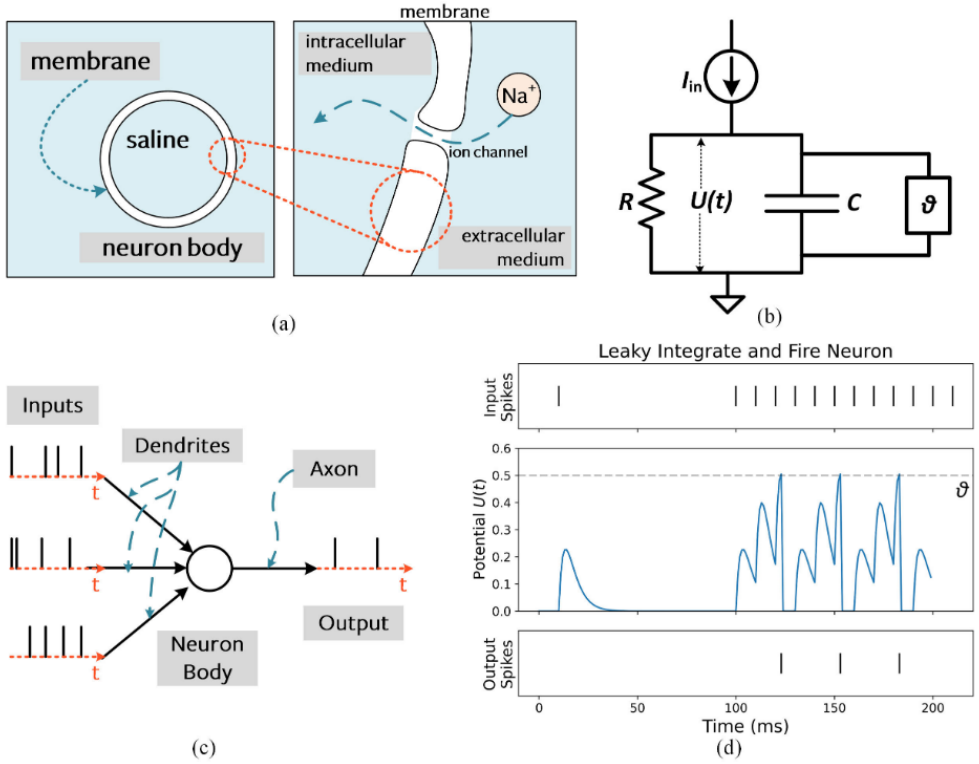

Figure 2.2: Leaky IF neuron model. (a) Insulating bilipid membrane separates the intracellular and extracellular medium. Gated ion channels allow charge carriers, such as Na+, to diffuse through the membrane. (b) Capacitive membrane and resistive ion channels form an RC circuit. When the membrane potential exceeds a threshold θ, a spike is generated. (c) Input spikes generated by Iin are passed to the neuron body via the dendritic tree. Sufficient excitation will cause spike emission at the output. (d) Simulation depicting the membrane potential U(t) reaching the threshold, arbitrarily set to $\theta = 0.5$ V, which generates output spikes [15].

The neuron model used in this work is the Leaky Integrate-and-fire (LIF) (Figure 2.2). In this model, the input spike $\mathbf{s}$ is integrated into the membrane voltage $\mathbf{V}$ multiplied with the layer weights $\mathbf{W}$. The result is added to the voltage from the previous time step multiplied by the voltage decay term β (Equation 2.1). Once the voltage is updated, the neuron generates a spike if the voltage is higher than the threshold V_{th} (Equation 2.2). If the neuron is activated and a spike $\mathbf{s}$ is generated, the voltage is reset to zero (hard reset mechanism).

$$\mathbf{V}^{(i)}(t) = \beta \cdot \mathbf{V}_{\mathrm{s}}^{(i)}(t-1) + \mathbf{W}^{(i)}\mathbf{s}^{(i-1)}(t) \tag{2.1}$$

$$s^{(i)}(t) = \begin{cases} 1, & \text{if } V^{(i)}(t) > V_{th} \\ 0, & \text{otherwise.} \end{cases} \tag{2.2}$$

The LIF model offers a balance between computational simplicity and biological realism,

which is why it is the most common model used in the neuromorphic machine learning field. As illustrated in Figure 2.2, the LIF neuron keeps the information from the previous activations in the voltage. LIF neurons can be configured in layers and build complex models following structures similar to traditional ANNs.

2.2 Quantization and Low-precision Learning

Quantization has long been recognized as an effective strategy for reducing neural networks' computational complexity and memory footprint. There are different approaches to applying this technique; each has a different trade-off between memory usage, computational cost, and model performance. The least invasive method in terms of performance influence is Post-Training Quantization (PTQ), where the model is trained on full precision and is later quantized to reduce the cost of inference [16]. Quantization-Aware Training (QAT) imposes constraints during the training process, simulating quantization processes to enhance the performance once the model is quantized for inference. While these methods enable efficient inference by discretizing weights and activations, they inherently rely on full precision computations during training [17]. Mixed-Precision Training represents a hybrid approach in which certain elements are quantized during training while others remain at full precision. This strategy balances the computational efficiency of low-precision arithmetic with the performance benefits of full-precision processing, thereby reducing training costs without significantly compromising accuracy [18].

2.3 Integer-based Computing

2.3.1 Fixed-point operations and integer-only arithmetic

Most computers use floating-point formats for numeric representations. This representations follow the IEEE 754 standard consisting of 3 components; the sign bit, the exponent, and the mantissa. The floating-point can represent a wide dynamic range of values, but offers higher precision in smaller numbers. On the other hand, fixed-point format represents the numbers assigning a fixed number of bits for the integer and fractional parts. The precision of the values is constant but the range is limited.

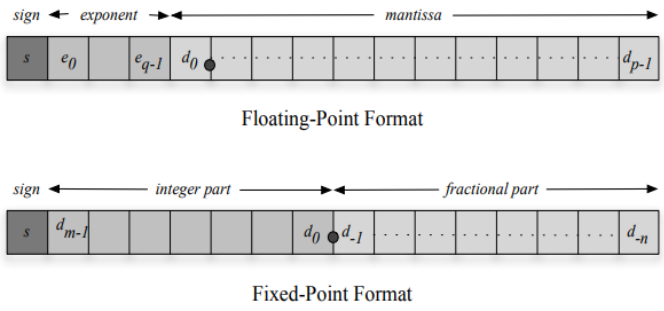

Figure 2.3: Difference between the floating-point and fixed-point format [19].

Arithmetic operations with these formats follow different processes. Floating-point operations involve complex computations like exponent manipulations, rounding, and normalization, whether fixed-point utilizes efficient integer arithmetic operations. Integer arithmetic restricts the arithmetic operations to integer-only types (e.g., addition, multiplication, and bit-shifting).

The predefined decimal position facilitates the operations and eliminates rounding errors, resulting in much more efficient processes [20].

2.3.2 Integer-based learning

Integer-based learning restricts operations to integer-only arithmetic, offering significant advantages for edge deployment by eliminating floating-point overhead[10]. By minimizing floating-point computations, these integer-based methods can potentially lead to faster training times and lower power consumption, thus unlocking the full potential of SNNs in real-world applications and resource-constrained environments.

Nonetheless, utilizing it to train SNNs presents unique challenges. First, integer arithmetic has a fixed range compared to dynamic range in floating formats, this limitation can cause loss of precision in gradient computations and weight updates for values with smaller magnitudes[21]. Second, overflow and underflow can be easily reached with integer arithmetic in deep learning models where the weight updates can grow exponentially layer by layer [22]. Previous works indicate the difficulty of full integer ANN training, remarking on the necessity of carefully adjusting scaling and clipping [23]. In SNNs, the temporal nature of spike computation amplifies quantization errors and increases the risk of underflow and overflow.

2.4 Training Spiking Neural Networks

2.4.1 Challenges in training SNNs

Despite these advantages, training SNNs introduces unique challenges, primarily due to the intrinsic properties of their neuron models and learning mechanisms. Unlike traditional ANNs that use continuous activation functions, SNNs operate with discrete spikes which are non-differentiable and thus not amenable to standard gradient-based optimization techniques. This necessitates the development of alternative training algorithms or using surrogate gradients to enable effective learning [24, 25]. Furthermore, the biologically inspired dynamics of SNNs, including their handling of time and sparsity of spikes, complicate the training process as these models require the integration of temporal dynamics within learning rules.

Traditional learning algorithms such as BPTT and RTRL are well-established in the training of ANNs, but present significant limitations when adapted to SNNs [7, 8]. BPTT, commonly used due to its relatively low computational complexity, faces challenges with SNNs primarily due to its high memory demands. This method requires the storage of all previous neuron activations to compute gradients, which becomes impractical for long input sequences. Although truncated versions of BPTT exist, they compromise the model's ability to retain information over long sequences, as truncation inherently results in the loss of historical data crucial for learning dependencies [26].

On the other hand, RTRL offers a promising alternative for SNNs due to its online learning capability, where gradients are computed in real-time without the need to unfold the entire sequence. This method is more memory-efficient than BPTT, as it does not require storing past activations and can capture dependencies across full-length sequences. However, the main drawback of RTRL lies in its high computational cost, as it necessitates continuous updates of a large number of partial derivatives at each timestep [27].

2.4.2 Alternative training methods in SNNs

In the field of training SNNs following the neuromorphic computational principles of event-based, local, and online computations, various learning strategies have been explored to overcome the limitations of traditional methods such as BPTT and RTRL. The high memory demands and computational inefficiencies of these algorithms have driven the search for more biologically plausible and efficient alternatives.

Several approaches have been proposed in the research field of efficient learning algorithms: Xiao et al. [28] developed an Online Training Through Time method that modifies BPTT to enable forward-in-time learning while managing constant training memory costs; Perez-Nieves and Goodman [29] explored the sparsity in SNNs and developed a sparse spiking gradient descent algorithm that leverages the inherent spatiotemporal sparsity of SNNs for enhanced speed and memory efficiency; Bellec et al. [30] proposed an online learning method based on eligibility traces.

Among these advancements, Tang. et. al. presented BioGrad [31], a biologically plausible local gradient-based online learning algorithm functionally equivalent to backpropagation. This algorithm utilizes a multi-compartment neuron model, which enables simultaneous integration of feedforward and feedback information, which is crucial for maintaining the biological plausibility of the learning process. Furthermore, including a "sleep" phase, where feedback weights are periodically adjusted through a local Hebbian rule, enhances the learning accuracy and approximates the traditional backpropagation more closely. These features make BioGrad particularly suitable for energy-efficient, on-device neuromorphic learning.

Parallel to advances in learning algorithms, research in neural network quantization has explored methods to balance computational efficiency, precision, and performance. Blake et al. [32] propose "unit scaling," which normalizes variance across weights, activations, and gradients at initialization, simplifying the training of LLM models like BERT in low-precision formats without dynamic rescaling. Different mixed-precision configurations are tested (FP31, FP16, FP8) and the authors claim that their method is versatile in models and tasks, but only LLM models were used in the research. Meanwhile, Schnöll et al. [17] introduced a dynamic rescaling QAT technique utilizing a power of 2 rescaler to simplify hardware implementation. They achieved significant improvements in training time and memory usage, despite the potential cost of quantization flexibility by only using powers of 2.

In the domain of quantization applied to SNNs, Acharya et al. [33] demonstrated the application of Quantization Aware Training (QAT), using the DECOLLE learning algorithm [34] to effectively reduce weight precision to 3-bits during inference without significant loss in accuracy. Complementarily, Shymyrbay et al. [35] utilized a differentiable quantization function based on sigmoid functions for SNNs, which helps reduce the gradient mismatch common in traditional methods. The paper by Hassan et al. [36] on "SpQuant-SNN" employs ultra-low precision QAT (1-3bits) and dynamic pruning to enhance efficiency on resource-constrained devices. The study substantially reduces memory usage and computational overhead, with less than 1.8% accuracy degradation for complex tasks like image classification and object detection. These studies are some of the most relevant works in the quantization field for ANNs and SNNs. The great variety in the quantization and mixed-precision strategies shows that the field, especially for SSNs, is under current exploration [37, 38, 39, 18].

On the topic of utilizing integers, Jacob et. al. [10] focus on quantization for efficient integer-arithmetic-only inference, particularly on mobile devices. It improves latency vs. accuracy trade-offs for models like MobileNets running on mobile CPUs. The training involves simulated quantization using floating-point arithmetic, then transitioning to integer-only arithmetic for inference. This method reduces model size and increases the speed of on-device inference. How-

ever, the paper does use integer-only arithmetic for the training, which could have a significant impact. Ghafari et. al. [23] extend the concept of integer computations to the entire training pipeline, including forward pass, back-propagation, and stochastic gradient descent. Employing a fixed-point mapping coupled with a non-linear inverse mapping, the scales are dynamically computed and don't need to be adjusted during the training. Nevertheless, the paper does not rely exclusively on integer-only arithmetic since it uses a custom numeric format to emulate the floating-point system.

PocketNN [21] introduces the first true integer-only training and inference framework using direct feedback alignment and specially designed pocket activations (integer-only activation functions). It avoids the complexity of traditional backpropagation and quantization methods and is particularly suited for low-end devices lacking floating-point units. The direct feedback alignment approach enables each layer to be trained independently without the cumulative error propagation associated with traditional backpropagation. This effectively prevents integer overflow—a key obstacle in integer-only implementations. The method is developed for Multi-Layer DNNs and is tested on MNIST and Fashion-MNIST. Following this work, NITRO-D [40] presents a novel framework for native integer-only training of deep convolutional neural networks. NITRO-D uses integer local-loss blocks to process and generate integer activations for both onward transmission and internal training within the same block. These blocks employ a custom scaling layer to adjust pre-activations to a suitable range for the NITRO-ReLU activation function, which is specially designed to handle integer values and includes mean centering to replace batch normalization. The training leverages IntegerSGD, an adaptation of stochastic gradient descent for integers with a weight decay term. This architecture is tested on standard datasets like MNIST and CIFAR10, showing its potential despite some performance trade-offs compared to traditional floating-point methods.

The topic of integer-only techniques for SNNs is also currently being explored. The work by Zou et. al. [41] presents an integer-based SNN that utilizes integer quantization in the ANN2SNN conversion. All network parameters (including membrane potentials, synaptic weights, and firing thresholds) are quantized to integers, and a new dynamic threshold adaptation technique is presented to eliminate synchronization problems in ANN2SNN conversion. Luo et al. [42] explore the application of integer-valued training in SNNs in object detection tasks. Their novel SpikeYOLO architecture simplifies YOLO by incorporating SNN blocks to handle the spiking nature of neurons. It proposes the Integer Leaky Integrate-and-Fire (I-LIF) neuron to mitigate quantization errors during the training phase. This neuron type allows integer values during training which are then converted to spike-driven processes during inference. Their approach showcases considerable improvements in performance metrics on datasets like COCO and Gen1, demonstrating how integer-valued training can effectively reduce quantization errors while maintaining the energy efficiency of SNNs. Despite the interesting integer approach of this paper, they use integer values but not integer-only arithmetic.

Given the landscape of biologically plausible and computationally efficient learning methods for SNNs, BioGrad emerges as an optimal foundation for this work. BioGrad's online, local and gradient-based learning address the high memory demands and the real-time constraints of BPTT and computational inefficiencies by RTRL. This approach aligns with the neuromorphic principles and optimizes energy efficiency, making it highly relevant for on-device learning in neuromorphic systems.

Chapter 3

Methodology

3.1 Learning Algorithm

The algorithm developed in this work is gradient-based, local, online, and suitable for integer-only arithmetic. It uses local eligibility traces to approximate the gradients required for learning in spiking neural networks and update the weights. The learning rule used in this algorithm is derived from the principles of spatiotemporal backpropagation and Real-Time Recurrent Learning (RTRL). The mathematical derivation can be found in the Appendix A.

3.1.1 Spiking Neural Network

The method was initially developed for fully connected SSNs with the LIF neuron model.

Forward Step

The voltage decay operation in the voltage update (Equation 2.1) is converted to a right bit-shift operation $\gg$ to increase efficiency and maintain the integer-only suitability (Equation 3.2).

$$\hat{\beta} = \lfloor \log_2(1/\beta) \rfloor \tag{3.1}$$

$$\mathbf{V}^{(i)}(t) = \mathbf{V}_{\mathrm{s}}^{(i)}(t-1) \gg \hat{\beta} + \mathbf{W}^{(i)}\mathbf{s}^{(i-1)}(t) \tag{3.2}$$

After the layer voltage is updated, a binary pseudo gradient is calculated by assigning a positive value to those voltages whose values are close to the firing threshold (Equation 3.3).

$$\widetilde{\nabla V}^{(i)} = |V^{(i)} - V_{th}^{(i)}| < Grad_{win}^{(i)} \tag{3.3}$$

This method utilizes two traces to capture information about how the weights influence the neuron's output: A presynaptic spike trace, which reflects the history of presynaptic neuron spiking (Equation 3.4), and a correlation trace, which captures the correlation between pre- and postsynaptic neuron activities (Equation 3.5).

$$T_{pre}^{(i)} = T_{pre}^{(i)} \gg \hat{\beta} + \mathbf{s}^{(i-1)}(t) \tag{3.4}$$

$$T_{corr}^{(i)} = T_{corr}^{(i)} + T_{pre}^{(i)} \cdot \widetilde{\nabla V}^{(i)} \tag{3.5}$$

Loss Calculation

During the inference, the output spikes from each time step are aggregated, and the prediction of the network is computed by choosing the neuron with the maximum number of spikes.

The loss function simplifies the softmax function to work with integer-only arithmetic by removing the exponential operations and introducing a bit-shifting. It is a normalization of the output spikes regulated with the loss precision parameter α. To increase the efficiency of the function in integer-only arithmetic, instead of dividing by the number of time steps, t_s is approximated to the closest power of 2 and then a bit-shifting operation is made. After the bit-shifting, the prediction error is calculated by subtracting the one-hot label y multiplied by the loss precision:

$$error^{pred} = (prediction \cdot \alpha) \gg \lfloor \log_2(t_s) \rfloor - y \cdot \alpha \tag{3.6}$$

One of the characteristics of the softmax function is that, due to the exponential operations, the loss value is positive for all the values, thus it helps to change the behavior of all the neurons. Using the simplified loss function results in very sparse feedback for the neurons and only a small fraction of the network is trained. For this reason, two variations of the loss function were implemented by adding positive values or random values to the zero loss values (Equations 3.7 and 3.8).

$$error^{pred} = \begin{cases} error^{pred}, & prediction \neq 0 \\ 1, & prediction = 0 \end{cases} \tag{3.7}$$

$$error^{pred} = \begin{cases} error^{pred}, & prediction \neq 0 \\ \mathcal{R}(-1,1), & prediction = 0 \end{cases} \tag{3.8}$$

Error Propagation Approaches

Three different approaches for the error propagation were considered and evaluated:

Feedback In the feedback approach, error calculation occurs at every time step following an initial soft start phase. The error signal is processed through two dedicated error neurons in the output layer: one for positive errors and another for negative errors. These neurons generate spike-based representations of the error $s^{(pos)}$ and $s^{(neg)}$, which are then propagated back to earlier layers. Each layer integrates the spikes through their feedback weights into an error voltage $v_{fb}^{(i)}$, influencing synaptic updates over time. The layer feedback weights W_{fb} are the transposed weights of the next layer in the inference computation order. This method allows dynamic adaptation and follows biological plausibility, but it is computationally expensive and prone to instability, especially when working with integer numbers.

$$v_{error}^{corr}(t) = v_{error}^{corr}(t-1) - s^{(pos)}(t-1) + error^{pred(pos)}(t) - error^{pred(neg)}(t) \tag{3.9}$$

$$v_{error}^{neg}(t) = v_{error}^{neg}(t-1) - s^{(neg)}(t-1) + error^{pred(pos)}(t) - error^{pred(neg)}(t) \tag{3.10}$$

$$s^{(pos)}(t) = v_{error}^{corr}(t) > V_{th}^{fb}(t) \tag{3.11}$$

$$s^{(neg)}(t) = v_{error}^{neg}(t) > V_{th}^{fb}(t) \tag{3.12}$$

$$v_{fb}^{(ouputt)}(t) = v_{fb}^{(output)}(t-1) + s^{(pos)}(t) - s^{(neg)}(t) \tag{3.13}$$

$$v_{fb}^{(i)}(t) = v_{fb}^{(i)}(t-1) + W_{fb}^{(i)} \times s^{(pos)}(t) - W_{fb}^{(i)} \times s^{(neg)}(t) \tag{3.14}$$

Final The final approach defers error computation until the last time step, at which point the error is processed following the same steps as in the feedback approach. This method significantly reduces computational costs and avoids instability caused by recurrent error propagation, making training more stable.

Direct In the direct approach, the error is also computed at the last time step, but unlike the previous methods, it bypasses error neurons. In the output layer, the error is multiplied with the trace of neural activity, while in the hidden layers, feedback weights process the error without additional spike-based transformations. This method simplifies the model, reducing memory usage and computational demands while maintaining efficient error propagation. However, removing explicit error neurons may reduce the expressiveness of the feedback signal.

$$v_{fb}^{(output)} = error^{pred} \tag{3.15}$$

$$v_{fb}^{(i)} = W_{fb}^{(i)} \times error^{pred} \tag{3.16}$$

Weights Update

Once the prediction error is computed, the network parameters are updated. The layer feedback voltage is multiplied with the weight trace, indicating which neurons had more relevance in the prediction to change their value. Finally, the result is summed along the batch dimension (Equation 3.17).

$$\Delta^{(i)} = \sum_{b=1}^{B} [v_{fb}^{(i)} \cdot T_{corr}^{(i)}] \tag{3.17}$$

The weights are updated by scaling the delta value with the learning rate η and incorporating the weight decay term regulated by the weight decay rate ρ (Equation 3.18).

$$W^{(i)} = W^{(i)} - \Delta^{(i)} \gg \hat{\eta^{(i)}} - W^{(i)} \gg \hat{\rho} \tag{3.18}$$

3.1.2 Convolutional Spiking Neural Networks

Adapting the algorithm to train Convolutional Spiking Neural Networks involved changes in how the weight traces are calculated and how the error is processed in the hidden layers.

The pre-synaptic trace computation is not altered in the CSNN case. However, the weight trace needs to be adapted since the shapes of the output and input are different. As shown in Figure 3.1, the weight trace is computed by taking, for each output value, the kernel values used in each convolution. Those patches are then multiplied with the pseudo-gradient (Equations 3.19 and 3.20).

$$patches^{(i)} = \text{unfold}(T_{pre}^{(i)}) \tag{3.19}$$

$$T_{corr}^{(i)} = T_{corr}^{(i)} + patches^{(i)} \cdot \widetilde{\nabla V}^{(i)} \tag{3.20}$$

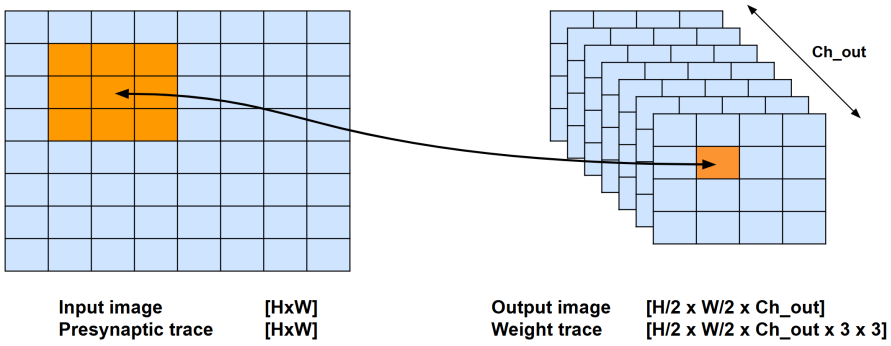

Figure 3.1: Diagram of how the weight trace is calculated in a CSNN.

The other major change in the CSNN implementation is in the weight update step. The feedback weights used in the SNN case are substituted by a transposed convolutional operation that transforms the prediction error (B $\times$ $Output_{dim}$) to be multiplied properly with the weight trace (B $\times$ Ch_{out} $\times$ H/2 $\times$ W/2 $\times$ Ch_{in} $\times$ kernel_size $\times$ kernel_size) (Equation 3.21).

$$v_{fb}^{(i)} = Conv^{T(i)}(error^{pred}) \tag{3.21}$$

The feedback voltage is then multiplied with the weight trace and summed over the height, weight, and batch dimensions (Equation 3.22):

$$\Delta^{(i)} = \sum_{i=1}^{B}\sum_{j=1}^{H}\sum_{k=1}^{W}[v_{fb}^{(i)} \cdot T_{corr}^{(i)}] \tag{3.22}$$

3.1.3 Recurrent Spiking Neural Networks

Implementing the recurrent connection in the spiking neural network implies a change in how the voltage is updated. Each layer has an extra set of weights that processes the voltage from the previous time step. To maintain numeric stability and prevent the voltage of uncontrolled growing, a bit-shifting operation is made to the voltage before multiplying it with the recurrent weights: only the first 16 bits of information are taken from the voltage (Equation 3.23).

$$\mathbf{V}^{(i)}(t) = \beta \cdot \mathbf{V}_{\mathrm{s}}^{(i)}(t-1) + \mathbf{W}^{(i)}\mathbf{s}^{(i-1)}(t) + \mathbf{W}_{rec}^{(i)}\mathcal{B}_{16}\big(\mathbf{V}^{(i)}(t-1)\big) \tag{3.23}$$

Instead of calculating specific traces and updating the recurrent weights, a random recurrent feedback approach has been followed in this work. Random recurrent mechanisms replace the need for training recurrent weights by using fixed, random matrices to process the previous voltage. The effectiveness of random feedback techniques has been demonstrated in different tasks inside the training of deep neural networks. Lillicrap et. al [43] showed that random feedback weights can effectively guide learning. Gradient-free methods based on random perturbations, as discussed by Fernandez et al. [44], provide an efficient alternative for training RNNs. These techniques have been integrated into state-of-the-art systems because they offer a simpler, more biologically realistic, and hardware-friendly method for training time-dependent network architectures.

3.2 Online Mixed-Precision Integer Training

Our full integer training method employs two weight representations:

- Shadow weights (16-bit integers): Store high-precision gradients during training.
- Low-precision weights (8-bit integers): Used for forward and backward passes.

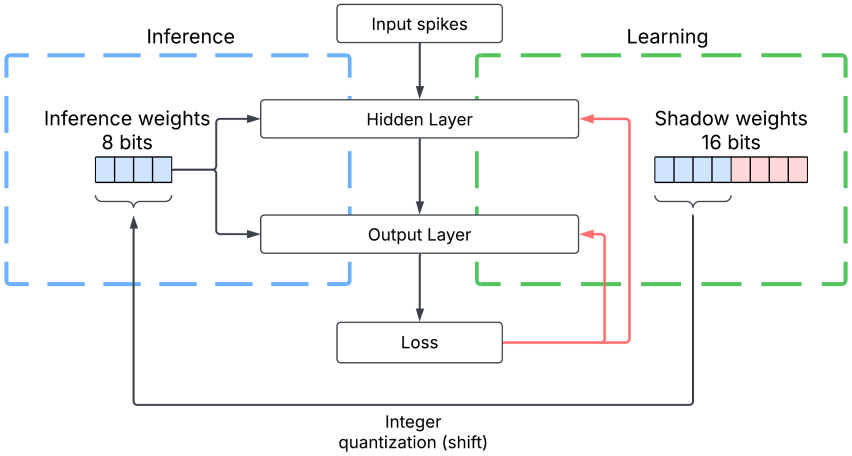

Figure 3.2: Overview of the Mix-Precision Integer Training method.

In each training iteration, the forward step and delta parameters are computed using the low-precision weights. The computed gradients are used to update the shadow weights, where the high precision allows for preserving update fidelity. Before starting a new training iteration, the low-precision weights are updated by applying bit-shifting to the shadow weights (Figure 3.2). This **integer-mixed-precision approach (MP)** balances precision and efficiency, enabling integer-only arithmetic while mitigating quantization noise.

$$W^{(i)}_{inference} = \mathcal{B}_8\left(W^{(i)}_{shadow}\right) \tag{3.24}$$

The other model used in this work is the **full-precision floating-point model (FP32)**, where 32-bit precision weights were used in every step.

3.2.1 Initialization of the weights

The weight initialization follows a structured approach to ensure numerical stability and consistency across layers. Initially, weights are assigned using Kaiming uniform initialization in floating-point precision, which is well-suited for deep networks as it maintains variance across layers and prevents gradient vanishing or explosion [45]. The maximum absolute weight value across all hidden and output layers is identified to preserve the relative scale of weights after quantization. Each weight matrix is then quantized proportionally to this global maximum. The quantization process follows a uniform scheme, where weights are scaled by a step size determined by the maximum absolute weight and the bit-width constraint (Equations 3.25 and 3.26).

$$\text{step} = \frac{2 \cdot \text{global\_max}}{2^{\text{num\_bits}} - 2} \tag{3.25}$$

$$W_q = \text{round}\left(\frac{W}{\text{step}}\right) \tag{3.26}$$

3.2.2 Gradient clipping

Training stability is one of the main challenges in integer-only training [40, 46] and gradient clipping has been widely used in floating-point and integer deep learning to obtain a smoother optimization [47, 46]. Another approach commonly used in mixed-precision training is gradient scaling [48]. The magnitude of the gradients is increased to prevent underflow in low-precision representations.

In integer-only training, the magnitude of the gradients can be large in comparison with the magnitude of the weights because, to avoid loss of information, mean operations are not included (Equations 3.17, and 3.22). Consequently, gradient scaling techniques are not needed to augment the gradient's magnitude. The gradient distributions across layers show that most of the gradients are within an acceptable range and only in a few cases the gradients become large enough to endanger training stability. In this work, we implemented gradient clipping to the delta parameters to prevent common overflow in integer training and to stabilize the training. A global clipping parameter Δ_{max} was used for all the layers.

$$\Delta = \begin{cases} -\Delta_{\max}, & \text{if } \Delta < -\Delta_{\max} \\ \Delta, & \text{if } -\Delta_{\max} \leq \Delta \leq \Delta_{\max} \\ \Delta_{\max}, & \text{if } \Delta > \Delta_{\max} \end{cases} \tag{3.27}$$

3.2.3 Optimizing the model

The optimization of SNNs trained with low-precision integer-arithmetic presents several challenges, such as guaranteeing convergence and numerical stability. This work employed several optimization techniques to enhance the training process, focusing on learning rate strategies, weight regularization, and hyperparameter search.

Layer-Specific Learning Rate

To improve stability and performance, we implemented a layer-wise learning rate strategy. This approach is particularly beneficial for handling the variance in the magnitude of the delta value. Factors like the initialization of the weights, the model's architecture, and the error propagation approach can cause the delta parameter to have very different magnitudes across the hidden and the output layers. This is a standard practice in the deep learning field. The intuition behind it is that the layers closer to the network's input learn more general patterns such as edges or basic shapes, and therefore a lower learning rate is recommended to prevent excessive weight updates. In comparison, layers closer to the network's output, whose purpose is to extract more complex features, will need more adaptation and therefore a larger learning rate [49].

Weight Decay Regularization

Weight decay, commonly used as a regularization technique, was applied to mitigate overfitting, enhance generalization and prevent weights overgrowth, a common problem in integer-only train-

ing. The regularization term added to the loss function is regulated by the weight decay rate ρ (Equation 3.18).

Hyperparameter Grid Search

To systematically explore the effect of key hyperparameters, we performed exhaustive grid searches over multiple configurations, including:

- Quantization parameters: Weight initialization bits
- Neuron parameters: Voltage decay, voltage threshold and gradient window.
- Loss: Loss function variations and loss precision.
- Training parameters: Train batch size, test batch size, gradient clip, weight decay rate.
- Learning rate per layer.

3.2.4 Multilayer model

In this study, all spiking neural network models were designed with a single hidden layer to ensure numerical stability and facilitate the precise implementation of integer-only arithmetic. Training deeper networks with low-precision representations introduces significant challenges, particularly in gradient propagation and weight updates, where quantization errors accumulate, leading to vanishing or exploding gradients. Such numerical instability would require extensive fine-tuning of scaling factors and dynamic range adjustments, complicating the core objective of this research. Furthermore, the primary goal of this work is not to optimize network depth for maximum performance but to validate the proposed mixed-precision training algorithm. By focusing on a single hidden layer, we minimize architectural complexity while isolating and rigorously analyzing the effects of low-precision training on spiking neural dynamics. This approach ensures that the results primarily reflect the effectiveness of the novel algorithm.

3.3 Efficiency of the Algorithm

The main objective of mixed-precision techniques is to achieve similar performance to full-precision approaches with improvements in memory usage and computational cost. This section calculates and compares the memory and computational costs of the mixed-precision and full-precision approaches.

3.3.1 Memory usage

The memory load of an algorithm can be divided into static and dynamic. Static memory refers to memory fixed at compile-time, it is allocated once at the beginning of the program's execution. It has faster access and is not modified during the execution. Dynamic memory is allocated and released depending on the program's needs. Dynamic elements are not needed in every stage of the process and do not have to be saved at the end of the program's execution.

Applied to this algorithm, the static memory consists of the network weights and the layer and training parameters. Although the weights are updated at the end of every training iteration, they are constant during the inference and update steps. Dynamic memory comprehends the elements that are updated during the training iteration and their final value is not needed for the next iteration.

Assumption on Hardware Support for Integer Precision

In this work, we assume the existence of specialized integer-based hardware capable of handling arbitrary precision formats that are not restricted to powers of two. This assumption allows flexibility for the precision of certain components (e.g., 9, 14, 17 bits) instead of being constrained to standard 8, 16, or 32-bit formats. Such hardware architectures are motivated by recent advancements in low-power neuromorphic computing [5].

Memory Usage Calculation

Table 3.1 shows the static and dynamic memory allocation of the learning algorithm, comparing the **integer-mixed-precision (MP 16-8) approach** (16 bits for the shadow weights and 8 bits for inference weights) with the **full precision floating point (FP32) approach**. The complete list of the static and dynamic elements is shown in the Appendix B.1 in Tables B.1 and B.2. The calculation is made for a training iteration of the biggest model trained on this work: the RSNN model and the SHD dataset. The MNIST dataset has binary inputs and thus, certain elements like traces and voltages require less precision. We used the following training parameters:

- I - Input size: 175
- H - Hidden neurons: 256
- B - Batch size: 128
- O - Output size: 20

Model Section	**MP 16-8**	**FP32**
Static memory		
Hidden Layer Forward Step	311,812	831,494
Output Layer Forward Step	15,364	40,968
Non-learnable Parameters	8	20
Total static memory	**327,184**	**872,482**
Dynamic memory		
Hidden Layer Forward Step	14,842,624	39,537,664
Output Layer Forward Step	749,376	2,728,256
Loss Calculation	2,560	10,420
Output Layer Update Step	11,520	40960
Hidden Layer Update Step	160,128	438,272
Total dynamic memory	**15,766,202**	**42,755,572**
Total memory	**16,093,386**	**43,628,054**

Table 3.1: Total memory usage of the algorithm by section in bytes. Calculated for the RSNN model trained on the SHD dataset.

The mixed-precision approach reduces memory usage by 63.11% compared to the full-precision floating-point approach. However, this analysis estimates the memory footprint of the training process. While the theoretical calculations provide an upper bound, in practice, hardware-specific memory optimizations (e.g., memory reuse, caching) may reduce the effective memory usage.

3.3.2 Computational cost

The computational cost of the algorithm is determined by the total number of operations made in a training iteration. The operations considered were addition, multiplication, binary multiplication, bit-shift, comparison, absolute value, and exponentiation. Subtraction operations can be counted as additions if the sign is switched and divisions can be counted as multiplications by the reverse number. Table 3.2 shows a breakdown of the operations made during a training iteration in the RSNN model. A complete list of the operations in each model section can be found in the Appendix B.2 in Tables B.3 and B.4.

The number of operations is a function of the shape and the sparsity of the elements. Sparsity has a big influence on the computational costs since the zero-valued elements increase the computation speed. Based on the training data, the effective number of operations was calculated using four different activity coefficients:

- A_I - **Input Activity** represents the proportion of active values (non-zero) in the input spikes. This factor depends on the dataset and influences the operations related to the activations, the voltage update, and the presynaptic trace.
- A_R - **Recurrent Activity** represents the proportion of active values (non-zero) in the recurrent connection. This factor depends on the whole training environment and influences the operations related to the activations, and the voltage update.
- A_W - **Weight Activity** represents the proportion of active values (non-zero) in the weight matrices. This factor depends on the whole training environment and influences the operations related to the activations, the voltage update, and the inference weights update.
- A_O - **Output Activity** represents the proportion of active values (non-zero) in the output spike. This factor depends on the voltage's distribution. In the hidden layer, it is the input sparsity of the next layer. In the output layer, it influences the final prediction and loss calculation.

Model Section	ADD	MUL	B-MUL	Bit-shift	CMP	ABS	EXP
Floating point							
Hidden - FW	116,180,812	115,402,342	655,360	1,059,452	655,360	327,680	-
Output - FW	2,149,454	1,996,881	51,200	361,080	51,200	25,600	-
Error Calculation	128	152	-	2,560	-	-	-
Output - Update	12,780	2,560	-	15,360	-	-	-
Hidden - Update	126,976	37,888	-	134,400	-	-	-
Total Operations	**118,470,151**	**117,439,825**	**706,560**	**1,572,853**	**706,560**	**353,280**	**-**
Integer-only							
Hidden - FW	116,180,812	115,730,022	655,360	-	655,360	327,680	-
Output - FW	2,475,459	2,328,535	51,200	-	51,200	25,600	-
Error Calculation	2,688	2,560	-	-	-	-	617
Output - Update	12,780	12,800	-	-	-	-	-
Hidden - Update	126,976	127,488	-	-	-	-	-
Total Operations	**118,798,715**	**118,201,405**	**706,560**	**-**	**706,560**	**353,280**	**617**

Table 3.2: Total computational cost of the algorithm by section. Calculated for the RSNN model and the SHD dataset.

In the integer-only version, operations like applying the voltage decay or multiplying the learning rate or weight decay rate are bit-shifting operations, while in the floating-point version

those are multiplication operations. Additionally, the difference in loss functions and the activity coefficients obtained experimentally (Table B.5 in Appendix B.2) explain the gap in the number of operations between floating-point and integer-only models.

While the computational cost breakdown includes binary multiplications (B-MUL), comparisons (CMP), absolute values (ABS), and exponentiations (EXP), these operations can be disregarded because they contribute minimally to total energy consumption and are the same for both approaches.

The integer-only approach achieves substantial energy savings by replacing expensive floating-point operations with low-precision integer arithmetic. As seen in Table 3.3, an 8-bit integer multiplication (0.2 pJ) is 18x more efficient than a 32-bit floating-point multiplication (3.7 pJ), while an 8-bit integer addition (0.03 pJ) is 30x more efficient than its 32-bit floating-point counterpart (0.9 pJ) [20]. Given that additions and multiplications form the bulk of computations in forward propagation and weight updates, this substantially reduces the total power consumption.

An additional efficiency gain comes from replacing certain multiplications with bit-shift operations. Bit-shifting is virtually free in hardware compared to multiplication, requiring only simple register shifts instead of complex arithmetic circuits [50].

Operation	**Integer**		**Floating Point**	
	8-bit	**32-bit**	**16-bit**	**32-bit**
Addition	0.03 pJ	0.1 pJ	0.4 pJ	0.9 pJ
Multiplication	0.2 pJ	3.1 pJ	1.1 pJ	3.7 pJ
Bit-shift	0.024 pJ	0.13 pJ	-	-

Table 3.3: Energy consumption (in pJ) for integer and floating-point operations at different bit-widths [20, 50].

3.4 Datasets

In this work, we evaluate our proposed method using two standard benchmarks in the neuromorphic field: MNIST for static vision and the Spiking Heidelberg Digits (SHD) dataset for neuromorphic temporal audio. Both datasets have been widely employed in the literature to assess the performance of spiking neural network models and related architectures.

3.4.1 MNIST

The MNIST dataset comprises 70,000 grayscale images of handwritten digits, partitioned into 60,000 training and 10,000 test samples. Each image is of size 28×28 pixels. The labels range from 0 to 9. Due to its simplicity and standardized preprocessing, MNIST has been extensively used as a benchmark in traditional and neuromorphic computing studies. In our experiments, 10,000 images extracted from the training set have been used for validation, leaving an approximated 70-15-15 train-val-test split.

To use the temporal dynamics of the spiking neural networks in this static dataset, the images fed to the network during the forward pass were the result of a rate-coding technique. The encoding consisted of, for every time step, generating a random image and comparing each value with the pixel intensity. Only the pixel values higher than their corresponding random values were considered as spikes (1), and the rest of the pixel values were set to 0. This process ensured that, along the different timesteps, the pixels with higher intensity would generate more spikes over time.

3.4.2 Spiking Heidelberg Digits

The Spiking Heidelberg Digits (SHD) dataset is a benchmark for temporal neuromorphic tasks, consisting of spiking data generated from audio recordings of spoken digits pronounced by 12 speakers. There are 20 labels corresponding to the 10 digits in English and German. Each digit is repeated approximately 40 times per speaker and there is a total of 10,420 samples divided into train (8156) and test (2264). Following the common split in the bibliography, the test set contains recordings from 2 speakers not present in the train set and 5% of the recordings from the rest of the speakers. Although this split can introduce bias in the test set, it was designed to ensure the generalization capabilities of the models.

The audio files are processed using a 700-channel hydrodynamic basilar membrane model, which simulates the frequency decomposition of the auditory system. The resulting channel movements are converted into spikes via a transmitter pool-based hair cell model, and a subsequent layer of Bushy cells is incorporated to enhance phase locking. These processing steps ensure that the temporal dynamics and spectral information of the original audio are accurately captured in the spike representation.

The SHD dataset is presented in an event format, where each event records the time and channel of a spike. Given the variability in recording lengths, ranging from 0.2 to 1.4 seconds, the spike events are aggregated into 10 uniform time frames per sample, with each frame summing the spike counts for every channel. To address the sparsity of the data and reduce the memory usage and computational cost, the 700 channels are further grouped into sets of 4, reducing the effective channel count to 175 while preserving essential temporal and spectral information.

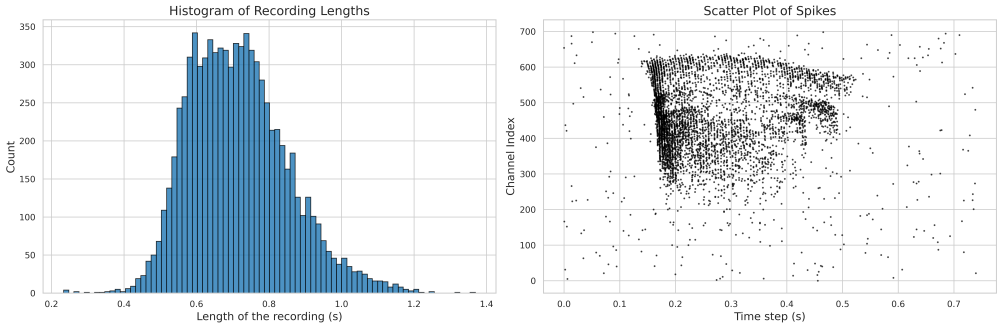

Figure 3.3: (Left) Distribution of the recording's lengths in the training set. (Right) Sample of the training set.

3.4.3 Evaluation Metrics

In this study, model performance is evaluated solely based on classification accuracy. Accuracy serves as a direct and interpretable metric, providing a clear measure of how well the models distinguish between different classes. This approach is consistent with prior research, where accuracy is the predominant metric reported for the MNIST and SHD datasets, facilitating direct comparison with existing work. The reason behind this is that both datasets have balanced class distributions, making it a reliable measure of model performance. In imbalanced datasets, metrics like precision, recall, or AUC are needed to account for class disparities in performance measurement, but they add no additional value in this situation.

Chapter 4

Experimental Results

4.1 Experimental Setup

All experiments were conducted training the models for 50 epochs. The models were evaluated on two datasets: MNIST and SHD, each processed with different temporal encoding strategies. For MNIST, a spike time window of 20 was used, while for SHD, a spike time window of 10 was applied. The number of neurons in the hidden layer was adjusted accordingly to the difficulty and complexity of the dataset: 100 neurons in the MNIST experiments, while 256 neurons in the SHD experiments to match the baseline. A training batch size of 128 and a test batch size of 256 was used in all the experiments. Training was performed for the **MP** and **FP32** configurations. Computations were executed on high-performance GPUs: NVIDIA RTX 6000 Ada Generation (48GB) and NVIDIA A100 (40GB). To mitigate the effects of stochasticity in training, all reported test accuracies represent the mean and standard deviation computed over 10 runs with different random seeds.

4.2 Error Propagation

The different error propagation strategies were assessed using a single-layer SNN trained on the MNIST dataset. The **MP** configuration had a precision of 16 bits in the shadow weights and 8 bits for the inference weights. For the feedback configuration the *soft_error_start* was set to 5, meaning that the error was not propagated during the first epochs due to the minor activity in the output layer. The experiment results are in Table 4.1. Statistical tests to support the improvement of the models can be found in the Appendix C.

Error approach	MP 16-8	FP32
Feedback	94.54 ± 1.89 %	97.33 ± 0.15%
Final	95.14 ± 0.27 %	96.10 ± 0.12 %
Direct	97.54 ± 0.05 %	97.18 ± 0.13%

Table 4.1: Test accuracy on the different loss propagation methods evaluated on the MNIST dataset using a single-layer SNN.

While biologically plausible and dynamically adaptive, the **Feedback** approach exhibited the lowest accuracy and the highest variability in the MP setting. This variability suggests continuous

error propagation through feedback neurons can introduce instability, particularly when working with integer-based arithmetic. Additionally, the computational cost associated with updating the error voltages at every time step makes this approach less appropriate for efficient training. Nevertheless, this approach exhibited the highest accuracy in the FP32 setting. In FP32, the feedback mechanism benefits from higher precision, allowing it to continuously update the voltage error without introducing instability. The **Feedback** propagation was applied in the following FP experiments.

The **Final** approach did not serve as an adequate strategy for any of the configurations. In the MP case it surpassed **Feedback** but not significantly (p-value = 0.2010) and in the FP case it obtained the worst performance. This hybrid option simplifying **Feedback** proved too unstable to process integer error and too simple to process high precision error.

The **Direct** approach achieved the highest test accuracy in the MP setting. This approach entails losing information about how the network's output evolves in the time direction. However, it is also a more stable measurement since it avoids the complexities given by the high variability in the temporal dynamics of the samples; some instances exhibit strong early activity that dissipates, while others show the opposite trend. Besides, integrating loss over time (such as by accumulating voltage changes) would necessitate additional scaling factors to handle the integer values properly, further complicating the training process and potentially obscuring critical information. Besides, since the final prediction reflects the cumulative processing of all input spikes, this approach simplifies the loss computation and focuses directly on it. Compared with the **Final** approach, this method avoids processing the error with the output error neurons, a step that can introduce noise in low-precision operations. Given its superior performance, the **Direct** strategy was selected for further MP experiments in subsequent sections.

4.3 Loss Function

The three loss function variations proposed in the methodology were evaluated in this experiment. The experiments aimed to determine which loss formulation led to the most stable and accurate learning dynamics. Training followed the experimental setup described previously, with a single-layer network on the **MNIST** dataset under the **MP 16-8** configuration. Statistical tests to support the improvement of the loss functions can be found in the Appendix C.

Loss function	SNN - MP 16-8
Standard	97.54 ± 0.05 %
Positive add.	95.06 ± 1.21 %
Random add.	97.55 ± 0.21 %

Table 4.2: Test accuracy on the different loss functions evaluated on the MNIST dataset using a single-layer SNN.

The results indicated that the **Standard Loss** and **Random Additive Loss** outperformed **Final Loss** (p-values of 0.0000 and 0.0009 respectively), with the **Random Additive Loss** exhibiting a higher variability. This suggests that introducing stochastic elements into the loss function may provide a slight generalization advantage and improved robustness to quantization effects. This observation aligns with previous literature experiments adding noise to the gradient or some other part of the SNN training [51, 52, 53]. The **Positive Additive Loss** function, while making the error distribution more similar to the softmax function, did not help the network to learn as much as the other functions. The difference between **Standard Loss** and

Random Additive Loss was not statistically significant (p-value = 0.3710). The **Standard Loss** function was selected for all subsequent experiments because it is computationally more efficient and has slightly lower variability. However, the addition of random noise in the loss function can not be completely dismissed for future works.

4.4 MNIST

4.4.1 Spiking Neural Network

The learning algorithm was first tested on the MNIST dataset. Table 4.3 reports the results obtained with mixed-precision configuration of a single-layer SNN model.

	Shadow weights		**FP32**
Inference weights	**8 bits**	**16 bits**	
4 bits	94.24 ± 0.27 %	95.47 ± 0.15 %	
8 bits	96.89 ± 0.14 %	97.55 ± 0.09 %	97.33 ± 0.15 %
16 bits	-	97.43 ± 0.14 %	

Table 4.3: Performance comparison on MNIST between different mixed-precision configurations. FP32 represents the floating-point full precision configuration.

An interesting finding is that the MP 16-8 configuration outperforms the FP32 baseline by 0.22% (p-value = 0.0012, Table C.2). Theoretically, FP32 should achieve superior performance due to its higher numerical precision, allowing for more precise weight updates and finer gradient adjustments. However, in this experiment, gradient clipping and weight decay techniques applied in the mixed-precision configurations were not applied in the FP32 model, potentially giving MP-16-8 an advantage. This decision was made because lower precision formats are more susceptible to gradient explosion and weight divergence due to reduced numerical precision and discretization effects. This introduces an additional optimization parameter, and further experiments could be conducted to isolate the effect of these techniques in the different configurations. Additionally, the quantization process itself can act in some cases as a form of implicit regularization, introducing controlled noise that reduces overfitting and helps the model generalize better [54]. This phenomenon could also explain the slightly lower accuracy of the MP 16-16 configuration, which is higher for some seeds.

The 4-bit inference weight configuration performed the worst, reaching 94.24% accuracy when paired with 8-bit shadow weights, indicating that 4-bit quantization introduces substantial information loss. The performance improved with 16-bit shadow weights (95.47%), confirming the idea that higher precision in shadow weights allows for more capacity to store information and features.

The results indicate that inference weight precision plays a more critical role in model performance than shadow weight precision. Increasing inference precision from 4-bit to 8-bit yields a substantial accuracy improvement, as seen in MP-16-4 (95.47%) to MP-16-8 (97.55%) and in MP-8-4 (94.24%) to MP-8-8 (96.89%), whereas increasing shadow weight precision from 8-bit to 16-bit has a smaller impact, as observed in MP-8-8 (96.89%) to MP-16-8 (97.55%), and in MP-8-4 (94.24%) to MP-16-4 (95.47%). This suggests that inference weights are marginally more crucial for maintaining representational capacity than shadow weights.

4.4.2 Convolutional Spiking Neural Network

Table 4.4 reports the results obtained with the single-layer CSNN model in the MNIST dataset. Different mixed-precision configurations were evaluated. For all the models, the **kernel size** was set to 5×5, with 32 **filters** per layer. The **padding** and **stride** were set to 0 and 2 respectively to ensure that the image size was reduced by half.

	Shadow weights		**FP32**
Inference weights	**8 bits**	**16 bits**	
4 bits	95.19 ± 0.84 %	97.73 ± 0.19 %	
8 bits	97.16 ± 0.20 %	98.1 ± 0.10 %	97.85 ± 0.12 %
16 bits	-	97.88 ± 0.17 %	

Table 4.4: Performance comparison on MNIST between different mixed-precision configurations. FP32 represents the floating-point full precision configuration.

The results obtained with the single-layer Convolutional Spiking Neural Network (CSNN) on MNIST reveal that higher inference weight precision leads to more significant performance improvements than higher shadow weight precision, similar to the trends observed in the SNN model. However, the impact of mixed precision in the CSNN appears to be less pronounced due to the structural advantages of convolutional layers in extracting robust spatial features.

The MP-16-8 configuration (98.1%) slightly outperformed FP32 (97.85%), suggesting that quantization effects may introduce a form of implicit regularization, as previously observed in the SNN model. A key finding is the notable drop in performance in the MP-8-4 configuration (95.19%), highlighting the diminishing capabilities of the model in ultra-low precision environments. Conversely, increasing shadow precision from 16-bit or inference precision to 8-bit results in a substantial improvement.

Overall, these results indicate that CSNNs are more resilient to quantization effects than fully connected SNNs, as evidenced by the small performance gap between MP-16-8 and MP-8-8 and MP-16-4. This suggests that convolutional architectures benefit from local receptive fields and weight sharing to mitigate the impact of quantization noise, making them better suited for mixed-precision training than SNNs.

4.5 SHD

The learning algorithm was also tested on the neuromorphic SHD dataset. The results are compared with the work done by Cramer et al. [12], which obtained the highest performances in the state-of-the-art with classic SNNs and RSNNs. The SNN and RSNN models in that work are trained using surrogate gradient descent in combination with BPTT. Table 4.5 reports the performance of the MP configurations and a comparison with the FP32 configuration and the work by Cramer et al.

4.5.1 Spiking Neural Network

For the SNN model, the MP-16-12 (62.06%) and MP-16-16 (61.92%) configurations outperform FP32 (55.27%), which could be attributed to quantization advantages similar to those observed in the MNIST experiments. The lack of gradient clipping and weight decay in the FP32 model, as well as the impact of fine-tuning, may have had a greater impact in this dataset. Notably,

Model	MP 16-4	MP 16-8	MP 16-12	MP 16-16	FP32
SNN	49.85 ± 1.29 %	50.10 ± 1.14 %	62.06 ± 1.16 %	61.92 ± 1.53 %	55.27 ± 1.97 %
SNN [12]	-	-	-	-	48.10 ± 1.60 %
RSNN	57.62 ± 0.95 %	64.63 ± 1.49 %	70.50 ± 1.43 %	67.75 ± 1.34 %	71.64 ± 0.95 %
RSNN [12]	-	-	-	-	71.40 ± 1.90 %

Table 4.5: Comparison of accuracy on SHD between the Mixed-Precision Integer and the FP32 approaches for the different models.

all mixed-precision configurations exceed the accuracy of the BPTT-trained baseline (48.1%), demonstrating the efficacy of the proposed online learning approach in SNNs.

Interestingly, no improvement is observed from MP-16-12 to MP-16-16. This aligns with previous findings where structured weight constraints, such as quantization, can improve generalization. It could also suggest that increasing inference weight precision beyond 12-bit does not provide additional benefits, but given the high complexity of the dataset, higher precision would theoretically be expected to enhance performance.

At lower inference precisions, performance drops significantly, with MP-16-4 (49.85%) and MP-16-8 (50.1%) matching the baseline but struggling to maintain the competitive accuracy of the higher-precision configurations. This highlights the complexity of the SHD dataset, where precise spike-timing information is crucial for classification. Unlike MNIST, which primarily relies on spatial feature encoding, SHD depends heavily on temporal feature representations, making it more sensitive to quantization-induced distortions.

4.5.2 Recurrent Spiking Neural Network

In the RSNN model, a similar trend is observed, but with some distinctions due to recurrent connections. The MP-16-12 and MP-16-16 configurations closely match the FP32 baseline (71.64%), confirming that random recurrent connections can achieve near-optimal performance while reducing computational training costs. The MP-16-16 model is again outperformed by MP-16-12, indicating that, like in the SNN case, additional precision does not guarantee better performance once a certain minimum is reached. In this case, only the MP-16-12 matched the accuracy of the BPTT-trained baseline (71.4%), further demonstrating the effectiveness of the proposed learning algorithm.

However, at lower precisions, the RSNN model experiences a significant accuracy drop (57.6% in MP-16-4, 64.63% in MP-16-8). The performance decline at lower bit widths is amplified in RSNNs, due to the recurrent connections. As they introduce additional weights and information into the voltage, the lower precision of the elements results in a bigger loss of information. These findings emphasize that recurrent architectures require greater precision in inference weights than feedforward SNNs to maintain effective temporal modeling.

These results indicate that mixed-precision training with at least 12-bit inference weights is a viable alternative to full-precision models, achieving competitive or even superior performance in some seeds. The strong performance of MP models over BPTT-trained baselines further supports the potential of the proposed learning approach for neuromorphic tasks. Furthermore, the observed performance degradation at lower precisions highlights the importance of maintaining sufficient numerical precision in recurrent spiking networks, particularly for tasks requiring temporal feature extraction.

Chapter 5

Discussion

This research proposed an efficient online integer-only mixed-precision training algorithm for SNNs, significantly reducing memory usage and computational costs compared to full-precision models. The MP method was implemented for SNNs, CSNNs and RSNNs.

The first experiments compared different error propagation approaches and loss functions. Then, the effectiveness of the proposed method was demonstrated across two benchmark datasets, MNIST and SHD, showcasing competitive performance while leveraging quantization advantages for improved efficiency. Lower inference precision introduced significant quantization noise that disrupted error propagation and learning dynamics, leading to greater performance degradation than lower shadow weight precision. While high-precision shadow weights help stabilize training, their benefit was constrained if inference weights lacked sufficient precision to preserve meaningful updates.

The results highlight the viability of the proposed mixed-precision integer-based SNN training method, successfully balancing computational efficiency and accuracy, and presenting a viable pathway for efficient neuromorphic learning.

5.1 Research Questions

How can quantization and low-precision concepts be applied to improve the efficiency in the learning process of spiking neural networks?

Quantization and low-precision concepts were integrated in multiple sections of the learning algorithm to enhance computational efficiency while maintaining competitive accuracy.

The mixed-precision approach uses one high-precision set of weights for gradient updates and a low-precision set for the inference pass computations. This method allowed the model to maintain competitive accuracy in static and dynamic datasets while reducing the memory footprint by over 60% compared to the full-precision floating point version.

Using integer-only arithmetic instead of floating-point operations in every part of the training significantly reduced the computational costs. The difference in energy consumption of integer and floating-point operations is boosted by using low-precision integer formats as part of the mixed-precision approach. In addition, key operations such as the voltage decay, loss function, learning rate, and weight decay rate were adapted to use bit-shifting instead of multiplication or division, further lowering the computational cost of the algorithm.

The selection of integer-only formats also influenced the error propagation strategy. The Direct error propagation approach achieved the best performance in the mixed-precision config-

urations and, at the same time, yielded more numerical stability, and reduced the number of operations.

By integrating these techniques, the learning algorithm became significantly more efficient in terms of memory usage and computational costs. The SSN benefited from the efficiency gain without compromising classification performance.

What trade-offs, if any, are introduced by the new methods in terms of training time, inference speed, and model accuracy?

The precision of the weights played a crucial role in the model's performance. while shadow weights at higher precision help stabilize learning, inference weight precision demonstrated a more crucial role in determining the final accuracy. Ultra-low precision in the inference weights led to diminishing results with higher variability due to the quantization noise, especially in time-related tasks like SHD. However, high precision configurations (MP-16-16) didn't achieve substantial improvements; in some cases, the performance was a bit lower than lower precision configurations. The trade-off between precision and performance implies that optimal configurations depend on the dataset complexity. For simpler, spatially oriented datasets like MNIST, even lower inference weight precision (8-bit) can yield near-optimal results, whereas more temporally complex datasets benefit from higher inference precision (12-bit).

Regarding the training time and inference speed, the algorithm has some properties that indicate that the execution should be faster than the full-precision floating-point approach. Integer multiplications and bit-shift operations are significantly more efficient than floating-point multiplications [20]. The reduced memory footprint contributes to the execution time, speeding up the data access processes. Besides, simplifying some operations like the loss function (multiplication and bit shifting instead of exponentiation) and the error propagation strategy also reduces the training time. While strict experimental measurements were not conducted, theoretical analysis suggests a faster training and inference time than full-precision floating-point models.

How can the learning algorithm be designed while considering realistic neuromorphic hardware processing for real hardware efficiency?

Realistic neuromorphic hardware presents important memory and computational constraints that need to be addressed when designing a learning algorithm. These constraints include limited on-chip memory, in-memory computing, energy limitations, real-time processing, locality constraints, precision limitations or scalability problems [55, 56, 57]. The proposed learning algorithm incorporates several techniques to address these demands:

Integer-only arithmetic facilitates the operations since fixed-point operations are faster and easier than floating-point. This design decision can be crucial for some integer-specific neuromorphic hardware. Bit-shift operations are preferred in these hardware, therefore replacing multiplications and divisions will make the algorithm more efficient. In addition, the flexibility of the mixed-precision method allows for more scalability and adaptability to fulfill the possible memory and precision constraints of specific hardware.

A simplified architecture can make the model more compatible with specialized neuromorphic hardware. The simplifications in the direct error propagation, and the loss function computation reduce the number of operations and intermediate variables. Local and online learning techniques further contribute to meet the possible constraints of real-time processing, locality computing, or limited on-chip memory.

5.2 Limitations

Although the learning method is derived and extended for multilayer SNNs, CSNNs, and RSNNs, only single-layer models have been evaluated in this work. This may limit the real efficacy of the proposed method for multi-layer models since extra measures should be implemented to control the numerical instability of the integer-only formats.

The method's gains in memory usage and computational cost compared with the full-precision floating-point approaches have only been computed theoretically. Empirical experiments would need to be carried out in integer-specific hardware but would add more reliability to the work.

The method performance is not compared with similar alternatives like surrogate gradient methods or event-driven learning rules. The inclusion of more baselines would contextualize better the efficacy of the proposed learning algorithm.

5.3 Future Work

The SNN model can be improved by implementing recent features like learning delays, encouraging the model to synchronize spikes and increase expressivity [58]. Besides, extending the method to more complex neuron models like adaptive LIF neurons would make the learning method more versatile.

Further experiments to achieve competitive accuracy with lower precision can be carried out. Improvements in quantization techniques like stochastic rounding or adaptive threshold could help the model to improve the performance in low-precision configurations.

Sparsity rules can be implemented to further increase the model's efficiency. Some examples of these rules are updating only the weights which neurons fired during the inference, limiting the layer percentage of activation, or reducing the connections between inactive neurons [29, 36].

Chapter 6

Conclusion

This work proposed a novel learning method for SNNs that is local, online, and based on back-propagation principles and RTRL. The approach relies on local eligibility traces to approximate gradient updates, enabling efficient training in spiking architectures while maintaining biological plausibility.

A key innovation of the method is its compatibility with integer-only arithmetic, restricting the available operations in exchange for huge efficiency. The learning method utilizes a mixed-precision approach in which a low-precision version of the weights is used in the inference and error propagation steps and a higher-precision set is used to update the gradients. A theoretical analysis showed that memory usage is reduced by over 60% compared with a full-precision floating-point version. The algorithm reduces operations by enabling a fast and direct error propagation strategy and simplifies multiplications and divisions by adapting them to bit-shifting operations. These features make the learning algorithm remarkably more efficient in terms of computational cost, and make it feasible for hardware-friendly deployment. The method was extended beyond fully connected SNNs to include Convolutional Spiking Neural Networks (CSNNs) and Recurrent Spiking Neural Networks (RSNNs), demonstrating its adaptability to various network architectures.

Experimental results validated the method's effectiveness, obtaining in some cases a higher accuracy than the full-precision floating-point models. Results on MNIST achieved competitive accuracy using the SNN and CSNN models with combinations of 8 and 16-bit precisions, but the performance dropped when introducing 4-bit precision for the inference weights. The CSNN model proved more resistant to low-precision in the inference weight. In the SHD dataset, the SNN and RSNN models exhibited competitive accuracy with a precision of 12 bits in the inference weights, lower precision led to the introduction of quantification noise and less capabilities.

In conclusion, the method achieved comparable accuracy to full-precision approaches while significantly reducing memory usage and computational cost. Future directions include improving multi-layer training stability, refining quantization techniques, and conducting empirical evaluations on dedicated neuromorphic processors. The findings highlight the potential of a complete integer-based, local and online learning method for SNNs, advancing neuromorphic computing and enabling more energy-efficient SNN implementations on specialized hardware for real-world applications.

Bibliography

[1] . Nature Computational Scienc. Boosting AI with neuromorphic computing. *Nature Computational Science*, 5(1):1–2, January 2025. Publisher: Nature Publishing Group.

[2] Shawn Cox. The future of control systems: trends and predictions, June 2023.

[3] Nitin Rathi, Indranil Chakraborty, Adarsh Kosta, Abhronil Sengupta, Aayush Ankit, Priyadarshini Panda, and Kaushik Roy. Exploring Neuromorphic Computing Based on Spiking Neural Networks: Algorithms to Hardware. *ACM Comput. Surv.*, 55(12):243:1–243:49, March 2023.

[4] Xubin Wang and Weijia Jia. Optimizing Edge AI: A Comprehensive Survey on Data, Model, and System Strategies, January 2025. arXiv:2501.03265.

[5] Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Shweta Jain, Yuyun Liao, Chit-Kwan Lin, Andrew Lines, Ruokun Liu, Deepak Mathaikutty, Steven McCoy, Arnab Paul, Jonathan Tse, Guruguhanathan Venkataramanan, Yi-Hsin Weng, Andreas Wild, Yoonseok Yang, and Hong Wang. Loihi: A Neuromorphic Manycore Processor with On-Chip Learning. *IEEE Micro*, 38(1):82–99, January 2018.

[6] Hector A. Gonzalez, Jiaxin Huang, Florian Kelber, Khaleelulla Khan Nazeer, Tim Langer, Chen Liu, Matthias Lohrmann, Amirhossein Rostami, Mark Schöne, Bernhard Vogginger, Timo C. Wunderlich, Yexin Yan, Mahmoud Akl, and Christian Mayr. SpiNNaker2: A Large-Scale Neuromorphic System for Event-Based and Asynchronous Machine Learning, January 2024. arXiv:2401.04491.

[7] Yujie Wu, Lei Deng, Guoqi Li, Jun Zhu, and Luping Shi. Spatio-Temporal Backpropagation for Training High-Performance Spiking Neural Networks. *Frontiers in Neuroscience*, 12:331, May 2018.

[8] Sumit Bam Shrestha and Garrick Orchard. SLAYER: Spike Layer Error Reassignment in Time, September 2018. arXiv:1810.08646.

[9] Tharuki De Silva and Pradeepa Yahampath. An Investigation of Latency-Accuracy Trade-off in Inter-frame Video Prediction using Quantized CNNs. In *2023 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 19–24, September 2023. ISSN: 2576-7046.

[10] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2704–2713, June 2018. Conference Name:

2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) ISBN: 9781538664209 Place: Salt Lake City, UT Publisher: IEEE.

[11] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998. Conference Name: Proceedings of the IEEE.

[12] Benjamin Cramer, Yannik Stradmann, Johannes Schemmel, and Friedemann Zenke. The Heidelberg Spiking Data Sets for the Systematic Evaluation of Spiking Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems*, 33(7):2744–2757, July 2022. Conference Name: IEEE Transactions on Neural Networks and Learning Systems.

[13] Wolfgang Maass. Networks of spiking neurons: The third generation of neural network models. *Neural Networks*, 10(9):1659–1671, December 1997.

[14] Kashu Yamazaki, Viet-Khoa Vo-Ho, Darshan Bulsara, and Ngan Le. Spiking Neural Networks and Their Applications: A Review. *Brain Sciences*, 12(7):863, June 2022.

[15] Jason K. Eshraghian, Max Ward, Emre O. Neftci, Xinxin Wang, Gregor Lenz, Girish Dwivedi, Mohammed Bennamoun, Doo Seok Jeong, and Wei D. Lu. Training Spiking Neural Networks Using Lessons From Deep Learning. *Proceedings of the IEEE*, 111(9):1016–1054, September 2023. Conference Name: Proceedings of the IEEE.

[16] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper, June 2018. arXiv:1806.08342.

[17] Daniel Schnöll, Matthias Wess, Matthias Bittner, Maximilian Götzinger, and Axel Jantsch. Fast, Quantization Aware DNN Training for Efficient HW Implementation. *2023 26th Euromicro Conference on Digital System Design (DSD)*, pages 700–707, September 2023. Conference Name: 2023 26th Euromicro Conference on Digital System Design (DSD) ISBN: 9798350344196 Place: Golem, Albania Publisher: IEEE.

[18] Deepika Bablani, Jeffrey L. Mckinstry, Steven K. Esser, Rathinakumar Appuswamy, and Dharmendra S. Modha. Efficient and Effective Methods for Mixed Precision Neural Network Quantization for Faster, Energy-efficient Inference, January 2024. arXiv:2301.13330.

[19] Matthieu Martel. Enhancing the implementation of mathematical formulas for fixed-point and floating-point arithmetics. *Formal Methods in System Design*, 35(3):265–278, December 2009.

[20] Mark Horowitz. 1.1 Computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14, February 2014. ISSN: 2376-8606.

[21] Jae-Su Song and Fangzhen Lin. PocketNN: Integer-only Training and Inference of Neural Networks via Direct Feedback Alignment and Pocket Activations in Pure C++. *ArXiv*, January 2022.

[22] Maolin Wang, Seyedramin Rasoulinezhad, Philip H. W. Leong, and Hayden K. H. So. NITI: Training Integer Neural Networks Using Integer-only Arithmetic. *IEEE Transactions on Parallel and Distributed Systems*, 33(11):3249–3261, November 2022. arXiv:2009.13108.

[23] Alireza Ghaffari, Marzieh S. Tahaei, Mohammadreza Tayaranian, Masoud Asgharian, and Vahid Partovi Nia. Is Integer Arithmetic Enough for Deep Learning Training?, January 2023. arXiv:2207.08822.

[24] Qingyan Meng, Mingqing Xiao, Shen Yan, Yisen Wang, Zhouchen Lin, and Zhi-Quan Luo. Towards Memory- and Time-Efficient Backpropagation for Training Spiking Neural Networks, August 2023. arXiv:2302.14311.

[25] Emre O. Neftci, Hesham Mostafa, and Friedemann Zenke. Surrogate Gradient Learning in Spiking Neural Networks, May 2019. arXiv:1901.09948.

[26] Christopher Aicher, N. Foti, and E. Fox. Adaptively Truncating Backpropagation Through Time to Control Gradient Bias. *ArXiv*, May 2019.

[27] Kazuki Irie, Anand Gopalakrishnan, and Jürgen Schmidhuber. Exploring the Promise and Limits of Real-Time Recurrent Learning, February 2024. arXiv:2305.19044.

[28] Mingqing Xiao, Qingyan Meng, Zongpeng Zhang, Di He, and Zhouchen Lin. Online Training Through Time for Spiking Neural Networks, December 2022. arXiv:2210.04195.

[29] Nicolas Perez-Nieves and Dan F. M. Goodman. Sparse Spiking Gradient Descent, May 2021.

[30] Guillaume Bellec, Franz Scherr, Anand Subramoney, Elias Hajek, Darjan Salaj, Robert Legenstein, and Wolfgang Maass. A solution to the learning dilemma for recurrent networks of spiking neurons. *Nature Communications*, 11(1):3625, July 2020.

[31] Guangzhi Tang, Neelesh Kumar, Ioannis Polykretis, and Konstantinos P. Michmizos. BioGrad: Biologically Plausible Gradient-Based Learning for Spiking Neural Networks, October 2021. arXiv:2110.14092.

[32] Charlie Blake, Douglas Orr, and Carlo Luschi. Unit Scaling: Out-of-the-Box Low-Precision Training. *International Conference on Machine Learning*, 2023. Publisher: arXiv Version Number: 2.

[33] Jyotibdha Acharya, Laxmi R Iyer, and Wenyu Jiang. Low Precision Local Learning for Hardware-Friendly Neuromorphic Visual Recognition. In *ICASSP 2022 - 2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 8937–8941, May 2022. ISSN: 2379-190X.

[34] Jacques Kaiser, Hesham Mostafa, and Emre Neftci. Synaptic Plasticity Dynamics for Deep Continuous Local Learning (DECOLLE). *Frontiers in Neuroscience*, 14:424, May 2020. arXiv:1811.10766.

[35] Ayan Shymyrbay, Mohammed E. Fouda, and Ahmed Eltawil. Low Precision Quantization-aware Training in Spiking Neural Networks with Differentiable Quantization Function. In *2023 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, June 2023. ISSN: 2161-4407.

[36] Ahmed Hasssan, Jian Meng, Anupreetham Anupreetham, and Jae-sun Seo. SpQuant-SNN: ultra-low precision membrane potential with sparse activations unlock the potential of on-device spiking neural networks applications. *Frontiers in Neuroscience*, 18:1440000, September 2024.

[37] S. G. Hu, G. C. Qiao, T. P. Chen, Q. Yu, Y. Liu, and L. M. Rong. Quantized STDP-based online-learning spiking neural network. *Neural Computing and Applications*, 33(19):12317–12332, October 2021.

[38] Chen Li, Lei Ma, and Steve Furber. Quantization Framework for Fast Spiking Neural Networks. *Frontiers in Neuroscience*, 16, July 2022. Publisher: Frontiers.

[39] Wenjie Wei, Yu Liang, Ammar Belatreche, Yichen Xiao, Honglin Cao, Zhenbang Ren, Guoqing Wang, Malu Zhang, and Yang Yang. Q-SNNs: Quantized Spiking Neural Networks, June 2024. arXiv:2406.13672.

[40] Alberto Pirillo, Luca Colombo, and Manuel Roveri. NITRO-D: Native Integer-only Training of Deep Convolutional Neural Networks. 2024. Publisher: arXiv Version Number: 2.

[41] Chenglong Zou, Xiaoxin Cui, Shuo Feng, Guang Chen, Yi Zhong, Zhenhui Dai, and Yuan Wang. An all integer-based spiking neural network with dynamic threshold adaptation. *Frontiers in Neuroscience*, 18, December 2024. Publisher: Frontiers.

[42] Xinhao Luo, Man Yao, Yuhong Chou, Bo Xu, and Guoqi Li. Integer-Valued Training and Spike-Driven Inference Spiking Neural Network for High-performance and Energy-efficient Object Detection, August 2024. arXiv:2407.20708.

[43] Timothy P. Lillicrap, Daniel Cownden, Douglas B. Tweed, and Colin J. Akerman. Random feedback weights support learning in deep neural networks, November 2014. arXiv:1411.0247 [q-bio].

[44] Jesus Garcia Fernandez, Sander Keemink, and Marcel van Gerven. Gradient-Free Training of Recurrent Neural Networks using Random Perturbations. *Frontiers in Neuroscience*, 18:1439155, July 2024. arXiv:2405.08967.

[45] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1026–1034. IEEE, 2015.

[46] Kang Zhao, Sida Huang, Pan Pan, Yinghan Li, Yingya Zhang, Zhenyu Gu, and Yinghui Xu. Distribution Adaptive INT8 Quantization for Training CNNs, February 2021. arXiv:2102.04782.

[47] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training Recurrent Neural Networks, February 2013. arXiv:1211.5063.

[48] Naveen Mellempudi, Sudarshan Srinivasan, Dipankar Das, and Bharat Kaul. Mixed Precision Training With 8-bit Floating Point, May 2019. arXiv:1905.12334.

[49] Bharat Singh, Soham De, Yangmuzi Zhang, Thomas Goldstein, and Gavin Taylor. Layer-Specific Adaptive Learning Rates for Deep Networks, October 2015. arXiv:1510.04609.

[50] Haoran You, Xiaohan Chen, Yongan Zhang, Chaojian Li, Sicheng Li, Zihao Liu, Zhangyang Wang, and Yingyan Lin. ShiftAddNet: A Hardware-Inspired Deep Network, October 2020. arXiv:2010.12785.

[51] Chunming Jiang and Yilei Zhang. A Noise-Based Novel Strategy for Faster SNN Training. *Neural Computation*, 35(9):1593–1608, August 2023.

[52] Arvind Neelakantan, Luke Vilnis, Quoc V. Le, Ilya Sutskever, Lukasz Kaiser, Karol Kurach, and James Martens. Adding Gradient Noise Improves Learning for Very Deep Networks, November 2015. arXiv:1511.06807.

[53] Ziming Wang, Runhao Jiang, Shuang Lian, Rui Yan, and Huajin Tang. Adaptive Smoothing Gradient Learning for Spiking Neural Networks. In *Proceedings of the 40th International Conference on Machine Learning*, pages 35798–35816. PMLR, July 2023. ISSN: 2640-3498.

[54] Ron Banner, Itay Hubara, Elad Hoffer, and Daniel Soudry. Scalable Methods for 8-bit Training of Neural Networks, June 2018. arXiv:1805.11046.

[55] Lorenz K. Muller, Pascal Stark, Bert Jan Offrein, and Stefan Abel. Neuromorphic Systems Design by Matching Inductive Biases to Hardware Constraints. *Frontiers in Neuroscience*, 14:437, May 2020.

[56] Rachmad Vidya Wicaksana Putra and Muhammad Shafique. NeuroNAS: A Framework for Energy-Efficient Neuromorphic Compute-in-Memory Systems using Hardware-Aware Spiking Neural Architecture Search, December 2024. arXiv:2407.00641.

[57] Christopher Wolters, Brady Taylor, Edward Hanson, Xiaoxuan Yang, Ulf Schlichtmann, and Yiran Chen. Biologically Plausible Learning on Neuromorphic Hardware Architectures, April 2023. arXiv:2212.14337.

[58] Ilyass Hammouamri, Ismail Khalfaoui-Hassani, and Timothée Masquelier. Learning Delays in Spiking Neural Networks using Dilated Convolutions with Learnable Spacings, December 2023. arXiv:2306.17670.

Appendix A

Mathematical derivation of the learning rule

The learning rule used in this algorithm is derived from the principles of spatiotemporal backpropagation and Real-Time Recurrent Learning (RTRL). The derivation follows a structured application of the chain rule to determine the weight updates.

First, considering the standard backpropagation formulation, the weight update is computed as:

$$\Delta W^{(i)} = \frac{\partial L(t)}{\partial W^{(i)}} = \sum_{t=1}^{T} \frac{\partial L(t)}{\partial V^{(i)}(t)} \cdot \frac{\partial V^{(i)}(t)}{\partial W^{(i)}} \tag{A.1}$$

where L is the loss function, $V(t)$ is the membrane potential at time t, and W represents the synaptic weights.

Following the RTRL framework, the derivative of the membrane potential $V(t)$ with respect to the weight W can be obtained separating the term into prior and present influences. The prior influences term ($\mathrm{t} \in [1, T-1]$) defines how the voltage from the previous timestep influences the present voltage. We compute this term deriving the update rule of the Leaky Integrate-and-Fire (LIF) neuron model (Equation 2.1). The present influence term translates into the spike input received by the neuron at the present timestep.

$$\frac{\partial V(t)}{\partial W} = \sum_{t=1}^{T} \frac{\partial V(t)}{\partial W} = \sum_{t=1}^{T-1} \frac{\partial V(t)}{\partial W} + \frac{\partial V(t)}{\partial W} = \sum_{t=1}^{T-1} \frac{\partial V(t)}{\partial V(t-1)} \cdot \frac{\partial V(t-1)}{\partial W} + \frac{\partial V(t)}{\partial W} \tag{A.2}$$

$$\frac{\partial V(t)}{\partial W} = \beta \cdot \frac{\partial V(t-1)}{\partial W} + S_{input}(t) \tag{A.3}$$

where $\frac{\partial V(t-1)}{\partial W}$ can be interpreted as the **presynaptic trace**, β is the voltage decay factor, and $S_{input}(t)$ is the presynaptic spike input. This recurrence effectively captures the temporal dependencies of the synaptic updates.

Next, we compute the derivative of the loss function with respect to the membrane potential:

$$\frac{\partial L(t)}{\partial V(t)} = \frac{\partial L(t)}{\partial S(t)} \cdot \frac{\partial S(t)}{\partial V(t)} \tag{A.4}$$

where $S(t)$ denotes the postsynaptic spike activity. The loss function is defined as the discrepancy between the network's cumulative output and the target:

$$L(t) = \frac{Output(t)}{t_s} - y = \frac{Output(t-1) + S_{out}(t)}{t_s} - y \tag{A.5}$$

where t_s is a scaling factor indicating the number of time steps, $Output(t)$ represents the cumulative network output, and y is the ground truth label.

To compute the derivative of the loss with respect to the spike activity, we observe that:

$$\frac{\partial L(t)}{\partial S(t)} = \frac{\partial L(t)}{\partial Output(t)} \cdot \frac{\partial Output(t)}{\partial S(t)} = \frac{1}{t_s} \cdot \frac{\partial Output(t-1)}{\partial S(t)} \tag{A.6}$$

Since $Output(t-1)$ always depends on the output layer, a local learning rule requires an approximation when calculating this term for the hidden layers. To ensure efficiency, we approximate this term using the feedback voltage $V_{fb}(t)$, which serves as an error signal.

$$\frac{\partial L(t)}{\partial S(t)} \approx V_{fb}(t) \tag{A.7}$$

For the hidden layers, the feedback voltage term $V_{fb}(t)$ will be calculated following the chain rule and using the transposed weights from the previous layer.

Furthermore, since the output spike $S(t)$ is non-differentiable due to the threshold operation, we approximate its derivative with respect to the weight using a pseudo-gradient:

$$\frac{\partial S(t)}{\partial W(t)} = \widetilde{\nabla V(t)} \tag{A.8}$$

where $\widetilde{\nabla V(t)}$ represents a surrogate gradient function, in our implementation a rectangular function.

Finally, combining all these components, the learning rule for synaptic weight updates is obtained as:

$$\Delta W = V_{fb}(t) \cdot \widetilde{\nabla V}^{(i)} \cdot \frac{\partial V(t)}{\partial W(t)} \tag{A.9}$$

We can consider $\widetilde{\nabla V}^{(i)} \cdot \frac{\partial V(t)}{\partial W(t)}$ a **post-synaptic trace**, since it captures the relationship between the neuron spikes, voltage and weights.

This formulation ensures that learning is local and online, allowing updates based on presynaptic and postsynaptic activity while maintaining computational efficiency.

Comparison with BioGrad

Several works have used learning rules with eligibility traces for SNN training [30, 31, 28]. In this section, we compare our learning rule with the BioGrad work. Although both methods are local, online and gradient-based, there are two main differences with our learning rule.

First, the BioGrad network uses two sets of weights in each neuron to be more efficient and biologically plausible. So the propagation of the feedback voltage through the hidden layers is made with the specific feedback weights of that layer. In this work, we use the transpose of the previous layer's weights for that purpose. This difference makes our algorithm less biologically plausible (the brain doesn't use identical copies of the weights) but more memory efficient since only one copy of the weights is necessary.

Second, the pre-synaptic trace in Biograd is also derived from spatiotemporal backpropagation but it is a more complex version. It includes an additional Decay term, which regulates the previous time-step value depending on the local neuronal states (output spike, voltage and pseudogradient) from past timesteps. This term introduces more complex dynamics in the learning process and allows the network to capture better the influence of the presynaptic inputs. However, the additional computational complexity would introduce more instability when training a model based on integer-only arithmetic.

Appendix B

Memory Usage and Computational Cost

B.1 Memory Usage

Table B.1: Static memory elements in a training loop in the RSNN model for the SHD dataset

Model Section	Element	Shape	Update Frequency	Precision Int	Precision Float
Hidden Layer Forward Step					
	W_{FW} - Shadow	(H, I)	1	16	32
	W_{FW} - Low Precision	(H, I)	1	8	32
	W_{Rec} - Shadow	(H, H)	1	16	32
	W_{Rec} - Low Precision	(H, H)	1	8	32
	Voltage threshold	(1)	0	16	32
	Gradient Window	(1)	0	16	32
Output Layer Forward Step					
	W_{FW} - Shadow	(O, H)	1	16	32
	W_{FW} - Low Precision	(O, H)	1	8	32
	Voltage threshold	(1)	0	16	32
	Gradient Window	(1)	0	16	32
Non-learnable Training Elements					
	Loss precision	(1)	0	8	-
	Ts	(1)	0	6	-
	Gradient clip	(1)	0	9	32
	Learning rate - Hidden	(1)	0	13	32
	Learning rate - Output	(1)	0	3	32
	Weight decay rate - Hidden	(1)	0	12	32
	Weight decay rate - Output	(1)	0	12	32

Table B.2: Dynamic memory elements in a training loop in the RSNN model for the SHD dataset

Model Section	Element	Shape	Update Frequency	Precision Int	Precision Float
Hidden Layer Forward Step					
	Spike Input	(B, I)	Spike time	8	32
	Volt	(B, H)	Spike time	17	32
	Pseudo gradient	(B, H)	Spike time	1	1
	Presynaptic trace	(B, H, I)	Spike time	8	32
	Weight trace	(B, H, I)	Spike time	16	32
	Spike_output	(B, H)	Spike time	1	1
Output Layer Forward Step					
	Spike Input	(B, H)	Spike time	1	1
	Volt	(B, O)	Spike time	16	32
	Pseudo gradient	(B, O)	Spike time	1	1
	Presynaptic trace	(B, O, H)	Spike time	1	1
	Weight trace	(B, O, H)	Spike time	8	32
	Network_output	(B, O)	Spike time	8	32
Error Calculation					
	Error	(B, O)	1	8	32
Output Layer Update Step					
	Delta	(O, H)	1	9	32
	Weight decay term	(O, H)	1	9	32
Hidden Layer Update Step					
	Error voltage	(B, H)	1	18	32
	Delta	(H, I)	1	9	32
	Weight decay term	(H, I)	1	9	32

B.2 Computational Cost

Model Section	Element	Type of Operation	Number of Operations	Operation Frequency
Hidden Layer FW				
	Forward activation	MM	MUL: $A_W^{hid} \cdot A_I \cdot B \cdot H \cdot I$ ADD: $A_W^{hid} \cdot A_I \cdot B \cdot H \cdot (I-1)$	Spike time
	Recurrent activation	MM	MUL: $A_W^{hid} \cdot A_{Wrec}^{hid} \cdot B \cdot H \cdot H$ ADD: $A_W^{hid} \cdot A_{Wrec}^{hid} \cdot B \cdot H \cdot (H-1)$	Spike time
	Bit-shifting of FW_act	BS	$A_W^{hid} \cdot A_I \cdot B \cdot H$	Spike time
	Bit-shifting of Rec_act	BS	$A_{Wrec}^{hid} \cdot A_{Wrec}^{hid} \cdot B \cdot H$	Spike time
	Voltage update	MUL/BS B-MUL ADD	$B \cdot H$ $B \cdot H$ $A_W^{hid} \cdot B \cdot H \cdot (A_I^{hid} + A_R^{hid})$	Spike time
	Spike output	CMP	$B \cdot H$	Spike time
	Pseudo gradient	SUB ABS CMP	$B \cdot H$ $B \cdot H$ $B \cdot H$	Spike time
	Presynaptic trace	BS ADD	$B \cdot I$ $A_I^{hid} \cdot B \cdot I$	Spike time
	Weight trace	B-MUL ADD	$B \cdot H$ $B \cdot H$	Spike time
Output Layer FW				
	Forward activation	MM	MUL: $A_W^{out} \cdot A_S^{hid} \cdot B \cdot O \cdot H$ ADD: $A_W^{out} \cdot A_S^{hid} \cdot B \cdot O \cdot (H-1)$	Spike time
	Bit-shifting of FW_act	BS	$A_W^{out} \cdot A_S^{hid} \cdot B \cdot O$	Spike time
	Voltage update	MUL/BS B-MUL ADD	$B \cdot O$ $B \cdot O$ $A_W^{out} \cdot A_S^{hid} \cdot B \cdot O$	Spike time
	Spike output	CMP	$B \cdot O$	Spike time
	Pseudo gradient	SUB ABS CMP	$B \cdot O$ $B \cdot O$ $B \cdot O$	Spike time
	Presynaptic trace	BS ADD	$B \cdot H$ $A_S^{hid} \cdot B \cdot H$	Spike time
	Weight trace	B-MUL ADD	$B \cdot O$ $B \cdot O$	Spike time
	Network Output	ADD	$A_S^{out} \cdot B \cdot O$	Spike time

Table B.3: Computational cost breakdown of one training iteration of the RSNN model - Forward Pass.
Abbreviations Used: MM = Matrix Multiplication, MUL = Multiplication, B-MUL = Binary Multiplication, ADD = Addition, SUB = Subtraction, BS = Bitwise Shift, CMP = Comparison.

Model Section	Element	Type of Operation	Number of Operations	Operation Frequency
Error Calculation - Floating-point				
	Softmax	EXP	$A_S^{out} \cdot B \cdot O$	
		ADD	$B \cdot O$	1
		MUL	$B \cdot O$	
	Subtraction	SUB	B	1
Error Calculation - Integer-only				
	Loss calculation	MUL	$A_S^{out} \cdot B \cdot O$	1
		BS	$B \cdot O$	
	Subtraction	SUB	B	1
Update Output Layer				
	Delta	MUL	$B \cdot O$	1
		ADD	$(B-1) \cdot O$	
	Learning rate	MUL/BS	$O \cdot H$	1
	Weight decay	MUL/BS	$O \cdot H$	1
	Weight update	ADD	$2 \cdot O \cdot H$	1
	Bit-shifting to W_{Shadow}	BS	$A_W^{out} \cdot O \cdot H$	1
Update Hidden Layer				
	Error voltage	MM	MUL: $H \cdot O$ ADD: $H \cdot (O-1)$	1
	Delta	MUL	$B \cdot H$	1
		ADD	$(B-1) \cdot H$	
	Learning rate	MUL/BS	$H \cdot I$	1
	Weight decay	MUL/BS	$H \cdot I$	1
	Weight update	ADD	$2 \cdot H \cdot I$	1
	Bit-shifting to W_{Shadow}	BS	$A_W^{hid} \cdot H \cdot I$	1

Table B.4: Computational cost breakdown of one training iteration of the RSNN model - Update Step.
Abbreviations Used: MM = Matrix Multiplication, MUL = Multiplication, B-MUL = Binary Multiplication ADD = Addition, SUB = Subtraction, BS = Bitwise Shift, CMP = Comparison, EXP = Exponentiation.

Activity Coeff.	SNN - MNIST		RSNN - SHD	
	MP 16-8	FP32	MP 16-8	FP32
A_I - Input Activity	0.8693	0.8693	0.5496	0.5496
A_{Irec} - Rec. Input Activity	-	-	0.9999	0.9999
A_W^{hid} - Hidden Weight Activity	0.9999	0.9999	0.9999	0.9999
A_{Wrec}^{hid} - Rec. Weight Activity	-	-	0.9999	0.9999
A_S^{hid} - Hidden Spike Activity	0.2741	0.2634	0.3047	0.3514
A_W^{out} - Output Weight Activity	0.9999	0.9999	0.9999	0.9999
A_S^{out} - Output Spike Activity	0.0541	0.1503	0.0597	0.2413

Table B.5: Activity coefficients in the integer and float models.

Appendix C

Statistical Tests

C.1 Normality Tests

Table C.1 shows the result of the Shapiro-Wilk test for normality with alpha set to 0.05. W parameter is the statistic value, the higher it is (closer to 1) the better the series fits a normal distribution.

Experiment	Series	Acc.	W	p-value
Error Propagation				
	MP - Direct	97.54 ± 0.05 %	0.9459	0.6202
	MP - Final	95.14 ± 0.27 %	0.9744	0.9286
	MP - Feedback	94.54 ± 1.89 %	0.8984	0.2102
	FP - Direct	97.18 ± 0.13 %	0.9169	0.3317
	FP - Final	96.10 ± 0.12 %	0.9334	0.4818
	FP - Feedback	97.33 ± 0.15 %	0.8635	0.0839
Loss Function				
	Standard	97.54 ± 0.05 %	0.9459	0.6202
	Positive add.	95.06 ± 1.21 %	0.9721	0.9096
	Random add.	97.55 ± 0.21 %	0.7845	0.0094
MNIST - SNN				
	MP - 16-8	97.54 ± 0.05 %	0.9459	0.6202
	FP32	97.33 ± 0.15 %	0.8635	0.0839

Table C.1: Normality test on the results of the Error Propagation experiments. SNN - MP16-8, MNIST dataset.

C.2 Series Comparisons

Table C.2 reports the results of a statistical test to prove if the mean of a series of results was significantly higher than another. If both series follow a normal distribution, a paired t-test was performed, and if one of the series did not follow a normal distribution, a Wilcoxon Signed-Rank test was performed instead. Since the null hypothesis is that both series come from the same

distribution or have identical average results, a p-value lower than 0.05 indicates that the first series is significantly higher than the second one.

Experiment	Series 1	Series 1 - Acc.	Series 2	Series 2 - Acc.	p-value
Error Propagation					
	MP - Direct	97.54 ± 0.05 %	MP - Final	95.14 ± 0.27 %	0.0000
	MP - Direct	97.54 ± 0.05 %	MP - Feedback	94.54 ± 1.89 %	0.0006
	MP - Final	95.14 ± 0.27 %	MP - Feedback	94.54 ± 1.89 %	0.2010
	FP - Feedback	97.33 ± 0.15 %	FP - Direct	97.18 ± 0.13 %	0.0427
	FP - Feedback	97.33 ± 0.15 %	FP - Final	96.10 ± 0.12 %	0.000
	FP - Direct	97.18 ± 0.13 %	FP - Final	96.10 ± 0.12 %	0.000
Loss Function					
	Standard	97.54 ± 0.05 %	Positive add.	95.06 ± 1.21 %	0.0000
	Random add.	97.55 ± 0.21 %	Positive add.	95.06 ± 1.21 %	0.0009
	Standard	97.54 ± 0.05 %	Random add.	97.55 ± 0.21 %	0.3710
MNIST - SNN					
	MP - 16-8	97.54 ± 0.05 %	FP32	97.33 ± 0.15 %	0.0012

Table C.2: Statistical test comparing if the mean of series 1 is significantly higher than the mean in series 2.

Appendix D

Experimental details

D.1 Optimal training parameters

General training parameters

Training parameter	MNIST	SHD
Number of epochs	50	50
Train batch size	128	128
Test batch size	256	256
Spike time	20	10
Soft_error_start	5	5

Table D.1: General training parameters by dataset.

Integer Weight Initialization

A critical training parameter was the selected bit-width to store initial weight values before training. A lower bit-width initialization could potentially increase the model's ability to learn by allowing the weights to grow naturally at the beginning of the training. Empirical findings suggested that in the integer-mixed-precision configurations the weights should always be initialized following a distribution whose minimum and maximum values are defined by the minimum and maximum values of the shadow weights precision bits.

Training parameters per model and dataset

Parameter	SNN		CSNN	RSNN
	MNIST	SHD	MNIST	SHD
Hiden Layer Parameters				
Voltage Threshold V_{th}	500	250	250	2000
Gradient Window $Grad_{win}^{(i)}$	1000	500	500	4000
Learning Rate	2^{-12}	2^{-14}	2^{-15}	2^{-12}
Weight Decay Rate ρ	0	2^{-12}	0	2^{-12}
Output Layer Parameters				
Voltage Threshold V_{th}	2000	2000	250	1600
Gradient Window $Grad_{win}^{(i)}$	4000	4000	500	1600
Learning Rate	0.5	1	1	0.5
Weight Decay Rate ρ	0	2^{-12}	0	2^{-12}
Other Parameters				
Voltage Decay (β)	0.5	0.5	0.5	0.5
Loss Precision α	128	128	32	128
Gradient Clip $\Delta_{\max}$	2048	512	2048	256

Table D.2: Training parameters for the different models and datasets for the MP - 16-8 configuration.

Parameter	SNN		CSNN	RSNN
	MNIST	SHD	MNIST	SHD
Hiden Layer Parameters				
Voltage Threshold V_{th}	0.3	0.3	0.5	2
Gradient Window $Grad_{win}^{(i)}$	0.3	0.6	0.5	4
Learning Rate	0.001	0.001	0.001	0.001
Output Layer Parameters				
Voltage Threshold V_{th}	0.3	0.3	1.5	1
Gradient Window $Grad_{win}^{(i)}$	0.6	0.6	1.5	1
Learning Rate	0.001	0.003	0.001	0.003
Other Parameters				
Voltage Decay (β)	1	1	0.5	0.5

Table D.3: Training parameters for the different models and datasets for the FP32 configuration.